

Quick Start Manual Shaped Gradient Library

Revision: Draft A
Author: M. Meiler
February 17, 2005

Table of Contents

Introduction.....	3
Purpose and Scope.....	3
Setup, and Files.....	3
Setup.....	3
Configuration file (sgl.h).....	3
Shaped Gradient Library file (sgl.c).....	4
Function declarations (add_on.h).....	4
Assumptions and Rules.....	4
Units and Assumptions.....	4
Units:.....	4
Assumptions:.....	5
Error Messages.....	5
Example:.....	5
Waveform Creation.....	5
Write-to-Disk Flag.....	6
Rollout-Flag.....	6
Waveform Name.....	6
Waveform Header.....	6
Example.....	6
Annotated Example of Simple Pulse Sequence.....	8
Annotated Example of Simple Pulse Sequence Using Wrapper Functions.....	20

Introduction

Purpose and Scope

The Shaped Gradient Library (SGL) Quick Start Manual is intended to give the user a fast overview on how to use the shaped gradient library and their wrapper functions for pulse sequence programming. The Shaped Gradient Library (SGL) is a collection of pulse sequence programming functions and structures to aid the creation of shaped gradient waveforms. It simplifies pulse sequence programming by providing higher level functions for commonly used pulse programming.

The goal of the Shaped Gradient Library is two fold. a) It provides the user with an easy interface to create and use shaped gradient pulses inside the pulse sequence code, and b) it provides functions and functionality to enable the user to create their own gradient waveform pulses tailored to their specific needs. Wrapper functions build on top the SGL further enhance the ease of use for the pulse sequence programmer. These functions are intended to reduce the programming by providing functionality for most of the commonly encountered gradient waveforms in most pulse sequences. Both the SGL functions and the wrapper functions will result in the same outcome, since the wrapper functions are build on top of the SGL. Use of one or the other is purely a personal choice of the pulse sequence programmer with each of them having its own advantages and disadvantages.

The attached gradient-echo pulse sequences (*demo_gems* and *demo_gems_wrap*) are examples of how to use the SGL with and without wrapper functions. Both sequences are identical except for the use of the wrapper functions. The sequence *demo_gems* is written without any wrapper functions and relies solely on the functions provided within the SGL, while the sequence *demo_gems-wrap* makes extensive use of the wrapper functions and the predefined structures within the wrappers.

Setup, and Files

Setup

The shaped gradient library consist of three (3) files – a configuration file named *sgl.h* the library file containing the functions named *sgl.c* and the *add_on.h* file containing the explicit function declarations for seqgen. All three files are required for the Shaped Gradient Library to compile and execute correctly. Waveforms created by the Shaped Gradient Library are stored in the subdirectory *~/vnmrsys/shapelib/* and are labeled by the postfix *.GRD*.

Configuration file (sgl.h)

The configuration file *sgl.h* defines default directory settings and hardware limits used by the Shaped Gradient Library. It is advisable to verify that the hardware limits represent the actual values used for the respective imaging / spectroscopy system. All constants defined by *sgl.h* are in capital letters to clearly mark them as defined constants and to avoid ambiguity with variables used throughout the code in the Shaped Gradient Library and pulse sequences.

Constants defined by *sgl.h* are described by the comment following their definition. A representative example of a constant definition extracted from the *sgl.h* file is the gradient granularity:

```
#define GRADIENT_RES 0.000005 /* Gradient resolution / granularity */
```

The constant used in the example above is used throughout the Shaped Gradient Library to ensure that all timing events and waveform points fall on the timing granularity of the gradient controller.

Shaped Gradient Library file (*sgl.c*)

The Shaped Gradient Library file, *sgl.c*, is a collection of functions for the creation and administration of gradient waveforms. Functions are divided into user level functions and support functions. User level functions are commonly used physics calculations and gradient waveform creation functions. Support functions are functions used within the gradient library to support physics calculations and waveform creation.

