VNMR Pulse Sequences

Programming and Hardware Aspects

Pub. No. 01-999014-00, Rev. A0398

VARIAN

VNMR Pulse Sequences
Programming and Hardware Aspects
Pub. No. 01-999014-00, Rev. A0398

By Rolf Kyburz
rolf@nmr.varian.ch
Varian International AG
Zug/Switzerland

Technical editor: James Welch

Overview of Contents

Table of Contents

List of Figures

List of Tables

Disclaimer

The information in this manual is intended to assist users in topics beyond the normal NMR spectrometer system hardware and software support provided by Varian. Some of this information is provided on an as-is basis, and Varian support and service personnel may be unable to answer questions related to information given in this manual.

Foreword

The main purpose of this manual is to complement the contents of the manual *VNMR User Programming* by providing comprehensive information on the `entire` subject of pulse sequence programming and execution as well as spectrometer control, trying to open up "black boxes" or "white spots" that might exist in a user's vision of a UNITY*plus*, UNITY, or VXR-S spectrometer. No attempt was made to describe all pulse sequence functions in detail—many of them will not even be mentioned here, but are covered by the manual *`VNMR User Programming`*.

Not all of the sections are equally important for the pulse sequence programmer. Most certainly it is possible to write correct pulse sequences without having access to the material presented here. It is the author's firm belief, however, that somebody who has more background knowledge on pulse sequence and spectrometer internals (as presented in this manual) will be more efficient and will less likely get stuck with the question of "How do I implement this on my spectrometer?".

The chapters on `seqgen` and `psggen` (chapters 2 and 3) are mainly written for people that want to learn about the programming internals of a pulse sequence, or for users that want to implement changes or new features in the pulse sequence overhead; sections that deal with Acode can be skipped by those who are not interested.

The material was originally written as documentation for the pulse sequence programming part of the Varian courses for users of Varian VXR-S, UNITY, and UNITY*plus* NMR spectrometers using VNMR software. Any input that would help improve this manual is very welcome.

Note that the specifics of the UNITY*INOVA* are *not* discussed—this will be covered in a future version of the manual. However, apart from some differences in the digital hardware, the basic concepts shown in this material are still valid also on the UNITY*INOVA* (see also the note below on software compatibility).

The present version of this manual specifically refers to VNMR version 5.1 and is partly inconsistent with earlier and later VNMR releases (in particular, the Acode structure for the UNITY*INOVA* is different, and the `apdecode` software mentioned in this manual will not work on the VNMR 5.2 software release). Most Acode printouts have been generated under VNMR 4.3 and are slightly different from Acode printouts obtained with `apdecode` under VNMR 5.1.

Rolf Kyburz
Varian International AG, Zug/Switzerland
February 1996

Acknowledgments

The author would like to thank his colleagues for their thorough proofreading, suggestions, and helpful discussions, as well as the Varian service personnel for withstanding the author's endless questions on the subject of this manual. In particular, Andy Myles (service supervisor in Darmstadt) has been extremely helpful in providing technical information.

Conventions in This Manual

The following notational conventions are used throughout all VNMR manuals:

- Typewriter-like characters are used to represent UNIX or VNMR commands, parameters, directories, and file names in the text of the manual; for example:

  The `shutdown` command is in the `/etc` directory.

- Typewriter-like characters are also used for text displayed on the screen, including the text echoed on the screen as you enter commands; for example:

  ```
  Self test completed successfully.
  ```

- Italicized typewriter-like characters are used for text displayed on the screen that is not the same every time; for example,

  Abort at *some_address*

  means the value of "*some_address*" depends upon when the abort command is made—what you might see on the screen is a message like this:

  ```
  Abort at 47F82
  ```

- Special characters are used for keys on the keyboard and menu buttons on the screen; for example,

  Press the Return key or select the Display button.

- Text shown between angled brackets in a syntax entry is optional. For example, if the syntax is `seqgen s2pul<.c>`, entering the "`.c`" suffix is optional, and typing `seqgen s2pul.c` or `seqgen s2pul` is functionally the same.

- Lines of text containing command syntax, examples of statements, source code, and similar material are often too long to fit the width of the page. To show that a line of text had to be broken to fit into the manual, the line is cut at a convenient point (such as at a comma near the right edge of the column), a backslash ("\") is inserted at the cut, and the line is continued as the next line of text. This notation will be familiar to C programmers. Note that the backslash is not part of the line and, except for C source code, should not be typed when entering the line.

- Because pressing the Return key is required at the end of almost every command or line of text you type on the keyboard, use of the Return key will be mentioned only in cases where it is *not* used. This convention avoids repeating the instruction "press the Return key" throughout most of this manual.

Chapter 1. Overview

This manual discusses the underlying functionality of the generation and execution of pulse sequences under VNMR. The idea is to provide the advanced pulse sequence programmer with a comprehensive insight into all aspects of pulse sequence generation. With better understanding, avoiding mistakes and writing more efficient pulse sequences should be possible, while at the same time moving towards more complex experiments.

1.1 Pulse Sequence Execution

Before going into details, let us review the various stages of pulse sequence generation and execution (see Figure 1):

- At the center of pulse sequence execution is the command `go`, which executes a compiled pulse sequence from `seqlib`, based on parameters from the current experiment and system configuration parameters (`/vnmr/conpar`). If external phase tables are used, these are retrieved from `tablib`, and pulse shapes or other waveform generator patterns are retrieved from `shapelib`. `go` stores its output (Acode, the data that result from executing the file in `seqlib`) in the directory `/vnmr/acqqueue`.
- Before `go` can be called, the parameters in the current experiment must be prepared. This is usually done by calling a macro (from `maclib`) with the name of the pulse sequence. This macro creates the necessary new parameters, usually from a library (`parlib`) with pulse sequence specific parameters, and it usually displays a text file with information about the pulse sequence.
- Before `go` can execute the pulse sequence file in `seqlib`, the pulse sequence file in `psglib` must be compiled using the command `seqgen`. This requires various files and libraries in a directory `psg`. `seqgen` itself internally is a complicated, multi-stage process, while for the user it looks like a simple command.
- The libraries of precompiled files required for `seqgen` have to be created first. This is done using a command `psggen`, which again involves numerous source and header files in the directory `psg`. This step is normally not necessary for the user, because a complete set of `psg` libraries is part of the standard distribution tape.
- Once the pulse sequence file in `seqlib` has been successfully executed, the program `Acqproc` takes the Acode from `/vnmr/acqqueue`, transmits it via SCSI bus to the acquisition CPU (via the HAL, the Host-to-Acquisition Link), monitors the acquisition, stores the experimental data in the appropriate FID file within the experiment from which the acquisition has been started, and notifies VNMR at specific points in the acquisition (error condition, block size, and number of transients or entire experiment completed).
- In the acquisition CPU, the Acode is executed (interpreted), which results in information being fed into the pulse programmer or directly into the spectrometer that interacts with the magnet and the probe.

- The pulse programmer board controls almost all of the spectrometer (rf channels, waveform generators, frequency synthesis, receiver chain), including the analog-to-digital converter (ADC), which produces a new FID that is transferred to the sum-to-memory board (STM), which adds the current FID to the previous data in the HAL memory, from where `Acqproc` retrieves the final (or preliminary) FID onto the Sun memory (and disk).

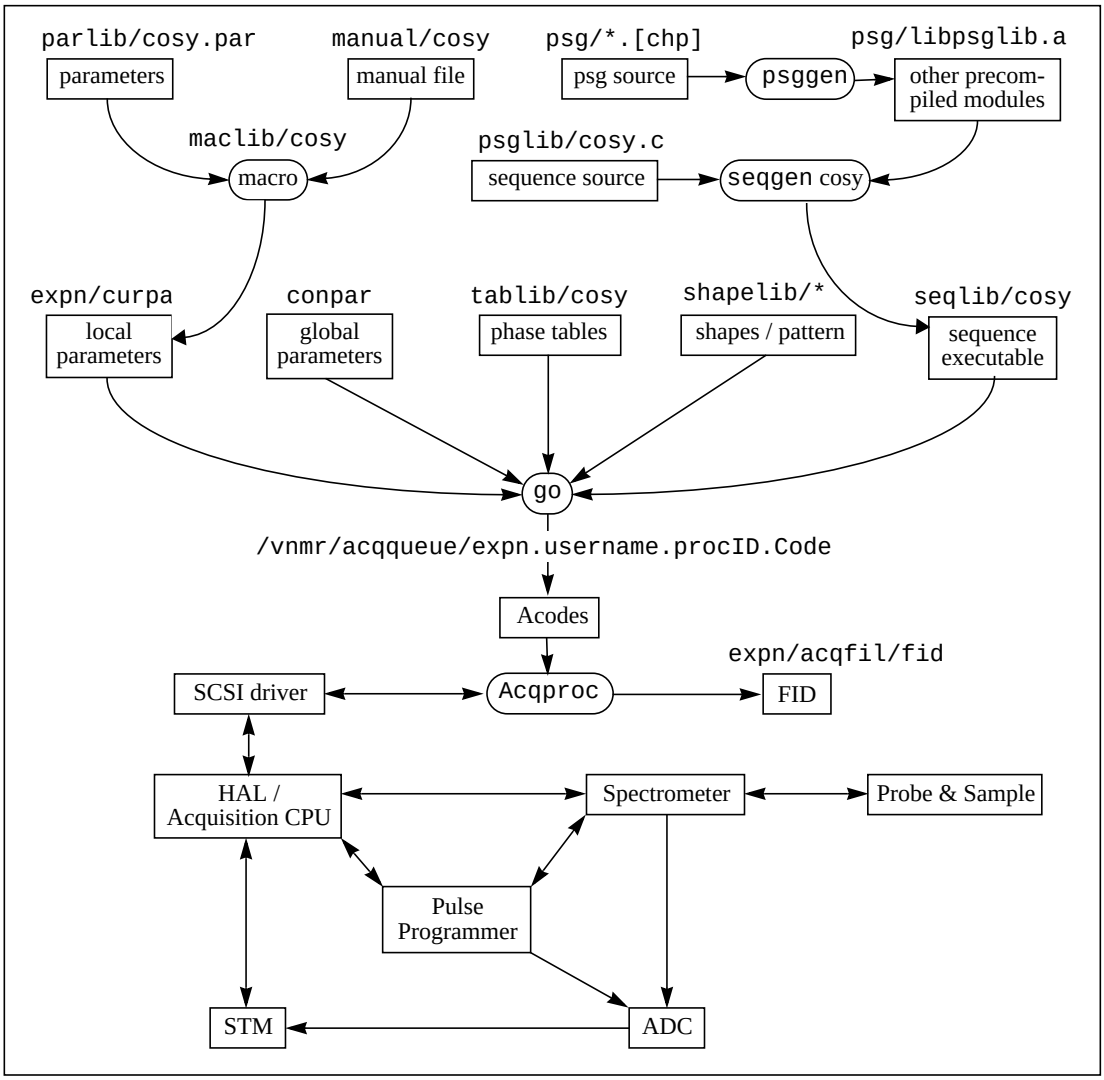


Figure 1. Stages of pulse sequence execution

1.2 What to Expect in This Manual

In the following chapters, every single step in the above process is discussed in detail, starting with pulse sequence generation and the `psggen` step, continuing with `go`, then looking into the things that happen in the acquisition CPU and the spectrometer, while continuously adding knowledge about pulse sequence statements (sometimes also called elements or functions in VNMR documentation).

In the end, we will come back to the “origin” and discuss how the front end for the user (the user interface of a pulse sequence) is constructed.

Chapter 2. Sequence Generation: seqgen

The central step in the creation of a new pulse sequence is the compilation of the pulse sequence source code, which results in an executable file (with the name of the pulse sequence in `seqlib`, the library for compiled pulse sequences).

2.1 Modifying the File for dps

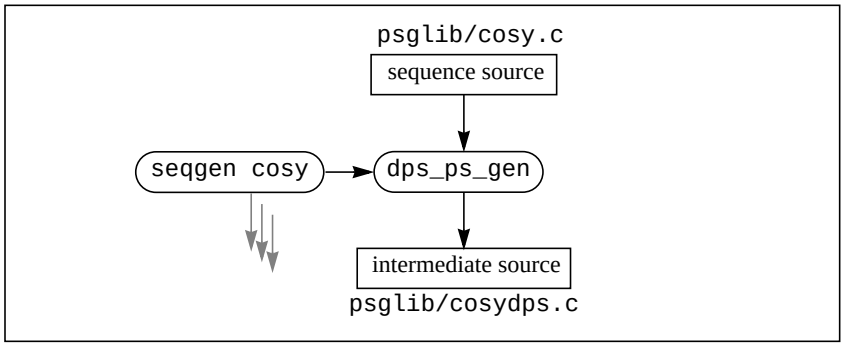
The VNMR command `seqgen` calls a UNIX shell script `/vnmr/bin/seqgen` with the same argument (the name of the pulse sequence with or without `.c` extension). `seqgen` can also be called from a UNIX environment:

```
seqgen sequencename<.c>
```

The UNIX file `/vnmr/bin/seqgen` is a lengthy shell script that first checks for all the files that are necessary for a successful compilation, then it checks for the presence of a program `/vnmr/bin/dps_ps_gen`. If this program is found, it is executed:

```
dps_ps_gen seqgencename.c $HOME/vnmrsys/psglib 2
```

This results in a new file `sequencenamedps.c` (the name of the pulse sequence source file with `dps` inserted between the extension and the body of the name:



The new `dps`-modified file contains both the original text of the pulse sequence in the first part, plus a new, second part for the `dps` command, which graphically displays the pulse sequence (enclosed in `#ifndef LINT ... #endif`, such that `lint` (the syntax checker) does not "look" at this part of the file):

```
#include <standard.h>

pulsesequence()
{
   /* equilibrium period */
   status(A);
   hsdelay(d1);

   /* tau delay */
   status(B);
   pulse(p1, zero);
   hsdelay(d2);
```

```
   * observe period */
   status(C);
   pulse(pw,oph);
}

#ifndef LINT

extern FILE   *dpsdata;

x_pulsesequence()
{
   fprintf(dpsdata, "\n status   A %f ",pw);
   fprintf(dpsdata, " D_d1  %.9f ", d1);

   fprintf(dpsdata, "\n status   B %f ",pw);
   fprintf(dpsdata, " P_p1  %.9f ", p1);
   fprintf(dpsdata, " D_d2  %.9f ", d2);

   fprintf(dpsdata, "\n status   C %f ",pw);
   fprintf(dpsdata, " P_pw  %.9f ", pw);
}

#endif
```

The second part of the `dps`-modified file contains all functions of the first part, prepended with "`x_`" (e.g., `x_pulsesequence()` instead of `pulsesequence()`). Comment is removed, and internal functions (pulse, delay, etc.) are replaced by functions that print some symbolic text into a temporary text file (`curexp/dpsdata`, referenced here as `dpsdata`). The VNMR `dps` command then decodes this text file. The internal structure and functionality of this text file will not be discussed any further here—this is beyond the scope of this manual.

If there was any C errors in the pulse sequence, such errors very likely would be propagated into the second part, leading to an increased number of error messages (despite the fact that the second part is "hidden from `lint`" through `#ifndef LINT` and `#endif`). For this reason, `seqgen` recompiles the pulse sequence *without* the `dps` modifications upon detection of errors with the compilation of the `dps`-modified file.

2.2 Running the make Program

All subsequent steps are performed using `make`, a UNIX program that facilitates the compilation of complex objects and also checks on the modification date of the files involved. `make` avoids an unnecessary compilation if an up-to-date executable already exists (see Chapter 3, "Object Library Generation: psggen," on page 29). `make` then calls `cpp` (the C preprocessor), `lint` (syntax checking), and `cc` (the C compiler and linker). If, at any of these stages, error messages are produced, the messages are stored in a separate file `errmsg`, which is displayed by `seqgen` at the end of the operation.

The `make` program takes its instructions from `/vnmr/acqbin/seqgenmake`, a special "make file." If at the end of the entire process an executable is obtained, the `dps` part of the name is removed by the `seqgen` script and the executable pulse sequence is stored in `seqlib`. Figure 2 is a diagram of the `make` process.

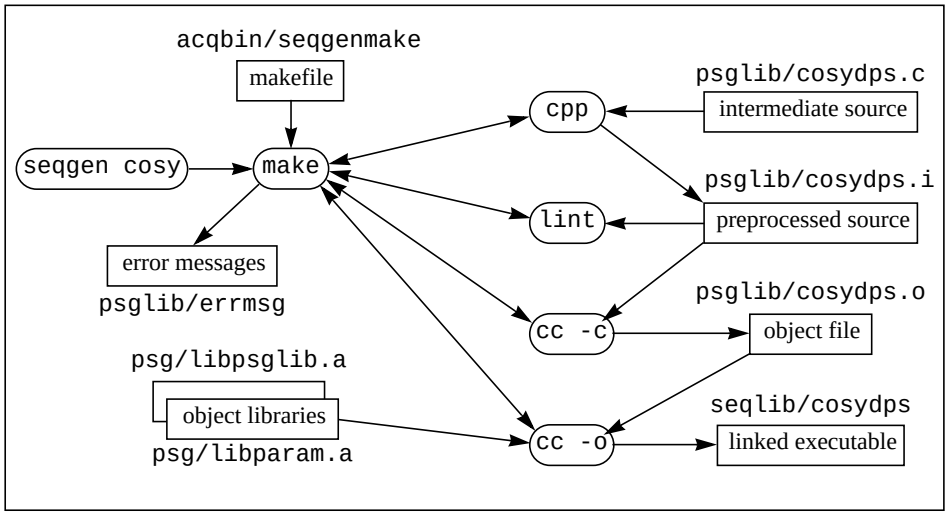


Figure 2. Compiling from the `make` utility

2.3 Calling the C Preprocessor

Compiling a C program always involves first a call to the C preprocessor, which has the task of removing comments and resolving all the preprocessor statements in the C code. The C preprocessor `cpp` (or `cc -P`) is normally invoked implicitly as part of the `cc` command, and its output is directly fed into the next stage of the compilation. For the `lint` syntax check, however, the C preprocessor is called explicitly:

```
cc -P -DLINT -I$HOME/vnmrsys/psg -I/vnmr/psg sequencename.c
```

This generates a file `sequencename.i`, which is then subjected to the `lint` check. The option `-DLINT` defines the preprocessor variable `LINT` (i.e., we are explicitly *including* source sections between `#ifdef LINT` and `#endif`, and explicitly *excluding* parts between `#ifndef LINT` and `#endif`). The `-I` options specify the path(s) to VNMR-specific `include` files.

The major task for the C preprocessor is the resolution of `include` statements, macros, and constants. In a first step, the `include` statements have to be identified and replaced with the contents of the file they indicate. Every C program has at least one `include` statement:

```
#include <stdio.h>
```

This statement incorporates the file `/usr/include/stdio.h`. VNMR pulse sequences usually also have a single `include` statement:

```
#include <standard.h>
```

This statement incorporates the file `/vnmr/psg/standard.h` (or the file `$HOME/vnmrsys/psg/standard.h`, both paths are specified in the call to `cc -P`), and `standard.h` is mostly a collection of further `include` statements.

The program `cpp` recursively resolves all these lines, until no other `include` lines are found. Figure 3 is a diagram of the `cpp` program incorporating the `include` files.

The first `include` file from `/vnmr/psg/standard.h` is `stdio.h` (as for any C program). All other `include` files (`oopc.h`, `acqparms.h`, `rfconst.h`, `aptable.h`, `power.h`, `macros.h`, and `apdelay.h`) are from the `psg` directory,

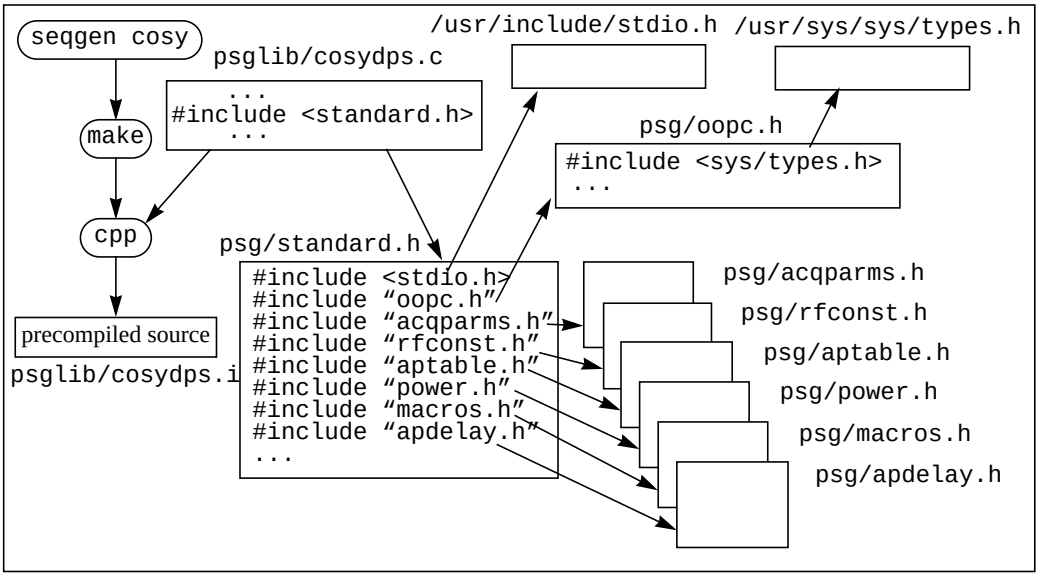


Figure 3. Diagram of the `cpp` program and `include` files

either in `/vnmr` or in `$HOME/vnmrsys`. The file `oopc.h` calls for another `include` file named `/usr/kvm/sys/sys/types.h`.

The purpose of all the `include` files is two-fold:

- To declare external definitions for functions and variables that are defined (or initialized) in another source module (which is precompiled and will simply be linked from an object library).
- To resolve constants and macros. These are pure preprocessor functions and must be resolved before the compiler can be called. These further modify the pulse sequence text for the C compiler (which doesn't understand constants or macros).

Most of the `include` files for pulse sequences deal with external declarations of functions and variables, or with the definition of constants. Macros are defined in the `include` file `psg/macros.h` (macros are just a more sophisticated type of constant). Because at least some of these `include` files contain information that can be very helpful to pulse sequence programmers, they are briefly discussed here:

- `stdio.h` defines the I/O functions for C. Nearly all C programs include `stdio.h`.
- `psg/oopc.h` defines some message structures and all the constants required in connection with general (object-oriented) routines like `G_Pulse` and `G_Delay`, which are described in the manual *VNMR User Programming*.
- `psg/acqparms.h` defines all (externally declared and initialized) variables related to acquisition parameters (`pw`, `p1`, `d1`, `d2`, `d3`, `d4`, etc.), externally defined code addresses (`ct`, `oph`, `zero`, `one`, `two`, `three`, `v1`, `v2`, `v3`, etc.) and objects (mostly structures), and the indices for the various rf channels.
- `psg/rfconst.h` defines constants such as `TRUE` and `FALSE`, fast bit offsets (hexadecimal constants) for the fast bits in VXR/UNIT- style output boards and acquisition control boards, device addresses (`TODEV`, `DODEV`, etc.), status field constants `A` (=0) to `Z` (=25), and a few others.

- `psg/aptable.h` defines constants and structures used in connection with tables, including the (externally defined) table addresses `t1` to `t60`.
- `psg/power.h` defines a few constants used in connection with the `G_Power` general power statement, described in the manual *VNMR User Programming.*
- `psg/macros.h` defines constants used specifically with macros as well as all the macros used in pulse sequences. Most of the statements used in pulse sequences are actually macros that are converted into very few general statements such as `G_Pulse` and `G_Delay`.
- `psg/apdelay.h` defines constants (macro functions in reality) that specify the implicit delays involved with AP bus statements. Whenever possible, these delay constants (e.g., `POWER_DELAY`, `SAPS_DELAY`, `OFFSET_DELAY`) should be used instead of direct numeric values (e.g., `14.95e-6`) in pulse sequences.

The reason why there are many different `include` files is that specific parts of the "overhead" defined in these files are used by various other source modules as well, and the "packaging" is defined by the needs of these other source modules. If there was only one source module (the pulse sequence itself), there would probably be a single `include` ("header") file.

Macros are a special kind of constant that allow for arguments. Let's take the definition of `decpulse` (from `psg/macros.h`) as an example:

```
#define decpulse(DECch,phaseptr)                        \
                 G_Pulse(PULSE_DEVICE,     DECch,       \
                         PULSE_WIDTH,      length,      \
                         PULSE_PHASE,      phaseptr,    \
                         PULSE_PRE_ROFF,   0.0,         \
                         PULSE_POST_ROFF,  0.0,         \
                         0)
```

The two arguments to the `decpulse` statement become arguments 4 and 6 of the `G_Pulse` statement; the other arguments in the general statement are determined by the macro name.

The C preprocessor dramaticallychanges the text of the original pulse sequence file, primarily by replacing `include` lines by the text of the `include` files (and recursively substituting text for `include` lines contained therein). The text of the pulse sequence function itself is different after the substitution of macros and constants (apart from the elimination of comment lines). Look at the simple `pulsesequence` function of a S2PUL (standard two-pulse) pulse sequence:

```
pulsesequence()
{
   /* equilibrium period */
   status(A);
   hsdelay(d1);

   /* tau delay */
   status(B);
   pulse(p1, zero);
   hsdelay(d2);

   /* observe period */
   status(C);
   pulse(pw,oph);
}
```

After the constants `A`, `B`, and `C` have been substituted, and the pulse macros have been replaced by the corresponding `G_Pulse` calls, the function consists of the following lines (the blank lines that result from the elimination of comments have been removed):

```
pulsesequence()
{
   status(0);
   hsdelay(d1);
   status(1);
   G_Pulse(1, p1, 5, zero, 0);
   hsdelay(d2);
   status(2);
   G_Pulse(1, pw, 5, oph, 0);
}
```

Note that the constants within the arguments to `G_Pulse` have been replaced by their numeric values: the result of the C preprocessor is not meant to be read by humans but rather adjusted for the C compiler. What is left are pure C control structures, function names, variables, and (numeric or string) constants. Remember that C knows symbolic constants only through preprocessor definitions.

The C preprocessor includes another useful feature that was briefly mentioned earlier in this chapter: conditional compilation. Based on the definition of a (preprocessor) variable, certain sections of a source file can be specifically excluded or included. This feature is used extensively in the "overhead" to pulse sequences (see Chapter 3, "Object Library Generation: psggen," on page 29). In pulse sequences, this feature is only used to hide things from `lint`, which would otherwise lead to an error message:

```
#ifndef LINT
static char SCCSid[] = "@(#)cosy.c 3.1...";
#endif
```

The second line would normally cause an error message from `lint`, because the array variable `SCCSid` is defined but not used (see below). Such preprocessor flags can either be defined in the file itself (`#define LINT`; note that this definition has a name, but no value) or through an argument (`-Dflagname`) with the preprocessor or the C compiler call, for example:

```
cc -P -DLINT -I$HOME/vnmrsys/psg -I/vnmr/psg sequencename.c
```

2.4 Checking Syntax with lint

One of the drawbacks of the C programming language is the fact that its compiler has little error detection. The compiler only finds some very coarse syntactical errors, such as a mismatch in the parentheses or braces, and eventually also missing semicolons. Anything more subtle—such as a variable type mismatch, the use of uninitialized (or even undefined) variables, a mismatch in the number of arguments to a function, etc.—remains undetected by the compiler itself. With all the syntactical traps and pitfalls (e.g., using `&` instead of `&&`, or `|` instead of `||` makes perfect sense in C), this weakness could make it a very difficult language for occasional pulse sequence programmers. Fortunately, there is `lint`, the C syntax checker.

The program `lint` can be called on any C source program. `lint` usually makes lots of output covering two aspects: definite errors and possible errors (the latter includes warnings about the possibility of exceeding the limits on arrays, if using an array). Error detection is not a problem with `lint`—it's rather that often the output is

overwhelming. For that reason, in checking pulse sequences, `lint` is called with some of its checking features turned off (e.g., options `-n`, `-u`, `-v`, `-z`; see `man lint`).

Like `cc`, `lint` first calls the C preprocessor `cpp` to resolve preprocessor statements (definitions, macros, and conditional sections). In the case of pulse sequences, this step would have the undesirable effect that the original macro syntax would be lost (and with it possibly errors in the arguments to macros) because macros would be resolved to object-oriented function calls, where argument type-checking is nearly impossible (at the `lint` level). For this reason, most of the file `psg/macros.h` is divided into two sections—one for the normal compilation (`#ifndef LINT`) and the other for the `lint` pass (`#else...#endif`). The `lint` version is generated in an explicit call to the C preprocessor:

```
cc -P -DLINT -I/vnmr/psg -I$HOME/vnmrsys/psg sequencename.c
```

The `lint` part of `psg/macros.h` does not resolve the macros into their C function equivalents. It simply changes the macro name from lower to upper case (leaving the arguments intact), while constants are still replaced. The simplified S2PUL pulse sequence above would then look as follows in the corresponding `s2pul.i` file:

```
pulsesequence()
{
   status(0);
   hsdelay(d1);
   status(1);
   PULSE(p1, zero);
   hsdelay(d2);
   status(2);
   PULSE(pw, oph);
}
```

Subsequently, `lint` checks the syntax in this version of the file, and it “knows” the argument number and types for all the uppercase “pseudo functions.”

How does `lint` know about syntax rules and, in particular, the syntax and types for externally defined functions and variables? The secret is a special library that is involved automatically with every `lint` call. For simple, standard C programs, this library is the file `/usr/lib/lint/llib-lc.ln`. This file is used automatically unless a specific rules file is specified with the `lint` call. For VNMR, a dedicated library file `psg/llib-lpsg.ln` has been created with a special `lint` call, using the `-C` option:

```
lint -a -n -z -DLINT -Cpsg lintfile.c
```

The file `lintfile.c` contains all the pulse sequence functions that are checked and all the externally defined variables and addresses. Different from the real functions, the functions in `lintfile.c` have a null body (`{}`). On the other hand, these functions have the same (complete) argument type declarations as their real equivalents.

As an example, let’s take the `lint` definition of the `rgpulse` statement:

```
RGPULSE(pulsewidth, phaseptr, rx1, rx2)
double pulsewidth, rx1, rx2; codeint phasptr; { }
```

For the `lint` check, `lint` is called with all the preprocessor options (even though it actually operates on the preprocessed file `sequencenamedps.i`):

```
lint -a -c -h -u -z -v -n -DLINT
       -I$HOME/vnmrsys/psg \
       -I/vnmr/psg sequencenamedps.i \
       /vnmr/psg/llib-lpsg.ln >> errmsg
```

The output of the `lint` check is fed into the error message file `errmsg` (see Figure 4).

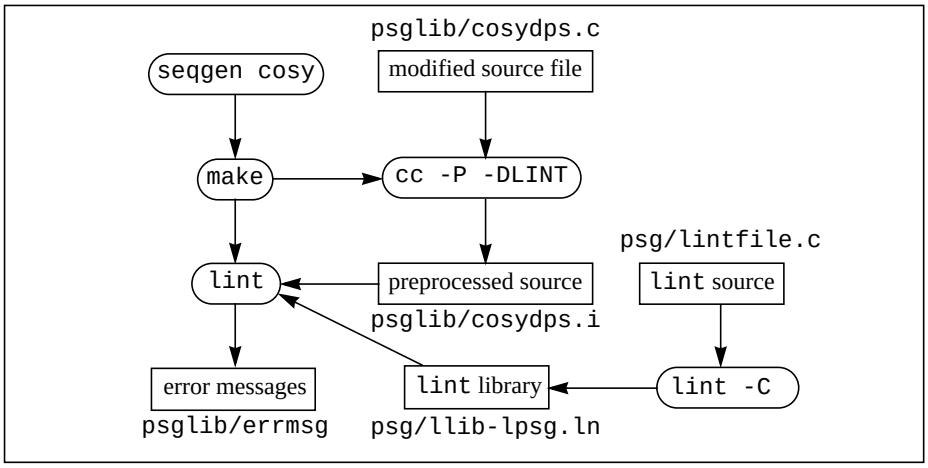


Figure 4. The `lint` check

Of course, VNMR is delivered with `psg/llib-lpsg.ln`, a complete `lint` library file. There seldom should be a need for a user to re-create this library file—perhaps only if somebody creates a new external function and wants to have it checked by `lint`. Note that unlike other files involved in pulse sequence generation, the `lint` library file *must* reside in `/vnmr/psg` unless the `make` file `seqgenmake` is modified for a local file.

Error messages and warnings from `lint` are added to the error messages file `errmsg`. The compilation does not stop upon error messages from the C preprocessor or the `lint` step; such messages are taken as warnings only and in the end lead to a message from `seqgen`: “Pulse sequence did compile, but may not function properly . . .”

2.5 Compiling and Linking

Once the syntax of the pulse sequence is checked, `make` proceeds to the compilation, which is done in two steps: first, the pulse sequence module is compiled and, second, the resulting object module is linked with other precompiled modules to form an executable program (see Figure 5).

The actual pulse sequence compilation is done by `make` with the command `cc -c`, which calls the C preprocessor implicitly. This time, of course, the `-DLINT` option is not specified, such that macros are resolved into the proper C functions. Also, sections that were skipped for the `lint` pass because they would lead to error messages, such as the definition of the `SCCSid` string (see above), are now included.

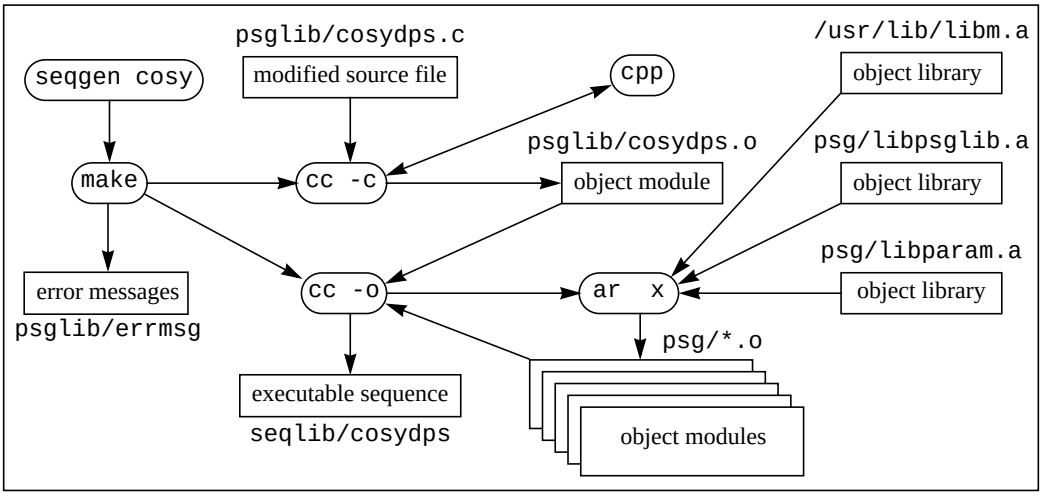


Figure 5. Compiling and linking to form an executable program

The compilation is performed with the following command string:

```
cc -O -I$HOME/vnmrsys/psg -I/vnmr/psg -c sequencenamedps.c
```

where `-O` is the code optimizer option. Note that `-O` is the same as `-O2`, which means level 2 optimization (see `man cc` for more information). The `-I` options are for the preprocessor, which needs to be told where to find the VNMR-specific `include` files. With the `-c` option, the compiler does not produce an executable program, but rather an object module file `sequencenamedps.o`. In case of compilation errors, this file is missing, `make` aborts, and `seqgen` reports all errors that were detected up to that point[1]. The resulting object module `sequencenamedps.o` is not executable, because it is the bare compiled pulse sequence function, with all the references to external variables and functions but without the actual code for them.

In order to arrive at an executable file, `make` calls `cc` again (in reality the call may differ slightly, depending on the software and hardware configuration; also, for various reasons, the syntax in the `make` file is somewhat different):

```
cc -O -Bstatic -s -L/vnmr/psg -o sequencenamedps \
        sequencenamedps.o -lpsglib -lparam -lm
```

The `-O` option again specifies the code optimizer. All other options are not for `cc` itself, but for `ld`, the so-called link editor, which is called by `cc`:

```
ld -s -Bstatic -L/vnmr/psg -o sequencenamedps \
        -lpsglib -lparam -lm sequencenamedps.o
```

The `-s` option causes `ld` to strip the so-called “symbol table and relocation bits” (used by debuggers) to save disk space. `-Bstatic` causes the externally referenced libraries to be linked (attached) statically (see below); the `-L` option permits specifying additional directories where `ld` searches for libraries; and the argument `-o sequencenamedps` specifies the name of the executable target file.

[1] The command `/usr/ucb/cc` (a shell script) works in two steps: the actual C compiler intermediately produces assembly language code (`*.s`), which is then converted to an object module using the `as` command (assembly language compiler). The first step can also be done explicitly using the `-S` option to the `cc` command.

The last three options (`-lpsglib`, `-lparam`, and `-lm`) specify the three libraries from where external modules are loaded: `libpsglib.a` and `libparam.a` (both normally in `/vnmr/psg`), and `/usr/lib/libm.a`. The latter contains the math libraries (extended floating point math functions), the former two contain all the VNMR-specific compiled external modules found in the directory `/vnmr/psg`. The link editor `ld` should normally be able to resolve all references to external functions and variables; otherwise, error messages result.

In the early releases of VNMR, the description above was the way the link loader worked. Meantime, pulse sequence overhead gained complexity, and each statically linked executable pulse sequence became 250 to 400 Kbytes (Sun-3 vs. Sun-4). With about 50 pulse sequences in the standard release, this takes about 10 to 20 Mbytes of disk space. Moreover, all of these sequences would have the *same* modules linked (i.e., a large portion of disk space is lost for 50 copies of almost the same software).

In this situation, it was very useful that Sun introduced the concept of dynamic binding and shared objects (libraries) with SunOS 4. Instead of binding (linking) all external modules at compile time, these objects are put into a special *shared* library: a library that is accessed and used by many object modules at run-time (when they are called)[2].

For pulse sequences, the run-time aspect of shared objects is not relevant, because the cases where two pulse sequences are called at the same time on a single system does not happen normally (the “call” in this case only means the seconds of execution time upon typing `go`). On the other hand, with run-time linking, every pulse sequence only occupies 16 or 24 Kbytes on the disk[3], saving considerable disk space. By default, `seqgen` uses dynamic binding (run-time linking) when compiling pulse sequences.

How is dynamic run-time linking enabled and used in practice? To prepare for using run-time linking, a new file has to be created that combines the contents of the library files `psg/libpsglib.a` and `psg/libparam.a` in a suitable format. For VNMR, such files are included under the names `psg/libpsglib.so.x.y` and `psg/libparam.so.x.y`, where `x` and `y` indicate the major and minor revision number (in VNMR 4.3A the revision number is 5.0). The idea behind the revision numbers is that if several revisions of these files coexist in the same directory, only the last one (the one with the highest revision number) is used. The `so` stands for “shared objects.”

During the compilation, static linking must be prevented. This outcome is achieved by omitting the `-Bstatic` option during the link loading (dynamic linking is the default). This way we get executables without the shared objects, and it turns out that for pulse sequences these files are only 16 or 24 Kbytes, again in multiples of 8 Kbytes (i.e., we save about 95% of the disk space that would be used with static binding). Invisibly for the user, at `go` time, the system now picks the executable from `seqlib`, links it with the `psg/libpsglib.so.x.y` library with the highest revision number, and executes the resulting code. The slow-down in `go` due to the run-time linking is negligible.

---

[2] For certain applications—not pulse sequences, though—this not only saves disk space, but also process/memory space because the shared objects (libraries) are only loaded into memory once and can the be accessed by several applications simultaneously.

[3] The size of executables is always a multiple of 8 Kbytes, because programs are used (loaded into memory) in 8-Kbyte pages anyway.

One feature of run-time linking sometimes becomes apparent to the user: at link time (i.e., when executing the command), the revision dates of the current shared objects is compared with the date of the objects present at compile time (which is included in the executable file), and if there is a revision mismatch, this leads to error messages—it may even cause the program to crash. Normally, this should be no problem—but there are possible dangers when the shared objects are modified by the user, see Chapter 3, “Object Library Generation: psggen,” on page 29.

One small point is left for `seqgen`: if the `dps` additions were included successfully, the pulse sequence name had the characters `dps` added all the way through the syntax check, compilation, and linking. In order for VNMR to find the executable pulse sequence, `seqgen` renames it in the end to the pulse sequence name without `dps`. Finally, all intermediate files are deleted, including the pulse sequence object file (`sequencenamedps.o`). This means that for every `seqgen`, the *entire* process is repeated, irrespective of whether the pulse sequence executable is already up-to-date or not.

Chapter 3. Object Library Generation: psggen

For creating the pulse sequence compilation libraries, over 80 C modules (mostly stored in `/vnmr/psg`) have to be compiled, using over 30 VNMR-specific `include` files. The compilation process is very similar to the compilation of a pulse sequence, except that no explicit C preprocessor call and `lint` syntax checking are involved.

3.1 How Are Object Libraries Generated?

The C preprocessor is called implicitly with the first compiler pass (`cc -c`). In the second C compiler pass (`cc -o`), no executables are created (the program, which is nothing but a single, big C program, is not complete because itt lacks the `pulsesequence` function), but instead, shared object libraries are generated.

Also different from the compilation of a pulse sequence, the intermediate object modules (`*.o`) are not discarded, but rather stored in object libraries (`libpsglib.a` for C modules supplied in `/vnmr/psg`, and `libparam.a` for C modules that are not distributed). This has a major advantage—upon recompilation, `make` only compiles C modules that have been updated because it compares the dates of the individual C modules and their corresponding object modules.

The process of creating the object libraries in the distributed software is so similar to the regeneration of these libraries upon user modifications that only the regeneration will be discussed in detail here.

3.2 Adding Changes to the Object Libraries

VNMR gives the user access to almost all of the pulse sequence-related source code and with that the ability to implement user modifications. In the past, this has also helped Varian to distribute bug fixes for the “pulse sequence overhead” as ASCII text —simple source code modifications for `/vnmr/psg`.

In accordance with the VNMR philosophy, such modifications are not done in `/vnmr/psg` directly, but rather in a local copy of that directory, which is generated with the `setuserpsg` shell script.

This script creates a local directory `~/vnmrsys/psg` containing the two object libraries `libpsglib.a` and `libparam.a`, the library for run-time linking `libpsglib.so.x.y` (`libparam.so.x.y` is not modified by the user and is therefore still stored in `/vnmr/psg`), and the `lint` library file `llib-lpsg.ln`. The principle is that upon re-generation of the libraries, or upon compilation or execution of a pulse sequence, any file missing in the local library is taken from `/vnmr/psg`.

If a user ever needs to use modifications in this area, it most likely involves changes to a single source file. The user either can follow some given recipe (like to fix a bug in the pulse sequence overhead) or can go into `/vnmr/psg` and locate the file that needs modification (typically by using `grep` and an editor to locate the “critical spot”). The source file to be modified is then copied into the local `psg` directory and modified. The shell script command `psggen` is then called, for example:

```
cp /vnmr/psg/aptable.c ~/vnmrsys/psg
cd ~/vnmrsys/psg
vi aptable.c
psggen
```

As shown in Figure 6, the psggen shell script (in /vnmr/bin) invokes a rather complex series of events.

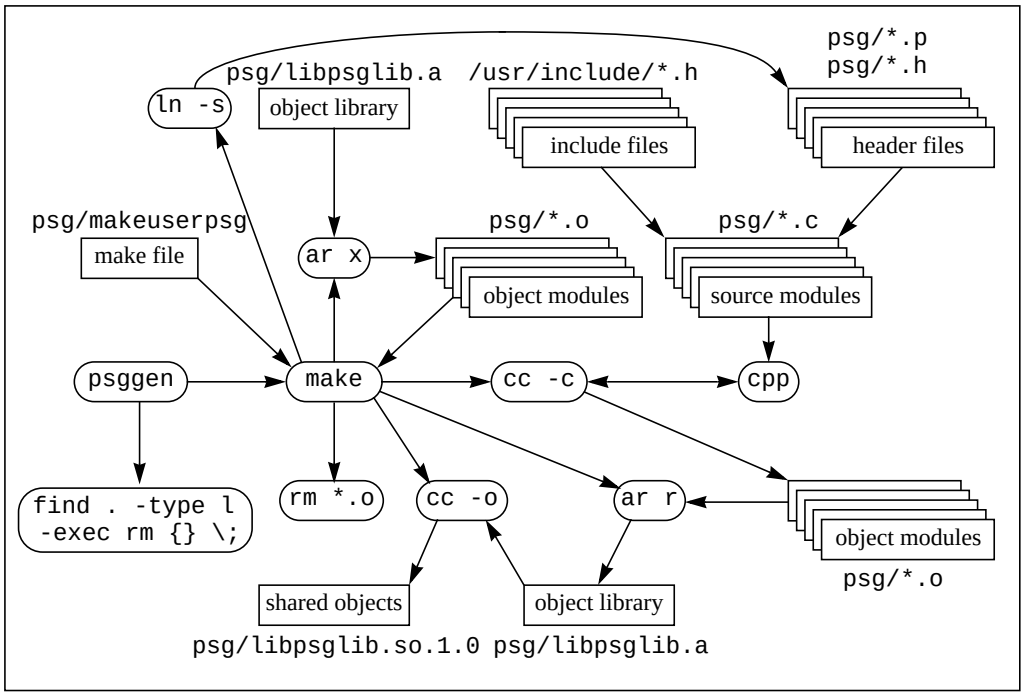


Figure 6. Events associated with the psggen shell script

First, psggen calls make with the file /vnmr/psg/makeuserpsg as the make file (make -fes makeuserpsg lib), and then almost all subsequent actions are done through make:

- For all header and source files (*.c, *.h, *.p) from /vnmr/psg that are *not* present in the local directory, a symbolic link is created. The make program expects to "see" a complete directory.
- All object files (*.o) are unpacked from the libpsglib.a library.
- The creation dates of all C source files and their corresponding object files are compared. Only those files in which the source file has a later modification date than the creation date of its object file are compiled. If you call psggen the first time in a newly generated local psg directory, all sources are compiled, because the original objects in libpsglib.a have been created at the Varian factory, whereas the source files carry a modification date from the time when they were stored in /vnmr/psg (i.e., for the first call of psggen all object files look "outdated"). Such a full compilation takes about five minutes on a SPARCstation 2 or equivalent. On subsequent calls, only the sources that have been modified since the last call to psggen are compiled. On a SPARCstation 2, psggen takes about one minute if only one module was updated.

- The `make` file (`psg/makeuserpsg`) contains all "object dependencies," which means that if a header file (`*.h` or `*.p`) is modified or updated, automatically all sources that include this file become part of the compilation.
- Depending on the above date comparisons, C modules are compiled (`cc -c`); the C preprocessor is called implicitly with that pass for each file.
- After the compilation, the object library (`psg/libpsglib.a`) is updated with the current object files.
- No executables are created in the second compiler (linker) pass, but instead, the shared object library is updated. For a local `psg` directory, this shared object library always has the name `libpsglib.so.1.0`.
- All object files are deleted from the directory. This seems similar to `seqgen`—but here, the objects are still available in archived form. The archive has the advantage that the directory listing can be kept small.
- In a last step, `psggen` removes all symbolic links (to source and header files in `/vnmr/psg`).

Now, pulse sequences can be compiled using the new libraries. Even if the external objects are not statically linked to the pulse sequence executable, pulse sequences should still be recompiled, to avoid internal inconsistencies.

The resulting pulse sequence generation libraries are active only for a single account, because they are in a local `vnmrsys/psg` directory. In principle, `vnmr1` can make changes available to "the world" by copying the modified files into `/vnmr/psg`, although caution must be used with this operation, because errors can cause an inability to compile pulse sequences in general. You may have to reload VNMR to recover.

The main problem is that object archive files should not be copied using the `cp` command because this alters the date of the target file, which can lead to date inconsistencies, and subsequently `make` (`seqgen`!) refuses to work. Note also that `make` does not work if the system date is older than the modification date of source and object files (e.g., by mistake the system date is set backwards by several months or years). Another possible problem lies in the fact that the run-time linker takes the shared library with the highest revision date if several versions coexist; hence, you cannot copy `libpsglib.so.1.0` from the local `psg` directory into `/vnmr/psg`. A procedure that should be safe is as follows:

1. Make a backup of `/vnmr/psg`:

```
cd /vnmr
mkdir psg.bk
cd psg
tar cf - * | (cd ../psg.bk; tar xvfBp -)
```

2. Transfer the local files:

```
cd; cd vnmrsys/psg
tar cf - * | (cd /vnmr/psg; tar xvfBp -)
cd ..
rm -r psg
cd /vnmr/psg
ls *.so.*
mv libpsglib.so.1.0 libpsglib.so.8.0
```

With the last command, the highest revision number is given to the updated copy (which was revision 1.0 in the local directory). The local directory can be deleted,

because all changes are now available from the system files. `cp -p` should also work instead of the `tar` pipe.

3.3 Adding a New Precompiled Object

Few users only need or want to make changes to the pulse sequence overhead; among those, even fewer come into a situation where they want to add a new file to the list of precompiled objects. This chapter was not written with the primary intention to teach how to do this, but rather to give some additional insight into the internal functionality of the `psggen` and `seqgen` commands, and into pulse sequence-related software in general.

A situation where somebody *might* want to add a new file to the list of precompiled modules is the case of a frequently used, very long, custom-built `include` file. Such `include` files can lengthen the pulse sequence compilation time considerably; however, with a precompiled module, the same functions could be made available without slow-down in compilation speed.

First, we create a local `psg` directory and store the new module there. As an example, take the current `include` file `/vnmr/psg/shape_pulse.c`:

```
setuserpsg
cp /vnmr/psg/shape_pulse.c ~/vnmrsys/psg
cd ~/vnmrsys/psg
vi shape_pulse.c
```

We now need to change the C module in order for it to become a standalone module. Pulse sequence `include` files draw all their references to external definitions and the macro references from the standard `include` file `standard.h.` As a result, such files typically have no, or only specific, `include` lines. As a start we can insert all the `include` lines from `/vnmr/psg/standard.h`:

```
#include <stdio.h>
#include "oopc.h"
#include "acqparms.h"
#include "rfconst.h"
#include "aptable.h"
#include "power.h"
#include "macros.h"
#include "apdelay.h"
```

The `include` notation uses the double quotes (e.g., `"oopc.h"`) to mean that local files (e.g., in `psglib`) are included. The angled bracket notation (e.g., `<stdio.h>`) causes `cpp` to only search the standard include libraries (e.g., `~/vnmrsys/psg`, `/vnmr/psg`, `/usr/include`). If files are specified with absolute paths, the `include` notation doesn't matter.

Later, we may want to determine whether we need all these `include` lines. Most likely at least some of them can be deleted, but for the time being, too many `include` lines can't hurt. If `include` lines already exist in the new file, we insert them between the standard C `include` lines (such as `#include <math.h>`) and any VNMR `include` lines, avoiding duplication.

Our example file unnecessarily already contains a reference to `stdio.h`; therefore, we will not add that again. On the other hand, we may have to also include the additional definitions and references from `standard.h`:

```
#define ALL 0
extern int rcvroff_flag,
           ap_ovrride;
extern double getval();
extern void   setprgmode(),
              prg_dec_off();
```

For `psggen` to recognize the new source file, we need to modify the `make` file `makeuserpsg`. If `psggen` has not been called beforehand, we get the `make` file from the system `psg` directory:

```
cp /vnmr/psg/makeuserpsg .
vi makeuserpsg
```

We need to change the `make` file in two places: starting at around line 50 is an alphabetical list of source files. Here we insert a line for our new module at the appropriate place, with a backslash at the end of the line:

```
        shape_pulse.c \
```

The other modification is in the section “Object dependencies,” starting at about line 250, where individual compilation instructions and source code dependencies are listed alphabetically. Our additions are at approximately line 600:

```
shape_pulse.o : $(@:.o=.c)       \
                oopc.h           \
                rfconst.h        \
                aptable.h        \
                power.h          \
                macros.h         \
                apdelay.h
        (umask 2; $(COMPILE.c) $(@:.o=.c) )
```

Most of this is analogous to other entries; therefore, we don’t really have to understand all the details of the `make` file language. A few basics, however, cannot be avoided:

- Constructs like `$(COMPILE.c)` refer to macros defined elsewhere in the `make` file or alternatively in the arguments to the `make` command.
- The whole thing represents the rules and dependencies for the object file `shape_pulse.o`; all continuation lines *must* start with *tabs, not spaces*.

The lines with the header file names describe the object dependencies (i.e., the header file changes imply a recompilation of this particular module). As long as we are in the development stage for that particular module (and maybe don’t know yet for sure what header files will finally be used), these lines aren’t really required; only in the end we should make sure that all used include files are also listed in the object dependency list, to ensure proper updating in the case of header file changes. As a start, we could use two lines only:

```
shape_pulse.o : $(@:.o=.c)
        (umask 2; $(COMPILE.c) $(@:.o=.c) )
```

Our particular example includes `math.h` and makes calls to functions in the math library, like `log10()`. It is not necessary to specify `$(LIBS)` in the last line (`LIBS` is defined as `-lm`, which would link `/usr/lib/libm.a`), because this flag is specified with the compilation of each pulse sequence.

Before we can call `psggen`, we need to make sure that a copy of the new source module exists in `/vnmr/psg`; otherwise, `make` results in error messages and refuses to work, then we can finally call `psggen`:

```
cp shape_pulse.c /vnmr/psg
psggen
```

For the first time, we receive an error message, because `make` can't find a copy of `shape_pulse.o` in `libpsglib.a`. This message is harmless—the file is compiled anyway— and after `psggen`, `libpsglib.a` contains this object module as well.

You can check this with the entry:

```
ar t libpsglib.a | more
```

You can double-check this by recompiling the new module. To do this, first simulate a change in that module by altering its modification date:

```
touch shape_pulse.c
psggen
```

You are now ready to debug the new source module (in the case of a new program) or to try and reduce the number of `include` files, in order to minimize the object dependencies. In the end, make sure the `make` file (`makeuserpsg`) contains object dependencies for all `include` files that are used. For complete debugging, we have to compile pulse sequences that use the new function. If previously the new source module was an `include` file, we need to delete that `include` line from the pulse sequence, because the file is not required any longer; it is now even incompatible as `include` file, because of its own `include` lines (which would lead to multiple error messages because of duplicate definitions).

The fact that the function now no longer is compiled together with the pulse sequence function leaves one open question: How does the pulse sequence (better: the compiler and linker) know that there is now a new function in some other object module? The answer is that external functions are found and incorporated properly as long *as they are of the default type (*`int`*) and do not return a value.* The vast majority of the functions used in pulse sequences are of the default type and do not return values—there is no need to declare them in the pulse sequence module. The same is true for the function that we just converted into a precompiled module.

In the standard `include` file, `standard.h`, we find a definition for the function `getval` that returns a double:

```
extern double getval();
```

This function is of the non-standard type (and returns a value), and hence it must be declared as an external function. Any module in `psg` that uses the function `getval` must also include the above line. The file `standard.h` also defines two more functions that are defined as type `void`:

```
extern void setprgmode(), prg_dec_off();
```

There is a potential disadvantage in making a file precompiled: `lint` no longer "sees" it, and therefore can no longer check the syntax in calls to functions that are now external. To compensate for this, you can add to the `lint` syntax file those functions in the new module that can be called in a pulse sequence. In our particular case, we would add the function definition (without contents, of course) to the file `lintfile.c`.

The lines to be added would be as follows in our case:

```
shape_pulse(shape,pws,phs,pwrtbl,phstbl,spwr,npulses,rx1,rx2)
char    shape[MAXSTR];
codeint phs, pwrtbl, phstbl;
int     npulses;
double  pws, spwr, rx1, rx2;
{}
```

This “dummy” function should be added to the section with plain functions (starting at about line 350) of the file `psg/lintfile.c`; then the `lint` syntax file needs to be generated and copied into `/vnmr/psg`, see Section 2.4, “Checking Syntax with lint,” on page 22:

```
cp /vnmr/psg/lintfile.c .
vi lintfile.c
lint -a -n -z -DLINT -Cpsg lintfile.c
cp llib-lpsg.ln /vnmr/psg
```

The last command allows `seqgen` to use the new `lint` library file; alternatively, we could modify the `seqgen` `make` file (`seqgenmake`, usually stored in `/vnmr/acqbin`) and/or the `seqgen` shell script, to work with the local `lint` library. For that, we should not change the `make` file in `/vnmr/acqbin`, but rather create a local copy in `~/vnmrsys/psg` (i.e., only modify the `make` file for the account that has a local `lint` library). This alternative is not described here.

Chapter 4. Time Events

In this chapter, we deduce more complex pulse sequence statements from simple delays, just to see what is their function. The syntax shown in this chapter is *not* the one used in the pulse sequence overhead; the idea here is not to explain how pulse sequence functions are defined exactly, but merely to understand their functionality in order to correlate the arguments of such a statement to what is happening in the spectrometer. At the same time, this is an opportunity to become familiar with the basic C pulse sequence syntax. Later we shall get a more detailed picture of what is actually happening in the acquisition CPU and the pulse programmer (see Chapter 8, “Acquisition CPU and Acode,” on page 69, and subsequent chapters).

4.1 How Do Delays Work?

A delay is nothing but an exactly specified waiting time for the pulse programmer. In other words, a delay is a time during which the pulse programmer does nothing but hold the current status (we’ll see later exactly what that means). Although even with a simple delay a fair amount of “internal software” is involved, we can think of a delay as a principal pulse sequence element. In fact, a delay or time event is one of the few very basic actions a pulse programmer can do.

Simple Delays

Even a function as simple as a delay has some logical decisions built in; namely, the property that if a length of zero is specified or if the specified duration is too short to be executed by the pulse programmer (see Chapter 9, “Pulse Programmers,” on page 85), the entire pulse sequence statement is skipped. The idea is to avoid unnecessary dead times in a pulse sequence, as well as statements, that could lead to problems at execution time. A possible coding for the statement `delay` could be as follows:

```
delay(length)
double length;
{
   if (length >= MINDELAY) time_event(length);
   if ((length < 0.0) && (ix == 1))
      printf("A negative delay has been specified.\n");
}
```

where `MINDELAY` is a constant that defines the duration of the smallest executable delay (0.2 microseconds); and `ix` is the index of the current FID (starting at 1, up to `arraydim`). Most warnings are issued for the first FID only, in order to avoid flooding the VNMR text window with error messages in arrayed and multidimensional experiments.

The `delay` statement actually is a macro in `/vnmr/psg` that is resolved to the `G_Delay` statement, and the function `time_event` doesn’t exist as such (instead, a statement `delayer` with a slightly different definition is used internally).

Delays With Homospoil Pulse

If we want to write or change the homospoil delay statement `hsdelay`, we first need a definition of what that statement is supposed to do. It is always better to first have an accurate definition before starting to code a program. Sometimes it is necessary to start with a simple definition that can be expanded during the implementation:

- The statement `hsdelay` should have a single argument, specifying the total duration of the homospoil delay: `hsdelay(length)`.
- If the specified length is zero, the entire pulse sequence statement is skipped.
- If the homospoil flag `hs` is `'n'` in the current status field, a simple delay should be performed. If `hs` does not cover the current status field, the last specified field of `hs` is taken instead (propagation of the last specified status field).
- Otherwise, execute a delay of length `hst` with the homospoil (Z1 gradient) turned on, followed by a normal delay, such that the total duration is the specified length.
- If the specified length is less than `hst`, issue a warning message and perform a simple delay instead.

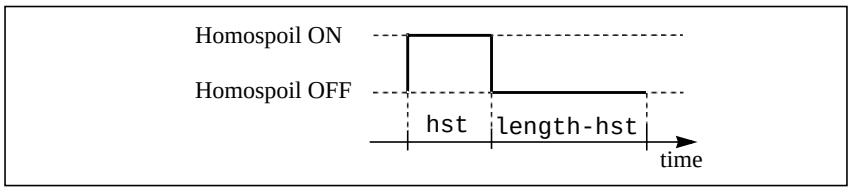
A possible implementation of the `hsdelay` statement could be as follows:

```
hsdelay(length)
double length;
{
  int hspos = statusindex;
  int hslen = strlen(hs);
  if (hspos >= hslen) hspos = hslen - 1;
  if (length >= MINDELAY)
  {
    if (hs[hspos] == ‘y’)
    {
      if (length >= hst)
      {
        HSgate(homospoil_bit,TRUE);
        delay(hst);
        HSgate(homospoil_bit,FALSE);
        delay(length - hst);
      }
      else
      {
        if (ix == 1)
          printf("delay shorter than hst - no homospoil \
          performed.");
        delay(length);
      }
    }
    else
      delay(length);
  }
  if ((length < 0.0) && (ix == 1))
    printf("A negative delay has been specified.\n");
}
```

where `statusindex` is a defined variable set by the `status` statement (to the value of its argument); `hs` and `hst` are standard parameters and do not have to be declared (the same as all parameters used in the `s2pul` pulse sequence). `TRUE` (1) and `FALSE` (0) are constants defined in `/vnmr/psg` (they are *not* part of the standard C definition).

The handling of the `hs` flag conforms to the general internal flag handling in VNMR, in that the last flag field is propagated throughout the rest of the string (e.g., `hs='ny'` is the same thing as `hs='nyyyyyyy...'`; flags can be up to 255 characters long). `statusindex` is a variable used and defined in `/vnmr/psg`, and `HSgate` is an undocumented internal statement[1] in `/vnmr/psg`. In principle, both can be used in pulse sequences.

The relatively complicated `hsdelay` statement above (for the "normal" case) leads to a fairly simple timing diagram:



In words, the homospoil (Z1 gradient) is turned on, a delay `hst` is executed, then the homospoil is turned off, and a second delay `length-hst` is executed. Although the function `hsdelay` has only one argument, it actually has two software-controlled variables: the specified `length`, and the `hst` duration of the homospoil pulse, which is a standard VNMR acquisition parameter. The `strength` (really a third parameter to `hsdelay`) of the homospoil pulse is adjustable in hardware only.

The `hsdelay` statement is a type of "automatic" or "high-level" statement in that users do not have to know how to turn on or off the homospoil gradient, or even what the standard parameter for the length of the homospoil pulse is, in order to use this statement in a pulse sequence. On the other hand, users often want *more control* in a pulse sequence, or perhaps for a specific reason they want to use a different parameter for the duration of the homospoil pulse. This is no problem, because the "low-level" statements can be taken from the above statement and used in a pulse sequence:

```
HSgate(homospoil_bit,TRUE);
delay(hst);
HSgate(homospoil_bit,FALSE);
delay(length - hst);
```

Other Delays

There are a few additional statements for delays: `idelay` for interactive parameter (length) adjustment during acquisition, `incdelay` and `vdelay` are delays where the actual length is calculated by the acquisition computer from a time base and a real-time counter. These other delays are no different than a simple delay in what they do to `spins`; they will not be discussed any further here.

[1] A price may be required to pay for the additional flexibility obtained in using undocumented lower-level statements like `HSgate`: The definition for such statements could change with a future VNMR release or the statement as such may cease to exist altogether, and such changes may not be documented in the standard VNMR manuals. It is up to the user to verify in `/vnmr/psg` that this statement still exists in its current form.

4.2 How Do Pulses Work?

Pulses on the Observe Channel

Several statements coexist for performing pulses on the observe channel; all of them are actually macros and are ultimately translated into calls to a generic function `G_Pulse` (see the manual *VNMR User Programming*). The most general of these statements is `genpulse` (not supported by `dps` at the time of this writing):

```
genpulse(length,phase,rx1,rx2,device);
```

where `length` is the duration of the pulse (see below) in seconds, `phase` is the pulse phase and is a reference to either a real-time variable (see below) or a phase table, `rx1` and `rx2` are "receiver gating times" (again see below), and finally `device` is the rf channel on which the pulse is to be performed (`OBSch` in the case of a pulse on the observe transmitter channel).[2] Note that within a pulse sequence all time events are in seconds, even if the parameter (of type "pulse") is in microseconds.

The statement `genpulse` is the most versatile, yet the most complex, of the statements for a pulse on a single channel at the level of the pulse sequence. It is not recommended except where the rf channel for a pulse should be kept under parameter control.

The statements normally used for a pulse on the observe transmitter channel are `rgpulse`, `pulse`, and `obspulse`. The function `rgpulse` has the following syntax:

```
rgpulse(length,phase,rx1,rx2);
```

This statement is equivalent to `genpulse(length,phase,rx1,rx2,OBSch)`, i.e., the device name (number) is implicit. The next simpler statement is `pulse`:

```
pulse(length,phase);
```

The `pulse` statement is equivalent to `rgpulse(length,phase,rof1,rof2)` or `genpulse(length,phase,rof1,rof2,OBSch)`, i.e., the device name (number) and the two receiver gating times are implicit; for the receiver gating times, parameters `rof1` and `rof2` are used. The simplest statement is `obspulse`:

```
obspulse();
```

This is equivalent to `pulse(pw,oph)`, `rgpulse(pw,oph,rof1,rof2)`, or `genpulse(pw,oph,rof1,rof2,OBSch)`, i.e., in addition to the implicit parameters in the `pulse` statement, the pulse length `pw` and the pulse phase `oph` is implied.

All these statements are more or less "automatic" variants of calling a "pulse" statement in a pulse sequence. What actually is happening in the rf during the execution of such a pulse? Why do we need the "receiver gating times," what are their implicit

---

[2] Traditionally (and exclusively in VNMR releases up to 4.3), devices and channels were treated equivalently. They were named `TODEV` (observe transmitter channel), `DODEV` (decoupler), `DO2DEV` (second decoupler), and `DO3DEV` (third decoupler). These addresses can basically still be used in the current version of VNMR to specify any one of the rf channels. However, these device names hardcode the channels so the channels do not reflect the rf channel assignments made with the `rfchannel` parameter. If the rf channel assignment is to be made dynamically (as per `rfchannel`), the new device names `OBSch` (observe channel), `DECch` (decoupler channel), `DEC2ch`, and `DEC3ch` must be used. For compatibility, it is recommended to translate all occurrences of `TODEV`-type constants to `OBSch` and its equivalents, even if the rf channels are to be assigned statically. Of course, it is even better to use statements that don't require a channel argument.

and desirable values? To answer these questions, let us first have a look at how the `rgpulse` statement could be coded in C:

```
rgpulse(length,phase,rx1,rx2)
double length,rx1,rx2;
codeint phase;
{
   int rcvrflag = rcvroff_flag;
   if (length >= MINDELAY)
   {
      rcvroff();
      txphase(phase);
      delay(rx1);
      xmtron();
      delay(length);
      xmtroff();
      delay(rx2);
      if (!rcvrflag)
         rcvron();
   }
}
```

Like with the `delay` or `hsdelay` statements, unnecessary dead times in the pulse sequence should be avoided: if the specified length is zero, the *entire* pulse is skipped. This actually may have unintended implications, because in the past people have misadjusted the receiver gating times for functional delays (e.g., refocusing delays) in a pulse sequence: the fact that upon setting the pulse length to zero (for testing or calibration purposes), not just the pulse but also the two receiver gating delays disappear, has sometimes led to unexpected results.

The last `if` statement in the above function definition causes the receiver *not* to be turned back on if the receiver has been turned off globally (i.e., with a call to `rcvroff()` prior to calling `rgpulse)`. The `rcvroff_flag` is used for UNITY-style and older rf only; for UNITY*plus* systems, a more complex construct is used within `/vnmr/psg`. (The value of `rcvroff_flag` needs to be stored in an intermediate variable, because the `rcvroff()` function alters that flag.)

What is the reason for using three time events in the above construction? Why can't we simply turn the amplifier on and off again? In order to discuss this, we need to take a detailed look at the timing diagram, which now (different from `hsdelay`) involves *several* devices: the transmitter, the receiver, and the transmitter phase setting.

As shown in Figure 7, it turns out that implicitly also the amplifier is involved in a particular way.

For the transmitter and the receiver (and the amplifier), simple gating (on/off) lines are involved. The 90-degree phase setting is slightly more complex, as it involves four possible values instead of a simple on/off information.

It turns out that we can deduce the four phase values from *two* on/off bits: an on/off bit (each bit represents the values 0 or 1, i.e., off or on) for the 180-degree phase shift and an on/off bit for the 90-degree phase shift.

In all, a pulse involves *five* on/off bits (fast bits, as we call them later): the transmitter, the receiver, the amplifier, the 180-degree phase shift, and the 90-degree phase shift.

In the table below, assume that we switch the transmitter phase from 90 to 180 degrees and that all other devices were in their default setting for a delay:

180	*90*	*Phase shift*
off	off	0 degrees
off	on	90 degrees
on	off	180 degrees
on	on	270 degrees

Obviously, a pulse is much more complex than a simple delay! Why all this? One of the points is that the receiver must be shut down while we are pulsing on the observe channel because both the pulse and the NMR signal are running partially through the same wire. Also, we want to prevent the NMR signal from going back into the amplifier, because this would cause a significant signal loss.

The diagram in Figure 8 shows the switching mode for receiving a signal (“receiver switched on”). In this mode, the T-switch (transmit switch) between the amplifier and the probe is open and the R-switch (read switch) is also open. This way, the NMR signal can only proceed into the preamplifier.

With the receiver switched off, both parts of the T/R-switch are closed, and the amplifier is now connected to the probe. The R-switch puts a specific point between the probe and the preamplifier to 0 volts (ground). At that point, the rf pulse is

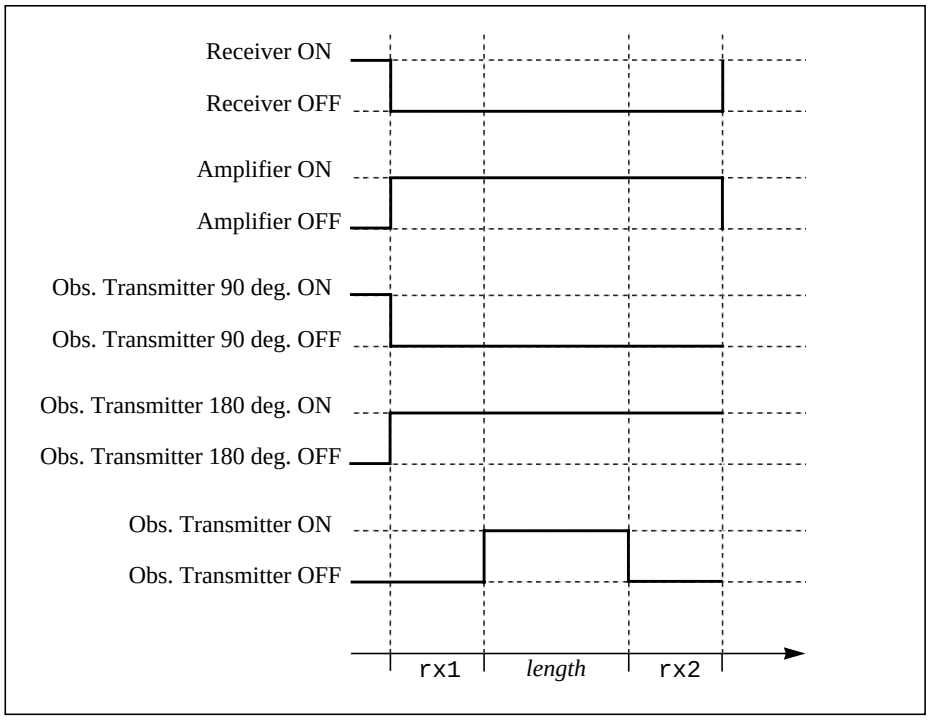


Figure 7. Timing diagram with transmitter, receiver, and transmitter phase

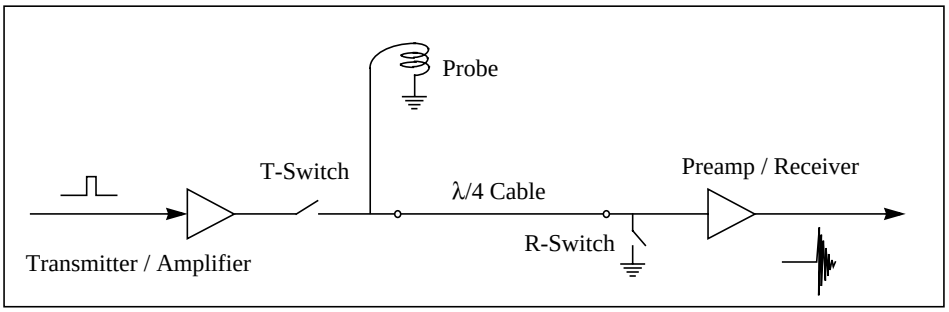


Figure 8. Switching mode for receiving an NMR signal

reflected, building up a standing wave back in the direction of the amplifier (if there was no probe connected we would in fact reflect the entire power back into the amplifier!).

If now the cable between the zero voltage point and the probe connector has the right length (1/4 or 3/4 of the wavelength), the standing wave has maximum amplitude at the probe connector and all the power is directed into the probe, as shown in Figure 9.

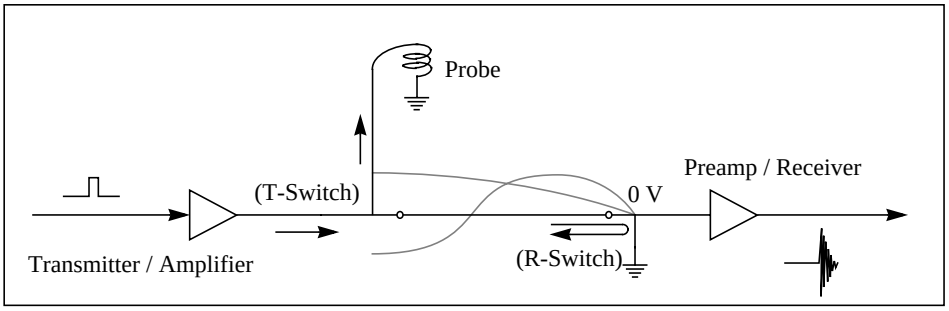
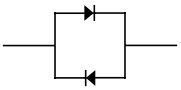


Figure 9. Maximum power directed into the probe

This works very efficiently and avoids a switch *in* the line between the probe and the preamplifier (which would cause a significant signal loss). The only disadvantage is that the length of the cable needs to be adjusted with the frequency. If the length of the quarter-wavelength cable is not adjusted, a part of the amplitude is reflected back into the amplifier, and we would have less power available in the probe; this would affect the length of the 90-degree pulse.

In earlier rf schemes, both the T-switch and the R-switch were not active switches but rather a pair of passive, crossed diodes:



Such an arrangement would act as a closed switch for voltages above 1 volt (such as an rf pulse), but for voltages below around 0.5 volts, the diodes act as an open switch, so that low voltages (like an NMR signal) cannot pass. In reality, there is still some leakage across the diodes, but as far as the NMR signal is concerned, this is negligible.

With the advent of shaped pulses, this kind of rf switch became impossible because the low-level (voltage) part of a pulse shape would be heavily distorted. To avoid that, the passive diodes have been replaced by actively switched PIN diodes, a special type of

diode that are switched by applying a high dc voltage across them. The switching of the T/R-switch occurs anti-parallel to the receiver switching.

Even when these diode switches are opened (i.e., when the receiver is on), there is still some signal leakage across them. The amount is sufficient for the amplifier noise to enter the receiver chain and ruin the signal-to-noise ratio. Linear amplifiers (which are prerequisite for performing shaped pulses) produce relatively high levels of rf noise that can only be suppressed by turning off the last stage of the amplifier (“blanking the amplifier”). For that reason, the observe channel amplifier is automatically switched off (“blanked”) whenever the receiver is on (anti-parallel switching).

All this explains the basic switching diagram shown in Figure 8, but does not explain why there should be *delays* before and after the actual pulse. In particular, we can’t yet deduce any recommendations as to what length of delay is required before and after the pulse. Also, we would like to know whether there are any hidden additional delays not shown on this diagram.

The second question depends on the construction of the pulse programmer and cannot be answered at this point in time (in reality—as we shall see—there are no additional hidden delays involved in the `rgpulse` function).

To explain the first point we should check how the diagram in Figure 7 looks like in “real life”; This is shown in Figure 10.

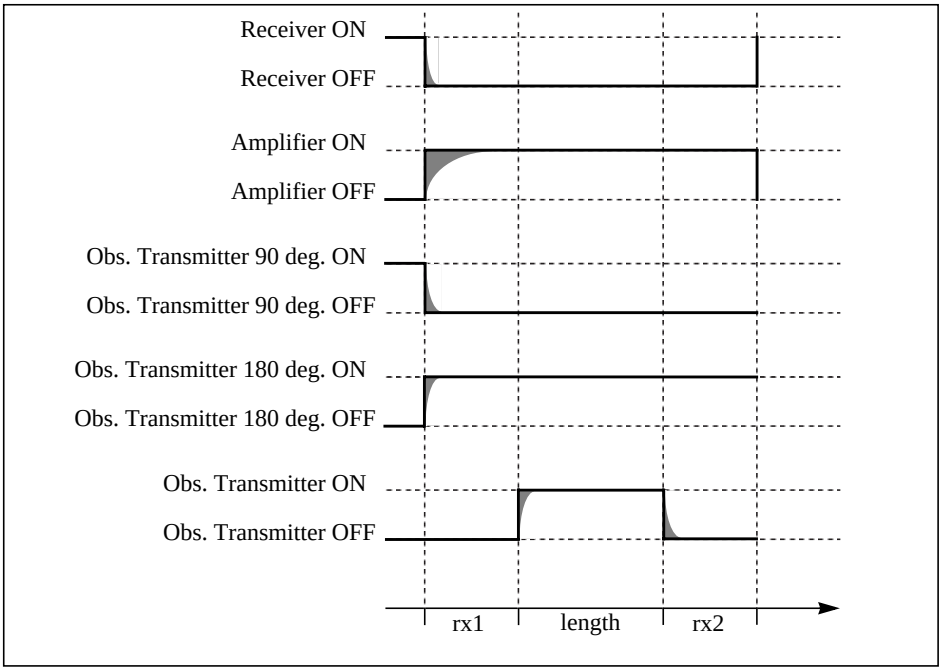


Figure 10. Timing diagram in “real life”

The real reason for introducing additional delays around an rf pulse lies in the fact that we are dealing with real hardware. No matter how quickly a pulse programmer switches lines, it always takes a finite time for the various states to establish themselves. Such delays are already introduced by the pulse programmer (which has a finite switching time), but since this delays all states by the same amount, we don’t have to consider this (as well as propagation delays through electrical cables) any further.

In order to judge the time constants or the time to completely change status on the devices involved in an `rgpulse` statement, we have to consider the individual hardware involved. First, consider gate switching around the beginning of the pulse.

- The **receiver gate** (without the linked amplifier blanking) operates a number of TTL (transistor-transistor logic) gates and PIN-diodes (such as the T/R switch) with very rapid switching times (typically in the order of nanoseconds). For this reason, the receiver can be considered switching instantaneously relative to the time scale involved in pulse sequences—no particular delay is required to allow the receive and related hardware to switch on or off.
- The opposite is true for the **amplifier blanking**: when the blanking is removed, it takes a relatively long time for linear amplifiers to reach full output amplitude and phase stability. The problem is not so much the turn-on time, but rather the time it takes for the amplitude to stabilize. The time for blanking the output after a pulse is not so critical. UNITY and UNITY*plus* systems use linear (class A/B) amplifiers that are trimmed for short unblanking time: they reach full amplitude stability after 2 to 4 microseconds. Class A linear amplifiers in early high-field VXR spectrometers take up to 40 microseconds to stabilize after unblanking.

  Class C amplifiers used in earlier equipment don't need to be blanked: they produce very little noise and are left on all the time. They are also much more efficient than linear amplifiers, but unfortunately they are unusable for pulse shaping. They are also not broadband by nature—an entire array of amplifiers is required to cover the full frequency range.

  The amplifier appears to be the prime reason for having a delay prior to starting a pulse. If the amplifier was blanked (switched off) beforehand, pulses on the observe channel should be preceded by a delay of 10 microseconds, to make sure the pulse only starts when full output stability is reached. The unblanking time can be determined by performing a single pulse experiment (`s2pul` sequence with a sample of doped D_2O) using a very short pulse width (typically 0.2 to 0.5 microseconds) and arraying `rof1` in the range of 0 to 10 microseconds.

  On the other hand, if the amplifier was already unblanked beforehand (perhaps because another pulse is preceding the current `rgpulse`), the amplifier is still turned on (unblanked) and does not require a special delay. From an amplifier point-of-view, back-to-back pulses are no problem at all.
- To find out about the 90-degree ("quadrature") **phase shifting times,** we need to know how these phase shifts are generated. There are fundamental differences between a UNITY*plus* and prior generations of instruments in the way 90-degree and small-angle phase shifts are generated. In the UNITY*plus*, both kinds of phase shifts are generated on the same (transmitter) board, but by dedicated hardware, as shown in Figure 11 (the mechanism of programming small-angle phase shifts is addressed in Chapter 12, "AP Bus Traffic," on page 137). The 90-degree phase shifts are generated in a special circuitry (a "twisted ring shift register") that generates four 10.5 MHz frequencies (0, 90, 180, and 270 degrees) from 42 MHz, out of which one is selected by means of two digital input lines (90 and 180 degrees phase shift flags)[3]. The switching between the four phases is virtually immediate (TTL gates, switching times in the order of nanoseconds).

[3] For wideline and multipulse spectroscopy, the 90-, 180-, and 270-degree phase angles can be fine adjusted in the order of millidegrees.

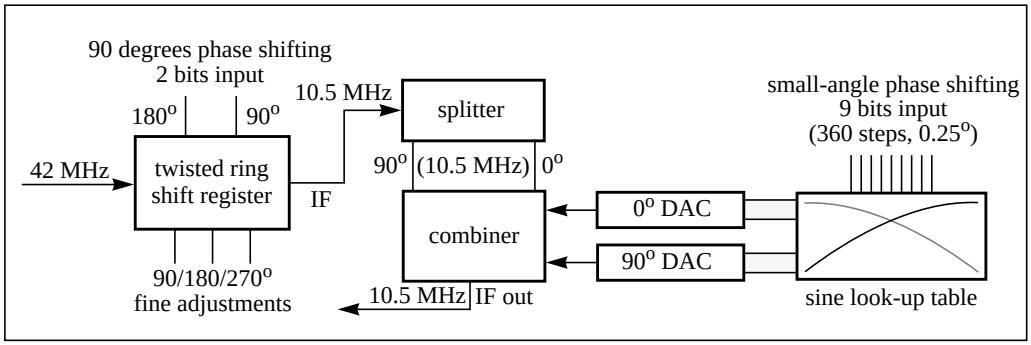


Figure 11. Phase shifting on UNITY*plus* systems

For the small-angle phase shifting, the 10.5-MHz frequency is split into 0- and 90-degree components. Small-angle phase shifting takes 9 bits of input: values run from 0 to 512, with 360 steps for phase angles of 0 to 89.75 degrees, in 0.25-degree resolution. These are translated into digital amplitude values for the 0-degree and 90-degree components: a DAC converts the numeric values into dc voltages, which are then used in Gilbert multipliers to set the amplitude of the two orthogonal 10.5-MHz components. These two components are combined again, generating again a single 10.5-MHz frequency that has the (digital) small-angle phase shifting added in. This is the 10.5-MHz IF (intermediate frequency) that propagates the 90-degree and small-angle phase shifts into the transmitter frequency. The small-angle phase shifting circuitry has no effect on the timing of a pulse on the observe channel.

Figure 12 shows phase shifting on systems prior to the UNITY*plus*. On these systems, both 90-degree and small-angle phase shifts were generated using a 720-step sine look-up table. This generated a frequency of 2.5 MHz, which in turn was used to generate the 10.5 MHz IF.

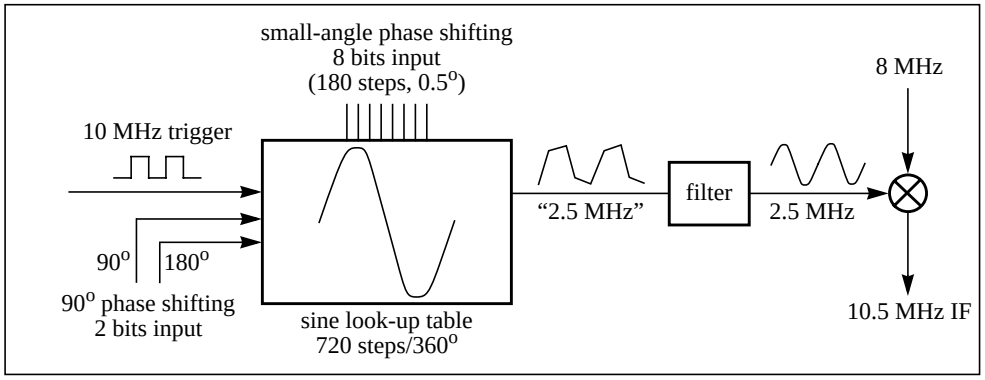


Figure 12. Phase shifting on systems prior to UNITY*plus*

With this type of rf generation, a 2.5-MHz frequency is generated by reading the sine value from a 360-degree look-up table (720 steps, 0.5 degrees resolution) at a rate of 10 MHz. The resulting output (with a very rough digitization in the time axis) is then filtered in order to obtain a pure 2.5-MHz frequency that can then be mixed with 8 MHz to generate the 10.5 MHz IF. Small-angle phase shifts are nothing but an offset (up to 180 steps of 0.5 degrees) in the same look-up table.

The 90-degree phase shifts are selected by means of two digital input lines that cause look-up offsets in increments of 180 addresses.

What is now the timing of a 90-degree phase shift? The look-up offset is implemented immediately (TTL gates), but it only has an effect at the next 10 MHz trigger point, which can be up to 0.1 microseconds later. Also, the filter in the 2.5-MHz line further delays the response, such that with this type of rf a 90-degree phase shift is completed typically after 0.4 microseconds. Thus, delays of 0.5 to 1 microseconds are recommended to allow for the phase to settle.

- The actual **pulse switching** has several components: the rf gates and associated hardware on the transmitter board, the properties of the amplifier, and the inductance of the coil. Similar to the receiver gating, the rf gates on the transmitter board (TTL switches) can be thought of as switching instantaneously (nanoseconds range); therefore, the pulse turn-on time determined only by the amplitude rise time of the amplifier, which is usually determined using an oscilloscope. It turns out that even pulses of only 0.2 microseconds duration is fairly rectangular. Typically, within a few tens of nanoseconds, the output reaches full amplitude.

  The pulse turn-on time is of no concern for the delay prior to the pulse, but it should at least be ensured that the rf is stable when the pulse is turned on. Also, we can do very little in terms of timing in a pulse sequence about turn-on effects from the inductance of the coil (see below for a discussion on probe ringdown).

  The high-power amplifiers used with high-power solid-state NMR experiments (1 kW $^{1}H/^{19}F$ or broadband) are at least as fast as the linear amplifiers used for liquids NMR (which feed the high-power amplifiers); thus, the amplifier considerations for solid-state systems are no different than for liquids experiments.

Taking all this into consideration, we can conclude that there are two cases for the delay preceding an rf pulse on the observe channel:

- If the amplifier was blanked beforehand, pulses on the observe channel should be preceded by a 10-microsecond delay, to allow for the amplifier to stabilize. At the same time, this delay is more than sufficient for the new phase to settle with all types of rf generation.
- If the amplifier was *not* blanked beforehand (either another pulse is preceding the current one or the receiver was switched off during the preceding [time] event), the delay prior to the pulse is mainly a phase-settling delay and can be reduced accordingly. On UNITY-type and earlier rf schemes, a delay of 1.0 microseconds should be more than sufficient for the phase to settle. With UNITY*plus*-type rf (with instantaneous phase switching), 0.2 microseconds is long enough—even back-to-back pulses do not cause problems.

What needs to be considered for determining the setting of the delay *following* a pulse on the observe channel? It turns out that this is simpler, because much less gating is involved and no phase change occurs.

- At the end of the pulse, the gates on the transmitter board close immediately; the **pulse turn-off time** should be in the order of nanoseconds. The amplifier turn-off time may be slightly longer, but this should not be a concern within most of the pulse sequence, except for the last pulse, where we should make sure that no “pulse break-through” occurs, which could affect preamplifier performance.
- The **receiver gates** operate at about the same speed as the transmitter gates.

- There are phenomena outside the actual rf that also need to be considered here, namely **probe ring-down** and related effects. The inductance (or better called the quality factor, *Q*) of the rf coil causes a finite delay in turning off the pulse (rf ring-down). With liquids probes, the rf ring-down time constant is typically 200 nanoseconds (high-field, ^{1}H) to several microseconds (low-gamma nuclei). Wideline and high-power CRAMPS probes have a lower *Q* and therefore much shorter ring-down time constants of 20 to 50 nanoseconds at proton frequency[4].

  Also (and often even more important), the coil experiences considerable mechanical forces during an rf pulse that shakes the coil mechanically (**acoustic ringing**), even if the coil is fixed firmly. These mechanical movements induce currents in the coil, and these currents in turn may affect the preamplifier and the receiver, distorting the beginning of the FID.

In conclusion, we have the following different situations for the delay after a pulse:

- If data are acquired immediately following the pulse (i.e., after an excitation pulse rather than after a refocusing pulse) or if pulses are performed in-between acquisition points, a delay is recommended after the last pulse in order to avoid probe ring-down saturating the preamplifier. For normal liquids experiments (^{1}H, ^{13}C, ^{31}P, ^{19}F), 10 to 20 microseconds should be sufficient; for low-gamma nuclei, longer values are often appropriate (if the T_2 permits); and for typical solid-state NMR experiments, shorter values can be used because these probes have a much shorter ring-down time.
- For all other pulses in a pulse sequence, preamplifier saturation should not be a problem because it generally has sufficient time to recover. Hence, in a typical liquids experiments, the post-pulse delay can be set to zero for all pulses except the last pulse.
- If the receiver is switched off “globally” within a pulse sequence (e.g., switched off before the first pulse), all post-pulse delays *except for the last* can be set to zero.

Simple Pulses on Other RF Channels

Pulses on any rf channel can be performed using the `genpulse` statement the same way as used for the observe channel, except that the device `OBSch` is replaced by `DECch`, `DEC2ch`, or `DEC3ch` for the first, second, or third decoupler channel, respectively (see the footnote on page 40). As mentioned before, `genpulse` is not generally used (nor recommended) except for cases where the rf channel (the device) should be held under parameter control.

Typically, such pulses are performed using one of the following statements:

```
decrgpulse(length,phase,rx1,rx2);
decpulse(length,phase);
dec2rgpulse(length,phase,rx1,rx2);
dec3rgpulse(length,phase,rx1,rx2);
```

[4] The rf ring-down time constant t_r is proportional to the quality factor *Q*, and inversely proportional to the rf frequency *f*: `tr=`$Q/(3f)$. Liquids probes have a high *Q* of around 300 to 400, the *Q* of high-power wideline and CRAMPS probes is typically 10 times lower. After that time, the voltage across the coil has decayed to 1/e of its initial value. The total ring-down time t_{tr} for a pulse at power *P* (in watts), down to a residual level of 1 microvolt at an impedance of 50 ohms can be calculated as

$$t_{tr} = \left(-ln\left(\frac{1}{\sqrt{50P}}\right)\right) \times Q/(3f)$$

These statements are equivalents to `rgpulse` and `pulse` (for the second and third decoupler channel, no `pulse` equivalents have been defined). Unlike `pulse`, the pre- and post-pulse delays in `decpulse` are set to zero. All these statements are macros and are resolved to calls to the generic `G_Pulse` statement by the C preprocessor.

The gating scheme around decoupler pulses is basically the same as for pulses on the observe channel. A difference exists insofar as the decoupler amplifier(s) are usually set into continuous wave (CW) mode (i.e., they are constantly on[5]). Amplifier (noise) blanking is usually not required on these channels because their output is almost always be directed onto a different rf coil and can hardly reach the receiver[6]. For this reason a pre-pulse delay is only used for the phase to settle (particularly on UNITY and earlier systems; 0.5 to 1 microseconds are recommended), and a post-pulse delay is almost never necessary. On UNITY*plus* systems both delays can be left at zero length.

Simultaneous Pulses on Different RF Channels

The VNMR definition of simultaneous pulses is such that they should occur centered on top of each other. Based on this, a gating scheme can be deduced, for example, for `simpulse` (a simultaneous pulse on the observe channel and the first decoupler):

1. Gate receiver off, set phase on all channels involved.
2. Perform pre-pulse delay.
3. Switch on transmitter gate on the channel with the longer pulse.
4. Perform a delay; length: half the difference between the two pulse lengths.
5. Switch on transmitter gate on the channel with the shorter pulse.
6. Perform a delay; length: the shorter of the two pulse lengths.
7. Switch off transmitter gate on the channel with the shorter pulse.
8. Perform a delay; length: half the difference between the two pulse lengths.
9. Switch off transmitter gate on the channel with the longer pulse.
10. Perform post-pulse delay.
11. Gate receiver on, unless it has been switched off globally.

The C construct for such a statement uses the same basic calls as the `rgpulse` statement described above; `if–else` statements are used to decide which device to turn on first and which to turn off last. Similarly, statements for simultaneous pulses on three and four channels can be constructed. All these statements exist; they have the syntax:

```
simpulse(len1,len2,ph1,ph2,rx1,rx2);
sim3pulse(len1,len2,len3,ph1,ph2,ph3,rx1,rx2);
sim4pulse(len1,len2,len3,len4,ph1,ph2,ph3,ph4,rx1,rx2);
```

The indices in the arguments refer to the observe channel (1) and the first three decoupler channels (2, 3, 4). In all these statements, the observe channel is involved; therefore, (because in general we cannot, and do not want to, make assumptions about

[5] Except in situations where the decoupler nucleus is identical to the observe nucleus or falls into the same amplifier frequency band (^{3}H, ^{1}H, ^{19}F vs. the nuclei at ^{31}P frequency and below). In this case, decoupling occurs through the same amplifier as the observe pulses, which are in pulse mode and are therefore blanked.

[6] Provided there is little or no crosstalk between the different coils in a probe. Exceptions would also be heteronuclear experiments on a single, double-tuned coil. In such cases the amplifier would have to be either put into pulsed mode or blanked explicitly, see Section 4.3, “Other State-Related Pulse Sequence Statements,” on page 52.

the length of the individual pulses) for the pre-pulse delay the same considerations apply as for the pre-pulse delay to `rgpulse`, assuming that the amplifier for the longest pulse needs to be unblanked first, unless the receiver was already switched off during the previous time event.

The above statements are macros in which `simpulse` and `sim3pulse` are resolved into calls to the following C functions:

```
gensim2pulse(len1,len2,ph1,ph2,rx1,rx2,dev1,dev2);
gensim3pulse(len1,len2,len3,ph1,ph2,ph3,rx1,rx2,dev1,dev2,dev3);
```

These functions take the corresponding number of device arguments (rf channel addresses, see the footnote on page 40); these devices can be specified in any order as long as the pulse lengths and the phases are specified in the same order. The macro `sim4pulse` is translated into a call to the function `G_Simpulse` that will not be discussed any further here. Similar to `genrgpulse` and `genpulse`, these functions are not supported by the `dps` command at this point in time, and their use is not recommended unless there is a need to keep the rf channel selection under parameter control (with the presence of the `rfchannel` parameter, those functions are not longer required).

Composite Pulses

From a pulse sequence or gating point of view, composite pulses are nothing but a sequence of ordinary pulses without explicit delays in-between. As outlined in the sections above, all pulses but the first can be preceded by a very short pre-pulse delay only, to allow for the phase shift to complete. On UNITY*plus* systems, even that delay can be set to zero (on other systems this may lead to a slight performance loss). For the pre-pulse delay of the first pulse, the same considerations apply as for a normal pulse (see above).

All post-pulse delays should be set to zero to avoid unnecessary gaps between the pulses. The maximum permissible gap between components of a composite pulse is determined by the question whether noticeable precession can occur during such delays. In liquids experiments, gaps of 1 microsecond should be harmless.

Considerations for the Delays Following the Last Pulse

Additional considerations apply to the post-pulse delay after the last rf pulse on the observe channel (better said as the pulse that generates the final, detected coherence), because that delay affects both the reliability of the autophasing (`aph`) as well as the first-order phase parameter (and with that the baseline flatness). Why is that?

In the last receiver stages before the ADC, the NMR signal (with IF +/– δ) is mixed with the intermediate frequency in order to obtain audio frequencies (signals at `-sw/2` to `+sw/2` Hz). At that level, any noise (and extra signals) outside the spectral range need to be filtered away in order to avoid folding noise and undesired signals into the spectrum (the extra noise would dramatically affect the signal-to-noise ratio). Such a filter delays the audio signal by a time that is inversely proportional to the filter bandwidth. This is taken care of by the software—the filter delay (shown in Figure 13) is calculated automatically and made part of the implicit acquisition, where prior to acquiring the first data point a delay of `alfa+1/(beta*fb)` is performed.

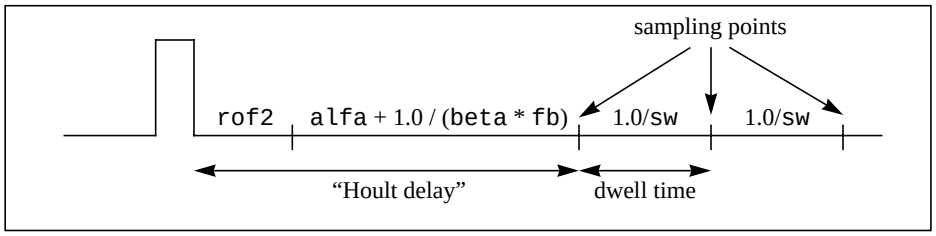


Figure 13. Filter delay

`alfa` is a standard parameter (in microseconds, defined as a pulse), `fb` is the filter bandwidth, and `beta` is a constant that depends on the type of audio filter in use. For 4-pole Butterworth filters (used in VXR and earlier spectrometers), `beta` is 2.0; for 8-pole quasi-elliptical filters (the standard filter in UNITY and UNITY*plus* systems), `beta` is 1.29; and for 6-pole Bessel filters (used for spectral windows above 100 kHz), it is 2.33[7]. With this setup, the user can adjust the (last) post-pulse delay, while having an independent parameter for regulating the total delay between the last pulse and the first sampling point.

In 1983, D. Hoult et al.[8] noted that by making the *total* delay between the last pulse and the first sampling point equal to the theoretical filter delay (for Butterworth filters, 1/(2*`fb`)), the best baseline flatness was obtained. This seems to disagree with the standard setup in Varian pulse sequences (with `rof2` and `alfa` being 10 and 20 microseconds). What are the consequences of these additional delays? It turns out that with these delays, the first-order phase correction parameter `lp` has a negative value that depends on the spectral window `sw`. In fact, the `lp` parameter is decremented by 360 degrees with every dwell time by which we right-shift the first sampling point. If the condition described by Hoult is fulfilled, `lp` is close to zero—an approximation for the `lp` parameter can be calculated easily from `rof2` and `alfa`[9]:

$$\mathtt{lp} = -\frac{\mathtt{rof2} + \mathtt{alfa}}{10^6 \times (1/\mathtt{sw})} \times 360 = -(\mathtt{rof2} + \mathtt{alfa}) \times \mathtt{sw} \times 360 \times 10^{-6}$$

The real problem is that due to the filter response, the first data point is never accurate. An offset in the first data point leads to a dc offset in the transformed spectrum (`lp=0`). The first-order phase correction converts this dc offset into a sinoidal baseline distortion: at a correction of ±360 degrees, a full sine wave is generated, and with every additional 360-degree deviation of `lp` from zero, an additional sine wave is obtained. Also, as `lp` increases, this baseline roll increases in amplitude (with very high values for `lp`, it is not unusual that the baseline roll becomes larger than the NMR signals!).

Using the above formula, we can estimate that for proton spectra (spectral windows of 2000 to 10000 Hz) first-order phase corrections of –20 up to –100 degrees are obtained with the standard settings for `rof2` and `alfa`. For typical carbon spectra (spectral windows of 10 to 50 kHz), the corrections are more severe with 100 to 500 degrees. In many cases—especially proton spectra on normal organic compounds—this baseline roll is noticeable but can easily be corrected by means of data processing software (`bc` command). In many other cases, such a baseline roll is undesirable—it can even ruin

[7] In the $^{1}H/^{13}C$ version of Gemini spectrometers, 3-pole Butterworth filters with a filter constant `beta` of 3.0 are used.
[8] D.I. Hoult, C.-N. Chen, H. Eden and M. Eden, *J. Magn. Reson.* **51**, 110 (1983).
[9] The division by 10^6 is needed because within VNMR `rof2` and `alfa` are defined as pulses and are entered in microseconds.

the quality of multidimensional spectra. To help cure this, the VNMR software contains the `hoult` command, which calculates `rof2` to be one-third of the theoretical filter delay (`1/(beta*fb)`), as described in Hoult's paper, and then sets `alfa` to the *negative* value of `rof2`. This only works because the implicit delay that is executed is the *sum* of `alfa` and the filter delay.

The `hoult` command brings `lp` down to a value close to zero—but for many applications, especially biological NMR, this is still not good enough. Why? The reason for that is twofold: first, even the best filters are never 100.0% accurate (all electronic components have a limited accuracy) and hence the real filter delay is always slightly differ from the theoretical value. Second, it cannot be assumed that the excitation point (the point where all excited spins are in phase) is at the end of the rf pulse, because the precession *during* the pulse cannot be neglected.

Overall, it is impossible to calculate the exact filter delay. Consequently, an *empirical* timing correction can be used: a (1D) spectrum is first acquired, transformed, and phase-corrected, and then the above equation is modified to recalculate `alfa` from `lp`. This is done by calling the macro `calfa`:

$$\texttt{alfa}_{\texttt{corr}} = \texttt{alfa} + (\texttt{lp} \times 10^6)/(360 \times \texttt{sw})$$

With this new value for `alfa`, the spectrum is then reacquired. It can be assumed, that this correction stays the same as long as the spectral window is not changed (although this probably is only true as long as the pulse width is also not altered).

Another implication from the delays between the last pulse and the first sampling point is the fact that the autophasing command `aph` first estimates a value for `lp`, assuming that both `rof2` and `alfa` were used after the last pulse. It must therefore be ensured that the last pulse is in fact performed with a post-pulse delay `rof2` (or an equivalent delay must be inserted after the last pulse); otherwise, the reliability of the autophasing can be severely affected.

4.3 Other State-Related Pulse Sequence Statements

Direct Gating

As shown in the sample coding from the previous sections, rf gates can also be operated directly. A number of simple, documented commands exist for the transmitter gates (`xmtron/xmtroff`, `decon/decoff`, `dec2on/dec2off`, `dec3on/dec3off`), receiver gate (`rcvron/rcvroff`), spare gates (`sp1on/sp1off`, `sp2on/sp2off`), and decoupler amplifier blanking control (`decblank/decunblank`, `dec2blank/dec2unblank`, `dec3blank/dec3unblank`). Pulse statements operate these gates implicitly and automatically. If a user decides to operate gates explicitly, it is the user's responsibility to switch them back into "normal" mode at an appropriate time in the pulse sequence.

The statements for explicit 90-degree phase shifting (`txphase`, `decphase`, `dec2phase`, `dec3phase`) can be regarded as special (phase) gating statements (with the difference that they do not control rf gates or blank an amplifier). All these gating statements have no immediate effect on what is happening in the rf: they only influence the gate settings of subsequent time events.

Implicit Gating

The `status` statement can partly be regarded as implicit gating statement. Primarily, `status` changes the value of the variable `statusindx` to the specified flag field. At a secondary level, it also switches the decoupler gates depending on the corresponding status fields in the `dm`, `dm2`, and `dm3` parameters. These only become active in subsequent time event. The definition is such that if a flag has less fields than addressed in the pulse sequence, the last flag field is taken instead (`status` takes an integer as argument; the constants `A` to `Z` [0 to 25] are defined in `psg/rfconst.h`).

In earlier rf schemes, the decoupler modulation mode (`dmm`) was also on direct status lines (two lines for up to four modulation modes) and was set by `status` the same way as the other gate lines; in UNITY and UNITY*plus* systems, the decoupler modulation modes (`dmm`, `dmm2`, `dmm3`) are set through a different mechanism, see Chapter 12, “AP Bus Traffic,” on page 137.

4.4 Basic Purpose of a Pulse Sequence

The main purpose of a pulse sequence is to define a sequence of *times* (delays) and a series of associated *states* (on/off information). Pulses—even if composite or simultaneous on various rf channels—can be deduced from a sequence of simple delays with variable gating information. From that point-of-view, the heart of the pulse programmer can be diagrammed as shown in Figure 14.

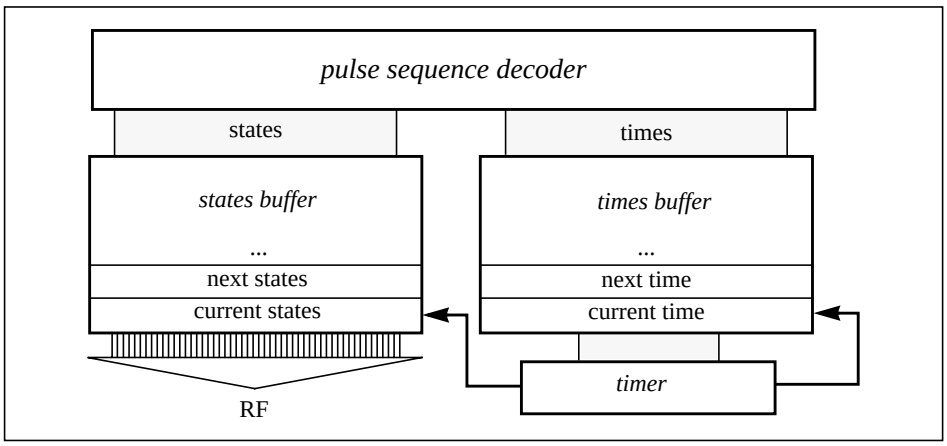


Figure 14. Block diagram for a pulse programmer

This is not only a block diagram for what the pulse programmer is doing, but, as we see later, this diagram contains the “essence” of the pulse programmer hardware in all Varian NMR spectrometers from the XL, VXR, and UNITY, to the Gemini, UNITY*plus*, *GEMINI 2000*, and UNITY*INOVA*. Of course, there are additional features (we shall learn about rf devices that can not be addressed simply with a limited set of on/off states), but the central idea is already in the diagram—the pulse programmer is a state–times buffer with associated timing electronics that makes the code interpretation (the acquisition CPU) independent (asynchronous) of the pulse sequence timing.

Chapter 5. Submit to Acquisition: go

The go command (and related commands such as au and ga) has the central task of running the compiled (executable) sequence from the seqlib directory and submitting the experiment to the acquisition process Acqproc (go does not perform the acquisition itself). go then hands the control back to VNMR. It is up to Acqproc to manage the acquisition and notify VNMR about completed blocks, FIDs, and experiments.

Figure 15 outlines the processes and actions around the go command.

5.1 The Tasks for go

The command go directly executes and initiates the following steps:

- From the parameter seqfil, the go command finds out what pulse sequence is to be executed. If such a pulse sequence executable is not found in the user or the system seqlib directory, go aborts with an error message.
- If an executable pulse sequence is found, go strips all non-acquisition parameters off the current parameter tree and adds acquisition parameters from the global tree along with all the system configuration parameters. This parameter collection is then transferred to the pulse sequence.[1]

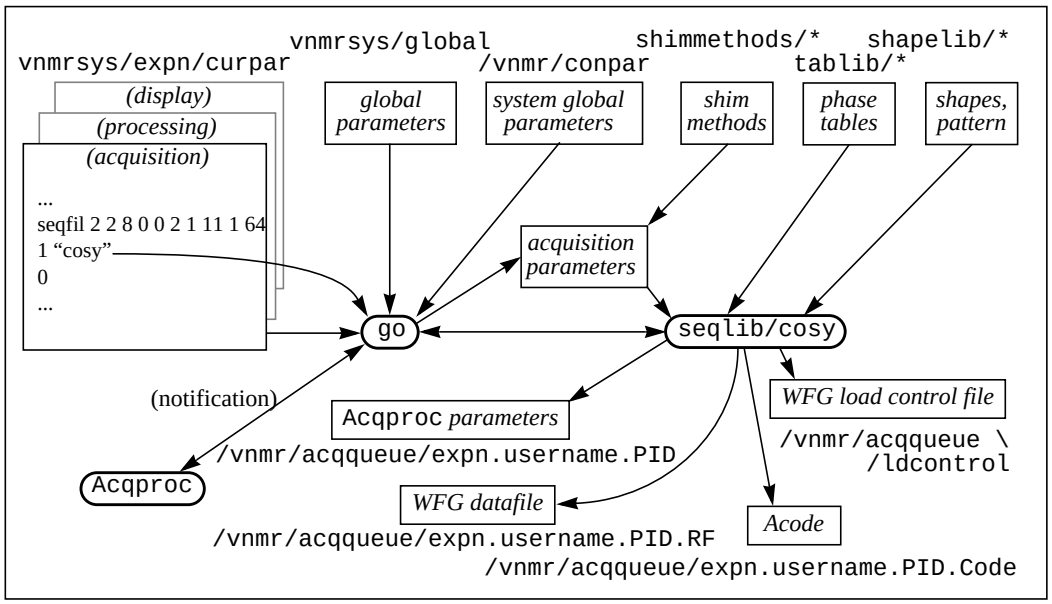


Figure 15. Processes surrounding the go command

[1] Because only acquisition-, sample- and configuration-related parameters reach the pulse sequence, a getval statement on a display, processing, or spin simulation parameter *does not work* (a zero value is returned instead for numeric parameters)!

- The shim method from the directory `shimmethods` (as specified by the parameter `method`) is packed into an internal parameter `com$string` and added to the set of transferred acquisition parameters.
- `go` then executes the sequence from `seqlib`. A compiled pulse sequence in the user's sequence library (`~/vnmrsys/seqlib`) takes precedence over a sequence (with the same name) from the system sequence library (`/vnmr/seqlib`).
- The execution of the pulse sequence first involves the run-time linker, which binds the other objects from `psg/libpsglib.so.x.y` (see Section 2.5, "Compiling and Linking," on page 24) to the pulse sequence. As part of the run-time linking, a revision check is made on the run-time link library `psg/libpsglib.so.x.y`. If the revision of that file (`.x.y` extension of the file name) is not correct, it results in a "run-time" error message or even in a crash. For such a case, check whether an old compiled pulse sequence (from a previous software release) was executed instead of an updated one in `/vnmr/seqlib`). If that happened, recompile the pulse sequence. It could also be that you used `psggen` (see Chapter 3, "Object Library Generation: psggen," on page 29) with an earlier software release, and the local `psg` directory (`~/vnmrsys/psg`) was not deleted when a new release was installed. In this situation, you must delete (or re-create and update) the local `psg` directory, and recompile the pulse sequence if it was a local file.[1]
- If the pulse sequence does not return an error code upon execution, `go` notifies the acquisition process `Acqproc` of the new experiment in the acquisition queue. After that, the VNMR part of the `go` command is finished, and VNMR is ready for executing the next command.
- `go` usually appends a new experiment to the end of the acquisition queue, except if `go` (or `au`) was called with the argument `'next'`, in which case the experiment is queued immediately after the running experiment (insertion at the front of the queue) or immediately before the next acquisition is started, when called as part of conditional `werr` or `wexp` processing of an experiment that was started with the command `au('wait')`.
- If VNMR cannot contact `Acqproc` after several attempts, an error message "`maximum number of retries exceeded . . PSG aborted`" or "`The acquisition daemon 'Acqproc' is not active!`" is issued, and no experiment is started. In this case, either `Acqproc` has died and needs to be started again, or for some reason VNMR is unable to contact `Acqproc` (it may be hanging), in which case `Acqproc` needs to be killed and restarted again. At this point, any running experiment needs to be restarted, and queued experiments need to be requeued (see Section 6.4, "Controlling Acqproc," on page 63). After this, call `go` again.
- The absence of `Acqproc` makes `go` non-functional on stand-alone data stations (except for `go('acqi')`, see also Section 5.3, "Using go('acqi')," on page 58).

[1] These problems are avoided if `makeuser` is called for every account and the local files are updated through this command. `makeuser` deletes old compiled pulse sequences (not the source file from `psglib`) and disables any local `psg` directory.

5.2 Tasks for the Pulse Sequence Executable

The primary task of the pulse sequence executable is to generate instructions and data that `Acqproc` later uploads into the acquisition computer plus data to be used by `Acqproc` to control that experiment. The resulting information is stored in several files in the acquisition queue directory `/vnmr/acqqueue`. The names for most of these files reflect the experiment name (`exp1` to `exp9`), the name of the user (`username`), and the process-ID number of the pulse sequence executable (`PID`). Obviously there would be a problem if a user did not have write permission into `/vnmr/acqqueue`.

- Parameters are read from the set of acquisition parameters that is handed over by `go`; they are stored in local variables.
- Specified waveform generator (WFG) shapes and patterns are read from `shapelib` and transformed into a binary file `expn.username.PID.RF` in `/vnmr/acqqueue`. This file is later uploaded via HAL into the waveform generator(s). Together with `expn.username.PID.RF`, a second file named `/vnmr/acqqueue/ldcontrol` is generated or updated. This second file contains information about the size and structure of all waveforms and decoupling pattern to be used. If a system is not equipped with waveform generators, this file is not generated or updated; if waveform generators are installed, the file `/vnmr/acqqueue/ldcontrol` is updated even if no waveform generator is actively used in the experiment.
- The main task for the pulse sequence executable is the generation of instructions to be interpreted by the acquisition CPU called the *Acode*. The Acode is a sequence of 16- and 32-bit words, including instructions and a complete set of local variables used by the acquisition CPU, that are stored in a file in the directory `/vnmr/acqqueue`. The file name is `expn.username.PID.Code`. Typically, the Acode size is 1 to 3 Kbytes *per FID*. For multidimensional experiments, the Acode can easily fill many megabytes of disk space and can eventually create "`disk full`" problems.
- External phase tables are read from the specified file in `tablib`; they are built into the file with the instruction list for the acquisition CPU (see Chapter 9, "Pulse Programmers," on page 85).
- A file `/vnmr/acqqueue/expn.username.PID` is generated that contains acquisition control parameters the parameters that `Acqproc` needs to control the acquisition, store the FID in the proper place, and initiate the right actions when certain key conditions (error condition, block size completed, FID or experiment completed) are reached. This file is a standard ASCII parameter file, containing a set of parameters that are required for `Acqproc` to do its job (see also Chapter 6, "Acquisition Process," on page 59).

5.3 Using go('acqi')

Entering `go('acqi')`, or the equivalent macro `gf`, serves to generate the Acodes necessary to run `acqi` in interactive (FID or spectrum) mode. The same as entering `go` with no arguments, the pulse sequence executable is called, but in the case of `go('acqi')`, nothing is submitted to `Acqproc`, and the resulting files are read solely by the `acqi` program. For this reason, different naming conventions are used:

- `acqi.Code` replaces `/vnmr/acqqueue/expn.username.PID.Code`.
- `acqi.RF` replaces `/vnmr/acqqueue/expn.username.PID.RF` (only used when shaped pulses or rf patterns involving a waveform generator are involved).
- `acqi.par` replaces `/vnmr/acqqueue/expn.username.PID`; this file (also an ASCII parameter file) is meant for `acqi` directly; it tells `acqi` how to display the FID or the spectrum interactively (`sp`, `wp`, etc.).
- `ldcontrol` is the same as with the normal `go` command (only used if waveform generators are present).

Entering `go('acqi')` or `gf` is often also used to test a pulse sequence up to the point of Acode generation (without doing a real acquisition yet) or to permit inspection of the Acode and associated files. Note that the Acode generated by `go('acqi')` differs in several points from the code generated through a normal `go`:

- Only Acode for the first FID is generated,
- Parameter `alock` is set to `'n'` (no automatic locking),
- Parameter `load` is set to `'n'` (shim values are not loaded),
- Parameter `wshim` is set to `'n'` (no automatic shimming occurs),
- Parameter `ss` is set to `0` (no steady-state pulses),
- Parameter `dp` is set to `'y'` (32-bit acquisition),

The macro `gf` modifies even more parameters:

- Parameter `cp` is set to `'n'` (the observe phase `oph` is not cycled automatically),
- Parameter `gain` is set to `'y'` (use fixed gain),
- The lock to be set into the “fast loop” mode.

By definition, the parameters used for shimming on the FID or on the spectrum (as well as the parameters for interactive parameter adjustment) are those for the first increment of an arrayed experiment. Also, `gf` sets the parameters such that excessive startup delays are avoided (`alock`, `wshim`, `ss`, and `gain` parameters). For the FID mode in particular, `cp='n'` avoids jumping baselines and/or intensities by stopping the phase cycling (assuming the `cp` flag or `oph` are used in the pulse sequence).

Note also that the Acode segment for the first FID—as obtained via `go('acqi')`—can *not* be taken as a model for subsequent code segments (both in size and contents), because it contains a number of rf initialization instructions that are not repeated for every other FID. The first Acode segment is always bigger than all following segments of an arrayed or multidimensional experiment.

Chapter 6. Acquisition Process

The acquisition process `Acqproc` operates in three phases: uploading and starting the acquisition operating system, uploading data for the current acquisition, and downloading acquired FIDs and associated information. `Acqproc` is the main software link from the host computer to the acquisition cabinet (the actual spectrometer)[1].

`Acqproc` is a background process that must be constantly running on a spectrometer. It is started at boot-up time through `/vnmr/rc.vnmr` (which is called by `/etc/rc.local`) if the file (flag) `/etc/acqpresent` is found. `/vnmr/rc.vnmr`, the UNIX kernel (`/vmunix`) with the Varian device driver, and the `acqpresent` flag are installed together with the VNMR software by the `setacq` installation script.

From a hardware point-of-view, the host computer is linked to the acquisition CPU through the HAL (Host-to-Acquisition Link) board, a board with a SCSI interface. Thus, the two computers are connected by a SCSI bus. Unless there is a second SCSI bus used exclusively for acquisition, the same SCSI bus is usually used for communication with hard disks, tape drives, and a CD-ROM drive (if present)[2]. Acting as a special device on the SCSI bus, the HAL board requires a special device driver in the UNIX kernel (i.e., a special, proprietary communication protocol is used between the the host computer and the acquisition CPU).

6.1 Starting the Acquisition Operating System

Upon startup, `Acqproc` checks if the acquisition CPU has resident software. If not (e.g., after initialization), `Acqproc` uploads the acquisition operating system (i.e., the software that permanently resides in the acquisition CPU). The files that form the acquisition operating system are stored in the directory `/vnmr/acq`:

- `rhmon.out` is the part of acquisition CPU operating system that controls the communication with the host computer through the HAL board.
- `autshm.out` is the autoshimming part of the operating system.
- `xrop.out` is the central part of operating system (Acode decoder) for systems with output board (63-step FIFO).
- `xrxrp.out` is the central part of operating system (Acode decoder) for systems with acquisition control board (1024-step FIFO, mostly UNITY systems).
- `xrxrh.out` is the central part of operating system (Acode decoder) for systems with pulse sequence control board (2048-step FIFO, UNITY*plus* systems).

---

[1] The interactive acquisition program `acqi` talks directly to the acquisition CPU, be it for locking, for shimming, or for interactive acquisition.

[2] Because the electrical specification of the SCSI bus would impose a severe restriction with respect to the maximum cable length between the two computers (or any two SCSI devices), Varian uses a special “differential box” that adds an electrically driven branch to the SCSI bus. This box permits using a maximum cable length of over 10 meters between the host computer (actually the differential box) and the acquisition cabinet.

- `xr.conf` is the part of operating system that returns information about the rf configuration to `Acqproc` and the `config` program (`/vnmr/bin/vconfig`).

In earlier VNMR releases, a symbolic link named `xr.out` was created to one of the two files `xrop.out` and `xrxrp.out` when calling the `setacq` script, in which the user had to reply whether the system was equipped with 63-step or 1024-step loop FIFO. The task of selecting the right version of `xrop.out`, `xrxrp.out`, or `xrxrh.out` is taken over by `Acqproc`, which gets the necessary hardware information from the `xr.conf` part of the acquisition CPU operating system.

It is possible (although unlikely) that after the installation of a new version of VNMR, the acquisition CPU still has an old version of the acquisition operating system loaded. In such a situation, `Acqproc` may not "realize" the revision discrepancy, because the acquisition CPU is still running on uploaded software, which in turn leads to error messages or even a system hangup—at the very least when trying to communicate with the acquisition CPU. After loading VNMR (before starting the new `Acqproc`), the acquisition CPU *must* be reinitialized.

6.2 Queuing and Starting the Acquisition

`Acqproc` monitors the acquisition CPU. If no experiment is running or when the acquisition CPU is ready to accept the next Acode segment (instruction set and data for one FID), `Acqproc` uploads that Acode (and waveform generator data files if required) to the acquisition CPU (actually to the Host-to-Acquisition Link) and initiates the acquisition. Figure 16 shows the communications scheme controlled by `Acqproc`.

`Acqproc` controls not only the data acquisition, but also the queuing of acquisitions from different experiments and different users. The experiments in the queue are executed in the order of their submission[3]. "Acquisitions" here also include queue

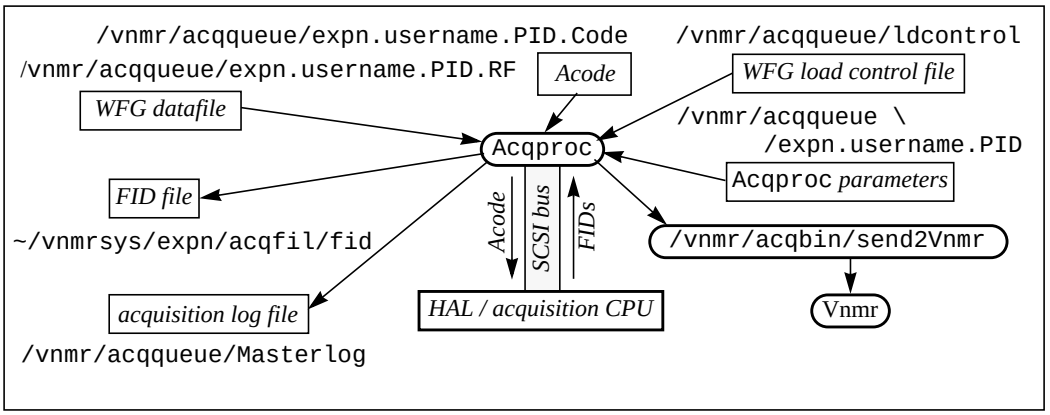


Figure 16. Acquisition control by `Acqproc`

[3] The only means of manipulating the queue is to abort the current acquisition (`aa` from the current experiment), to stop it temporarily and go to the next queued experiment (`sa` from the active experiment), to remove an acquisition from the queue (`sa` from the queued experiment before it becomes active), or to add a new experiment to the end of the queue.

items that do not generate NMR data as submitted by `change`, `spin`, `lock`, `shim`, `sample`, and `su`; these will not be discussed in detail here because their actions can also be made part of a “normal” experiment (as started with the `go`, `ga`, or `au`).

The process of starting up an experiment (after its submission by `go`) involves the following steps:

- If either the acquisition CPU is idle or has just terminated the previous experiment, the next item in the acquisition queue is launched.
- If waveform generator-shaped pulses, decoupling pattern, or shaped gradients are involved, the necessary waveform generator data are uploaded into the acquisition CPU memory from `/vnmr/acqqueue/expn.username.PID.RF`. The file `/vnmr/acqqueue/ldcontrol` contains the control information that permits `Acqproc` to instruct the acquisition CPU about the structures within the waveform generator data file, and where to load these structures (pattern, waveform).
- The Acode for the first FID is uploaded into the acquisition CPU as well, and its interpretation is initiated.
- In arrayed or multidimensional experiments, the Acode segment for the next FID is loaded ahead of time, such that the acquisition can switch from one FID to the next without having to upload Acode in between FIDs. This is called buffered acquisition; it dramatically reduces the dead-times between FIDs (no extra delays for data transfers or disk access).
- `Acqproc` also acts as a message handler and transmits interrupt messages (such as generated by the `aa` or `sa` commands) to the acquisition CPU.

6.3 Downloading the FID

`Acqproc` polls the acquisition CPU at regular (dynamic) intervals. If data are ready (after a block size or a completed FID or experiment), `Acqproc` downloads it onto the host computer memory and stores the data in the experiment directory from which the acquisition was started. This involves the following actions and mechanisms:

- The data transfer from the acquisition CPU (the HAL) to the host computer occurs through polling by `Acqproc`. `Acqproc` polls at regular intervals, but tries adjusting the polling rate to the rate at which data are ready in the acquisition CPU. This avoids unnecessary dead times.
- If data are ready (block size or FID completed, `sa` interrupt completed), `Acqproc` downloads the FID from the HAL memory to the host computer.
- `Acqproc` then checks whether a lock file `~/vnmrsys/lock_n.primary`[4] exists for the current experiment. If a lock file is found, its contents are inspected

[4] The lock file `~/vnmrsys/lock_n.primary` has the experiment number coded in the file name (`lock_n`). The extension `.primary` indicates that it is a *primary lock file*. A *secondary lock file* is generated temporarily before generating the primary one, in order to avoid that two processes lock the same experiment at the same time (i.e., between detecting that “no lock file exists” and the creation of a lock file, another process could in theory create the same lock file; the secondary lock file avoids this). The lock file contains a code for the kind of lock (`3` = foreground), the host name of the system on which the locking process runs, and the process-ID (PID) of the locking process. The PID permits detecting whether the locking process is still active.

in order to find out whether a foreground VNMR is active in the current experiment.

- The FID is stored in the appropriate experiment by over-writing or updating the applicable section of the FID file (`~/vnmrsys/expn/acqfil/fid`). The ownership of the FID file is transferred (`chown`) to the experiment owner. During this process, the active experiment is locked by `Acqproc` (acquisition lock).
- If a foreground VNMR is running in the current experiment, it is notified through `send2Vnmr`, and its (buffered) processed parameter tree is updated (`ct` and `celem` parameters). If conditional processing was specified (`werr`, `wbs`, `wnt`, or `wexp` parameters), this processing is initiated through the `send2Vnmr` call. The `send2Vnmr` call uses the following syntax:

  ```
  ps -ax | /vnmr/acqbin/send2Vnmr “VNMR command string”
  ```

- If no foreground VNMR is found, or if no foreground VNMR process is running in the current experiment but conditional processing was specified, that processing is performed through a background VNMR call:

  ```
  Vnmr -mback -nExp# “VNMR command string”
  ```

  During this process, the active experiment is locked by VNMR (background lock).
- A log file `/vnmr/acqqueue/Masterlog` is maintained that contains information about submitted experiments, interrupts and completion points, as well as information about acquisition-related processing.
- When the experiment completes, the queue files `expn.username.PID.Code`, `expn.username.PID.RF` (if present), and `expn.username.PID` in `/vnmr/acqqueue` are deleted, and the next experiment is submitted to the acquisition CPU.

All the information that `Acqproc` needs for handling downloaded FIDs are contained in the parameter file `/vnmr/acqqueue/expn.username.PID`. This file includes the following parameters:

- Parameters `dp`, `np`, `nf`, and `arraydim` that contain data format information.
- Parameters `bs`, `nt`, `ct`, `celem`, and `interleave` (the latter corresponding to `il`) that allow `Acqproc` to determine whether a block or an FID are completed and which code segment to load next (if `interleave='y'`, cycle block-wise through all FIDs; otherwise, switch to the next FID when the previous one is completed).
- Parameters `werr`, `wbs`, `wnt`, and `wexp` that specify what action needs to be initiated in VNMR in the case of an error condition, a completed block, a completed FID, or completed experiment.
- Parameter `wait` that specifies whether the following experiment can be started immediately following the current one (`wait='n'`, the standard case) or whether the conditional (`wexp` or `werr`) processing should be completed first (`wait='y'`). Setting `wait` to `'y'` allows for another experiment to be started on the same sample using `au('next')`, before switching to the next sample (i.e., at the front of the acquisition queue).
- Parameters `gain` and `spin` are used in connection with the Acquisition Status window. Unused are parameters `priority`, the queuing priority, and `suflag`.

Path and other file information:

- Directory `fidpath` stores the FID file.
- `curexp` is usually the parent of `fidpath`.
- `userdir` is usually the parent of `curexp`.
- The variable `systemdir` is usually `/vnmr`.
- `id` is the name of the parameter file itself; it is also the root part of the names for the Acode and waveform generator data files.
- `date` is a time stamp for the submission of the experiment

6.4 Controlling Acqproc

Because `Acqproc` needs to be able to store FIDs and modify parameter files in every user's experiment directories, it must be a process owned by `root`. Also, only `root` can use `chown` to transfer the ownership to the respective users. If `Acqproc` were owned by a standard VNMR user, only that user could acquire data[5] (note that only one copy of `Acqproc` per system can run at a time).

The `root` ownership for the process `Acqproc` can create problems: in case of errors (in particular, experiencing communication failures on the SCSI bus) every user should be able to kill and restart `Acqproc`. And in the case of software troubles in the acquisition system, the acquisition CPU sometimes needs to be restarted, which should only be done while `Acqproc` is shut down (otherwise very likely `Acqproc` will not be able to communicate with the HAL board). UNIX processes can only be killed by their respective owners, or by `root`; this implies that only `root` can kill `Acqproc`, and `root` must restart it, unless the SUID protection bit (4000) is set. A consequence could be that many people need to know the `root` password, which of course is totally unacceptable from a security point of view.

The solution for this situation is a special account `acqproc` that is equipped with `root` privileges (UID and GID are the same as for `root`) and which may or may not be equipped with a password. In networked environments, a password (different from the `root` password, of course) is certainly recommended; however, this precludes resetting `Acqproc` from the `.rootmenu` to generate such an account. A shell script `/vnmr/bin/makesuacqproc` is provided, which must be executed as `root`.

When logging into the account `acqproc`, no shell is obtained (as with usual UNIX accounts), but a shell script `/vnmr/bin/execkillacqproc` (only executable by `root`) is executed instead, after which the user is automatically logged out again immediately. `/vnmr/bin/execkillacqproc` kills `Acqproc` if it is found in the process table; otherwise, it starts `Acqproc` (as `root`). This means that for restarting (killing and starting) `Acqproc`, the user must call `su acqproc` twice. Broadcast messages indicate whether `Acqproc` has been aborted or started (this message is displayed in every active UNIX window and the VNMR master window).

[5] The other users could open up the permissions in their directories and files, but this would not solve all problems; also, it is undesirable to loosen the file system security by making so many files writable by everybody (or at least the group `nmr`).

For SunView, by adding the line

```
"Fix Acquisition"    su acqproc
```

to a user's `~/.rootmenu` file, this function becomes available in the SunView `.rootmenu`. This can only work if the account `acqproc` has no password, because there is no keyboard input to commands that are executed from the `.rootmenu`.

For OpenWindows, the menu file is `~/.openwin-menu`, and the line to be added is as follows:

```
"Fix Acquisition"    exec su acqproc
```

Chapter 7. Digital Components

For the remaining sections of this manual, having at least a rough idea about the structure of the digital part of the spectrometer is essential. Figure 17 gives on overview of the relevant components that are involved—more details are provided in the following chapters.

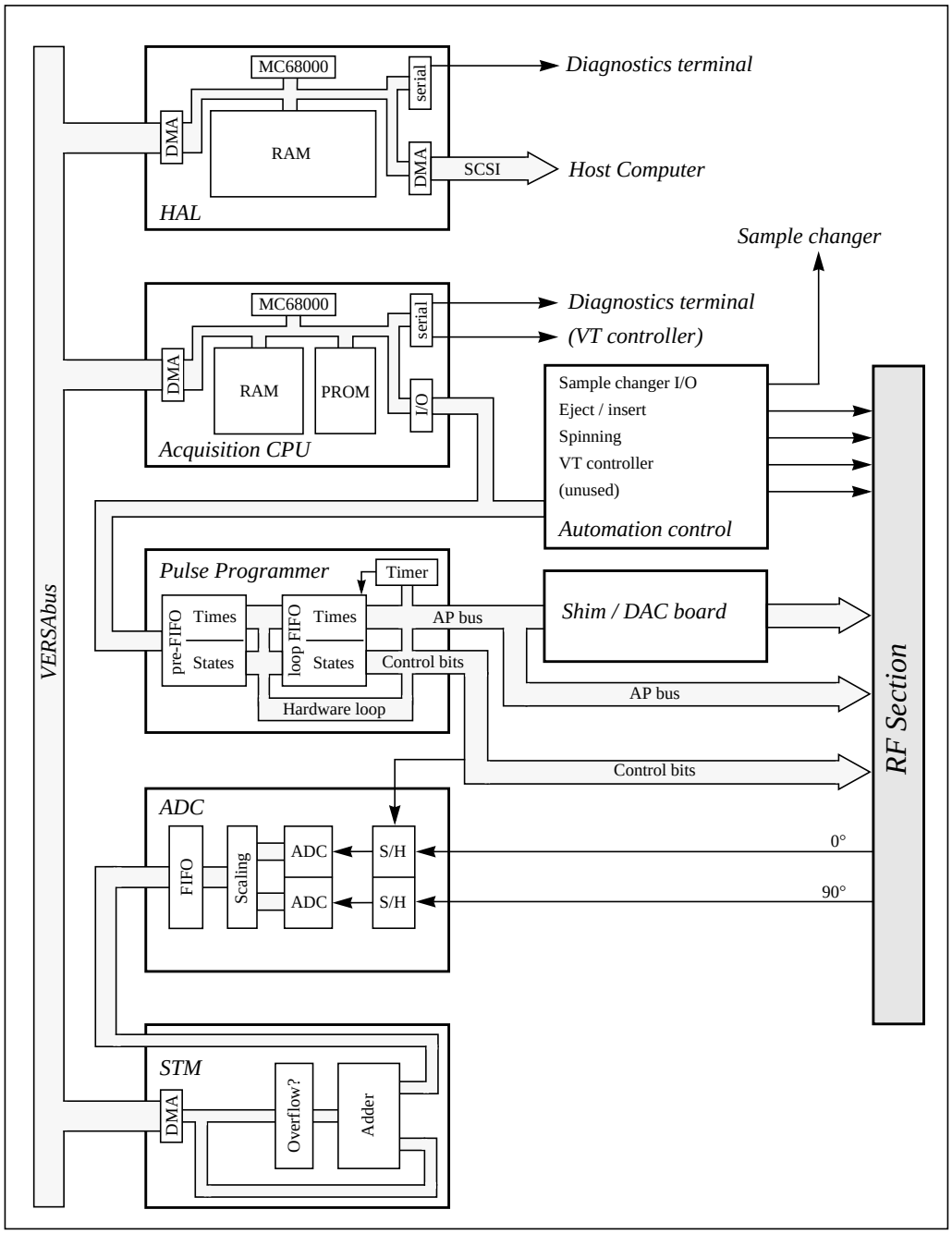


Figure 17. Digital components of a spectrometer

It turns out that there are two CPU boards in the acquisition computer: the Host-to-Acquisition Link (HAL) and the acquisition CPU. Both use a Motorola MC68000 CPU chip and have a serial port for a diagnostics terminal. The two computers fulfil different tasks (the HAL is dedicated to communications with the host computer and it holds the actual NMR data), but they operate on the same bus, share the same address space, and work on the same data. Therefore, in subsequent chapters we treat them both as one single computer (the fact that there are two CPUs is irrelevant to the user).

7.1 Main Boards

The tasks for the main components in the digital part of the spectrometer can be summarized as follows:

- The main purpose of the **HAL board** is to act as a link between the host computer and the acquisition CPU. Through its SCSI interface, the host computer (`Acqproc`) can upload not only Acode (see Chapter 8, “Acquisition CPU and Acode,” on page 69), but also the acquisition operating system through DMA (direct-to-memory access) into the acquisition CPU. After an experiment, the NMR data are downloaded through the same interface (see Chapter 6, “Acquisition Process,” on page 59).

  The other purpose of the HAL board is to hold the current NMR data in its RAM. For arrayed and multidimensional experiments, there is sufficient RAM to hold both the current and the next FID. After completion of the scans for an FID, this permits continuing with the next data set without delay (“buffered acquisition”).

- The main task of the **acquisition CPU** is the interpretation of the Acode, generating the information that is fed into the pulse programmer. With the aid of the programmer, it also perform the tasks of autolocking and autoshimming (the spinner speed regulation is done by dedicated peripheral circuitry).

  The acquisition CPU is equipped with boot PROMs, containing a primitive operating system that is running after switching on or resetting the CPU, up until the real acquisition operating system is loaded (see Section 6.1, “Starting the Acquisition Operating System,” on page 59). The acquisition operating system fills most of its RAM, which during acquisition also holds the Acode, the data, and instructions needed for performing an experiment (see Chapter 8, “Acquisition CPU and Acode,” on page 69).

  The acquisition CPU sends information to the pulse programmer and exchanges information with the automation board through an on-board parallel I/O channel (see Chapter 13, “Acquisition CPU Communication,” on page 145).

  The acquisition CPU has two serial ports, one of which can be used for a diagnostics monitor (either to display diagnostic information during NMR experiments or to run PROM-based on-board diagnostics software). In UNITY and earlier spectrometers, the other serial port was used to exchange information with the VT controller (the Oxford VTC-4); in UNITY*plus* spectrometers, this is done via automation board.

- In a UNITY*plus* spectrometer, the **automation board** drives five serial ports, through which information is exchanged with the sample changer (ASM-100 or SMS), the magnet leg pneumatics control circuitry (eject/insert, slow drop, and bearing air flows), the spinner control circuitry, and the VT controller (one serial port is currently unused). In UNITY and earlier spectrometers, the second serial

port of the acquisition CPU was used for the VT controller. Instead, on these systems the automation board also drove the lock power, phase and gain controls, and the receiver gain; all these functions are now addressed via the AP bus (see Chapter 12, “AP Bus Traffic,” on page 137).

In a UNITY*plus*, the automation board also contains the bootup mode selector and some battery-buffered RAM (“zero-power RAM”) to store the rf configuration.

- The main task of the **pulse programmer** is to accurately control the timing during an NMR experiment. It also acts as a buffer for timing and control information; it directly controls most of the rf gates and has a dedicated output channel (the AP bus, see Chapter 12, “AP Bus Traffic,” on page 137) through which it can send information to most parts of the spectrometer. It controls the shim gradients and all devices that set frequency offsets, attenuations and pulse amplitudes, phase shifts and pulsed field gradient amplitudes (see also Chapter 9, “Pulse Programmers,” on page 85).
- The **ADC (Analog-to-Digital Converter) board** receives two audio signals (0 and 90 degrees) from the receiver board (via an audio filter). Each of the channels is fed into sample-and-hold and ADC circuitry. The data sampling is triggered by the pulse programmer. If necessary, the data are also scaled down on the ADC board. At its output, there is a FIFO (first-in-first-out) buffer, from which the data are fed directly into the sum-to-memory board.
- The **STM (Sum-to-Memory) board** reads in the current FID from the HAL board through DMA and adds in the new FID from the FIFO buffer on the ADC board, checks for mathematical overflow (at which point the FID will be scaled by another bit), and stores the result back in the HAL memory. The STM board does complex additions according to the current receiver phase (see Chapter 18, “Acquiring Data,” on page 205).

7.2 Bus Structures

The main components in the digital part of the spectrometer are linked with different bus structures that carry the data traffic:

- **SCSI (Small Computer System Interface) bus** is an 8-bit parallel bus that links the HAL board, the host computer, and most host computer peripherals (disks, tapes, CD-ROM, etc.).
- **Acquisition CPU bus** is a 16-bit parallel bus structure that is integrated in the cardcage backplane (**VERSAbus**). It carries the data traffic between the acquisition CPU, the HAL board, and the STM board. It also provides dc power to the other boards in the same card cage (e.g., the automation board).
- **I/O bus** is an 8-bit parallel bus that is driven by the acquisition CPU. This bus connects the acquisition CPU, automation board, and pulse programmer.
- **AP (analog port) bus** is the most important link between the pulse programmer and most of the rf devices. Apart from the gating lines, the AP bus carries information on frequency offsets, attenuation and power modulation levels, pulsed field gradient amplitudes, small-angle phase shifts—all the numeric information for the rf part in general. The AP bus is also used to transfer shapes and patterns to the waveform generators. On the UNITY*plus*, lock power, lock phase, lock gain, and the receiver gain are set over the AP bus (on UNITY and earlier spectrometers,

this was done with the automation board). The AP bus is 16-bit parallel (see also Chapter 12, “AP Bus Traffic,” on page 137).

- On UNITY*plus* systems, the gating information (“fast lines”) is routed in its own bus, the **HS (high-speed) bus** (see Chapter 9, “Pulse Programmers,” on page 85).

The components shown in Figure 17 do not cover the entire digital part of the spectrometer, but mainly the computing part. Especially on more recent instruments, such as the UNITY*plus*, digital components reach much further into the spectrometer:

- The waveform generators (see Chapter 16, “Waveform Generators,” on page 163) are completely digital.
- Each transmitter board consists of two parts: the rf board and a digital control board (see Section 16.1, “How Does a Waveform Generator Fit Into the System?,” on page 163 and Section 4.2, “How Do Pulses Work?,” on page 40).

Because virtually everything in the spectrometer is digitally controlled, digital components are found on every subunit of the spectrometer (see also Section 12.2, “What Devices are Driven by the AP Bus?,” on page 139).

Chapter 8. Acquisition CPU and Acode

After the acquisition CPU is “running” (i.e., once the acquisition operating system has been uploaded and started) and “active” (performing an experiment), it in essence holds the following data blocks:

- Acquisition operating system (see Section 6.1, “Starting the Acquisition Operating System,” on page 59), including the Acode interpreter.
- Current FID.
- FID-specific data and instructions (Acode).

8.1 CPU Address Space

The operating system and the Acode are located in the RAM of the acquisition CPU; the FID is stored in the RAM of the HAL board. For the rest of this chapter, we will discuss the Acode alone. With respect to the acquisition CPU and the HAL board, you don’t have to be concerned with anything but the data in the address space of the acquisition CPU and the way these data are interpreted and used.

The CPU address space also includes status registers on the STM (sum-to-memory) board. This enables communicating with a board driven by firmware (“hard-coded software”) and permits transferring acquisition information (number of points, observe phase) to the STM board (see also Chapter 18, “Acquiring Data,” on page 205).

8.2 Looking at Acode

Acode (`acqi.Code` from `go('acqi')` or `expn.username.PID.Code` for normal acquisitions, both in `/vnmr/acqqueue`, see also the previous chapters) is primarily a binary file, mostly consisting of 16-bit integers, interspersed with bit patterns (organized in 16-bit binary words) and some 32-bit long integers. As 16-bit units dominate (and everything is organized in 2- or 4-byte units), the `od -i` command (or `od -s` under Solaris 2.3) is an almost adequate way to look at this file:

```
UNITY400:vnmr1 - 1> od -i /vnmr/acqqueue/acqi.Code
0000000       0       1       0       6       0     421       0  -28096
0000020       0      16       0       0       0       0       0       0
0000040       0       0       0       0       0       0       0       0
0000060       0       0       0       1       0       0     268      94
0000100       1     293       3     106     106      85       4       0
0000120       0       0       0       0      64       0      16      16
0000140       0       0       0       0     512    1024    1536       0
0000160    4096    8192   12288       0       0       1       0       0
0000200       0       0       0       0       0       0       0       0
0000220       0       0       0       0      63       0       0       0
0000240       0       1       2       3       0       0       0       0
0000260       0       0       0       0       0       0       0       0
0000300       0       0       0       0       0       0       0    1542
0000320       0       0       0       0      20      31      59       1
0000340      15       1       0       0       0       8       0      16
0000360       0      24       0      24      11     148      15       2
0000400       0       0       0     256       0     512       0     768
0000420       0     768      11     164      53       0       1      19
0000440       5     157       1       4       6       6  -21696  -17651
0000460  -25856  -21676  -17428  -25809  -25713       6       8  -22752
```

```
0000500   -18688   -26690   -18449   -18514   -18437   -18498   -26879   -18688
0000520      159        1        8   -22752   -18688   -26690   -18449   -18514
0000540   -18437   -18498   -26879   -18688        0       16      257        0
0000560       68        1      360       65      769        0       59     2867
0000600     1025       79       16       55       59     2966      514     4095
0000620        0     4095        6        3   -21608   -17454   -25856   -25856
0000640        6        1   -21614   -17504        6        1   -21605   -17664
0000660        6        1   -21615   -17536        6        1   -21616   -17647
0000700        6        7   -32255   -32236   -32224   -32207   -32191   -32171
0000720   -32156   -32143       16      258        0       59     2866     1025
0000740       79       16       30       59     2982      514     4095        0
0000760     4095        6        3   -21592   -17622   -25856   -25856        6
0001000        1   -21598   -17472        6        1   -21589   -17664        6
0001020        1   -21599   -17536        6        1   -21600   -17646        6
0001040        1   -21688   -17663        6        1   -21708   -17579        6
0001060        1   -21707   -17664        6        1   -21683   -17631        6
0001100        1   -21693   -17663        6        1   -21687   -17664        6
0001120        1   -21706   -17658        6        1   -21605   -17528        6
0001140        1   -21589   -17528      151     4129       71       63        2
0001160      440    30000      250        0      151    12297        7        9
0001200       95        4      151    12787        7        9       75        1
0001220        7     1792    15363        0        0     5229    27904        0
0001240       43        1        0      256     8390     4123       19        5
0001260       98      150        0        0        6        1   -21605   -17528
0001300      150        0        0        6        1   -21589   -17528      150
0001320        0        0      150        0        0      150        0        0
0001340      150        0        0       16        1       74      150        0
0001360        1      150        0        1      151     4489      150        0
0001400        3      151     4369      150        0        1      151     4489
0001420      150        0        0      150        0        0       39       74
0001440       39       16        1       74       16        2       74      152
0001460     8487     4117       90       99        1        0   -28096     8390
0001500     4123        7       97       20      232
0001512
```

This Acode is for a simple experiment (`s2pul`, `p1=0`, `d1=0`, `d2=0`) and obtained with the `gf` macro ("real" Acode for arrayed or multidimensional experiments can be *very* long, see Section 5.2, "Tasks for the Pulse Sequence Executable," on page 57). Looking at the above code, we can already (roughly) distinguish two parts: the top part that consists mainly of "simple numbers" and the bottom part that has a lot of "funny numbers" (indicating that there are things other than 16- or 32-bit integers).

The `-i` option causes `od` to interpret the file given in the argument as 16-bit signed integers; bit patterns that have the most significant bit set, large unsigned 16-bit integers, negative or very large 32-bit integers result in large negative numbers being shown. `od -l` would interpret most 32-bit integers correctly (those which start at an odd address), but would misinterpret most of the file by taking two 16-bit numbers for a single 32-bit number; bit pattern would again mostly be shown as very large numbers. Other options for `od` (for octal or hexadecimal output) make the result even less interpretable. Clearly, a tool is needed that properly takes all the bits and pieces of the Acode apart.

Methods of Interpreting the Contents of Acode Files

The information on the contents of the Acode is found (somehow) in `/vnmr/psg`, although not in a single place or file, but rather convoluted and spread over many source and header files. In general it is *not* necessary to know about the contents of the Acode, although for people who do low-level pulse sequence programming (or modify files in `/vnmr/psg`), an Acode decoder can sometimes provide extremely valuable debugging information—especially because the `dps` command does not interpret certain low-level commands and does not show phase tables, real-time phase math, and even some of the "normal" statements, such as `offset`.

- In the remaining parts of this manual, Acode contents are shown as output by a program that has been submitted to the user library (`userlib/bin/apdecode`). This program shows Acode structures in detail (all contents); the instruction part is decoded in that only the instruction codes themselves are shown, whereas arguments to code functions are (partly) decoded.

  `apdecode` is certainly not a perfect solution because it doesn't try decoding everything; it will also not work with all possible rf configurations. Also, because the Acode undergoes extensive changes with every VNMR release, `apdecode` is undergoing constant adaptations, and it, in general, only works with the VNMR release for which it has been written. The main purpose for showing `apdecode` output in this manual is to illustrate specific contents of the Acode and to show the effect that specific functions (phase calculations, phase tables) have on Acode.

  An alternative option for looking at Acode (and related things) is the `'debug'` option to the `go` command, which will display extensive (debugging) information in the window from which VNMR was called (the parent window). For anyone who is not an expert, `go('debug')` cannot be recommended because its output is much too detailed to be of real value. The output of `apdecode` is much more compact (but may be incomplete at times).

- The output of the Acode decoder `apdecode` on the same Acode as shown above looks as follows:

```
Number of traces: 1
Code Start: 6
Code Ends: 421

 aaddr faddr laddr value   comment
 ----- ----- ----- -----   -------
                           ########### LC (Low Core) Structure ###########
     6     0     0 37440   LC->np (long)
     8     2     2    16   LC->nt (long)
    10     4     4     0   LC->ct (long)
    12     6     6     0   LC->isum (long)
    14     8     8     0   LC->rsum (long)
    16    10    10     0   LC->dpts (long)
    18    12    12     0   LC->autop (long)
    20    14    14     0   LC->stmar (long)
    22    16    16     0   LC->stmcr (long)
    24    18    18     0   LC->rtvptr (long)
    26    20    20     1   LC->elemid (long)
    28    22    22     0   LC->squi (long)
    30    24    24   268   LC->idver
    31    25    25    94   LC->o2auto (Offset to AUTOD)
    32    26    26     1   LC->ctctr
    33    27    27   293   LC->dsize (blocks)
    34    28    28     3   LC->asize (blocks)
    35    29    29   106   LC->codeb (Offset to Code)
    36    30    30   106   LC->codep
    37    31    31    85   LC->status
    38    32    32     4   LC->dpf (0=int, 4=long)
    39    33    33     0   LC->maxscale
    40    34    34     0   LC->icmode
    41    35    35     0   LC->stmchk
    42    36    36     0   LC->nflag
    43    37    37     0   LC->scale
    44    38    38    64   LC->check
    45    39    39     0   LC->oph
    46    40    40    16   LC->bsval
    47    41    41    16   LC->bsctr
    48    42    42     0   LC->ssval
    49    43    43     0   LC->ssctr
    50    44    44     0   LC->ctcom
    51    45    45         LC->dptab      0  0x200  0x400  0x600
    55    49    49         LC->obsptb     0 0x1000 0x2000 0x3000
```

```
 59    53    53     0   LC->rfphpt
 60    54    54     0   LC->curdec (unused)
 61    55    55     1   LC->cpf (cycle phase flag)
 62    56    56     0   LC->maxconst
 63    57    57     0   LC->tablert
 64    58    58     0   LC->output card status reg
 65    59    59     0   LC->analog port status reg
 66    60    60     0   LC->analog port data reg
 67    61    61     0   LC->analog port address/control reg
 68    62    62     0   LC->input card status reg
 69    63    63     0   LC->input card data reg
 70    64    64     0   LC->input card ocsr reg
 71    65    65     0   LC->stm card status reg
 72    66    66     0   LC->tpwrr (xmtr power)
 73    67    67     0   LC->dpwrr (dec power)
 74    68    68     0   LC->tphsr (xmtr phase shift)
 75    69    69     0   LC->dphsr (dec phase shift)
 76    70    70    63   LC->dlvlr (dec level)
 77    71    71     0   LC->srate (High Speed Rotor Freq)
 78    72    72     0   LC->rttmp (temp real time->interlock)
 79    73    73     0   LC->spare1 (unused)
 80    74    74     0   LC->zero
 81    75    75     1   LC->one
 82    76    76     2   LC->two
 83    77    77     3   LC->three
 84    78    78     0   LC->v1
 85    79    79     0   LC->v2
 86    80    80     0   LC->v3
 87    81    81     0   LC->v4
 88    82    82     0   LC->v5
 89    83    83     0   LC->v6
 90    84    84     0   LC->v7
 91    85    85     0   LC->v8
 92    86    86     0   LC->v9
 93    87    87     0   LC->v10
 94    88    88     0   LC->v11
 95    89    89     0   LC->v12
 96    90    90     0   LC->v13
 97    91    91     0   LC->v14
 98    92    92     0   LC->(v15)
 99    93    93     0   LC->(v16)

                         ############ AUTOD: Automation Data ############
100    94     0     0   AUTOD->checkmask (long)
102    96     2     0   AUTOD->when_mask: load='n' wshim='n'
103    97     3  1542   AUTOD->control_mask
104    98     4     0   AUTOD->best
105    99     5     0   AUTOD->loops
106   100     6     0   AUTOD->sample_mask (tray location)
107   101     7     0   AUTOD->sample_error
108   102     8    20   AUTOD->recgain
109   103     9    31   AUTOD->lockpower
110   104    10    59   AUTOD->lockgain
111   105    11     1   AUTOD->lockphase

                         ############## Instruction Section ##############
112   106     0    15   SETPHATTRibutes CH1       0    0x8   0x10   0x18
                                       allBits =   0x18, addr = 0xb0094
126   120    14    15   SETPHATTRibutes CH2       0  0x100  0x200  0x300
                                       allBits =  0x300, addr = 0xb00a4
140   134    28    53   ACQBITMASK 0
142   136    30     1   CBEGIN
143   137    31    19   INITialize acq
144   138    32     5   CLEAR
145   139    33   157   LocKFILTER fast = 1, slow = 4
148   142    36     6   APBOUT  7 items 0xab40 0xbb0d 0x9b00 0xab54 0xbbec
                                         0x9b2f 0x9b8f
157   151    45     6   APBOUT  9 items 0xa720 0xb700 0x97be 0xb7ef 0xb7ae
                                         0xb7fb 0xb7be 0x9701 0xb700
168   162    56   159   TUNE_FREQ       CH1  9 words
                           0xa720 0xb700 0x97be 0xb7ef 0xb7ae 0xb7fb 0xb7be
                           0x9701 0xb700
```

```
180   174    68     0   NO_OP
181   175    69    16   SETPHAS90       CH1c     0
184   178    72    68   PHASESTEP       CH1    360 units (90.00 degrees)
187   181    75    65   SETPHASE        CH1f     0
190   184    78    59   APChipOUT APaddr 11, reg 51, +logic, 1 byte
                          max 79, offset 16, value 55
196   190    84    59   APChipOUT APaddr 11, reg 150, -logic, 2 bytes
                          max 4095, offset 0, value 4095
202   196    90     6   APBOUT  4 items 0xab98 0xbbd2 0x9b00 0x9b00
208   202    96     6   APBOUT  2 items 0xab92 0xbba0
212   206   100     6   APBOUT  2 items 0xab9b 0xbb00
216   210   104     6   APBOUT  2 items 0xab91 0xbb80
220   214   108     6   APBOUT  2 items 0xab90 0xbb11
224   218   112     6   APBOUT  8 items 0x8201 0x8214 0x8220 0x8231 0x8241
                                       0x8255 0x8264 0x8271
234   228   122    16   SETPHAS90       CH2c     0
237   231   125    59   APChipOUT APaddr 11, reg 50, +logic, 1 byte
                          max 79, offset 16, value 30
243   237   131    59   APChipOUT APaddr 11, reg 166, -logic, 2 bytes
                          max 4095, offset 0, value 4095
249   243   137     6   APBOUT  4 items 0xaba8 0xbb2a 0x9b00 0x9b00
255   249   143     6   APBOUT  2 items 0xaba2 0xbbc0
259   253   147     6   APBOUT  2 items 0xabab 0xbb00
263   257   151     6   APBOUT  2 items 0xaba1 0xbb80
267   261   155     6   APBOUT  2 items 0xaba0 0xbb12
271   265   159     6   APBOUT  2 items 0xab48 0xbb01
275   269   163     6   APBOUT  2 items 0xab34 0xbb55
279   273   167     6   APBOUT  2 items 0xab35 0xbb00
283   277   171     6   APBOUT  2 items 0xab4d 0xbb21
287   281   175     6   APBOUT  2 items 0xab43 0xbb01
291   285   179     6   APBOUT  2 items 0xab49 0xbb00
295   289   183     6   APBOUT  2 items 0xab36 0xbb06
299   293   187     6   APBOUT  2 items 0xab9b 0xbb88
303   297   191     6   APBOUT  2 items 0xabab 0xbb88
307   301   195   151   EVENT1_TWRD     1.000 usec
309   303   197    71   GAINAutomation
310   304   198    63   SETVT Oxford PID 440, temp 3000.0, vtc 25.0
315   309   203     0   NO_OP
316   310   204   151   EVENT1_TWRD        10 msec
318   312   206     7   STartFIFO
319   313   207     9   StopFIFO
320   314   208    95   PADelay 4 words
322   316   210   151   EVENT1_TWRD       500 msec
324   318   212     7   STartFIFO
325   319   213     9   StopFIFO
326   320   214    75   SHIMAutomation mode  = 1, 7 words
                          0x700 0x3c03      0      0 0x146d 0x6d00      0
341   335   229    43   NOISE loop 256 pts, dwell 199 usec + 1.000 usec
342   336   230    19   INITialize acq
343   337   231     5   CLEAR
344   338   232    98   NextSCan
345   339   233   150   HighSpeedLINES  (void)
348   342   236     6   APBOUT  2 items 0xab9b 0xbb88
352   346   240   150   HighSpeedLINES  (void)
355   349   243     6   APBOUT  2 items 0xabab 0xbb88
359   353   247   150   HighSpeedLINES  (void)
362   356   250   150   HighSpeedLINES  (void)
365   359   253   150   HighSpeedLINES  (void)
368   362   256   150   HighSpeedLINES  (void)
371   365   259    16   SETPHAS90       CH1   zero
374   368   262   150   HighSpeedLINES  RXOFF
377   371   265   150   HighSpeedLINES  RXOFF
380   374   268   151   EVENT1_TWRD     10.000 usec
382   376   270   150   HighSpeedLINES  RXOFF TXON
385   379   273   151   EVENT1_TWRD     7.000 usec
387   381   275   150   HighSpeedLINES  RXOFF
390   384   278   151   EVENT1_TWRD     10.000 usec
392   386   280   150   HighSpeedLINES  (void)
395   389   283   150   HighSpeedLINES  (void)
398   392   286    39   ASSIGNFUNC      zero oph
401   395   289    16   SETPHAS90       CH1 zero
404   398   292    16   SETPHAS90       CH2 zero
```

```
  407   401   295   152   EVENT2_TWRD     296 usec + 850 nsec
  410   404   298    90   SETInputCardMode
  414   408   302    99   ACQXX loop np=37440, dwell 199 usec + 1.000 usec
  417   411   305     7   STartFIFO
  418   412   306    97   HouseKEEPing
  419   413   307    20   BRANCH          Offset 232

================================================================
Total code size = 421 words / 842 Bytes / 0.8 KB
================================================================
```

8.3 Structure of Acode Files

The Acode consists of four parts:

- File header
- Data structure `LC` (low-core)
- Second data structure `AUTOD` (automation data)
- Instruction section

In the case of arrayed and multidimensional experiments, the Acode contains one file header plus three other parts: the `LC` data structure, `AUTOD` data structure, and an instruction section *per FID*. The `LC` and `AUTOD` data structures are always the same (within one software release at least): rigid assemblies of 16- and 32-bit numbers according to a predefined scheme. The instruction section is variable in length and contents, and varies considerably with the hardware configuration. The last part of the instruction sections depends on the current pulse sequence. As mentioned before, the first instruction segment is always much longer than all others, because of all the initialization instructions.