The Shaped Gradient Library file, *sgl.c*, automatically includes the configuration file, *sgl.h*. All structure definitions, structure initialization, shaped gradient library function, and explicit function declarations are contained within the Shaped Gradient Library file *sgl.c* itself. Including the Shaped Gradient Library file, *sgl.c*, with an include statement at the top of the pulse sequence file makes all functions immediately available to the pulse sequence programmer.

Functions inside the Shaped Gradient Library can be divided into user level functions and support functions. The user level functions can be further organized into:

Structure initialization functions

Structure display functions

Calculation functions (calc-functions)

Function declarations (*add_on.h*)

This file is required since the seqgen parser can not create the required explicit function declarations from library files. It contains all the function declarations and inline functions defined in *sgl.c* prefixed with “*t_*” and “*x_*”.

Assumptions and Rules

The gradient library has a few assumptions and rules. The rules mainly concern timing and parameter units.

Units and Assumptions

The Shaped gradient library assumes that all values passed have the following units.

Units:

Description	Unit
Times are in seconds	[s]
Gradients are in gauss per centimeter	[G/cm]
Bandwidth and spectral width are in Hertz	[Hz]

Description	Unit
0 th moments are in seconds gauss per centimeter	[s G /cm]
1 st moments are in seconds squared gauss per centimeter	[s ² G /cm]
Field of View is in millimeter	[mm]
Slice thickness is in millimeter	[mm]

The Shaped Gradient Library will force all timing values to the next integer multiple of the granularity/resolution. Gradient duration's supplied to the Shaped Gradient Library that do not adhere to the timing granularity of the gradient channel will be flagged and an error message will be returned.

Assumptions:

The Shaped Gradient Library will not stop the compilation or execution if an error occurs. It is the responsibility of the user / pulse sequence programmer to check the status of the error flag after each invocation. The library was designed with the concept that all structure members are descriptors and not controls. Hence they provide a description of the waveform, but do not control it.

Error Messages

The Shaped Gradient Pulse Library will display an error message as soon an error is detected and return to the calling function / sequence, but will not abort execution of the pulse sequence. The user / pulse sequence programmer is advised to evaluate the error flag after the function call. Every time an error is reported the error flag in the respective structure is set, indicating the cause of the error. The user /pulse sequence programmer then has two ways in handling the error. He/She can either modify the function call and call the function again with the corrected parameters or abort the sequence with an abort message.

All error messages contain an error number as defined in the Shaped Gradient Library, an associated text message describing the error, the function reporting the error, and the line number within the Shaped Gradient Library, *sgl.c*, at which the error occurred.

An example of an error message is given below.