Let us now have a more detailed look at the different parts of the Acode. No attempt will be made to explain all parts of the Acode—in particular, only those parts of the data structures are explained that are relevant to the following chapters or which are needed to explain the Acode by itself.

Acode File Header

The Acode has a variable-size file header, consisting of 32-bit (long) integers. The first number describes the *number of code segments* (corresponds to the value of the `arraydim`, except for `go('acqi')` or `gf`, which always produces one code segment only). In the above example (obtained with `gf`), there is one code segment only.

The next (32-bit integer) number is the *offset to the first code segment* as 16-bit (2-byte) address. In this case (one code segment), the header is three 32-bit words long, resulting in an offset of 6 for the first segment (the first number of the header is located at address 0).

The following 32-bit numbers are the number of 16-bit words up to (and including) the last code word for every code segment, or the *addresses of the next code segments*, if any more segments follow. For a standard acquisition (not in Acode generated with `go('acqi')` or `gf`), there are `arraydim` code segments; therefore, the header is `2*(arraydim + 2)` words (16 bits or 2 bytes each) long, or `4*(arraydim + 2)` bytes.

The header information is used by Acqproc to extract individual code segments, because in arrayed or multidimensional experiments the code is transferred to the acquisition CPU segment by segment during the acquisition.

LC Data Structure

For the rest of the Acode, the program apdecode lists three different offsets (in 16-bit words): an absolute word count throughout the entire Acode file (aaddr), a word count per code segment (faddr), and a local word count or address (laddr)—this kind of code offset is used by the Acode interpreter itself: addresses are either offsets within the LC or AUTOD structures, or within the instruction segment (see below).

As explained previously, the term LC stands for "low core." This terminology is historic and has little meaning with respect the memory organization in the acquisition CPU. Both structures (AUTOD and LC) are stored in the acquisition memory the way they are stored in /vnmr/acqqueue, and the same is done with the instruction section, but the three sections are *not* stored in consecutive memory locations (in particular, the instruction section is *not* stored after the two data structures). After they are loaded into their proper memory locations, the two data structures form the "workspace" for the instruction section. Such a workspace could also be "built" by the acquisition software, but there is of course a good reason for creating an image of that workspace together with the instruction section as part of the Acode: the two data structures also serve as container for numeric information that is transferred between the two computers. This becomes obvious when we look at some of the contents of LC (only selected parts are explained here).

LC contains many items that sound very familiar to the pulse sequence programmer: it turns out that all real-time variables (ct, v1, one, etc., see below) are actually (16-bit) *addresses* to some specific word (16-bit integer) within the LC structure. Because they are addresses and not normal integers, a special type codeint was created, describing C variables that contain addresses (code offsets) within the LC structure. As we see later, by using those codeint variables, the pulse sequence program is able to generate Acode that contains these addresses. Based on that, during the acquisition the Acode interpreter can do real-time calculations and operations using the addressed memory locations.

LC starts with a series of long (32-bit and two 16-bit word) integers, containing:

- LC->np, the number of points to acquire, corresponding to the np parameter in VNMR; LC->np is the C syntax for the nt element of a structure named LC—we use this syntax to differentiate the structure element from the np variable in C and the np parameter in VNMR.
- LC->nt, the number of transients to acquire, corresponding to the nt parameter.
- The number of acquired transients, corresponding to the ct parameter in VNMR; as we will see below, all so-called real-time variables in a pulse sequence are 16-bit integers, and all real-time math operations are 16-bit operations; the codeint variable ct in a pulse sequence therefore is defined as the "low-order" half of the LC->ct structure element: ct is first set to the offset to LC->ct and then incremented by one (this is found in /vnmr/psg/psg.c); all code offsets are in (16-bit) words, *not in bytes*. LC->ct is incremented *after* every scan (i.e., during the first scan ct and LC->ct is 0, and the acquisition for one FID is finished when the two long integers LC->ct and LC->nt are the same).

- The index of the current FID and Acode segment (in C pulse sequence code corresponding to the variable `ix` in the C code of a pulse sequence; the index for the first FID is 1, not 0). `LC->ix` is an unsigned, 32-bit integer; therefore, the theoretical maximum number of elements is 4,294,967,296. This may sound excessive, but earlier releases of VNMR had `LC->ix` defined as a 16-bit integer, which limited the number of elements to 32767, a value that is easily exceeded in 3D or 4D experiments.
- "squi", the current set of "quiescent" states: in this location the acquisition software stores (at real time) the set of gating information (the "fast bits") to be used with the next pulse programmer event, and to (or from) which new gate setting is added and subtracted.

The remainder of `LC` consists of 16-bit words and integers, among others including:

- `LC->o2auto`, the offset to the `AUTOD` structure (within the code segment, in 16-bit words), in other words: the size of the `LC` structure in 16-bit words.
- `LC->codeb`, the offset to the instruction section within the code segment (`faddr` in the `apdecode` output) or the size of the sum of the `LC` and `AUTOD` structures.
- `LC->codep`, a pointer to the current code word.
- `LC->dpf`, the "double-precision flag"—an integer that contains either 0 (for 16-bit acquisitions) or 4 (for 32-bit acquisitions, `dp='y'`).
- `LC->dsize`, the data size in 512-byte blocks (for `dp='y'`: `4*np/512`, rounded up to the next full block).
- `LC->asize`, the Acode size in 512-byte blocks (minimum: 3 blocks).
- `LC->scale`, the number of binary scaling operations during an acquisition: whenever the maximum or minimum number in an FID would exceed the numeric range at the selected precision, the ADC output and the current FID are scaled down by a factor of 2 (this corresponds to a right-shift in the binary numbers). For 16-bit acquisitions (`dp='n'`), the range is from -32768 to +32767; for 32-bit acquisitions (`dp='y'`), it is from -2,147,483,648 to +2,147,483,647.
- `LC->maxscale`, the maximum number of permissible scaling operations (right-shifts): before starting any acquisition, the system measures the amount of noise by acquiring 256 data points at the conditions of the experiment. From that noise the maximum number of scaling operations is determined, such that after the scaling the noise is still properly digitized. If at that point more scaling would be required, the acquisition is stopped with the message "maximum number of transients accumulated".
- `LC->cpf`, the "cycle phase flag": 0 stands for `cp='n'`, 1 for `cp='y'` in VNMR; if `cp='n'`, the observe phase `oph` remains constant; otherwise, it is incremented with the `ct` counter.
- `LC->ssval`, the number of steady-state pulses, corresponding to the `ss` parameter in VNMR. A `codeint` (real-time) variable `ssval` contains the *address* (the offset) to `LC->ssval`, such that in a pulse sequence we can instruct the acquisition CPU to perform (mathematical) operations based on the value stored in `LC->ssval`.
- `LC->ssctr`, the counter location, in which the acquisition CPU counts the steady-state pulses; this location is initially set to the value of `LC->ssval` and the

decremented after each transient. During “real” transients, `LC->ssctr` remains set to zero. Also here (equivalent to `ssval`), a `codeint` variable `ssctr` exists that allows performing real-time calculations based on the value of `LC->ssctr`.

- `LC->bsval`, the block size, corresponding to the `bs` parameter in VNMR. A `codeint` (real-time) constant `bsval` contains the offset to `LC->bsval`, such that in a pulse sequence we can instruct the acquisition CPU to perform (mathematical) operations based on the value stored in `LC->bsval`.
- `LC->bsctr`, the counter location, in which the acquisition CPU counts through the block size; at the beginning of the acquisition this location is set to the value of `LC->ssval` and is decremented after each transient; whenever the location `LC->bsctr` contains zero, the FID is stored, and `LC->bsctr` is reset to the value of `LC->bsval`. Again, a `codeint` variable (`bsctr`) exists that allows performing real-time calculations based on the value of `LC->bsctr`, or even recalculating the value of `LC->bsctr` (e.g., for implementing dynamic block sizes).
- `LC->oph`, the observe phase real time variable: `oph` in a pulse sequence is nothing but the `address` (the local offset, `laddr` in the `apdecode` output) to that structure element. If `cp='n'` (`LC->cpf` is 0), `LC->oph` contains 0, otherwise (`cp='y'`, and `LC->cpf` is 1) `LC->oph` is the same as `ct` (the low-order half of `LC->ct`). As the observe phase can only assume four values (0, 1, 2, 3), it doesn’t matter if `oph` contains the values from `ct` that are incremented up to `nt-1`: the software simply looks at the last two (the two least-significant) bits of that 16-bit number, which is identical to a *modulo 4* function, resulting in a sequence 0, 1, 2, 3, 0, 1, 2, 3, etc. Of course, `oph` can also be calculated (or set from a phase table, see the following chapters); in any case, it is *not* necessary to perform a modulo function to ensure values in the proper range of 0 to 3—for the modulo (mod 4) function is implicit!
- `LC->v1`, `LC->v2`, . . . `LC->v14`, 14 storage locations for results of phase calculations, flags, etc.: in a pulse sequence these locations can be addressed via the `codeint` (“real-time”) variables `v1`, `v2`, . . . `v14`. Since real-time math happens detached from the C compiler and the execution of the C pulse sequence, the only way to instruct the acquisition CPU to perform math with specified variables is to the addresses on which mathematical and logical operations are to be performed. There are two kinds of such addresses: addresses to the variables `v1`, `v2`, . . . `v14`, `oph`, `ssctr`, and `bsctr` (the latter two are rarely used as variables), and addresses to constants (see below) and “system variables” like `ct` (as well as `ssval`, `ssctr`, `bsval`, `bsctr`, `tablert`). The integers `LC->v1` up to `LC->v14` are initialized with zero value, unless the user specifies a different value using the `initval` statement. This also is the only way to set the contents of these locations directly from within C; once the experiment is running, the values can and will only be changed by real-time math.
- `LC->zero`, `LC->one`, `LC->two`, `LC->three`, four locations with the numeric values 0, 1, 2, and 3: these numbers are accessible via the real-time (`codeint`) variables `zero`, `one`, `two`, and `three`. As numeric C constants (or variables) cannot exist in the Acode translation, these four frequently used numeric values are stored in real-time locations and made accessible by the address variables `zero`, `one`, `two`, `three`. They all could theoretically constructed from a single value (`zero` or `one`), but it is more convenient to have at least these three numbers available for real-time math.

- `LC->tablert`, this is the location in which phase values from a table are stored in table calls (e.g., the `pulse(pw,t1)` statement), unless the table value is extracted into a real-time variable (`v1`, `v2`, `... v14`, using the `getelem` statement, or into `oph` with the `setreceiver` statement, see also Chapter 11, "Phase Tables," on page 115).

The other elements (addresses) in the `LC` structure are used internally within the acquisition CPU and are rarely of interest to the user, not even for debugging.

The `LC` structure (together with `AUTOD`) is defined in `/vnmr/psg/lc.h`. It is initialized in `/vnmr/psg/psg.c`, which also defines and sets the corresponding `codeint` (real-time) variables. The `LC` and `AUTOD` structures are to be regarded as non-user modifiable, because there are matching counterparts within the VNMR module and in the acquisition operating system.

The AUTOD Data Structure

In the current software, the `AUTOD` structure serves two purposes: it provides parameters and intermediate storage locations for the autoshimming routine, and it contains the current values for the operations that are performed via the automation control board (see Chapter 7, "Digital Components," on page 65). Compared to the `LC` structure, `AUTOD` is small, containing one long (32-bit) integer and ten 16-bit integers:

- `AUTOD->when_mask` is a 16-bit integer that contains the information from the VNMR `load` and `wshim` parameters.
- `AUTOD->checkmask`, `AUTOD->control_mask`, `AUTOD->best`, `AUTOD->loops` are used for the autoshimming.
- `AUTOD->samplemask` is the sample changer location (0 = not used).
- `AUTOD->sample_error` contains sample changer error codes.
- `AUTOD->recgain` is the receiver gain (value as set by the parameter, or as set by the autogain (if `gain='n'`).
- `AUTOD->lockpower`, `AUTOD->lockgain`, `AUTOD->lockphase` are the lock parameters—again, either as set by the parameters or (in case the autolock is selected) as set by the autolock function.

In earlier VNMR releases, `AUTOD` was larger and contained many more functions:

- 32 shim gradient coil values. With the increasing number of gradients and the many different possible gradient sets, this became obsolete and was moved into the instruction section. No fixed space is allocated for the gradient settings.
- The shim method as ASCII text (up to 128 characters). This was moved into the instruction section, where the method text appears in parsed (semi-interpreted) form. No fixed amount of space is allocated for the shim method any longer.
- Knobs information—memory space that was allocated for processes internal to the acquisition CPU. This was removed from `AUTOD` as well.

The Instruction Section

The instruction section of each Acode segment is structured by itself and consists of two subsections:

- The first part does all the initializations and is executed only once per FID.
- The second part contains the code that is generated by the pulse sequence function itself and is looped over `nt`.

We will now take a more detailed look at the two parts of the instruction section, without discussing individual functions (these will be discussed in later chapters).

The Initialization Part

The initialization part of the instruction section contains definitions and initializations. To a very large extent, this part of the code depends on the rf configuration. The more channel and the more devices that exist, the longer this section becomes. Of course, it also depends on the parameter settings (gradient settings, frequencies, power levels, etc.), but these mainly change the numeric (and binary) values used in this section and not so much its length.

In arrayed and multidimensional experiments, this section also depends on the FID index. Certain segments and instructions (as indicated below) only show up in the instruction segment for the first FID and, in general, the instruction segments for subsequent FIDs are slightly shorter. The difference is about 50 to 150 Acode words (about 100 to 300 bytes), depending on the spectrometer configuration.

At the beginning of the initialization part, the binary values for the fast bits that determine the 90-degree phase shifts are defined for each channel, together with an internal address used in connection with phase shifting. This part only shows up in the Acode for the first FID.

```
112   106    0    15   SETPHATTRibutes CH1       0    0x8   0x10   0x18
                                    allBits =   0x18, addr = 0xb0094
126   120   14    15   SETPHATTRibutes CH2       0  0x100  0x200  0x300
                                    allBits =  0x300, addr = 0xb00a4
```

The next, large section includes initialization output to virtually every digital and rf device in the spectrometer (including magnet leg control and shims), ensuring that they are in their proper state, as defined by configuration and non-pulse sequence dependent acquisition parameters. Most of this section consists of AP output (see Chapter 12, “AP Bus Traffic,” on page 137). The lock filter response times are set, all shim gradient settings, frequencies, and power levels are set, amplifiers, rf switches, and relays are set into a proper state. In this section, the `TUNE_FREQ` instructions (setting the PTS frequency synthesizers), the instructions that initialize the waveform generators, and a few other functions are only present as part of the first Acode segment.

```
140   134   28    53   ACQBITMASK 0
142   136   30     1   CBEGIN
143   137   31    19   INITialize acq
144   138   32     5   CLEAR
145   139   33   157   LocKFILTER fast = 1, slow = 4
148   142   36     6   APBOUT  7 items 0xab40 0xbb0d 0x9b00 0xab54 0xbbec
                                       0x9b2f 0x9b8f
157   151   45     6   APBOUT  9 items 0xa720 0xb700 0x97be 0xb7ef 0xb7ae
                                       0xb7fb 0xb7be 0x9701 0xb700
168   162   56   159   TUNE_FREQ       CH1  9 words
                        0xa720 0xb700 0x97be 0xb7ef 0xb7ae 0xb7fb 0xb7be
                        0x9701 0xb700
180   174   68     0   NO_OP
```

```
181   175    69    16   SETPHAS90       CH1c     0
184   178    72    68   PHASESTEP       CH1    360 units (90.00 degrees)
187   181    75    65   SETPHASE        CH1f     0
190   184    78    59   APChipOUT APaddr 11, reg 51, +logic, 1 byte
                          max 79, offset 16, value 55
196   190    84    59   APChipOUT APaddr 11, reg 150, -logic, 2 bytes
                          max 4095, offset 0, value 4095
202   196    90     6   APBOUT  4 items 0xab98 0xbbd2 0x9b00 0x9b00
208   202    96     6   APBOUT  2 items 0xab92 0xbba0
212   206   100     6   APBOUT  2 items 0xab9b 0xbb00
216   210   104     6   APBOUT  2 items 0xab91 0xbb80
220   214   108     6   APBOUT  2 items 0xab90 0xbb11
224   218   112     6   APBOUT  8 items 0x8201 0x8214 0x8220 0x8231 0x8241
                                        0x8255 0x8264 0x8271
234   228   122    16   SETPHAS90       CH2c     0
237   231   125    59   APChipOUT APaddr 11, reg 50, +logic, 1 byte
                          max 79, offset 16, value 30
243   237   131    59   APChipOUT APaddr 11, reg 166, -logic, 2 bytes
                          max 4095, offset 0, value 4095
249   243   137     6   APBOUT  4 items 0xaba8 0xbb2a 0x9b00 0x9b00
255   249   143     6   APBOUT  2 items 0xaba2 0xbbc0
259   253   147     6   APBOUT  2 items 0xabab 0xbb00
263   257   151     6   APBOUT  2 items 0xaba1 0xbb80
267   261   155     6   APBOUT  2 items 0xaba0 0xbb12
271   265   159     6   APBOUT  2 items 0xab48 0xbb01
275   269   163     6   APBOUT  2 items 0xab34 0xbb55
279   273   167     6   APBOUT  2 items 0xab35 0xbb00
283   277   171     6   APBOUT  2 items 0xab4d 0xbb21
287   281   175     6   APBOUT  2 items 0xab43 0xbb01
291   285   179     6   APBOUT  2 items 0xab49 0xbb00
295   289   183     6   APBOUT  2 items 0xab36 0xbb06
299   293   187     6   APBOUT  2 items 0xab9b 0xbb88
303   297   191     6   APBOUT  2 items 0xabab 0xbb88
307   301   195   151   EVENT1_TWRD     1.000 usec
```

The receiver gain is set to the value specified in the corresponding VNMR parameter, and the VT controller is initialized (a temperature setting of 3000 degrees switches off the active VT regulation, the PID regulation parameters are set with every acquisition, and the VT cut-over value is set at the same time). Unless a temperature array is specified, the `SETVT` instruction is only used in the first FID.

```
309   303   197    71   GAINAutomation
310   304   198    63   SETVT Oxford PID 440, temp 3000.0, vtc 25.0
315   309   203     0   NO_OP
316   310   204   151   EVENT1_TWRD        10 msec
318   312   206     7   STartFIFO
319   313   207     9   StopFIFO
```

Next, the preacquisition delay (`pad`) is executed; in Acode segments other than for the first FID, this is omitted.

```
320   314   208    95   PADelay 4 words
322   316   210   151   EVENT1_TWRD       500 msec
324   318   212     7   STartFIFO
325   319   213     9   StopFIFO
```

Depending on the `wshim` parameter, autoshimming is now performed on the first FID only, or for each increment. The shim method has become part of the Acode instructions (it does *not* show up in ASCII text, but rather in parsed form).

```
326   320   214    75   SHIMAutomation mode  = 1, 7 words
                           0x700 0x3c03      0      0 0x146d 0x6d00      0
```

The last element of the initialization part is the noise measurement: for the first FID, 256 noise points are acquired using the spectral window (as well as the gain and filter settings, of course) of the “real” FID. This noise measurement is used to calculate the maximum number of down-scalings (right-shifts) that can be performed under the

current conditions (in case the maximum or minimum integer is reached in the FID), such that the noise is still sufficiently digitized.

```
341   335   229    43   NOISE loop 256 pts, dwell 199 usec + 1.000 usec
342   336   230    19   INITialize acq
343   337   231     5   CLEAR
```

With these instructions, the pulse sequence independent part of the instruction section terminates. The keyword for the pulse-sequence-related section is `NextSCan`, or the Acode instruction `92`.

Pulse-Sequence-Related Part

The pulse-sequence-related part of the instruction section can be as short as 80 Acode words, as in the example (`s2pul`) below. In other cases it can be up to several thousand Acode words. The maximum Acode size for a single FID (including the initialization section) currently is 10,000 words.

The starting point for the pulse sequence section is always the instruction `92` (`NextSCan`). Typically, the pulse sequence function itself is a sequence of math functions (not in this example), AP output (`APBOUT`) statements, instructions that set fast bits (`HighSpeedLINES`, `SETPHAS90`), and time events. Any pulse sequence statement call that can possibly change the fast bits (state information) generates a `HighSpeedLINES` call. In this particular example, we recognize only three time events: `pw` was set to 7 microseconds, and `rof1` and `rof2` both set to 10 microseconds. The other pulse (`p1`) was set to zero (thus the time events associated with `p1` do not show up), and the two delays are set to zero and are therefore skipped.

```
344   338   232    98   NextSCan
345   339   233   150   HighSpeedLINES  (void)
348   342   236     6   APBOUT  2 items 0xab9b 0xbb88
352   346   240   150   HighSpeedLINES  (void)
355   349   243     6   APBOUT  2 items 0xabab 0xbb88
359   353   247   150   HighSpeedLINES  (void)
362   356   250   150   HighSpeedLINES  (void)
365   359   253   150   HighSpeedLINES  (void)
368   362   256   150   HighSpeedLINES  (void)
371   365   259    16   SETPHAS90       CH1   zero
374   368   262   150   HighSpeedLINES  RXOFF
377   371   265   150   HighSpeedLINES  RXOFF
380   374   268   151   EVENT1_TWRD     10.000 usec
382   376   270   150   HighSpeedLINES  RXOFF TXON
385   379   273   151   EVENT1_TWRD     7.000 usec
387   381   275   150   HighSpeedLINES  RXOFF
390   384   278   151   EVENT1_TWRD     10.000 usec
392   386   280   150   HighSpeedLINES  (void)
395   389   283   150   HighSpeedLINES  (void)
```

The remainder of the code is related to the implicit acquisition. The observe phase (`oph`) is set to zero, because this code was generated using the `gf` macro that disables the observe phase cycling (to avoid jumping dc levels in the real-time FID display). All rf channels are then reset to zero phase to avoid a center glitch due to rf leakage. Of course, this is only relevant in real acquisitions with `cp='y'` (i.e., *with* observe phase cycling). After that, the filter delay, `alfa+1/(beta*fb)`, is executed.

The instruction `90` (`SETInputCardMode`) transmits the observe phase to the sum-to-memory board (see Chapter 18, “Acquiring Data,” on page 205). This is done automatically before acquiring the first data point (after which the receiver phase can’t be changed any longer); here, the entire FID is acquired in a single instruction.

```
398   392   286    39   ASSIGNFUNC      zero oph
401   395   289    16   SETPHAS90       CH1 zero
```

```
404   398   292    16   SETPHAS90       CH2 zero
407   401   295   152   EVENT2_TWRD     296 usec + 850 nsec
410   404   298    90   SETInputCardMode
414   408   302    99   ACQXX loop np=37440, dwell 199 usec + 1.000 usec
417   411   305     7   STartFIFO
```

The last instruction before “branching” (jumping) back to the instruction `NextSCan` is for housekeeping. Housekeeping is necessary for the sum-to-memory card to return to its proper state, finishing storing the FID in the RAM. It also increments the `ct` counter (`LC->ct`), decrements block size or steady-state counters (`LC->bsctr` or `LC->ssctr`), checks for the end of the an acquisition block size or for the termination of the steady-state scans, and starts the necessary actions (see also Chapter 18, “Acquiring Data,” on page 205).

```
418   412   306    97   HouseKEEPing
419   413   307    20   BRANCH          Offset 232
```

8.4 Acode Interpretation

The acquisition operating system interprets the Acode instruction section. This process involves a number of different tasks:

- Setting internal status registers.
- Setting or reading status registers on other boards, such as the sum-to-memory board or the HAL.
- Math operations on internal registers (see also Chapter 10, “Phase Calculations,” on page 95).
- Table manipulations (in essence, extraction of single values from tables, see also Chapter 11, “Phase Tables,” on page 115) and the main task.
- Constructing FIFO words that are then suitably “packed” and sent to the pulse programmer through the host I/O bus (see Chapter 7, “Digital Components,” on page 65).

FIFO Flow

The last acquisition operating system task, constructing FIFO words, needs to be coordinated with the pulse programmer; otherwise, the pulse programmer might obtain a few short time events or AP bus words, and execute them, and then run out of FIFO words before the acquisition CPU is able to deliver the continuation of the pulse sequence. With a 0.2 microsecond minimum time event, or 1.15 microseconds (2.15 microseconds for older systems) per AP bus word, the execution of these events can easily be faster than the delivery of information from the acquisition CPU. If the FIFO (see Chapter 9, “Pulse Programmers,” on page 85) runs empty, this generates an error message “FIFO underflow”. On the other hand, if the FIFO is full, the pulse programmer sets a status register that indicates to the acquisition CPU that the FIFO is full, and no more FIFO words are generated.

Initially, the FIFO is empty, but it is stopped (no information is released at the “other” end); this way the acquisition CPU can feed the FIFO words from the initialization part without danger that the FIFO runs empty. Actually, it is exactly the initialization part that consists of only fast (AP bus) events. Only after most of the initialization data has been fed into the pulse programmer and the VT controller has been initialized, the FIFO output is started with Acode instruction `7` (`STartFIFO`, at local offset `206` in

the above example). The following block (preacquisition delay) is surrounded by Acode instruction `9` (`StopFIFO`) and another `STartFIFO`, and also the pulse sequence code itself is surrounded by these function.

In theory, it is possible to have only very few, fast events in the pulse sequence statement (e.g., no delay, a very short pulse, and a very short acquisition at a very large spectral window). In such a case, it must be ensured that the sequence of events during the execution of the pulse sequence is not disrupted by a FIFO underflow.

It can be assumed that during the execution of a pulse sequence, the pulse programmer normally does not run empty or spend extra waiting times in a “stopped” state, because this would also disrupt steady states.

Acode Size Limitations, Acode Buffering

The acquisition CPU has somewhat less than 22 Kbyte of memory space reserved for storing Acodes (including the `LC` and `AUTOD` structures and the instruction segment). That defines the absolute maximum for the Acode size on these systems: about 10900 words in the instruction segment. There hasn’t been an experiment yet that exceeded this limit (but we certainly could construct one, for the sake of the argument), so this should not be a point to be concerned about for pulse sequence programmers.

The topic of Acode size is more complex than that, however. If we were to fill the Acode space in the acquisition CPU to more than 50%, then only one Acode segment can be held in the acquisition CPU at any given time during an experiment. This means, that after finishing one increment of a multi-FID experiment the system would first have to upload the next Acode segment from the host (and this again can only happen when the acquisition CPU is *polled* by the host) before it can continue with the acquisition of the next trace. This leads to inter-increment delays of over one second (the actual length of that delay is unpredictable) and totally disrupts the steady state in arrayed and *n*D experiments (apart from lengthening the overall experiment time, making any calculation/prediction of the experiment duration very unreliable).

If, however, the Acode segment size is less than 50% of the available memory, more than one Acode segment will be stored in the acquisition CPU, and when one increment is finished, the interpretation of the next Acode segment can start immediately (the system will store as many Acode segments in the acquisition CPU as it can, given the current memory limitations). So, to maintain any steady-state across multiple increments, we should not exceed the limit of approximately 5400 words (10800 bytes) per Acode segment[1]. To estimate the size of an Acode segment, enter `go('acqi')` in VNMR and check the size of file `/vnmr/acqqueue/acqi.Code`[2].

A limit of 5400 words still is fairly large for an Acode segment, but it is definitely not impossible to exceed that limit, in particular in some complex sequences with a long, explicitly coded spin locking sequence, or when using either extremely long phase tables and/or trying to use `apshaped_pulse` with complex pulse shapes (over 1000 to 2000 slices).

---

[1] We haven’t confirmed whether 5400 words is the real and accurate limit beyond which the multiple buffering is lost. The pulse sequence generation software has a built-in limit of 11000 words per Acode segment (or about 10900 words in the instruction section).

[2] This size is approximate because `go('acqi')` calculates the Acode only for the first increment (which always has some extra instruction overhead), and also because `go('acqi')` temporarily alters some of the parameters (see also Section 5.3, “Using go('acqi'),” on page 58).

For most cases there is a solution that avoids the problem:

- In tables use the division return factor (see also Chapter 11, “Phase Tables,” on page 115).
- The number of shape slices is definitely limited with `apshaped_pulse` (see also Section 16.5, “What If a Waveform Generator Is Not Available,” on page 189).
- As for DANTE-type pulses, explicitly coded programmed spin locking sequences, etc., it is very inefficient to code such elements using the `rgpulse` statement. `rgpulse` generates several extra `HighSpeedLINES` Acode instructions that can be avoided by using constructs like

  ```
  txphase(phase);
  delay(delta);
  xmtron(); delay(length); xmtroff();
  ```

  which is much more economic in terms of Acode space and execution (see also Section 20.2, “Sideband Suppression in MAS Experiments,” on page 230).
- If you have a waveform generator, use that for modulated spin locking instead of programming it explicitly.

Chapter 9. Pulse Programmers

In earlier chapters, we have shown out how a pulse programmer could be constructed in the context of a Varian spectrometer. Of course, this was a rough sketch, showing only the general working aspect, rather than a detailed picture of its internal functionality. In this chapter, we look at the *real* functionality of this board, which is the most central piece of hardware for the execution of a pulse sequence.

9.1 Layout of the Pulse Programmer

Figure 18 shows a functional diagram of the pulse programmer. The pulse programmer gets its input via the CPU I/O bus from the acquisition CPU (see also Chapter 7, “Digital Components,” on page 65). This information is fed into a FIFO buffer, a memory buffer into which information is fed sequentially and which releases the

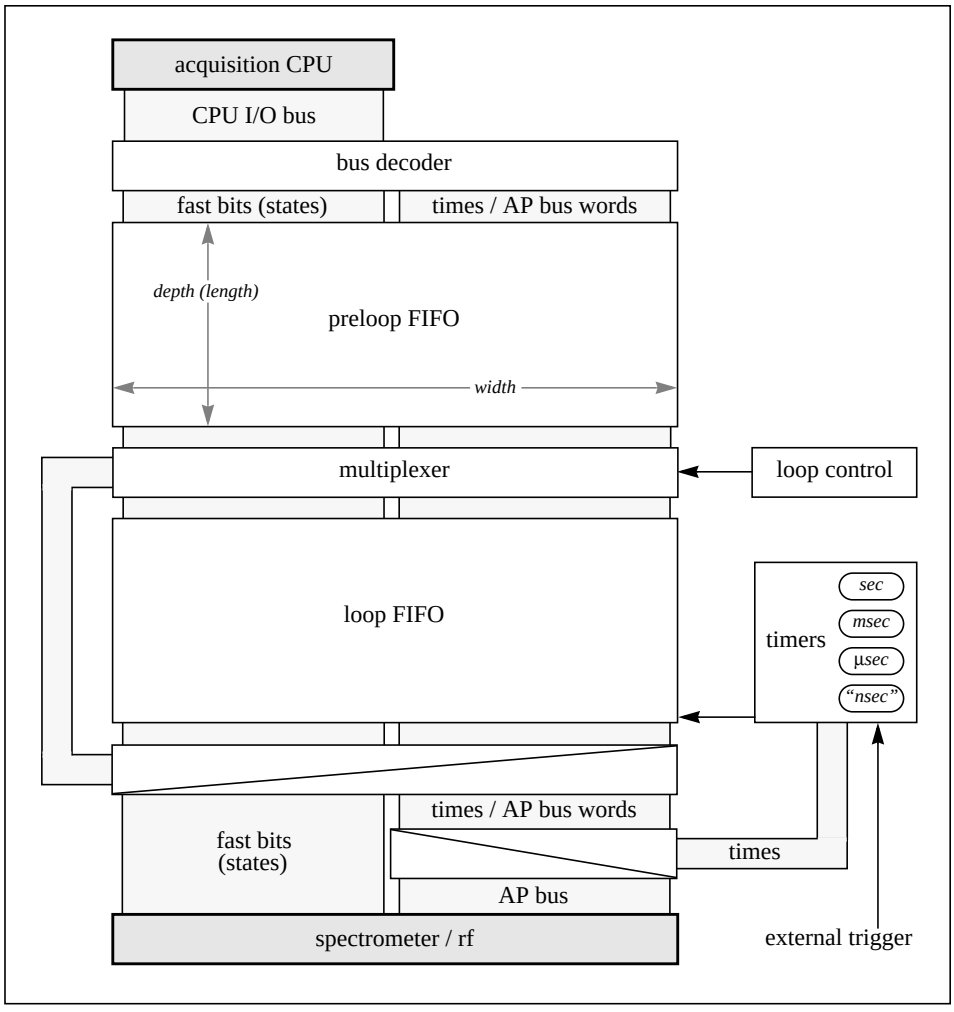


Figure 18. Structure of the pulse programmer

information in the same sequence through the second port[1]. It is a special kind of dual-port memory containing words of a certain width. The number of words in a FIFO buffer is often called the *depth* or the *length* of the FIFO.

Since the width of the CPU I/O bus (16-bits parallel) is not the same as the width of the FIFO (28, 36, or 54 bits), the input logic (bus decoder) composes FIFO words from several I/O bus words.

There are actually two FIFO buffers in sequence. Normally, the information (FIFO words) "falls through" to the end of the second FIFO, or (if there is already information in the buffer) down to the position behind the last word that was fed into the buffer previously—until both buffers fill up. At that point, the pulse programmer signals "FIFO full" to the acquisition CPU, which then temporarily stops producing FIFO words until free space is again available in the FIFO.

The information that comes out at the end of the buffer can be fed back into the second FIFO: a multiplexer (a parallel "Y switch" with two inputs and one output) determines whether the information going into the second FIFO is taken from the first FIFO (the normal case) or from the output. The latter is used for the *hardware looping*, in which the information in the second FIFO (the loop FIFO) is circulating for a predefined number of cycles. At the beginning of the last loop pass, the loop control circuitry (a 16-bit loop counter) switches the input (the multiplexer) back to the preloop FIFO.

A part of every FIFO word is state-related information, often called *fast bits* (because they can be switched instantaneously with any FIFO word. At the output, these bits directly drive rf gates and switches, set phases in 90-degree increments, and blank (or unblank) amplifiers.

From the remaining bits in the FIFO word, 16 are used to define the time; out of these, 12 define the *time count* (1 to 2^{12}; i.e., 1 to 4096 units), the remaining four are used to define the *time base* (seconds, milliseconds, microseconds, "nanoseconds"). The timing information is fed into the timing circuitry, which activates the corresponding timer. When the timer has counted down to zero, a trigger signal is given to the FIFO, causing it to release the next FIFO word. Alternatively, the newer versions of the pulse programmer board can disable the timers for specific FIFO words and cause the following FIFO word to be released upon sensing an external trigger signal. This allows synchronizing a pulse sequence with external events (such as a rotor period for solids experiments, or the heart beat or a respiration cycle for imaging experiments).

Because there are only four different time bases, only two bits are needed to select the time base. The four control bits at the same time can *redefine* the timing part of a FIFO word to be information that is fed into the AP bus by which 16 devices can be addressed directly (i.e., a part of the 16 timing bits serves as address). The rest is numeric information that is transferred to the addressed device (e.g., the small-angle phase shifters).

Alternatively, the AP bus can be used in indirect mode, in which the address is first transmitted in a separate AP word, followed by AP words containing the numeric information. This way, an unlimited number of devices (several thousand) can be addressed, and numeric information of arbitrary precision can be transmitted (e.g.,

[1] This differs greatly from the way information is stored and recalled randomly in RAM. In a FIFO, no addressing is involved: only one word at a time can be read out, and the read-out sequence is given by the order of the input.

frequency information for the PTS synthesizer). During AP bus traffic, the timing circuitry is still activated, ensuring that every AP bus word remains on the bus for a specific, well-defined time (1.15 microseconds for UNITY*plus* systems, 2.15 microseconds on earlier systems), such that there is sufficient time for the addressed devices to decode the address and read the information off the bus.

The last four bits in the FIFO word include the "command to convert" (CTC) bit that triggers the ADC. Unlike all other bits, this bit is not held up during an entire time event, but is electronically reformed to a short trigger pulse. The remaining three bits are used for loop and FIFO control.

All Varian pulse programmers have essentially the same layout—be it the output board used in XLs and VXRs, the acquisition control board used in VXRs and UNITY spectrometers, the pulse sequence control board used in UNITY*plus* spectrometers, or similar boards in other Varian spectrometers. With few exceptions, the differences are only quantitative: both the depth and the width of the FIFO has changed over the years (adding more fast bits), the timing resolution has been improved, and the external trigger has been added.

In summary, the various pulse programmers can be characterized as shown in Table 1. The differences can be summarized as follows:

- The FIFO width varies from 28 bits (Gemini) and 36 bits (XL, VXR, and UNITY) to 54 bits (UNITY*plus*); the number of fast bits in essence is equal to the total number of bits per FIFO word minus 20.
- The length of the loop FIFO has been increased from 63 words (Gemini, XL, and VXR) to 1024 words (VXR-S and UNITY) and 2048 words (UNITY*plus*).
- The preloop FIFO does not exist on the Gemini output board. It was half the size of the loop FIFO in the output board and is of the same depth as the loop FIFO in newer systems.
- On the acquisition control board, the timing resolution has been improved to 25 nanoseconds (100 nanoseconds in earlier boards).
- In the UNITY*plus*, the AP bus has been speeded up by almost a factor of two (see Chapter 12, "AP Bus Traffic," on page 137).

Table 1. Pulse programmer characteristics

	Output Board (GEMINI)	*Output Board*	*Acquisition Control Board*	*Pulse Sequence Control Board*
Use	Gemini	XL, VXR, early VXR-S	late VXR-S, UNITY	UNITY*plus*
FIFO width (fast bits)	28 (8) bits	36 (16) bits	36 (16) bits	54 (34) bits
Pre-loop FIFO	none	32 words	1024 words	2048 words
Loop FIFO	63 words	63 words	1024 words	2048 words
Timing Resolution	100 nsec	100 nsec	25 nsec	25 nsec
AP Bus Speed	2.15 µsec/word	2.15 µsec/word	2.15 µsec/word	1.15 µsec/word
External Trigger	no	no	yes	yes

- The external trigger only became available with the acquisition control board.

Fast bits are covered in the next section, and timing characteristics are considered in more detail at the end of this chapter. The consequences of various preloop and loop FIFO lengths are discussed more in Section 14.3, “Hardware Loops,” on page 150.

9.2 Fast Bits

Fast bits have been provided for those states that possibly need to be changed with every time event, even as short as 0.2 microseconds, such as transmitter gates, 90-degree phase shifts, receiver gates, and amplifier blanking, plus eventually external devices, such as a laser for CIDNP experiments; also waveform generators need to be triggered with a fast bit.

The demand for fast bits has been growing over the years. The Gemini had only very modest needs in terms of fast bits (using only two rf channels), a UNITY*plus* spectrometer with up to six rf channels is much more demanding in terms of the number of fast-switching lines. For a long time, systems (all except Gemini and UNITY*plus*) were equipped with a fixed number of 16 fast bits—with the consequence that in systems with third rf channel and waveform generators (particularly UNITY systems) some of the fast bits had to be reassigned by shifting some of the fast bit functionality (the decoupler modulation mode `dmm` in particular) to the AP bus. This cut-over happened with the transition from AP interface board type 2 to type 3 of the same board (only type 3 permits controlling `dmm` via the AP bus).

Table 2 lists the assignment for the fast bits in pulse programmers with 16 fast bits. The hexadecimal number in the first row describes the numeric value of the corresponding

Table 2. Fast bit assignments, output boards and acquisition control boards

HEX Value	*AP Interface Type 2*	*AP Interface Type 3*
0x1	`VAR1` (Varian-reserved line 1)	`WFG1` (fast line for WFG #1)
0x2	`VAR2` (Varian-reserved line 2)	`WFG2` (fast line for WFG #2)
0x4	`SP1` (spare bit 1)	`SP1` (spare bit 1)
0x8	`SP2` (spare bit 2)	`SP2` (spare bit 2)
0x10	`DECLVL` (decoupler level switching)	`DECUPLR2` (decoupler 2, gate)
0x20	`MODMA` (`dmm`, bit 1)	`DC2_90` (decoupler 2, 90 deg. phase)
0x40	`MODMB` (`dmm`, bit 2)	`DC2_180` (decoupler 2, 180 deg. phase)
0x80	`HomoSpoilON` (homospoil gate)	`HomoSpoilON` (homospoil gate)
0x100	`DECPP` (homodecoupler gating)	`WFG3` (fast line for WFG #3)
0x200	`DC90` (decoupler, 90 deg. phase)	`DC90` (decoupler, 90 deg. phase)
0x400	`DC180` (decoupler, 180 deg. phase)	`DC180` (decoupler, 180 deg. phase)
0x800	`DECUPLR` (decoupler gate)	`DECUPLR` (decoupler gate)
0x1000	`RFP90` (observe xmtr. 90 deg. phase)	`RFP90` (observe xmtr. 90 deg. phase)
0x2000	`RFP180` (observe xmtr. 180 deg. phase)	`RFP180` (observe xmtr. 180 deg. phase)
0x4000	`TXON` (observer xmtr. gate)	`TXON` (observer xmtr. gate)
0x8000	`RXOFF` (receiver off gate)	`RXOFF` (receiver off gate)

bit within the 16-bit “fast-bit word” or within the “quiescent states” word in the `LC` structure. This information is taken from the header file `rfconst.h` found in `/vnmr/psg`. The CTC (command-to-convert) bit does not count as a regular fast bit, because it behaves differently (see above).

Evidently, it is a bad idea to try addressing the ambiguous bits (`DECLVL/DECUPLR2`, `MODMA/DC2_90`, `MODMB/DC2_180`, and `DECPP/WFG3`) directly (using low-level statements taken from `/vnmr/psg`), because this would generate code that is only correct for one particular type of hardware and could create havoc when executed on a different system. Actually, *no* fast bit should be addressed directly (in the UNITY*plus*, the assignments changed altogether, and in fact, the `gate` statement for manipulating fast bits directly no longer exists in VNMR 4.3 or later.)

The pulse sequence control board in the UNITY*plus* has a totally different fast bit assignment, as can be seen from Table 3. The fast bits are organized in groups of five bits per rf channel; each channel has its own “receiver gate” (the amplifier blanking), a transmitter gate, a gate for the waveform generator, and two gates for the 90-degree phase shifts. The observe is switched together with the blanking line for the observe channel amplifier. Four additional bits control the homospoil pulse, the rotor synchronization hardware, and the two spare gating lines (for triggering external devices). The latter two are no longer part of `LC->squi` (the 32-bit “quiescent states” word in the LC structure), but are handled separately.

9.3 Timers and Timer Words

All pulse programmers use four different timers, and the time count (the number of time units to be counted down by the timer) is a 12-bit binary number, resulting in values between 1 and 4096. The four timers include:

- Seconds timer (1 to 4096 seconds)
- Milliseconds timer (1 to 4096 milliseconds)
- Microseconds timer (1 to 4096 microseconds)
- “Nanosecond” timer, which is in units of 100 nanoseconds for output boards and on the Gemini (resulting in a theoretical range of 0.1 to 409.6 microseconds), or in units of 25 nanoseconds for acquisition control boards and pulse sequence control boards (resulting in a theoretical range of 0.025 to 102.4 microseconds).

The minimum timer word is 0.2 microseconds on all pulse programmers. Timer words shorter than that cannot be executed and are suppressed already at C level in the pulse sequence software (they would result in a pulse programmer error message).

A problem seems to exist with the timers listed above, in that it seems that time events above 4 seconds can only be executed in steps of 1 second, time events between 4 milliseconds and 4 seconds can be executed with a precision of 1 milliseconds, and time events between 102 microseconds (409 microseconds on older boards) and 4 milliseconds can be performed with a precision of 1 microsecond only. This seems rather limited and would in fact have a severe impact on multidimensional experiments, because most likely the 2D increments could probably not be spaced properly due to the “granularity” of the duration of time events. The same limitation would apply to the 1D dwell time, which would impose severe restrictions to the settability of the `sw` parameter. All this is clearly unacceptable.

Table 3. Fast-bit assignments on pulse sequence control boards

HEX Value	*RF channel*	*Function*
0x1	channel 1	amplifier blanking / receiver gating
0x2	channel 1	transmitter gate
0x4	channel 1	waveform generator gate
0x8	channel 1	90 degrees phase shift
0x10	channel 1	180 degrees phase shift
0x20	channel 2	amplifier blanking (“receiver”)
0x40	channel 2	transmitter gate
0x80	channel 2	waveform generator gate
0x100	channel 2	90 degrees phase shift
0x200	channel 2	180 degrees phase shift
0x400	channel 3	amplifier blanking (“receiver”)
0x800	channel 3	transmitter gate
0x1000	channel 3	waveform generator gate
0x2000	channel 3	90 degrees phase shift
0x4000	channel 3	180 degrees phase shift
0x8000	channel 4	amplifier blanking (“receiver”)
0x10000	channel 4	transmitter gate
0x20000	channel 4	waveform generator gate
0x40000	channel 4	90 degrees phase shift
0x80000	channel 4	180 degrees phase shift
0x100000	channel 5	amplifier blanking (“receiver”)
0x200000	channel 5	transmitter gate
0x400000	channel 5	waveform generator gate
0x800000	channel 5	90 degrees phase shift
0x1000000	channel 5	180 degrees phase shift
0x2000000	channel 6	amplifier blanking (“receiver”)
0x4000000	channel 6	transmitter gate
0x8000000	channel 6	waveform generator gate
0x10000000	channel 6	90 degrees phase shift
0x20000000	channel 6	180 degrees phase shift
0x40000000		homospoil gate
0x80000000		rotor synchronization
[0x10]		spare gate 1
[0x20]		spare gate 2

The solution is to perform *double-precision timer words*, where necessary. If there is a remainder after performing the major part of a time event in one unit (e.g., seconds for time events above 4.096 seconds), a *second* time event is included that performs the remainder of the time event in the next smaller unit. Single-precision timer words are used only if there is no remainder from a single time unit. With these two options for time events, we get the following possibilities:

- Time events above 4100 seconds are performed as a double timer word with the seconds timer; the maximum time event is 8192 seconds. The round-off error (up to 0.5 seconds) is less than 0.012% for this range of durations.
- Time events above 4.1 seconds can be performed with millisecond precision, with a round-off error of less than 0.012% (up to 0.5 milliseconds absolute).
- Time events between 4.2 milliseconds (4.5 milliseconds on boards with 100 nanoseconds resolution) and 4.1 seconds are performed with microsecond precision, the maximum round-off error being again below 0.012% (up to 0.5 microseconds absolute).
- Time events smaller than 4.2 milliseconds (4.5 milliseconds on boards with 100 nanoseconds resolution) are performed with the timing resolution of the pulse programmer (100 or 25 nanoseconds). By nature, the round-off error can be considerable in this range: for the output board it is up to 50 nanoseconds absolute, or 12 ppm at the upper end, and 25% at the lower end (0.2 microseconds). With newer boards, it is up to 12.5 nanoseconds in absolute terms, or up to 3 ppm at the upper limit and below 6% for the shortest time intervals.

The reality is somewhat more complicated. On all pulse programmers it takes 150 nanoseconds for the hardware to decode the time base, store the time count in the time counter, and start the corresponding timer. Consequently, there is a 150 nanoseconds dead-time involved with every timer word (300 nanoseconds with double-precision time events). For small time events, the software corrects for this error by adjusting the nanosecond count, for larger durations this error remains uncorrected, but is negligible (0.007% maximum).

One more thing was neglected up to now: the numeric range of a 12-bit binary number is actually from 0 to 4095, but the timers, on the other hand, always perform at least one time count to begin with (0 input results in a count of 1). Thus, the 12-bit input range translates to a count range of 1 to 4096 units.

Table 4 summarizes the various possibilities with single- and double-precision time events. Single precision (except for those with the smallest units) is, of course, only performed if there is no remainder. The somewhat “odd” lower limits are due to the fact that smaller delays will be performed with the next smaller time base (and because there is some overlap in the ranges of the individual time bases). Whenever the “nanoseconds” time base is involved on boards with 25-nanosecond resolution, 6 or 12 counts are subtracted from the “nanoseconds” part for single- and double-precision time events, respectively; these time events will be accurate to 25 nanoseconds.

On the output board (63-word loop FIFO), short time events have slightly different characteristics, because the timing resolution is 100 nanoseconds only and timing errors are not corrected for this hardware. Table 5 covers output boards.

It is not necessary to fully understand all of the tables shown here, but this information should help in understanding how certain events are translated into single- or double-precision timer words. The question whether double-precision timer words are used or not isn’t relevant for most cases either, but may play an important role in the execution

of hardware loops, especially on systems with 63-word loop FIFO (see also Section 14.3, “Hardware Loops,” on page 150).

9.4 Problems with Timer Word Errors

Compared with the total duration of a timer word, the possible timing errors seem small and, overall, the timer word precision seems more than sufficient for the common NMR experiments. There is one important exception, however: in *n*D experiments the evolution delay is performed as a single time event that is calculated for every increment (see also Section 19.1, “Indirect Time Domain Incrementation,” on page

Table 4. Single- and double-precision timer word characteristics

Type	*Time Base(s)*	*[Counts (min/max)]* *Duration range*	*Resolution*	*Timing Error*
double	sec / sec	[4096 + 1 up to 4096 + 4096] *4097 sec - 8192 sec*	1 sec	300 nsec longer
single	sec	*5 sec - 4096 sec*	1 sec	150 nsec longer
double	sec / msec	[4 + 97 up to 4096 + 4096] *4.097 sec - 4100.096 sec*	1 msec	300 nsec longer
single	msec	*5 msec - 4096 msec*	1 msec	150 nsec longer
double	msec / µsec	[4 + 97 up to 4096 + 4096] *4.097 msec - 4.100096 sec*	1 µsec	300 nsec longer
single	µsec	a “nsec” time event is always added to correct for dead times		
double	µsec / 25 nsec	[102 + 11 (+ 12) / 4096 + 4096 (+ 12)] *102.575 µsec - 4198.7 µsec*	0.025 µsec	accurate
single	25 nsec	[2 (+ 6) - 4096 (+ 6)] *0.2 µsec - 102.55 µsec*	0.025 µsec	accurate

Table 5. Single- and double-precision timer word characteristics, output boards

Type	*Time Base(s)*	*[Counts (min/max)]* *Duration range*	*Resolution*	*Timing Error*
double	sec / sec	[4096 + 1 up to 4096 + 4096] *4097 sec - 8192 sec*	1 sec	300 nsec longer
single	sec	*5 sec - 4096 sec*	1 sec	150 nsec longer
double	sec / msec	[4 + 97 up to 4096 + 4096] *4.097 sec - 4100.096 sec*	1 msec	300 nsec longer
single	msec	*5 msec - 4096 msec*	1 msec	150 nsec longer
double	msec / µsec	[4 + 97 up to 4096 + 4096] *4.097 msec - 4.100096 sec*	1 µsec	300 nsec longer
single	µsec	*411 µsec - 4096 µsec*	1 µsec	150 nsec longer
double	µsec / 100 nsec	[409 + 7 up to 4096 + 4096] *409.7 µsec - 4505.6 µsec*	0.1 µsec	300 nsec longer
single	100 nsec	[2 up to 4096] *0.2 µsec - 409.6 µsec*	0.1 µsec	150 nsec longer

215). What counts in the end is the *difference* between the increments, which should be as equal as the dwell times during the standard acquisition. Any random variation in the "evolution dwell time" will translate to noise in the indirect dimension of the final spectrum and, much worse, any periodic variation will lead to extra signals (sidebands) in the indirect domain.

What is the nature of these timing errors in *n*D experiments? Typically (with spectral windows of a few kHz) the evolution time for first increments on an *n*D experiment is below 4 msec (i.e., it is performed accurately with the full timing precision of the pulse programmer, 25 nsec on most systems). But for those increments that have an evolution time *greater than 4 msec* (see the above tables) two errors occur:

- The delays are rounded off to full µsec time events. This will lead to a periodic error in the course of the evolution time incrementation.
- The timer word overhead is no longer corrected for (i.e., evolution times larger than 4.097 msec are 300 nsec too long, or 150 nsec for single-precision millisecond time events).

We could set the spectral window in the indirect dimensions such that the evolution time is always a multiple of a full µsec (which certainly would be inconvenient), but then still there would be a single incrementation discontinuity when crossing the 4.097 msec limit, beyond which the timing overhead is not corrected. This alone probably wouldn't be too bad, but worse is that when the evolution time happens to "fall onto" a full msec duration, which will occur periodically, the timing overhead would only be half of what it is for the other time events above 4.097 msec. This can translate to visible artifacts (sidebands) in nD spectra with a large dynamic range.

To correct for these problems, as of VNMR 5.1 the module `/vnmr/psg/delay.c` has been modified such that any time event above 4 msec and below 4 sec is performed in *three timer words*:

- a (single) msec timer word, with the msec portion of the duration *minus 2 msec*.
- a double (usec/nsec) timer word of 2 msec, *plus* the sub-msec fraction, *minus the timing overhead of the first timer word*.

This change has the effect of producing triple-precision timer words for durations between 4 msec and 4 sec, and with this *all time events up to 4 sec are performed at the full precision of the pulse programmer* (25 nsec on most systems). These "triple-precision time events" will, of course, lead to extra FIFO words in the pulse programmer; however, this only affects delays above 4 msec, and hence should not lead to problems such as having too many events in a hardloop, etc.

9.5 Timer Words and Fast Bits in the Acode

For acquisition control boards and pulse sequence control boards, time events are encoded with the instructions `151` and `152`, `EVENT1_TWRD` and `EVENT2_TWRD`. `EVENT1_TWRD` instructions are followed by a 16-bit timer word (the time base is in the 4 most-significant bits), `EVENT2_TWRD` are followed by two 16-bit timer words.

Fast bits are set by separate instructions: gate-related information (excluding the 90-degree phase shifts) is set by instruction `150` (`HighSpeedLINES`, followed by a 32-bit pattern), 90-degree phase shifts are set with the instruction `16` (`SETPHAS90`, followed by a channel identifier and the address of a location in the `LC` structure):

```
362  356  250  150   HighSpeedLINES   (void)
365  359  253  150   HighSpeedLINES   (void)
368  362  256  150   HighSpeedLINES   (void)
371  365  259   16   SETPHAS90        CH1   zero
374  368  262  150   HighSpeedLINES   RXOFF
377  371  265  150   HighSpeedLINES   RXOFF
380  374  268  151   EVENT1_TWRD      10.000 usec
382  376  270  150   HighSpeedLINES   RXOFF TXON
385  379  273  151   EVENT1_TWRD      7.000 usec
387  381  275  150   HighSpeedLINES   RXOFF
390  384  278  151   EVENT1_TWRD      10.000 usec
392  386  280  150   HighSpeedLINES   (void)
```

Any pulse sequence statement that can possibly alter the fast bits (like `status` or the various gating instructions within the `G_pulse` statement, even if the pulse duration is zero) produces a `HighSpeedLINES` instruction in the Acode. Because many time events in a pulse sequence may be set to zero (like in the above example using the `s2pul` pulse sequence), it is not unusual to find a series of `HighSpeedLINES` with identical arguments and no time events in-between. Of course, only instructions `151` and `152` (`EVENT1_TWRD` and `EVENT2_TWRD`) actually produce FIFO words (1 and 2, respectively); the `HighSpeedLINES` and `SETPHAS90` only change specific bits in the "quiescent states" register `LC->squi`.

For pulse programmers with 16 fast bits (output boards and acquisition control boards), the two spare lines (set and unset in the pulse sequence code by the statements `sp1on()`, `sp1off()`, `sp2on()`, and `sp2off()`) are part of the standard fast bit pattern and are set the same as the other gating bits, through the `HighSpeedLINES` instruction. For the pulse sequence control board, the two spare lines are set with a special instruction `156` (`SPARE12`), followed by a 16-bit word containing the numeric equivalent to the two fast bits (0x10 for spare line 1, 0x20 for the spare line 2), because these two bits are not part of the standard 32-bit fast bit word `LC->squi`.

Earlier software releases (prior to UNITY*plus* and the pulse sequence controller board) did not have the Acode instructions `150` to `156`; at that time, the high-speed lines (including the two spare lines) were set together with the time event instructions `EVENT1` and `EVENT2` (`46` and `47`). These instructions were followed by one or two timer words, plus an extra 16-bit word with the fast bits. A `HighSpeedLINES` instruction was not required; only the 90-degree phase shifts were set by the same instruction `SETPHAS90` (`16`) as today.

Chapter 10. Phase Calculations

Phase cycling in pulsed NMR experiments is the alteration of rf phases as a function of the scan number, thereby co-adding experiments with different rf phases, by altering the phases in a pulse sequence as a function of the transient counter variable `ct`. There are many reasons to do phase cycling, including:

- Coherence pathway selection, such as multiple-quantum filtering, f_1 or f_2 quadrature selection, etc.
- Cancellation of artifacts arising from spectrometer imperfections, such as channel imbalance (quadrature images), dc offsets (center glitch), phase errors, pulse imperfections (off-resonance effects).
- Cancellation of phase errors due to J-coupling.
- Cancellation of phase errors due to (sometimes deliberately) improper refocusing.

10.1 How Do Phase Calculations Work?

The most natural thing seems to be to calculate the phases directly from the `ct` counter via some mathematical algorithm. Of course, this presumes that such an algorithm exists: we need a mathematical prescription that generates a given vector (a one-dimensional array of scalar numbers; for 90-degree phase shifting typically integer numbers between 0 and 3) from the vector of natural numbers (i.e., the `ct` counter). It can be shown that for any *repetitive* sequence of numbers, such an algorithm exists.

Phase cycles by nature are repetitive; therefore, the calculation is feasible, supposing we have the necessary mathematical tools. The complexity of the mathematical procedure (the number of mathematical operations and the storage requirements for intermediate results) heavily depends on the (repetitive) sequence of numbers (the target vector) that is to be created; the means that VNMR provides should actually be sufficient for creating *any* phase cycle used in NMR experiments.

True random sequences of numbers cannot be generated this way, but pseudo-random sequences of sufficient length and quality can be generated using integer math (using modulo functions, see Section 10.8, “Real-Time Random Numbers,” on page 112).

The Tools

The basis for phase mathematics is formed by two main ingredients: *operators* (the math operations themselves) and *operands* (the objects—numbers and storage locations—that are used and manipulated by the operators). A third ingredient, *real-time logical decisions*, is sometimes also used for the construction of more complex phase cycles. Also, *C constructs* are often used to make phase calculations dependent on certain VNMR parameters like `phase`.

C constructs can only happen on a *per-FID* basis, because after the Acode generation C constructs can have no effect on the execution of real-time math. Also, C math operators cannot be used for real-time math: phase math operators are interpreted by the acquisition CPU in “real time”; operands are addresses (offsets in the `LC` structure), on which the real-time math operations are performed.

Real-Time Operands

The operands for real-time math can be divided into constants and variables:

- Simple numeric constants: `zero`, `one`. `two`, `three` (`LC` elements with numeric values 0, 1, 2 and 3, respectively).
- Constants `ssval` and `bsval`, holding the number of steady-state and blocksize transients.
- Predefined variables `ct`, `ssctr`, and `bsctr`, holding the transient counter (least-significant half only, see "LC Data Structure" on page 75), the steady-state transient counter, and the block size transient counter, respectively.
- True real-time variables `v1`, `v2`, . . . `v14`.

It obviously doesn't make sense to change the values of the numeric constants and, with very few exceptions, to modify other constants and predefined variables. Therefore, results are normally placed only in the true real-time variables `v1` to `v14`.

Real-Time Operators

A full set of basic math operators exists for the construction of phase cycles. These include unary operators (one argument only), and operators with two and three arguments. Always the last argument is the *target operand* and that contains the result of the calculation (it is therefore *modified* by the operator), preceding operands are not modified by the calculation. Table 6 shows the set of real-time math operators (`ph1`, `ph2`, etc. are real-time operands, see above.

With all these operators, the last argument is the target operand and is modified by the real-time calculation. Math results can, of course, be used in subsequent calculations. Complex algorithms are built by chaining a series of simple operations.

All calculations are integer operations. Divisions always result in an integer, and fractional numbers are truncated to the next lower integer number. All operations are

Table 6. Real-time math operators

Type	*Syntax*	*Meaning*
assignment	`assign(ph1,ph2);`	`ph2 = ph1`
increment / decrement	`incr(ph1);` `decr(ph1);`	`ph1 = ph1 + 1` `ph1 = ph1 - 1`
addition / subtraction	`add(ph1,ph2,ph3);` `sub(ph1,ph2,ph3);`	`ph3 = ph1 + ph2` `ph3 = ph1 - ph2`
multiplication	`dbl(ph1,ph2);` `mult(ph1,ph2,ph3);`	`ph2 = 2 * ph1` `ph3 = ph1 * ph2`
division	`hlv(ph1,ph2);` `divn(ph1,ph2,ph3);`	`ph2 = ph1 / 2` `ph3 = ph1 / ph2`

modulo	`mod2(ph1,ph2);`
	`mod4(ph1,ph2);`
	`modn(ph1,ph2,ph3);`
		`ph2 = ph1 % 2`
		`ph2 = ph1 % 4`
		`ph3 = ph1 % ph2`

performed in 16-bit integer format with a range of –32768 up to 32767. Integer overflow is suppressed—results are subjected to an implicit (mod 32768) operation.

Variable operands can contain positive as well as negative numbers. In most of the typical applications for phase calculations (e.g., the calculation of 90-degree or small-angle phase shift steps, the use of a real-time variable is as a table index) the function for which the result is used has a very limited numeric range, such as 0 to 3 for 90-degree phase shifts, or 0 to 7 for 45-degree phase steps. In these cases, the value in the variable undergoes an implicit modulo function (without altering the variable itself): for 90-degree phase shifts, the two least-significant bits are extracted (-1 mod 4 is 3), for 45-degree phase steps, the three least-significant bits are taken (-1 mod 8 is 7). In other words, as long as the number of possible (phase) values is a power of two (2, 4, 8, ...), a negative number *n* in phase cycling behaves like 32768 – |*n*|.

As mentioned above, in most cases the result of a real-time calculation is subject to some modulo function in the end (either implicitly or explicitly). For instance, with 90-degree phase shifting it normally does not matter whether a variable contains the numbers 3, 7, 11, or –1. In fact, many people tend to think that in such situations “4 is the same as 0” and mentally apply a modulo function to every intermediate result in a phase calculation. This thinking is certainly not correct, but it still leads to the proper result, *as long as no division (*`divn`, `hlv`*) is involved.*

New Real-Time Numeric Constants

The real-time numeric constants `zero`, `one`, `two`, and `three` exist for calculations. In principle, three of those could be calculated from a single value, for example:

```
assign(zero,v1);        /* 0 */
incr(v1);               /* 1 */
dbl(v1,v2);             /* 2 */
add(v1,v2,v3);          /* 3 */
```

In other words, three of the four constants are not really a necessity, but are a pure convenience for the programmer (actually, we save some Acode space because saving three more constants costs three 16-bit words, while the above calculations take up 12 AP words). Sometimes we would like to make calculations with much higher numeric values, the value 127 perhaps (for an example see Section 10.8, “Real-Time Random Numbers,” on page 112). Again, such numbers can be calculated:

```
dbl(two,v1);            /* 4 */
dbl(v1,v1);             /* 8 */
mult(v1,v1,v1);         /* 64 */
dbl(v1,v1);             /* 128 */
decr(v1);               /* 127 */
```

On the other hand, it would seem convenient if we could just create a new numeric constant. This is in fact possible, using the `initval` statement:

```
initval(127.0,v1);
```

This initializes `v1` (`LC->v1`) with the numeric value 127 (instead of 0). This way we haven’t wasted a single word of Acode! However, there is a big *caveat* in this method: `v1` in fact contains the value 127 when the above statement is used, but this happens only once: upon typing `go`, when the Acode is generated (*better*: when the `LC` structure is created and initialized). If *any* real-time calculation overwrites the value of `v1` (i.e., if `v1` is used as target operand), the initial value is lost for the current transient and any

subsequent transients for that FID! The consequence is that the variable that is initialized using `initval` is lost as a real-time variable for that pulse sequence.

There is also a potential danger that the programmer uses `initval` at the beginning of a pulse sequence and then by mistake overwrites the value. This leads to very subtle pulse sequence errors that can be hard to find and debug (certainly, the syntax checker does not find such errors!).

The conclusion is that `initval` allows simplifying the pulse sequence code, but in general it is probably better to avoid using it, thus saving real-time variables and avoiding pulse sequence errors. We should be careful when using `initval` to store power levels in real-time variables (this practice was common in Varian pulse sequences for a long time). If by mistake power levels are altered by real-time calculations, excessive rf load and heating may damage the probe and precious samples. It is better to avoid setting power levels through real-time variables (see also Chapter 12, “AP Bus Traffic,” on page 137).

The fact that `initval` happens at `go` time and not at real time means that it doesn’t matter where the `initval` statement is placed. It can be the last statement in the pulse sequence, but the value can be used in the first real-time calculation.

Phase Calculations in the Acode

Phase calculations are the part of the pulse sequence program that is transferred into Acode almost directly. Every statement creates one Acode instruction, and every real-time math argument translates to an Acode word (`LC` address pointer). The following example is generated from a slightly modified version of the standard `inadqt.c` pulse sequence in VNMR 4.3:

```
 384  378  272   37   MOD4FUNC          ct    v3
 387  381  275   34   HLVFUNC           ct    v9
 390  384  278   34   HLVFUNC           v9    v9
 393  387  281   39   ASSIGNFUNC      zero   v10
 396  390  284   39   ASSIGNFUNC        v9    v1
 399  393  287   34   HLVFUNC           v9    v9
 402  396  290   39   ASSIGNFUNC        v9    v2
 405  399  293   34   HLVFUNC           v9    v9
 408  402  296   34   HLVFUNC           v9    v9
 411  405  299   36   MOD2FUNC          v9    v9
 414  408  302   33   DBLFUNC           v1    v1
 417  411  305   29   ADDFUNC           v9    v1    v1
 421  415  309   39   ASSIGNFUNC        v1   oph
 424  418  312   33   DBLFUNC           v2    v8
 427  421  315   29   ADDFUNC           v9    v2    v2
 431  425  319   29   ADDFUNC           v8   oph   oph
 435  429  323   33   DBLFUNC           v3    v4
 438  432  326   29   ADDFUNC           v3    v4    v4
 442  436  330   29   ADDFUNC           v3    v9    v3
 446  440  334   29   ADDFUNC           v4   oph   oph
 450  444  338   29   ADDFUNC          v10   oph   oph
```

Considering the fact that this code generates several phase cycles that are 64 steps long, the above code can be regarded as being quite efficient: only 70 Acode words (21 instructions) are used to generate these phase cycles (and this isn’t even the shortest possible coding!).

10.2 Case 1: Decoding Phase Calculations

Anyone programming sequences with real-time math has two hurdles to overcome in connection with the real-time math: to understand how a sequence with real-time math works, and then to find an appropriate real-time math algorithm that generates the desired phase cycle. The understanding part is the easier of the two tasks and is therefore discussed first.

If we take the above example, we might find a phase cycling (calculation) section in the pulse sequence that looks as follows (assuming that there are no comments):

```
mod4(ct,v3);
hlv(ct,v9); hlv(v9,v9);
assign(v9,v1);
hlv(v9,v9); assign(v9,v2);
hlv(v9,v9); hlv(v9,v9); mod2(v9,v9);
dbl(v1,v1); add(v9,v1,v1);
assign(v1,oph);
dbl(v2,v8);
add(v9,v2,v2);
add(v8,oph,oph);
dbl(v3,v4); add(v3,v4,v4);
add(v3,v9,v3);
add(v4,oph,oph);
```

The only safe way (apart from using a computer program) to evaluate the resulting phase cycles is to write down the result of every single calculation step. For the beginner, an easy method is to write the vectors in columns (at least for shorter and less complex phase cycles):

	mod4 (ct,v3)	hlv (ct,v9)	hlv (v9,v9)	assign (v9,v1)	hlv (v9,v9)	assign (v9,v2)	hlv (v9,v9)
ct	v3	v9	v9	v1	v9	v2	v9
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	2	1	0	0	0	0	0
3	3	1	0	0	0	0	0
4	0	2	1	1	0	0	0
5	1	2	1	1	0	0	0
6	2	3	1	1	0	0	0
7	3	3	1	1	0	0	0
8	0	4	2	2	1	1	0
9	1	4	2	2	1	1	0
10	2	5	2	2	1	1	0
11	3	5	2	2	1	1	0
12	0	6	3	3	1	1	0
13	1	6	3	3	1	1	0
14	2	7	3	3	1	1	0
15	3	7	3	3	1	1	0
16	0	8	4	4	2	2	1
17	1	8	4	4	2	2	1
18	2	9	4	4	2	2	1

	mod4 (ct,v3)	hlv (ct,v9)	hlv (v9,v9)	assign (v9,v1)	hlv (v9,v9)	assign (v9,v2)	hlv (v9,v9)
ct	v3	v9	v9	v1	v9	v2	v9
19	3	9	4	4	2	2	1
20	0	10	5	5	2	2	1
21	1	10	5	5	2	2	1
22	2	11	5	5	2	2	1
23	3	11	5	5	2	2	1
24	0	12	6	6	3	3	1
25	1	12	6	6	3	3	1
26	2	13	6	6	3	3	1
27	3	13	6	6	3	3	1
28	0	14	7	7	3	3	1
29	1	14	7	7	3	3	1
30	2	15	7	7	3	3	1
31	3	15	7	7	3	3	1
32	0	16	8	8	4	4	2
33	1	16	8	8	4	4	2
34	1	17	8	8	4	4	2
35	2	17	8	8	4	4	2

For more complex phase cycles, the vectors are better written in rows instead of columns, either on a piece of paper or, even better, directly in the pulse sequence as a comment behind every single function call:

```
mod4(ct,v3);      /* v3  = 01230123 */
hlv(ct,v9);       /* v9  = 001122334455 ... */
hlv(v9,v9);       /* v9  = 000011112222333344 ... = ct/4 */
assign(v9,v1);    /* v1  = ct/4 */
hlv(v9,v9);       /* v9  = ct/8 */
assign(v9,v2);    /* v2  = ct/8 */
hlv(v9,v9);       /* v9  = ct/16 */
hlv(v9,v9);       /* v9  = ct/32 */
mod2(v9,v9);      /* v9  = (32*0) (32*1) */
dbl(v1,v1);       /* v1  = 0000222244446666 ... */
                  /*     = 0000222200002222 (quadrature phases) */
add(v9,v1,v1);    /* v1  = 00002222000022220000222200002222
                         11113333111133331111333311113333 */
assign(v1,oph);   /* oph = v1 */
dbl(v2,v8);       /* v8  = 0000000022222222 */
add(v9,v2,v2);    /* v9  = 00000000222222220000000022222222
                         11111111333333331111111133333333 */
add(v8,oph,oph); /* oph = 00002222222200000000222222220000
                         11113333333311111111333333331111 */
dbl(v3,v4);       /* v4  = 02460246 */
add(v3,v4,v4)     /* v4  = 03690369 = 03210321 */
add(v3,v9,v3);    /* v3  = 01230123230123010123012323012301
                         12301230301230121230123030123012 */
add(v4,oph,oph); /* oph = 03212103210303210321210321030321
                         10323210321010321032321032101032 */
```

Phase cycles can be rather long. To avoid excessive typing, a shorthand syntax is very helpful (although it makes addition of phase vectors more difficult). It is recommended to use the shorthand syntax that is also used in the definition of phase tables (covered in the section "Shorthand Notation" on page 116):

`( a b c d )n` repeat the entire sequence within the parentheses `n` times (e.g., for `n=2`: `a b c d a b c d`)

`[ a b c d ]n` repeat each individual element within the brackets `n` times (e.g., for `n=2`: `a a b b c c d d`)

Such a simplified comment would look as follows:

```
mod4(ct,v3);      /* v3  =   0 1 2 3 */
hlv(ct,v9);       /* v9  = [ 0 1 2 3 ]2 */
hlv(v9,v9);       /* v9  = [ 0 1 2 3 ]4 */
assign(v9,v1);    /* v1  = [ 0 1 2 3 ]4 */
hlv(v9,v9);       /* v9  = [ 0 1 2 3 ]8 */
assign(v9,v2);    /* v2  = [ 0 1 2 3 ]8 */
hlv(v9,v9);       /* v9  = [ 0 1 2 3 ]16 */
hlv(v9,v9);       /* v9  = [ 0 1 2 3 ]32 */
mod2(v9,v9);      /* v9  = [ 0 1 ]32 */
dbl(v1,v1);       /* v1  = [ 0 2 ]4 */
add(v9,v1,v1);    /* v1  = [ 0 2 0 2 0 2 0 2  1 3 1 3 1 3 1 3 ]4 */
assign(v1,oph);   /* oph = [ 0 2 0 2 0 2 0 2  1 3 1 3 1 3 1 3 ]4 */
dbl(v2,v8);       /* v8  = [ 0 2 ]8 */
add(v9,v2,v2);    /* v9  = [ 0 2 0 2 1 3 1 3 ]8 */
add(v8,oph,oph); /* oph = [ 0 2 2 0 0 2 2 0  1 3 3 1 1 3 3 1 ]4 */
dbl(v3,v4);       /* v4  =   0 2 0 2 */
add(v3,v4,v4);    /* v4  =   0 3 2 1 */
add(v3,v9,v3);    /* v3  = ( 0 1 2 3 0 1 2 3  2 3 0 1 2 3 0 1 )2
                         ( 1 2 3 0 1 2 3 0  3 0 1 2 3 0 1 2 )2 */
add(v4,oph,oph); /* oph = ( 0 3 2 1 2 1 0 3  2 1 0 3 0 3 2 1 )2
                         ( 1 0 3 2 3 2 1 0  3 2 1 0 1 0 3 2 )2 */
```

10.3 Case 2: Creating Phase Math for Given Phase Tables

While the reconstruction of phase cycles from real-time math statements is not very difficult and can be solved by applying pure logic (e.g., in a computer program), the opposite task is quite different: nobody has yet come up with a simple logical approach for the construction of real-time math statements for a given phase cycle. The problem of creating a real-time math algorithm is largely a question of intuition, experience, and (mental) pattern recognition. Generating phase cycling math is also not an "exact science" in the sense that for any phase cycle there are many algorithms that lead to the correct result. The general goal is to create short and understandable algorithms, but what finally counts is the resulting phase cycle. For the execution of the pulse sequence in the pulse programmer, it normally doesn't matter *how* a phase cycle was generated.

Simple Phase Cycles

As we shall see later, most phase cycles (except perhaps the most simple ones) can be constructed by co-adding simpler (sub-)cycles: simple phase cycles can form the tools from which we can construct complex cycles. We should, therefore, first learn how to construct some simple phase cycles from `ct`.