Example:

```
!!!!!!!!!!!!!!!!!!!!!! E R R O R !!!!!!!!!!!!!!!!!!!!!!!
Granularity violation - Timing does not adhere to granularity.

ERROR No.: 12
File:      sgl.c
Function:  calcGeneric
Line:     7169
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Waveform Creation

The Shaped Gradient Library uses floating points numbers for the description of waveform points. Conversion to normalized DAC units occur only when waveform is written to disk. All waveforms / waveform segments are created in memory for later concatenation by the respective higher level functions in the generic and physics layer.

Note: Waveforms are normalized to the maximum DAC value. During payout the waveform will be scaled

to the correct value with the maximum gradient magnitude supplied in the structure. This latter value is also entered in the gradient header.

Note: Gradient waveforms contain the gradient polarity and hence should only be multiplied by the magnitude for playout and not the amplitude. The shaped gradient playout functions multiply the waveform by the scaling factor and the waveform already contains the correct polarity!

All created gradient waveforms are positive with the exception of the primitive gradients. These gradients will carry the user defined sign. Generic gradients create positive waveforms by default, but can create negative waveforms by changing the structure element *polarity*.

Write-to-Disk Flag

The Write-to-Disk flag, contained within the structure of all waveform creating functions, determines whether a waveform is written to disk or only resides in memory space. Enabling of the Write-to-Disk flag will cause the waveform to be converted into DAC (Digital to Analog Converter) values, and be written to disk in the correct format for waveform play-out including a header file describing the basic waveform parameters. Physics functions have a default setting causing all waveforms to be written to disk. If a different behavior is required the user / pulse sequence programmer must change the value of the Write-to-Disk flag. Allowed values for the Write-to-Disk flag are *TRUE*, which writes the waveform to disk and *FALSE*, which will suppress writing the waveform to disk. Irrespective of the Write-to-Disk flag state, the waveform is always resident in memory.

Rollout-Flag

The Rollout-Flag controls if the gradient waveform is written to disk in a compressed mode or in a rolled out mode. The compressed mode allows the specification of waveform points for up to 255 tick values, while the rollout mode assigns each tick an explicit waveform value. Independent of the Rollout-Flag waveforms in memory are always rolled out. The Rollout-Flag only takes effect if waveforms are written to disk.

Waveform Name

All waveform functions use a default waveform name reflecting the waveform or its shape and a numerical suffix. Every time a waveform is created, the Shaped Gradient Library checks if a waveform with the same name already exists, and increments the suffix if it finds another waveform with the same name. Users / pulse sequence programmer can assign a different text name to the waveforms if desired. The behavior of the function for user assigned names is equivalent to that of default names. All names will carry an integer suffix, with auto-incrementing on duplication.

Note: Numbers appended to the waveform name start at one, and will be increased by one until the name appended by the number does not already exist. The number appended is hence not necessarily equal to the highest number, but rather the first numerical void.

Waveform Header

All waveforms written to disk will contain a gradient waveform header describing the basic waveform properties (waveform name, number of points, resolution, and maximum gradient strength of the waveform). The last line of the gradient header is defined by a pound sign followed by a colon “#:” . The following is an example of the gradient header for a readout gradient, as described by the default gradient waveform name.

Example

```
# Name:          read_12
# Points:        490
# RESOLUTION:   5.00000000e-06
# STRENGTH:     4.53440799e+01
#:
1365           1
2731           1
4096           1
5461           1
6826           1
8192           1
9557           1
10922          1
12288          1
13653          1
15018          1
.
.
.
```

Waveform Administration

The Shaped Gradient Library contains internal functions for waveform administration. All waveforms written with the `writeToDisk` function are added to a waveform list. Gradient administration allows initialization (zeroing) the list, adding gradient waveforms to the list or removing gradient waveforms from the list. There are two basic function for waveform administration 1) `gradAdmin`, which allows initialization, adding and removing of the gradient waveform list, and `removeFiles`, which clears the list and deletes all gradient waveform files contained in the list from disk

Annotated Example of Simple Pulse Sequence

The following is an annotated example of a simple gradient-echo (GE) pulse sequence using the Shaped Gradient Library. All annotations are printed *inbold italic* style, while the pulse sequence itself is printed in the font `Courier`.

```
/*
Multi slice Gradient-Echo sequence using the
Shaped Gradient Library
- multislice
- pre-pulse
- phase rewind
- crusher
*/

#include <standard.h>
#include <stdio.h>
#include <string.h>          /* include the string handling library */
#include <math.h>           /* include the math library */
#include "sgl.c"            /* include the Shaped Gradient Library */
#include "add_on.h"        /* include explicit function declaration for dsp */

Load the slew rate limiter with the minimum values to desable the slew rate limiter and take advantegous of the sjhaped
gradients..
pre_fidsequence()
{
    delay(0.1);
    sgl_download_master_decc_values();
}

pulsesequence()
{
    /* Internal variable declarations *****/
    double seqtime;
    char    rewind[MAXSTR];
    char    rfcoil[MAXSTR];
    int     list_number = 0;
    double  thk;
    double  debug;          /* debug flag to print out gradients 1 = on, print out, 0 = off*/
    double  maxGrad3;
    /* sequence timing variables */

```

```

double maxGradTime, maxRampTime, maxRewindTime, maxRewindRampTime;
double minTE, delay_90_acq, minTR, postDelay;
double fatsatTTime;