Phase Incrementation

Incrementing phases are generated either from `ct` directly or by using division operation to slow `ct` down (the final `mod4` operations are optional for quadrature phases):

```
assign(ct,v1);      /* 0 1 2 3 */

mod4(ct,v1);        /* 0 1 2 3 */

hlv(ct,v1);         /* 0 0 1 1 2 2 3 3 4 4 5 5 ... */
mod4(v1,v1);        /* 0 0 1 1 2 2 3 3 */

hlv(ct,v1);         /* [ 0 1 2 3 ]2 */
hlv(v1,v1);         /* [ 0 1 2 3 ]4 */
hlv(v1,v1);         /* [ 0 1 2 3 ]8 */

divn(ct,three,v1);  /* [ 0 1 2 3 ]3 */
```

Alternating Phases

Alternating phases are obtained by doubling an incrementing sequence (of 90-degree phase shifts). Again, for quadrature phases, the final `mod4` operations are optional:

```
dbl(ct,v1);         /* 0 2 0 2 */

add(ct,ct,v1);      /* 0 2 0 2 */

hlv(ct,v1);         /* 0 0 1 1 2 2 3 3 4 4 ... */
dbl(v1,v1);         /* 0 0 2 2 4 4 6 6 8 8 ... */
mod4(v1,v1);        /* 0 0 2 2 */

hlv(ct,v1);         /* 0 0 1 1 2 2 3 3 4 4 ... */
mod2(v1,v1);        /* 0 0 1 1 0 0 1 1 */
dbl(v1,v1);         /* 0 0 2 2 0 0 2 2 */

hlv(ct,v1);         /* [ 0 1 2 3 ]2 */
hlv(v1,v1);         /* [ 0 1 2 3 ]4 */
dbl(v1,v2);         /* [ 0 2 ]4 */
hlv(v1,v1);         /* [ 0 1 2 3 ]8 */
dbl(v1,v1);         /* [ 0 2 ]8 */

divn(ct,three,v1); /* [ 0 1 2 3 ... ]3 */
dbl(v1,v1);         /* [ 0 2 4 6 ... ]3 */
mod4(v1,v1);        /* [ 0 2 ]3 */
```

Decrementing Phases

Decrementing phase cycles can be obtained by adding three copies of an incrementing phase cycle (multiplying an incrementing cycle by the constant `three`) or by subtracting an incrementing phase from a fixed phase:

```
add(ct,ct,v1);      /* 0 2 4 6 ... */
add(ct,v1,v1);      /* 0 3 6 9 ... */
mod4(v1,v1);        /* 0 3 2 1 0 3 2 1 */

mult(ct,three,v1); /* 0 3 6 9 ... = 0 3 2 1 */

hlv(ct,v1);         /* 0 0 1 1 2 2 3 3 ... */
```

```
add(three,one,v2); /* 4 (= 0) */
sub(v2,v1,v1);     /* 4 4 3 3 2 2 1 1 0 0 -1 -1 */
mod4(v1,v1);       /* [ 0 3 2 1 ]2 */

hlv(ct,v1);        /* 0  0  1  1  2  2  3  3  4 ... */
sub(zero,v1,v1);   /* 0  0 -1 -1 -2 -2 -3 -3 -4 ... */
mod4(v1,v1);       /* 0  0  3  3  2  2  1  1  0 ... */
```

Shifted Pattern

Shifted patterns “look like” simple patterns but are “not positioned right” (e.g., (`0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0`) looks like (`0 0 0 0 1 1 1 1`), but its values are shifted by two positions). Such phase cycles can be obtained by altering `ct` first:

```
add(ct,two,v1);    /* 2 3 4 5 6 7 8 9 ... */
hlv(v1,v1);        /* 1 1 2 2 3 3 4 4 ... */
hlv(v1,v1);        /* 0 0 1 1 1 1 2 2 ... */
mod2(v1,v1);       /* 0 0 1 1 1 1 0 0 */
sub(one,v1,v1);    /* 1 1 0 0 0 0 1 1 */
mult(v1,three,v3); /* 0 0 3 3 3 3 0 0 */

add(ct,one,v1);    /* 1 2 3 4 5 6 7 8 ... */
hlv(v1,v1);        /* 0 1 1 2 2 3 3 4 ... */
dbl(v1,v1);        /* 0 2 2 0 0 2 2 0 */
```

Complex Phase Cycles

The trick for generating complex phase cycles with real-time math is to recognize “internal periodicities,” or repeating patterns *within* a phase cycle, and (more difficult) a shifted pattern in a phase cycle. The target is to “decompose” a complex phase cycle into statements that can either be generated directly or through one of the short algorithms presented above (“reverse synthesis”). In the pulse sequence, we can then compose the complex phase cycle simply by adding up the elements. We now look at a few examples (all used for quadrature phase shifting with a range of 0 to 3).

0 0 2 2 2 2 0 0 Pattern

This phase cycle can either be regarded as a shifted pattern (see above) or it can be thought of as being identical to (`0 0 2 2 2 2 4 4`). We can split that into two groups of four phases, within which the phase is alternated: (`0 0 2 2`), (`2 2 4 4`). The second of these groups is shifted by two units compared to the first group. This leads to the following decomposition:

```
0 0 2 2 2 2 4 4 = 0 0 2 2 0 0 2 2 + 0 0 0 0 2 2 2 2
```

We can, therefore, generate this phase cycle from two simple cycles. Note that an intermediate result from the second part can be used for the first part:

```
hlv(ct,v1);        /* 0 0 1 1 2 2 3 3 */
hlv(v1,v2);        /* 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 */
dbl(v1,v1);        /* 0 0 2 2 */
dbl(v2,v2);        /* 0 0 0 0 2 2 2 2 */
add(v1,v2,v1);     /* 0 0 2 2 2 2 4 4 */
mod4(v1,v1);       /* 0 0 2 2 2 2 0 0 */
```

The treatment as shifted pattern (as shown previously) is slightly more efficient:

```
add(ct,two,v1);          /* 2 3 4 5 6 7 8 9 ... */
hlv(v1,v1);              /* 1 1 2 2 3 3 4 4 ... */
hlv(v1,v1);              /* 0 0 1 1 1 1 2 2 ... */
dbl(v1,v1);              /* 0 0 2 2 2 2 4 4 ... */
```

0 2 1 3 Pattern

This phase cycle can be interpreted as being composed from two subcycles (`0 2`) and (`0 0 1 1`), with the second two pulses shifted by one unit (90 degrees) against the first two. This leads to the following algorithm:

```
dbl(ct,v2);              /* 0 2 0 2 */
hlv(ct,v1);              /* 0 0 1 1 2 2 3 3 */
mod2(v1,v1);             /* 0 0 1 1 */
add(v1,v2,v1);           /* 0 2 1 3 */
```

0 2 3 1 Pattern

Exchanging the last two phases in the previous pattern changes things completely. We get a cycle composed of the simple cycle (`0 0 1 1`) and a shifted pattern (`0 2 2 0`):

```
add(ct,one,v2);          /* 1 2 3 4 */
hlv(v2,v2);              /* 0 1 1 0 */
dbl(v2,v2);              /* 0 2 2 0 */
hlv(ct,v1);              /* 0 0 1 1 2 2 3 3 */
mod2(v1,v1);             /* 0 0 1 1 */
add(v1,v2,v1);           /* 0 2 3 1 */
```

0 2 1 3 2 0 3 1 Pattern

Splitting this cycle, which is equivalent to (`0 2 1 3 2 4 3 5`), into two groups of four leads to the two subcycles, (`0 0 0 0 2 2 2 2`) and (`0 2 1 3 0 2 1 3`), with the (`0 2 1 3`) part of the second subcycle being composed from two more subcycles, (`0 2`) and (`0 0 1 1`).

```
dbl(ct,v2);              /* 0 2 0 2 */
hlv(ct,v1);              /* 0 0 1 1 2 2 3 3 */
hlv(v1,v3);              /* [ 0 1 2 3 ]4 */
dbl(v3,v3);              /* [ 0 2 ]4 */
mod2(v1,v1);             /* 0 0 1 1 */
add(v1,v2,v1);           /* 0 2 1 3 */
add(v1,v3,v1);           /* 0 2 1 3 2 0 3 1 */
```

It turns out that there is a much simpler solution if we take four groups of two phases, in which case the elements are (`0 2`) and (`0 0 1 1 2 2 3 3`):

```
hlv(ct,v1);              /* 0 0 1 1 2 2 3 3 */
dbl(ct,v2);              /* 0 2 */
add(v1,v2,v1);           /* 0 2 1 3 2 0 3 1 */
```

0 2 3 1 2 0 1 3 2 0 1 3 0 2 3 1 Pattern

Splitting this phase cycle into four groups of four phases we get an A-B-B-A pattern with A=(0 2 3 1) and B=(2 0 1 3). B is phase-shifted by 180 degrees (2 units); this becomes obvious if we think about the equivalent of 4 and 0 or 5 and 1 for quadrature phase shifting: B is (2 4 5 3). The decomposition of phase cycle (0 2 3 1 2 0 1 3 2 0 1 3 0 2 3 1) is equal to (0 2 3 1) plus (0 0 0 0 2 2 2 2 2 2 2 0 0 0 0), a simple shifted pattern, [0 2]8 left-shifted by four steps.

```
add(ct,one,v2);        /* v2 = 1 2 3 4 */
hlv(v2,v2);            /* v2 = 0 1 1 0 */
dbl(v2,v2);            /* v2 = 0 2 2 0 */
hlv(ct,v1);            /* v1 = 0 0 1 1 2 2 3 3 */
mod2(v1,v1);           /* v1 = 0 0 1 1 */
add(v1,v2,v1);         /* v1 = 0 2 3 1 */
dbl(two,v3);           /* v3 = 4 */
add(ct,v3,v4);         /* v4 = 4 5 6 7 8 9 10 11 12 13 .. */
dbl(v3,v3);            /* v3 = 8 */
divn(v4,v3,v4);        /* v4 = 0 0 0 0 1 1 1 1 1 1 1 1 2 .. */
dbl(v4,v4);            /* v4 = [ 0 2 2 4 4 6 6 8 8 10 ... ]4
                         v4 = [ 0 2 2 0 ]4 */
add(v1,v4,v1);         /* v1 = 0 2 3 1 2 4 5 3  2 4 5 3 0 2 3 1
                         v1 = 0 2 3 1 2 0 1 3  2 0 1 3 0 2 3 1 */
```

Note that even for quadrature phases the “implicit mod4 function” can only be applied if no division is involved (after a division it is all right again)!

0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 1 0 3 2 3 2 1 0 3 2 1 0 1 0 3 2 Pattern

The best approach for this phase cycle is to start by splitting into two groups, and then splitting the resulting 16-step cycle into four groups of four steps each:

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

0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1  0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 +
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 =
0 3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 +
0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0
```

These are all subcycles we have discussed previously: (0 3 2 1) is a simple decrementing cycle, [0 2 2 0]4 was used in the previous phase calculation, and the subcycle [0 1]16 can be calculated as (ct/16) mod 2:

```
dbl(two,v1);           /* v1 = 4 */
add(ct,v1,v2);         /* v2 = 4 5 6 7 8 9 10 11 12 13 .. */
dbl(v1,v1);            /* v1 = 8 */
divn(v2,v1,v2);        /* v2 = 0 0 0 0 1 1 1 1 1 1 1 1 2 .. */
dbl(v2,v2);            /* v2 = [ 0 2 2 4 4 6 6 8 8 10 ... ]4
                         v2 = [ 0 2 2 0 ]4 */
dbl(v1,v1);            /* v1 = 16 */
divn(ct,v1,v3);        /* v3 = ct/16 = [ 0 1 2 3 ]16
mod2(v3,v3);           /* v3 = [ 0 1 ]16
sub(zero,ct,v4);       /* v4 = 0 -1 -2 -3 = 0 3 2 1 */
add(v4,v2,v4);         /* v4 = 0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 */
add(v4,v3,v4);         /* v4 = 0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1
                                1 0 3 2 3 2 1 0 3 2 1 0 1 0 3 2 */
```

This phase cycle is longer than the previous one, but it requires fewer math statements. If we skip the `mod2` step, we would even obtain a phase cycle of 64 steps, which uses less Acode (39 words) than if we would store the resulting phase table in the Acode (as 16-bit numbers)! Divisions normally multiply the phase cycle length by the divisor, multiplications and modulo steps usually shorten the length of a phase cycle.

Phase Cycles for Many Pulses

By tradition, Varian pulse sequences were mostly written with a code section containing the real-time phase math for all pulses (and the receiver), followed by the actual pulse sequence. This has the advantage of keeping things apart for more clarity, and also it often permits using intermediate results or the final phase cycle for one pulse also for the calculation of other phase cycles, reducing the total number of math statements for the entire pulse sequence. On the other hand, if there are many pulses with many different phase cycles, there may be a problem with the limited number of real-time variables (14 total). There may be more phase cycles in a single pulse sequence than real-time variables. Also, such a convoluted, complex phase cycling section is often difficult to decode.

The other extreme would be to calculate each phase *just before using it*. This keeps the individual phase calculations simpler and avoids conflicts with the number of real-time variables (after having calculated and used one phase cycle, all variables can be reused for the next calculation). On the other hand, this method may dramatically increase the total number of math statements in a pulse sequence, can make it difficult to read (pulses and delays are interspersed with phase calculations), and blows up the Acode. In particular, this method also neglects the phase *relations* within a pulse sequence—the fact that intermediate results and entire phase cycles can often be reused for other phase calculations is not incidental, but due to the close relationship between all phases in a pulse sequence! More about that in Chapter 11, “Phase Tables,” on page 115.

10.4 Real-Time Logical Decisions

The Acode interpreter not only has the ability of performing a linear interpretation, but it can also make decisions and perform branching based on the contents of real-time variables (a looping capability is also built in, see Chapter 14, “Repeating Events,” on page 147). The use of real-time decisions for the conditional creation of FIFO words (conditional pulses or pulse sequence fragments) is discussed in more detail in Section 15.2, “Real-Time Decisions,” on page 160. Here, we just discuss the application of real-time decisions to the construction of phase cycles using real-time math.

Let’s look at a phase cycle used in a phase-cycling-based implementation of the E.COSY experiment (basically a linear combination of different double-quantum filtered COSY experiments). In this experiment, the first two pulses undergo a complicated scheme of 45-degree phase shifts:

```
0 1 0 7 0 1 2 7 0 1 4 7 0 1 6 7  0 1 0 7 0 1 2 7 0 1 4 7 0 3 6 5
```

There must be a direct mathematical way to construct this kind of phase cycle, but that algorithm would be excessively complicated. Instead, let us take a different approach by separating the phases for the even and odd scans (the phases are in 45-degree increments, the range of values therefore is 0 to 7) :

```
0 1 0 7 0 1 2 7 0 1 4 7 0 1 6 7  0 1 0 7 0 1 2 7 0 1 4 7 0 3 6 5 =

0   0   0   2   0   4   0   6    0   0   0   2   0   4   0   6
  1   7   1   7   1   7   1   7    1   7   1   7   1   7   3   5
```

The new (sub-)phase cycles look much more manageable than the original one, at least the subcycle for the odd pulses (for the even scans we need some exception handling, because the last two phases behave differently from the others).

What we now need is some construct that can be described as follows:

```
if (odd scan)
    (use phase cycle a);   /* 0 0 0 2 0 4 0 6  */
else
    (use phase cycle b);   /*  1 7 1 7 1 7 1 7 */
```

Fortunately, this is very easy to do. We just use a real-time variable that is zero for all the odd transients by applying `mod2` to `ct`, and then we can use real-time branching to calculate specific phases for both the odd and even transients (for more information in real-time decisions, see Section 15.2, “Real-Time Decisions,” on page 160):

```
mod2(ct,v1);     /* 0101 */
ifzero(v1);
  ...            /* phase selection for odd transients */
elsenz(v1);
  ...            /* phase selection for even transients */
endif(v1);
```

Using this mechanism, we can switch between two phase cycles with every scan. Unlike shown above, the partial phase cycles do not have gaps, but the “empty positions” are filled with suitable numbers (such that the partial phase cycles are easy to calculate).

```
Full cycle: 0 1 0 7 0 1 2 7 0 1 4 7 0 1 6 7  0 1 0 7 0 1 2 7 0 1 4 7 0 3 6 5

Cycle A:    0 0 0 0 0 2 2 0 0 4 4 0 0 6 6  0 0 0 0 0 2 2 0 0 4 4 0 0 6 6
Cycle B:    1 1 7 7 1 1 7 7 1 1 7 7 1 1 7  1 1 7 7 1 1 7 7 1 1 7 7 3 3 5 5
Flag A/B:   0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

The two resulting subcycles are definitely easier, but still not trivial to construct using standard math—but we can use exactly the same algorithm to construct the two subcycles:

```
Cycle A:  0 0 0 0 0 0 2 2 0 0 4 4 0 0 6 6  0 0 0 0 0 0 2 2 0 0 4 4 0 0 6 6

Cycle C:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Cycle D:  0 0 0 0 2 2 2 2 4 4 4 4 6 6 6 6  0 0 0 0 2 2 2 2 4 4 4 4 6 6 6 6
Flag C/D: 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1  0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
```

Subcycle A can be derived from two trivial phase cycles by switching between the two secondary subcycles with every pair of scans. Subcycle B can be constructed in a similar way (x stands for a non-zero value in the flag variable):

```
Cycle B:  1 1 7 7 1 1 7 7 1 1 7 7 1 1 7 7  1 1 7 7 1 1 7 7 1 1 7 7 3 3 5 5

Cycle E:  3 3 5 5 3 3 5 5 3 3 5 5 3 3 5 5  3 3 5 5 3 3 5 5 3 3 5 5 3 3 5 5
Cycle F:  1 1 7 7 1 1 7 7 1 1 7 7 1 1 7 7  1 1 7 7 1 1 7 7 1 1 7 7 1 1 7 7
Flag E/F: x x x x x x x x x x x x x x x x  x x x x x x x x x x x x 0 0 0 0
```

In the actual coding, we first construct the four secondary subcycles (one of them—subcycle C—is trivial) and the three flag variables, and then construct the final phase cycle with a series of nested real-time `if` statements (this allows bypassing the intermediate construction of the subcycles A and B). The flag variable "`E/F`" is best created as a shifted incrementing cycle:

```
Flag E/F: 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4  5 5 5 5 6 6 6 6 7 7 7 7 0 0 0 0
```

This is constructed using `modulo 8` of `ct/4+1`. The entire phase "calculation" can be coded as follows:

```
mod2(ct,v10);          /* v10 = 0 1 = flag A/B */
hlv(ct,v1);            /* v1  = ct/2 */
mod2(v1,v11);          /* v11 = 0 0 1 1 = flag C/D */
hlv(v1,v12);           /* v12 = ct/4 */
incr(v12);             /* v12 = ct/4 + 1 */
dbl(two,v13);          /* 4 */
dbl(v13,v13);          /* 8 */
modn(v12,v13,v12)      /* [ 1 2 3 4 5 6 7 0 ]2 = flag E/F */

mod2(v1,v2);           /* v2 = 0 0 1 1 */
dbl(v2,v2);            /* v2 = 0 0 2 2 */
mult(v2,three,v3);     /* v3 = 0 0 6 6 */
add(v2,three,v2);      /* v2 = 3 3 5 5 = subcycle E */
incr(v3);              /* v3 = 1 1 7 7 = subcycle F */
hlv(v1,v1);            /* v1 = ct/4 */
dbl(v1,v1);            /* v1 = [ 0 2 4 6 8 10 12 ... ]4 */
modn(v1,v13,v1);       /* v1 = [ 0 2 4 6 ]4 = subcycle D */

ifzero(v10);           /* odd scans */
   ifzero(v11);        /* scans 1(,2),5(,6),etc. */
      assign(zero,v1); /* subcycle C */
   endif(v11);         /* (subcycle D is default in v1) */
elsenz(v10);           /* even scans */
   ifzero(v12);        /* scans 30,32 */
      assign(v2,v1);   /* select subcycle E */
   elsenz(v12);        /* scans 2,4,8,...,28 */
      assign(v3,v1);   /* select subcycle F */
   endif(v12);
endif(v10);
stepsize(45.0, TODEV);
xmtrphase(v1);
...
```

The full E.COSY phase cycle is six times longer than the one shown above, with yet another "exception" in the sixth loop. Although it is obviously simpler to generate such phase cycles using tables, the main intention here was to demonstrate the possibility of generating "arbitrary" phase cycles using real-time math in combination with real-time decisions.

10.5 Steady-State Phase Cycling

All phase calculations up to now were based on the `ct` real-time variable. This works fine for all standard pulses up to `nt=32768`. After that many transients, the phase cycling continues as if `ct` would restart at 0, because the real-time variable `ct` is only the low-order half of `LC->ct`. For scan 32768, `LC->ct` is set to 32768, which is 1 in the high-order half (word) and 0 in the low-order word. This should be fine, because the number of steps in most phase cycling schemes is in powers of two anyway; even if this is not the case. After that many scans, a disruption in the phase cycling has no noticeable effect on the final spectrum.

Nevertheless it is a *bad idea* to use `ct` as exclusive basis for phase cycling. During steady-state pulses, the `ct` counter remains at 0, and therefore no phase cycling occurred during steady-state transients! This means that no matter how big a number of steady-state scans is selected, the steady state during the `ss` pulses is *different* from the steady state reached after the first few real scans. In all but the most primitive experiments (`s2pul` with `cp='n'`), the true steady state is only reached after the first "real" scans! This is particularly bad with cancellation experiments, unless some presaturation scheme (homospoil-90-homospoil, or two long orthogonal spin locking pulses at the beginning of each scan) is used to erase all residual coherence from previous scans.

The proper solution is *not* to use `ct` directly for phase calculations, but to construct a counter that varies though the steady-state pulses. This is possible by using the `ssctr` real-time variable. `ssctr` is equal to the value of `ssval` for the first steady-state scan and is decremented after each `ss` scan. When the `ssctr` variable is zero, the real scans start. Various schemes have been proposed to combine the `ct` and `ssctr` counters, like the following:

```
ifzero(ssctr);
   assign(ct,v9);          /* 0, 1, 2, ... nt-1 */
elsenz(ssctr);
   sub(ssval,ssctr,v9);    /* 0, 1, 2, ... ss-1 */
endif(ssctr);
```

After this, `v9` would be used instead of `ct` for all phase calculations. This scheme takes into account the fact that `ssctr` "counts backwards" (it is decremented, instead of incremented like `ct`) so phase cycling starts the same way as with the "real" scans.

This scheme is certainly better than using `ct` only, but it still has some deficiencies. After the steady-state scans, the phase cycling counter jumps back to zero and may, therefore, disrupt the steady state. This can again be particularly bad with cancellation experiments (e.g., using an odd number of steady-state scans with double-quantum filtered experiments). In principle, this scheme requires setting `ss` to the total length of the phase cycle! The idea behind steady-state scans is to "make the spins believe that we have been performing an infinite number of scans in the past," before the "real" scans start. This can be achieved by performing *the last elements* of the phase cycle rather than the beginning elements during the steady-state scans. This can be achieved in a very simple way:

```
sub(ct,ssctr,v9);           /* -ss,-ss+1,.. -1,0,1,.. nt-1 */
```

This works fine, because the signed binary value -1 is equivalent to the unsigned binary value 1,111,111,111,111,111 (decimal 65535) or 2^n-1. As long as the length of the phase cycle is a power of two, this will execute the last elements of a phase cycle. As there are only very few examples of pulse sequences where the phase cycle length is

not a power of two (COSY-3 is one of them), this should be made the default way to generate a base counter for phase cycling.

10.6 C Constructs and Phase Calculations

C constructs are used to implement parameter-dependent phase cycling. The most frequently used parameter-dependent phase cycling is the type for phase-sensitive multidimensional NMR, where a coding similar to the type below is applied. A pseudo-variable `phase` is used to differentiate between the TPPI (time-proportional phase incrementation) and hypercomplex (States-Haberkorn-Ruben) types of phase-sensitive 2D (*n*D) spectra. The Varian convention is to use `phase=3` for TPPI spectra (adding 90 degrees to the phase shift of the first pulse with every time increment) and `phase=1,2` for hypercomplex experiments, where `phase=1` is unshifted, `phase=2` has 90 degrees added to the pulses prior to the evolution phase (and relevant to the phase of the observed signal), as well as to the observe phase. In the coding below, `v1` is supposed to be the phase of the pulse prior to the evolution (e.g., in a NOESY experiment):

```
int t1_counter = (int) (d2 * sw1 + 0.1);
int phase1 = (int) getval("phase");
if (phase1 == 2)
   incr(v1);
else if (phase1 == 3);
{
   initval((double) t1_counter), v10);
   add(v1,v10,v1);
}
```

This is only the most basic implementation for phase-sensitive NMR. Modern sequences use refinements for the hypercomplex method that are discussed in more detail in Chapter 19, “Multidimensional Experiments,” on page 215.

There is another nice example of the use of C constructs for implementing variable phase cycles: in the `relayh` sequence’s relayed COSY variant (`relay` greater than 0), in which the number of refocusing elements is determined by the parameter `relay`. The refocusing pulses cannot (or at least should not) be cycled only synchronously, because this would lead to an accumulation of the errors due to imperfections in the refocusing pulses. In the Varian implementation of this sequence, C constructs are used in two places to adjust the phase cycling to the actual setting of the `relay` parameter:

- The length of the `00112233` phase cycling is calculated in a C `for` loop.
- The phase cycling for the refocusing periods is calculated “on the fly”, while coding the relay (refocusing) intervals such that the 90- degree pulse after the first relay has the fastest phase alternation and the phase inversion of 90-degree pulses following subsequent relay periods is slowed down progressively (by a factor of two per relay period).

This is a slightly simplified version of this pulse sequence:

```
/*    relayh - relayed cosy, including regular cosy    */

pulsesequence()
{
   int i, relay = (int) (getval("relay") + 0.5);
   double tau = getval("tau");
```

```
   hlv(ct, v1);                    /* v1 = 00112233 */
   for (i = 0; i < relay + 1; i++)
      hlv(v1,v1);                  /* [ 0 1 2 3 ]2**(relay+2) */
   dbl(ct,oph);                    /* oph = 0202 0202 */
   add(ct,v1,v2);                  /* v2  = 0123 0123 + 00112233 */
   add(oph,v1,oph)                 /* oph = 0202 0202 + 00112233 */
   hlv(ct,v3);                     /* v3  = 0011 2233 4455 ... */

   status(A);
      hsdelay(d1);                 /* preparation period */
   status(B);
      pulse(pw,v1);
      delay(d2);                   /* evolution period */
      pulse(pw,v2);                /* start of mixing period */
      for (i = 0; i<relay; i++)    /* relay coherence */
      {
         delay(tau/2);
         pulse(2.0*pw,v2);
         delay(tau/2);
         hlv(v3,v3);              /* v3  = [ 0 1 2 3 ]2**(relay+1) */
         dbl(v3,v4);              /* v4  = [ 0 2 ]2**(relay+1) */
         add(v2,v4,v5);           /* v5  = v4+v2 (with 00112233) */
         pulse(pw,v5);
      }
   status(C);
}
```

10.7 Why Phase Calculations?

There is no doubt that real-time phase calculations are not the simplest way to generate phase cycling in pulse sequences, because it may be difficult (some users may even call it painful) for inexperienced users to decode phase calculations, or even more to generate a real time calculation algorithm for a given phase table if a pulse sequence is published with the phase cycles in the form of numeric tables. On the other hand, there are definitely also arguments for phase calculations (as opposed to using tables):

- For very long phase cycles, the calculations are more efficient in terms of Acode space (see also Chapter 11, “Phase Tables,” on page 115).
- Scientists who generate new pulse sequences do *not* think in phase tables but rather in algorithms (like phase cycling individual pulses and/or the observe phase for a specific coherence level selection, multiple-quantum filtering, etc.). Real-time phase calculations offer a way to directly implement such algorithms.
- Understanding the internal phase cycling mechanisms from long numeric tables may be as difficult (if not more difficult) than trying to understand real-time math. Changing the phase cycling order with tables may be more difficult than with real-time math.

10.8 Real-Time Random Numbers

There are many examples of a systematic variation of a variable during the execution of a pulse sequence, such as:

- Incrementation of the evolution time as function of the increment number.
- Systematic incrementation or decrementation of the mixing time or associated time intervals in some NOESY-type experiments.
- Time-proportional phase incrementation for phase-sensitive *n*D experiments.

There are also examples of random variation of variables, like the variation of the mixing time in some (other) types of NOESY experiments. In this case, the randomization happens from increment to increment, and a C construct can be used to generate individual random numbers for every code segment:

```
#define CONSTANT 1073741824
double rv;
if (ix == 1)
   srandom(getpid());
rv = ((double) (random() - CONSTANT))/((double) CONSTANT);
mix = mix * (1.0 + rv * getval("mixvar"))
```

The randomizer must be initialized once (with the first increment) by calling the `srandom` function (calling it with the current process-ID ensures true randomization, also between subsequent calls of the same pulse sequence). The function `random` then generates a random number between 0 and 2,147,483,647 (the maximum positive 32-bit integer). `mixvar` is a scaling factor (between 0 and 1.0 in this case, some implementation may use a scaling between 0 and 100.0) that allows controlling the degree of randomization (the proportion of the variation range and the total duration of the delay. The implementation above varies the delay in both directions.

In some rare cases—like some Z-filters, or randomization of `d1` for avoiding phasing problems in the case of long T_2-relaxation (sharp lines) and short repetition rates—such a randomization should occur within a single increment, and systematic variation is not desired or not possible. This is a tricky issue, because the acquisition operating system (the Acode interpreter) does not provide a random number generator.

This problem has two solutions: randomization using a table of random numbers (see also Section 11.7, “Using Tables as Source for Random Numbers,” on page 135) and randomization using a real-time pseudo-random number generator.

An algorithm for the second solution has been presented in *Magnetic Moments*[1]; the coding below is adapted from that article:

```
double random = getval("random");
if ((d1 - random/2.0) > 0.2e-6)
   delay(d1 - random/2.0);
if (random > 0.0)
{
   initval((double) ((ix*13) % 511),v14);              /* seed */
   initval(4.0,v13);                    /* v13 = 4 = phasecycle */
   dbl(two,v7); dbl(v7,v7);                            /* v7  = 8 */
   mult(v7,v7,v7);                                    /* v7  = 64 */
   dbl(v7,v8);                                       /* v8  = 128 */
   decr(v7);                                          /* v7  = 63 */
```

[1] R. Boyko, B. Sykes & G. Gray, *Magnetic Moments*, **III.3**, p.4.

```
   decr(v8);                                        /* v7  = 127 */
   modn(ct,v13,v6);              /* v6  = 0 1 2 .. phasecycle-1 */
   ifzero(v5);
      divn(ct,v13,v6);                    /* v6  = ct/phasecycle */
      mult(v7,v14,v14);                       /* v14 = (seed*63) */
      add(v14,v6,v14);       /* v14 = (seed*63 + ct/phasecycle) */
      modn(v14,v8,v14); /* v14 = (seed*63 + ct/phasecycle)%127 */
   endif(v6);
   loop(v14,v6);
      delay(random/126.0);
   endloop(v6);
}
```

This method generates random numbers in the range of 0 to 126. The authors of the algorithm verified that this mechanism generates nearly ideally distributed numbers within the specified range. The initialization of `v14` (the random seed) with a function of the increment number `ix` also ensures true randomization between different traces. Unlike the variation using a C randomization based on the process-ID, this type of randomizer generates the *same random numbers with every* `go` *command*. Ultimately, the real-time random number is used to determine the (real-time) loop cycles over a small delay (1/126 of the total variation range). Loops are discussed in detail in Chapter 14, “Repeating Events,” on page 147.

Note that in the above code, the real-time variable `v14` (the random seed) is initialized using the `initval` statement. This number is then modified *by purpose* (in the vast majority of the cases this is “forbidden”). The random variation occurs at the beginning of every phase cycle. It is a good idea to complete every phase cycle under identical conditions in order to ensure proper subtraction and cancellation.

Chapter 11. Phase Tables

The basic syntax and the mechanisms of using phase tables are simple and easy to understand, construct, and use.

11.1 Basic Syntax

A simple, external ASCII file, such as the file `/vnmr/tablib/sample` of `~/vnmrsys/tablib/sample`, takes up the actual phase tables:

```
/* sample phase table file */
t1 = 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3     /* 1st pulse */
t2 = 0 2 1 3 1 3 2 0 2 0 3 1 3 1 0 2     /* 2nd pulse, oph */
```

The table file syntax is nearly trivial. Comments are defined the same way as in a C program:

- Tables have names between `t1` and `t60`.
- Table values are positive integer numbers (16-bit, range between 0 and 32767), separated by spaces.
- Table values are separated from the table name by an *equal sign* (`=`) *that must be surrounded by spaces*.
- Table length is arbitrary. If necessary, tables can be extended over several lines.

In the pulse sequence, the table file must be loaded explicitly (specifying its name); thereafter, the specified tables (`t1`, `t2` in the above example) can be used directly:

```
#include "standard.h"
pulsesequence()
{
   loadtable("sample");
   status(A);
      hsdelay(d1);
   status(B);
      pulse(p1,t1);
      hsdelay(d2);
   status(C);
      pulse(pw,t2);
      setreceiver(t2);
}
```

The table file name is relative. In this example `~/vnmrsys/tablib/sample` has preference over `/vnmr/tablib/sample`. The table names `t1` and `t2` are used like real-time variables. They are of type `codeint`, but unlike real-time variables do *not* point to some element in the `LC` structure. As we will see later, the table names are used in the C code only to differentiate between the various tables (the table names are lost in the Acode, see below). The receiver phase is defined in the real-time variable `oph`. If the receiver phase should be taken from a phase table, `oph` must be set from a table. This is usually done with the `setreceiver` statement, but the same can also be achieved by the `getelem` statement:

```
getelem(t2,ct,oph);
```

This explicitly extracts element number `ct` from the table `t2`. `getelem` can also be used to extract phases from tables into other real-time variables, for example:

```
getelem(t1,ct,v10);
```

This way, real-time math can be performed on phases from phase tables (real-time math does not work with table names). Applications for this feature will be discussed in detail in Section 11.6, “Combining the Best of the Two Worlds,” on page 129.

Shorthand Notation

Two types of shorthand notation exist for external phase table files: repeated table sequences and repeated table elements.

Repeated Table Sequences

By definition, any table is a repeated sequence. If the table index (the pointer to the current table element) is larger than the number of elements in the table, the table look-up automatically restarts at the beginning of the table. Thus, the table index is automatically taken *modulo of the table length*. This occurs individually for each table. For example, even if the total phase cycle length (in a pulse sequence) is 16, it is not necessary to define:

```
t1 = 1 3 1 3 1 3 1 3 1 3 1 3 1 3 1 3
```

This would be a waste of Acode space. The much shorter table

```
t1 = 1 3
```

defines exactly the same table, but requires much less space. We don’t need a shorthand notation for this case[1]. On the other hand, a shorthand notation is useful in the case of a table like:

```
t1 = 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
     1 3 1 3 1 3 1 3 1 3 1 3 1 3 1 3
     2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0
     3 1 3 1 3 1 3 1 3 1 3 1 3 1 3 1
```

In shorthand notation this would be written as

```
t1 = ( 0 2 )8  ( 1 3 )8  ( 2 0 )8  ( 3 1 )8
```

The sequence between the parentheses is repeated as many times as indicated by the repetition number that must follow the closing parenthesis *without space* (the spaces between parentheses and the numbers inside are optional).

Repeated Table Elements

Very often tables consist of repeated elements, such as

```
t1 = 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
     1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
```

In shorthand notation, this can be dramatically simplified and written as

```
t1 = [ 0 2 1 3 ]8
```

[1] There is an exception: table-to-table math functions (`ttadd`, `ttsub`, `ttmult`, `ttdiv`) require that the target table (which is also one of the two operand tables) is at least as long as the second operand table. For this reason, we may have to write down a table sequence more than once (usually done using shorthand notation). See the manual *VNMR User Programming* for more information.

The syntax around the brackets is the same as with the parentheses, except that the index after the brackets indicates the number of repetitions for each table element between the brackets.

Brackets and parentheses can be used sequentially in within the same table but cannot be nested. Note that the shorthand notation is just an easy and simple way of *defining and entering* long phase tables—*the tables are internally generated at full length* (i.e., no Acode space is saved by using shorthand notation, see also below).

Advanced Features

Several advanced features, including the division factor and an automatically incrementing index, make working with tables easier.

Division Factor

The last table described above can also be written differently:

```
t1 = { 0 2 1 3 }8
```

Again, the same syntax as with the parentheses and the brackets applies to the braces ("curly brackets"). This looks like exactly the same thing as the shorthand syntax using the brackets and, from a superficial point-of-view, this is true. The index after the braces defines how many times each element is repeated before passing to the next element in the table. However, internally the braces work differently. In the case of the braces, the table generated internally is only the table elements between the braces.

The index after the braces defines what is called the *division factor* (sometimes also called the *division return factor*). Before reading a table element, the table index (the pointer to the current table element) is *divided* by the division factor, before the modulo function (modulo the table length) is performed and the table element is extracted. Every table has a division factor. The default setting for the division factor is 1, and the braces simply alter that value.

In other words, the braces not only are a shorthand notation for tables with repeated elements, they also define a "shorthand table" internally (i.e., different from the brackets, the braces also save Acode space, see below). *Therefore, it is recommended to use braces instead of brackets wherever possible*[2].

Brackets and parentheses cannot be nested within themselves, but they can be nested within braces, as seen in the following examples:

```
t1 = { ( 0 2 )4 ( 1 3 )4 }4
t2 = { [ 0 2 ]4 ( 1 3 )4 }4
```

In conventional syntax, these tables would be written as follows:

```
t1 = 0 0 0 0 2 2 2 2 0 0 0 0 2 2 2 2
     0 0 0 0 2 2 2 2 0 0 0 0 2 2 2 2
     1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3
     1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3
```

[2] There is an exception to this as well. Table-to-table math operations (`ttadd`, `ttsub`, `ttmult`, and `ttdiv`) directly operate on the vectors as they are defined internally, without taking into consideration any division factors. This can lead to unexpected results if division factors different from 1 are used, apart from complications due to the restrictions with respect to the table lengths in such operations (see the previous footnote).

```
t2 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
     1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3
     1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3
```

Note that the division factor always applies to the entire table; whereas, brackets (and parentheses) can also be used on parts of a table only:

```
t1 = [ 0 2 ]4  2 0 2 0 2 0 2 0
```

This table is perfectly acceptable, but the same line with braces instead of brackets *is not*.

The maximum division factor is 64. For tables with a higher number of repetitions on the individual table elements, the following definitions can be used:

```
t1 = { 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 }64
t1 = { [ 0 2 1 3 ]4 }64
```

The division factor can not only be defined in the table file, but it can also be defined or altered within the pulse sequence, using the `setdivnfactor` statement, for example:

```
setdivnfactor(t1,16);
```

This feature is used frequently in the type of coding described in Chapter 19, “Multidimensional Experiments,” on page 215.

Tables with Automatically Incrementing Index

The table index (the pointer to the current table element) is normally either specified explicitly (using the `getelem` statement), or the transient counter `ct` is automatically (implicitly) taken as index (when specifying a table address `t1` . . . `t60` as phase argument to statements like `pulse`, `txphase`, etc.). This way, the table index is incremented automatically with every transient (implicit use) or set as specified by the index argument to `getelem`.

However, there are applications where we would like to “scan” through a entire table (or parts of it) during a single transient, for instance, when the table defines the amplitudes of a pulse shape or the phases of an explicitly programmed decoupling pattern. Using table features discussed up to now, this could be programmed in the following manner:

```
assign(one,v9);             /* table index */
initval(32.0,v10);          /* loop cycles (no. of table elements) */
loop(v10,v11);
  getelem(t1,v9,v1);        /* extract table value */
  rgpulse(pw,v1,0.0,0.0);   /* perform one pulse element */
  incr(v9);                 /* increment table index */
endloop(v10);
```

Much of the code above is for the pointer generation and incrementation after reading an using each table element in turn (note that the number of loop cycles is *not necessarily* equal to the number of table elements). To make it easier to use tables for reading out successive amplitude and phase elements from tables within a single pulse sequence pass, phase tables have been equipped with an *autoincrement attribute*. This attribute is unset by default. It can be set in the table file by using “+=” instead of the simple equals sign “=” between the table name and the table elements:

```
t1 += { 0 2 2 0 2 0 0 2 }4
```

Alternatively, the autoincrement attribute can be set with the `setautoincrement` statement (there is no function to unset the autoincrement attribute):

```
setautoincrement(t1);
```

With this feature, a “scan through a phase table” can be realized in a much simpler way:

```
initval(32.0,v10);         /* loop cycles (no. of table elements) */
loop(v10,v11);
   rgpulse(pw,t1,0.0,0.0); /* perform one pulse element */
endloop(v10);
```

Note that with the autoincrement attribute the table index is reset *once per FID* (when creating the code). If a table is not read out completely during one scan, the table index is incremented from *somewhere within the table* with the next scan.

Autoincrementing tables can be used not only directly (e.g., as phase argument to a `pulse` or `txphase` statement) but also through the `setreceiver` and `getelem` statements; however, in the case of `setreceiver` and `getelem`, the specified table index is *disregarded*.

How Does a Table Work?

To the user, a table consists of a table *name*, table *values*, and the *division factor* and *autoincrement* attributes (*autoincrement* is a flag). At the C level, each table has two more attributes: the current table *index* and the table *length*. The table index is only used in connection with the autoincrement attribute. Both the table length and the division factor are used to extract a table element either from an implicit or a specified table index:

```
extract_index=((index / division_factor) % table_length)
```

If the extraction index is *not* specified (“direct use” of phase tables as argument to statements like `pulse`, `txphase`, etc.), `ct` is used as table index (unless the table has the autoincrement attribute set). The variable `ct` is the default table index.

With the `getelem` statement, the table value is extracted into the specified real-time variable. With the `setreceiver` function, it is extracted into the `oph` real-time variable. Where tables are used “directly,” the table value is extracted into a dedicated real-time variable `tablert` (the address of `LC->tablert`):

```
txphase(t3);
```

is equivalent to

```
getelem(t3,ct,tablert);
txphase(tablert);
```

In other words, statements that take a real-time variable as an argument check whether the real-time variable is a table name or a normal variable, and then take the corresponding actions.

For example, the `txphase` statement can be written as follows[3]:

```
(void) txphase(ph)
codeint ph;
{
   if ((ph >= t1) && (ph <= t60))
```

[3] The actual coding is quite different (a `setquadphase` statement does not actually exist), but the equivalent to this construct is still used in the real software (the statement `setphase90` in `/vnmr/psg/rfchan_device.c`).

```
   {
      getelem(ph,ct,tablert);
      setquadphase(tablert, TODEV);
   }
   else
      setquadphase(ph, TODEV);
}
```

This also works with autoincrementing tables, because this simply causes the `ct` index to be disregarded at a lower level (in the Acode interpreter).

Note that with the standard VNMR software the default table index is `ct` (e.g., whenever a table that is not autoincrementing is used directly). Therefore, there is *no phase cycling during steady-state pulses*. It is strongly recommended to not use the table variables directly, but to use `getelem` with a pseudo `ct` counter:

```
sub(ct,ssctr,v13);
getelem(t1,v13,v1);
getelem(t2,v13,t2);
```

Extracting table values into the *corresponding* real-time variables (`t1` -> `v1`, `t2` -> `v2`, etc.) makes it easier to read and understand the sequence.

11.2 Inline Phase Tables

It is possible to avoid a separate file for tables by defining the tables as (static) arrays in the pulse sequence source file itself between the header (include lines) and the function `pulsesequence`:

```
static int table1[8] = {0,2,1,3,3,1,2,0};
static int table2[16] = {0,0,0,0,1,1,1,1,2,2,2,2,2,3,3,3,3};
```

The name of the static variable is arbitrary. This static array of integers is then converted to a table using the `settable` statement:

```
settable(t1,8,table1);
settable(t4,16,table2);
```

In contrast to the table definition using external files, tables defined through `settable` can also contain negative numbers (i.e., numbers in the full range of signed 16-bit integers, from -32768 to 32767). Division factors and the autoincrement attribute have to be set separately (after the call to `settable`), using the `setautoincrement` and `setdivnfactor` statements.

With the exception of cases where negative numbers are required in a table (e.g., if the table is used to set the amplitude of pulsed field gradients), using inline tables in general is not recommended because of the following reasons:

- The definition syntax is more complicated (there is `lint` checking on it, though),
- No shorthand syntax is available.
- Division factors and autoincrement attribute cannot be set together with the table definition (it is less obvious what the tables actually are if division factors are involved).
- Changing phase tables requires recompiling the pulse sequence.

11.3 Table Math

Tables cannot be created or defined only as static objects. Scalar and vector math are possible with tables. Four statements are provided for scalar operations on tables:

```
tsadd(tablename,scalarvalue,modulovGalue);
tssub(tablename,scalarvalue,modulovalue);
tsmult(tablename,scalarvalue,modulovalue);
tsdiv(tablename,scalarvalue,modulovalue);
```

The argument `tablename` is one of the table addresses, `t1` to `t60`. The operation (addition, subtraction, multiplication, division) is applied to every value in the table. At the end, a modulo function is applied to the result, unless the last argument (`modulоval`) is set to zero, in which case the numeric range of the resulting table often exceeds the numeric range of the table prior to the execution of the scalar operation (in the vast majority of the cases, this has no adverse effects due to the implicit modulo operation that occurs when using the table). Obviously, the divisor (second argument) to `tsdiv` cannot be zero, because division by zero would lead to mathematical overflow.

There is also a set of table-to-table (vector-to-vector) statements available for more complex math with tables:

```
ttadd(target_table,operand_table,modulovalue);
ttsub(target_table,operand_table,modulovalue);
ttmult(target_table,operand_table,modulovalue);
ttdiv(target_table,operand_table,modulovalue);
```

The mathematical operation in these statements is applied to each “equivalent” pair of numbers in two tables, and the result is stored in one of the two specified tables. It is not possible to generate new tables using table math. Also, the operand table cannot be longer than the target table, because table math cannot alter the length of tables.

The following example should illustrate the functionality of table-to-table math. Let’s assume we have defined the following two tables:

```
t1 = 0 0 0 0 2 2 2 2
t2 = 0 1 2 3
```

In the pulse sequence, we now apply the following function:

```
ttadd(t1, t2, 0);
```

This causes each element in table `t2` to be added to the corresponding element in table `t1`. If the operand table is shorter than the target table, the shorter table is expanded to the same length before performing the operation. The result of the above function is the following table `t1`:

```
t1 = 0 1 2 3 2 3 4 5
```

(no modulo value was specified; therefore, values above 3 are obtained). The other table-to-table operations use the same working principle. For `ttdiv`, the operand table must not contain zeroes, of course, because this would lead to math overflow.

There is a fundamental difference between table math and real-time math using variables. Different from the real-time variables, table math occurs in the host computer, when executing the pulse sequence code. That means that a table may not be changed *within* a pulse sequence. Once a table has been built into the Acode (see below), it cannot be modified. In fact, a table cannot be modified using table math after

it has been used the first time (be it as argument to a pulse or phasing function, through `getelem` or `setreceiver` calls)—this will lead to a run-time error message.

This precludes the alteration of tables within composite pulses and similar pulse sequence elements (like BIRD pulses) that use multiple closely related (“parallel”) phase cycles. For such cases it is either necessary to define separate tables or to use real-time math to derive the related phases (see also Section 11.6, “Combining the Best of the Two Worlds,” on page 129).

11.4 Phase Tables in the Acode

Tables are incorporated into the instruction segment of the Acode *at the point where they are used for the first time*. Unused tables are discarded. The table values are stored as 16-bit integers. Evidently, long tables can occupy a lot of Acode space.

Besides the table values, four more 16-bit integers per table define the table length, the autoincrement flag, the division factor, and the table index. The table name itself is not at Acode level. Tables are identified by their address in the instruction segment (actually the offset of the first attribute). The table size attribute also permits the Acode interpreter to calculate the offset to the next instruction after the table.

Let’s first have a look at the code generated by a `getelem` statement (extracted from the Acode for a `noesy` experiment):

```
369   36325739 ASSIGNFUNC   zero v5
372   366260105TABLE 261size 2, autoinc 0, divn_ret 1, ptr 0
                0   2
379   373267106TASSIGN table 261 ct v1
383   377271105TABLE 272size 2, autoinc 0, divn_ret 16, ptr 0
                0   2
390   384278106TASSIGN table 272 ct v2
394   388282105TABLE 283size 4, autoinc 0, divn_ret 2, ptr 0
                0   1   2   3
403   397291106TASSIGN table 283 ct v3
407   40129529 ADDFUNC      v1 v5 v1
411   40529929 ADDFUNC      v1 v2 oph
415   40930329 ADDFUNC      v3 oph oph
419   413307105TABLE 308size 2, autoinc 0, divn_ret 8, ptr 0
                0   1
426   420314106TASSIGN table 308 ct v4
430   42431829 ADDFUNCo     ph v4 oph
434   42832229 ADDFUNC      v1 v4 v1
438   43232629 ADDFUNC      v2 v4 v2
442   43633029 ADDFUNC      v3 v4 v3
```

Each of the four `getelem` calls in this part of the pulse sequence generates a `TABLE` instruction (code `105`), which inserts the table into the Acode as described above, followed by a `TASSIGN` instruction (code `106`), which extracts a table value into a real-time variable. `TASSIGN` is followed by the table address, the table index (`ct` in this case), and the target address (a real-time variable in `LC`). For each of the tables, the `TABLE` instruction is inserted at the point where a table is first referred to.

The `setreceiver` statement generates the same kind of Acode (except that the target address is `LC->oph`):

```
846  840  734  105  TABLE 735   size 4, autoinc 0, divn_ret 1, ptr 0
                0   1   2   3
855  849  743  106  TASSIGN       table 735 ct oph
```

Even using tables directly (here as phase arguments to a `simpulse` statement) does not require a new Acode instruction. The target address is `LC->tablert` in this case.

```
603  597  491  150   HighSpeedLINES    (void)
606  600  494  105   TABLE 495   size 4, autoinc 0, divn_ret 8, ptr 0
                           1   2   3   0
615  609  503  106   TASSIGN     table 495 ct tblrt
619  613  507  16    SETPHAS90         CH1 tblrt
622  616  510  105   TABLE 511    size 4, autoinc 0, divn_ret 1, ptr 0
                          0   1   2   3
631  625  519  106   TASSIGN       table 511 ct tblrt
635  629  523  16    SETPHAS90          CH2 tblrt
638  632  526  150   HighSpeedLINES      (void)
641  635  529  150  HighSpeedLINES   XOFF
644  638  532  150  HighSpeedLINES   RXOFF DECRG
647  641  535  150  HighSpeedLINES   RXOFF DECRG
650  644  538  151  EVENT1_TWRD      1.000 usec
652  646  540  150  HighSpeedLINES   RXOFF DECRG DEC
655  649  543  151  EVENT1_TWRD      7.925 usec
657  651  545  150  HighSpeedLINES   RXOFF TXON DECRG DEC
660  654  548  151  EVENT1_TWRD      27.725 usec
662  656  550  150  HighSpeedLINES   RXOFF DECRG DEC
665  659  553  151  EVENT1_TWRD      7.925 usec
667  661  555  150  HighSpeedLINES   RXOFF DECRG
670  664  558  150  HighSpeedLINES   DECRG
673  667  561  150  HighSpeedLINES   (void)
```

11.5 Tables vs. Real-Time Calculations

In this section, we make a point-by-point comparison of tables and real-time calculations. The comparison is illustrated by examples.

Point-to-Point Comparison

Tables and real-time calculations can be compared on length, complexity, Acode space required, understandability, constructionally, and changeability.

Possible Length, Complexity

Real-time calculations permit calculating phase cycles that are virtually unlimited in length (a phase cycle with 1024*1024 steps is no problem!), but limited in complexity (“arbitrary” phase cycles may require an enormous amount of real-time calculations that are difficult to construct and understand). *Tables* are limited in length (up to 8192 steps per table, or about 9500 steps maximum per pulse sequence in total, excluding division factors), but of unlimited complexity.

Acode Space Consumption

Real-time calculations are inefficient for relatively short and simple phase cycles like `0 2 1 3 3 1 2 0`, but long phase cycles can often be constructed with relatively few math statements. *Tables* are efficient as long as they are short. Long tables can consume a lot of Acode space. Acode space efficiency can be crucial in 3D and 4D experiments (although these normally have very short phase cycles).

Easy to Understand? Easy to Construct?

Finding out the phase cycle generated by *real-time calculations* is often not easy. Constructing a real-time math algorithm for a given, non-trivial phase table is definitely a difficult task for non-specialists. Given a complete (phase) table in literature, a *table file* is trivial to construct, and it also is trivial to see what the actual numeric sequence is. On the other hand, with longer tables it can be *more difficult* to understand what the *underlying phase cycling algorithms* are than with real-time math, because with the latter such algorithms can often be directly coded. For the designer of a new pulse sequence, real-time math may be the more adequate choice. NMR scientists often think more in terms of phase-cycling algorithms, rather than final phase tables (of course, in publications it is easier to simply print out phase tables rather than having to explain the various phase cycling elements and algorithms).

Easy to Change?

Changing a phase cycle generated by *real-time math* normally requires analyzing and rebuilding the entire algorithm and can be time-consuming. Changing a phase cycle from full *tables* first requires understanding and analyzing the underlying algorithms, and then changing (all) the tables. With long tables, this can be as complicated as changing real-time calculations, maybe even more difficult sometimes.

Comparison by Examples

To emphasize the differences, and in particular the advantages and disadvantages of the two methods for generating phase cycles, a pulse sequence with relatively long phase cycles (not excessive, though), `inadqt.c`, has been selected for the following comparison (*the sequence was simplified for the purpose of this chapter*).

Using Calculations Only

Let’s first have a look at a sequence that is written with phase calculations only:

```
pulsesequence()
{
   double tau = 1.0 / (4.0 * getval("jcc"));
   int phase = (int) (getval("phase") + 0.5);

   mod4(ct, v3);
   hlv(ct, v9);
   hlv(v9, v9);
   assign(v9, v1);
   hlv(v9, v9);
   assign(v9, v2);
   hlv(v9, v9);
   hlv(v9, v9);
   mod2(v9, v9);/* v9 = F2 quad. image suppression */
   dbl(v1, v1);/* v1 = suppresses artifacts from
                        imperfections in 1st 90 deg. pulse */
   add(v9, v1, v1);
   assign(v1, oph);
   dbl(v2, v8);
   add(v9, v2, v2);/* v2 = suppresses artifacts from
                        imperfect 180 refocusing pulse */
   add(v8, oph, oph);
   dbl(v3, v4);
```

```
   add(v3, v4, v4);
   add(v3, v9, v3);
   add(v4, oph, oph);/* v3 = selects DQC during t1
                         evolution period */
   assign(zero, v10);/* v10 = F1 quadrature */
   if (phase == 2)
      incr(v10);
   add(v10, oph, oph);

   status(A);
      hsdelay(d1);
   status(B);
      stepsize(45.0, TODEV);
      xmtrphase(v10);
      rgpulse(pw, v1, rof1, 0.0);
      delay(tau);
      rgpulse(2.0*pw, v2, rof1, 0.0);
      delay(tau);
   status(C);
      rgpulse(pw, v9, rof1, 0.0);
      xmtrphase(zero);
      delay(d2);
      pulse(pw, v3);
   status(D);
}
```

The phase calculation section is rather long. There are five different phases, and the phase cycle length is 64 steps. Even an experienced spectroscopist could spend many minutes figuring out what the phase cycle is for the various pulses. In the Acode, the phase calculations occupy 70 words, which isn't too bad for phase cycles of this length.

```
275   26916398 NextSCan
276   27016437 MOD4FUNCct v3
279   27316734 HLVFUNCct v9
282   27617034 HLVFUNCv9 v9
285   27917339 ASSIGNFUNCv9 v1
288   28217634 HLVFUNCv9 v9
291   28517939 ASSIGNFUNCv9 v2
294   28818234 HLVFUNCv9 v9
297   29118534 HLVFUNCv9 v9
300   29418836 MOD2FUNCv9 v9
303   29719133 DBLFUNCv1 v1
306   30019429 ADDFUNCv9 v1 v1
310   30419839 ASSIGNFUNCv1 oph
313   30720133 DBLFUNCv2 v8
316   31020429 ADDFUNCv9 v2 v2
320   31420829 ADDFUNCv8 oph oph
324   31821233 DBLFUNCv3 v4
327   32121529 ADDFUNCv3 v4 v4
331   32521929 ADDFUNCv3 v9 v3
335   32922329 ADDFUNCv4 oph oph
339   33322739 ASSIGNFUNCzero v10
342   33623029 ADDFUNCv10 oph oph
346   340234 6 APBOUT2 items 0xa511 0xb57c
350   344238150HighSpeedLINESDECUP
353   347241151EVENT1_TWRD1500 msec
355   349243150HighSpeedLINESDECUP
358   35224668 PHASESTEP CH190 units (45.00 degrees)
361   35524965 SETPHASE CH1v10
364   35825216 SETPHAS90 CH1v1
367   361255150HighSpeedLINESRXOFF DECUP
370   364258151EVENT1_TWRD40.000 usec
```

```
372   366260150HighSpeedLINESRXOFF TXON DECUP
375   369263151EVENT1_TWRD10.400 usec
377   371265150HighSpeedLINESRXOFF DECUP
380   374268150HighSpeedLINESDECUP
383   377271152EVENT2_TWRD6 msec + 250 usec
386   38027416 SETPHAS90CH1 v2
389   383277150HighSpeedLINERXOFF DECUP
392   386280151EVENT1_TWRD40.000 usec
394   388282150HighSpeedLINESRXOFF TXON DECUP
397   391285151EVENT1_TWRD20.800 usec
399   393287150HighSpeedLINESRXOFF DECUP
402   396290150HighSpeedLINESDECUP
405   399293152EVENT2_TWRD6 msec + 250 usec
408   402296150HighSpeedLINESDECUP
411   40529916 SETPHAS90CH1 v9
414   408302150HighSpeedLINESRXOFF DECUP
417   411305151EVENT1_TWRD40.000 usec
419   413307150HighSpeedLINESRXOFF TXON DECUP
422   416310151EVENT1_TWRD10.400 usec
424   418312150HighSpeedLINESRXOFF DECUP
427   421315150HighSpeedLINESDECUP
430   42431865 SETPHASECH1 zero
433   42732116 SETPHAS90CH1 v3
436   430324150HighSpeedLINESRXOFF DECUP
439   433327151EVENT1_TWRD10.000 usec
441   435329150HighSpeedLINESRXOFF TXON DECUP
444   438332151EVENT1_TWRD10.400 usec
446   440334150HighSpeedLINESRXOFF DECUP
449   443337150HighSpeedLINESDECUP
452   446340150HighSpeedLINESDECUP
455   44934316 SETPHAS90CH1 zero
458   45234616 SETPHAS90CH2 zero
461   455349150HighSpeedLINESRXOFF DECUP
464   458352151EVENT1_TWRD10.000 usec
466   460354150HighSpeedLINESDECUP
469   463357152EVENT2_TWRD122 usec + 350 nsec
472   46636090 SETInputCardMode
476   47036499 ACQXX loop np=4096, dwell 80.000 usec
479   473367 7 STartFIFO
480   47436897 HouseKEEPing
481   47536920 BRANCH Offset 159

============================================================
Total code size       483 words /       966 Bytes /       0.9 KB
============================================================
```

The Acode segment for the first FID is 477 words or 954 bytes. We now want to compare this with another version of the same pulse sequence, where all the math statements have been replaced by phase tables.

Using "Pure Tables"

If we remove all the phase calculations and write a table file instead, the pulse sequence text is dramatically simplified. The only phase-related statements left are those that deal with the f_1 quadrature detection (see also Chapter 19, "Multidimensional Experiments," on page 215). Apart from the fact that tables are used instead of phase calculations, the functionality of the following version is exactly the same as with the version above (at least as far as the pulse programmer is concerned).

```
pulsesequence()
{
   double tau = 1.0 / (4.0 * getval("jcc"));
   int phase = (int) (getval("phase") + 0.5);

   loadtable("inadqt");
   setreceiver(t5);
   assign(zero,v6);
   if (phase == 2)
   {
      incr(v6);
      incr(oph);
   }

   status(A);
      hsdelay(d1);
   status(B);
      stepsize(45.0, TODEV);
      xmtrphase(v6);
      rgpulse(pw, t1, rof1, 0.0);
      delay(tau);
      rgpulse(2.0*pw, t2, rof1, 0.0);
      delay(tau);
   status(C);
      rgpulse(pw, t3, rof1, 0.0);
      xmtrphase(zero);
      delay(d2);
      pulse(pw, t4);
   status(D);
}
```

Shorthand syntax allows us to simplify the coding of all the phase tables involved. Unfortunately, only three of the five tables can be shortened using division factors—two tables must be left at full length. These tables also reveal a potential disadvantage of the shorthand syntax: in order to correlate the different phase tables, we have to (mentally) translate the tables back to full length!

Very rarely phase cycling occurs isolated in one table only. Changing the phase of a pulse most likely causes the observe phase, if not also the phase of other pulses, to change as well. It would, therefore, help if the tables were written at full length, on a single line per table, such that the phase cycles could be vertically correlated. This would of course prohibit any shorthand syntax and division factors and would, therefore, make the tables even less efficient (in terms of Acode space consumption). Also, long (full) tables aren't necessarily more readable due to their length (each of them would fill two or four lines if the line length is limited to less than 80 characters).

```
t1 = { ( 0 2 )4  ( 1 3 )4 }4                  /* 1st 90 */
t2 = { 0 1 2 3 1 2 3 0 }8                     /* 180  */
t3 = { 0 1 }32                                /* 2nd 90 */
t4 = ( 0 3 2 1 )8  ( 1 0 3 2 )8               /* 3rd 90 */
t5 = ( 0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 )2
     ( 1 0 3 2 3 2 1 0 3 2 1 0 1 0 3 2 )2     /* oph */
```

In the Acode, the phase tables (at least those with a division factor of 1) occupy a considerable number of words:

```
275     269     163     98      NextSCan
276     270     164     105     TABLE 165size  64, autoinc 0, divn_ret 1, ptr 0
                                 0    3    2    1    2    1    0    3
                                 2    1    0    3    0    3    2    1
                                 0    3    2    1    2    1    0    3
                                 2    1    0    3    0    3    2    1
                                 1    0    3    2    3    2    1    0
                                 3    2    1    0    1    0    3    2
                                 1    0    3    2    3    2    1    0
                                 3    2    1    0    1    0    3    2
345   339   233     106     TASSIGN table 165     ct     oph
349   343   237      39     ASSIGNFUNC      zero    v6
352   346   240       6     APBOUT  2 items 0xa511 0xb57c
356   350   244     150     HighSpeedLINES  DECUP
359   353   247     151     EVENT1_TWRD      1500 msec
361   355   249     150     HighSpeedLINES  DECUP
364   358   252      68     PHASESTEP CH1 90 units (45.00 degrees)
367   361   255      65     SETPHASE        CH1     v6
370   364   258     105     TABLE 259 size 16,autoinc 0,divn_ret 4,ptr 0
                                 0    2    0    2    0    2    0    2
                                 1    3    1    3    1    3    1    3
391   385   279     106     TASSIGN table 259     ct   tblrt
395   389   283      16     SETPHAS90       CH1  tblrt
398   392   286     150     HighSpeedLINES  RXOFF DECUP
401   395   289     151     EVENT1_TWRD     40.000 usec
403   397   291     150     HighSpeedLINES  RXOFF TXON DECUP
406   400   294     151     EVENT1_TWRD     10.400 usec
408   402   296     150     HighSpeedLINES  RXOFF DECUP
411   405   299     150     HighSpeedLINES  DECUP
414   408   302     152     EVENT2_TWRD         6 msec +   250 usec
417   411   305     105     TABLE 306 size 8, autoinc 0,divn_ret 8, ptr 0
                               0    1    2    3    1    2    3    0
430   424   318     106     TASSIGN table 306     ct   tblrt
434   428   322      16     SETPHAS90       CH1  tblrt
437   431   325     150     HighSpeedLINES  RXOFF DECUP
440   434   328     151     EVENT1_TWRD     40.000 usec
442   436   330     150     HighSpeedLINES  RXOFF TXON DECUP
445   439   333     151     EVENT1_TWRD     20.800 usec
447   441   335     150     HighSpeedLINES  RXOFF DECUP
450   444   338     150     HighSpeedLINES  DECUP
453   447   341     152     EVENT2_TWRD         6 msec +   250 usec
456   450   344     150     HighSpeedLINES  DECUP
459   453   347     105     TABLE 348 size 2, autoinc 0,divn_ret 32,ptr 0
                             0    1
466   460   354     106     TASSIGN table 348     ct   tblrt
470   464   358      16     SETPHAS90       CH1  tblrt
473   467   361     150     HighSpeedLINES  RXOFF DECUP
476   470   364     151     EVENT1_TWRD     40.000 usec
478   472   366     150     HighSpeedLINES  RXOFF TXON DECUP
481   475   369     151     EVENT1_TWRD     10.400 usec
483   477   371     150     HighSpeedLINES  RXOFF DECUP
486   480   374     150     HighSpeedLINES  DECUP
489   483   377      65     SETPHASE        CH1   zero
492   486   380     105     TABLE 381 size 64, autoinc 0,divn_ret 1,ptr 0
                             0    3    2    1    0    3    2    1
                             0    3    2    1    0    3    2    1
                             0    3    2    1    0    3    2    1
                             0    3    2    1    0    3    2    1
                             1    0    3    2    1    0    3    2
                             1    0    3    2    1    0    3    2
                             1    0    3    2    1    0    3    2
                             1    0    3    2    1    0    3    2
```

```
561   555   449     106   TASSIGN table 381     ct   tblrt
565   559   453      16   SETPHAS90       CH1  tblrt
568   562   456     150   HighSpeedLINES  RXOFF DECUP
571   565   459     151   EVENT1_TWRD     10.000 usec
573   567   461     150   HighSpeedLINES  RXOFF TXON DECUP
576   570   464     151   EVENT1_TWRD     10.400 usec
578   572   466     150   HighSpeedLINES  RXOFF DECUP
581   575   469     150   HighSpeedLINES  DECUP
584   578   472     150   HighSpeedLINES  DECUP
587   581   475      16   SETPHAS90       CH1   zero
590   584   478      16   SETPHAS90       CH2   zero
593   587   481     150   HighSpeedLINES  RXOFF DECUP
596   590   484     151   EVENT1_TWRD     10.000 usec
598   592   486     150   HighSpeedLINES  DECUP
601   595   489     152   EVENT2_TWRD 22 usec +   350 nsec
604   598   492      90   SETInputCardMode
608   602   496      99   ACQXX loop np=4096, dwell 80.000 usec
611   605   499       7   STartFIFO
612   606   500      97   HouseKEEPing
613   607   501      20   BRANCH          Offset    159

============================================================
Total code size       615 words /      1230 Bytes /       1.2 KB
============================================================
```

In this case, the Acode segment for the first FID is 609 words or 1218 bytes (compared to 477 words or 954 bytes with phase calculations). The Acode size is increased by 254 bytes per FID (28%). This may seem rather marginal, however, we should also consider 2D and 3D experiments.

Furthermore, there are certainly cases of complex pulse sequences that generate a lot of Acode by themselves. In such cases, it is not impossible that phase tables can cause the Acode size to exceed the upper limit (10000 words), in which case a sequence would not execute at all. There are also 1D pulse sequences such as `cccc.c` (composite pulse inadequate) with a large number of different phase cycles of 256 or even up to 1024 steps each. We can certainly imagine cases where the tables themselves would not fit into the available Acode space unless they can be abbreviated.

11.6 Combining the Best of the Two Worlds

Obviously, neither pure calculations nor pure phase tables provide an optimum solution for programming phase cycles. What we are looking for is a way of defining the phase cycling that is at the same time all of the following:

- Easy to understand and analyze.
- Easy to create and compose.
- Easy to change the phase cycling scheme (this is mostly a feature for pulse sequence developers).
- Visualizes the internal phase cycling algorithms.
- Does not use excessive Acode space.

A possible solution to this problem is to use a combination of phase tables and real-time calculations. Why not use phase tables to define the (simple) *phase cycling elements* and real-time math to realize *phase cycling algorithms* (i.e., phase dependencies, synchronous phase variation, etc.)?

It turns out that most phase cycling elements are simple—the most common elements are the following:[4]

```
{ 0 1 2 3 }n
{ 0 2 }n
{ 0 3 2 1 }n
{ 0 1 }n
{ 0 2 1 3 }n
```

Furthermore, most phase cycling algorithms (in the sense of phase cycling *relations*) are almost trivial. In most sequences, it simply involves adding up certain phases in order to obtain the observe phase and to add some "macroscopic" phase cycling to all phase cycles (at least for pulses on the observe channel).

The most striking case for using this solution is in the area of composite pulses. These pulses use phases that are normally simply orthogonal or anti-phase relative to each other, and it would be a real waste to define phase table for each of the pulses. It is much simpler to extract one phase and to calculate the others from that one.

How can we realize this concept? Let's take the same pulse sequence as used above, `inadqt.c`, analyze it, and see whether the new concept is feasible. We first start with the full phase table file from the last version:

```
t1 = { ( 0 2 )4  ( 1 3 )4 }4                    /* 1st 90 */
t2 = { 0 1 2 3 1 2 3 0 }8                       /* 180 */
t3 = { 0 1 }32                                  /* 2nd 90 */
t4 = ( 0 3 2 1 )8  ( 1 0 3 2 )8                 /* 3rd 90 */
t5 = ( 0 3 2 1 2 1 0 3 2 1 0 3 0 3 2 1 )2
     ( 1 0 3 2 3 2 1 0 3 2 1 0 1 0 3 2 )2       /* oph */
```

In order to find out about phase relations, it is best (at least for beginners) to first write out the full phase cycles:

```
t1 = 0 0 0 0 2 2 2 2  0 0 0 0 2 2 2 2  0 0 0 0 2 2 2 2  0 0 0 0 2 2 2 2
     1 1 1 1 3 3 3 3  1 1 1 1 3 3 3 3  1 1 1 1 3 3 3 3  1 1 1 1 3 3 3 3
t2 = 0 0 0 0 0 0 0 0  1 1 1 1 1 1 1 1  2 2 2 2 2 2 2 2  3 3 3 3 3 3 3 3
     1 1 1 1 1 1 1 1  2 2 2 2 2 2 2 2  3 3 3 3 3 3 3 3  0 0 0 0 0 0 0 0
t3 = 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0
     1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
t4 = 0 3 2 1 0 3 2 1  0 3 2 1 0 3 2 1  0 3 2 1 0 3 2 1  0 3 2 1 0 3 2 1
     1 0 3 2 1 0 3 2  1 0 3 2 1 0 3 2  1 0 3 2 1 0 3 2  1 0 3 2 1 0 3 2
t5 = 0 3 2 1 2 1 0 3  2 1 0 3 0 3 2 1  0 3 2 1 2 1 0 3  2 1 0 3 0 3 2 1
     1 0 3 2 3 2 1 0  3 2 1 0 1 0 3 2  1 0 3 2 3 2 1 0  3 2 1 0 1 0 3 2
```

The one fact common to *all* these phase tables is that after 32 steps all phase cycles, including the observe phase `t5`, are repeated with a phase shift of 90 degrees (all tables are incremented by 1). This is the quad image suppression in f_2 (the observed dimension)[5] and is the slowest of the phase cycles in this sequence. It turns out that `t3`

[4] One could even further simplify this list, saying that the most frequently used phase cycling elements are just `{ 0 1 }n` and `{ 0 2 }n`; `{ 0 1 2 3 }n` and `{ 0 2 1 3 }n` are just combinations of the two simpler ones: `{ 0 1 2 3 }n` = `{ 0 1 }n` + `{ 0 2 }(2n)`, and `{ 0 2 1 3 }n` = `{ 0 2 }n` + `{ 0 1 }(2n)`. The cycle `{ 0 3 2 1 }n` is just the reversal of `{ 0 1 2 3 }n`.

[5] This forms part of a CYCLOPS phase cycling (A.D. Bain, *J. Magn. Reson.* **56**, 418 (1984); D.I. Hoult, R.E. Richards, *Proc. Roy. Soc. London Ser.* A **344**, 311 (1975)), which is present only partially in this pulse sequence. The "other half" of the full cycle (`{ 0 2 }64`), which would reduce any axial signal ("center glitch") in f_2, is not present (nor required) in this case.

consists of only this phase cycle. It can, therefore, be regarded as constant phase (0), with quad image suppression added. If we "subtract" (isolate) that phase cycle from all tables, we end up with a much simpler list of tables. The subtracted phase cycle is stored in a new table that we can add to all the other tables using real-time math:

```
t1 = 0 0 0 0 2 2 2 2
t2 = 0 0 0 0 0 0 0 0  1 1 1 1 1 1 1 1  2 2 2 2 2 2 2 2  3 3 3 3 3 3 3 3
t3 = 0
t4 = 0 3 2 1
t5 = 0 3 2 1 2 1 0 3  2 1 0 3 0 3 2 1  0 3 2 1 2 1 0 3  2 1 0 3 0 3 2 1
t6 = { 0 1 }32    /* macroscopic phase cycling (f2 quad image
                     suppression) to be added to t1 - t5 */
```

You might think that we have already reached a sufficient degree of simplification (`t1` and `t2` can be shortened dramatically using division factors), but this is not the end. The observe phase often is simply a function of some (or all) of the pulse phases. In this case, it is obvious that both `t1` and `t4` (and `t3`?) are contained in the observe phase cycle. If we subtract these phases from the observe phase, we obtain

```
t5 = 0 0 0 0 0 0 0 0  2 2 2 2 2 2 2 2  0 0 0 0 0 0 0 0  2 2 2 2 2 2 2 2
```

If we then write this cycle as

```
t5 = 0 0 0 0 0 0 0 0  2 2 2 2 2 2 2 2  4 4 4 4 4 4 4 4  6 6 6 6 6 6 6 6
```

we recognize that it is in fact twice the values of `t2`! This mechanism has been called EXORCYCLE[6]. Each time the phase of a refocusing pulse is altered by 90 degrees, the phase of the observed signal (and hence the receiver phase) changes by 180 degrees. Therefore, the observe phase in the `inadqt` experiment can be expressed as

```
t5 = t1 + 2*t2 + t4
```

In fact, we don't need to define the long observe phase cycle as a table at all. This phase will be calculated in the pulse sequence. The phase table can, therefore, be reduced to the following list:

```
/* inadqt phase tables */

t1 = { 0 2 }4                /* 1st 90 */
t2 = { 0 1 2 3 }8            /* 180 */
t3 =   0                     /* 2nd 90 */
t4 =   0 3 2 1               /* 3rd 90 */
t6 = { 0 1 }32               /* macroscopic phase cycling */

/* phase cycling is performed in the following order:
1) t4 (nt=4):  double-quantum coherence selection (4-step cycle)
2) t1 (nt=8):  suppression of artifacts due to imperfections in
               the first 90 (2-step cycle)
3) t2 (nt=32): EXORCYCLE phase cycling to remove artifacts from
               imperfect spin refocusing (4-step cycle)
4) t6 (nt=64): f2 quad image suppression through macroscopic phase
                cycling (2-step cycle) */
```

In the pulse sequence, we first create a pseudo `ct` counter to also obtain steady-state phase cycling. With this counter as index, we then extract all phases from the tables, calculate the observe phase using the formula we have obtained from the analysis, add the "macroscopic" `{ 0 1 }32` phase cycle (`t6`), and finally adjust the phases for f_1 quadrature detection.[7]

[6] G. Bodenhausen, R. Freeman, and D.L. Turner, *J. Magn. Reson.* **27**, 511 (1977).

```
pulsesequence()
{
   double tau = 1.0 / (4.0 * getval("jcc"));
   int phase = (int) (getval("phase") + 0.5);

   loadtable("inadqt");
   sub(ct, ssctr, v10);        /* pseudo ct counter */
   getelem(t1, v10, v1);       /* extract phases from tables */
   getelem(t2, v10, v2);
   getelem(t3, v10, v3);
   getelem(t4, v10, v4);
   getelem(t6, v10, v6);
   dbl(v2, oph);               /* calculate observe phase */
   add(oph, v1, oph);
   add(oph, v4, oph);
   add(v1, v6, v1);            /* add macroscopic phase cycling */
   add(v2, v6, v2);
   add(v3, v6, v3);
   add(v4, v6, v4);
   add(oph, v6, oph);
   assign(zero, v5);           /* f1 quadrature (phase=1,2) */
   if (phase == 2) incr(v5);
   add(oph, v5, oph);

   status(A);
      hsdelay(d1);
   status(B);
      stepsize(45.0, TODEV);
      xmtrphase(v5);
      rgpulse(pw, v1, rof1, 0.0);
      delay(tau);
      rgpulse(2.0*pw, v2, rof1, 0.0);
      delay(tau);
   status(C);
      rgpulse(pw, v3, rof1, 0.0);
      xmtrphase(zero);
      delay(d2);
      pulse(pw, v4);
   status(D);
}
```

You might argue that the new calculation section complicates the sequence again (compared to the version with “pure tables”). Also, we again are using what you might call “real-time calculations that are difficult to understand.” However, a closer inspection of the above code shows that there is no black magic in the real-time math statements used here. We simply add up the various phases—the math operations involved are almost trivial. Also, you have to realize that (with the exception of some pulsed field gradient and some trivial 1D experiments) the phase cycling to a pulse sequence is usually as important as the pulses and the delays themselves. A phase cycling section as long as the example above (especially if it also nicely visualizes the internal phase cycling algorithms and relations involved) is certainly justified.

[7] Remember that this pulse sequence has been simplified for this chapter. More explanation on how to manipulate the phase cycles in order to achieve quadrature detection in various modes in nD spectra is given in Chapter 19, “Multidimensional Experiments,” on page 215.

Finally, we can double-check how much Acode we are using by combining phase tables and real-time calculations:

```
275   269   163     98   NextSCan
276   270   164     30   SUBFUNC           ct ssctr   v10
280   274   168    105   TABLE 169 size 2, autoinc 0,divn_ret 4, ptr 0
                          0 2
287   281   175    106   TASSIGN table 169      v10    v1
291   285   179    105   TABLE 180 size 4, autoinc 0,divn_ret 8,ptr 0
                          0 1 2 3
300   294   188    106   TASSIGN table 180      v10    v2
304   298   192    105   TABLE 193 size 1, autoinc 0, divn_ret 1, ptr 0 0
310   304   198    106   TASSIGN table 193      v10    v3
314   308   202    105   TABLE 203 size 4, autoinc 0, divn_ret 1, ptr 0
                          0 3 2 1
323   317   211    106   TASSIGN table 203      v10    v4
327   321   215    105   TABLE 216 size 2, autoinc 0,divn_ret 32,ptr 0
                          0 1
334   328   222    106   TASSIGN table 216      v10    v6
338   332   226     33   DBLFUNC           v2   oph
341   335   229     29   ADDFUNC          oph    v1   oph
345   339   233     29   ADDFUNC          oph    v4   oph
349   343   237     29   ADDFUNC           v1    v6    v1
353   347   241     29   ADDFUNC           v2    v6    v2
357   351   245     29   ADDFUNC           v3    v6    v3
361   355   249     29   ADDFUNC           v4    v6    v4
365   359   253     29   ADDFUNC          oph    v6   oph
369   363   257     39   ASSIGNFUNC      zero    v5
372   366   260     29   ADDFUNC           oph    v5   oph
376   370   264      6   APBOUT  2 items 0xa511 0xb57c
380   374   268    150   HighSpeedLINES  DECUP
383   377   271    151   EVENT1_TWRD      1500 msec
385   379   273    150   HighSpeedLINES  DECUP
388   382   276     68   PHASESTEP CH1     90 units (45.00 degrees)
391   385   279     65   SETPHASE        CH1     v5
394   388   282     16   SETPHAS90       CH1     v1
397   391   285    150   HighSpeedLINES  RXOFF DECUP
400   394   288    151   EVENT1_TWRD     40.000 usec
402   396   290    150   HighSpeedLINES  RXOFF TXON DECUP
405   399   293    151   EVENT1_TWRD     10.400 usec
407   401   295    150   HighSpeedLINES  RXOFF DECUP
410   404   298    150   HighSpeedLINES  DECUP
413   407   301    152   EVENT2_TWRD         6 msec +   250 usec
416   410   304     16   SETPHAS90       CH1     v2
419   413   307    150   HighSpeedLINES  RXOFF DECUP
422   416   310    151   EVENT1_TWRD     40.000 usec
424   418   312    150   HighSpeedLINES  RXOFF TXON DECUP
427   421   315    151   EVENT1_TWRD     20.800 usec
429   423   317    150   HighSpeedLINES  RXOFF DECUP
432   426   320    150   HighSpeedLINES  DECUP
435   429   323    152   EVENT2_TWRD         6 msec +   250 usec
438   432   326    150   HighSpeedLINES  DECUP
441   435   329     16   SETPHAS90       CH1     v3
444   438   332    150   HighSpeedLINES  RXOFF DECUP
447   441   335    151   EVENT1_TWRD     40.000 usec
449   443   337    150   HighSpeedLINES  RXOFF TXON DECUP
452   446   340    151   EVENT1_TWRD     10.400 usec
454   448   342    150   HighSpeedLINES  RXOFF DECUP
457   451   345    150   HighSpeedLINES  DECUP
460   454   348     65   SETPHASE        CH1   zero
463   457   351     16   SETPHAS90       CH1     v4
466   460   354    150   HighSpeedLINES  RXOFF DECUP
469   463   357    151   EVENT1_TWRD     1.000 usec
471   465   359    150   HighSpeedLINES  RXOFF TXON DECUP
474   468   362    151   EVENT1_TWRD     10.400 usec
```

```
476   470   364     150   HighSpeedLINES  RXOFF DECUP
479   473   367     150   HighSpeedLINES  DECUP
482   476   370     150   HighSpeedLINES  DECUP
485   479   373      16   SETPHAS90       CH1   zero
488   482   376      16   SETPHAS90       CH2   zero
491   485   379     150   HighSpeedLINES  RXOFF DECUP
494   488   382     151   EVENT1_TWRD     10.000 usec
496   490   384     150   HighSpeedLINES  DECUP
499   493   387     152   EVENT2_TWRD       122 usec +   350 nsec
502   496   390      90   SETInputCardMode
506   500   394      99   ACQXX loop np=4096, dwell 80.000 usec
509   503   397       7   STartFIFO
510   504   398      97   HouseKEEPing
511   505   399      20   BRANCH          Offset    159

================================================================
Total code size       513 words /      1026 Bytes /       1.0 KB
================================================================
```

Here, the Acode segment for the first FID is 507 words (1014 bytes), compared to 477 words, 954 bytes, with pure real-time calculations, and 609 words, 1218 bytes, with “pure tables”: we have almost reached the Acode space efficiency of real-time math, while providing a C code that is superior to both other approaches in terms of clarity and simplicity, while making it easier to change things and experiment with different versions of the phase cycling.

Supposing we decided that the phase cycling on the first pulse is less important than the EXORCYCLE (on the refocusing pulse); therefore, we wanted to cycle the phase of the refocusing pulse before alternating the phase of the first pulse. Instead of changing complex phase tables or rebuilding complicated real-time math, we could simply supply an alternate phase table file:

```
t1 = { 0 2 }16          /* 1st 90 */
t2 = { 0 1 2 3 }4       /* 180 */
t3 =   0                /* 2nd 90 */
t4 =   0 3 2 1          /* 3rd 90 */
t6 = { 0 1 }32          /* macroscopic phase cycling */
```

We wouldn’t even have to change the text of the pulse sequence.

This is, of course, mostly a point for pulse sequence designers; “standard” users seldom alter the phase cycling of a pulse sequence. Also, the sequence of importance and relative order of phase cycling certainly is a non-trivial issue. In cancellation experiments (like double-quantum filtering or double-quantum spectroscopy like the `inadqt` experiment shown above), we always do the main cancellation first, then we would like to start cancelling the biggest artifacts (like the artifacts from off-resonance effects, pulse missettings and imperfections, rf inhomogeneity) and gradually proceed to the minor artifacts (like f_2 quadrature images and axial signals).

The reason for this order is that in the case of changing conditions (such as environmental or instrumental instabilities, or a decaying sample), the cancellation quality suffers if excessive time passes between the two scans that are supposed to subtract the artifact from itself (by alternating its phase). The best chance for a good cancellation is if the two scans follow each other immediately; hence, double-quantum filters and the like are performed with first priority. Minor artifacts leave only very small residual signals, even if they are imperfectly cancelled[8].

Obviously, changing existing pulse sequences (whether they are written with real-time math or with full phase tables) to the new style would require first analyzing the phase cycling, which (as mentioned before) may be a non-trivial task for many users; however, we do not suggest that people rewrite all existing (working) pulse sequences. It's more the idea to provide a new, powerful concept for people that generate *new* pulse sequences.

11.7 Using Tables as Source for Random Numbers

As an alternative to generating random numbers within a single FID using real-time calculations (e.g., for Z-filtering, or for `d1`-randomization, see also Section 10.8, "Real-Time Random Numbers," on page 112), such random numbers can, of course, also be taken from a table. The table solution provides the option of using a fairly large range of random numbers (up to the full or positive range of 16-bit integers, depending on the method of definition). The length of the sequence of random numbers from tables is almost unlimited (up to 8192 numbers), but this length has to be "paid" in Acode space (2 bytes per number). For small numeric values, the method using real-time calculations is superior, because long sequences can be generated with a limited number of math statements, but this solution is probably limited to a linear distribution within a given range.

For non-linear (e.g., Gaussian) distributions or for random numbers with large numeric ranges (e.g., values between 0 and 4095), the table approach is the preferable solution. Linearly distributed random numbers are best generated using a suitable C construct, such as the following for 1024 values between 0 and 4095:

```
static int randomtable[1024];
pulsesequence()
{
  int i;
  if (ix == 1) srandom(getpid());
  for (i = 0; i < 1024; i++)
    randomtable[i] = \
       (int) (4095.999*((double) random())/2147483647.0);
  settable(t1,1024,randomtable);
  ...
```

For non-linear distributions (like a Gaussian distribution of values), a suitable C construct has to be found. For an approximate Gaussian distribution, the data can also be taken from an FID with noise only (no signal). The corresponding procedure would be as follows (1024 numbers, Gaussian distribution between 0 and 4095):

- In VNMR, set the following parameters and then acquire an FID:
  `pw=0` (avoid signal)
  `sw=100000` (the noise increases with `sw`)
  `np=n` (where `n` is the desired number of random numbers)
  `nt≥1` (for 16-bit ADC) or `nt≥4` (for 15-bit ADC) or `nt≥256` (for 12-bit ADC)

[8] The question of the optimum order in the phase cycling depends on the relative intensity of the artifacts (which may be a question of parameter selection, or a question of rf and probe quality), and on the environmental and instrumental stability. There is no global recipe for the phase cycling order. It can be assumed that the VNMR pulse sequences have been optimized in this respect; however, depending on the conditions and the instrument configuration, an alternate phase cycling order may sometimes improve the results.

```
ss=0
```
increase the `gain` such that the ADC is almost filled

- In UNIX (Bourne shell script):

```
#!/bin/sh
cd
cd vnmrsys/expn/acqfil
echo -n "t50 =   " > $HOME/vnmrsys/tablib/randomtable
od -iw2 fid +74 | awk ‘
  {  if ((NR > 0) && (((NR - 1) % 8) == 0)) printf(“\n\t”)
     printf(" %7d",($2+32768.0)/65535.0*4095.999)
  }’ >> $HOME/vnmrsys/tablib/randomtable
```

This generates a table file (table `t50`) with eight numbers per line.

Chapter 12. AP Bus Traffic

States that are driven by the fast bits can change instantaneously (at least as far as the programming goes), they can be set and unset within time intervals as short as 200 nanoseconds, with a timing precision of 25 (or 100) nanoseconds. It would be nice if all spectrometer states could be changed that quickly! Unfortunately, with instruments and experiments becoming more and more complex, for a fully equipped modern spectrometer this would require over a thousand fast lines. Also, pulse programming would be rather inefficient considering the fact that with most time events only one or two of the fast bits would be altered, and yet we would have to feed the pulse programmer with over 100 bytes for every FIFO word (and mostly the same bytes over and over again). The system designers, therefore, divided the devices into three classes:

- Devices that require very fast switching (down to 200 nanoseconds for switching in both directions): typically rf gates and 90-degree phase shifts (see Section 9.2, “Fast Bits,” on page 88).
- Devices that need to be switched with accurate timing, but not necessarily at the speed of the fast bits, devices that would take longer anyway to switch to another state, and devices that are not switched on and off in rapid succession, but rather keep their state for many time events (or the entire sequence). These are the units for which the AP (analog port) bus was built.
- Devices that do not need to be synchronized with the pulse sequence, such as the sample changer, spinner control, sample insert and eject, and the like (these aren’t even handled by the pulse programmer, see Chapter 13, “Acquisition CPU Communication,” on page 145).

It was decided to limit the fast bits to those devices that really need the speed of the state bits. Most other devices are addressed by the AP bus.

12.1 What Is the AP Bus

As the name indicates, the AP bus is a true bus structure. In computer terminology, a *bus* involves a linear structure (a cable, sometimes a computer backplane) with several parallel lines (16 in the case of the AP bus). The lines are used to transmit address and control information as well as data to a (potentially large) number of devices that are all hooked up to the AP bus with special interface chips (the AP bus chip). Some devices (like attenuators and the linear amplifiers on UNITY systems) share a common interface to the AP bus, the AP interface card.

The information sent over the AP bus is mostly numeric; a maximum of 8 bits (1 byte) of information can be transmitted per AP word. Depending on the device, two different transmission modes can be used:

- Direct binary information, as used by attenuators, phase and amplitude modulators, shim or pulsed field gradient DAC values, etc. Depending on the way the hardware works (positive or negative logic), this can be normal or bit-inverted (negated) binary information.
- BCD (binary coded decimal) information. Here, every digit of a decimal number is transmitted in a separate binary AP word. This may sound inefficient (only 4 out

of 8 data bits are used per AP word), but it is adequate for some devices like the PTS frequency synthesizers and other similar devices (decoupler modulator [dmf], offset synthesizers, etc.). These devices operate in (frequency) *decades* and would otherwise have to generate the decimal information internally.

Apart from numeric values, the information transmitted through the AP bus can also be of simple binary nature, like the "command" or flag that sets a linear amplifier into CW mode or back into pulsed mode, or a "strobe", a command that causes devices to carry out previously transmitted numeric information. The strobe is used for small-angle phase shifting on the UNITY*plus* (where the phase value is transmitted in two words but only carried out when the second word is received), or in the case of PTS frequency synthesizers with latching: With these, all decades are first transmitted in BCD mode, but the frequency only changes when the strobe is received in a separate AP word[1].

For the vast majority of the cases, the AP bus operates in "indirect" mode (i.e., the first word sent over the AP bus contains the address and the following words contain the information to be transferred to the device). In this mode, the address fills 12 bits of the first AP word. Thanks to this long address, the number of devices that can be driven by the AP bus is virtually unlimited (4096 addresses are available). In very few cases (e.g., small-angle phase shifting on UNITY and earlier systems), the AP bus is used in direct addressing mode, where address and data are transmitted with the same AP word. This mode is limited to 16 devices (4 address bits).[2]

The AP words pass the pulse programmer within normal FIFO words. The difference to a time event is that the timing part of the FIFO word in this case contains the AP bus bits instead of time and time base. This part of the FIFO word is fed either into the timer(s) or into the AP bus, depending on a special bit within the FIFO word that differentiates timer words and AP words. The definition of the AP bus is such that every word must be held on the bus lines for 1 microsecond (UNITY*plus*) or 2 microseconds (UNITY and earlier systems) to allow for the devices to decode the address and read the information off the bus. AP words are, therefore, timed (accurately) by the pulse programmer. It takes 150 nanoseconds to decode the timer or AP word and initialize the timer circuitry; the total time per AP word is therefore 1.15 microseconds on a UNITY*plus* and 2.15 microseconds on all earlier systems. This is the AP bus cycle time.

The time to transmit information to any device is always a multiple of the bus cycle time. For instance, it takes 9 AP words to change the frequency on a PTS synthesizer on a UNITY*plus*. Therefore, it takes 10.35 microseconds to change a frequency with the `offset` statement. Complete tables for the number of FIFO and AP words involved, as well as for time requirements are given in the manual *VNMR User Programming.*

With respect to the fast bits, AP words behave like normal (fixed length) time events. The fast bit part of the FIFO words is filled with the current settings of the fast bits.

---

[1] On PTS synthesizers without latching, each decade changes (or keeps) its frequency value.

[2] The addressing is actually more complex than that. There are 16 "board addresses" (4 bits) that are shared between devices operating in direct mode (not used on UNITY*plus*) and devices that are addressed via AP bus chip. For each of the board addresses (AP bus chips), there are 256 register subaddresses, which allows for a variety of functions to be performed. Each AP bus chip can address many devices.

Fast bits are slightly different on older systems with an *output board* (63-word loop FIFO). On these systems, the fast bits maintain the settings of the *last executed time event*. If an AP word would immediately follow a pulse without post-pulse delay, that pulse would be prolonged by the duration of the AP bus event(s).

To avoid this, AP functions that generate AP words also generate a 200-nanosecond delay preceding the AP words. The total duration of AP events on these systems (early VXR systems) is, therefore, 0.2 microseconds longer than on systems with the acquisition control board (UNITY). The programmer can use the `apovrride` statement to turn off this additional 0.2-microsecond delay for the *next* AP event (e.g., for the case of multiple AP events in sequence or if the fast bits were already set correctly from the previous delay).

The AP bus overall is a fast transport medium: even in BCD mode, the equivalent of over 3.5 million bits (UNITY*plus*) or 1.8 million bits (on earlier systems) can be transmitted per second (in binary mode the transfer rate is twice as great).

12.2 What Devices are Driven by the AP Bus?

As instruments become more complex, the number of devices addressed by AP bus has been growing constantly over the years. In VXR spectrometers, the following devices were driven through the AP bus:

- PTS frequency synthesizers (not on fixed-frequency rf channels).
- Offset synthesizers with fixed frequency and broadband (as opposed to direct synthesis) rf channels.
- Decoupler high-power level (class C decoupler amplifier).
- Decoupler low-power attenuator (class C decoupler amplifier).
- Decoupler modulator frequency.
- Audio filter bandwidth.
- Small-angle phase shifts (with direct synthesis rf only).
- Shim DAC values.

The UNITY spectrometer added a fair number of additional devices; including the following:

- RF power attenuators (one per channel with linear amplifiers), 63 or 79 dB.
- Fine attenuators (optional, mostly used with systems having a solid-state module); available for the first two channels only; the same port or AP words can be used to drive an additional (“third”) 63-dB attenuator to facilitate pulse shaping without waveform generators.
- Linear amplifier status (cw and pulse modes).
- Waveform generator shapes and pattern (see also Chapter 16, “Waveform Generators,” on page 163).
- Waveform generator shape selection and time scaling.
- Amplitude of a pulsed field gradient (PFG amplifier).
- Imaging gradient amplitudes.

The AP interface board (of which several versions exist) interfaces rf coarse and fine attenuators and AMT linear amplifiers with the AP bus. The last version of this board (type 3) allow setting the decoupler modulation mode (`dmm` parameter) by the AP bus.

Finally, in the UNITY*plus* the following devices were added to the list:

- Lock power, lock gain, and lock phase.
- Receiver gain.
- Linear amplitude modulator.

The AP interface board is not present in UNITY*plus* spectrometers.

12.3 AP Bus Words in the Acode

AP bus traffic is predominant in the initialization section of the instruction section in the Acode, as can be seen from the following part of the (interpreted) Acode (note that with the exception of rf attenuator commands, the contents of the AP bus words are *not* decoded with the `apdecode` command used here):

```
148   142  36   6   APBOUT   7 items 0xab40 0xbb0d 0x9b00 0xab54
                                     0xbbec 0x9b2f 0x9b8f
157   151  45   6   APBOUT   9 items 0xa720 0xb700 0x97be 0xb7ef
                                     0xb7ae 0xb7fb 0xb7be 0x9701
                                     0xb700
168   162  56   159 TUNE_FREQ        CH1 9 words 0xa720 0xb700 0x97be 0xb7ef
                                     0xb7ae 0xb7fb 0xb7be 0x9701 0xb700
180   174  68   0   NO_OP
181   175  69   16  SETPHAS90       CH1c 0
184   178  72   68  PHASESTEP       CH1 360 units (90.00 degrees)
187   181  75   65  SETPHASE        CH1f 0
190   184  78   59  APChipOUT  APaddr 11, reg 51, +logic, 1 byte
                                          max 79, offset 16, value 55
196   190  84   59  APChipOUT  APaddr 11, reg 150, -logic, 2 bytes
                                          max 4095, offset 0, value 4095
202   196  90   6   APBOUT   4 items 0xab98 0xbbd2 0x9b00 0x9b00
208   202  96   6   APBOUT   2 items 0xab92 0xbba0
212   206  100  6   APBOUT   2 items 0xab9b 0xbb00
216   210  104  6   APBOUT   2 items 0xab91 0xbb80
220   214  108  6   APBOUT   2 items 0xab90 0xbb11
224   218  112  6   APBOUT   8 items 0x8201 0x8214 0x8220 0x8231
                                     0x8241 0x8255 0x8264 0x8271
234   228  122  16  SETPHAS90        CH2c 0
237   231  125  59  APChipOUT  APaddr 11, reg 50, +logic, 1 byte
                                          max 79, offset 16, value 30
243   237  131  59  APChipOUT  APaddr 11, reg 166, -logic, 2 bytes
                                          max 4095, offset 0, value 4095
249   243  137  6   APBOUT   4 items 0xaba8 0xbb2a 0x9b00 0x9b00
255   249  143  6   APBOUT   2 items 0xaba2 0xbbc0
259   253  147  6   APBOUT   2 items 0xabab 0xbb00
263   257  151  6   APBOUT   2 items 0xaba1 0xbb80
267   261  155  6   APBOUT   2 items 0xaba0 0xbb12
271   265  159  6   APBOUT   2 items 0xab48 0xbb01
275   269  163  6   APBOUT   2 items 0xab34 0xbb55
279   273  167  6   APBOUT   2 items 0xab35 0xbb00
283   277  171  6   APBOUT   2 items 0xab4d 0xbb21
287   281  175  6   APBOUT   2 items 0xab43 0xbb01
291   285  179  6   APBOUT   2 items 0xab49 0xbb00
295   289  183  6   APBOUT   2 items 0xab36 0xbb06
299   293  187  6   APBOUT   2 items 0xab9b 0xbb88
303   297  191  6   APBOUT   2 items 0xabab 0xbb88
```

Mostly for AP bus traffic, the code 6 (APBOUT) is used. This Acode instruction is grouped with an Acode word describing the number of AP words that follow (actually, the number of words minus one) and the AP bus part (shown here as hexadecimal value) of each FIFO word that will be generated. The code 59 (APCOUT) is used specifically for rf attenuators and power modulators. The waveform generator instructions (using the AP bus) have a special format.

12.4 Timing Considerations

The timing on the AP bus (or of AP bus FIFO words) is inherent and implicit. No delay needs to be specified for the AP bus traffic to occur. Code like

```
delay(tau);
offset(getval("offset1"),TODEV);
pulse(pw,oph);
```

is acceptable and complete. In general, inherent (hidden) AP delays are negligible compared to most delays (*n*D evolution, J-evolution, refocusing), which are mostly are an order of milliseconds or longer. However, there is a danger of losing coherence due to chemical shift evolution (precession) in (hypothetical) constructs like the following:

```
pulse(pw,v1);
delay(tau);
pulse(2.0*pw,v2);
delay(tau);
offset(getval("offset1"),TODEV);
rlpower(tpwr,TODEV);
pulse(pw,v3);
```

The AP bus delays after the second refocusing delay are over 26 microseconds on a UNITY with latching in the frequency synthesis (8.05 microseconds on a UNITY*plus*). This makes the second refocusing interval longer by an amount that can cause considerable precession due to chemical shift evolution with large spectral windows. With any refocusing, make sure the two refocusing delays are *absolutely equal*. Therefore, the hidden AP delays in the above construct should be added to the first delay or (more correctly) be subtracted from the delay adjacent to the AP events. At the same time, symmetry with respect to the pre- and post-pulse delays should be ensured:

```
rgpulse(pw,v1,rof1,0.0);
delay(tau-rof1);
rgpulse(2.0*pw,v2,rof1,0.0);
delay(tau-rof1-15.05e-6-2.15e-6);
offset(getval("offset1"),TODEV);
rlpower(tpwr,TODEV);
rgpulse(pw,v3,rof1,0.0);
```

In this example, the programmer has looked up the length of the hidden AP delays in the tables in the manual *VNMR User Programming* (note that at the level of the pulse sequence, all time events are defined in seconds, irrespective of the VNMR parameter definition involved). Because these delays are specific to certain types of hardware (a UNITY without latching on the observe PTS synthesizer, in this case), this construct (and with it the pulse sequence) becomes specific to an instrument and cannot be ported to other systems without rechecking (and maybe adjusting) the delay corrections.

Therefore, even if the example of code above is correct in every aspect, it should be regarded as bad programming practice! VNMR software provides the tools for making this *correct code for every architecture:*

```
rgpulse(pw,v1,rof1,0.0);
delay(tau-rof1);
rgpulse(2.0*pw,v2,rof1,0.0);
delay(tau-rof1-OFFSET_DELAY-POWER_DELAY);
offset(getval("offset1"), TODEV);
rlpower(tpwr,TODEV);
rgpulse(pw,v3,rof1,0.0);
```

The file `/vnmr/psg/apdelay.h` defines a series of such (pseudo) constants (macros that select the right constant for the current architecture) to be used instead of fixed delays. Table 7 gives a list of the predefined AP delay constants. The AP delays

Table 7. Predefined AP bus delay constants.

Constant Name	*Comments*
`POWER_DELAY`	coarse and fine attenuators with linear amplifiers (`power`, `rlpower`)
`SAPS_DELAY`	small-angle phase shifting (`xmtrphase`, `dcplrphase`, `dcplr2phase`, `dcplr3phase`)
`OFFSET_DELAY`	`offset`, `obsoffset`, `decoffset`, `dec2offset`, `dec3offset` (PTS synthesizer without latching)
`OFFSET_LTCH_DELAY`	`offset`, `obsoffset`, `decoffset`, `dec2offset`, `dec3offset` (PTS synthesizer with latching)
`WFG_START_DELAY`	starting one waveform generator (start of `shaped_pulse`, `decshaped_pulse`, `dec2shaped_pulse`, `dec3shaped_pulse`)
`WFG_STOP_DELAY`	stopping one waveform generator (at end of `shaped_pulse`, `decshaped_pulse`, `dec2shaped_pulse`, `dec3shaped_pulse` (0.0 on UNITY*plus*)
`WFG2_START_DELAY`	starting two waveform generators (e.g., start of `simshaped_pulse`)
`WFG2_STOP_DELAY`	stopping two waveform generators (e.g., at end of `simshaped_pulse`) (0.0 on UNITY*plus*)
`WFG3_START_DELAY`	starting three waveform generators (e.g., start of `sim3shaped_pulse`)
`WFG3_STOP_DELAY`	stopping three waveform generators (e.g., at end of `sim3shaped_pulse`) (0.0 on UNITY*plus*)
`PRG_START_DELAY`	`obsprgon`, `decprgon`, `dec2prgon`, `prg_dec_on`
`PRG_STOP_DELAY`	`obsprgoff`, `decprgoff`, `dec2prgoff`, `prg_dec_off` (0.0 on UNITY*plus*)
`SPNLCK_START_DELAY`	start of `spinlock`, `decspinlock`, `dec2spinlock`, `genspinlock`
`SPNLCK_STOP_DELAY`	end of `spinlock`, `decspinlock`, `dec2spinlock`, `genspinlock` (0.0 on UNITY*plus*)
`SPN2LCK_START_DELAY`	start of `gen2spinlock`
`SPN2LCK_STOP_DELAY`	end of `gen2spinlock` (0.0 on UNITY*plus*)
`GRADIENT_DELAY`	`rgradient`, `vgradient`; start and end of `zgradpulse`

associated with the setting of the fine attenuators using the `pwrf` and `rlpwrf` statements (6.45 microseconds on UNITY, 4.6 microseconds on UNITY*plus* systems) are *not* covered by constants defined in `/vnmr/psg/apdelay.h`.

Another place it is strongly recommended to compensate for "hidden" AP bus delays is in the evolution time in *n*D experiments, where constructs like the following one can be used (at the same time we compensate for the precession during two adjacent 90-degree pulses):

```
rgpulse(pw,v1,rof1,0.0);
if ((d2- rof1-SAPS_DELAY-4.0*pw/3.14159) > MINDELAY)
    delay(d2-rof1-SAPS_DELAY-4.0*pw/3.14159);
xmtrphase(zero);
rgpulse(pw,v2,rof1,0.0);
```

Note that UNITY and VXR systems were normally equipped with PTS synthesizers *without* latching, and on UNITY*plus* and UNITY*INOVA* systems, OFFSET_DELAY and OFFSET_LTCH_DELAY have identical values; therefore, using OFFSET_DELAY covers the vast majority of the hardware configurations.

Note also that status changes can cause hidden AP delays to occur (see the manual *VNMR User Programming*).

Chapter 13. Acquisition CPU Communication

Apart from the pulse programmer, the acquisition CPU has yet an other channel to communicate with devices in the spectrometer. It can use one of its RS-232 ports, and it can use the host I/O bus that is also used to transfer information to and from the pulse programmer. For the latter, the CPU communicates with the automation control board (see Chapter 7, “Digital Components,” on page 65).

Because it is not passing the pulse programmer, such communication with acquisition devices can only be loosely coordinated with the timing of the acquisition. It has to happen when the FIFO (the pulse programmer) is stopped (or hasn’t been started yet). This is appropriate for devices that typically have to be set or regulated once per sample (sample changer, magnet leg pneumatics, etc.), where the reaction time is seconds or minutes rather than microseconds (VT controller) or that would not possibly change during the pulse sequence (lock power, gain, phase).

13.1 Regular Pulse Sequence Communication

What exactly are the tasks that are achieved this way? Let’s first list the devices for VXR and UNITY spectrometers:

- Communication to and from the VT controller is done from one of the two serial (RS-232) ports of the acquisition CPU (the other port being used for acquisition diagnostics purposes).
- The automation control board is the communication link to the sample changer (ASM-100 or SMS), the magnet leg pneumatics (eject/insert, slow drop period, bearing control), the spinner control circuitry, the lock parameters (power, gain, phase), and finally the receiver gain setting. This board is connected with the acquisition CPU via the same bus (the host I/O bus) as the pulse programmer.

On the UNITY*plus* spectrometer, the tasks that were achieved via these channels have changed slightly:

- Communication with the VT controller has been moved to the new automation control board.
- Lock parameters (power, gain, phase) and receiver gain are now set via the AP bus.
- The tasks left for the automation control board include the magnet leg pneumatics control, and interaction with slow devices that involve two-way communication, like the sample changer (ASM-100 or SMS) and the VT controller[1].

[1] The interfaces to the sample changer and the VT controller are standard serial (RS-232) ports. For setup and diagnostics purposes, these devices can also be operated off-line, with a “dumb” terminal. Both devices should not be disconnected as long as the software is configured for them; otherwise, because of the two-way communication scheme, acquisitions can abort with error messages. Before disconnecting or switching off either of these two devices, always reconfigure VNMR by either using `config`, or by setting `vttype=0` (to disable VT control) or `traymax='n'` (to disable the sample changer).

13.2 Diagnostics and Error Output

One of the two serial (RS-232) ports on the acquisition CPU can be used for diagnostics purposes by hooking up a terminal. Different types of information can be obtained this way, depending on the acquisition bootup selector switch (a thumbwheel beside the acquisition CPU cardcage on VXR and UNITY spectrometers, or a toggle switch on the UNITY*plus* automation control board):

- If the switch is in position “0”, error output is obtained only for serious incidents (CPU hangup and the like). This information may be valuable to software and hardware engineers for debugging purposes, but is seldom used in the field.
- If the switch is in position “1” or “2”, continuous diagnostics output is made during an experiment. Position “1” creates output for terminals with cursor addressing (typically Televideo 915 or 925 terminals), position “2” is for “dumb” terminals. With dumb terminals, the output appears line by line; with addressable terminals, the output is organized as full-page display and is, therefore, easier to read and follow.

It is important to know that diagnostics (as opposed to error) output is generated during the housekeeping delay at the end of each scan. If the bootup selector switch is in non-zero position, the diagnostics output is generated, regardless whether a terminal is connected or not. This makes the length of the housekeeping delays *strictly non-deterministic!*

Hooking up a terminal may cause a further slow-down if the communication speed or the terminal are slow (low baud rate or handshaking, due to a slow terminal). Typically, with non-zero setting on the bootup selector, only about 5 scans can be performed per second, because of long housekeeping delays. Position “0” should, therefore, be the normal setting[2].

Even with the acquisition bootup selector switch in position “0”, the residual housekeeping delay is still in the order of milliseconds. If an acquisition should be faster than that, this is only possible by doing multi-FID scans (using multiple `acquire` calls in the pulse sequence in connection with the `nf` and `cf` parameters). This way, we can acquire several FIDs with a single housekeeping delay. This technique is used extensively in multiecho imaging experiments.

[2] The position of the bootup switch is checked with every `go`; therefore, it is *not required* to reboot the acquisition CPU when changing the bootup selector setting.

Chapter 14. Repeating Events

Repetitious events in a pulse sequence may be defined better by C loops, real-time loops, or hardware loops. Each type of loop is discussed in this chapter.

14.1 C Loops

Any C construct can basically be used in a pulse sequence; therefore it is *formally* correct to generate successive, repetitive real-time events in a pulse sequence using a C loop:

```
int i;
for (i = 0; i < 1000; i++)
{
  rgpulse(pw,zero,rof1,0.0);
  rgpulse(pw,one,rof1,0.0);
}
```

This is not only correct C code, but it also creates correct Acodes that should execute properly—maybe! What is the problem?

- This construct is extremely inefficient in that it creates huge amounts of Acode (1000 copies of the same sequence of four time events with `HSLINES` instructions in-between), such that there can easily be an Acode overflow (10000 Acode words per FID maximum).
- Depending on the length of the time events involved (`pw` and `rof1` in the above example), the Acode interpretation could become the rate-determining step, in which case the sequence would abort with a “FIFO underflow” message.

This does not mean that C loops are “forbidden” in pulse sequences, but in general they should not be used in sections that generate Acodes. A good example for an exception to this rule is the `relayh` pulse sequence that adds a variable number of relay periods (depending on the parameter `relay`) to a standard COSY pulse sequence using a C loop (see Section 10.6, “C Constructs and Phase Calculations,” on page 110).

The following code from `/vnmr/psg/shape_pulse.c` is maybe an interesting exception. Here, a `while` loop construct is used to set a real-time variable (`v12`) to the value of a C (integer) variable (`npulses`), without using `initval`. This is particularly interesting for external procedures: `initval` statements reserve a variable completely and also restrict its use. With this construct, the variable can still be used elsewhere in the pulse sequence:

```
assign(zero,v12);              /* v12 = 0 */
mult(three,three,v13);         /* v13 = 9 (RT increment) */
i = 9;                         /* i = v13 (C increment) */
while (npulses)                /* npulses > 0? */
{
   if (npulses >= i)           /* difference > increment? */
   {
      add(v12,v13,v12);        /* add increment */
      npulses -= i;            /* calculate remainder */
   }
   else                        /* difference<increment */
```

```
   {
      divn(v13,three,v13);    /* divide increment (RT) */
      i /= 3;                 /* divide increment (C) */
  }
 }
```

This construct is not optimized for large numbers, but it shows a way to circumvent `initval` calls even in cases where a real-time variable should be initialized to a value that is parameter-based (i.e., not known prior to the function call).

14.2 Real-Time Loops

A much more adequate way to define repeating events is to use *real-time loops* (sometimes also called “software loops”). Different from C, where there are three different looping mechanisms (the `for` loop, the `while` loop, and the `do . . . while` loop), there is only one mechanisms for generating loops in the Acode interpretation, as shown in the following example of an explicitly encoded decoupling sequence on the transmitter channel (WALTZ-16, the text shown includes the table file and a part of a hypothetical pulse sequence):

```
t1 += 2 2 0  0 2 0  2 0 2  2 0 2
      0 0 2  2 0 2  0 2 0  0 2 0
      0 0 2  2 0 2  0 2 0  0 2 0
      2 2 0  0 2 0  2 0 2  2 0 2


initval(tau/(16.0*6.0*pw),v10);   /* # WALTZ cycles */
dbl(two,v12);                     /* 4 */
mult(v12,v12,v12);     /* 16 = composite pulses per cycle */
xmtron();
loop(v10,v11);
  loop(v12,v13);
    txphase(t1);
    delay(pw);
    txphase(t1);
    delay(2.0*pw);
    txphase(t1);
    delay(3.0*pw);
  endloop(v13);
endloop(v11);
xmtroff();
```

In this example, the number of WALTZ loop cycles (`v10`) is calculated first. The WALTZ sequence itself is constructed using a nested loop of composite pulses, whereby the pulse phases are taken from an autoincrementing table. Real-time loops can easily be nested over to several levels. The corresponding Acode instruction segment looks as follows:

```
314   308   202      33   DBLFUNC          two   v12
317   311   205      31   MULtFUNC         v12   v12   v12
321   315   209     150   HighSpeedLINES  TXON
324   318   212      39   ASSIGNFUNC      zero   v11
327   321   215      42   IFMInusFUNC      v10   one Offset =   315
331   325   219      39   ASSIGNFUNC      zero   v13
334   328   222      42   IFMInusFUNC      v12   one Offset =   309
338   332   226     105   TABLE 227 size 48,autoinc 1, divn ret 1,ptr 0
                             2    2    0    0    2    0    2    0
                             2    2    0    2    0    0    2    2
                             0    2    0    2    0    0    2    0
```

```
                                0    0    2    2    0    2    0    2
                                0    0    2    0    2    2    0    0
                                2    0    2    0    2    2    0    2
391   385   279     106   TASSIGN table 227    tblrt
394   388   282      16   SETPHAS90       CH1  tblrt
397   391   285     151   EVENT1_TWRD     8.000 usec
399   393   287     106   TASSIGN table 227    tblrt
402   396   290      16   SETPHAS90       CH1  tblrt
405   399   293     151   EVENT1_TWRD     16.000 usec
407   401   295     106   TASSIGN table 227    tblrt
410   404   298      16   SETPHAS90       CH1  tblrt
413   407   301     151   EVENT1_TWRD     24.000 usec
415   409   303      27   INCRFUNC         v13
417   411   305      42   IFMInusFUNC      v13   v12 Offset =   226
421   415   309      27   INCRFUNC         v11
423   417   311      42   IFMInusFUNC      v11   v10 Offset =   219
427   421   315     150   HighSpeedLINES  (void)
```

The two arguments specified to the `loop` statement specify the loop *count* (`v10` and `v12` in this example) and the loop *counter* (the variable that is used to count the loop cycles). The loop count variable itself is not altered by the loop construct. The loop counter variable is also supplied to the `endloop` statement—mainly to define which `loop` statement the `endloop` call refers to (it also makes it easier to read the code and helps avoiding mistakes). Strictly speaking, the argument to `endloop` isn't really necessary, because loops always must be *either sequential or hierarchically nested*: it doesn't make sense to start a loop inside an other loop, but to terminate it outside.

The Acode interpreter works on a more primitive level than the pulse sequence language itself. First, the loop counter (`v11` for the outer loop, `v13` for the inner loop) is initialized to zero. Then, the system checks whether "loop counts - one" is negative (i.e., the loop count is zero), in which case the entire loop segment is skipped (jump to offset 315 for the outer loop, to offset 309 for the inner loop). At the end of each loop cycle (supposing a non-zero loop count was specified), the loop counter is incremented, and if "loop counter - loop count" is negative (i.e., "loop counter < loop count"), a *branch* to the first instruction inside the loop is performed (jump to offset 226 for the inner loop, to offset 219 for the outer loop).

There are some principal limitations with the above implementation of a delay with WALTZ decoupling. The length of the delay has a distinct "granularity", in that it is a multiple of the duration of a WALTZ cycle[1]. Also, the WALTZ modulation can only run synchronously, and the only way to program events (e.g.: pulses) simultaneous to this delay is to break up the delay into several segments, with the additional events in-between. Of course, this would even increase problems with the "duration granularity".

In general, the above implementation of an explicit modulation scheme is quite elegant and efficient (in terms of Acode space usage); however, it may suffer from some further limitation—through the software loop we feed the pulse programmer with a large number of FIFO words. If the events inside the loop are now all very short (a few microseconds only), the Acode interpretation may become the rate-determining step, and the system may run into a FIFO underflow problem (the sequence would abort at that point). The real-time loop is a very efficient, flexible and elegant tool, but it may not be suited for very fast loops as they are needed in many cases like CRAMPS and

[1] One could in principle always round down the WALTZ loop cycles and perform an additional delay to fill the remainder such that the total delay is accurate, but then that remainder would not be properly modulated (unless more complex coding is used).

similar experiments. Note also that the loop count is a 16-bit real-time variable, and the number of loop cycles is therefore limited to 32767.

As shown in the above example, a real-time loop cycle may include phase changes (and phase calculations). It could even include real-time `if` statements (see Section 15.2, “Real-Time Decisions,” on page 160). The loop cycles can therefore be variable not only in the phase, but even in the number and kind of events inside the loop.

14.3 Hardware Loops

The hardware loop (i.e., the possibility to cycle words in the loop FIFO of the pulse programmer) was created to compensate for the basic deficiency of real-time loops: the lack of ultimate speed (i.e., to cover the difference between the speed of Acode interpretation and the possible speed of propagation of FIFO words in the pulse programmer). Typical examples for experiments that require the hardware looping capability of the pulse programmer are multipulse and CRAMPS type of experiments such as MREV-8 or BR-24 (a part of the MREV-8 sequence is shown here):

```
pulsesequence()
{
   double tau = getval("tau"),
          dtau = tau - pw - rof1 - rof2;
   ...
   initval(np/2.0, v9);
   ...
   delay(d1);
   rgpulse(pw,v4,rof1,rof2);          /* prep pulse */
   starthardloop(v9);
      delay(dtau);
      rgpulse(pw,v4,rof1,rof2);       /*  x */
      delay(dtau);
      rgpulse(pw,v3,rof1,rof2);       /* -y */
      delay(tau+dtau);
      rgpulse(pw,v1,rof1,rof2);       /*  y */
      delay(dtau);
      rgpulse(pw,v2,rof1,rof2);       /* -x */
      delay(tau+dtau-2.0e-7);
      acquire(2.0,2.0e-7);            /* acquire */
      rgpulse(pw,v2,rof1,rof2);       /* -x */
      delay(dtau);
      rgpulse(pw,v3,rof1,rof2);       /* -y */
      delay(tau+dtau);
      rgpulse(pw,v1,rof1,rof2);       /*  y */
      delay(dtau);
      rgpulse(pw,v4,rof1,rof2);       /*  x */
      delay(tau);
   endhardloop();
}
```

In the case of the MREV-8 sequence, a series of 8 pulses and 9 delays (typically 25 single-precision time events in total, with `rof2` set to zero) is inserted *into each sampling interval* (the dwell time being 10 to 100 microseconds). The BR-24 multi-pulse experiment even asks for 24 pulses and 25 delays (73 time events in total) to be squeezed into the same time interval. All these time events typically are on the order of 1 to a few microseconds. There would be no chance for this experiment to work with a soft loop, but for the pulse programmer with its hardware looping capability this is no

problem, as long as the number of events in the loop doesn't exceed the number of words in the loop FIFO. Older systems with output boards (63-word FIFO) cannot perform BR-24 experiments, but all newer systems (using acquisition control boards and pulse sequence control boards) should have no limitation in the area of multipulse experiments.

Hardware loops have some inherent limitations:

- The length of a hardware loop is limited to the size of the loop FIFO: 63 FIFO words for output boards, 1024 FIFO words for acquisition control boards, and 2048 FIFO words for pulse sequence control boards (real-time or software loops can be as big as an entire code segment). When calculating the number of FIFO words in a hardware loop, be aware of double-precision time events. The manual *VNMR User Programming* contains detailed lists on the number of FIFO words involved in the statements that use the AP bus.
- All loop cycles are identical (in real-time or software loops, real-time decisions can be made inside the loop, and phase can be altered either by real-time calculations, by using auto-incrementing tables, or by recalculating table indices).
- Hardware loops can not be nested, because only one loop is implemented in real hardware (the loop FIFO of the pulse programmer).
- The limitation in the number of loop cycles is the same as for real-time loops, because the (hardware) loop counter is also a 16-bit number (i.e., *the maximum number of loop cycles is 32767*).

Multiple hardware loops can be used sequentially. If hardware loops are placed back-to-back (no time event in-between), there is a restriction in the duration of all but the last (back-to-back) hardware loops, in that the total duration of all FIFO words in a loop must be *at least* 0.4 microseconds for each FIFO word in the loop that follows.

With output boards, there are additional restrictions and limitations in that there must be at least a single time event (e.g., a delay of 0.2 microseconds) between any two hardware loops. A hardware loop cannot be shorter than 6.3 microseconds per cycle (the "fall-through time" of the loop FIFO), and with multiple hardware loops following each other, the length of all but the first loop cycles must be at least 80 to 100 microseconds. Also, try to limit the number of events between sequential hardware loops (but keep at least one event in-between); otherwise, it is possible that the loop FIFO runs empty when starting the second loop. Due to the limitation in pre-loop FIFO size, there may not be enough space to pre-load the second loop. Explicit hardware looping is not available on Gemini spectrometers[2].

Note that also the implicit acquisition and an explicit acquisition over the full FID[3] is performed using hardware looping (see Chapter 18, "Acquiring Data," on page 205).

There are also some *programming restrictions* with respect to hardware loops: the use of *real-time math* and of the use of autoincrementing *tables* is not permitted inside hardware loops. This is mainly because it would make programmers believe that real-time math and the incrementation of table pointers also continues during the execution of hardware loops. In addition to that, *real-time decisions* (see Section 15.2, "Real-

[2] The Gemini pulse programmer is not equipped with a pre-loop FIFO; therefore, no events can be pre-loaded after a hardware loop. The consequence is that only one hardware loop per pulse sequence can be performed at the very end of the sequence. This almost by definition is the acquisition loop.

[3] More exactly, the `acquire` statement for more than two data points (per call) outside a hardware loop.

Time Decisions," on page 160) and *real-time loops* are not permitted inside hardware loops. This mostly has to do with the way hardware loops are programmed in Acode[4].

Therefore, C constructs (loops and calls to statements) are the only way to simplify the coding of hardware loops at C level. This also implies that hardware loops may not be very efficient in terms of Acode space. Let's see how the WALTZ-16 decoupling sequence would be coded in a pulse sequence—the same or similar types of constructs have been used in explicitly programmed MLEV-16, MLEV-17, and similar modulation schemes.

```
#include <standard.h>

waltza()
{
  txphase(two); delay(3.0*pw); txphase(zero); delay(4.0*pw);
  txphase(two); delay(2.0*pw); txphase(zero); delay(3.0*pw);
  txphase(two); delay(1.0*pw); txphase(zero); delay(2.0*pw);
  txphase(two); delay(4.0*pw); txphase(zero); delay(2.0*pw);
  txphase(two); delay(3.0*pw);
}

waltzb()
{
  txphase(zero);  delay(3.0*pw); txphase(two); delay(4.0*pw);
  txphase(zero);  delay(2.0*pw); txphase(two); delay(3.0*pw);
  txphase(zero);  delay(1.0*pw); txphase(two); delay(2.0*pw);
  txphase(zero);  delay(4.0*pw); txphase(two); delay(2.0*pw);
  txphase(zero);  delay(3.0*pw);
}

pulsesequence()
{
  status(A);
    delay(d1);
  status(B);
    rgpulse(pw,zero,rof1,0.0);
    initval(getval("tau")/(16.0*6.0*pw),v10);
    xmtron();
    starthardloop(v10);
      waltza(); waltzb(); waltzb(); waltza();
    endhardloop();
    xmtroff();
  status(C);
    pulse(pw,oph);
}
```

We could of course have followed the coding scheme of the real-time loop solution and split up the sequence in simple composite (90-180-270) pulses. As mentioned before, the hardware loop solution is not very efficient in Acode space usage, as you can see

[4] The number of FIFO words inside a hardware loop (which is an argument to the `HDLOOP` Acode instruction) is calculated by the time when the Acode is built (i.e., when typing `go`), and not during execution time.

from the printout below[5] (what really counts, of course, is the experimental flexibility, not the ultimate level of Acode economy):

```
317   311   205       0   NO_OP
321   315   206      67   HWLOOP with acq interrupt    36 words   v10
322   316   210      16   SETPHAS90       CH1    two
325   319   213     151   EVENT1_TWRD     24.000 usec
327   321   215      16   SETPHAS90       CH1   zero
330   324   218     151   EVENT1_TWRD     32.000 usec
332   326   220      16   SETPHAS90       CH1    two
335   329   223     151   EVENT1_TWRD     16.000 usec
337   331   225      16   SETPHAS90       CH1   zero
340   334   228     151   EVENT1_TWRD     24.000 usec
342   336   230      16   SETPHAS90       CH1    two
345   339   233     151   EVENT1_TWRD     8.000 usec
347   341   235      16   SETPHAS90       CH1   zero
350   344   238     151   EVENT1_TWRD     16.000 usec
352   346   240      16   SETPHAS90       CH1    two
355   349   243     151   EVENT1_TWRD     32.000 usec
357   351   245      16   SETPHAS90       CH1   zero
360   354   248     151   EVENT1_TWRD     16.000 usec
362   356   250      16   SETPHAS90       CH1    two
365   359   253     151   EVENT1_TWRD     24.000 usec
367   361   255      16   SETPHAS90       CH1   zero
370   364   258     151   EVENT1_TWRD     24.000 usec
372   366   260      16   SETPHAS90       CH1    two
375   369   263     151   EVENT1_TWRD     32.000 usec
377   371   265      16   SETPHAS90       CH1   zero
380   374   268     151   EVENT1_TWRD     16.000 usec
382   376   270      16   SETPHAS90       CH1    two
385   379   273     151   EVENT1_TWRD     24.000 usec
387   381   275      16   SETPHAS90       CH1   zero
390   384   278     151   EVENT1_TWRD     8.000 usec
392   386   280      16   SETPHAS90       CH1    two
395   389   283     151   EVENT1_TWRD     16.000 usec
397   391   285      16   SETPHAS90       CH1   zero
400   394   288     151   EVENT1_TWRD     32.000 usec
402   396   290      16   SETPHAS90       CH1    two
405   399   293     151   EVENT1_TWRD     16.000 usec
407   401   295      16   SETPHAS90       CH1   zero
410   404   298     151   EVENT1_TWRD     24.000 usec
412   406   300      16   SETPHAS90       CH1   zero
415   409   303     151   EVENT1_TWRD     24.000 usec
417   411   305      16   SETPHAS90       CH1    two
420   414   308     151   EVENT1_TWRD     32.000 usec
422   416   310      16   SETPHAS90       CH1   zero
425   419   313     151   EVENT1_TWRD     16.000 usec
427   421   315      16   SETPHAS90       CH1    two
430   424   318     151   EVENT1_TWRD     24.000 usec
432   426   320      16   SETPHAS90       CH1   zero
435   429   323     151   EVENT1_TWRD     8.000 usec
437   431   325      16   SETPHAS90       CH1    two
440   434   328     151   EVENT1_TWRD     16.000 usec
442   436   330      16   SETPHAS90       CH1   zero
445   439   333     151   EVENT1_TWRD     32.000 usec
447   441   335      16   SETPHAS90       CH1    two
450   444   338     151   EVENT1_TWRD     16.000 usec
452   446   340      16   SETPHAS90       CH1   zero
455   449   343     151   EVENT1_TWRD     24.000 usec
457   451   345      16   SETPHAS90       CH1    two
```

[5] The WALTZ-16 solution using real-time loops shown previously (see Section 14.2, “Real-Time Loops,” on page 148) required 113 Acode words (226 bytes). The coding with a hardware loop shown here takes up 184 Acode words or 368 bytes. This is *not* necessarily a typical example.

```
460   454   348     151   EVENT1_TWRD     24.000 usec
462   456   350      16   SETPHAS90       CH1   zero
465   459   353     151   EVENT1_TWRD     32.000 usec
467   461   355      16   SETPHAS90       CH1    two
470   464   358     151   EVENT1_TWRD     16.000 usec
472   466   360      16   SETPHAS90       CH1   zero
475   469   363     151   EVENT1_TWRD     24.000 usec
477   471   365      16   SETPHAS90       CH1    two
480   474   368     151   EVENT1_TWRD     8.000 usec
482   476   370      16   SETPHAS90       CH1   zero
485   479   373     151   EVENT1_TWRD     16.000 usec
487   481   375      16   SETPHAS90       CH1    two
490   484   378     151   EVENT1_TWRD     32.000 usec
492   486   380      16   SETPHAS90       CH1   zero
495   489   383     151   EVENT1_TWRD     16.000 usec
497   491   385      16   SETPHAS90       CH1    two
500   494   388     151   EVENT1_TWRD     24.000 usec
502   496   390     150   HighSpeedLINES  (void)
```

The HDLOOP instruction that starts the hardware loop in the Acode is followed by the number of FIFO words inside the loop (*not* the number of Acode words!), and the loop count (a real-time variable). The endhardloop statement does not generate an Acode instruction by itself, but it causes the number of loop FIFO words to be written into the HDLOOP instruction.

Chapter 15. Decisions

The use of C-based (or previously Pascal-based) decisions in pulse sequences has a long tradition in Varian pulse sequences.

15.1 Decisions and Branchings in C

C-based decisions are based on both qualitative as well as quantitative checks. They are used to increase the reliability of many pulse sequences, to make them easier to use, and to increase their flexibility.

Aborting a Sequence in Case of a Improperly Set Parameter

Aborting a sequence can involve duty cycle calculations and checking, checks for parameter settings that could possibly damage the spectrometer hardware (like rf coils or amplifiers in the case of excessive rf power), and avoiding “impossible” parameter settings (like trying to decouple a 2D spectrum with antiphase magnetization). The following example is from the HMQC pulse sequence:

```
if ((dm[A] == 'y') || (dm[B] == 'y'))
{
   printf("DM must be set to either 'nny' or 'nnn'.\n");
   abort(1);
}
```

Standard output created by the `printf` function shows up in the VNMR text window. The `abort` function terminates the `go` command and the Acode generation through the compiled pulse sequence, and Acode generated up to that point is discarded.

Ensuring Compatibility with Spectrometer Hardware

Many pulse sequences contain statements that can only be used on certain types of spectrometer hardware. Therefore, many sequences include constructs such as:

```
if (newtrans)
{
   stepsize(base, TODEV);
   xmtrphase(v1);
}
else
   phaseshift(base,v1,TODEV);
```

Although programmers prefer to write “universal” pulse sequences, it would often make a pulse sequence very complicated if we tried to cover the entire range of spectrometers that can be driven by VNMR software. Thus, some sequences use constructs such as the following:

```
if (!newdec)
{
   printf("This sequence requires direct synthesis RF  \
           on DEC.\n");
   abort(1);
}
```

As shown in Table 8, there are a number of predefined flag variables that can be used in such constructs. Their definitions can be found in the header file `/vnmr/psg/acqparms.h`.

Table 8. PSG hardware flag and configuration variables

Flag Variable	*Meaning / Function*
`newtrans`	True if system equipped with direct synthesis (as opposed to broadband or fixed frequency) rf on observe channel.
`newdec`	True if system equipped with direct synthesis (as opposed to broadband or fixed frequency) rf on (first) decoupler channel.
`newtransamp`	True if system equipped with linear (as opposed to class C) amplifiers on observe channel.
`newdecamp`	True if system equipped with linear (as opposed to class C) amplifiers on (first) decoupler channel.
`vttype`	0 = none, 1 = Varian, 2 = Oxford
`H1freq`	Proton frequency of instrument: 200, 300, 400, 500, 600, 750
`automated`	True if system has computer-controlled lock, gain, and decoupler power. This flag is left over from old software; all spectrometers running VNMR are automatic.
`fifolpsize`	Size of loop FIFO: 63 (output board), 1024 (acquisition control board), or 2048 (pulse sequence control board).
`NUMch`	Number of rf channels that are configured: 2, 3, or 4.

Issuing Warning Messages in the Case of Questionable Parameter Settings

Sometimes it may not be necessary to abort the pulse sequence, but it is perhaps adequate to have the sequence display a warning message:

```
if ((newtransamp) && (rof1 < 9.9e-6) && (ix == 1))
   printf("Warning: ROF1 is less than 10 usec.\n");
```

In the case of arrayed and multidimensional experiments, it is strongly recommended to only print a warning for the first increment; otherwise, the VNMR text window may be flooded with error messages. Of course, there may be cases where a check should be performed for each increment. The above example should just remind the operator that `rof1` is below its normal value for linear amplifiers (which could result in pulse amplitude instabilities).

Turning On or Off Pulse Sequence Features

A very popular and powerful feature in VNMR pulse sequences is the use of *flags* for enabling and disabling individual pulse sequence statements. This allows combining many pulse sequences (in the sense of a defined sequence of pulses and delays) into a single pulse sequence program. There are numerous examples of pulse sequences with additional experimental flexibility through C decisions, like the HMQC sequence with its optional nulling of protonated signals through a BIRD inversion pulse (simplified coding), such as the following:

```
if ((null > 0.0) && (mbond[A] == 'n'))
{
```

```
   rgpulse(pw,v1,rof1, 0.0);
   delay(bird-rof1);
   decrgpulse(pwx,v9,rof1,0.0);
   simpulse(2.0*pw,2.0*pwx,v1,v1,1.0e-6,0.0);
   decrgpulse(pwx,v9,1.0e-6,0.0);
   delay(bird-rof1);
   rgpulse(pw,v2,rof1,0.0);
   hsdelay(null);
}
```

The second conditional in this example ensures than BIRD nulling is not used when the long-range option (`mbond='y'`) is used.

Other examples include sequences like `hetcor` (with a built-in long-range option and the possibility to suppress proton multiplets using the `hmult='n'`, which causes an inversion pulse to be replaced by a BIRD pulse), and many others. Sometimes, numeric parameters are used to make decisions (like checking whether the delay `null` is non-zero in the above example), but mostly such switches are based on flag parameters (such as `mbond` and `hmult` in the above examples).

Dynamically Arranging the Sequence of Events

Sometimes it is necessary to rearrange the sequence of events depending on the value of a parameter. One example is found in the HMQC pulse sequence, where there is a 180-degree proton pulse in the middle of the evolution time. The principal sequence of events is as follows:

```
decpulse(pwx,v3);
delay(d2/2.0-rof1);
pulse(2.0*pw,v4);
delay(d2/2.0-rof2);
decpulse(pwx,v5);
```

This construct *looks* correct, but it has several deficiencies. The duration of the central pulse is not compensated for in the evolution time, and the precession during the 90-degree pulses surrounding the evolution time is not taken into account (see Chapter 19, “Multidimensional Experiments,” on page 215). Additionally, the first increment the delay `d2` is zero, giving an error message about negative delays.

Apart from that (which is just a programming problem), the first increment has a considerable gap between the two 90-degree X-pulses, which causes problems in the first trace and baseline distortions in the final 2D spectrum. In order to do it properly, the pulses on both rf channels should be performed on top of each other for the first increment. Unfortunately, the two X-pulses don’t have the same phase; therefore, a single `simpulse` statement cannot be used. One can minimize interpulse dead times by keeping the amplifiers unblanked for parts of a pulse sequence:

```
corr = 2.0*pwx/3.1416 + pw + 1.0e-6;
if (d2/2.0 > corr)
{
   rcvroff();
   decrgpulse(pwx,v3,rof1,0.0);
   delay(d2/2.0-corr);
   rgpulse(2.0*pw,v4,1.0e-6,0.0);
   delay(d2/2.0- corr);
   decrgpulse(pwx,v5,1.0e-6,0.0);
   rcvron();
}
else
```

```
{
   simpulse(pw,pwx,v4,v3,rof1,0.0);
   simpulse(pw,pwx,v4,v5,1.0e-6,0.0);
}
```

This is a simplified solution: Strictly speaking, this is still not perfect. If pwx is less than pw, there is still a gap between the X-pulses, but with most configurations used for this type of experiment, the X-pulses are longer than the proton pulses, and in this case the above solution is accurate for the first increment. Also, increments other than the first one can fall into a domain where the X pulses overlap with the proton 180, which may require some modifications to the above algorithm.

An interesting example is the combination of two pulse sequences into a single one—for the same experiment. The example shown here is a constant-time heteronuclear correlation experiment, where a simultaneous inversion pulse moves within a fixed time interval[1]. Such experiments often suffer from limited resolution because the moving pulse at some point (increment) reaches the end of the fixed delay. In the example shown here, the fixed interval contains a BIRD pulse, which would normally set a limit to the number of increments. Thanks to use of C decisions (and to the fact that an Acode segment is calculated individually for each 2D increment), we can apply the trick of letting the simultaneous inversion "jump over the BIRD pulse" and continue behind it up to the very end of the fixed delay, which doubles the achievable resolution in f_1 (only relevant parts of the pulse sequence are shown here):

```
decpulse(pp,t1);                    /* excitation pulse */
if (d2/2.0 < d3)
{
   if (d2/2.0 > rof1)               /* start of evolution time */
      delay(d2/2.0-rof1);
   rgpulse(pw,t1,rof1,0.0);         /* 180 H + 180 X */
   simpulse(2.0*pw,2.0*pp,t2,t3,1.0e-6,0.0);
   rgpulse(pw,t1,1.0e-6,0.0);
   if (d3 - d2/2.0 > rof1)
      delay(d3-d2/2.0-rof1);
   decrgpulse(pp,t1,rof1,0.0);      /* BIRD-pulse */
   delay(tau-rof1);
   rgpulse(pw,t1,rof1,0.0);
   simpulse(2.0*pw,2.0*pp,t2,t4,1.0e-6,0.0);
   rgpulse(pw,t1,1.0e-6,0.0);
   delay(tau-rof1);
   decrgpulse(pp,t1,rof1,0.0);
   delay(d3 -tau -rof1);
}

else
{
   delay(d3-rof1);
   decrgpulse(pp,t1,rof1,0.0);      /* BIRD-Pulse */
   delay(tau-rof1);
   rgpulse(pw,t1,rof1,0.0);
   simpulse(2.0*pw,2.0*pp,t2,t4,1.0e-6,0.0);
   rgpulse(pw,t1,1.0e-6,0.0);
   delay(tau-rof1);
   decrgpulse(pp,t1,rof1,0.0);
```

[1] M. Perpick-Dumont, W.F. Reynolds & R. Enriquez, *Magn. Reson. Chem.* **26**, 358 (1988); W.F. Reynolds, S. McLean, M. Perpick-Dumont & R. Enriquez, *Magn. Reson. Chem.* **26**, 1068 (1988).

```
   delay(d2/2.0-d3-rof1);
   rgpulse(pw,t1,rof1,0.0);          /* 180 H + 180 X */
   simpulse(2.0*pw,2.0*pp,t2,t3,1.0e-6,0.0);
   rgpulse(pw,t1,1.0e-6,0.0);
   delay(2.0*d3-d2/2.0-tau-rof1);
}
simpulse(pw,pp,t1,t5,rof1,0.0);      /* 90 H + 90 X */
delay(tau/2.0);
```

Implicit Decisions

Of course, decisions are not limited to the pulse sequence itself. Decisions are involved internally in most pulse sequence statements, whether to do the right thing for the given hardware or to avoid unnecessary Acode and FIFO words (like time events with zero duration).

Decisions Set by the status Statement

Another class of decisions is hidden in the `status` statement (see also "Implicit Gating" on page 53). Most pulse sequences are divided into several basic sections by inserting `status` calls at suitable points in the pulse sequence:

```
   ...
   status(A);
      hsdelay(d1);
   status(B);
      pulse(p1,zero);
      hsdelay(d2);
      pulse(pw,oph);
   status(C);
}
```

This is a simplified version of the `s2pul` pulse sequence. It is split into three basic sections: status `A`, which includes the relaxation period (`d1`); status `B`, which includes the two pulses and the evolution time (`d2`); and status `C`, which is valid during the acquisition time. The argument to the status function is a simple integer: `A` is defined as 0, `B` is defined as 1, etc., up to `Z`, which is defined as 25. `status(A)` simply means that during the following time events, the first ("0th" in C syntax) character in predefined multifield flag variables (`dm`, `dm2`, `dm3`, `dmm`, `dmm2`, `dmm3`, `homo`, `homo2`, `homo3`, and `hs`) is active up to the next `status` call with a different value in the argument. These flag variables can be up to 255 characters long (beyond the predefined constants `A` to `Z`), but in practice only 3 to 5 fields are used.

These multifield flags are a very easy way of controlling the gating of transmitters and modulation modes, and enabling or disabling homospoil pulses in `hsdelay` calls, which significantly enhances the flexibility of most pulse sequences.

The definition is that if more fields are addressed in the pulse sequence than there are characters in a flag string, the last character is propagated as much as necessary (i.e., `dm='ny'` is the equivalent to `dm='nyyyyyyyyyyy...'`). At the end of a scan (and at the end of the experiment), the system *implicitly returns to* `status(A)`. This is most relevant to experiments that use high-power decoupling, or decoupling during the acquisition in general. For example, when using `s2pul` with decoupling, it is better not to use `dm='y'`, but rather `dm='nyy'`, even if delay `d1` is not used. This switches off the decoupler at the end of the experiment and avoids sample heating due to the rf power fed into the probe continuously. If a relaxation delay is to be used with `s2pul`

(and p1=0), we can use d2 instead of d1, and dm='nyy' for full decoupling or dm='nyn' for gated decoupling.

Note that with some configurations (UNITY*plus* and UNITY spectrometers), the status statement may not just change some fast bits (status lines), but may involve AP bus FIFO words (depending on which flags change at that point) that take a finite time (see Chapter 12, “AP Bus Traffic,” on page 137).

Checking Flag Parameters

Simple flag tests are easy. Constructs like if (dm[A] == 'y') are adequate in most cases and work also with flags that are newly created for a specific pulse sequence (like if (mbond[A] == 'n') in a previous example in this chapter). Problems occur when flag fields *other than the first* should be tested: if (dm[C] == 'y') sounds like a correct test—but what if the parameter dm has been set to 'y'? In this case, dm[C] returns a random value (dm[B] would return a null character, but still not 'y'!), and the test would fail, because the test returns FALSE instead of TRUE. A proper construct for testing flag fields, other than the first one, is not that simple:

```
int flagsize,
    index = statusindx;
char xflag[MAXSTR];
getstr("xflag",xflag);
...
flagsize = strlen(xflag);
if (index >= flagsize)
  index = flagsize - 1;
if (xflag[index] == 'y')
  ...
```

The variable statusindx is the currently active status, as set by the status statement. Another solution would be to ensure (by means of a macro or parameter entry restrictions) that this particular flag has the required length. Fortunately, the vast majority of user-created flags use only one field.

15.2 Real-Time Decisions

Decisions that need to be made from transient to transient cannot be programmed in C, because only one Acode segment is generated per FID, which performs all scans. In such a case, we need real-time decisions—decisions made by the Acode interpreter.

Programming Real-Time Decisions

Suppose that in a particular sequence, with every odd scan we wanted to add a refocusing period to a pulse sequence. This could be achieved as follows:

```
mod2(ct,v10);              /* 0 1 0 1 */
ifzero(v10);               /* every odd scan (ct=0,2,4...) */
   delay(tau-rof1);
   rgpulse(pw,v1,rof1,0.0);
   delay(tau);
endif(v10);
```

There is only one logical test for real-time decisions: the test whether a real-time variable (v10 in this case) is zero. To fulfil this condition for every odd scan, we

calculate `ct mod 2`. The above construct generates the following Acode (`pw=7, tau=1/140, rof1=10`):

```
298   292   186     36   MOD2FUNC          ct   v10
301   295   189     41   IFNotZeroFUNC    v10  zero Offset =   218
305   299   193    152   EVENT2_TWRD         7 msec +   133 usec
308   302   196     16   SETPHAS90       CH1     v1
311   305   199    150   HighSpeedLINES  RXOFF
314   308   202    151   EVENT1_TWRD     10.000 usec
316   310   204    150   HighSpeedLINES  RXOFF TXON
319   313   207    151   EVENT1_TWRD     7.000 usec
321   315   209    150   HighSpeedLINES  RXOFF
324   318   212    150   HighSpeedLINES  (void)
327   321   215    152   EVENT2_TWRD         7 msec +   143 usec
330   324   218    150   HighSpeedLINES  (void)
```

The `ifzero` statement generates an `IFNotZeroFUNC` instruction in the Acode, which performs a jump to address 218 (i.e., after the `endif` statement) if the variable (`v10` in this case) is non-zero. The `endif` statement itself does not generate Acode, but it is used to determine the jump address in the `IFNotZeroFUNC` instruction.

The argument with `endif` is not used in the Acode, but it serves to associate the `endif` with the corresponding `ifzero` call: `ifzero` constructs can also be nested (see also the last example in Section 10.4, “Real-Time Logical Decisions,” on page 106), and in such cases the argument is an easy way for the software to “know” in which `IFNonZeroFUNC` instruction it has to set the jump address[2].

Mostly in the case of such conditional events, we would like to ensure that all scans are performed with exactly the same overall timing, to avoid adding up FIDs with different phase and amplitude. In this case, the `elsenz` function is also used between the `ifzero` and the `endif` calls:

```
mod2(ct,v10);              /* 0 1 0 1 */
ifzero(v10);               /* every  odd scan (ct=0,2,4..) */
   rgpulse(pw/2.0,v1,rof1,0.0);
elsenz(v10);               /* every even scan (ct=1,3,5..) */
   delay(pw/2.0+rof1);
endif(v10);
```

In this example, we are performing a 45-degree pulse with every odd scan, and with the even scan numbers this pulse is replaced by a delay. Of course, the delay length must not only include the pulse length, but also any pre- and post-pulse delay included with the `rgpulse` statement (beware of implicit delays in statements like `pulse`!)[3].

In the Acode, the above construct looks as follows:

```
352   346   240     36   MOD2FUNC          ct   v10
355   349   243     41   IFNotZeroFUNC    v10  zero Offset =   268
359   353   247     16   SETPHAS90       CH1     v1
362   356   250    150   HighSpeedLINES  RXOFF
```

[2] Strictly speaking, this is not true because it is always clear which `ifzero` and `endif` belongs to, as long as the `ifzero` ... `endif` constructs are hierarchically stacked, and non-hierarchical stacking (e.g., `ifzero(v1)`... `ifzero(v2)`... `endif(v1)`... `endif(v2)`) doesn’t make sense at all (the same is true for real-time looping).

[3] To be accurate, we cannot simply replace a pulse with a delay of the same length, since we also need to take into account the precession during the pulse. This can be done using a correction term that shortens the delay. If `pw` is a 90-degree pulse (`pw/2.0` therefore 45 degrees), the accurate delay length is `pw/2.0 - pw/3.1416 + rof1` (see also Chapter 19, “Multidimensional Experiments,” on page 215).

```
365   359   253     151   EVENT1_TWRD     10.000 usec
367   361   255     150   HighSpeedLINES  RXOFF TXON
370   364   258     151   EVENT1_TWRD     3.500 usec
372   366   260     150   HighSpeedLINES  RXOFF
375   369   263     150   HighSpeedLINES  (void)
378   372   266      20   BRANCH          Offset    270
380   374   268     151   EVENT1_TWRD     13.500 usec
382   376   270     150   HighSpeedLINES  (void)
```

The `elsenz` statement again takes the same argument as the `ifzero` and `endif` calls (the real-time variable `v10` in this case). It terminates the `if` part of the construct with a `BRANCH` (jump) instruction and sets the jump address in the `IFNotZeroFUNC` instruction to the first instruction after the `if` part (after the `BRANCH` instruction). The `endif` statement in this case sets the jump address in the `BRANCH` instruction at the end of the `if` part.

Real-time decisions can also be used in real-time math for the calculation of complex phase cycles (see Chapter 10, “Phase Calculations,” on page 95). A table index should not be incremented with each scan, but perhaps with every fourth scan only.

Generating the Flag Variable

In most cases, the flag variable for real-time decisions is generated using some kind of (real-time) modulo statement. To generate a flag variable with a period of *n* scans, one has to take modulo *n* of a variable that is incremented with every scan like `ct`, or modulo *n*/2 if the variable is incremented every second scan (`ct/2`), etc. Here are some examples for the construction of `ifzero` flag variables (`x` in the comment stands for a non-zero value):

```
mod2(ct,v1);          /* 0 1 0 1 */
sub(one,v2);          /* 1 0 1 0 */

mod4(ct,v3);          /* 0 x x x 0 x x x */

add(ct,one,v4);       /* 1 2 3 4 5 6 7 8 */
mod4(v4,v4);          /* x x x 0 x x x 0 */

hlv(ct,v5);           /* 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 */
add(v5,two,v5);       /* 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 */
mod4(v5,v5);          /* x x x x 0 0 x x x x x x 0 0 x x */

modn(ct,three,v6);    /* 0 x x 0 x x */
```

To invert the logic in a simple `ifzero` construct (without `elsenz` branch), simply use the `elsenz` branch (to simulate a non-existing ifnotzero statement):

```
add(ct,two,v7);       /* 2 3 4 5 6 7 8 9 */
hlv(v7,v7);           /* 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 */
mod4(v7,v7);          /* x x x x x x 0 0 x x x x x x 0 0 */
ifzero(v7);
elsenz(v7);
  ...
endif(v7);
```

Chapter 16. Waveform Generators

The optional waveform generator (or programmable pulse modulator as it was called in UNITY spectrometers) is primarily a board that permits changing the amplitude and the (small-angle) phase of the output of a transmitter board simultaneously and quickly. Different from the pulse programmer, it can set both the amplitude and the small-angle phase shifts using fast status lines. You can regard it as "autonomous fast bit extension of the pulse programmer."

16.1 How Does a Waveform Generator Fit Into the System?

Figure 19 is a diagram of waveform generator circuitry in the UNITY*plus*. The waveform generator sends output through the transmitter digital control board to the (small-angle and 90-degree) phase and amplitude modulation circuitry on the (direct synthesis) transmitter board. Both the amplitude and the phase modulation are also accessible from the AP bus, but—as we will see—the waveform generator can do the same thing, but much faster. The waveform generator is tightly coupled to a particular transmitter board and can only interact with the channel on which it is installed (any channel can be equipped with a waveform generator). Reconfiguring the waveform generator for an other channel involves jumper changes on the board (the AP bus addressing is channel-specific). It is not recommended as a routine operation.

The phase control circuitry on the transmitter board was also described in Section 4.2, "How Do Pulses Work?," on page 40. It does the small-angle phase shifts (0.25 degrees resolution) and the 90-degree phase shifts in two separate, consecutive steps. It takes 42 MHz as input frequency; the output is the phase-modulated 10.5 MHz intermediate frequency (I.F.). The phase control takes digital inputs from the pulse programmer (90-degree phase shifts via fast lines, small-angle phase shifts via AP bus), from the waveform generator (small-angle and 90-degree phase shifts via fast direct lines), and from the phase modulator (just the 90-degree phase shifts via fast status lines). Each channel is equipped with a phase modulator (for preprogrammed decoupling sequences like square wave, swept square wave, noise, MLEV-16, WALTZ-16, XY-32, GARP-1), and optionally with a waveform generator (freely programmable decoupling sequences, pulse shaping). The 90-degree and small-angle phase shifts are *added* to the phase setting of the pulse programmer set via AP bus.

The amplitude modulation circuitry adjusts the amplitude of the 10.5 MHz I.F. signal in 4096 *linear* steps (12 control bits), as opposed to the (63 or 79 dB) attenuators that operate in dB (i.e., in logarithmic units). The 4096 linear steps correspond to a power range of 72 dB. The amplitude modulator takes input from either the pulse programmer (via AP bus) or from the waveform generator (via fast status lines). From the 12 control bits, the waveform generator only controls the 10 most significant bits (1024 steps, 60-dB power range). The amplitude modulation circuitry takes the phase-modulated 10.5 MHz I.F. from the phase modulation circuitry. Its output is the phase- and amplitude-modulated I.F., which is then mixed with the local oscillator frequency (which is the same as the observe frequency *plus* the I.F.) that is generated directly by a PTS frequency synthesizer and is also used in the receiver (see Chapter 18, "Acquiring Data," on page 205).

The output of the mixer (the mixing product) is the observe frequency, which carries the phase- and amplitude modulation from the 10.5 MHz (I.F.) input. The observe frequency that passes the transmitter gate (which can also be controlled by the waveform generator) and the 79 dB (or 63 dB) attenuator is the amplified in the rf power amplifier and finally enters the probe. The waveform amplitude takes precedence over the pulse programmer amplitude that is set via AP bus.

Spectrometers earlier than UNITY*plus* (UNITY and VXR) used a variety of different transmitter boards:

- Only the AM/PM (amplitude modulation and phase modulation) direct synthesis transmitter board was compatible with a waveform generator, and the amplitude modulation was only accessible via the waveform generator, not via the AP bus. The AM-PM board had 0.5-degree phase resolution and 1024-step (60-dB) linear amplitude range.

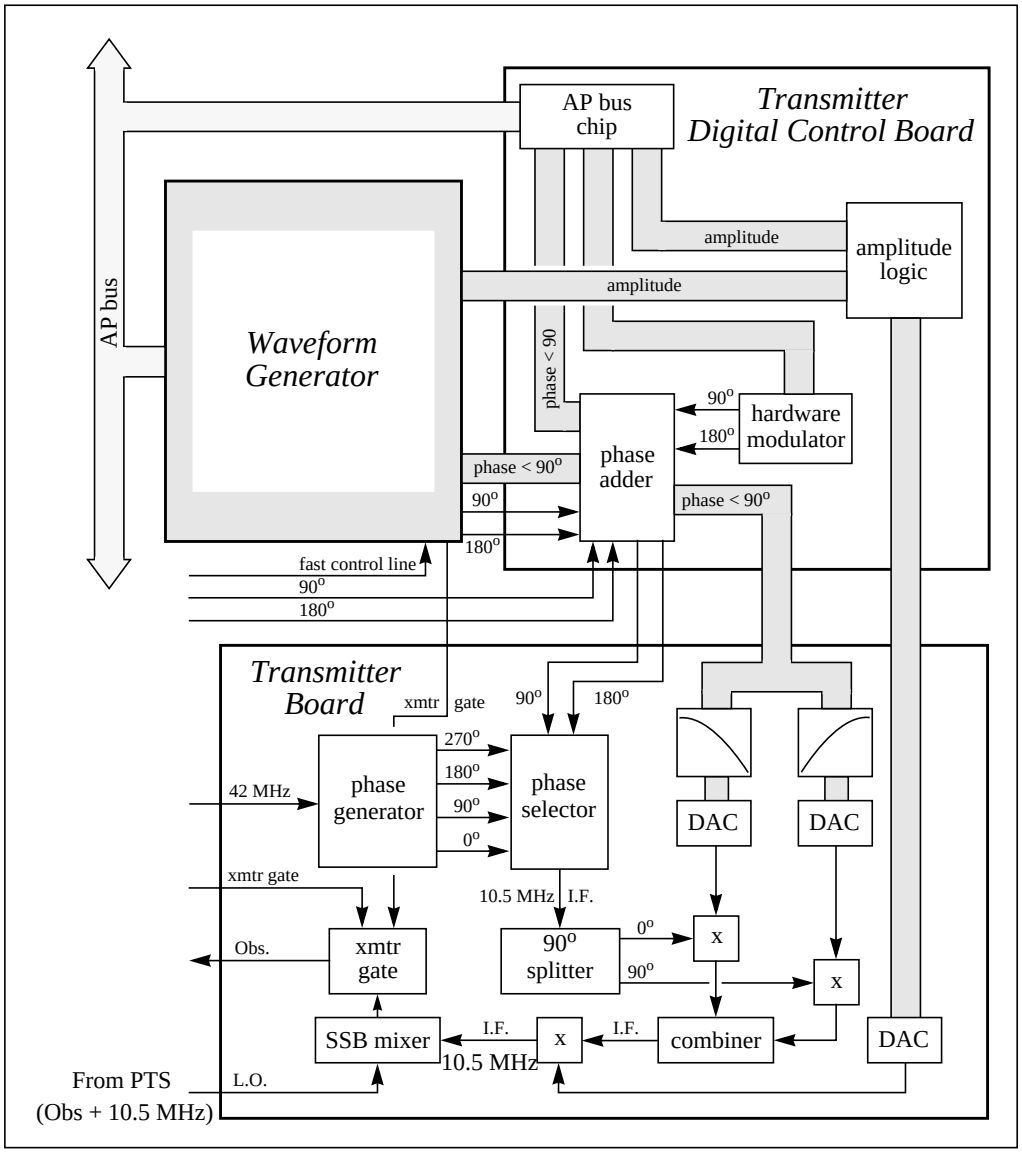


Figure 19. UNITY*plus* waveform generator circuitry

- Previous direct synthesis boards only had phase modulation capability (0.5-degree resolution, like the AM-PM board), accessible via the AP bus (and fast lines for the 90-degree phase shifts). Due to the lack of amplitude modulation capability, these boards were incompatible with waveform generators.
- Fixed frequency and broadband-type transmitter boards did not have amplitude or phase modulation capability (except for 90-degree phase shifts) and were not compatible with a waveform generator either.

Because the AM/PM transmitter board used in UNITY spectrometers did not allow setting the modulator amplitude by AP bus and because the AM/PM board combines both the digital and the analog functions on a single board, the corresponding connection scheme (shown in Figure 20) is slightly simpler.

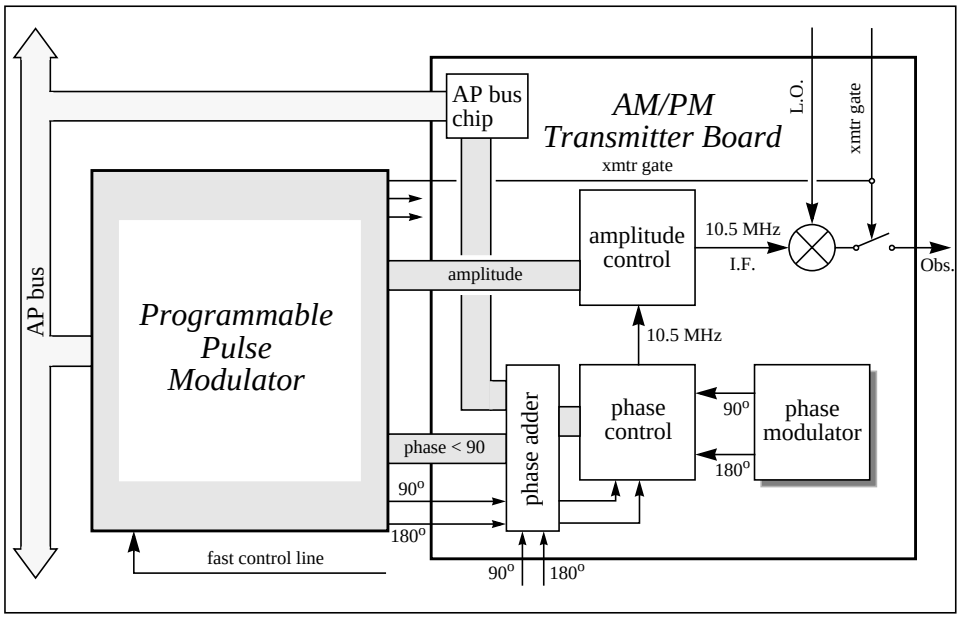


Figure 20. UNITY programmable pulse modulator circuitry

16.2 How Does a Waveform Generator Work?

The waveform generator is similar to a pulse programmer in that it directly controls spectrometer hardware (via fast lines only) and provides accurate timing, independent of the Acode computing. However, there are some distinct differences: the pulse programmer is built as a *flow-through* (FIFO) *buffer*, because the information it handles is *dynamic* and in essence non-repetitive (subsequent scans are normally not identical, because of phase cycling etc.). A waveform generator, on the other hand, is a *static* buffer (based on static RAM), built to handle a constant set of *generic* shapes and pattern within a given pulse sequence. The term generic means that the duration (the duration scaling) of a pulse shape may vary an arbitrary number of times within a pulse sequence, but the phase and amplitude pattern remain constant. To better understand the way a waveform generator works, we need to have a more detailed look at the way it is built and how it operates.

As shown in Figure 21, the central part of the waveform generator board consists of a 256-Kbyte memory (static RAM, i.e., random access memory that does not require

constant refreshing). That memory is organized in 65536 words (64 Kwords) of 32 bits each. A *variable size* portion is used for instructions; the rest is used to store pattern and shapes (see below). The other components of the waveform generator include an interface to the AP bus, an output buffer, control and timing circuitry (20 MHz, derived from the 40 MHz input frequency), loop control circuitry, and an amplitude multiplier that is only used (and present) for shaping pulsed field gradients.

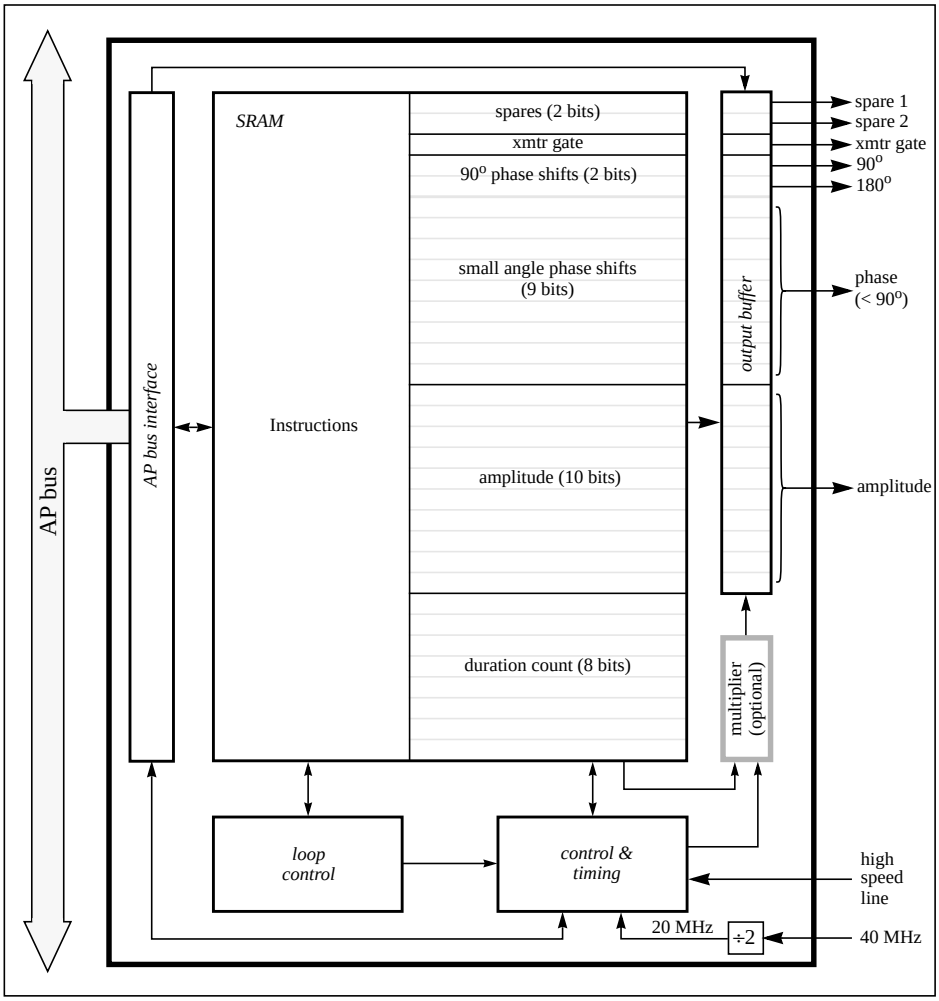


Figure 21. Waveform generator board

The waveform generator is very fast in pulse shaping. All parameters under its control (amplitude, 90-degree and small-angle phase shifts, and the transmitter gate) are changed *instantaneously and simultaneously* (unlike changing these parameters via AP bus, see Chapter 12, “AP Bus Traffic,” on page 137). The *minimum time slice* for waveforms and modulation pattern is 0.2 microseconds, the *timing resolution* is 50 nanoseconds on the UNITY*plus*, and 100 nanoseconds on UNITY spectrometers.

This is the instruction timing. Apart from internal propagation delay (discussed in the section “Shaped Pulses in the Acode” on page 176), there are always some minor delays until the parameter change takes effect in real hardware, especially with respect to phase changes in UNITY spectrometer (see also Section 4.2, “How Do Pulses Work?,” on page 40). For shaped pulses and modulation patterns, these minor hardware delays are irrelevant because all time slices experience the same delays.

Overall, the waveform generator is fast enough to shape even hard pulses as short as a few microseconds. On the other hand, its pattern memory is large enough to hold even the most complex pulse shapes and modulation pattern known today.

Sequence of Events in a Waveform Generator

While the internal data and information paths on the waveform generator board seem complicated, the entire functionality can be explained fairly well based on the organization and the usage of the RAM. Let's first have a look at the sequence of events during the execution of a pulse sequence that uses the waveform generator:

- The pulse sequence executable reads the shape or pattern name from the statement (`shape_pulse`, `obsprgon`, etc.) in the sequence or from the modulation sequence parameter (`dseq`, etc.), reads and interprets the shape or pattern file (more about this below), and generates a waveform generator data file named `expn.username.PID.RF`[1] in `/vnmr/acqqueue`. This file contains both the waveforms and the instruction blocks that will be uploaded to the waveform generators. Each data segment (pattern and instruction blocks) is preceded by a short header (8 bytes) that contains the AP bus address of the waveform generator (to which the data will be sent) plus the starting RAM address and the number of (32-bit) words in the pattern or block.
- At the same time, a file `/vnmr/acqqueue/ldcontrol` is generated or updated. This file contains a memory map for all waveform generators of the system. It shows where in the waveform generator memory (and in which waveform generator) the current waveform(s) and the corresponding instruction blocks are located (i.e., the starting address). This starting address is also used in the pulse sequence when generating Acode instructions that call waveform generator instruction blocks.
- Before uploading the Acode for a new experiment that uses waveform generators, `Acqproc` uploads the waveform generator *data* (pulse shapes and decoupling pattern) and *instruction blocks* in the file `expn.username.PID.RF` from `/vnmr/acqqueue` into the HAL memory. The acquisition also uploads the data via the pulse programmer and the AP bus into the waveform generator pattern and instruction memory.
- The pulse sequence is then initiated by `Acqproc`, the same as any other experiment. Whenever a waveform generator is to be started within a pulse sequence (e.g., at the beginning of a shaped pulse), the execution of a specific instruction block is initiated via AP bus. Among other information, that instruction block contains the pattern start address, the pattern length and the duration of a time slice for that particular shape call, see below. To perform a particular shaped pulse, it is sufficient to send the (2-byte) address of an instruction block to a waveform generator, plus a one-byte command that starts its execution (5 AP bus words in total).

[1] The name of the file includes the experiment name, the user name, the process-ID of the pulse sequence executable, and the suffix `.RF` (see also Chapter 5, "Submit to Acquisition: go," on page 55). The name alone for `Acqproc` allows the association of such a file with a given Acode file for a specific experiment of a particular user.

- The final execution of a pulse shape or decoupling pattern is (normally) triggered by a fast status line (there is one fast line per waveform generator, see Section 9.2, “Fast Bits,” on page 88).

How Are Patterns Stored in a Waveform Generator?

To better understand the functioning of the waveform generator, let’s now have a look at the organization of its RAM:

Waveforms and modulation patterns are stored in the waveform generator as a *generic pattern* that describe the principal amplitude and phase modulation and the *relative duration* of each slice of a shaped or modulated pulse sequence element. The actual length of a slice duration unit is defined in each instruction block that calls that particular shape. This way, each shape only needs to be stored once. It can be called an almost infinite number of times with *individual duration scaling* for every single call.

The shapes or pattern are stored in 32-bit words organized as follows (see also the schematic drawing in Figure 21):

- 8 bits for the duration count—any element of a shape can be one up to 255 time slices long. The duration of a time slice is not defined in the pattern, but will be given through the instruction block.
- 10 bits for the amplitude—this allows setting the amplitude in 1024 steps (1 to 1024[2]) on the linear modulator. On the UNITY*plus* these are the 10 most significant bits of the 12-bit linear modulator; via AP bus the full amplitude range (4096 steps) can be addressed: the smallest step on the waveform generator corresponds to four amplitude steps on the linear modulator. On UNITY spectrometers, the linear modulator was 10 bits only, and these could be fully addressed via the waveform generator only.
- 11 bits (10 bits on the UNITY) for the phase of the pattern element—out of this, 2 bits are the 90-degree phase shift, the remaining 9 bits (8 bits on UNITY) are the small-angle phase shift (360 quarter-degree steps, or 180 half-degree steps on the UNITY). On the transmitter board, this phase shift is *added* to the current phase shift (as set by the pulse programmer through two fast lines and the AP bus) such that the 90-degree and small-angle phase shifting within a shaped pulse occurs *on top of* any preexisting phase shift.
- 1 bit for the transmitter gate—this can also be regarded as an additional amplitude bit that allows setting the amplitude to zero. Note that the regular amplitude bits have a gap between the positive amplitudes and “negative amplitudes” (amplitudes with phase inversion). Taking into account 180- degree phase shifts (and there are many pulse shapes with phase inversion), the “regular” amplitude settings are –1024 to –1 and 1 to 1024; the zero is accessible only through the transmitter gate bit (the linear modulator does not have a zero amplitude).
- 2 bits are spares. These bits are not used or addressed by any regular software and hardware up to now. These could possibly be used to trigger additional gates or other devices *during* a waveform or decoupling pattern (this would currently require changing `psg/wg.c` via `psggen` mechanism, because there is no function or feature that would allow setting these bits).

[2] In the shape or modulation files in `shapelib`, these amplitude values are represented by numbers ranging from 0 to 1023 (1 less than actually obtained); this has sometimes led to confusion.

Patterns or shapes can be between 1 and almost 64 Kwords long. Note that the 64 Kword pattern memory is shared among all shapes and instruction blocks for the active experiment. A variable size part of the waveform generator memory is used for instruction blocks.

The waveform generator shape definition is incomplete insofar as it only contains a relative duration in each pattern word. In essence, this allows replacing successive shape slices with identical phase and amplitude by a single slice with a larger duration count. The vast majority of the pulse shape definitions have a (default) duration count of 1 in each slice. In modulation patterns, pulse angles are translated to duration counts; therefore, decoupler modulation pattern frequently use slices with larger duration counts, and the slice duration can be defined in units between 1 and 90 degrees.

Waveform Generator Instruction Words

Patterns are stored at the high end (0xFFFF in 32-bit words is the last memory location) of the RAM address range. Instruction blocks are stored at the bottom end, from address 0x0000 up. A typical instruction block for an rf pattern (pulse shape or modulation pattern) consists of six 32-bit words. Based on the 3 most significant bits, we can distinguish 8 different types of instruction words. The structure and the contents depend on the word type: bits 29 to 31 are the word type, bit 28 is reserved in all types for (default) transmitter gating, the rest of the block is variable. Major parts of the instruction word are often not used and contain no further information.

Using Table 9, let's look at the various kinds of instruction words (unused parts of the instruction words were left white).

Table 9. Waveform generator instruction words

Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IB_START	0	0	0	G	L				RAM start address																							
IB_STOP	0	0	1	G					RAM stop address																delay count							
IB_SCALE	0	1	0	G					loop count								amplitude scale															
IB_DELAYTB	0	1	1	G	delay time count, 50 nsec units																											
IB_WAITHS	1	0	0	G																												
IB_PATTB	1	0	1	G	pattern time count, 50 nsec units																											
IB_LOOPEND	1	1	0	G																												
IB_SEQEND	1	1	1	G																												

Each instruction block starts with a `IB_START` instruction word that specifies the 16-bit starting address of the associated pattern block; the next instruction word, `IB_STOP`, defines the (16-bit) address of the first word behind the pattern. Note that multiple pattern are stored next to each other without start or end marks.

The `IB_STOP` instruction can also contain a duration count (0 to 255) for an optional delay preceding the pattern in bits 0 through 7. This duration count goes together with a `IB_DELAYTB` instruction that defines the duration time base in a 28-bit number (0 to 268,435,455). The time base is in 50-nanosecond units (a 20 MHz clock frequency is used for the waveform generator timer); therefore, the time base can be up to 13.42 seconds. Together with a duration count, this permits specifying a delay of up to 3422

seconds, or almost an hour. The minimum duration time base is 200 nanoseconds. Bit 28 determines whether the transmitter is on during that delay or not. The `IB_DELAYTB` instruction (together with the duration count in the `IB_STOP` instruction) could be used to specify additional delays preceding and following a shaped pulse or other pattern. It is currently not used for regular shaped pulses or programmed decoupling (see below).

The `IB_SCALE` instruction has two purposes. It allows specifying an amplitude multiplier for shaped gradients (see Section 16.6, “Using a Waveform Generator for Shaping Gradient Pulses,” on page 195), and it also defines an 8-bit loop counter for cycling with a defined loop count over shapes or pattern. For pulse shapes, the loop counter is set to 1. Programmed decoupling normally uses infinite looping (with the loop count set to 1).

The `IB_WAITHS` instruction makes the waveform generator to wait for a trigger signal on its dedicated pulse programmer high-speed line before starting to execute a pattern.

The `IB_PATTB` instruction defines the pattern time base the same way as the `IB_DELAYTB` for the delay time base. The pattern time base can be between 0.2 microseconds and 13.42 seconds—together with a duration count of 1 to 255 in the pattern definition, *each slice* of a shaped pulse or pattern can be between 0.2 microseconds and 3422 seconds long.

The `IB_LOOPEND` instruction is used in connection with pattern looping, and the `IB_SEQEND` instruction word terminates each instruction block.

Waveform Generator Data File

Upon typing `go`, all pattern and instruction blocks for the current experiment are collected in a single file `/vnmr/acqqueue/expn.username.PID.RF` (`PID` being the process-ID of the pulse sequence executable), or `/vnmr/acqqueue/acqi.RF` for data generated through `go('acqi')`. This file consists of short headers consisting of 4 unsigned integers, each followed by either an instruction block or a data block. The headers consist of the AP bus address of the target waveform generator, the starting address in the RAM of the waveform generator, the number of 32-bit words (- 1) in the instruction block or pattern, and a “spare” number that has no real function in the software, except that it allows the reader to distinguish between instruction blocks (0xabcd) and data blocks (0xfedc). The file ends with a 32-bit trailer containing the pattern `0xa5b6c7d8`. From the headers, the acquisition CPU “knows” which waveform generator (i.e., which AP bus address) a block needs to be sent to, and at what (RAM) address in the target device it needs to store the data (or instruction) block.

Unlike Acode (or NMR data structures), the command `od` is of little help in inspecting waveform generator data, because all the (32-bit) words are complex bit patterns, either instruction words or pattern data. On the other hand, pattern data have a fixed structure, and there are only eight different kinds of instruction words, and also the headers are very simple.

Therefore, with the aid of the information found in `/vnmr/psg/wg.c`, it is relatively easy to write a little program that decodes these data. Let's look at a simple example[3]:

```
INSTRUCTION BLOCK:
AP address = 0x0c18, WG start address =   0000,      6 words
-------------------------------------------------------------
0x08ffcc00  IB_START:   RAM start address = 0xffcc
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xa000063f  IB_PATTB:   Time count = 1600 (0.00008000 sec / 80.00 usec)
0xe0000000  IB_SEQEND:  End of instruction block

RF DATA BLOCK:
AP address = 0x0c18, WG start address = 0xffcc,     36 words
-------------------------------------------------------------
count  amplitude   phase  gates
-----  ---------  ------  -----
   3     1024     180.00
   4     1024       0.00
   2     1024     180.00
   .....
   .....
   4     1024     180.00
   2     1024       0.00
   3     1024     180.00

END OF FILE
```

This is a data file for waveform-generator-based WALTZ-16 decoupling (see also below for a more detailed interpretation of the data contents). The file begins with an instruction block, followed by a data block. Of course, there are often several instruction and data blocks—there is at least one instruction block per data block. If the same waveform or pattern is used several times with different parameters in the same pulse sequence, there may be several instruction blocks referring to the same data block. All instruction blocks are collected at the beginning of the file, followed by the data blocks in the order of their usage in the pulse sequence (the order of the data blocks is actually not relevant).

Both blocks are for the decoupler waveform generator (AP bus address `0x0c18`[4]). The instruction block will be stored at location `0` in the waveform generator RAM, and the data block at location `0xffcc`. The instruction block is 6 words long, the pattern is 36. The data block address is referred to in the `IB_START` instruction word, the address in `IB_STOP` is `0xffcc` + `0x24` = `0xfff0` (65484 + 36 = 65520). The delay count in the same instruction word is the default; there is no delay in this instruction block. The instruction block ends with the instruction word `IB_SEQEND`.

The other parts of the instruction block and the data block will be explained in Section 16.4, "Using Waveform Generators for Programmed Modulation," on page 177 below.

[3] The decoding shown here was produced using a C program `wgdecode` from the VNMR user library (part of `userlib/bin/apdecode`).

[4] The observe channel waveform generator has AP bus address `0x0c10`, `0x0c48` is the address for the second decoupler channel, `0x0c40` is for the third decoupler channel.

Executing Waveform Generator Patterns

Through the AP bus and AP interface, the acquisition CPU can not only write to waveform generator RAM, but it can also write *directly into the output register* (output buffer) and set the output status of any waveform generator. For each call, it sends the *address of an instruction block* (using Acode instruction `102`, `WG3`) and *control codes* (using Acode instruction `101`, `WGCMD`) into the waveform generator *control register* (see also the Acode examples in the following sections). Control code 5 is used to start a shaped pulse, control code 7 is used for programmed decoupling, and control code 1 for starting shaped gradients. Termination of the execution of an instruction block is achieved with control code 0, which causes the waveform generator to halt at the end of the current pattern.

For programmed modulation, it is normally not desirable to wait for the end of the pattern: in such a case the instruction block execution is aborted with control code 0x80 (`RESET`). If the instruction block contains an `IB_WAITHS` instruction word, the execution of an instruction block is halted until the dedicated high-speed line is set to ON by the pulse programmer. This allows for an accurate coordination of waveform generator events with events that are directly controlled by the pulse programmer.

Instruction blocks (i.e., the associated pattern) can be looped for a predefined number of cycles (the `IB_STOP` instruction word contains an 8-bit loop count, allowing for up to 255 loop cycles), or they can be looped infinitely, with interruption either at the end of a pattern (loop cycle) or immediately, depending on whether code 0 is sent (“soft stop”) or control code 0x80 (`RESET`), see above. It is also possible to loop over sequences of instruction blocks (in which case all but the last instruction block have bit 27 set in the `IB_START` instruction).

16.3 Using Waveform Generators for Shaped Pulses

There is one statement per rf channel for performing a simple shaped pulse:

```
shaped_pulse(name,width,phase,rx1,rx2);
decshaped_pulse(name,width,phaser,rx1,rx2);
dec2shaped_pulse(name,width,phase,rx1,rx2);
dec3shaped_pulse(name,width,phase,rx1,rx2);
```

All these statements have the same set of five arguments: `name` is the base name (without “`.RF`” extension) of a pulse shape file in `/vnmr/shapelib` or `$vnmruser/shapelib`; the other arguments are exactly the same as for an ordinary `rgpulse` statement (see also Section 4.2, “How Do Pulses Work?,” on page 40). It turns out that the four statements are actually macros, all of them calling the same statement `genshaped_pulse` (the macros are defined in `/vnmr/psg/macros.h`):

```
genshaped_pulse(name,width,phase,rx1,rx2,g1,g2,device);
```

The last argument is the device (`OBSch`, `DECch`, `DEC2ch`, and `DEC3ch` for the four statements above, see the footnote on page 40). The two additional arguments `g1` and `g2` are additional delays surrounding the shaped pulse (but with the transmitter turned on). Because they can be defined in waveform generator instruction block, both are set to 0.0 in all four macro calls listed above (no statement currently uses these delays).

There are also statements for two and three simultaneous shaped pulses for the observe and decoupler, or the observe and the two first decoupler channels, respectively:

```
simshaped_pulse(n1,n2,w1,w2,ph1,ph2,rx1,rx2);
sim3shaped_pulse(n1,n2,n3,w1,w2,w3,ph1,ph2,ph3,rx1,rx2);
```

Also these are macro calls to the following functions:

```
gensimshaped_pulse(n1,n2,w1,w2,ph1,ph2,rx1,rx2,g1,g2,dev1,dev2);
gensim3shaped_pulse(n1,n2,n3,w1,w2,w3,ph1,ph2,ph3,rx1,rx2,  \
                    g1,g2,dev1,dev2,dev3);
```

With these two statements, it is possible to perform simultaneous shaped pulses on *any* combination of two or three rf channels. Note that with the presence of the `rfchannel` parameter, these “gen” functions should not longer be required.

Programming Shaped Pulses: An Example

To illustrate what has been explained in the previous sections, let’s now take a practical (simple) example.

The Pulse Sequence

We take the `sh2pul` pulse sequence, which is basically a simple `s2pul` sequence with the two rectangular pulses replaced by shaped pulses:

```
#include <standard.h>
pulsesequence()
{
   char p1pat[MAXSTR], pwpat[MAXSTR];
   getstr("p1pat",p1pat);
   getstr("pwpat",pwpat);
   ....
   status(A);
      hsdelay(d1);
   status(B);
      shaped_pulse(p1pat,p1,zero,rof1,rof2);
      hsdelay(d2);
   status(C);
      shaped_pulse(pwpat,pw,oph,rof1,rof2);
}
```

The Shape Definition

We set both the `p1pat` and the `pwpat` parameter to `'gauss'`, which causes the file `/vnmr/shapelib/gauss.RF` to be interpreted when typing `go`. This file (the “`.RF`” extension indicating that it contains a pulse shape definition) consists of several columns, each line defining one pulse slice:

- The phase angle of the slice in degrees; phases can be both positive or negative; phases will be converted to positive values between 0 and <360 degrees (better: the corresponding positive range of phase shift units) internally.
- The amplitude, in values between 0.0 and 1023.0 (the file interpreter reads fractional numbers; therefore, shape files can use fractional numbers throughout, but this is not a requirement). Note that these numbers correspond to real amplitudes of 1 up to 1024. If this column is not specified, the amplitude defaults to 1023.0 (1024). The values are rounded off internally, hence any positive fractional number up to 1023.0 is allowed (because it may be delivered from a shape calculation program).
- The relative slice duration, in values between 1.0 and 255.0; the default for this column is 1.0.
- The gate settings: 1 turns on the transmitter gate (`TXON`), 2 turns on the first spare line, 4 turns on the second spare line. This defaults to 1.0 for pulses (transmitter

on) and is normally omitted. The values for the gates can just be added in column 4, resulting in the possibilities shown in Table 10.

Table 10. Waveform generator gate control for pulse shapes

Value	*TXON*	*Spare #1*	*Spare #2*
0	OFF	OFF	OFF
1	ON	OFF	OFF
2	OFF	ON	OFF
3	ON	ON	OFF
4	OFF	OFF	ON
5	ON	OFF	ON
6	OFF	ON	ON
7	ON	ON	ON

- The gate field (value 0.0) can be used to generate a slice with zero output, to compensate for the fact that amplitude 0.0 in reality is amplitude 1. For short (hard) shaped pulses starting with a non-zero (internal) phase shift it may be desirable to precede the shape with a slice with the phase of the first real slice, but with the gate turned off.

The file `shapelib/gauss.RF` defines a Gaussian pulse with 256 slices:

```
#
# Gaussian Pulse:    256 points, 5-sigma
# This pulse is amplitude modulated and selectively excites
# a bandwidth (Hz) approximately equal to 2e+6/pulse_length
# (usec).
0.00000011.3651.000000
0.00000012.1891.000000
0.00000013.0661.000000
...
(120 lines deleted)
...
0.0000001016.0001.000000
0.0000001018.5141.000000
0.0000001020.4741.000000
0.0000001021.8771.000000
0.0000001022.7191.000000
0.0000001023.0001.000000
0.0000001022.7191.000000
0.0000001021.8771.000000
0.0000001020.4741.000000
0.0000001018.5141.000000
0.0000001016.0001.000000
...
(120 lines deleted)
...
0.00000013.9991.000000
0.00000013.0661.000000
0.00000012.1891.000000
```

The Waveform Generator Data File

If both parameters `p1pat` and `pwpat` in the `sh2pul` sequence are set to `'gauss'`, we get the following waveform generator data file for `p1=1024` and `pw=256`:

```
INSTRUCTION BLOCK:
AP address = 0x0c10, WG start address =   0000,      6 words
------------------------------------------------------------
0x08feef00  IB_START:   RAM start address = 0xfeef
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xb000004f  IB_PATTB:   Time count = 80 (0.00000400 sec / 4.00 usec)  TXON
0xe0000000  IB_SEQEND:  End of instruction block

INSTRUCTION BLOCK:
AP address = 0x0c10, WG start address = 0x0006,      6 words
------------------------------------------------------------
0x08feef00  IB_START:   RAM start address = 0xfeef
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xb0000013  IB_PATTB:   Time count = 20 (0.00000100 sec / 1.00 usec)  TXON
0xe0000000  IB_SEQEND:  End of instruction block

RF DATA BLOCK:
AP address = 0x0c10, WG start address = 0xfeef,    257 words
------------------------------------------------------------
count  amplitude   phase  gates
-----  ---------  ------  -----
   1       12       0.00  TXON
   1       13       0.00  TXON
   1       14       0.00  TXON
   ....
   (120 lines deleted)
   ....
   1     1017       0.00  TXON
   1     1019       0.00  TXON
   1     1021       0.00  TXON
   1     1022       0.00  TXON
   1     1023       0.00  TXON
   1     1024       0.00  TXON
   1     1023       0.00  TXON
   1     1022       0.00  TXON
   1     1021       0.00  TXON
   1     1019       0.00  TXON
   1     1017       0.00  TXON
   ....
   (120 lines deleted)
   ....
   1       14       0.00  TXON
   1       13       0.00  TXON
   0        1       0.00

END OF FILE
```

Because we have used the same waveform twice, we get *two instruction blocks* (at locations 0000 and 0006) *referring to the same pattern block* starting at location `0xfeef`, ending before location `0xfff0`. The pulse is triggered by the high-speed line from the pulse programmer, hence the instruction `IB_WAITHS`. There are 256 slices (actually 257, as we will see). For a pulse duration of 256 microseconds, this results in a slice duration of 1 microsecond. Since each slice has a duration count of 1, the duration time base for the first pulse is 1 microsecond (20 counts of 50 nanoseconds each).

The TXON bit is not only set in the pattern itself, but also in the IB_PATTB instruction word. The second pulse was selected four times longer; therefore, the duration time base is 4 microseconds. Of course, the duration time base is rounded off to the *50 nanoseconds timing resolution* of the waveform generator clock (100 nanoseconds on UNITY spectrometers). For short pulses and pulses with a large number of slices or total duration counts, the software may report round-off errors if they distort the pulse length substantially.

In the pattern block, we see that the software automatically adds a slice with zero amplitude (amplitude value 1, TXON is not set). During all other slices, the transmitter is gated on. This is the default for the case that only three fields are specified in the shape definition file. It is the waveform generator that turns on the transmitter during a shaped pulse; therefore, there is no need to gate the transmitter explicitly using xmtron and xmtroff.

Shaped Pulses in the Acode

The Acode, and the timing during the experiment, differs in two aspects between UNITY and UNITY*plus* spectrometers (apart from inherent timing differences with respect to the AP bus):

- On UNITY*plus*, shaped pulses are terminated when the high-speed line goes down. On UNITY, the waveform generator needs to be stopped using a “soft stop” by sending a control code 0 (stop at the end of the pattern). Therefore, there is a 2-word AP bus delay (4.3 microseconds) after shaped pulses on a UNITY; whereas on a UNITY*plus*, there is no implicit post-pulse delay due to AP bus traffic.
- There is a propagation delay in the waveform generator relative to the pulse programmer timing. On UNITY*plus*, this propagation delay is 450 nanoseconds. On UNITY, it is 1.5 microseconds (this is an approximation and varies slightly from system to system). This delay needs to be taken care of, or else the pulse programmer might abort the waveform generator or start some other event while the shaped pulse is still executing. To correct for the delay, shaped pulses are explicitly delayed by the expected waveform generator propagation delay.

Here is the Acode for the first of the above two pulses created on a UNITY spectrometer:

```
323   317   211     102   WG3             AP addr 0x0c10, IB addr = 0x0000
326   320   214     101   WGCMD           AP addr 0x0c10, WFG cmd =   0x05
329   323   217      16   SETPHAS90       CH1   zero
332   326   220     150   HighSpeedLINES  (void)
335   329   223     150   HighSpeedLINES  RXOFF
338   332   226     151   EVENT1_TWRD     10.000 usec
340   334   228     150   HighSpeedLINES  RXOFF WFG1
343   337   231     151   EVENT1_TWRD     1.500 usec
345   339   233     152   EVENT2_TWRD       255 usec + 1.000 usec
348   342   236     150   HighSpeedLINES  RXOFF
351   345   239     151   EVENT1_TWRD     10.000 usec
353   347   241     150   HighSpeedLINES  (void)
356   350   244     101   WGCMD           AP addr 0x0c10, WFG cmd = 00
```

The WG3 instruction (three AP bus words) sets the instruction block address, the WGCMD instruction (two AP bus words) sets the control register on the waveform generator, see above: control word 0x05 initiates the execution of the instruction block, control word 0 terminates it.

On a UNITY*plus*, the propagation delay is shorter, and the terminating WGCMD instruction is missing:

```
409   403   297     102   WG3             AP addr 0x0c10, IB addr = 0x0000
412   406   300     101   WGCMD           AP addr 0x0c10, WFG cmd =   0x05
415   409   303      16   SETPHAS90       CH1   zero
418   412   306     150   HighSpeedLINES  (void)
421   415   309     150   HighSpeedLINES  RXOFF
424   418   312     150   HighSpeedLINES  RXOFF
427   421   315     151   EVENT1_TWRD     10.000 usec
429   423   317     150   HighSpeedLINES  RXOFF WFG1
432   426   320     151   EVENT1_TWRD       450 nsec
434   428   322     152   EVENT2_TWRD       255 usec + 1.000 usec
437   431   325     150   HighSpeedLINES  RXOFF
440   434   328     151   EVENT1_TWRD     10.000 usec
442   436   330     150   HighSpeedLINES  (void)
```

From the Acode, it can be concluded that it takes 5 AP bus words to start a shaped pulse (causing hidden delays of 10.75 microseconds on UNITY spectrometers, and 5.75 microseconds on the UNITY*plus*), and on the UNITY there is a hidden delay of 4.3 microseconds incurring from two more AP bus words *after* the shaped pulse.

Special Cases

To demonstrate the effect of delays that are executed as part of the instruction block, a the `shaped_pulse` statement was replaced by

```
genshaped_pulse(p1pat,p1,zero,rof1,rof2,0.1,0.2,TODEV);
```

This is, of course, a hypothetical example. It turns out that the `g1` and `g2` delays in `genshaped_pulse` (0.1 and 0.2 seconds in this case, respectively) are in fact built into the instruction block and are not coded in the Acode:

```
INSTRUCTION BLOCK:
AP address = 0x0c10, WG start address =   0000,      9 words
------------------------------------------------------------
0x08feef00  IB_START:   RAM start address = 0xfeef
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0x701e847f  IB_DELAYTB: Time count=2000000 (0.1000000sec/100000.0usec) TXON
0xb000004f  IB_PATTB:   Time count=80 (0.0000040 sec/4.00usec)  TXON
0x50010000  IB_SCALE:   loop count=1  TXON
0x703d08ff  IB_DELAYTB: Time count=4000000 (0.2000000sec/200000.0usec) TXON
0xe0000000  IB_SEQEND:  End of instruction block
```

From an Acode point-of-view, the duration of this shaped pulse is `p1` + `g1` + `g2` (`p1` + `0.1` + `0.2` in this example). There is currently no code or pulse sequence that uses these two additional delays (in all the macro calls to `genshaped_pulse` and similar statements, the delays `g1` and `g2` are set to 0.0).

16.4 Using Waveform Generators for Programmed Modulation

Programmed modulation—whether for broadband decoupling or for spinlock experiments—is nothing but a special kind of shaped pulse with the addition of infinite looping (the looping is the main reason for calling it via specific statement, and not with a `shaped_pulse` call). Apart from this, the requirements and priorities are slightly different for programmed modulation; hence, the pattern definition differs slightly from the definition of shaped pulses.

Programming Pattern Decoupling and Spinlock Experiments

Software for programmed modulation using a waveform generator can be set up in several ways.

Decoupler and Modulation Control

The simplest way to use a waveform generator is for programmed decoupling controlled through `status` fields in the pulse sequence (see "Implicit Gating" on page 53). Using the waveform generator this way requires no special programming effort at all—it works with any pulse sequence (as long as a waveform generator is available).

The primary control of the transmitter gates is done through the status-related parameters `dm`, `dm2`, and `dm3`. This has nothing to do with any kind of modulation. The modulation mode depends on the parameters `dmm` (decoupler modulation mode), `dmm2`, and `dmm3` for the three decoupler channels: if these parameters are set to `'c'` in the current status field, no modulation occurs. All other settings produce some kind of phase modulation on the corresponding rf channel. This modulation is *independent* of the gate (`dm`, `dm2`, `dm3`) setting! Note that whenever pulses should be performed on a channel, the corresponding modulation mode parameter must be set to `'c'` in the current status field; otherwise, the modulation is turned un asynchronously and produces random phases on any pulses in that same status field.

There are two possibilities for modulating a decoupler (through `status` fields): using a "hardware modulator," or using a waveform generator (if one is available). The hardware modulator can be regarded as "software (a given modulation pattern) that is "programmed" (wired) in hardware and cannot be changed (except by changing or modifying the modulator hardware). The standard modulators only operate on the 90- and 180-degree phase shift lines (some even just on 180-degree phase shifting), small-angle phase shifting and amplitude modulation is excluded. On UNITY and earlier systems, the following modulation modes are available through a hardware modulator (for literature references, see the manual *VNMR Command and Parameter Reference*):

`'f'`	"fm-fm" or swept square-wave modulation
`'g'`	GARP-1 decoupling (UNITY*plus* only) (100 pulses with odd pulse angles and 180 degrees phase shifts)
`'m'`	MLEV-16 decoupling (UNITY*plus* only) (based on 90_x-180_y-90_x composite inversion pulses)
`'n'`	noise modulation
`'r'`	square-wave modulation (UNITY*plus* only)
`'w'`	WALTZ-16 modulation (based on 90_x-180_{-x}-270_x composite inversion pulses)
`'x'`	XY-32 modulation (UNITY*plus* only) (based on 90_x-180_y-90_x composite inversion pulses)

On the UNITY*plus*, there is also a mode `'u'` for selecting external (user-supplied) hardware modulation.

Most of these methods (definitely `'g'`, `'m'`, `'w'`, and `'x'`) need to be calibrated (i.e., for a given decoupler strength, the length of the 90-degree pulse must be determined). The modulator then needs to be triggered at the speed or frequency of the 90-degree pulses. In practice, the modulation frequency (4 times the decoupler field strength, in Hz) is entered directly using the `dmf`, `dmf2`, or `dmf3` parameter.

Waveform Generator Control through the status Statement

For systems with waveform generator on a decoupler channel, the character `'p'` in a `dmm` (or `dmm2`, or `dmm3`) field activates waveform generator-based modulation. Also here, the modulation rate is defined through the `dmf` (`dmf2`, `dmf3`) parameter, but unlike the hardware modulator (where the `dmf` frequency is generated as input to the modulator), with the waveform generator `dmf` is converted back into a 90-degree pulse width, which is then used to determine the length of a duration unit in the waveform generator instruction word.

A major advantage of the waveform generator is that it is freely programmable. Any decoupling modulation can be programmed, including methods using small-angle phase shifting, and methods with amplitude modulation (we might even think of using shaped pulses for decoupling).

Setting `dmm` to `'p'` does not define a modulation mode, but rather selects modulation hardware (the waveform generator). In this case, the modulation mode is defined using the string parameter `dseq` (or `dseq2`, `dseq3` for the other decoupler channels). The value of this parameter is the basename of a file in `/vnmr/shapelib` or `$vnmruser/shapelib` (just the body of the name, without "`.DEC`" extension). Although this file is stored in the same directory as pulse shapes (`shapelib/*.RF`), it has a totally different format:

```
# WALTZ-16 Broadband Decoupling Sequence
270.0   180.0
360.0     0.0
180.0   180.0
270.0     0.0
 90.0   180.0
180.0     0.0
360.0   180.0
180.0     0.0
270.0   180.0
270.0     0.0
360.0   180.0
180.0     0.0
270.0   180.0
 90.0     0.0
180.0   180.0
360.0     0.0
180.0   180.0
270.0     0.0
#
270.0     0.0
360.0   180.0
180.0     0.0
270.0   180.0
 90.0     0.0
180.0   180.0
360.0     0.0
180.0   180.0
270.0     0.0
270.0   180.0
360.0     0.0
180.0   180.0
270.0     0.0
 90.0   180.0
180.0     0.0
360.0   180.0
180.0     0.0
270.0   180.0
```

Different from pulse shapes, which normally are described as amplitude and phase function in the time axis, modulation modes are described more often as a sequence of pulses of a certain tip angle and a certain rf phase. The amplitude remains constant throughout the modulation pattern in the vast majority of the case. For this reason, the format of the modulation files (`*.DEC`) in `shapelib` is different from the one for shaped rf pulses, even though it also has up to four columns describing pulse length, phase, amplitude and the transmitter gate setting:

- The first column describes the *tip angle in degrees* (positive number only).
- The second column defines the phase of the slice (equivalent to column 1 of a pulse shape file).
- The third field describes the amplitude (equivalent to column 2 of a pulse shape file) and defaults to 1023.0 (full amplitude on the linear modulator). As decoupler modulation is usually constant, this column is normally omitted.
- The last field controls the waveform generator gate settings. If a waveform generator is activated through status fields, there is an implicit gating function (`decon` and `decoff`) activated through the `status` statement, which overrides any transmitter gate settings in the modulation pattern definition, but the gating for the two spare lines (see Table 10) remains active[5].

For translating the above pattern file into a waveform generator file, the software needs one more parameter that translates the tip angle into duration counts. We need to specify whether the tip angle in the pattern file is to be interpreted as multiples of 90 degrees, or whether the tip, angle is in multiples of a smaller angle. The `dres` parameter specifies the tip angle resolution. For WALTZ and MLEV-type modulation `dres` should be set to 90, for GARP-type decoupling it is normally set to 1 in order to get the most accurate tip angles (see also below).

Explicit Programming

Waveform generator-based modulation can also be turned on explicitly within the pulse sequence. There are two classes of statements for this: one to switch on and off the modulation, and the other that also handles the gating and runs for a given number of cycles. The following statements are used to switch on the waveform generator-based modulation:

```
obsprgon(name,pw90,pwres);
decprgon(name,pw90,pwres);
dec2prgon(name,pw90,pwres);
dec3prgon(name,pw90,pwres);
```

where `name` is the base name (without "`.DEC`" extension) of a pattern file in `$vnmruser/shapelib` or `/vnmr/shapelib`, `pw90` is the length of a 90-degree pulse (allowing the function to calculate the absolute duration of each pattern element), and `pwres` is the tip-angle resolution, which is needed to determine the length of a duration unit and the number of duration units per pattern element. These four statements are actually macros that call a single C function:

```
prg_dec_on(name,pw90,pwres,device);
```

[5] The two transmitter gate signals are combined (ORed). If either of the two devices (the pulse programmer or the waveform generator) turns the transmitter on, it will be on. In other words, gating the transmitter on explicitly in the pulse sequence deactivates (overrides) control through the pattern file. Of course, this does not apply to the two spare gates on the waveform generator.

where the `device` argument is the rf device (`OBSch`, `DECch`, `DEC2ch`, and `DEC3ch` for the four statements above, see the footnote on page 40). For switching the modulation off again, there is another set of four statements:

```
obsprgoff();
decprgoff();
dec2prgoff();
dec3prgoff();
```

These statements are all macros that call a single C function:

```
prg_dec_off(2,device);
```

The `2` in the first argument causes a hard abort (reset) in execution of the waveform generator pattern; a 0 in lieu of the 2 would create a soft stop, allowing the waveform generator to stop at the end of the pattern[6].

Note that these functions do not turn on the transmitter gate, and the default transmitter gating in the waveform generator pattern is *off*. In other words, unless the transmitter is gated on explicitly in the pattern file (through an odd value in column four), the transmitter has to be gated on explicitly in the pulse sequence using statement such as `xmtron` and `xmtroff`, `decon`, and `decoff`; for example:

```
obsprgon("mlev17",1.0/dmf,dres);
xmtron();
delay(getval("mix"));
obsprgoff();
xmtroff();
```

The sequence of the statements may be relevant. If the `xmtron` precedes the `obsprgon` call, the rf is turned on unmodulated during the AP bus traffic to the waveform generator (five AP bus words or 5.75 microseconds on a UNITY*plus*, 10.75 microseconds on UNITY spectrometers). On the other hand, with the above coding, there is a gap between any preceding time event (e.g., a pulse) and the start of the modulation. Similar considerations apply to the end of the modulation on UNITY spectrometers, where there is a 4.3 microseconds hidden delay (there is no implicit delay behind the modulation on a UNITY*plus)*. Which solution is best depends on the pulse sequence—no general recommendation can be given here.

The advantage of the `obsprgon` and `obsprgoff` mechanism is that the modulation can be turned on at any point in a pulse sequence, and it can be left on while other events (pulses, delays—anything that does not use the modulated channel) take place, and it can be turned off at any point later in the sequence. There are cases, however, like the TOCSY sequence, where we would just like to perform a mixing time with spinlock on one channel, and there it would seem simpler if we could just activate the waveform generator with a single command. In fact, such statements exist:

```
spinlock(name,pw90,pwres,phase,nloops);
decspinlock(name,pw90,pwres,phase,nloops);
dec2spinlock(name,pw90,pwres,phase,nloops);
dec3spinlock(name,pw90,pwres,phase,nloops);
```

Once more, these are all macros (as defined in `/vnmr/psg/macros.h`), calling a single C function:

```
genspinlock(name,pw90,pwres,phase,nloops,device);
```

[6] This is usually undesirable, because the modulation would then continue asynchronously up to the end of the pattern (i.e., the pulse programmer would *not* wait for the end of the modulation before continuing with the next FIFO words).

The first three arguments to these statements are the same as with `obsprgon` and related functions. In the `phase` argument, the basic 90-degree phase shift for the spinlocking pulse train can be specified (often the spinlock pulses are subjected to phase cycling), and in the `nloops` argument to the macros, the number of pattern loop cycles is specified (`nloops` must be an *integer*, not a floating point number!). This is not the parameter the spectroscopist is normally interested in. In most cases, the relevant parameter is the spinlock time. This requires calculation of the possible number of loop cycles in the pulse sequence, which is no real problem for a given modulation pattern:

```
ncycles = (int)(getval("mix")/(64.667*pw90));
spinlock("mlev17",pw90,90.0,zero,ncycles);
```

It would not make sense to define the name of the decoupling pattern as a variable, because the number of 90-degree pulse lengths per pattern depends on the pattern[7].

The `spinlock` statements have some other peculiarities. Compared to the `obsprgon` type of statement, the default gate setting in inverted so that, by default, the transmitter gate is turned on instead of off. This gating does not happen through the pulse programmer (using `xmtron`, `decon`, etc,), but it is done in the waveform generator data file (see below) to allow for gate switching *during* the modulation, as for instance required with the “clean” spinlocking sequences that have delays between the elements of each composite pulse. This seems correct and desirable, but it can create confusion.

As long as no gating occurs within the pattern, any `spinlock` statement can easily be replaced by the following C construct using an `obsprgon` type of function—this even avoids calculating the number of loop cycles:

```
xmtron();
txphase(zero);
obsprgon("mlev17",pw90,90.0);
delay("getval("mix")");
obsprgoff();
xmtroff();
```

This is more than just equivalent—it would even allow replacing the delay with a sequence of other events: performing pulses and delays *during* the spinlock period, without interrupting the modulation.

Now consider the case where somebody wanted to perform a “clean”-type of spinlock. In this case the `xmtron` and `xmtroff` statements cannot be used, because this would override any gating done by the shape (the transmitter would be continuously on). The new construct will be:

```
txphase(zero);
obsprgon(“clean_mlev17”, pw90, 90.0);
delay(“getval(“mix”));
obsprgoff();
```

Now we need a pattern definition that includes the gating column. To specify a gating value (column 4) we also need to specify the (default) amplitude. A programmer could now save some work by specifying only the gating where the transmitter must be on (because the transmitter gate is off by default):

[7] Having the pattern name as a variable would require parsing the pattern to find out about the pattern length, which of course would exceed the scope of normal pulse sequence programming (and probably the capabilities of many pulse sequence programmers).

```
# Clean MLEV-17
 90.0     0.0   1023.0   1.0
 90.0   270.0
180.0   270.0   1023.0   1.0
 90.0     0.0
 90.0     0.0   1023.0   1.0
 90.0     0.0   1023.0   1.0
 90.0   270.0
180.0   270.0   1023.0   1.0
 90.0     0.0
 90.0     0.0   1023.0   1.0
....
```

There is no real problem with this definition—as long as it is used together with `obsprgon`! Now suppose somebody wanted to use this same pattern definition for the `spinlock` statement. Suddenly the experiment would not work, because with `spinlock` the transmitter gate in on by default, and we are not explicitly gating the transmitter off during the gaps! The conclusion is that if gating is used *within* a pattern, specify it for *all* slices. This makes the pattern usable with both types of functions:

```
# Clean MLEV-17
90.0     0.0   1023.0   1.0
90.0   270.0   1023.0   0.0
180.0  270.0   1023.0   1.0
 90.0    0.0   1023.0   0.0
 90.0    0.0   1023.0   1.0
 90.0    0.0   1023.0   1.0
 90.0  270.0   1023.0   0.0
180.0  270.0   1023.0   1.0
 90.0    0.0   1023.0   0.0
 90.0    0.0   1023.0   1.0
....
```

The file `/vnmr/psg/wg.c` also defines a statement `gen2spinlock` that allows simultaneous (asynchronous) spinlocking and decoupling on two channels; currently no macro calls this undocumented statement. In general, it is as easy to use the `obsprgon` type of statement to achieve the same thing; therefore, this will not be further discussed here.

How Does Pattern Modulation Work Internally?

Let's first have a look at the Acode generated by *status*-based waveform generator modulation

Acode and Pattern File for Programmed Modulation

The following Acode is generated when using the waveform generator for modulated (broadband) decoupling using the first decoupler channel on a UNITY spectrometer:

```
300   294   188     102   WG3             AP addr 0x0c18, IB addr = 0x0000
303   297   191     101   WGCMD           AP addr 0x0c18, WFG cmd =   0x07
306   300   194     150   HighSpeedLINES  DECUP WFG2
309   303   197     150   HighSpeedLINES  DECUP WFG2
312   306   200     151   EVENT1_TWRD      1000 msec
314   308   202     150   HighSpeedLINES  WFG2
317   311   205     150   HighSpeedLINES  (void)
320   314   208     101   WGCMD           AP addr 0x0c18, WFG cmd = 0x80
```

There are three principal differences to the Acode produced for a shaped pulse:

- The (decoupler) transmitter gate is turned on explicitly (overriding the waveform generator's transmitter gating).
- The waveform generator is started using control code `0x07` (for infinite looping).
- At the end of the modulation period the waveform generator is stopped using control code `0x80` (reset or abort), causing the modulation to halt immediately.

The Acode for a UNITY*plus* is very similar:

```
380   374   268     150   HighSpeedLINES  DEC WFG2
383   377   271     102   WG3             AP addr 0x0c18, IB addr = 0x0000
386   380   274     101   WGCMD           AP addr 0x0c18, WFG cmd =   0x07
389   383   277     150   HighSpeedLINES  DEC WFG2
392   386   280     150   HighSpeedLINES  DEC WFG2
395   389   283     151   EVENT1_TWRD      1000 msec
397   391   285     150   HighSpeedLINES  DEC WFG2
400   394   288     150   HighSpeedLINES  WFG2
403   397   291     150   HighSpeedLINES  (void)
```

The main difference is that on the UNITY*plus* the waveform generator modulation is stopped by resetting the fast status line, and no AP bus traffic is required to achieve this. The Acode was created using `dm='ynn'`, with a `d1` of 1 second.

The Acode produced by the `obsprgon` and `obsprgoff` combined with the `spinlock` type of statements is identical, except that these statements do not turn on the pulse programmer rf gate. For the `obsprgon` type of statement, this is normally done through explicit gating in the pulse sequence using `xmtron`, `decon`, etc., which would make the Acode indistinguishable from the code above.

In the case of `spinlock` and the related statements, the system does *not* use finite looping on the waveform generator. The waveform generator has only an 8-bit loop counter, allowing for 255 loop cycles maximum. The number of loop cycles in spin locking may be much larger than that. Instead, the waveform generator is started with *infinite* looping. From the loop count, the total duration of the spinlock time is calculated. In the Acode the system performs a delay of that length and then stops the waveform generator (using control code 0x80 on a UNITY or by resetting the fast status line on a UNITY*plus*).

Let's see the waveform generator data (instruction block and pattern data) for WALTZ-16 decoupling during a status field:

```
INSTRUCTION BLOCK:
AP address = 0x0c18, WG start address =   0000,      6 words
-------------------------------------------------------------
0x08ffcc00  IB_START:   RAM start address = 0xffcc
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xa000063f  IB_PATTB:   Time count = 1600 (0.00008000 sec / 80.00 usec)
0xe0000000  IB_SEQEND:  End of instruction block

RF DATA BLOCK:
AP address = 0x0c18, WG start address = 0xffcc,     36 words
-------------------------------------------------------------
count  amplitude   phase  gates
-----  ---------  ------  -----
   3     1024     180.00
   4     1024       0.00
   2     1024     180.00
   3     1024       0.00
   1     1024     180.00
   2     1024       0.00
   4     1024     180.00
   2     1024       0.00
   3     1024     180.00
   3     1024       0.00
   4     1024     180.00
   2     1024       0.00
   3     1024     180.00
   1     1024       0.00
   2     1024     180.00
   4     1024       0.00
   2     1024     180.00
   3     1024       0.00
   3     1024       0.00
   4     1024     180.00
   2     1024       0.00
   3     1024     180.00
   1     1024       0.00
   2     1024     180.00
   4     1024       0.00
   2     1024     180.00
   3     1024       0.00
   3     1024     180.00
   4     1024       0.00
   2     1024     180.00
   3     1024       0.00
   1     1024     180.00
   2     1024       0.00
   4     1024     180.00
   2     1024       0.00
   3     1024     180.00

END OF FILE
```

In this case, the parameter `dres` was set to 90, and the duration count in each slice corresponds to the tip angle divided by the resolution (90 degrees) in each slice. The parameter `dmf` (i.e., the inverse 90 degrees pulse length) was set to 12500; therefore, the pattern time units is set to 80 microseconds by the `IB_PATTB` word in the instruction block.

The (infinite) looping is not specified in the instruction block. This is done through the control code (`0x07`) sent to the waveform generator at run time. The instruction and data blocks generated by `obsprgon` (`decprgon`, etc.), are identical to those

generated by `status`-based decoupling (assuming the same parameters and pattern selection): the transmitter gate is off by default.

The waveform generator data are different for the `spinlock` statement and its relatives. Here the transmitter is on by default—not through the pulse programmer (because this would override transmitter gating from the waveform generator), but through the waveform generator itself. The following data are for (decoupler based) MLEV-16 spinlocking:

```
INSTRUCTION BLOCK:
AP address = 0x0c18, WG start address =   0000,      6 words
-------------------------------------------------------------
0x08ffbf00  IB_START:   RAM start address = 0xffbf
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xa000063f  IB_PATTB:   Time count = 1600 (0.00008000 sec / 80.00 usec)
0xe0000000  IB_SEQEND:  End of instruction block

RF DATA BLOCK:
AP address = 0x0c18, WG start address = 0xffbf,     49 words
-------------------------------------------------------------
count  amplitude   phase  gates
-----  ---------  ------  -----
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
```

```
   1     1024     180.00  TXON
   2     1024     270.00  TXON
   1     1024     180.00  TXON
   1     1024       0.00  TXON
   2     1024      90.00  TXON
   1     1024       0.00  TXON

END OF FILE
```

The transmitter gate is switched on throughout the pattern, but not within the instruction block (for pulses, the transmitter would also be on during eventual delays in the instruction block). The parameter `dmf` was again set to 12500.

The Influence of the dres Parameter

The duration count in a modulation pattern should be proportional to the tip angle that is specified in the first column of the pattern file. The number of counts per tip angle depends on the `dres` parameter that defines the tip angle resolution. For modulation pattern like WALTZ-16 or MLEV-16, where all the pulses are in multiples of 90 degrees, `dres` can be set to 90, which gives one count for a 90-degree pulse, two counts for a 180-degree pulse, etc. (as shown in the above examples). For sequences like MLEV-17 or GARP-1 `dres=90` would not work, because it only allows for tip angles in multiples of 90 degrees: all pulses in a pattern should ideally be multiples of the tip angle resolution `dres`; otherwise, they are *rounded to multiples of* `dres` in the final pattern.

Two examples: The MLEV-17 modulation pattern ends with a 60 degrees pulse, all other pulses are 90 or 180 degrees. In order to get an accurate reproduction of that pattern, `dres` must be set to 30, 15, 7.5 or 3.75 (the last value only for the UNITY*plus*). For reasons shown below, the preferred value 30. The GARP-1 modulation pattern consists of 100 pulses with 23 different tip angles ranging from 26 to 268 degrees:

```
# GARP-1 Broadband Decoupling Sequence
 31.0     0.0
 55.0   180.0
258.0     0.0
268.0   180.0
 69.0     0.0
 62.0   180.0
 85.0     0.0
 92.0   180.0
135.0     0.0
256.0   180.0
 66.0     0.0
 46.0   180.0
 26.0     0.0
 73.0   180.0
.....
```

The only way to get an accurate reproduction of such a pattern is with `dres=1`. This produces the following instruction and data blocks for the waveform generator:

```
INSTRUCTION BLOCK:
AP address = 0x0c18, WG start address =   0000,      6 words
-------------------------------------------------------------
0x08ff7a00  IB_START:   RAM start address = 0xff7a
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010000  IB_SCALE:   loop count = 1
0x80000000  IB_WAITHS:  Wait for high-speed line trigger
0xa0000011  IB_PATTB:   Time count = 18 (0.00000090 sec / 0.90 usec)
0xe0000000  IB_SEQEND:  End of instruction block
```

```
RF DATA BLOCK:
AP address = 0x0c18, WG start address = 0xff7a,    118 words
-----------------------------------------------------------
count  amplitude   phase  gates
-----  ---------  ------  -----
  31     1024       0.00
  55     1024     180.00
 255     1024       0.00
   3     1024       0.00
 255     1024     180.00
  13     1024     180.00
  69     1024       0.00
  62     1024     180.00
  85     1024       0.00
  92     1024     180.00
 135     1024       0.00
 255     1024     180.00
   1     1024     180.00
  66     1024       0.00
  46     1024     180.00
  26     1024       0.00
  73     1024     180.00
....
```

The maximum duration count is 255. If a tip angle results in a greater duration count, these counts are spread over several pattern words. In the above example, the parameter `dmf` was left at a value of 12500. This results in a duration unit of 900 nanoseconds, as can be seen from the instruction block above. The smaller the `dres` parameter is set, the smaller a duration unit is obtained.

The minimum duration unit is 200 nanoseconds. Even high modulation rates (large decoupling ranges) can be performed at a high tip-angle resolution (low values for `dres`). It seems that all modulations can be performed with the highest possible tip angle resolution. The only price we seem to pay is that for large tip angles, additional pattern words are required, but the result should be the same.

There is a possible drawback with small values in `dres`, however. The smaller the duration units (which are proportional to `dres` and inversely proportional to `dmf`), the larger a timing error we can get. With a duration unit of 0.5 microseconds and a timing resolution of 0.1 microseconds (UNITY), the round-off error can be up to 10%, on a UNITY*plus* with 50-nanosecond timing resolution it can still be up to 5%. The conclusion is that the smaller a `dres` is selected, the greater the overall timing error. If the duration unit is rounded down by 10%, all pulses in the pattern are too short by 10%. The software checks on the timing error when constructing the instruction block, and any time base (round-off) error above 2% is reported. Of course, we can adjust `dmf` (or the pulse length, the second argument to the `spinlock` and `obsprgon` functions), but this just avoids the error and does not remove the error in the effective tip angle in every single pulse in the pattern[8]!

The conclusion is that the tip angle should be selected *as large as the pattern permits*, and, unless a pattern like GARP-1 requires it, it is better not to work with the minimum value in `dres` and related parameters.

[8] In order to keep the modulation algorithm working accurately, we would have to adjust the power to the new (executable) value of `dmf`: a 5% error in the pulse length requires a power adjustment of about 1 dB.

How small can the tip-angle resolution be? It would not make sense to allow for tip angle resolutions as small as 0.1 degree because such values would lead to extremely small duration units, even at moderate modulation rates, and round-off errors would be obtained very often. Also, the pattern file would be blown up excessively—a 270-degree tip angle would result in 11 pattern words. In addition to that, a tip angle resolution much below one degree does not make sense anyway, because a 90 degree pulse is never calibrated with this accuracy. The software, therefore, prevents setting `dres` and related parameters incorrectly, and the tip-angle resolution in `obsprgon` and `spinlock` type functions in various places:

- The parameter `dres` (as well as `dres2`, `dres3`) has a lower limit of 1 (degree). This can be changed easily using the VNMR `setlimit` command.
- In `/vnmr/psg/cps.c`, values smaller than 1 in these parameters are reset to a minimum of 1. This means, that for status-field controlled modulation `dres` can't be any smaller than 1.
- The `obsprgon` and `spinlock` families of statements limit the tip-angle resolution to a minimum of 0.7 degrees (in `/vnmr/psg/wg.c`). Altering this would require changes to the pulse sequence overhead (i.e., using `psggen` to change the precompiled libraries). Using this lower limit requires using a parameter different from `dres` (this would be reset to 1, see above) or (assuming modified parameter limits in VNMR) using `getval("dres")` as third argument to `obsprgon` / `spinlock`.

16.5 What If a Waveform Generator Is Not Available

For decoupling, the main disadvantage in not having a waveform generator is that the user is limited to the built-in, pre-programmed decoupling methods.

Programmed Decoupling

On a UNITY*plus*, programmed decoupling includes GARP-1, WALTZ-16, MLEV-16, XY-32 and others—a range that should cover all of today's needs for broadband decoupling. There are more limitations on UNITY and earlier spectrometers, where in essence only WALTZ-16 is available for broadband decoupling. This can be an experimental limitation, especially in cases where a large chemical shift range needs to be decoupled, where GARP-1 permits using much lower power, therefore reducing the sample heating. To some degree, it is possible to explicitly program other, synchronous decoupling methods as explicit acquisition with pulses between the sampling points, but for moire complex methods like GARP-1 this is rather complex, to say the least.

For single-broadband UNITY systems performing indirect detection, a waveform generator on the observe channel can be a real benefit, in that it allows for continuous, asynchronous broadband X decoupling. Without waveform generator the only possibility is to explicitly program and acquisition loop with interleaved decoupling pulses. On systems with an output board (63-word FIFO), this is limited to WALTZ-4 and also may suffer from power or duty-cycle problems, because for large proton spectral windows a composite (90_x-180_{-x}-270_x) pulse may easily be longer than the dwell time. On systems with an acquisition control board (1024-word FIFO), XY-32 can be used, which definitely outperforms WALTZ-4.

For spinlocking experiments (modulation on the transmitter channel), only explicitly coded modulation is possible. This is no real limitation, because even complex modulation algorithms (up to several hundred pulses, depending on the pulse programmer) can be implemented using software or hardware looping.

Shaped Pulses

With a slice width of down to 200 nanoseconds, the waveform generator is unique in its capability to shape even short, hard pulses. The range of possibilities for systems without waveform generators depends on the type of instrument:

UNITY and Earlier Spectrometers

Systems without a waveform generator do not have a linear amplitude modulator, so that the only possibility for performing shaped pulses is through rf attenuators. Several hardware conditions must be fulfilled before pulse shaping can be done successfully:

- Linear amplifiers are prerequisite, otherwise the power level cannot be regulated well enough.
- The T/R switch is also required, otherwise (with crossed diodes) the pulse shapes would be distorted dramatically at voltage levels below 0.5 volts (see Section 4.2, “How Do Pulses Work?,” on page 40).
- The standard 63 or 79 dB attenuator does not provide a power range that allows shaping selective pulses (with excitation bandwidths below several hundred Hz). An additional fast, switchable (PIN-diode-based) attenuator is required for the channel on which pulse shaping should occur. Early systems did not have the ports for additional (fine) attenuators. On these, the only way out would be to take the decoupler attenuator and route it into the transmitter channel, in series with the standard attenuator. This leaves the system without decoupler, or with a decoupler with fixed attenuation. Systems with newer AP interface cards have ports for fine attenuators (normally used for solids NMR experiments). These ports can also be used (improperly) for an additional 63-dB attenuator, which then gives the power range required for shaping even very selective pulses.

If these hardware conditions are fulfilled, the `include` file `shape_pulse.c` from `/vnmr/psg` can be used. It contains a statement for performing Gauss- and Hermite-type shaped pulses using attenuators. The limitations are that only built-in (programmed) pulse shapes can be used, small-angle phase shifting is not provided, and the minimum slice width is 10 microseconds (which precludes shaping hard pulses). For more information, see the text of the include file[9]. Shape definitions for waveform generators cannot be used for attenuators, because attenuators are calibrated in dB, whereas the amplitude modulator on the AM/PM transmitter board is linear.

UNITYplus

UNITY*plus* users are in a more favorable position. Both the amplitude and the phase modulators are included with every rf channel and can be addressed via the AP bus, even if no waveform generator is present. This permits using the shape definitions for

[9] More powerful functions have been submitted to the user library, allowing for free shape programming through tables, including more pre-programmed pulse shapes (such as the BURP-type of pulse shapes), or even including small angle phase shifting.

waveform generators, basically allowing for any shapes, even those using small-angle phase shifting. This has been realized in the following statements:

```
apshaped_pulse(name,width,phase,tbl1,tbl2,rx1,rx2);
decapshaped_pulse(name,width,phase,tbl1,tbl2,rx1,rx2);
dec2apshaped_pulse(name,width,phase,tbl1,tbl2,rx1,rx2);
```

which are macros calling the following C function:

```
gen_apshaped_pulse(name,width,phase,tbl1,tbl2,rx1,rx2,device);
```

The arguments are the same as for the statements and macros for waveform generators, with the exception of two *unused* phase table names (`t1` to `t60`) that must be supplied with every call (for multiple calls of these functions, new names have to be supplied with every call). These tables are used to internally store the amplitude and phase vectors that are read out of the specified *standard waveform generator shape file* (`shapelib/*.RF`).

The C function `gen_apshaped_pulse` decodes the shape file and stores the phase and amplitude vectors in auto-incrementing tables. It then performs the shaped pulse in a real-time loop. The minimum slice length is given by the time it takes to set the power on the linear modulator and the small-angle phase shift via the AP bus (2.3 + 3.45 microseconds, 5.75 microseconds in total, plus a minimum delay of 0.2 microseconds, resulting in a minimum slice length of 5.95 microseconds). The central part of this function is coded as follows[10]:

```
/*-------------------------------------------------------| Calculate
time spent for AP bus events within each slice  |
+-------------------------------------------------------*/
aptime = POWER_DELAY + SAPS_DELAY;

if ( (plength - aptime) < MINDELAY)
{
   text_error("apshaped_pulse: pulse too short or too many  \
              elements in shape");
   abort(1);
}
/*----------------------------------------+
| set 90 degrees phase, set phase step size |
+----------------------------------------*/
txphase(phs);
stepsize(0.25,OBSch);
/*--------------------------------+
| gate receiver off, do rx1 delay |
+--------------------------------*/
rcvroff();
if (rx1 - aptime > 0.0)
   delay(rx1-aptime);
/*-----------------------------------------------------+
| before turning on the transmitter and entering the  |
| pulse loop preset phase and amplitude for the first |
| slice, to avoid a glitch at the start of the pulse  |
+-----------------------------------------------------*/
```

[10] This is *not* the original coding. To keep the text understandable, low-level functions have been replaced by their standard equivalents that are also used in pulse sequences. The coding shown here is written specifically for the observe channel. The “real” function uses generalized, low-level functions that work for all rf channels.

```
pwrf(pwrtbl,OBSch);
xmtrphase(phstbl);
txphase(phs);
decr(v12);
/*-----------------------------------------------------+
| turn on transmitter, then in a soft loop execute one |
| slice, then set power and phase for the next slice   |
| after the loop do the last slice, then switch rf off |
+-----------------------------------------------------*/
xmtron();
loop(v12, v13);
   delay(plength-aptime);
   pwrf(pwrtbl,OBSch);
   xmtrphase(phstbl);
   txphase(phs);
endloop(v13);
delay(plength);
xmtroff();
rlpwrf(4095.0, OBSch);
rlxmtrphase(zero);
txphase(phs);
/*-----------------------------------------------+
| perform rx2 delay, gate receiver back on again |
+-----------------------------------------------*/
if (rx2 - aptime > 0.0)
   delay(rx2-aptime);
rcvron();
```

In this code, `plength` is the slice length, calculated as pulse length divided by the number of slices in the shape file. Slices with duration counts greater than 1 are translated into multiple slices with a duration count of 1. The real-time variable `v12` contains the number of loop cycles. `pwrtbl` and `phstbl` are the two table names that are given as arguments; they have been “filled” with the amplitude and table values from the shape file. Note that the amplitude values from the table have to be multiplied by 4 in order to get the proper amplitude range, because the waveform generator only addresses the ten most significant bits (0 to 1023), whereas through the AP bus the full range (12 bits, 0 to 4095) can be used.

As an example, if we take the 256-step gaussian pulse that was discussed previously in this chapter, we obtain a rather impressive piece of Acode from this single function:

```
 399   393   287     150   HighSpeedLINES  (void)
 402   396   290      39   ASSIGNFUNC      zero   v12
 405   399   293      31   MULtFUNC       three three   v13
 409   403   297      31   MULtFUNC         v13 three   v13
 413   407   301      31   MULtFUNC         v13 three   v13
 417   411   305      31   MULtFUNC         v13 three   v13
 421   415   309      29   ADDFUNC          v12   v13   v12
 425   419   313      32   DIVFUNC          v13 three   v13
 429   423   317      32   DIVFUNC          v13 three   v13
 433   427   321      32   DIVFUNC          v13 three   v13
 437   431   325      29   ADDFUNC          v12   v13   v12
 441   435   329      32   DIVFUNC          v13 three   v13
 445   439   333      29   ADDFUNC          v12   v13   v12
 449   443   337      32   DIVFUNC          v13 three   v13
 453   447   341      29   ADDFUNC          v12   v13   v12
 457   451   345      16   SETPHAS90       CH1    oph
 460   454   348      68   PHASESTEP       CH1      1 units (0.25 degrees)
 463   457   351     150   HighSpeedLINES  RXOFF
 466   460   354     150   HighSpeedLINES  RXOFF
 469   463   357     151   EVENT1_TWRD     4.250 usec
 471   465   359     105   TABLE 360   size 256, autoinc 1, divn_ret 1, ptr 0
```

```
                         45   49   52   56   60   64   69   73
                         78   84   89   95  102  108  115  123
                        131  139  148  157  166  176  187  198
                        210  222  235  249  263  277  293  309
                        326  343  362  381  401  421  443  465
                        488  512  537  563  590  617  646  676
                        706  738  770  804  838  874  910  948
                        986 1026 1066 1108 1150 1193 1238 1283
                       1329 1377 1425 1474 1523 1574 1626 1678
                       1731 1784 1838 1893 1949 2004 2061 2118
                       2175 2232 2290 2348 2406 2464 2523 2581
                       2639 2697 2754 2812 2869 2925 2981 3036
                       3091 3145 3198 3250 3302 3352 3401 3449
                       3496 3541 3585 3628 3669 3708 3746 3783
                       3817 3850 3880 3909 3936 3961 3984 4005
                       4024 4040 4055 4067 4077 4085 4091 4094
                       4095 4094 4091 4085 4077 4067 4055 4040
                       4024 4005 3984 3961 3936 3909 3880 3850
                       3817 3783 3746 3708 3669 3628 3585 3541
                       3496 3449 3401 3352 3302 3250 3198 3145
                       3091 3036 2981 2925 2869 2812 2754 2697
                       2639 2581 2523 2464 2406 2348 2290 2232
                       2175 2118 2061 2004 1949 1893 1838 1784
                       1731 1678 1626 1574 1523 1474 1425 1377
                       1329 1283 1238 1193 1150 1108 1066 1026
                        986  948  910  874  838  804  770  738
                        706  676  646  617  590  563  537  512
                        488  465  443  421  401  381  362  343
                        326  309  293  277  263  249  235  222
                        210  198  187  176  166  157  148  139
                        131  123  115  108  102   95   89   84
                         78   73   69   64   60   56   52   49
732   726   620      106   TASSIGN table 360    tblrt
735   729   623       59   APChipOUT APaddr 11, reg 150, -logic, 2 bytes
                                     max 4095, offset 0, value tblrt
741   735   629      105   TABLE 630   size 256, autoinc 1, divn_ret 1, ptr 0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
                           0    0    0    0    0    0    0    0
```

```
                              0    0    0    0    0    0    0    0
                              0    0    0    0    0    0    0    0
                              0    0    0    0    0    0    0    0
1002   996   890     106   TASSIGN table 630    tblrt
1005   999   893      65   SETPHASE        CH1f tblrt
1008  1002   896      16   SETPHAS90       CH1    oph
1011  1005   899      28   DECRFUNC         v12
1013  1007   901     150   HighSpeedLINES  RXOFF TXON
1016  1010   904      39   ASSIGNFUNC      zero   v13
1019  1013   907      42   IFMInusFUNC      v12   one Offset =   937
1023  1017   911     151   EVENT1_TWRD     44.250 usec
1025  1019   913     106   TASSIGN table 360    tblrt
1028  1022   916      59   APChipOUT APaddr 11, reg 150, -logic, 2 bytes
                                  max 4095, offset 0, value tblrt
1034  1028   922     106   TASSIGN table 630    tblrt
1037  1031   925      65   SETPHASE        CH1f tblrt
1040  1034   928      16   SETPHAS90       CH1    oph
1043  1037   931      27   INCRFUNC         v13
1045  1039   933      42   IFMInusFUNC      v13   v12 Offset =   911
1049  1043   937     151   EVENT1_TWRD     50.000 usec
1051  1045   939     150   HighSpeedLINES  RXOFF
1054  1048   942      65   SETPHASE        CH1f  zero
1057  1051   945      16   SETPHAS90       CH1    oph
1060  1054   948     151   EVENT1_TWRD     4.250 usec
1062  1056   950    150   HighSpeedLINES  (void)
```

The purpose of the real-time math section at the beginning (not shown in the C code above) is to store the number of slices in a real-time variable without using `initval` (see also “New Real-Time Numeric Constants” on page 97). The most prominent feature in the following section are the two tables containing all 256 amplitude and phase values of that pulse, taking up a full Kbyte (two times 256 16-bit words) of Acode space. Of course the second table could, in theory, be reduced to a single constant `zero`, but the program cannot “know” that this shape file contains no phase changes.

Apart from the amount of Acode that is necessary to perform such a pulse through Acode, and some limitations with respect to the modulation rate, everything seems to be fine, and `apshaped_pulse` in fact is a reasonable replacement for a waveform generator for many experiments, in particular, selective excitation. There are, however, a few fundamental differences and limitations to this solution. Some of them lie in the way the waveform generator works; others are a consequence of the above coding and of the way how phase shifting works.

One problem lies in small-angle phase shifting. Any internal phase shifting on a waveform generator is performed *on top of* any current quadrature and small-angle phase shifts. This cannot be emulated in software. The internal phase shifting in `gen_apshaped_pulse` is set via `xmtrphase` or equivalent statements, with work in an absolute frame. They even reset any existing quadrature (90-degree) phase shift! The latter is added in again after the `xmtrphase` call (`gen_apshaped_pulse` therefore works properly on top of any quadrature phase shift), but this cannot be done for the small-angle phase shifting. Small-angle phase shift coaddition is not provided, and hence `gen_apshaped_pulse` does not work properly on top of small angle phase shifting.

The other problem lies in the fact that quadrature phase shift needs to be *reestablished* after each `xmtrphase` or equivalent call. The statement `txphase` (or its equivalents for the current channel) is called once for each slice, which implies that

autoincrementing tables cannot be used as phase variables to `apshaped_pulse` or equivalent macros.

Over all, what are the limitations of the `apshaped_pulse` approach?

- The minimum slice length is 5.95 microseconds, compared to 0.2 microseconds with the waveform generator. Real short hard pulses cannot be shaped;
- Phase and amplitude changes don't occur simultaneously. The phase and the amplitude profile of the pulse are shifted against each other by 3.45 microseconds;
- Slices with the transmitter gate switched off are simply skipped;
- `apshaped_pulse` and related functions don't work as expected on top of small angle phase shifting;
- The phase variable cannot be an autoincrementing table;
- Simultaneous shaped pulses are not possible;
- `apshaped_pulse` generates lots of Acode, which may possibly limit its use for multidimensional (3D, 4D) experiments;
- Even though the Acode size may theoretically allow for a single shape of up to about 4000 slices (see also "Acode Size Limitations, Acode Buffering" on page 83), such long shapes will most likely lead to FIFO underflow. On a UNITY or UNITY*plus* system, the FIFO is only filled at a rate of below one word per 20 µsec[11], which may in many cases be the real rate-limiting step.

Nevertheless: `apshaped_pulse` permits performing most selective excitation experiments (even demanding ones like those using shifted laminar pulses[12] or SS pulses[13]) perfectly without a waveform generator, and some of the inherent limitations can be bypassed by programming measures.

16.6 Using a Waveform Generator for Shaping Gradient Pulses

The waveform generator was first designed for shaping field gradients and rf pulses in imaging experiments. Only later it was adapted for high-resolution machines. By the addition on one component (the amplitude multiplier), a UNITY*plus* waveform generator can be converted for shaping field gradients (of course, the AP address has to be changed by reconfiguring its jumper settings).

The gradient waveform generator works along the same principles as an rf waveform generator, except that it controls the amplitude of a single field gradient instead of phase and amplitude of an rf signal and rf gates. Therefore, a gradient pattern word has a totally different layout, as shown in Table 11.

Bits 0 to 7 are again the duration count (same as for the rf pattern), bits 8 to 23 form a 16-bit field gradient amplitude value (-32768 to 32767), and the bits 24 to 31 are unused (field gradients are dc by nature, so there is no `phase` parameter[14]). Pulsed field gradients may not only be scaled in the time scale, often (mostly, in imaging

[11] This is under optimum conditions, without lots of Acode overhead from soft looping or table access, etc. (both are used in the `apshaped_pulse` function). The actual transfer rate for the `apshaped_pulse` function has not been determined.

[12] S.L. Patt, *J. Magn. Reson*. **96**, 94 (1992).

[13] S.H. Smallcombe, *J. Am. Chem. Soc*. **115**, 4776 (1993).

[14] Instead, the amplitude is signed (it can be positive or negative), but the rf amplitude is measured and regulated in magnitude only.

Table 11. Comparison of waveform generator pattern words

Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rf	SP		T	phase															amplitude											duration			
Gradient									amplitude																					duration		

experiments) gradients are scaled in their amplitude. For an imaging experiment, a spatial dimension can be encoded by stepping a gradient through an array of (positive and negative) values.

With the rf definition of a waveform generator, we would have to define as many shapes as there are amplitude values. To avoid this, field gradient shapes are made generic in two ways: they can be scaled in duration (the same way as rf shapes and pattern) and in amplitude. The amplitude value in the pattern is not the final one. The gradient waveform generator has a built-in multiplier that multiplies the amplitude value in every pattern word with a scalar value in the `IB_SCALE` instruction word of the corresponding instruction block (see also “Waveform Generator Instruction Words” on page 169).

Most imaging experiments use at least one stepped gradient for spatial encoding. Often, there are 32, 64, 128, or 256 different gradient increments. This means that in the pattern file, there are that many instruction blocks pointing to the same data (pattern) block, each of them with a different scaling factor in the `IB_SCALE` instruction word.

In terms of basic data handling, the gradient waveform generator is treated the same way as its rf equivalent. Here is a (hypothetical) data file for a “hsine”-shaped gradient without amplitude scaling:

```
INSTRUCTION BLOCK:  Z GRADIENT
AP address = 0x0c30, WG start address =   0000,      5 words
------------------------------------------------------------
0x08ff7100  IB_START:   RAM start address = 0xff71
0x20fff001  IB_STOP:    RAM stop  address = 0xfff0, delay count = 1
0x40010001  IB_SCALE:   loop count = 1, amplitude scale = 1
0xa000b88b  IB_PATTB:   Time count = 47244 (0.00236220 sec / 2362.20 usec)
0xe0000000  IB_SEQEND:  End of instruction block

DATA BLOCK:  Z GRADIENT
AP address = 0x0c30, WG start address = 0xff71,    127 words
------------------------------------------------------------
count  amplitude
-----  ---------
   1      810
   1     1620
   1     2429
   ...
   (57 lines deleted)
   ...
   1    32704
   1    32744
   1    32764
   1    32764
   1    32744
   1    32704
   ...
   (57 lines deleted)
   ...
```

```
   1     2429
   1     1620
   1      810
   1        0

END OF FILE
```

The gradient shape definition files are also stored in `shapelib`, but with the extension "`.GRD`". Their format is simpler than the format for rf shapes and modulation pattern. The two columns in the file contain the gradient amplitude and the duration count (usually one count per slice):

```
#
#      half sine for GRADIENTS
#
  810    1.0
 1620    1.0
 2429    1.0
(57 lines deleted)
32704    1.0
32744    1.0
32764    1.0
32764    1.0
32744    1.0
32704    1.0
(57 lines deleted)
 2429    1.0
 1620    1.0
  810    1.0
    0    1.0
```

Chapter 17. Pulsed Field Gradients

The most prominent difference between pulsed field gradients (PFG) and non-PFG pulse sequences is the addition of statements that generate the pulsed field gradients. In certain PFG techniques (like coherence pathway selection and multiple-quantum filtering using pulsed field gradients), little or no phase cycling is used. Some PFG sequences use pulses with constant phase only and may therefore look simpler than their non-PFG equivalents. Where non-PFG pulse sequences eliminate artifacts and unwanted signals by phase cycling (e.g., subtraction), PFG sequences dephase unwanted coherences (i.e., make them non-observable, PFG experiments are often less subject to spectral artifacts than their non-PFG equivalents).

The PFG accessory involves a dc and audio frequency power amplifier with linear amplitude control through the AP bus, connected to a special probe with a Z gradient coil. The field gradient coil is shielded. While generating a strong Z field gradient inside (at the sample), an external compensating field (with opposite sign) is generated, such that the total field outside the gradient coil is greatly diminished. This helps reduce eddy currents in the rest of the probe body and in the metal walls of the magnet dewar (although eddy currents can't be totally avoided).

17.1 Pulse Sequence Statements for PFG Gradient Control

The statements for gradient control in typical PFG experiments are simple. They reflect the straightforward, scalar nature of a (linear) field gradient. A single parameter determines the amplitude of the gradient, which can have values between -32768.0 and 32767.0 (the gradient amplifier uses a 16-bit DAC, negative values are permitted and usually required in most PFG experiments).

What also simplifies programming pulsed field gradients is the fact that the gradient amplifier is constructed practically noise-free; therefore, there is no need for blanking it during off-intervals. The amplifier can be put into "standby" mode by setting the VNMR parameter `pfgon` to `'nnn'`, while setting `pfgon` to `'nny'` turns on the (Z) gradient amplifier. There is also no fast line (gating) involved with gradient control. Typical gradient pulses are on the order of milliseconds; therefore, it is more than sufficient to set the gradient amplitude with the AP bus. Even gradient shaping can be done this way (see Section 17.2, "Shaping Pulsed Field Gradients," on page 202).

The most basic statement to set the gradient amplitude is `rgradient`[1]:

```
rgradient(gid,amplitude);
```

For PFG experiments, the first argument (`gid`, the gradient identifier) is `'z'` or `'Z'`, the second argument (`amplitude`) is a number between `-32768.0` and `32767.0` (a floating point number of type double).

[1] There is also a statement `vgradient` that allows defining the amplitude from real-time variables. Typical PFG sequences use a few, predefined gradient levels that do not vary within the sequence or from transient to transient; therefore, using `vgradient` would be an unnecessary complication. Up to now, `vgradient` has only been used in imaging sequences (see also Chapter 21, "(Micro)Imaging Experiments," on page 239).

A typical pulse sequence construct for a gradient pulse of length `gz1` and amplitude `gz1lvl` using the `rgradient` function could be written as follows:

```
rgradient('Z',getval("gz1lvl"));
delay(getval("gz1"));
rgradient('Z',0.0);
```

The higher-level statement `zgradpulse` allows reducing these lines to a single call:

```
zgradpulse(amplitude,duration);
```

Using this statement, the above three lines for a gradient pulse of duration `gt1` and amplitude `gz1lvl` would be written as

```
zgradpulse(getval("gz1lvl"),getval("gt1"));
```

Although this statement has the simplicity of the statements for rf pulses (that use gating through fast lines), one should still not forget the “underlying three lines” (i.e., that there is AP bus traffic before and after the actual gradient pulse, taking up a finite time). While the time spent in AP bus traffic is negligible compared to the duration of the gradient pulse itself, it is long enough to cause considerable dephasing for large chemical shift ranges. It is, therefore, strongly recommended to compensate for this time in the pulse sequence, in particular during a refocusing interval:

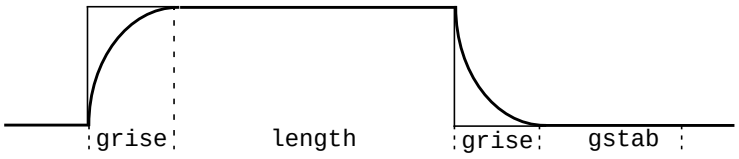
```
rgpulse(pw,t1,rof1,0.0);
delay(tau-rof1);
rgpulse(pw,t1,rof1,0.0);
zgradpulse(getval(“gz1lvl”), gt1);
delay(tau-gt1-2.0*GRADIENT_DELAY-rof1);
rgpulse(pw,t1,rof1,0.0);
```

Up to now we have taken gradient pulses as ideal, rectangular pulses, which obviously was a simplification because any gradient change is slowed down by eddy currents (this applies to switching both on and off). In first approximation, the time constant of the decay of these eddy currents is given by the geometry and the construction of the gradient coil and its surroundings and should not depend on the gradient strength. Corrective delays have often been implemented in pulse sequences using pulsed field gradients to compensate for gradient pulse imperfections like the finite slew rate. Such corrections (in addition to eddy current compensation) are most necessary in imaging experiments, where eddy currents are a much bigger problem than for high-resolution PFG experiments (see also Chapter 21, “(Micro)Imaging Experiments,” on page 239).

The first Varian PFG sequences were written with microimaging experiment concepts in mind. A typical gradient pulse was programmed as follows:

```
rgradient('Z',gz1lvl);
delay(gz1+grise);
rgradient('Z',0.0);
delay(grise);
delay(gstab);
```

This generates the following timing scheme (times not shown proportionally):



`grise` is supposed to be the time intervals before and after the pulse that are affected by the eddy currents (delayed gradient buildup, delayed gradient decay), and `gstab` is the time required to reestablish full homogeneity after a gradient pulse. It turns out that after a typical gradient pulse of a few milliseconds, it takes about 50 microseconds until the system has recovered from the gradient, such that there are no observable phase errors (to regain full amplitude may take slightly longer). It is certainly not a good idea to have a gradient pulse followed immediately by an rf pulse (or data acquisition), but as long as some delay (of around 50 microseconds) follows the gradient pulse, there should be no need for an additional delay `gstab`.

It can be assumed that eddy currents affect both the gradient buildup and gradient turn-off times the same way (i.e., what we lose at the beginning of the gradient pulse we regain at the end of the pulse). The gradient pulse is not quite rectangular (neither is any pulse!). All that counts is the *area* (amplitude times duration) of the gradient pulse because that determines the amount of dephasing achieved. To avoid transversal relaxation and spin diffusion, it is desirable to have short gradients. On the other hand, very strong gradients require strong amplifiers that produce lots of noise and, therefore, affect the overall homogeneity.

With PFG probes, the eddy current time constants are definitely below 10 microseconds. Even if the time constants for turning on and off the gradient pulse were slightly different, that difference would be negligible, because their magnitude is only fractions of a percent of the total pulse duration (typically a few milliseconds). Under this assumption, it is certainly *wrong* to compensate for eddy current effects using a finite delay `grise`, as this prolongs the gradient pulse.

Most PFG pulse sequences use a gradient pulse to refocus magnetization (coherence) that was dephased by a preceding pulse. This can only work if the two gradient pulses have very accurate and well-defined gradient areas. `grise` in the above scheme would distort the dephasing ratio between two gradient pulses with different areas, because

$$\frac{(\texttt{gt1} \times \texttt{gz1lvl})}{(\texttt{gt2} \times \texttt{gz2lvl})} \neq \frac{((\texttt{gt1} + \texttt{grise}) \times \texttt{gz1lvl})}{((\texttt{gt2} + \texttt{grise}) \times \texttt{gz2lvl})}$$

Still, many spectroscopists are using `grise`. By intuition, you might think that these delays “take care of eddy currents.” In fact, effects can be seen in many experiments where below certain values for `grise` (typically 10, sometimes more) some pulse sequences would simply not work properly. What has happened in those cases was that the gradients may have been slightly out of balance due to the addition of `grise` to their lengths (if they had different lengths or amplitudes) but by an amount that is not noticeable yet. What made the experiments work is that the second `grise` (in the above scheme) acted as gradient recovery delay.

In many pulse sequences, an rf pulse follows a gradient pulse, and for these cases we need to insert a recovery delay; otherwise, the rf pulse generates phase errors. This is most critical in multiple-quantum filtering experiments, where phase or amplitude distortions after a gradient pulse can seriously hamper the performance of the sequence. For most sequences a recovery delay of 10 to 20 microseconds is adequate and sufficient. For multiple-quantum filtering experiments (like gradient MQCOSY or E.COSY), the recovery time should be adjusted to 50 to 60 microseconds.

In conclusion, `grise` delays are not recommended (as shown in the scheme above), but if an rf pulse follows a gradient pulse, implementing a gradient recovery delay is *strongly recommended*. This delay can be made part of an `rgpulse` call:

```
zgradpulse(getval("gz1lvl"),getval("gt1"));
rgpulse(pw,v1,gstab,0.0);
```

17.2 Shaping Pulsed Field Gradients

It has been suggested that using trapezoidal gradients (or gradients with other shapes) would allow minimizing eddy current effects. Because PFG probes use actively shielded gradient coils, eddy current effects are minimized; therefore, it was found that it is not necessary to use shaped gradients. Also, in order to minimize losses due to transverse relaxation and spin diffusion, it is desirable to use short, strong gradient pulses. A shaped gradient by definition has a lower duty cycle and is, therefore, longer than a rectangular gradient with the same "area." Thus, for most liquids applications, it is undesirable to use shaped gradients.

Still, there may be situations where shaped gradient pulses have advantages. For example, in diffusion experiments with very high gradient strengths, a gradient coil with high-inductive load may cause transition problems to the gradient amplifier when turning on the gradient. A trapezoidal gradient may alleviate such problems.

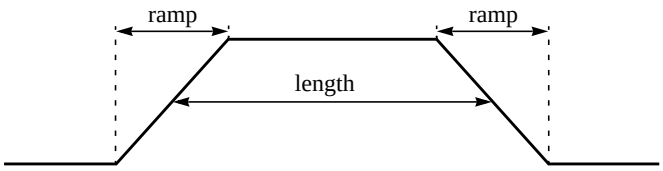
How can gradients be shaped? For imaging, a gradient control unit with waveform generator provides for an easy access to shaped gradients, but the PFG accessory does not include a waveform generator. Therefore, gradients have to be shaped through the AP bus, similar to the approach taken in the `apshaped_pulse` function (see Section 16.5, "What If a Waveform Generator Is Not Available," on page 189).

A function for creating a (trapezoidal) shaped gradient is shown below:

```
rampgrad(amp, length, ramp)
double amp, length, ramp;
{
  int i, steps;
  double iramp, initval, incr_val;
  if (length >= MINDELAY)
  {
    if (ramp > length)
    {
      fprintf(stdout,"ramp parameter larger than  \
              gradient delay\n");
      abort(1);
    }
    steps = (int) (amp / 1000.0);
    if ((double) steps * GRADIENT_DELAY > ramp)
      steps = (int) (ramp / GRADIENT_DELAY);
    if (steps > 1)
    {
      incr_val = amp / steps;
      initval = incr_val;
      iramp = ramp / (double) steps - GRADIENT_DELAY;
      for (i=0; i<steps; i++)
      {
        rgradient('Z',initval);
        delay(iramp);
```

```
        initval += incr_val;
      }
      rgradient('Z',amp);
      if (ramp-(double)steps*(iramp + GRADIENT_DELAY) > MINDELAY)
        delay(ramp - (double) steps * (iramp + GRADIENT_DELAY));
    }
    else
    {
     rgradient('Z',amp);
     if (ramp - GRADIENT_DELAY >= 2e-7)
       delay(ramp - GRADIENT_DELAY);
    }
    if (length - ramp >= MINDELAY) delay(length - ramp);
    if (steps > 1)
    {
      for (i=0; i<steps; i++)
      {
        rgradient('Z',initval);
        delay(iramp);
        initval -= incr_val;
      }
      rgradient('Z',0.0);
      if (ramp - (double)steps*(iramp + GRADIENT_DELAY) >=  \
          GRADIENT_DELAY)
        delay(ramp - (double)steps * (iramp + GRADIENT_DELAY));
    }
    else
    {
      rgradient('Z',0.0);
      if (ramp - GRADIENT_DELAY >= MINDELAY)
        delay(ramp - GRADIENT_DELAY);
    }
  }
}
```

This function creates a shape with a linear ramp at the beginning and at the end of the gradient pulse. The amplitude is changed in steps of 1000 ADC units or 1/32 of the maximum amplitude (or in bigger steps if the ramp duration isn't long enough). The shape is calculated such that the gradient "area" is the same as for a rectangular gradient of the same specified length:



Obviously, when the specified length is less than the duration of one ramp (specified in the third argument), this could not be maintained; therefore, an error message is produced.

17.3 PFG Experiments Using Homospoil Pulses

Spectroscopists who don't have access to a PFG accessory might ask whether it is possible to perform PFG experiments using the standard gradient (shim) coils. There are several problems with this approach:

- The achievable gradient strength is very limited. This would lead to very long gradient pulses (probably 10 to 100 milliseconds) causing losses due to transverse relaxations and spin diffusion.
- The shim coils are not shielded; therefore, the recovery time from a shim coil gradient pulse is several milliseconds. The gradient recovery delays would further increase the losses.
- There are no user-accessible functions that allow setting shim DACs from within a pulse sequence (although certainly such a function could be created).
- Even if a user-accessible function were created, its use would still be very unhandy. At least for one of the polarities, the achievable amplitude may be strongly limited if the standard gradient amplitude is far off from zero.

Of course, we can decide to use the homospoil pulse to avoid the above problems. The homospoil pulse can be triggered with existing statements in a pulse sequence (see "Delays With Homospoil Pulse" on page 38), but the amplitude for the homospoil pulse is not under software control. In particular, the sign of the homospoil pulse amplitude cannot be altered, which excludes a large number of PFG experiments.

Overall, it would certainly be possible to perform a limited range of (simple) PFG experiments using homospoil pulses, but there are rather serious limitations, especially for larger molecules and other samples with short transverse relaxation times. This method will therefore not be discussed any further here.

Chapter 18. Acquiring Data

A prominent feature in most VNMR pulse sequences is the absence of code dealing with data acquisition. Of course, there is a statement that does data acquisition:

```
acquire(points,dwell_time);
```

where `points` is the number of data points to be acquired and `dwell_time` is the time interval between the sampling trigger pulses, or the time that is apparently taken to acquire a complex data point[1]. Because the receiver and ADC acquire data in complex points (real + imaginary), `points` (the number of values measured) must be a multiple of two.[2] At the same time, the `dwell_time` is equal to the inverse spectral window and determines the largest frequency that can be measured. Higher frequencies are folded in (this is also called “aliasing”).

Larger spectral windows require shorter dwell times. With a 0.2-microsecond minimum time event, the maximum spectral window which the pulse programmers can possibly handle is 5 MHz[3].

Still, most pulse sequences end with the last pulse. How then is the acquisition done?

18.1 Implicit Acquisition

One thing that `acquire` does (apart from coding the specified number of sampling intervals) is to increment a global variable `acqtriggers` that initially (before calling the function `pulsesequence`) is set to zero. If a pulse sequence does not contain the `acquire` statement (i.e., if there was no explicit acquisition, see Section 18.2, “Explicit Acquisition,” on page 208), `acqtriggers` still is set to zero after the call to `pulsesequence`, and the software automatically completes the sequence with an implicit acquisition.

The following code segments are found in the function `createPS` from within the module `psg/cps.c`:

```
acqtriggers = 0;
...
pulsesequence();   /* generate Acodes from USER pulse sequence */
...
test4acquire();    /* if no acquisition done yet, do it */
write_Acodes();    /* write out generated lc, auto & instructions */
return;
```

[1] The `acquire` statement is coding `points/2` periods (with a duration as specified by the second argument), each starting with an ADC trigger pulse.

[2] If using the Output board (63-word FIFO), points must be 2 or a multiple of 64.

[3] There is no need for “off” times in-between sampling intervals. Every time event can trigger a complex data point if the CTC bit (the “command to convert” or acquisition trigger) is set to high: unlike normal fast lines, the CTC is electronically reformed into a very short trigger pulse at the very beginning of each sampling interval (see also Chapter 9, “Pulse Programmers,” on page 85).

Function `test4acquire` is found in `psg/hwlooping.c`; and starts with the lines:

```
if (acqtriggers == 0)            /* No data acquisition yet? */
{
   if (nf > 1.0)
   {
      text_error("Number of FIDs (nf) Not Equal to One\n");
      abort(0);
   }
   if (ap_interface < 4)
      HSgate(rcvr_hs_bit,FALSE);        /* turn receiver On */
   else
      SetRFChanAttr(RF_Channel[OBSch],SET_RCVRGATE,ON,0);
   for (i = 1; i <= NUMch; i++)         /* zero HS phaseshifts */
      SetRFChanAttr(RF_Channel[i],SET_RTPHASE90,zero,0);
   acqdelay = alfa + (1.0 / (beta * fb) );
   G_Delay(DELAY_TIME,acqdelay,0);      /* alfa delay */
   acquire(np,1.0/sw);                  /* acquire data */
}
...
```

From these lines of code, we can extract the following information on the implicit acquisition:

- Implicit acquisition does *not* work for multi-FID experiments (`nf > 1`), such as multiecho imaging experiments (see Chapter 21, “(Micro)Imaging Experiments,” on page 239) or sequences like COCONOESY (combined COSY—NOESY), i.e., sequences that acquire more than one FID *within* the pulse sequence.
- The receiver is gated on as part of the implicit acquisition. It should be harmless if this is forgotten within the pulse sequence.
- The quadrature phase of all transmitters is reset to zero before acquiring data. In combination with receiver phase cycling, this avoids coherent signal buildup in case of an rf leakage, which could produce a large, narrow peak (“glitch”) in the center of the spectrum (with signal averaging this can only hurt in the case of the observe transmitter itself, or with decoupler transmitters operating at the frequency of the observe transmitter).
- The acquisition is preceded by a delay `alfa+(1.0/(beta*fb))`. Thus, the `alfa` and the filter group delays allow for a proper timing of the first data point (see “Considerations for the Delays Following the Last Pulse” on page 50).
- The number of points (values) acquired, and the dwell time used in the implicit acquisition are given by the two parameters `np` and `sw` (these C variables have the same name as the VNMR parameter from which they are initialized).

After the conditional branch for the implicit acquisition, the function (i.e., also for the case of an explicit acquisition) `test4acquire` resets any PFG or imaging gradient, adds the instruction for the housekeeping delay (see Section 18.5, “Housekeeping Delays,” on page 214), and finally adds the code to jump back to the NSC (next scan) instruction in the Acode (see “The Instruction Section” on page 79).

The implicit acquisition through `acquire` is performed as a hardloop with a certain number of delays and with the CTC (command to convert) set to high. The number of dwell times per hardloop depends on the parameter and hardware configuration. With the pulse sequence controller board (2048 word loop FIFO), up to 1024 or 2048 dwell times can be coded per hardloop (depending on whether single- or double precision

time words are used). With a maximum of 32767 loop cycles, this allows for up to 64 or 128 million data points to be acquired, or half as much using the acquisition controller board (1024 word loop FIFO). From a parameter point of view, `np` is limited to multiples of 64. Residual dwell times are performed after the hardloop[4]. On systems with an output board (63-word FIFO), the acquisition is performed as hardloop with 16 or 32 dwell times (64 data points) per loop cycle, allowing for up to 1 or 2 million data points[5].

The number of points has limitations other than those imposed by the looping capability of the pulse programmer. Primarily, the standard memory size on the HAL board limits the number of points. With the standard 2-Mbyte HAL memory, up to 1,048,576 data points can be added in single precision (16-bit) acquisitions or up to 524,288 points in double precision (32-bit) acquisitions. It is possible to expand the RAM on the HAL board: the MC-68000 CPU address space is 24 bits, or up to 16 Mbytes (some of the address space is used up by the acquisition CPU, ROM, and status registers), theoretically allowing for up to 4 or 8 million data points.

The STM board (see Chapter 7, “Digital Components,” on page 65) uses a 24-bit counter to count the number of data points added up per transient. That number is compared with the number stored in the first long word (`LC->np`) in the `LC` structure of the Acode (see Section 8.2, “Looking at Acode,” on page 69). If these numbers don’t match, an error message “`number of points acquired not equal to np`” is produced. The STM counter can handle up to 16,777,216 points. This number is higher than those imposed by the memory and address space limitations—the STM board is not a limiting factor for the number of points.

The standard ADC for liquids NMR can handle data rates of up to 200,000 points per second (spectral windows of up to 100,000 Hz). The STM board can safely handle such data rates[6]. For larger spectral windows, a wideline receiver has to be used. The two wideline receiver/ADC models available can sample data at rates of up to 2 and 5 MHz (corresponding to 4 or 10 million values per second), much more than the STM board can possibly handle. Therefore, these boards are equipped with a fast on-board buffer memory, into which the data are acquired and from which the data are piped into the STM board (2-MHz digitizer) or transferred onto the HAL board (5-MHz digitizer).

The latest wideband (12-bit) ADC board allows for spectral windows of up to 5 MHz (as much as the pulse programmer can handle) and is equipped with 512 Kbyte of buffer memory, resulting in a maximum of 131,072 data points[7]. The older wideline ADC board allowed for spectral windows of up to 2 MHz and was equipped with 64 Kbyte of buffer memory, allowing a maximum of 16,384 points (8192 complex) only.

---

[4] There is no technical reason why `np` could not be any multiple of two (even values below 64 should be permissible), but this has not been tested out. At the very least this would require changing the parameter limits for `np`.

[5] With this configuration, all points must be acquired within the hardloop (no extra dwell times following the loop); therefore, the number of points acquired (`np`) *must* be a multiple of 64.

[6] There is also a FIFO buffer between the ADC and the adder on the STM board.

[7] This board has on-board STM functionality.

18.2 Explicit Acquisition

In some cases, it is necessary to code the acquisition explicitly because implicit acquisition lacks the desired functionality. These cases fall into three categories:

- Sequences where a pulse or other event should follow the acquisition, such as in the flipback experiment used sometimes in solids NMR, to recollect residual (spin-locked) proton magnetization after the acquisition time ("forced relaxation").
- Sequences that acquire multiple FIDs within the pulse sequence (`nf > 1`).
- Sequences that require performing pulses or other events in-between the acquisition of single data points[8] (single-point acquisition), as used in sequences with explicit (synchronous) decoupling (see also the second example in Section 14.3, "Hardware Loops," on page 150), or for sequences with so-called multipulse line narrowing (see also the first example in the same section).

Typically, single-point acquisitions are coded using a hardloop, such as the following:

```
initval(np/2.0,v14);
starthardloop(v14);
   acquire(2.0,1.0/sw <length of other events in dwell time>);
   <additional events>
endhardloop();
```

Because two data points are acquired per loop cycle, the number of loop cycles is equal to half the number of data points. To obtain properly scaled and referenced spectra, it is essential that the time specified in the `acquire` statement, plus all the other events in the same dwell time (including hidden delays!), make up *exactly* the expected dwell time, `1/sw`. In experiments with multipulse line narrowing, this is not so relevant because the scaling in these spectra is distorted anyway due to chemical shift scaling.

Of course, it is possible to acquire more data points per hardloop, for example, to allow changing the phase of the pulses between the data points:

```
initval(np/8.0, v14);
starthardloop(v14);
   acquire(2.0,(1.0/sw-2.0*rof1-pw)/2.0);
   rgpulse(pw,v1,rof1,rof1);
   delay((1.0/sw-2.0*rof1-pw)/2.0);
   acquire(2.0,(1.0/sw-2.0*rof1-pw)/2.0);
   rgpulse(pw,v2,rof1,rof1);
   delay((1.0/sw-2.0*rof1-pw)/2.0);
   acquire(2.0,(1.0/sw-2.0*rof1-pw)/2.0);
   rgpulse(pw,v3,rof1,rof1);
   delay((1.0/sw-2.0*rof1-pw)/2.0);
   acquire(2.0,(1.0/sw-2.0*rof1-pw)/2.0);
   rgpulse(pw,v4,rof1,rof1);
   delay((1.0/sw-2.0*rof1-pw)/2.0);
endhardloop();
```

The number of time events in a hardloop must not exceed the loop FIFO size of the pulse programmer (otherwise FIFO errors would result at execution time). This also limits the number of points per `acquire` statement within a hardloop. Because nested hardloops are not possible, `acquire` in a hardloop codes a linear sequence of FIFO words and, in the best case, the number of points that can be collected in a single `acquire` statement within a hardloop is equal to the size of the loop FIFO for single

[8] In practice, this means in-between acquiring *pairs* of data points.

precision timer words, or half that for double precision timer words. Only the second condition is checked at runtime (and aborts the *go* command with an error message if necessary). This is a rather loose test, because there are usually additional time events within the hardloop; otherwise, the entire FID could be collected in a single `acquire` statement without an explicit hardloop (see also Chapter 9, "Pulse Programmers," on page 85 and Section 14.3, "Hardware Loops," on page 150).

With this type of explicit acquisition (and irrespective of parameter limits), `np` is limited to multiples of the number of points acquired per hardloop, unless special provisions are taken in the pulse sequence to acquire any remaining points in separate `acquire` statements outside the hardloop. With explicit acquisition, the user must ensure that the correct (`np`) number of points is acquired with every transient. A mismatch in the number of points acquired leads to the error message "`number of points acquired not equal to np`".

An explicit acquisition can also be made part of a conditional part of the pulse sequence, as in the example of the following fragment of a proton flipback sequence:

```
  ...
  rgpulse(pw,oph,rof1,0.0);
  decphase(zero);
  status(C);
  if (dm[C] == 'y')
  {
    txphase(zero);
    delay(alfa+1.0/(beta*fb));
    acquire(np,1.0/getval("sw"));
    decrgpulse(pw,three,0.0,0.0);
    status(A);
  }
}
```

In explicit acquisition, the user's responsibility is to ensure proper timing for the first data point and to avoid center glitches through rf leakage by resetting the relevant transmitter phases to zero (see also Section 18.1, "Implicit Acquisition," on page 205).

18.3 Multi-FID Sequences

VNMR has the ability to handle data with multiple FIDs per FID file trace (i.e., data from experiments where more than one FID was collected in a single pulse sequence transient). In such a case, all FIDs must be collected using explicit acquisition, as in the following (partial) example of a combined COSY and NOESY pulse sequence:

```
...
status(A);
  hsdelay(d1);
status(B);
  rgpulse(pw,v1,rof1,0.0);
  delay(d2-rof1-4.0*pw/3.1416);
  rgpulse(pw,v2,rof1,rof2);
status(C);
  txphase(zero);
  delay(alfa+1.0/(beta*fb));
  acquire(np,1.0/sw);
  hsdelay(mix-rof2-(alfa+1.0/(beta*fb))-(np/2.0)*1.0/sw-rof1);
  rgpulse(pw,v3,rof1,rof2);
status(D);
```

```
  txphase(zero);
  delay(alfa+1.0/(beta*fb));
  acquire(np,1.0/sw);
}
```

Of course, proper timing of the first data point must be ensured for *all* FIDs that are acquired, to avoid phasing and baseline problems throughout the experiment. Note that the restrictions in the number of points that were discussed in the previous section apply to the *sum* (`np*nf`) of all FIDs acquired in a pulse sequence, not just to the `np` parameter itself, i.e., failure to acquire `nf` FIDs per transient results in the (slightly misleading) error message "`number of points acquired not equal to np`". Other examples of multi-FID acquisitions are discussed in Chapter 21, "(Micro)Imaging Experiments," on page 239.

18.4 Receiver Phase Shifting

There are no fast lines from the pulse programmer that transmit the observe phase into some hardware, so how is receiver phase shifting done? How about small-angle receiver phase shifting? How are NMR signals detected?

Detection of NMR signals

All signals are measured relative to the reference frequency, which is constructed from the local oscillator (L.O.) and the intermediate frequency (I.F.), both of which are usually fixed in phase and frequency during an experiment.

As shown in Figure 22, the UNITY*plus* uses an I.F. of 10.5 MHz (i.e., the local oscillator is 10.5 MHz above the observe frequency). The signal from the probe is first amplified in the **preamplifier** (54 dB gain) and then passed through a first switchable

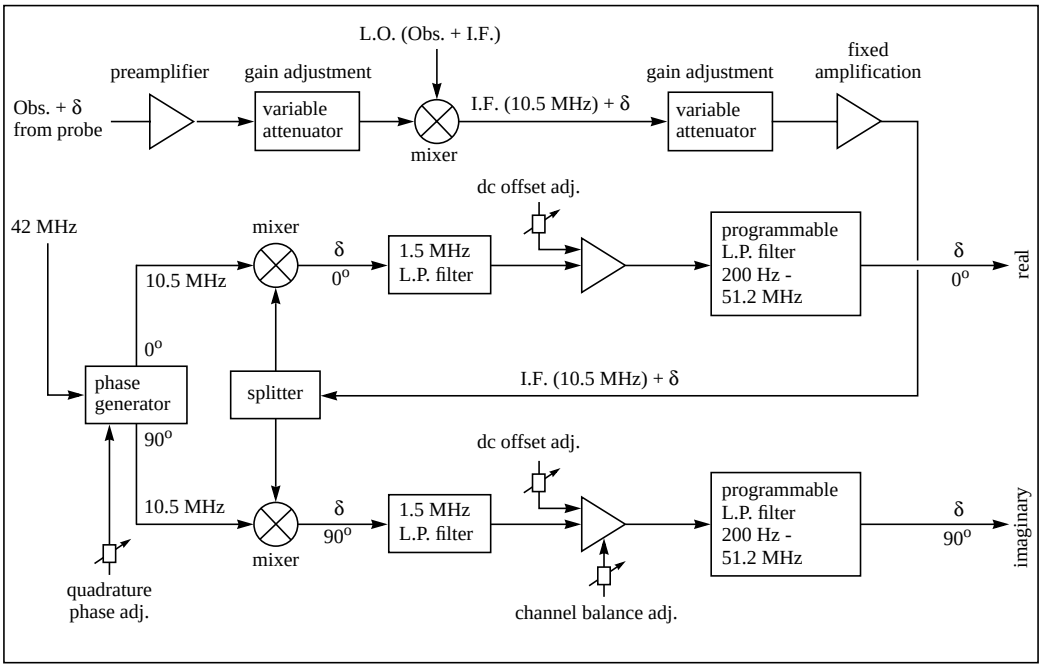


Figure 22. Detection of NMR signals

attenuator (6 and 12 dB, allowing for 0 to 18 dB attenuation, in steps of 6 dB). After that it is mixed with the local oscillator frequency in a double balanced **mixer**, resulting in a 10.5 MHz signal that is modulated with the observed signals (i.e., the difference between the transmitter frequency and the observed signals). All parts behind that first mixer operate at constant frequency, which simplifies the design of the receiver[9].

After the mixer, the signal passes an **amplification and gain regulation** stage. Figure 22 simplifies this. The UNITY*plus* receiver contains three successive amplification stages (14 dB amplification each), each of which is preceded by switchable attenuators (14 dB each, one of them 2 + 4 + 8 dB) that allow setting the gain in steps of 2 dB between 0 and 42 dB[10]. In total, the UNITY*plus* preamplifier and receiver provide for a gain of 96 dB (54 + 3x14 dB) and a variable attenuation of up to 60 dB[11].

The amplified signal (δ + 10.5 MHz) is then split into two identical components that are fed into a pair of mixers. In these (double-balanced) **mixers**, the signals are then mixed with two 10.5 MHz I.F. components that are phase-shifted by 90 degrees against each other (they are both generated from a single, fixed 42 MHz frequency using a phase generator and frequency divider). The output from these mixers are two audio signal components that are phase-shifted against each other by 90 degrees, corresponding to the **real and imaginary components** of the audio signal.

After passing a pair of fixed filters, the two signals are then amplified to 10 $V_{p\text{-}p}$ maximum[12] and finally fed into a pair of **programmable audio filters** (8-pole quasi-elliptical filters) that remove noise outside the spectral window (which otherwise would be folded into the observed spectral window).

These two audio signals are finally fed into the **ADC** (not shown in Figure 22), digitized and added to the current FID through the sum-to-memory (STM) board (see also Chapter 7, “Digital Components,” on page 65). Because the transmitter frequency is located at the center of the observed spectral window, we need to detect both positive and negative frequencies. Sampling real and the imaginary signal components allows us to determine the sense of rotation of every signal component, and hence distinguish between positive and negative signal components. This is called *quadrature detection.*[13]

For liquids experiments (standard spectral windows of up to 100 kHz), UNITY and earlier systems used a similar scheme in their receivers, but mixed the 10.5 MHz signal (after the L.O. mixer) first with 10.0 MHz, resulting in a signal at 500 kHz. Mixing with a 500 kHz reference frequency then generated the audio signal. Reducing the number

---

[9] Wideband amplifiers—and rf devices in general—are much more demanding in their construction, and the receiver is certainly one of the most critical parts in the spectrometer.

[10] It turns out that the “gain” in reality is controlling attenuators! We still call it gain, as the parameter runs “backwards” (to make it look like a gain parameter): 0 dB gain means maximum (18 + 14 + 14 + 8 + 4 + 2 dB) attenuation, 60 dB gain means no attenuation.

[11] The overall gain of the receiver chain can be varied between 36 and 96 dB.

[12] Any dc offset in the audio signals can be corrected at these amplifiers, and the gain of one of the amplifiers can be adjusted, allowing to accurately balance the two channels. Also, the relative phase shift of the two channels can be fine-adjusted down to fractions of a degree at the 42/10.5 MHz phase generator.

[13] Errors in the relative phase of the two components, as well as any channel imbalance can lead to imperfect quadrature detection, resulting in quadrature (“mirror”) images of all signals (partial folding of the signals around the center of the spectrum). If one of the two channels has zero amplitude, no quadrature detection is obtained, and the Fourier transform produces all signals both as positive and negative frequencies.

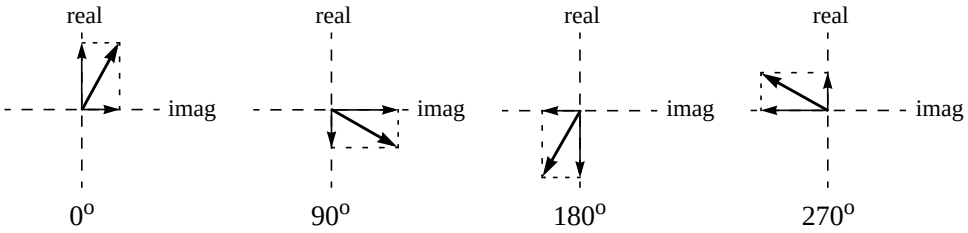
of mixing stages in the UNITY*plus* has improved performance, and the better distribution of amplification and gain attenuation at the receiver input has increased the dynamic range by avoiding overload with strong samples at intermediate levels.[14]

Quadrature Receiver Phase Shifts

Although dc offsets and the channel balance can be adjusted, there will always be slight differences between the two signal components. This can be observed in single-transient (or constant phase) spectra where often quadrature images and a center glitch can be observed[15]. Cycling the receiver through all four quadrature phases eliminates such artifacts, because both signal components have then passed through the two receiver and ADC channels (90-degree phase shifting), and dc offsets are cancelled out through phase alternation (180-degree phase shifting).

In most pulse sequences, the phase of the observed FID, as well as the phase of the receiver is therefore altered through phase cycles such as 0, 90, 180, 270, or 0, 180, 90, 270, or the like—the first example providing for fast quadrature image suppression, the second example offering faster dc offset (center glitch) cancellation[16].

To better understand how receiver phase shifting works, let's look at a particular signal vector and its real and imaginary signal components at a particular point in time within the FID, during four successive transients with the phases 0, 90, 180 and 270 degrees.



If we added these four signals as shown, we would obtain exactly zero!

Proper signal addition (averaging) is easy to obtain. We simply need to properly route the two signal components before we add them to the real and imaginary parts of the stored FID. As shown in Table 12, to change the receiver phase, we should tell the STM board how to combine the stored FID with the incoming data: 90-degree receiver phase shifting is simple math and signal routing.

This is done implicitly with `SETICM` (set input card mode) instruction in the Acode, which tells the acquisition CPU to read the `oph` register from the `LC` structure and to transfer the observe phase information to the appropriate status register on the STM board. Such an instruction is placed ahead of the *first* acquisition instruction (be it implicit or explicit) or ahead of the first hardloop that contains an `acquire` statement.

---

[14] Such overload could drive amplifiers in the receiver into saturation. At these levers they become non-linear, typically leading to intermodulation distortions in the spectrum (non-linear devices act as mixers), and typically adding mixing products of dominant NMR signals to the spectrum (ghost peaks at the distance of strong signals). This becomes most evident if more than one strong signal is present in the spectrum.

[15] Under normal circumstances, these artifacts should be below 0.5% of the main signals.

[16] Many sequences only do a partial receiver phase cycle (quad image suppression through 0° and 90° phase shifts), mainly in experiments where a residual solvent (water) signal would cover eventual center glitches (from dc offsets) anyway.

Table 12. Dealing with real and imaginary signal components

Signal phase	*Signal component*	*Operation*
0^o	0^o	add to real part
	90^o	add to imaginary part
90^o	0^o	subtract from imaginary part
	90^o	add to real part
180^o	0^o	subtract from real part
	90^o	subtract from imaginary part
270^o	0^o	add to imaginary part
	90^o	subtract from real part

Note that because the `SETICM` instruction is placed implicitly and automatically with the first `acquire` statement only, it is *impossible* to change the "receiver phase" after the first `acquire` (`oph` can, of course, be changed, but this would have no effect on the way the signals are co-added. In other words, all data points within an FID are acquired with the same "receiver phase." Even in multi-FID experiments (see Section 18.3, "Multi-FID Sequences," on page 209), all FIDs are acquired with the *same receiver phase*.

Small Angle Receiver Phase Shifting

As the receiver phase can be shifted in increments of 90 degrees through mathematical operations, one could certainly imagine a more complex STM board that shifts the incoming data by *any* phase angle, using the following formula set:

$$R_i = r_i \cos\Theta + i_i \sin\Theta$$
$$I_i = i_i \cos\Theta - r_i \sin\Theta$$

Unfortunately, the STM board does not provide such a functionality, hence proper small-angle receiver phase shifting is not possible. There are, however, a few possible workarounds for this problem:

- If *all* transients should be shifted by the *same* small angle phase (apart from quadrature phase shifts), the above transformation can be performed after the acquisition on the final FID file on the disk, either through an external program[17] or from within VNMR by going through an ASCII file (`writefid` command), using a macro to reforming these data and creating a modified ASCII file, which can the be used to re-build a phase-shifted binary FID using the `makefid` command.
- On systems with older (type a or b) rf generation on the observe channel (some VXR and earlier systems only), the `phaseshift` statement not only shifts the transmitter phase, but it changes the phase of the entire reference frame by shifting

[17] A C program with this functionality is supplied as part of the user library.

the phase of the local oscillator using the so-called “phase-pulse technique”[18] (see the section “Pulse Sequence Statements: Spectrometer Control” in the manual *VNMR User Programming)*. In fact, there are a few systems that can do small-angle receiver phase shifting! This technique may not work with newer rf generation types, because not all decades in the PTS frequency synthesizer switch frequencies in a phase-coherent way. In fact, the `phaseshift` statement is programmed to only work through the offset synthesizer used in older systems.

- Instead of shifting the receiver phase by a small angle, it is of course possible (and almost equivalent) to *shift the phase of all pulses* (at least those on the observe channel) by the same amount (using the `xmtrphase` statement), but *in the other direction*. Note that it is still advisable to do some quadrature receiver phase shifting to compensate for an eventual channel imbalance, quadrature phase error, or dc offset. It is a bad idea to keep the receiver phase constant for all scans while just shifting the phase of all pulses instead.

It turns out that from all pulse sequences and experiments, there are only a few rare examples that require small-angle receiver phase shifting at all, and for those, the solutions presented above are totally sufficient.

18.5 Housekeeping Delays

Before jumping back to the beginning of the pulse sequence, every pulse sequence performs an instruction 97 (`HouseKEEPing`), which introduces an extra delay for a variety of clean-up tasks that need to be performed before the next scan can be started:

- After every transient, the `ct` counter is incremented and compared with `nt` (to check whether the experiment is finished). Further, a check is made whether an `sa` (stop acquisition) signal has been received, and the number count from the STM board is checked to ensure the proper number of data points was acquired. The duration of this delay has changed over the different instrument and software generations: it currently is around 14 milliseconds. The only way to avoid this delay is to program a multi-FID pulse sequence (see Section 18.3, “Multi-FID Sequences,” on page 209).
- After the first FID that acquires real data, the first 256 points of the FID are checked for ADC overflow, in cases where a fixed gain was used. For this task, the housekeeping delay is lengthened by *X* milliseconds. Autogain (`gain='n'`) totally disrupts the timing after the first transient.
- Additional small delays occur at the beginning of the first “real” transient (i.e., after the steady-state transients) as well as with the start-up of every increment. The start-up delay is used to set up all the hardware (see also Section 8.2, “Looking at Acode,” on page 69).
- If a diagnostics terminal is connected to the acquisition CPU and the bootup selector switch is set to a non-zero position (even if just the switch setting is the case), long housekeeping delays (around 0.2 seconds or more) occur, because the information for the diagnostics terminal needs to be prepared and sent. This was discussed in Section 13.2, “Diagnostics and Error Output,” on page 146.

[18] This method temporarily (for a few microseconds) changes the offset on the (observe) channel, which causes the reference frame to change its orientation.

Chapter 19. Multidimensional Experiments

It cannot be the purpose of this manual to explain the mechanistics of *n*D NMR experiments; only programming issues will be discussed here. Also, apart from the names of the parameters that are used, all indirect dimensions (t_1 up to t_{n-1} in *n*D experiments, where *n* currently is up to 4) are treated the same way; therefore, only the 2D case will really be discussed here. The following tables should allow transferring the information in this chapter to higher dimensions in *n*D experiments (all the parameters in the 2D column will be discussed in detail below):

Table 13. VNMR acquisition parameters used for *n*D experiments

Parameter description	*2D*	*3D*	*4D*
Number of `tn` increments	`ni`	`ni2`	`ni3`
Spectral width in `fn`	`sw1`	`sw2`	`sw3`
`fn` coherence selection mode parameter *(optional)*	`phase`	`phase2`	`phase3`
Flag for `fn` axial peak displacement *(proposed/optional)*	`fad`	`fad2`	`fad3`
Flag for inverting folded peaks in `fn` *(proposed/optional)*	`f1180`	`f2180`	`f3180`

Table 14. Variables used in *n*D pulse sequences

Variable description	*2D*	*3D*	*4D*
Evolution delay (`double`)	`d2`	`d3`	`d4`
Spectral width in `fn` (`double`)	`sw1`	`sw2`	`sw3`
Real-time index for evolution in `tn` (`codeint`)	`id2`	`id3`	`id4`
`fn` coherence selection mode (`integer`)	`phase1`	`phase2`	`phase3`
Evolution time increment in `tn` (`double`)	`inc2D`	`inc3D`	`inc4D`

19.1 Indirect Time Domain Incrementation

One of the prominent features of VNMR *n*D pulse sequences is the absence of any explicit coding for the looping and the evolution delay incrementation. Similarly, any arrayed experiment does not complicate the pulse sequence: the `pulsesequence()` function is simply called once per array element. In the case of *n*D experiments, we don't even define an array. The presence of the `ni` and `sw1` parameters alone causes the pulse sequence software to set up an implicit array of `ni` elements on the parameter `d2`. That array is *not* shown in VNMR, but the `arraydim` parameter does reflect the extra dimension in that it multiplies the number of traces from explicit arrays with `ni`, `ni2` and `ni3`.

Note that the implicit array is starting *with the current value of* `d2` (`d3`, `d4`)[1], the array increment is `1/sw1` (`1/sw2`, `1/sw3`):

$d2_{implicit}$=d2, d2+1/sw1, ... d2+(ni-1)/sw1

$d3_{implicit}$=d3, d3+1/sw2, ... d3+(ni2-1)/sw2

$d4_{implicit}$=d4, d4+1/sw3, ... d4+(ni3-1)/sw3

For a basic *n*D experiment, all we need to do in terms of defining the sequence of pulse sequence events is, to include the evolution delay (`d2` for 2D, `d2` and `d3` for 3D, etc.). Some sequences will use a refocusing pulse in the middle of an evolution time, which can be easily achieved with a construct like

```
delay(d2/2,0 - rof1);
rgpulse(2.0*pw90,v1,rof1,0.0);
delay(d2/2.0);
```

Other sequences might use what is sometimes called a “fixed evolution time”, with a pulse that moves within that fixed delay. Also this can be realized easily:

```
pulse(pw90,v1,rof1,0.0);
delay(d2-rof1);
pulse(2.0*pw90,v2,rof1,0.0);
delay(tau-d2-rof1);
pulse(pw90,v3,rof1,0.0);
```

In order to obtain flat baselines in phase-sensitive experiments, it is furthermore important to not only compensate for any delays around the adjacent pulses but also for the precession during these pulses. The correction term is `-2.0*pw90/3.14159` *per* 90 degrees pulse of length `pw90` adjacent to the evolution time. Hence, for pulse sequences like COSY or NOESY the evolution period should be coded as follows:

```
pulse(pw90,v1,rof1,0.0);
if (d2 - 4.0*pw90/3.14159 - rof1 > 0.0)
   delay(d2-4.0*pw90/3.14159-rof1);
pulse(pw90,v2,rof1,0.0);
```

The `if` statement serves to suppress the error message from the fact that for the first increment the calculated delay is negative. This construct actually implies that the spacing between the first two increments is not the same as between all the other increments: all traces except for the first one (where theoretically the two pulses would have to overlap) are measured correctly, and this means that there will be a (minor) error in the first data point in the indirect dimension. Fortunately, this error can easily be corrected either by applying a dc offset correction after the Fourier transformation, or by reconstructing the first data point using linear prediction.

No correction for spin precession is required for TOCSY-type spin locking, and for ROESY spinlocks it seems easier to use an empirical correction term that is adjusted experimentally (such that no first order phase correction is required in the indirect dimension).

---

[1] This is a useful feature because it allows (re-) acquiring *any* part of an *n*D experiment with little effort, but it also is dangerous in that `d2` (`d3`, `d4`) can inadvertently be set to some (possibly large) non-zero value, which can have serious consequences for the result of the experiment (like causing strong first-order phase shifts or the observation of noise only). Therefore, as of VNMR 5.1, it is a good idea to have the `go` macro issue a warning if *n*D experiments are started with non-zero evolution delays.

19.2 *n*D Quadrature Detection

This is not the place for an exhaustive discussion of methods for achieving quadrature detection in *n*D experiments; however, the most important methods will be presented here as a guideline for the implementation of new pulse sequences.

Absolute Value *n*D Experiments

F1 quadrature in 2D experiments is achieved by either co-adding two data sets with the "phase-relevant" pulses prior to the evolution period shifted by 90 degrees, or by independently incrementing (or decrementing[2]) the phase of these pulses (mostly just one pulse) in steps of 90 degrees. The first method only requires half the number of scans and is completely sufficient. This is a major cancellation step (the N+P-type spectrum is added or subtracted from the N-P type spectrum, resulting in either an N- or a P-type 2D spectrum. In other words, half the 2D spectrum—the anti-diagonal and the associated crosspeaks in homonuclear correlation spectra—is cancelled in this step) and should be one of the faster steps in the overall phase cycling. Certainly it is advisable to perform this step *before* CYCLOPS.[3]

One peculiar aspect of this type of absolute-value *n*D experiment is that when we do the f_1 coherence selection, we seem not to accumulate signal-to-noise ratio (as at that point we are in fact cancelling half the signal). This is most obvious when the phase-cycling (or the cancellation efficiency) is checked in a 1D array using

```
ni=1 nt=1,2,4,8,16,32
```

where we expect a 2signal-to-noise improvement with every step. For example, if this is a simple double-quantum filtered COSY experiment where we first do a four-step double-quantum coherence selection (`nt=4`), followed by a two-step f_1 quadrature selection phase cycle (`nt=8`), followed by a four-step CYCLOPS phase cycle (`nt=32`), we would observe relative signal-to-noise ratios of 1, $\sqrt{2}$, 2, 2, $\sqrt{2}$*2, 4 in the 1D trace, instead of 1, $\sqrt{2}$, 2, $\sqrt{2}$*2, 4, $\sqrt{2}$*4 (as for standard 1D experiments). This at the same time is an easy test that indicates the minimum number of transients per increment for achieving quadrature detection in f_1[4].

Phase-Sensitive *n*D Experiments: States/Haberkorn/Ruben

The above method does not allow phasing the *n*D spectrum, because the real and imaginary parts of the spectrum are not separated in the indirect frequency domain; hence, the display in absolute-value mode. Various methods have been proposed and used to separate the real and imaginary (i.e., absorption and dispersion) parts of a *n*D spectrum. The most "natural" method on Varian instruments seems to be the technique proposed by States et al.[5], also called "hypercomplex" mode, which involves acquiring

[2] The sign of the phase incrementation determines whether P- or N-type spectra are obtained. Both can be processed by VNMR such as to give a "normal" presentation (using the `'ptype'` argument to the `ft2d` command to obtain "normal" orientation for P-type spectra.

[3] As the short-term fluctuations in an instrument (or its environment) are usually smaller than the long-term variations, it is advisable to perform the major cancellation steps (line multiple quantum filtering, f_1 quadrature) before performing steps like CYCLOPS for cleaning up minor artifacts.

[4] Note that for multiple-quantum-filtered experiments, the evolution delay (`d2` for 2D) must be set to a non-zero value in order to see a double-quantum-filtered signal in the 1D trace; do *not* forget to reset that delay to 0 before starting the real experiment!

[5] D.J. States, R.A. Haberkorn & D.J. Ruben, *J. Magn. Reson.* **48**, 286 (1982).

two separate data sets where in the second data set the phase of the (phase-relevant) pulses prior to the evolution time is incremented by 90 degrees compared to the first set. In VNMR we define an array using a parameter `phase`, which is set to the values 1 and 2 (`phase=1,2`). This parameter is available within pulse sequences in the integer variable `phase1` (note the difference in the names!)[6]. In the pulse sequence, we then can use a simple construct such as

```
if (phase1 == 2) incr(v1);
```

where `v1` is the phase of the first pulse[7]. This method of achieving f_1 quadrature gives the maximum in processing flexibility. It falls in line with the simultaneous sampling in f_2, and, compared to TPPI, it usually gives better baseline flatness. There is also a disadvantage in that any axial peaks (artifacts that often cannot be avoided, in particular with biomolecular NMR spectra) show up in the center of the spectrum, whereas with TPPI (see below) they are moved to the edge of the spectrum. This can be fixed, however, by combining the hypercomplex method with FAD (`fn` axial peak displacement, described below).

Note that the `phase` array is performed *before* incrementing `d2` (i.e., the *implicit* array is `array='d2,phase'`), which is in line with the requirement that scans or data that are to be subtracted from each other for cancellation should be measured as close to each other in time as possible, to make the experiment less susceptible to environmental variations and give better cancellation. The same holds true for 3D and 4D experiments, where the *implicit* arrays usually are

`array='d2,d3,phase,phase2'` (for 3D), or
`array='d2,d3,d4,phase,phase2,phase3'` (for 4D).

We could argue that this method requires twice (3D) and four times (4D) as long to make the first complete plane available for viewing, but this method will definitely give better cancellation than

`array='d3,phase2,d2,phase'` (for 3D), or
`array='d4,phase3,d3,phase2,d2,phase'` (for 4D).

But of course it is entirely possible to set up such arrays explicitly from within VNMR. For example, for `3D` experiments:

```
array('d2',ni,0,1/sw1)
array('d3',ni2,0,1/sw2)
ni=0 ni2=0
array='d3,phase2,d2,phase'
```

The real problem with this method (apart from giving bad cancellation efficiency) is, that such data *cannot* be processed using VNMR (unless we write extra software to rearrange the FIDs).

---

[6] In earlier VNMR releases this parameter had to be fetched from the parameter table using constructs such as `int phase1 = (int)(getval("phase") + 0.5);`

[7] Note that early releases of VNMR also incremented the receiver phase. This results in data sets that require using a different selection of coefficient arguments with the `wft2d` command, like `wft2d(1,0,0,0,0,0,0,-1)` (i.e., typically the last two arguments need to be exchanged compared with the current "standard"). This early mode also had the disadvantage that the second data set was 90 degrees out of phase in f_2 compared to the first set.

Axial Peak Displacement (FAD)

It turns out that there is a trick to move the axial artifacts in hypercomplex *n*D spectra to the edge of the spectrum by inverting the phase of the (phase-relevant) pulses prior to the evolution time and the receiver phase with every even time increment, both for `phase=1` and `phase=2`. In the vast majority of the cases this method will be used by default for hypercomplex experiments, but sometimes is may be desirable to have this extra phase inversion under flag control. For these cases we would propose the flag names `fad1` (or `fad`), `fad2`, and `fad3`. If we disregard the flag control, the pulse sequence construct for a hypercomplex experiments will be a bit more complex than the one shown in the previous section:

```
if ((phase1 == 1) || (phase1 == 2))
{
   dbl(id2,v13);
   add(v1,v13,v1);
   add(oph,v13,oph);
   if (phase1 == 2) incr(v1);
}
```

where `v1` is the phase of the (phase-relevant) pulse(s) prior to the evolution time, and `id2` is a real-time variable that contains the number of evolution time increments (`0`, `1`, `2`, ... , `ni-1`)[8].

Phase-Sensitive *n*D Experiments: TPPI

On instruments with sequential sampling, quadrature detection is achieved by the receiver phase with every data point. An equivalent method can be applied to *n*D spectroscopy by continuously incrementing the phase of the (phase-relevant) pulses prior to the evolution time by 90 degrees with every increment. This is also called *TPPI method*, or *time proportional phase incrementation*.[9] This separated the absorption and dispersion parts of the spectrum in a single data set, but it requires acquiring twice the spectral window and twice the number of increments in the indirect dimension to achieve the same digital and spectral resolution. The transformed spectrum then consists of two parts: one with the absorption lines and one with the dispersive contribution, separated by the axial peak (artifacts).[10] Because TPPI leads to a single data set, the processing seems somewhat simpler, but it apparently is more difficult to achieve good baseline flatness.

---

[8] In earlier VNMR releases, the “increment counter” `id2` did not exist and constructs like the following were used instead:

```
int t1_counter =(int)(d2 * getval(“sw1”) + 0.1);
(...)
if ((phase1 == 1)||(phase1 == 2))
{
   initval((double)(2*(t1_counter%2)), v13);
   add(v1,v13,v1);
   add(oph,v13,oph);
   if (phase1 == 2) incr(v1);
}
```

[9] D. Marion & K. Wüthrich, Biochem. *Biophys. Res. Commun.* **113**, 967 (1983).

[10] The VNMR support for processing TPPI spectra is somewhat limited, in that it is not possible to discard the dispersion part of the spectrum, which means that the entire data matrix must be carried along. This can be a problem, especially with 3D processing where the transformed data matrix may require as much as four times the disk space compared to the hypercomplex method in which the imaginary parts of the spectrum can be discarded.

The VNMR convention is that TPPI is done with `phase` (`phase2`, `phase3`) set to 3, rather than 1 and 2. In the pulse sequence, we can use a simple construct like

```
if (phase1 == 3) add(v1,id2,v1);
```

where `v1` is the phase of the (phase-relevant) pulse(s) prior to the evolution time, and `id2` is a real-time variable that contains the number of evolution time increments (`0`, `1`, `2`, ... , `ni-1`)[11].

The main advantage of the TPPI method is that only four coefficients are required for the `ft2d` command (which accepts up to *32* coefficients). If we want to measure a genuine array of phase-sensitive 2D spectra (e.g.: an array of mixing times with NOESY or TOCSY) using the hypercomplex method, we would end up with a double array. This would lead to a long and complex set of coefficients for the `ft2d` command, and the number of array elements for the "genuine" array (e.g., `mix`) would be limited to 4. With TPPI, up to 8 array elements (e.g., 8 different mixing times) can be measured, and in addition to that there is a macro `wft2dac` that makes it easy to select and process individual data sets from the array (and this macro does not cope with arrayed hypercomplex data).

Phase-Sensitive *n*D Experiments: Arrayed TPPI

The TPPI implementation discussed above has the disadvantage of leading to frequency doubling in f_1; there is an alternative that doesn't have that disadvantage: *arrayed TPPI*. For this mode, the parameter convention `phase=1,4` has been used. The method involves acquiring two data sets, with the (phase-relevant) pulses, prior to the evolution time, shifted by 90 degrees in the second data set, the same as in the hypercomplex method, but in addition to that, for the second data set the evolution time is increased by half an increment:

```
if (phase1 == 4)
{
   incr(v1);
   d2 += inc2D/2.0;
}
```

As already mentioned, this method does not lead to frequency doubling in f_1, but it requires processing the data set with real (instead of complex) FT in f_1 (`proc1='rft'`). This again has some disadvantages in that some processing options, like linear prediction or `lsfrq`, are not implemented for real FT; therefore, this method can't really be recommended. In addition to that, most current sequences add FAD for the case of `phase=1,2` (hypercomplex method, see above), which conflicts with the definition of `phase=1,4` for arrayed TPPI (where FAD would unnecessarily be added for `phase=1`, but not for the traces with `phase=4`). If arrayed TPPI still is to be used, it would probably be better to change the convention to `phase=4,5` for this method, or

[11] In earlier VNMR releases, the "increment counter" `id2` did not exist and had to be created using an standard real-time variable in constructs like

```
int t1_counter = (int)(d2 * getval("sw1") + 0.1);
(...)
if (phase1 == 3)
{
   initval((double) t1_counter),v13);
   add(v1,v13,v1);
}
```

alternatively, to have FAD under flag control (the former would be preferable, as the flag solution would still allow for a parameter mis-setting).

Folding in Indirect Dimensions

In hypercomplex phase-sensitive *n*D spectra, folded peaks (in an indirect domain, where no audio filter is involved) are in phase with the "normal" signals, provided the evolution time is corrected for the precession of the spins during adjacent pulses, as shown in Section 19.1, "Indirect Time Domain Incrementation," on page 215ff. In 3D and 4D experiments, the number on increments that can possibly be performed in any indirect dimension is very limited, and yet for many biomolecular 3D and 4D experiments, rather large homonuclear and heteronuclear shift ranges must be covered, which imposes severe restrictions on the achievable digital resolution.

In this situation, we would welcome using folding in one dimension, because this allows doubling the digital resolution with the same number of increments—or acquiring half as many increments only in one dimension (which also cuts the overall experiment time in half). Of course, this only works if there is no significant overlap between folded and "real" peaks, and if there is an easy and safe way of distinguishing between folded and "real" signals.

The method of choice to distinguish between the two groups of signals is to increase all t_n evolution delays by half an t_n increment, `0.5/sw1` (or `0.5/sw2`, `0.5/sw3`). This is the equivalent to increasing `alfa` by `0.5/sw` for the observe dimension and leads to a 180 degrees first-order phase shift in the corresponding frequency domain, which inverts all signals that are folded in once in that domain (remember that there is no attenuation due to audio filters in the indirect domains!). If there was any first point distortion, this would lead to a half-sinoidal baseline distortion in that domain. Fortunately, there are no filter distortions in these domains, and with that addition to the evolution time we may actually be able to even sample the first data point correctly (assuming the corrections for `rof1` and for the precession during adjacent pulses is less than half an evolution time increment).

There are various ways in which this could be implemented. A "minimalist approach" would be to simply set `d2=0.5/sw1` in the VNMR parameter set. We don't think this is an optimal solution, because (in order to avoid operator errors) we would rather prefer to see the evolution time delays to be set to zero for all *n*D experiments, with the exception of the case where either we want to test signal levels somewhere *within* the evolution delay or where we want to *re*acquire specific traces from an *n*D experiment (the latter case would also be made more difficult with this approach). Also, once the parameter is entered, it's value doesn't have an obvious meaning to the user, and so later it is not clear what was intended with that parameter setting.

Still we would like this feature to be under parameter control, because it certainly is not something that should be applied to all phase-sensitive *n*D experiments (in particular, not in homonuclear correlation experiments on biomolecular samples where baseline flatness is a primary requirement).

Several existing *n*D sequences use flags named `f1180`, `f2180`, and `f3180` to control this feature in f_1, f_2, and f_3. In the pulse sequence this can then be used as follows:

```
double d2incr;
char f1180[MAXSTR];
getstr("f1180",f1180);
if (f1180[1] == 'y')
   d2incr = inc2D/2.0;
else
   d2incr = 0.0;
(...)
pulse(pw90,v1,rof1,0.0);
if (d2 + d2incr - 4.0*pw90/3.14159 - rof1 > 0.0)
   delay(d2+d2incr-4.0*pw90/3.14159-rof1);
pulse(pw90,v2,rof1,0.0);
(...)
```

Combined Implementations

The majority of *n*D pulse sequences have the hypercomplex method (with FAD) and (standard) TPPI built in. This can be done with the following construct:

```
if ((phase1 == 1) || (phase1 == 2))
{
   dbl(id2,v13);
   add(v1,v13,v1);
   add(oph,v13,oph);
   if (phase1 == 2) incr(v1);
}
else if (phase1 == 3) add(v1,id2,v1);
```

where once more `v1` is the phase of the (phase-relevant) pulse(s) prior to the evolution time, and `id2` is a real-time variable that contains the number of evolution time increments (`0`, `1`, `2`, ... , `ni-1`). This should be more than sufficient for most phase-sensitive *n*D experiments.

There is a number of pulse sequences around (also in `/vnmr/psglib`) that combines all of the above *and* absolute-value acquisition in a single pulse sequence. This is certainly achievable, but it turns out to be somewhat non-trivial, because it requires combining two different phase cycles in one sequence. In order to do things in an optimum way, the f_1 quadrature phase incrementation isn't simply added at the end of the phase cycle, but instead it is *inserted* into the phase cycling sequence (see also "Absolute Value nD Experiments" on page 217). Where this is done with real-time calculations, it makes the phase cycling difficult to decode and understand, because for most of the calculation steps we have to consider the absolute value and phase-sensitive cases separately, as shown in the following (streamlined) example from a NOESY pulse sequence:

```
/* CALCULATE PHASECYCLE */
sub(ct,ssctr,v12);      /* ctss */
mod2(v12,v1);           /* 01 */
hlv(v12,v3);            /* ct/2 */
dbl(v1,v1);             /* 02 */
hlv(v3,v10);            /* ct/4 */
hlv(v10,v10);           /* ct/8 */
if (phase == 0)
{
   assign(v10,v9);      /* ct/8 */
   hlv(v10,v10);        /* ct/16 */
   mod2(v9,v9);         /* [01]8 */
}
else assign(zero,v9);
hlv(v10,v2);            /* ct/32  av
                           ct/16  ph */
dbl(v2,v2);             /* [02]32 av
                           [02]16 ph */
add(v9,v1,v1);          /* (02)4 (13)4  av
                            02           ph */
mod2(v10,v10);          /* [01]16 av
                           [01]8  ph */
add(v2,v1,oph);         /* (02)4 (13)4 (20)4 (31)4 av
                           (02)8 (20)8             ph */
add(v3,oph,oph);        /* 0213203113203102 2031021331021320 av
                           0213203102132031 2031021320310213 ph*/
add(v10,oph,oph);       /* 0213203113203102 3102132002132031 av
                           0213203113203102 2031021331021320 ph*/
add(v10,v2,v2);         /* [0123]16 av
                           [0123]8  ph */
add(v10,v1,v1);         /* (02)4 (13)4 (13)4 (20)4  av
                           (02)4 (13)4              ph */
add(v10,v3,v3);         /* [01230123 12301230]2  av
                           [01231230]2           ph */
if ((phase == 1) || (phase == 2))   /* hypercomplex + FAD */
{
  dbl(id2,v11);
  add(v1,v11,v1);
  add(oph,v11,oph);
  if (phase == 2) incr(v1);
}
else if (phase==3) add(v1,id2,v1);    /* TPPI */
```

Needless to say, in the original sequence the phase cycle is explained verbally (at least!) as follows: “The first 90-degree pulse is cycled first to suppress axial peaks. This requires a two-step phase cycle consisting of (`v1`, `0 2`). The third 90-degree pulse is cycled next using a four-step phase cycle (`v3`) designed to select both longitudinal magnetization, J-ordered states, and zero-quantum coherence (ZQC) during the mixing period. If the experiment is to collect data requiring an absolute-value display, the first pulse is next incremented by 1 to achieve f_1 quadrature. If the data are to be presented in a phase-sensitive manner, this step is not done. Next, the second 90-degree pulse is cycled to suppress axial peaks (`v2`). Finally, all pulse and receiver phases are incremented by 90 degrees (`v10`) to achieve quadrature image suppression due to receiver channel imbalance.”

With phase tables, the same phase cycling can be achieved in a somewhat easier way, but still it requires altering the “division return factor” for those tables that define phase

cycles that have lower priority than the f_1 quadrature phase incrementation and are "slowed down" for the case of an absolute-value experiment.

This is the phase table definition for the same pulse sequence:

```
t1 =   0 2             /* 1st pulse */
t2 = { 0 2 }16         /* 2nd pulse (phase=0: divn_return=32) */
t3 = { 0 1 2 3 }2      /* 3rd pulse */
    /* calculate oph=t1+t2+t3 in sequence */
t4 = { 0 1 }8          /* CYCLOPS (phase=0: divn_return=16) */
t5 = { 0 1 }8          /* f1 quadrature for phase=0 */
```

In this table, definition the phase cycling elements have been separated in order to achieve simple phase tables. The division return factor for the tables `t2` and `t4` is altered within the pulse sequence, before the phases are combined:

```
sub(ct,ssctr,v10);
if (phase1 == 0)
{
  setdivnfactor(t2,32);   /* modify tables for phase=0 */
  setdivnfactor(t4,16);
  getelem(t5,v10,v5);
}
else assign(zero,v5);
getelem(t1,v10,v1);       /* extract phases from tables */
getelem(t2,v10,v2);
getelem(t3,v10,v3);
add(v1,v5,v1);            /* f1 quadrature (phase=0) */
add(v1,v2,oph);           /* calculate oph */
add(v3,oph,oph);
getelem(t4,v10,v4);       /* add CYCLOPS */
add(oph,v4,oph);
add(v1,v4,v1);
add(v2,v4,v2);
add(v3,v4,v3);
if ((phase == 1) || (phase == 2))  /* hypercomplex + FAD */
{
  dbl(id2,v11);
  add(v1,v11,v1);
  add(oph,v11,oph);
  if (phase == 2) incr(v1);
}
else if (phase==3) add(v1,id2,v1);     /* TPPI */
```

But still: combining absolute-value and phase-sensitive experiments in the same pulse sequence *complicates the phase cycling setup*—just to incorporate a rarely used option! Therefore, it is strongly recommended to instead write two different pulse sequences, where an absolute-value option really makes sense, or at the very least, we could also think of supplying two separate phase tables with one sequence:

```
   /* phase tables for phase-sensitive NOESY */
t1 =   0 2                    /* 1st pulse */
t2 = { 0 2 }16                /* 2nd pulse */
t3 = { 0 1 2 3 }2             /* 3rd pulse */
   /* calculate oph=t1+t2+t3 in sequence */
t4 = { 0 1 }8                 /* CYCLOPS */
t5 =   0                      /* unused */
```

```
   /* phase tables for absolute value NOESY */
t1 =   0 2                    /* 1st pulse */
t2 = { 0 2 }32                /* 2nd pulse */
t3 = { 0 1 2 3 }2             /* 3rd pulse */
   /* calculate oph=t1+t2+t3 in sequence */
t4 = { 0 1 }16                /* CYCLOPS */
t5 = { 0 1 }8                 /* f1 quadrature */
```

Now a single pulse sequence could use these two files to implement both absolute value and phase-sensitive NOESY phase cycles:

```
sub(ct,ssctr,v10);
getelem(t1,v10,v1);              /* extract phases from tables */
getelem(t2,v10,v2);
getelem(t3,v10,v3);
getelem(t4,v10,v4);               /* CYCLOPS */
getelem(t5,v10,v5);               /* f1 quadrature (phase=0) */
add(v1,v5,v1);                    /* f1 quadrature (phase=0) */
add(v1,v2,oph);                   /* calculate oph */
add(v3,oph,oph);
add(oph,v4,oph);                  /* add CYCLOPS */
add(v1,v4,v1);
add(v2,v4,v2);
add(v3,v4,v3);
if ((phase == 1) || (phase == 2))  /* hypercomplex + FAD */
{
  dbl(id2,v11);
  add(v1,v11,v1);
  add(oph,v11,oph);
  if (phase == 2) incr(v1);
}
else if (phase==3) add(v1,id2,v1);    /* TPPI */
```

Of course, for the case of two different pulse sequences, the individual sequences could be further simplified in that the `if` statement that deals with phase-sensitive experiments could be left away for the absolute-value case, etc.

Coherence Selection through Gradients

Pulsed field gradients can be used in a number of ways in *n*D NMR. For just scrambling transverse magnetization (such as in some gradient-NOESY pulse sequences), up to multiple-quantum filtering and coherence selection in general. In the former case, there will still be a phase cycling section in the sequence, using the above mechanisms for f_1 coherence selection and (partial) artifact suppression. In other cases, the coherence selection will be done using gradients, which often dramatically simplifies the phase cycling (sometimes no phase cycling is used at all).

Using pulsed field gradients for coherence selection has the important advantage of not requiring subtraction (cancellation) through phase cycling; therefore, these experiments are much less susceptible to environmental variations (which otherwise often lead to bad cancellation). From a programming point-of-view, there is very little to add here that hasn't been discussed already.

Chapter 20. Solid-State NMR Experiments

Apart from the rotor synchronization feature, which consists of dedicated hardware and software for specific solid-state experiments, there is mostly a gradual difference between standard liquids and solid-state experiments. As far as software and the execution of pulse sequences are concerned, the typical spectral window in solids experiments is much larger and, because the signals themselves are often extremely wide, the pulse sequence timing becomes much more of an issue. Extra delays of even a few microseconds only can cause a severe loss of coherence, or will at the very least lead to severe phasing problems in the final spectrum.

We can not cover the entire area of solids NMR spectroscopy. In this chapter, just a few typical highlights are picked and discussed.

20.1 Cross-Polarization MAS Experiments

AP Bus Events in CP/MAS Experiments

The simplest CP/MAS experiment consists of a 90-degree pulse on protons, followed by the cross-polarization period, during which both protons and the X-nucleus are spinlocked with the same rf field (Hartmann-Hahn condition). After that, the decoupler is switched to full power to remove the dipolar line broadening during the acquisition. The following code is simplified and written for a UNITY*plus*, with AP bus control of the linear modulator:

```
status(A)
decpwrf(getval("crossp"));
delay(d1);
decrgpulse(pw,t1,rof1,0.0);
status(C);
decphase(zero);
rgpulse(getval("contact"),t2,0.0,0.0);
decpwrf(getval("dipolar"));
delay(rof2); rcvron();
```

`status(C)` turns the decoupler on. After the contact time, the decoupler stays on and is switched to the higher "dipolar" level using the `pwrf` statement, which takes 4.6 µsec on a UNITY*plus*. It is possible and acceptable to switch the linear modulator while the rf is turned on for that channel (a 4.6 µsec gap would be unacceptable anyway)—after all, that's what is happening all the time during a shaped pulse.[1]

[1] Note that for these applications it is not desirable to switch the 63 or 79 dB attenuator with the rf turned on (even though that would take less AP bus cycles). On older (UNITY) systems, we have seen transition phenomena during the switching: there is no guarantee that all the bits of these attenuators switch at exactly the same time. On such older systems, we have also observed full power coming through for a very short period (there may also be a very small gap with less power). Even though that period lasted only a few nsec, it would be enough to make a probe arc with the 1-kW amplifiers switched on. (In liquids experiments there is much less power involved, and there should be less of a problem—after all, people have been creating soft shaped pulses using these attenuators.)

If we took an oscilloscope to see what really happens during this experiment, we would find that after the pulse, during the first three AP bus words (out of four), the power stays at the same level, because the new power level is being latched (all the bits are pre-stored and then enabled at the same time, with the last AP bus word). On a UNITY*plus*, the actual switching was found to happen 350 nsec into the last AP bus period. The first 150 nsec are the pulse programmer overhead, then it seems to take another 200 nsec for the AP bus chip to read the information and set the hardware. So, the power switching occurs 3*1.15 + 0.15 + 0.2 = 3.8 µsec after the transmitter pulse (while the entire `decpwrf` statement takes 4.6 µsec). For the standard CP/MAS experiment, this means that the full dipolar decoupling will occur with a slight delay that may influence the first 1 or 2 data points, but overall, it still seems acceptable.

There are experiments such as CPCOSY or CPNOESY, however, that require that the protons are spinlocked at high (dipolar decoupling) power *immediately* after the Hartmann-Hahn polarization transfer. To do this accurately, we would have to shorten the `rgpulse` by 3.8 µsec, then turn the transmitter on during the `decpwrf` statement and then turn the transmitter off *during* the last AP bus word; however, it is impossible to switch any high-speed lines *during* AP bus events—we can only switch them before or after the `decpwrf` call. In this case, the second option is the better approximation:

```
status(A)
decpwrf(getval("crossp"));
delay(d1);
decrgpulse(pw,t1,rof1,0.0);
status(C);
decphase(zero);
rgpulse(getval("contact") - 4.6e-6,t2,0.0,0.0);
xmtron();
decpwrf(getval("dipolar"));
xmtroff();
delay(d2);
rgpulse(pw,t3,0.0,rof2);
rcvron();
```

The only error we make with this construct is that the decoupler is switching to dipolar decoupling amplitude for a maximum of 0.8 µsec at the end of the contact time, but that effect is minimal and certainly acceptable.

Using a Waveform Generator in CP/MAS Experiments

A similar problem occurs when a waveform generator is to be used in such experiments (e.g., for amplitude modulated spinlocking). A construct like

```
decrgpulse(pw,t1,rof1,0.0);
decphase(zero);
decprgon(pattern,pw,5.0);
status(C);
rgpulse(getval("contact"),t2,0.0,0.0);
```

is clearly unacceptable, because the 5.75 µsec that it takes to set up the waveform generator (on a UNITY*plus*) are much too long. Immediately after the proton 90-degree pulse, the Hartmann-Hahn spinlocking must start; otherwise, the proton magnetization would be lost completely before the spinlocking starts. In this case it may not even help to play the "trick" that was discussed in the previous section (shortening the proton 90-degree pulse by `WFG_START_DELAY`, and then turn the decoupler back on while setting up the waveform generator via AP bus), because the

duration of WFG_START_DELAY may be longer than the proton 90-degree pulse. Also, starting the spinlocking while setting up the waveform generator is undesirable, because this would lead to 5.75 µsec of unmodulated cross-polarization (plus 0.45 µsec for the waveform generator propagation delay on a UNITY*plus*). Clearly, a better method is needed here.

It would be nice if it were possible to start the waveform generator without the AP bus overhead. It turns out that this is possible! Let's first analyze what happens during the decprgon call (see also Section 16.4, "Using Waveform Generators for Programmed Modulation," on page 177):

```
300   294   188     102   WG3             AP addr 0x0c18, IB addr = 0x0000
303   297   191     101   WGCMD           AP addr 0x0c18, WFG cmd =   0x07
306   300   194     150   HighSpeedLINES  DECUP WFG2
309   303   197     150   HighSpeedLINES  DECUP WFG2
312   306   200     151   EVENT1_TWRD      1000 msec
```

First, the instruction block address is sent to the waveform generator (3 AP bus words, followed by the "waveform generator command" (2 AP bus words), then the high speed line for the waveform generator is set, after which the modulated time event starts (with the modulation actually being delayed by the waveform generator propagation delay of 0.45 µsec on a UNITY*plus*). So, decprgon prepares the waveform generator and sets the high speed line, but that high speed line is only going to be actuated *with the next time event*. We can use this to do what we want:

```
#define OBS_WFG      0x4
#define DEC_WFG     0x80
#define DEC2_WFG  0x1000
#define DEC3_WFG 0x20000
(...)
decprgon(pattern,pw, 5.0);
HSgate(DEC_WFG,FALSE);
decrgpulse(pw,t1,rof1,0.0);
HSgate(DEC_WFG,TRUE);
decphase(zero);
status(C);
rgpulse(getval("contact"),t2,0.0,0.0);
decprgoff();
```

This requires some explanation. After the decprgon call, the high-speed line for the waveform generator is set, as discussed above. *Immediately after that call* (before the next FIFO word is produced), we reset that fast bit to FALSE, such that it isn't actually turned on in the hardware. To do that we need the HSgate function with an address constant that we have extracted from Table 3 in Section 9.2, "Fast Bits," on page 88 for the UNITY*plus*.[2] During the pulse that follows, the waveform generator is ready (it has got the instruction block address and knows how to execute it), but it doesn't start yet. This only happens when we set the high-speed line *after* the decoupler pulse.

With this solution we are able to start a waveform generator without overhead. Of course, this only works for one pattern per waveform generator, and if the same should be repeated in the sequence with the same waveform generator, we would still need some time to perform the decprgon in-between. Note that on a UNITY, the statement

[2] The corresponding constants for UNITY systems can be taken from Table 2 in the same section of this manual. Note that the entire construct has been tested only on a UNITY*plus* and will probably not work on UNITY systems. Of course, only one value (DEC_WFG) is used in this pulse sequence fragment; the other values are only given here as reference and wouldn't have to be defined if they are not used.

`degprgoff` also involves AP bus traffic and hence will still introduce a delay (`WFG_STOP_DELAY`). On a UNITY*plus* the line

```
HSgate(DEC_WFG,FALSE);
```

is actually identical to

```
decprgoff();
```

and does not introduce an extra delay.

The only small timing error that we still have with the above construct is the waveform generator propagation delay (450 nsec on a UNITY*plus*), which will introduce 450 nsec of unmodulated spinlocking, but even that can be taken care of if we want to get a really perfect solution:

```
#define DEC_WFG     0x80
(...)
decprgon(pattern,pw,5.0);
HSgate(DEC_WFG,FALSE);
decrgpulse(pw-WFG2_OFFSET_DELAY,t1,rof1,0.0);
HSgate(DEC_WFG,TRUE);
decon(); delay(WFG2_OFFSET_DELAY); decoff();
decphase(zero);
status(C);
rgpulse(getval("contact"),t2,0.0,0.0);
decprgoff();
```

Here, we interrupt the 90 degrees decoupler pulse to enable the waveform generator high-speed line such that it *really* starts at the beginning of the spinlock period[3].

20.2 Sideband Suppression in MAS Experiments

TOSS (total sideband suppression) has several variants. All form a relatively simple pulse sequence element: a series of pulses with some specific spacing (which is a function of the rotor speed). In principle, there isn't much to discuss here, but we picked TOSS as an example on how to code a series of spaced pulses and also to perhaps eliminate some common misunderstandings and help you to optimize your coding under various aspects.

An early version of TOSS coding, taken from `/vnmr/psglib/xpolar.c` in VNMR 4.3, is the following:

```
rcvroff();
delay((0.1226/srate)-pw);
rgpulse(2.0*pw,v3,0.0,0.0);
rcvroff();
delay((0.0773/srate)-2.0*pw);
rgpulse(2.0*pw,v4,0.0,0.0);
rcvroff();
delay((0.2236/srate)-2.0*pw);
rgpulse(2.0*pw,v3,0.0,0.0);
rcvroff();
delay((1.0433/srate)-2.0* pw);
rgpulse(2.0*pw,v4,0.0,rof2);
delay((0.7744/srate)-pw-rof2);
```

[3] Instead of `WFG2_OFFSET_DELAY` we could also take `WFG_OFFSET_DELAY` as a constant, because all waveform generators in a system will generally behave the same way.

There is a misunderstanding involved in this coding. It is not necessary to switch off the receiver after every pulse, because after a first `rcvroff` statement the receiver is regarded to be “globally off” and is not switched back on after a pulse, but only by the next (implicit or explicit) `rcvron` call, as shown in Section 4.2, “How Do Pulses Work?,” on page 40 (although this has no negative effect, of course, other than creating an unnecessary `HighSpeedLINES` call in the Acode). This has been cleaned up in a more recent coding from the user library (slightly simplified):

```
rcvroff();
(...)
if (toss[0] == ‘y’)
{
   fprintf(stdout,”, TOSS”);
   delay((0.1226/srate)-pw);
   rgpulse(2.0*pw,v3,0.0,0.0);
   delay((0.0773/srate)-2.0*pw);
   rgpulse(2.0*pw,v4,0.0,0.0);
   delay((0.2236/srate)-2.0*pw);
   rgpulse(2.0*pw,v3,0.0,0.0);
   delay((1.0433/srate)-2.0*pw);
   rgpulse(2.0*pw,v4,0.0,0.0);
   delay((0.7744/srate)-pw);
}
```

Because this avoids the extra `rcvroff` calls, it looks easier to read, but it certainly still isn’t perfect. In both versions the transmitter phase is not preset; therefore, over the first 500 nsec of each pulse (UNITY*plus*) there will be some phase transition error (there is no pre-pulse delay). However, the following construct is definitely undesirable:

```
delay((0.1226/srate)-pw-rof1);
rgpulse(2.0*pw,v3,rof1,0.0);
delay((0.0773/srate)-2.0*pw-rof1);
rgpulse(2.0*pw,v4,rof1,0.0);
delay((0.2236/srate)-2.0*pw-rof1);
rgpulse(2.0*pw,v3,rof1,0.0);
delay((1.0433/srate)-2.0*pw-rof1);
rgpulse(2.0*pw,v4,rof1,0.0);
delay((0.7744/srate)-pw);
```

This fixes the phase error but creates a new problem. A known trouble with the TOSS pulse sequence element is that it stops working above a certain rotor speed. In the expression `(0.0773/srate)-2.0*pw`, the value starts becoming negative as soon as `2*pw>(0.0773/srate)`. The above coding increases that problem, because we now also subtract `rof1` from that delay.

The following, very compact coding avoids this problem *and* allows for phase-presetting:

```
rgpulse(2.0*pw,v3,(0.1226/srate)-pw,0.0);
rgpulse(2.0*pw,v4,(0.0773/srate)-2.0*pw,0.0);
rgpulse(2.0*pw,v3,(0.2236/srate)-2.0*pw,0.0);
rgpulse(2.0*pw,v4,(1.0433/srate)-2.0*pw,0.0);
delay((0.7744/srate)-pw);
```

Unfortunately, we have now traded in another problem: the `dps` command in VNMR hides pre- (and post-) pulse delays. Therefore, this element will be shown as four back-to-back pulses, which again is undesirable. We can, of course, explicitly pre-set the phase using the `txphase` function, as in the following coding[4]:

```
txphase(v3);
delay((0.1226/srate)-pw);
rgpulse(2.0*pw,v3,0.0,0.0);
txphase(v4);
delay((0.0773/srate)- 2.0*pw);
rgpulse(2.0*pw,v4,0.0,0.0);
txphase(v3);
delay((0.2236/srate)-2.0*pw);
rgpulse(2.0*pw,v3,0.0,0.0);
txphase(v4);
delay((1.0433/srate)-2.0*pw);
rgpulse(2.0*pw,v4,0.0,0.0);
delay((0.7744/srate)-pw);
```

This finally has the phases preset, and will be doing overall what we want. It also avoids unnecessary receiver gating (assuming the receiver is off globally), and it works with the `dps` command. Alternatively, we could use a coding that is also more efficient in terms of Acode space and execution speed (if that ever becomes an issue), because with the above coding the `rgpulse` function will create extra `SETPHAS90` and `HighSpeedLINES` calls in the Acode:

```
rcvroff();
(...)
if (toss[A] == 'y')
{
   txphase(v3);
   delay((0.1226/srate)-pw);
   xmtron(); delay(2.0*pw);xmtroff();
   txphase(v4);
   delay((0.0773/srate)-2.0*pw);
   xmtron(); delay(2.0*pw);xmtroff();
   txphase(v3);
   delay((0.2236/srate)-2.0*pw);
   xmtron(); delay(2.0*pw);xmtroff();
   txphase(v4);
   delay((1.0433/srate)-2.0*pw);
   xmtron(); delay(2.0*pw);xmtroff();
   delay((0.7744/srate)-pw);
}
```

Although at first it seems odd not to use `pulse` or `rgpulse` for something that "is a pulse", this coding isn't really too complex, and it fulfills all the needs. It is readable, it does the proper thing in hardware, and it also is most efficient in terms of Acode size and execution speed. This efficiency can be relevant in cases where multiple hardloops or a large number of pulses in general is involved, or in *n*D experiments, where excessive Acode size can lead to a loss of the Acode buffering (leading to extra delays and a disruption of the steady-state between increments).

[4] Note that `fprintf(stdout,...)` in the previous version can always be replaced by the simpler `printf(...)` function. `printf` sends the output to `stdout`.

20.3 Rotor Synchronization

Rotor synchronization experiments require dedicated hardware (consisting of a counter that counts a given number of rotor periods, and a timer/counter that counts *within* the rotor period). This accessory can be used in three different ways, as described in the following sections.

Measuring the Rotor Period Duration

Measuring rotor period duration is achieved with the `rotorperiod(vn);` statement. In this mode, the rotor synchronization hardware *returns* the number of 100 nsec clock cycles *to the acquisition CPU*. The accessory actually counts the clock cycles for every rotor period and stores the last value in a register. `rotorperiod` causes the pulse programmer to *read* that register through the AP bus (which is bidirectional) and store it in one of its own registers, from where the acquisition CPU can retrieve it into the specified `vn` (`v1` to `v14`) variable. This could be used to perform pulse sequence events at defined pulse angles (relative to the trigger position), using also the external trigger of the pulse programmer. In the example below, we want to perform a pulse at 0, 120, and 240 degrees rotor positioning on successive scans:

```
rotorperiod(v1);        /* rotor period in 100 nsec units */
modn(ct,three,v2);      /* 0 1 2 0 1 2 3 */
divn(v1,three,v3);      /* rotorperiod/3 */
dbl(v3,v3);             /* 50 nsec units */
dbl(v3, v3);            /* 25 nsec units */
mult(v3,v2,v3);         /* 0, 120, 240 degrees, in 25 nsec */
(...)
xgate(1.0);             /* wait for next trigger */
ifzero(v3);             /* don't wait for 0 degrees */
elsenz(v3);
  sub(v3,three,v3);     /* subtract 150 nsec overhead */
  sub(v3,three,v3);
  vdelay(NSEC,v3);      /* perform 25 nsec time event */
endif(v3);
rgpulse(....
```

After reading the rotor period duration using `rotorperiod` and constructing the scan-based multiplier, we calculate the desired angle resolution in 100-nsec units and then convert that to 25-nsec units. We then multiply that number by the scan-based multiplier and get the number of 25-nsec counts to perform on every scan.

We then wait for the next rotor trigger using `xgate(1.0)`. Because we don't want to (and cannot) perform a `vdelay` with zero counts, we exclude that case (0 degrees angle) using a real-time `ifzero` statement—a pulse could directly follow. For the other cases (non-zero phase angles), we subtract the pulse programmer overhead of 6 timer units (150 nsec), and then we perform a `vdelay`, which is a delay with a given time base and a timer count given as real-time variable.

However, we have to be careful, in that we should consider that

- Real-time variables are 16-bit integers with a maximum value of 32767.
- Any counter on the pulse programmer is only 12 bits, allowing for a maximum value of 4096.

At 10 KHz rotor speed, the rotor period is 100 nsec, corresponding to 4000 clock cycles of 25 nsec. In this particular case, we would be dealing with a maximum of 2/3 of a full rotor cycle, which must be less than 4096*25 + 150 nsec or 102.55 µsec. This corresponds to a rotor period of 153.825 µsec, or a rotor speed of 6500 Hz. Therefore, it may therefore be safer to do the same thing using the µsec time base:

```
rotorperiod(v1);          /* rotor period in 100 nsec units */
modn(ct,three,v2);        /* 0 1 2 0 1 2 3 */
divn(v1,three,v3);        /* rotorperiod/3 */
add(two,three,v4);        /* 5 */
dbl(v4,v4);               /* 10 */
divn(v3,v4,v3);           /* usec units */
mult(v3,v2,v3);           /* 0, 120, 240 degrees, in usec */
(...)
xgate(1.0);               /* wait for next trigger */
ifzero(v3);               /* don't wait for 0 degrees */
  delay(0.35e-6);
elsenz(v3);
  delay(0.2e-6);
  vdelay(USEC,v3);        /* perform 25 nsec time event */
endif(v3);
rgpulse(....
```

In this case, we have to face round-off errors of up to 500 nsec (below 2 degrees at typical rotor speeds). To avoid extra errors from the timer word overhead of 150 nsec, we should perform a 150 nsec delay for the 0-degree case. Because this is not possible (with the minimum delay being 200 nsec), we perform a 350 nsec delay for the 0-degree case, and a 200 nsec delay where we use a `vdelay`. This solution gives us slightly less accuracy, but definitely no problem with rotor speeds as low as a few hundred Hz.

Waiting for Triggers

The `xgate` function (which was also used in the example in the previous section) loads the (12-bit) time counter on the pulse programmer with the specified number of counts and switches the counter to the external time base (the external trigger). On each trigger, the counter counts one unit down, and the next pulse sequence event starts when the count reaches zero. Often that time count will be just 1 (1.0, as the argument must be a floating point number). If in the above example the final pulse is to be performed after a longer delay, we have the options to perform a normal delay, followed by the `xgate(1.0)` call, or we could calculate how many rotor cycles that delay would be (this is typically done based on a VNMR parameter `srate`) and then perform `xgate` with that calculated number of rotor triggers. Note, however, that the number of rotor cycles that can be counted this way is 4096 only, because the pulse programmer uses a 12-bit counter. At typical rotor speeds of 5 to 10 KHz, this limits the "counted" delay to 0.8 to 0.4 seconds.

Rotor-Synchronized Experiments

True rotor-synchronized experiments go a bit further than the above examples. In the rotor-synchronized version of NOESY (XNOESYSYNC pulse sequence) the requirement is that the two pulses adjacent to the mixing time be performed *at the same rotor angle*, after a given number of rotor periods. This is done with the `rotorsync` pulse sequence element (simplified coding):

```
(...)
initval(srate*mix,v10);
rgpulse(pw,t2,0.0,0.0);
txphase(t3);
rotorsync(v10);
rgpulse(pw,t3,0.0,rof2);
(...)
```

The rotor synchronization accessory counts through every rotor period, as mentioned before. The `rotorsync` function stops that counter through a high-speed line. Then the number of rotor periods specified in the real-time argument variable (calculated from the `srate` parameter and the duration of the delay) is loaded into the accessory, which starts counting rotor (tachometer) triggers until that number is reached (it actually counts from that number down to zero). After that it takes the number of 100 nsec clock cycles, at which it was stopped initially, and counts these *down* to zero. After that, it sends a trigger to the pulse programmer, which has been set to “external timebase” mode with a trigger count of 1. Hence, this causes the next pulse sequence element (the pulse that follows) to be executed.

This complex sequence of events guarantees that when the trigger is sent back to the pulse programmer, the rotor is at exactly the same angle as when the pulse programmer stopped the 100 nsec timer through the high-speed line. Note that in this case we do not use the timer from the pulse programmer to count the rotor cycles; therefore, there is no 12-bit (4096) count limit. The limit is 32767, the largest number that can be stored in a real-time variable.

20.4 Multipulse Experiments

Multipulse line narrowing consists of a series of pulses that is performed between each pair of acquired data points. Although it very much looks like a TOSS pulse train (see Section 20.2, “Sideband Suppression in MAS Experiments,” on page 230), the requirements are much different:

```
pulsesequence()
{
   double tau = getval("tau"),
          dtau = tau - pw - rof1 - rof2;
   ...
   initval(np/2.0,v9);
   ...
   delay(d1);
   rgpulse(pw,v4,rof1,rof2);          /*prep pulse */
   starthardloop(v9);
      delay(dtau);
      rgpulse(pw, v4,rof1,rof2);      /* x */
      delay(dtau);
      rgpulse(pw,v3,rof1,rof2);       /* -y */
      delay(tau + dtau);
```

```
      rgpulse(pw,v1,rof1,rof2);       /*  y */
      delay(dtau);
      rgpulse(pw, v2, rof1, rof2);    /* -x */
      delay(tau + dtau - 2.0e-7);
      acquire(2.0,2.0e-7);            /* acquire */
      rgpulse(pw,v2,rof1,rof2);       /* -x */
      delay(dtau);
      rgpulse(pw,v3,rof1,rof2);       /* -y */
      delay(tau + dtau);
      rgpulse(pw,v1,rof1,rof2);       /* y */
      delay(dtau);
      rgpulse(pw, v4,rof1,rof2);      /* x */
      delay(tau);
   endhardloop();
}
```

Unlike the TOSS pulse train, we cannot keep the receiver switched off globally; otherwise, we would not be able to see any signal! Also, the pulses here are extremely short (typically 1 to 2 μsec), so a 500 nsec phase or amplitude glitch at the beginning of each pulse would be clearly unacceptable. Also, the amplifier blanking/unblanking needs to be timed very carefully. For good signal-to-noise, we need to minimize the pre-and post-pulse delays. On the other hand, we also need the amplifiers fully unblanked, and we need to avoid destroying the preamplifiers with the ring-down voltages from the 1 KW pulses (which rather calls for longer pre- and post-pulse delays!). Still, there will be preamplifier saturation from the residual probe ring-down; therefore, we should place the acquisition as late in a delay as possible.

Note also that in the above coding, the maximum number of points that can be acquired is 65K, because the 16-bit hardloop count limits the number of loop cycles to 32768. If more points were to be acquired, we would have to pack multiple pulse trains (and acquisition triggers) into a single loop cycle (in which case again we must set the parameter `np` to a number that corresponds to full loop cycles; otherwise, the error message "`number of points acquired not equal to np`" is obtained.

20.5 Other Line-Narrowing Techniques

A new line narrowing technique that is now gaining popularity is the so-called "frequency-switched Lee-Goldberg" (FLSG2) method[5], which involves rf irradiation at a certain field strength with rapid frequency switching between two frequencies on either side of the proton spectrum.

The critical point about this technique is that the frequency switching must occur phase-coherently and very rapidly (actually with a 180-degree phase shift at the frequency switch). It was often thought that these requirements can be met only with special frequency synthesizers with latching (i.e., the frequency information is sent first and all the digits of the frequency change at the same time) and extremely fast (and phase coherent) frequency switching. the latter can only be obtained with overrange (i.e., with an extended range of DDS (direct digital frequency synthesis)). Such frequency synthesizers are available, but it is well known that digital frequency synthesis at higher frequencies causes increased spurious output (it is just much harder

[5] A. Bielecki, A.C. Kolbert & M.H. Levitt, *Chem. Phys. Lett.*, **155**, 341 (1989). For the actual implementation see the article by Jun Ashida & David Rice in *Magnetic Moments*, Winter, 1996, Vol. VIII,1.

to do high frequencies/offsets digitally, rather that using analog techniques). Also, any phase error during the frequency and phase transition will lead to errors that accumulate and deteriorate the efficiency of the experiment.

It turns out that with the waveform generator, we can implement these frequency offsets as linear phase ramps (shifted laminar pulses or SLP) We used 52 steps per ramp, with 4 degrees phase shift per slice. When the frequency changes, the phase ramp changes direction (and 180 degrees are added to the phase).

This turns out to be an ideal way to implement this technique. The frequency is generated using small-angle phase shifts that are performed within 100 nsec, making frequency switching extremely rapid. And the 180-degree phase switching is done with the same hardware; therefore, there is no question that both the phase and frequency switching occur in an absolutely synchronous and accurate manner, with no accumulation of phase errors or the like.

Chapter 21. (Micro)Imaging Experiments

Although, with few exceptions, imaging experiments use the same spectrometer hardware, the pulse sequences for imaging and the associated data (parameters, macros) look totally different from any pulse sequence for liquids or solids NMR. This has a variety of reasons:

- When setting up an imaging experiment, we are dealing with the physical dimensions of the object we are looking at. On the parameter side, there is a complex relationship between the physical dimensions in the image (e.g., the field of view, the slice thickness and orientation) and the experimental parameters. Ultimately it all relates to a field gradient strength, but this gradient strength again depends on the gradient amplifiers, the gradient coil, etc. This all leads to a fairly large number of parameters involved in imaging experiments. That doesn't mean that the imaging experiment is extremely complex. We do not have to interact with all those parameters! It is just that the parameter aspect is totally different from the setup of a high-resolution NMR experiment.
- Most or all imaging experiments use multi-echo/multi-FID acquisition for an entire plane, if not for a complete 3D image.
- Some or all of the indirect dimensions are not using time incrementation but a gradient strength (sometimes a frequency offset) is varied instead. Many imaging experiments use waveform generators for field gradient pulse scaling and shaping.
- To make complex imaging sequences still readable and understandable, the pulse sequence code has been broken up into a number of C functions (such as for performing a shaped pulse on top of a gradient pulse) that are defined in the pulse sequence file itself and which are called from within the `pulsesequence` function. This leads to a coding style that is totally different from anything in standard liquids and solids NMR.

Another difference from the rest of the pulse sequences is that all imaging pulse sequences are now defined consistently, both in terms of the parameters involved as well as in the C coding used in the pulse sequence file itself. Overall, the *requirements* for imaging experiments are totally different from those of any other NMR experiment, which justifies the fundamental difference in the pulse sequence coding.

It is beyond the scope of this manual to explain all the special mechanisms involved in imaging experiment; however, if you look into the dedicated imaging functions, you will see that it still uses standard pulse sequence elements. The main and basic difference is the implicit coding for the `nf`-type multi-FID acquisition.

Chapter 22. Role of Macros and Parameters

Most pulse sequences are more complex than the `s2pul` standard sequence for 1D experiments. They include more pulses, more delays, pulses on other channels, gradient pulses, etc., and all these additional pulse sequence elements will depend on parameters.

In principle, it is of course possible to “hard-code” the parameters for such new pulse sequence elements into the C program, for example:

```
pulse(13.5e-6,v1,10e-6,0.0);
```

which would avoid any extra parameter definition etc. Unfortunately, such parameters are rarely constant, because they will depend on a large number of “external” factors, such as the machine and amplifier type, the frequency, the probe type, rf and gradient amplitude settings, and even the rf routing, but also the probe tuning and the sample properties (like the susceptibility). Even though simple delays are not hardware-dependent, their optimum setting typically depends on the NMR properties of the molecules under investigation.

All this would make it necessary to change the pulse sequence and recompile it for the specific requirements of the experiment to be run, essentially for each and every acquisition. This is highly undesirable. The C coding of most pulse sequences is difficult to understand for most of the simple NMR users, and most users don’t even look at the C code of a pulse sequence. Therefore, it is strongly recommended to keep anything in a pulse sequence variable that may possibly need adjustment for a given hardware setup and a given sample.

On the other hand, many of today’s pulse sequences use a large number pulses on various rf channels, combined with several gradient pulses, possibly involving several waveform generator shapes and pattern. Keeping all this under variable control can lead to an enormous number of variables, which again can make a pulse sequence difficult and cumbersome to use.

Therefore, the target should be to *keep the number of variables at a minimum without giving up pulse sequence flexibility*. One obvious way to achieve this is to use *parameter dependencies* where possible (e.g., by deriving all pulse widths at a given channel and power level from a single pulse length, typically the length of a 90-degree pulse).

Assuming amplifier linearity and perfect pulse shapes (modern spectrometer hardware fulfils this criterion to a large extent), we could even extend this principle to pulses at different power levels, theoretically up to a point where all pulse widths (assuming the same pulse shape) on a given rf channel are derived from a single pulse width parameter at a reference power level. Similar considerations apply to gradient amplitudes and delay lengths.

22.1 Creating New Parameters in VNMR

As mentioned before, we want to be able to manipulate pulse sequence parameters from within VNMR. This way it is not necessary for the user to read and understand the C code, and complex experiments can be driven with a simple(r) user interface. In this scheme VNMR parameters become the primary medium for interacting with a pulse sequence.

VNMR comes with a basic collection of parameters: configuration parameters (`/vnmr/conpar`), global parameters (`~/vnmrsys/global`), in addition to the current and processed parameter trees (`~/vnmrsys/exp`*n*`/curpar` and `~/vnmrsys/exp`*n*`/procpar`). For driving a pulse sequence, only the configuration parameters, plus the acquisition parameter group in both the global and the current parameter trees, are relevant. Only the parameters in use in the `s2pul` pulse sequence can be assumed to be present in a general case (in many cases the user will, in fact, first do a simple 1D experiment before switching to a more complex pulse sequence). Hence, for a new, complex pulse sequence, the user has to *create* the necessary new parameters, which is typically done using the `create` command in VNMR:

```
create('parameter_name'<,'type',<'tree'>>)
```

Typical pulse sequence parameters are created in the *current* tree (which is the default); therefore, it is normally not necessary to specify the parameter tree. From a parameter *value* point-of-view it is most important to distinguish *pulses* from any other numeric parameters, because pulses are specified in microseconds. Using a real or integer, frequency (Hz) or delay (sec) parameter for pulses by mistake could lead to pulses of several seconds[1].

From a parameter handling and security point-of-view (avoiding more subtle parameter mis-settings), it is definitely important to specify the correct parameter type, which is one of `string`, `flag`, `delay`, `frequency`, `real`, `integer`, or `pulse`. This leads to parameters with the properties shown in Table 15.

Table 15. VNMR parameter types and propertie

Type	*Default properties*
`string`	character string, enumerals indicate entire string values
`flag`	character string, enumerals are possible string elements
`delay`	real, maximum 8190 sec, minimum 0, stepsize 100 or 25 nsec
`frequency`	real, maximum 1e9 Hz, minimum –1e9 Hz
`real`	real, maximum 9.99999984307e17, minimum –9.99999984307e17
`integer`	real, maximum 32767.0, minimum 0.0, stepsize 1.0
`pulse`	real, maximum 8190 µsec, minimum 0, stepsize 0.1 or 0.025 µsec

[1] As of VNMR 5.1, the built-in probe protection software should prevent probes from being burnt or amplifiers from being destroyed due to such mistakes, but it is of course better not to rely on this.

The parameter limits for the pulse and delay type parameters are “indirect” limits[2] and are taken from the parameters `parmin`, `parmax`, and `parstep` in `/vnmr/conpar` (pulse type parameters use element number 13 in these parameters, delay type parameters use element number 14). These “global” parameter limits (in particular, the pulse and delay step size) are set by the `config` program and depend on the actual spectrometer hardware (pulse programmer).

The “indirect” parameter limit is a very useful feature. There are countless parameter sets around, each of them with numerous delays and pulses. By having the parameter limits for these parameters stored in one central place (`/vnmr/conpar`), it is not necessary to adjust the parameter limits individually when moving data or software between spectrometers with different acquisition hardware.

Note that there is no “proper” integer parameter type. All numeric parameters are stored as floating point numbers. Not all pulse sequence parameters are covered by an appropriate parameter type. Power levels (rf attenuator settings, linear modulator amplitude levels), gradient amplitudes and some others do not have their own parameter types. Many of these “missing types” are covered by integers with the appropriate parameter limits. Wherever possible, these limits are defined indirectly in `/vnmr/conpar`, via the parameters `parmax`, `parmin` and `parstep` (see Table 16).

For new parameters it is advisable to limit the parameter entry to applicable values by activating the appropriate parameter limits. Wherever such parameter limits may be hardware-dependent it is advisable to use indirect parameter limits from Table 16. This leads to the following recipe for creating new parameters:

- For *frequency offsets* (and also for parameters holding coupling constants), you can use either the default `frequency` parameter properties or the parameter limits[3] for the `tof` parameter[4]:

  ```
  create('parameter_name','frequency')
  setlimit('parameter_name',7)
  ```

- For *delays*, and standard *rf pulses* the default parameter properties should be sufficient:

  ```
  create('parameter_name','delay')
  create('parameter_name','pulse')
  ```

- For *shaped or selective rf pulse lengths* and parameters that define the *duration of gradient pulses*, the default parameter maximum for pulses is too small, because selective pulses can be up to tenths of a second long. You can use either a delay parameter to define such pulse lengths, or, probably better, use a pulse parameter

[2] The definition is such that if protection bit 13 (value 8192, see the *VNMR Command and Parameter Reference* for the `setprotect` command) is set, the parameter definition does not contain the actual parameter limits, but rather indices into the three `systemglobal` parameters `parmin`, `parmax`, and `parstep`, which are numeric arrays (see also the *VNMR Command and Parameter Reference* for the `setlimit` command).

[3] Note that as of VNMR 5.1, `setlimit` with two arguments automatically activates protection bit 13 (value 8192). Under earlier versions of VNMR, this would be equivalent to

```
setlimit('parameter_name',x,x,x)
setprotect('parameter_name','on',8192)
```

[4] To be very correct, we would have to use index 8 for offsets on the decoupler channel, and indices 16 and 20 for offsets on the second and third decoupler channel, respectively. However, these limits are typically the same for all channels and not very critical. Also, with the possibility to reassign rf channels, that difference becomes irrelevant.

with modified parameter limits (*do not try* simulating millisecond parameters by multiplying pulse parameter values by 1000 or dividing delay values by 1000 in a pulse sequence, because this just complicates and obscures the issue!).

```
create('parameter_name','pulse')
setlimit('parameter_name',1e6,0,0.025)
```

- For *pulse power levels (attenuator settings)*, you should use the indirect limits used for the `tpwr` parameter. To be very correct, we should use the power parameter limits for the rf channel the new parameter refers to, but this is often not appropriate, because the `parmax[9]` (used for `dpwr`) is meant to be for a power level used in continuous decoupling and is usually set to 49, which is inappropriate for hard rf pulses.

```
create('parameter_name','integer')
setlimit('parameter_name',17)
```

- For *power levels used for continuous irradiation*, you can use the parameter limits also used with the `dpwr` parameter (where the maximum is at a safer level):

```
create('parameter_name','integer')
setlimit('parameter_name',9)
```

Table 16. Predefined, indirect parameter limits

Indirect parameter limit definition		*Parameter examples*
Index	*Typical max / min / stepsize*	
1	500 / 0 / 0.1	`sc`
2	840 / 5 / 0.1	`wc`
3	500 / 0 / 0.1	`sc2`
4	520 / 5 / 0.1	`wc2`
5	100000 / 100 / 25e-9	`sw`
6	51200 / 200 / 200	`fb`
7	100000 / -100000 / 0.1	`tof`
8	99000 / -99000 / 0.1	`dof`
9	49 / -16 / 1	`dpwr,dhp`
10	39 / 0 / 1	`dlp`
11	2e6 / 1 / 1	`dmf`
12	500 / -500 / 1	`loc`
13	8190 / 0 / 0.025	`p1,pw,pw90,rof1,rof2,alfa`
14	8190 / 0 / 25e-9	`d1,d2,pad,dod,vtwait`
15	1e6 / 0 / 0.025	`---`
16	100000 / -100000 / 0.1	`dof2`
17	63 / -16 / 1	`tpwr`
18	63 / -16 / 1	`dpwr2`
19	32767 / -32768 / 1	`(shim gradient values)`
20	100000 / -100000 / 0.1	`dof3`
21	49 / -16 / 1	`dpwr3`

- For *linear modulator amplitude settings*, you must define the limits explicitly:

```
create('parameter_name','integer')
setlimit('parameter_name',4095,0,1)
```

- The same is true for *pulsed field gradient (PFG) amplitudes*[5]:

```
create('parameter_name','integer')
setlimit('parameter_name',32767,-32767,1)
```

- For *string parameters* (typically used for shape and pattern names), no parameter limits need to be defined. It may occasionally be useful to define enumerals[6] to avoid parameter entries that don't make sense:

```
create('parameter_name','string')
setlimit('parameter_name',17)
```

- For *multi-field flags* (typically only allowing for the characters `'y'` and `'n'`) enumerals are very useful, because they simplify the value testing within the pulse sequence (e.g., if there are only two allowed character values `'y'` and `'n'`, we can simply test for one of the values, and if that test fails, the other value can be assumed). Also, flag enumerals are an easy way to define a large number of possible values in multifield flags[7].

```
create('parameter_name','flag')
setenumeral('parameter_name',2,'y','n')
```

- A *single-field flag* can alternatively be defined as string parameter with enumerals, which would be more restrictive in the parameter entry:

```
create('parameter_name','string')
setenumeral('parameter_name',2,'y','n')
```

Needless to say, any new parameter should be *filled with a sensible value*. It is best to do this right after defining the parameter, because otherwise it may be forgotten! Note in particular that defining enumerals for strings and flag parameters does *not* automatically select or fill in a "legal" value: `create` followed by `setenumeral` will leave a parameter with an empty string (`''`), which may lead to unexpected results when testing for either `'y'` or `'n'` alone in the pulse sequence!

---

[5] The proper limits for Performa I type gradients are 2047, -2047, 1; but still it is recommended to use the limits for Performa II gradients as shown in the text, because this makes the parameter set portable between systems with different PFG amplifiers.

[6] See the *VNMR Command and Parameter Reference* for the `setenumeral` command. The problem with enumerals in strings is that VNMR is not very helpful in the case of an "illegal" parameter entry. An error message is issued, but no hint as to what the allowed values are is given. You have to use `display('parameter_name')` to see the enumerals for a parameter. For pulse shapes and WFG patterns, it is therefore better *not* to define enumerals (you will get an error message from the `go` command if a non-existent pattern or pulse shape is specified).

[7] For a flag with *n* fields, allowing for the characters `'y'` and `'n'`, the number of possible values is 2^n, or even more if we account for abbreviated versions (i.e., the fact that by convention the last character of a flag parameter is propagated to any subsequent position: `'n'` stands for `'nnnnn...'`).

22.2 Using New Parameters in C

In order to be able to use the value of a new parameter that has been defined in VNMR, we have to use the special functions `getval` (for numeric parameters) and `getstr` (for flag and string variables) that read such parameter values off the VNMR parameter table and make them available in the pulse sequence environment.

Numeric Parameters

The `getval` function serves to extract numeric parameter. It returns a double, irrespective of the parameter type. Note that values from *pulse parameters* are converted and passed to the pulse sequence environment *in seconds*! Typically, the `getval` function is used to initialize a C variable:

```
double pwx;
pwx = getval("pwx");
```

The VNMR parameter name in the argument of the `getval` function and the name of the C variable don't necessarily have to be the same, but of course this is very much preferable. Because it is advisable to keep pulse sequences simple, we should always watch out for syntax simplifications. One such possibility is to combine variable declaration and initialization:

```
double pwx = getval("pwx"),
       pwx2 = getval("pwx2");
```

If a parameter value is used only once in a pulse sequence, it is not really necessary to define a C variable that holds the value. The function `getval` can also be used directly as argument to an other function:

```
rgpulse(2.0*getval("pwx"),zero,rof1,0.0);
```

And, of course, `getval` functions can also be used within mathematical expressions:

```
tau = 1.0/(2.0 * getval("j"));
```

String Parameters

String parameters are slightly more complicated. In principle it would be possible to create a function similar to `getval` for strings (i.e., a function returning a (pointer to a) string). The problem with this is that the user would have to deal with pointer variables, and that was considered to be too complex for a simple pulse sequence language (apart from that, pointers are a possible source for errors that are hard to debug in C programs). Consequently, string parameters must be filled into a string C variable:

```
char xpol[MAXSTR];
getstr("xpol",xpol);
```

`MAXSTR` is defined as 256. Note that after the (fixed-length) string variable definition, the string is *not* automatically initialized; its contents can be assumed to be random.

Do not forget to initialize variables! This mechanism is used both for strings and flag parameters. For checking individual character fields within a status-related flag variable, see also "Checking Flag Parameters" on page 160.

22.3 Adding New Parameters to the Display

In VNMR, creating and setting the new parameters isn't just enough. You want to be able to see what the values are, and eventually print out complete parameter listings. Typing `parameter_name?`, as the only way to see a parameter value, isn't a good solution, because it is very easy to overlook parameters that are not shown on the screen or listed in a printout.

Editing display (`dg`, `ap`) templates is usually by `paramvi('parameter_name')` or by `paramedit('parameter_name')`. If you have defined an environment variable `vnmreditor` in your `~/.login`[8] file, `paramedit` can be set up to use `textedit` instead of the `vi` editor, which is more convenient for editing parameters like `dg` and `ap`, which typically consist of a single, long line (in which occasional users of `vi` will have difficulties in setting the insertion point[9]). Entering such a long parameter value directly in VNMR (`dg='.....'`) is difficult, if not impossible. Apart from that, you don't want to re-enter the entire template, but you selectively want to add the new parameters.

The parameter display templates are explained in Chapter 5, "Modifying Parameter Displays in VNMR," in the manual *VNMR User Programming*, Be careful with conditional displays. Referring to non-existent parameters in a condition expression causes nasty error messages!

As of VNMR 5.1. it is possible to split a display template into substrings by setting up an array of strings—typically one substring per column title. Although such split display templates are incompatible with earlier versions of VNMR, you will find the shorter substrings much more manageable when editing (particularly with `vi`). It even becomes feasible to substitute entire substrings, for example:

```
dg[2] = '1:PULSE SEQUENCE:seqfil,d1,p1(p1):1,d2(d2),pw:1;'
```

It is advisable to at least define a `dg` and an `ap` template (if you want, you can also use the `dg` template for the `ap` and `pap` commands, although that will create 4-column output and arrays will not be shown). You certainly want to make sure that all relevant acquisition parameters (in particular those that are pulse sequence-specific) are on contained in a template. For the `ap` template to work properly with arrays, it should be set up with two columns only.

A potential problem with the `dg` command and template is that the available space is very limited, and you often have to use several display templates to cover all relevant parameters. The `dg` command is well-behaved if there are too many parameters in a column. Extra parameters are shown on the top of the next column (except for the last one, where extra parameters will simply not be shown!). Note, however, that `dg` does *not* handle properly the case where the "parameter overflow" is more than one column!

---

[8] You can also type `setenv vnmreditor textedit` in a shell window before starting VNMR with the `vn` command.

[9] The best way to position the insertion point in such a long line using `vi` is to search for a text string (`/substring`), rather than using the keyboard arrows.

22.4 Doing It All by Macro

Obviously, there is a fair amount of “extra setup work” related to creating a new pulse sequence. We certainly don’t want to repeat that every time when we want to use such a sequence! Therefore, many years back, we started using pulse sequence-specific macros that allow us to easily switch between a simple 1D setup experiment and any more complex sequence.

Of course, we could also use the strategy of storing one complete and ready-to-use parameter set for every pulse sequence, which can simply be recalled before starting the desired sequence. The disadvantage of this approach is that sample-specific parameters that have been calibrated in the 1D setup experiment (such as transmitter offsets, spectral window, referencing parameters, probe- and sample-specific pulse widths and power levels) are lost and would have to be restored manually, which of course is a source for errors and omissions. We therefore strongly recommend creating such pulse sequence-specific macros.

Apart from the tasks described above, such a macro can carry the action a bit further and also adjust additional parameters (existing ones that need adjustment), display a manual file for the pulse sequence, etc. And of course the macro should also select the pulse sequence it is written for by setting the `seqfil` parameter. The ideal pulse sequence macro should be written in such a way that an inexperienced user can do a 1D setup experiment, then simply type the name of the desired pulse sequence (i.e., execute the pulse sequence-specific macro) and start the acquisition. This user should then end up with a *reasonable* spectrum using the built-in default parameter values, as set by the macro.

Macros for 1D Pulse Sequences

Possible tasks for a 1D pulse sequence setup macro can be described as follows:

1. Select the pulse sequence by setting the `seqfil` parameter.
2. Create new, pulse sequence-specific parameters.
3. Where necessary, set protection bits and limits for the new parameters.
4. Fill reasonable values into the new parameters.
5. Set other parameters to pulse sequence-specific settings, where necessary (e.g.: set `nt` to a multiple of the pulse sequence phase cycle length).
6. Set up parameter display templates (`dg`, `ap`).
7. Preadjust processing and display parameters, where necessary.
8. Display information on the new pulse sequence (typically a manual file) with further information.
9. Display the pulse sequence using `dps` (optional).

The first five tasks listed above can be done with standard MAGICAL constructs and commands listed above:

```
seqfil='dept'
create('mult','real')
setlimit('mult',2,0,0) mult=0.5,1,1,1.5
create('tau','delay')
create('pp','pulse')
```

```
create('pplvl','integer')
setlimit('pplvl','17)
...
```

There is a problem with this construct, in that `create` results in an error if a parameter already exists. A safer construct would be:

```
seqfil='dept'
exists('mult','parameter'):$e
if not($e) then
  create('mult','real')
  setlimit('mult',2,0,0)
endif
mult=0.5,1,1,1.5
...
```

Even though that looks complex for sequences with many parameters, it certainly is readable. Where this concept falls apart is with the parameter display templates. We can't edit a parameter from within a macro that may be called in background (we don't want to re-edit the templates upon recalling a sequence anyway), and we don't want to set the entire `dg` and `ap` parameters with constructs like

```
dg = ''
dg[1] = '1:SAMPLE:date,temp,solvent,file(file<>\'exp\');'
dg[2] = ...
```

because this would make setting up such macros quite complicated!

Instead, the following concept has been adopted by most users: We generate the new parameters once (as shown above) and fill in the appropriate default values, and we generate the modified display template parameters. Then we *save the entire parameter set* in `parlib` (`/vnmr/parlib` or `~/vnmrsys/parlib`). Now, in the pulse sequence start-up macro, we simply *pick the new or modified parameters with their values* from that parameter set:

```
psgset('dept','mult','j','tau','presat','dg','ap')
```

This saves us from checking whether a parameter already exists (an existing parameter will simply be overwritten by the imported one), and it also allows setting the `dg` and `ap` parameters in a simple way (once they have been modified and stored, of course!).

The first argument to the `psgset` command (a macro) is not only the value that is going to be filled into the `seqfil` (and `pslabel`) parameter, it also is the (body of the) file name (without "`.par`" extension) of the parameter file from which the specified parameters are picked. As expected, a local parameter file takes precedence over a parameter file with the same name, but stored in `/vnmr/parlib`. The `psgset` macro can pick up to 11 parameters in one call (not counting the first argument: the maximum number of arguments is 12 with this command). To retrieve more parameters, several `psgset` calls are required.

Before recalling the parameters and their values (using the `rtv` command), `psgset` calls `prune` to remove any extra parameter in the local set that is not present in the `parlib` file. This avoids an excessive accumulation of parameters when calling different pulse sequence macros consecutively.

After having “imported” the new parameters plus the display templates, most pulse sequence macros are supposed to also set default values for existing parameters. This can be achieved in two different ways:

- The values can be stored in the parameter set in `parlib` and recalled with `psgset`, which will simply replace existing values with those from `parlib`.
- The values can be set directly, if necessary, with `if` constructs:

```
if d1=0 then d1=1 endif
```

The problem with the `parlib` approach is that the parameter values are well hidden away, and to change the defaults we have to recall the parameter set, change the values, and then overwrite the saved parameter set again. In the author’s opinion, it is much easier (and more transparent) to change a macro than to change a parameter set, and hence it is recommended to rather use the first approach: *use* `psgset` *to recall and set the new and specific parameters and the display templates, but use simple MAGICAL assignments to set existing (standard) parameters.*

There is a special case. Some macros want to set `nt` based on the number of transients for the previous set-up experiment. In the case of the `dept` pulse sequence, the macro is supposed to decrease `nt` by a factor of 32 (taking into account the sensitivity enhancement due to polarization transfer), but ensuring that `nt` still is a multiple of 4 (the basic phase cycle for that sequence). Taking the current value of `nt` may not be appropriate, because in the preceding experiment `nt` may have been set to a very large number, and the actual experiment was perhaps stopped after having reached a satisfactory signal-to-noise ratio. Taking `ct`, on the other hand, can be wrong also, because any change in an acquisition parameter will set `ct` to zero! Therefore, it is a good idea to *first* capture the value of `ct` (note the precaution for arrayed `nt`!):

```
“dept - convert standard parameter set to dept”
if ct>0 then $nt=ct else $nt=nt[1] endif
psgset('dept','mult','j','tau','presat','dg','ap')
if $nt>128 then nt=$nt/128 nt=4*nt else nt=4 endif
if d1=0 then d1=1 endif
in='n' il='y' gain='y' pw=pw90 hs='nn'
dof=0-2.5*dfrq            “move decoupler from 5 to 2.5 ppm”
dm='nny'
if waltz='y' then dmm='ccw' else dmm='ccf' endif
ai
wexp='deptp' array='mult'
man(seqfil) dps
```

Note that for the `dept` sequence, `mult` is an normally arrayed parameter, but using `psgset` and `rtv` to retrieve such an arrayed parameter does *not* automatically set the `array` parameter! Also, some parameter settings may have to be set depending on the actual hardware. Constructs like the one used for the `dmm` parameter (depending on the presence of WALTZ decoupler modulation capability) make a pulse sequence (macro) more portable between different hardware platforms.

Macros for 2D Pulse Sequences

An important additional task in 2D pulse sequence set-up macros is the creation of the 2D specific parameters `ni` and `sw1`. This is typically done by the `set2d` macro utility:

```
“relayh - set up parameters for relayh pulse sequence”
if ct>0 then $nt=ct else $nt=nt[1] endif
av set2d('relayh', 6)                “6 Hz resolution in F2“
psgset('relayh','relay','tau','dg','ap')
...
```

The `set2d` macro sets the `seqfil` parameter, then calls `par2d(seqfil)`, a macro that creates the standard 2D acquisition parameters `ni`, `sw1` and `phase`. For homonuclear 2D spectra (`dn=tn`), it sets

```
sw1=sw ni=sw1/24 rfl1=rfl rfp1=rfp
```

(12-Hz default digital resolution if f_1). For the heteronuclear case, it calls

```
psgset(seqfil,'ni','sw1')
```

or if no argument was specified, it sets `ni=256 sw1=2*sw` as “default guesses”.

Back to the `set2d` macro. After having called `par2d`, the `set2d` macro switches to a fixed gain (i.e., that is something you will not have to do in the set-up macro!), then sets up the acquisition and processing parameters `np`, `fn`, `ni`, `fn1` for the specified (or the default) digital resolution if f_1 and f_2. The default is 6 Hz in f_2, 12 Hz in f_1 (the desired digital resolutions can be specified in arguments 2 and 3).

The `set2d` macro then selects sinebell weighting functions in both dimensions for absolute-value spectra (`dmg='av'`), or a gaussian apodization for phase-sensitive 2D spectra (`dmg='ph'`). *It is therefore important that* `set2d` *is only called when the correct display mode has been selected!* It is recommended to call `set2d` prior to the first `psgset` call, and to always have it preceded by `av` or `ph`, to ensure the proper display mode selection, as in the example above.

In a last step, the `set2d` macro adjusts the display parameters for the 2D spectrum, and it disables the “automation parameters” `wbs`, `wnt`, `wexp`, and `werr`.

For *heteronuclear 2D* experiments, the set-up macro has the additional tasks of setting up the acquisition parameters for the “other” nucleus (in f_1). It would be nice if these parameters (`dof`, `sw1`, `rfl1`, `rfp1`) were taken from the equivalent parameters (`tof`, `sw`, `rfl`, `rfp`) in a second set-up experiment, typically a 1D experiment in an other experiment file. Starting with VNMR 5.1, this can also be an “internal” data set in the same experiment directory. This can be done by adding a lengthy construct like the following to the set-up macro (simplified, for VNMR 5.1):

```
jexp:$experiment
exists(curexp+'/subexp/H1','directory'):$internal
if $internal then
  $enumber=$experiment
else
  $enumber=0
endif
if ($#<1) then
  input('exp# containing 1H spectrum (0=none)? '):$enumber
else
  $enumber=$1
endif
if ($enumber<0) or ($enumber>9) then
```

```
  write('error','%s - illegal experiment number',$0)
  return(1)
endif
if ($enumber>0) then
  "get proton parameters"
  if ($enumber=$experiment)
    if not($internal) then
      write('error','%s - no 1H data found in current exp',$0)
      return(1)
    else
      svtmp('tmp') rttmp('H1')
      $savesw=sw $savetof=tof
      $saverfl=rfl $saverfp=rfp $savewp=wp $savesp=sp
      rttmp('tmp')
    endif
  else
    jexp($enumber)
    $savesw=sw $savetof=tof
    $saverfl=rfl $saverfp=rfp $savewp=wp $savesp=sp
    jexp($experiment)
  endif
  sw1=$savesw dof=$savetof
  rfl1=$saverfl rfp1=$saverfp wp1=$savewp sp1=$savesp
else
  sw1=10*dfrq dof=0
  write('line3','a full 10 ppm 1H window will be used')
endif
fn1=sw1/3 ni=fn1/4
```

The released VNMR 5.1 `hetcor` macro actually uses a much more complex construct! This is, of course, a typical example for a construct that should be put into a utility macro. The only reason why this hasn't happened yet is that within the standard VNMR pulse sequences it was always only the `hetcor` macro that used such a construct. A number of related pulse sequences (`coloc`, `flock`, phase-sensitive `hetcor`, but also indirect detection sequences like `hmqc`, etc.) could profit from such a utility.

22.5 Switching Between Similar Sequences

The above macro scheme is acceptable for the way an NMR spectrometer is used in many organic chemistry and industrial environments, where users typically perform a series of set-up experiments (like one for ^{1}H, one for ^{13}C, maybe also extra experiments with reduced spectral window), and then start a limited number of special experiments—mostly just one or two:

- ^{1}H, <^{1}H (reduced `sw`),> COSY
- ^{1}H, <^{1}H (reduced `sw`),> NOESY
- ^{1}H, <^{1}H (reduced `sw`),> ^{13}C, DEPT, HETCOR
- ^{1}H, COSY, ^{13}C, DEPT, HETCOR

Note that for homonuclear 2D spectra it is not necessary to acquire a spectrum with reduced spectral window. You *first* adjust the chemical shift referencing, then you save the 1D spectrum either in a subfile, using `cptmp`, or by moving into a different experiment (you will use the 1D spectrum for the 2D plotting). Now you can just place

the cursors around the relevant signals, type `movesw`, and *then* call the 2D pulse sequence set-up macro.

However, a typical spectroscopist in biological NMR will work in a totally different way. Often a sample stays in the magnet for a week or more, while a series of *similar or closely related* experiments are performed, such as HNCOCA, HNCA, etc. Under such circumstances, the above macro scheme has several disadvantages:

- The complex sequences used in this environment require a large number of parameters that have been carefully calibrated on that particular sample; `psgset` would simply overwrite these values with the ones from `parlib`, which is highly undesirable.
- Some of these macros may assume that they are called in a 1D experiment and may not work properly when called on a 2D or 3D data set (e.g., some of these macros cannot be called twice in sequence!).

Users have adopted two philosophies to cope with these deficiencies:

- Rather than calling a pulse sequence macro, they simply recall a parameter set from `parlib`. The referencing between different biological samples (at least those in one lab) is often almost identical, so this is not a problem. However, this will require *re-entering all calibrated parameters* for each experiment (considering the fact that many of these experiments last for several days, this seems acceptable to many users).
- Assuming all these pulse sequences use convergent parameter definitions, it may be possible to simply change the `seqfil` parameter to switch to a related sequence. The problem is, that it *cannot* be assumed that parameter sets are consistent between sequences in a general case (unless all sequences involved were developed locally), because there is no compulsory style guide for parameter naming and usage!

Clearly, this is an area which requires improvement. In fact, there are planned improvements:

- A new style of setup macro is being developed that can be used repeatedly and does not overwrite calibrated parameters.
- A detailed style guide covering parameter naming will be put forward in a future version of this manual.

Chapter 23. Putting It All Together

In this chapter, we look at approaches for programming a new pulse sequence, testing a sequence and its associated macros and files, and submitting a sequence to the Varian user library.

23.1 Starting a New Sequence

To write your own pulse sequence, you have two principal options:

1. Look for a pulse sequence (in `/vnmr/psglib`, or in `userlib/psglib`, or in any other source) and modify or rework that into your desired pulse sequence. You can do the same thing with the associated parameter sets, macro, phase table, etc.
2. Start a new sequence "from scratch."

Programming by Modifying an Existing Pulse Sequence

Definitely, the first approach is very often much easier and is preferred simply because it is less work. It does have some disadvantages, such as:

- It is more susceptible to programming errors.
- It is more difficult to debug because the program is not developed in a "top-down" approach.
- It is much more difficult to do systematic programming and development this way.
- It tends to perpetuate bad syntax and style from existing sequences and old—maybe outdated—pulse sequence programming features. The syntax tends to get worse with every transformation of a pulse sequence, unless the programmer is very disciplined.

Programming by the Top-Down Approach

We are not going to further discuss the first approach (there isn't much to say about this method anyway). On the other hand, it certainly is worthwhile giving some hints for the second approach, pulse sequence programming from the "top-down," for which (given the above) some obvious advantages exist:

- It can lead to clean, well-debugged programs.
- The resulting sequence will have a more systematic and simpler syntax.
- It probably uses the latest pulse sequence programming features and, therefore, is more up-to-date to begin with.

Although the top-down approach may be somewhat more work for the programmer, it certainly is the more valuable method from a didactic point-of-view.

The idea of the top-down approach is simple: *don't start with the details*! Start with a rough framework, then start filling in details level by level. Try keeping the syntax complete and compilable all the time, and recompile the program frequently. This way

you never have too many bugs at once, and it is much easier to find the bugs than if you start compilation when the program is several pages long.

The best way to do this is to open up two UNIX windows side by side (the development of a complete, new sequence; don't block the NMR interface by calling an editor from within VNMR). In one window you edit the sequence (using your preferred editor); in the other window you *compile the sequence periodically*, whenever it is appropriate. You can leave these windows running for as long as the programming lasts (or until you want to log out, of course). Just close the windows to icons when you want to use the VNMR user interface.

Periodically save your work—don't give Murphy's law a chance to destroy several hours of working effort!

While you program, always add comments. Don't wait to insert comments until you are finished, because then you will want to work with the new sequence and the comments will never be added. Sensible comments never make a sequence more complicated; they keep it understandable.

Don't write novels into the comments. Use short comments, but add comments to every part of a complicated pulse sequence. Short sequences on the other hand (if they are well-written) should be more or less self-explanatory and usually require very little comment.

As you write the pulse sequence, maintaining a **manual file** for the new pulse sequence is recommended. Often, the manual has been taken from the first part of the comment of the pulse sequence text; it is not necessary to duplicate that text. Either write a separate manual and don't add that text also to the sequence, or just do it in the pulse sequence for simplicity (you can still move that text out of the sequence and into a separate manual file later).

23.2 Testing a Sequence and Related Files

At some point in the programming you start using new variables (delays, pulse widths, power levels, etc.). When this happens, always add the parameter definition and initialization in the pulse sequence (such that it still compiles without error message). At the same time you can start developing the **macro** and creating the **parameter set** for the pulse sequence. Reserve an experiment in VNMR for that new sequence, and successively create (and adjust) new parameters, as you add them to the pulse sequence. Also, you can keep the `dg` and `ap` templates up-to-date at the same time. In the macro, very likely you will have to add an argument to a `psgset` command. You also want to periodically save the macro and the parameter set, even when they aren't finished yet, just to secure your work.

Also, as you start using phase tables, you should of course maintain a **table file** for the sequence.

Continue doing this until you think the sequence is complete. When the sequence then compiles without error, you should complete the macro (of course, you will check with a similar, existing macro to ensure that you don't forget anything in the new macro).

Typing `dps` in VNMR will now indicate whether the parameter set is complete, and it will also give you a first "clear view" onto the features of the new sequence: are all the pulses and delays there, in the right sequence? When `dps` works without problems, you

should double-check the dg and ap templates. and then save the parameter set in parlib.

Now either retrieve a simple 1D spectrum or use the VNMR setup command to retrieve standard 1D parameters, and then call the pulse sequence macro and check again with dps. This will tell you whether the macro is complete and functional.

You can go even further with the testing, still without involving a spectrometer. You may want to check the pulse sequence for run-time errors and problems by typing go('acqi'). This executes the pulse sequence and generates Acodes, at least for the first increment (in case of arrayed or *n*D experiments). It also serves as a test for the parameter checking built into the pulse sequence. If you then want, you could even use the apdecode tool (bin/apdecode from the user library) to have a look at the Acodes that were generated.

All this can be done on a data station. In order to be able to call go('acqi') you must call config and declare the system a spectrometer—but *don't* call setacq on a data station, of course!

The next steps must happen on the spectrometer. Try a 1D run (nt=1) first. Then you may want to *check the phase cycling* by performing an array with

```
nt=1,2,4,8,16,32,64,128
```

You may have to adjust the parameter defaults in the macro, as you continue with the testing.

Finally make sure the last versions of all related files are complete and saved:

- Pulse sequence (from psglib).
- Table file in tablib.
- Macro.
- Parameters in parlib.
- Manual file.
- Shape and pattern files, if required.

If the new sequence is functional and works to your satisfaction, don't you want to submit it to the Varian NMR user library and share the sequence with other users?

23.3 Submitting a Pulse Sequence to the User Library

To submit a new sequence to the user library, first fill in a submission form (in the user library the form becomes the README file, myseq.README in this example). You find a blank form in /vnmr/userlib/SUBMISSION. Then you collect all files related to your sequence, except for the compiled version (because this is specific to your software version, it would just take up lots of space in the user library—also, it is very easy and quick to recompile a sequence):

- If you want to send the submission by e-mail, you can directly create a uuencoded, compressed tar file:

```
cd; cd vnmrsys
tar cf - psglib/myseq.c tablib/myseq parlib/myseq.par \
        maclib/myseq manual/myseq | compress | \
        uuencode myseq.tar.Z > myseq.uu
```

Now you can send the submission form and the data to the `userlib` librarian:

```
cat myseq.README myseq.uu | \
     mail -s 'myseq submission' rolf@nmr.varian.ch
```

- If you want to send the submission by floppy, you create a compressed tar file:

```
cd; cd vnmrsys
tar cf - psglib/myseq.c tablib/myseq parlib/myseq.par \
        maclib/myseq manual/myseq | compress > myseq.tar.Z
```

Then you store both `myseq.README` and `myseq.tar.Z` on a floppy and send it to the librarian: Rolf Kyburz, Varian International AG, Chollerstrasse 38, CH-6303 Zug / Switzerland.

Chapter 24. Syntax Guidelines

It is planned to include a complete style guide for writing pulse sequences in a future version of this manual. For the time being (as long as that style guide hasn't been defined yet), we'll just include some hints for writing pulse sequences such that they are portable to other systems and readable and easily understood by other users.

24.1 General C Syntax

Being disciplined in the use if the C syntax can be a great help in keeping a pulse sequence readable. A few points in more detail:

Comments

Adding comments is definitely recommended, but add reasonable comments: enough to explain what is not obvious, but don't pack too much theory into the comments (you can add a literature reference. Too many comments can also clutter a C program! You can have a larger comment segment at the top of the sequence, but for the actual C code it is better to use one-line comments only.

Indentation

Using proper and systematic indentation helps in two ways: it definitely makes a sequence more readable, and (even more important) it helps avoiding mistakes! The author recommends using the following indentation rules (no examples shown—there are enough examples for C coding in the rest of this manual!):

- Use indentation, but don't indent too much (such as an entire tab stop: this would lead to excessive line lengths in nested if statements etc.). For short sequences (one page or less), two spaces per indentation level should be enough. For longer sequences, it may be better to indent by three or four spaces.
- Always have corresponding opening and closing braces at the same indentation level. Do *not* add the opening brace to the end of an `if` line!
- Have `if` and the corresponding `else` at the same indentation level.
- Where an `if` or an `else` branch contains only one function call or statement, braces are not really necessary, and you should leave them away, to keep the coding shorter (although with beginners in C, this is a possible source for errors if additional statements are added to any of the branches.
- If the splitting into `status` fields is a prominent feature in a pulse sequence (most traditional pulse sequences were written this way), you can use an additional indentation after `status` calls.
- Additional indentation should also be used for functions between `ifzero`, `elsenz`, and `endif` calls, as well as for statements between `loop` and `endloop`, and between `starthardloop` and `endhardloop`.
- Place comments either behind a C line (preferably at an identical tab stop) or, for longer comments, at the same indentation level as the surrounding C statements

(starting comments at the beginning of a line would destroy the effect of the indentation).

When dealing with a program that has been coded with a “strange” indentation style, the author often uses the UNIX `indent` command to fix it up (see `man indent` for more information). `indent` can be used with arguments, but it is much easier if it is configured with a local `~/.indent.pro` profile. A good set of indent settings would be the following line in `~/.indent.pro`:

```
-bap -bad -bl -ncdb -nce -d0 -di2 -eei \
    -nfc1 -i3 -nip -lp -npsl -sc -nsob
```

Using `-in` is the principal indentation. Use `-i2`, `-i3`, or `-i4`, as appropriate. The `indent` program usually does a very nice job, but of course it does not “know” about pulse sequence specifics (i.e., you will have to fix up the result by adding indentations after status, after `ifzero` and `elsenz`, after `loop`, and after `starthardloop`).

Variables

Try keeping things simple and short! Doing variable initialization within the variable declaration saves many lines of code, *and* it ensures that all variables are initialized! For numeric parameter values that are use only once, you don’t need to define a variable at all—you can use the `getval` function directly as function argument.

24.2 Outdated PSG Utilities

The use of outdated PSG utilities may not affect the functionality of a pulse sequence *on your system*, and *the way you use it*; however, it makes a sequence less portable, in that it may be incompatible with different hardware or with the way an other person uses the spectrometer[1].

Device Addresses

A number of pulse sequence functions require specifying a device (rf channel) address. The device names used in earlier versions of VNMR are `TODEV` (observe channel), `DODEV` (decoupler channel), `DO2DEV`, and `DO3DEV`. As of VNMR 5.1, things have *changed*! These device addresses still exist, but they now refer to the physical, rf channels (`TODEV` is channel 1, `DODEV` is channel 2, etc.) and *don’t* reflect the setting of the `rfchannel` parameter in VNMR (i.e., `TODEV`, `DODEV`, etc. will always point to the same physical channel).

If you ever want to use the `rfchannel` mechanism for re-assigning the *physical* channels to different *logical* channels (and if you don’t, some other user of your pulse sequence may!), you should now use the new symbols `OBSch` (the logical observe channel), `DECch` (the logical decoupler channel), `DEC2ch`, `DEC3ch` instead. Table 17 summarizes the old and new naming conventions. A simple translation will make your pulse sequence compatible with the `rfchannel` mechanism.

[1] On the other hand, using new utilities may make a sequence partly incompatible with previous versions of VNMR, but as the majority of the VNMR users use the current release, “horizontal” portability (same release, but different platforms) is more important that trying to keep a sequence compatible with older versions of VNMR.

Table 17. RF channel naming convention

Old symbol	*New symbol*
TODEV	OBSch
DODEV	DECch
DO2DEV	DEC2ch
DO3DEV	DEC3ch

Functions with Device Addresses

The channel naming complication above can be avoided if functions requiring a device address are replaced by functions that addresses that logical channel directly. There are only very few exceptions (namely, some `genpulse` type function calls) where this isn't possible (or at least not straightforward). Table 18 gives some guidelines on what functions to use:

Table 18. Equivalent PSG functions with and without device address

Functions with device address	*Equivalent functions without device address*
`rlpower, power`	`obspower, decpower, dec2power, dec3power`
`rlpwrf, pwrf`	`obspwrf, decpwrf, dec2pwrf, dec3pwrf`
`offset`	`obsoffset, decoffset, dec2offset, dec3offset`
`stepsize`	`obsstepsize, decstepsize, dec2stepsize, dec3stepsize`
`genqdphase`	`txphase, decphase, dec2phase, dec3phase`
`gensaphase`	`xmtrphase, dcplrphase, dcplr2phase, dcplr3phase`
`genpulse`	`pulse, decpulse`
`genrgpulse`	`rgpulse, decrgpulse, dec2rgpulse, dec3rgpulse`
`genshaped_pulse`	`shaped_pulse, decshaped_pulse, dec2shaped_puls, /dec3shaped_pulse`
`gen_apshaped_pulse`	`apshaped_pulse, apshaped_decpulse, apshaped_dec2pulse`
`genspinlock`	`spinlock, decspinlock, dec2spinlock, dec3spinlock`
`prg_dec_on`	`obsprgon, decprgon, dec2prgon, dec3prgon`
`prg_dec_off`	`obsprgoff, decprgoff, dec2prgoff, dec3prgoff`

Note that, strictly speaking, `power` and `rlpower` (and also `pwrf` and `rlpwrf`) are not equivalent functions, because they require different argument types (real-time integer vs. C double). The next section covers this difference further.

The "gen" type functions are "not officially supported" (i.e., there is no manual page for these utilities). They have been used by people that wanted to implement rf channel independent pulse sequences before the `rfchannel` mechanism existed. This feature

alone added considerable complexity to such sequences and made them very hard to read. Also, the “gen” type functions have the additional disadvantage of not being recognized by the `dps` utility. All these sequences can now be written for fixed logical rf channels, and the channel switching can be achieved much more easily using the `rfchannel` parameter.

The only case where “gen” type functions are still recommended is for some particular types of simultaneous pulses (shaped and rectangular). For pulses on “adjacent” logical channels (starting with `OBSch`), `simpulse`, `sim3pulse`, `simshaped_pulse` and `sim3shaped_pulse` are perfectly adequate. However, if simultaneous pulses are to be performed on either non-adjacent channels (e.g., `OBSch`, `DEC2ch`) or on adjacent channels not including `OBSch`, it is still better to use “gen” type functions. The simpler “sim” pulse functions don’t handle the case the case of any of the pulse lengths being set to zero very well.[2] Such functions may be acceptable for testing purposes, but for a real experiment you don’t want to really use these functions with any of the pulse widths set to zero (at least for the current VNMR release). As mentioned before, using “gen” type functions will unfortunately make these pulses “invisible” to `dps`.

Replacing power and pwrf Statements

The following construct was used in a large number of pulse sequences:

```
double pwxlvl;
(..)
pwxlvl = getval(“pwxlvl”);
(..)
initval(pwxlvl,v10);
(..)
power(v10,TODEV);
```

This has numerous disadvantages:

- It is complicated and lengthy;
- It requires reserving one of the few real-time variables (remember that with `initval` you *must not* use this real-time variable for anything else in the sequence!);
- There are often pages between the `initval` call and the actual use of the real-time variable containing the power value. Upon reading such sequences, it is often very hard to tell what power level is actually used, because with every `power` statement you have to go back in the sequence and look for the corresponding `initval` call.

It is much preferable to simply use

```
obspower(getval(“pwxlvl”));
```

which avoids using and initializing a real-time variable and avoids an extra C variable declaration and initialization, and the VNMR parameter name occurs where it is actually used in the sequence.

[2] In the case of `simpulse` and `sim3pulse`, some extra (non-sense) FIFO words are produced for the pulse on the “unused” channel (i.e., it tries performing a timer word of zero length), and for `simshaped_pulse` and `sim3shaped_pulse` the waveform generator on the unused channel is still being reset (as if there was a shaped pulse of zero length).

C Constructs for Phase-Sensitive *n*D NMR

Many pulse sequences use outdated (rather complicated and sometimes even questionable) constructs for f_n coherence selection (States, TPPI, FAD). A preferred set of constructs is described in detail in Chapter 19, “Multidimensional Experiments,” on page 215. In particular, it should be noted that the contents of the VNMR `phase`, `phase2` and `phase3` parameters are now already extracted and can be used in the *integer* variables `phase1` (*not* `phase`!), `phase2`, and `phase3`. Also, the real-time variables `id2`, `id3`, and `id4` should be used for setting up TPPI or FAD. This not only is simpler than referring the constructs used earlier, it also is safer and easier to understand.

24.3 General Considerations

This sections just provides a few additional hints of more “philosophical” nature.

Multipurpose Sequences

One of the nice features of VNMR pulse sequences is that you can have logical branchings based on VNMR parameters, allowing combining a number of related experiments into a single pulse sequence. But don’t overuse this feature by trying to “pack the world” into a single sequence! You can use this feature to combine *closely related* experiments (such as HMQC, HMQC with nulling by partial inversion recovery, and HMBC). Essentially use this capability to turn on or off specific features of a single experiment, and write separate sequences where the combination would be hard to understand or would result in most of the pulse sequence being in an `if` or `else` branch!

Using dps

The `dps` command is a useful and powerful tool for quickly controlling the experiment setup before starting a real acquisition. We fully appreciate the value of this command (we use it ourselves all the time!), and we will continue to further develop `dps`. However, nothing is perfect, and `dps` has its deficiencies, mainly:

- It does not display “gen”-type pulses and simpulses. “G” type functions (not discussed in this manual) have the same problem.
- It does not display pre-pulse and post-pulse delays.
- It does not display a number of other statements that may be relevant to certain users.
- The statement `zgradpulse` is not displayed (this has been fixed in VNMR 5.1).
- The number of events that can be displayed is limited and may not be sufficient for very complex experiments (like certain heteronuclear 3D PFG sequences).
- Anumber of other—perhaps minor—problems have been noticed by users.

Most of these problems can be circumvented, and different approaches have been used to achieve this:

- Some users have created a modified version of a pulse sequence just for the purpose of using `dps`, while the “real” pulse sequence is used for the “real”

acquisition only (with the known deficiencies of `sim2shaped_pulse` and `sim3shaped_pulse`, this may actually make real sense!).

- Others have resorted to explicit coding of pre-pulse and post-pulse delays by using explicit gating and extra delays around rf pulses, instead of relying on the features built into functions like `rgpulse`.
- Even extra `if` and `else` branchings were built into certain sequences (one branch for `dps`, one for acquisition).

In all these cases, you should ask yourself: is it worth the effort? Is it worth giving up the readability of a sequence, just to make `dps` work?

Some of the approaches like the explicit coding of pre-pulse and post-pulse delays may not work because they result in more pulse sequence elements than `dps` can display! In any case, it will be a long way until `dps` can display each and every detail (many users may not want that, because it can lead to excessively complex `dps` displays).

The question then would be: couldn't tools, such as `apdecode`[3] that display the Acode instructions, replace certain aspects of `dps` (because they *really* show *every detail*)? Displaying the Acode, on the other hand, can be too complex for routine use, but it certainly can be useful for certain stages of pulse sequence debugging!

[3] `apdecode` is available from the user library, see also Section 8.2, "Looking at Acode," on page 69. It is a tool to display the data and instructions in the Acode, as they are executed by the acquisition CPU.

Chapter 25. Debugging a Pulse Sequence

Many of today's pulse sequences have grown into an astounding complexity, and in order to cope with the growing range of experimental demands, modern NMR instruments have become extremely complex machines. Even though Varian tries hard to keep the rf and digital scheme as simple as possible (not because this is cheaper but because in most cases simpler schemes are better and less prone to failures). On the other hand, every programmer makes mistakes[1], and in a complex instrument there is also a certain chance that something fails or does not work as expected.

Overall, there is a fair chance that a new, complex experiment does not work as expected in a first attempt. If this happens, it would be a bad idea to just call a Varian service or customer support person claiming that "my xyz sequence does not work." At the very least one should be able to tell the following:

- What is the instrument type and configuration?
- What was the exact VNMR software release?
- Is this a standard sequence? A sequence from the user library (in this case you should consider contacting the author of the sequence directly, e.g., via e-mail or telephone)? One that you modified yourself?
- What exactly are the symptoms? (Can you fax a plot illustrating the problem or send an FID by e-mail, or mail it on a floppy?)
- Was there any error message?
- What exactly were the parameters in use? (You may want to consider sending a parameter printout by fax or sending an entire parameter set by e-mail or on a floppy!)
- What did you do to further locate the problem? (The sections below should give you some hints on what you could do to track down a problem.)
- Do other (similar or simpler) experiments work without problems? (Very often it is not really clear whether one is dealing with a hardware or a software problem!)
- What was the last experiment that worked properly?
- What happened since then?

Even with all this information, you might spend considerable time on the phone or exchanging e-mail, and it is in the last two points where the user can be most helpful in diagnosing and fixing the problem, saving service time and costs, making e-mail or telephone calls more efficient, and—last but not least—speeding up the entire process of getting the experiment to work.

Given the complexity of modern experiments, this may not always be an easy task. It may require a fair amount of analytical and creative thinking to locate and fix a particular problem, and certainly knowledge about the functionality of the experiment, as well as about the internal functionality of the NMR instrument is most helpful for this task. Of course, it takes time to acquire the knowledge that is necessary to debug a pulse sequence. One motivation for creating this manual was to shorten the learning curve for advanced spectrometer operation and debugging. With the right approach, you should not only be faster in designing and debugging new pulse sequences, but you

[1] Many people go as far as claiming that there is no software without bugs!

will also be able to save service time and costs by providing a more accurate description of eventual problems.

It is difficult, if not impossible, to provide a general recipe fro debugging a pulse sequence or locating an instrumental problem. The tips below are just options. The "correct" choice and sequence of actions is depending on the nature of the problem.

25.1 Debugging the Parameters

First, you want to find out whether the problem is due to a simple parameter mis-setting, because this would be the easiest and fastest problem to solve.

- Check `dg` and `dgs` for obvious parameter errors.
- Check with `dps` whether the parameters really are what you think they are. You may have used a pulse type parameter for a delay or, much worse, a delay parameter for a pulse duration (this can burn your probe and damage your sample!).
- Important parameter may be hidden from the display. Check for the values of all parameters on which you do a `getval` or a `getstr` in the pulse sequence.

Parameter mis-settings may result from errors or omissions in the setup macro—you may want to check for problems there. Certainly, if there is a parameter problem, this should be fixed in the setup macro, to avoid the same problem next time.

25.2 Debugging the Software

Next, you want to check whether there *is* a software problem, either a faulty sequence or a malfunction in the acquisition software.

- You also may want to kill `Acqproc`, restart it, and retry the experiment, just to check whether the problem is reproducible. Power surges, vibrations or temperature variations can severely affect cancellation experiments.
- Analyzing the FID or the spectrum can often indicate the nature of a problem (lack of cancellation, artifacts, etc.).
- Use `dps` to verify that the sequence corresponds to what you intended it to do.

If you don't see any obvious source for the problem, try narrowing down the area in which the problem is located:

- Simplify the experiment either by setting selective parameters to zero or by changing flag parameters (if there are any).
- Check for phase cycling errors by using `nt=1,2,4,8,16,32,64,128`. In cancellation experiments, you should see whether you get cancellation, and how much. Apart from the cancellation, you should see a steady increase of the signal-to-noise ratio by a factor of 1.4 with every increment. For 2D experiments, use `ni=0` (measure the first increment only)[2]. For multiple-quantum filtered experiments (or *n*D experiments that acquire echo-type signals in general), it is also necessary to do the same test with `d2=ni/(2*sw1)` to check whether

[2] In non-PFG absolute-value 2D experiments, there will be one step in the `nt` array that shows no increase in signal-to-noise. This is the point where f_1 quadrature detection is achieved by subtracting (N+P) from (N-P) type spectra.

multiple-quantum signals are collected at all (the first increment should contain only noise if the experiment is functioning properly). For States-type phase-sensitive *n*D experiments, use `phase=1` (and/or `phase2=1`).

- Use `apdecode` (supplied in the VNMR user library) to verify that the sequence of events in the pulse sequence is what you want, including events that are not shown by the `dps` command.
- Make the pulse sequence print parameter values and variables, maybe at several points in the sequence. This can reveal misconceptions such as the use of real-time variables in C calculations or any attempts to extract values from a displ.ay or processing parameter (only acquisition parameters can be accessed in a pulse sequence).
- It is easy to get mixed up in C decisions (or complex C constructs, in general). If the execution path within a complex sequence is not clear from looking at the `dps` or `apdecode` output, you may want to check points at certain places in the sequence, such as the line:

  ```
  printf("starting refocusing period\n");
  ```

- If all this doesn't help, simplify the pulse sequence code by progressively removing or commenting out sections of the pulse sequence (of course, you should keep a copy of the original sequence!) to try locating the problem within the sequence.

25.3 Debugging the Hardware

WARNING: **High voltages are present inside the console!**

WARNING: **Always have all cables properly connected or terminated when performing experiments, or while rf output is being generated (you could damage expensive power amplifiers!). For analyzing the output of power amplifiers (in particular for the 1 KW amplifiers used in solids NMR) you will need special power attenuators.**

WARNING: **Don't pull boards while the power is switched on, and never pull boards without protection against static electricity!**

The one important rule for debugging hardware problems is to use *simple* tools (pulse sequences and experiments)! Complex pulse sequences may be a good measure for testing the overall performance of a spectrometer, but they are useless for locating a specific hardware problem in an NMR spectrometer.

- Is the probe tuned properly? *Can* it be tuned at all, and on all channels?
- Has the proper quarter-wavelength cable been selected?
- Is there no problem with the lock?
- Try a similar, but different experiments (a different pulse sequence), just to double-check whether you have a hardware problem.
- If that fails as well, try very simple experiments, like `s2pul`, `d2pul`, `sh2pul`, `g2pul` or the like, to further narrow down the possibilities.
- Double-check connections between the console, magnet leg, preamplifiers, and probe.

- Check fuses in the console.
- Periodically check the fans in the console: overheated boards can easily cause a malfunction of the spectrometer!

Further hardware troubleshooting requires hardware diagnostics tools such as an oscilloscope, BNC cables and a set of rf attenuators. You may want to seek the help of a local expert for accessing cables and connectors within the console and for analyzing power rf output.

- Check whether rf (pulses, decoupler frequency) is generated and gated properly at the output of the transmitter board.
- Then follow the signal path and check the input of the amplifiers (after the attenuators).
- Given the appropriate attenuators (**observe the warnings above!),** you may now check the power output at the amplifier outlets.
- To check the rf path in the magnet leg, you can then repeat this test at the input to the probe.
- To check for the signal path *from* the probe, you need a strong sample; otherwise, you just observe noise. First, check for the presence of the L.O. frequency at the preamplifier (without L.O. you will not even observe noise behind the mixer!). The signal at the input to the preamplifier may be difficult or impossible to see on the oscilloscope, because the signal level is very low at this point. You may have a better chance at the input to the receiver.
- Carefully analyze the FID: Do the signals look normal? Is there truncation? Do you see unnatural “spikes”? Do both channels have the same or a similar amplitude? A good and easy way to check for signal overload at the ADC is the command `ddff(1)` that shows the numeric contents of the FID.

Symbols

Numerics

A

Index

B

C

D

E

F

Index

J

K

L

M

N

Index

Q

R

S

Index

T

U

V

W

X

Z