/* crusher pulse variables */
double gspoil, tspoil;

/* RF pulse variables */
double, p1, tpwrlf, tpwrl, flip1;
char plpat[MAXSTR];

/* pre pulse variables */
char fsat[MAXSTR], fsatpat[MAXSTR]; /*Fat suppression flag, FATSAT pulse pattern */
double tpwrfsat, tpwrfsatf; /* FATSAT power setting coarse/fine */
double fsatflip; /* FATSAT flip anle */
double pfsat, fsatfreq; /* FATSAT duration, FATSAT frequency */
double gcrushfs, tcrushfs; /* FATSAT crusher amplitude / duration */

```

Define the required variables holding the structures used in the pulse sequence. Each gradient used in the sequence will have a variable carrying its structure. One should note the refocus and dephase gradients have the same structure as well as the phase encode and phase rewind gradient. The phase encode and phase rewind gradient could be identical, but were created with different structures to allow duration equalization of the phase rewind gradient with the crusher gradient without effecting the echo time period.

```

SLICE_SELECT_GRADIENT_T slice; /* define slice select gradient structure */
REFOCUS_GRADIENT_T refocus; /* define refocus gradient structure */
PHASE_ENCODE_GRADIENT_T phase; /* define phase encode gradient structure */
PHASE_ENCODE_GRADIENT_T phase_rewind; /* define phase encode rewind gradient structure */
READOUT_GRADIENT_T readout; /* define readout gradient structure */
REFOCUS_GRADIENT_T dephase; /* define dephase gradient structure */
GENERIC_GRADIENT_T crusher; /* define crusher gradient structure */
GENERIC_GRADIENT_T fatsat; /* define crusher gradient structure for fatsat*/
RF_PULSE_T rfPulse; /* define RF-pulse (excitaion pulse) structure */
RF_PULSE_T fatsatPulse /* define RF FATSAT pulse structure */

/* Initialize parameter *****/
initparms_sis();
loop_check();

```

Here we initialize all structure variables created previously. The init functions will populate the structure members with the most common values, and

in doing so, reduce the amount of explicit assignment during the gradient waveform calculation and creation.

```
initSliceSelect(&slice);          /* init slice select structure */
initRefocus(&refocus);           /* init refocus gradient structure */
initPhase(&phase);               /* init phase encode gradient structure */
initReadout(&readout);           /* init readout gradient structure */
initDephase(&dephase);           /* init dephase gradient structure */
initGeneric(&crusher);           /* init crusher gradient structure */
initPhase(&phase_rewind);        /* init phase rewind gradient structure */
initGeneric(&crusher);           /* init crusher fatsat gradient structure */
initRf(&rfPulse);                /* init the RF excitation pulse structure */
initRf(&fatsatPulse);            /* init the RF FATSAT pulse structure */
```

These variables are used during gradient duration equalization and hold the duration of the longest gradient duration and longest ramp time. In order to avoid retaining values or having arbitrary values resident in memory it is good practice to initialize all variables.

```
/****** Initialize variables *****/
maxRewindTime    = 0.0;
maxGradTime     = 0.0;
maxRampTime     = 0.0;
maxRewindRampTime = 0.0;
fatsatTime      = 0.0;
maxGrad3        = gmax/1.75;      /* gradient limit to play 3 gradients */
```

Now we need to import VNMRJ variables from the GUI for use in the pulse sequence.

```
p1 = getval("p1");                /* get RF-pulse duration */
thk = getval("thk");              /* get slice thickness */
tpwrlf = getval("tpwrlf");
tpwrl = getval("tpwrl");
tspoil = getval("tspoil");
gspoil = getval("gspoil");
getstr("rewind",rewind);
getstr("rfcoil",rfcoil);

/* retrieve RF pulse variables */
p1 = getval("p1");                /* RF pulse duration */
getstr("plpat",plpat);           /* RF pulse shape */
flip1 = getval("flip1");         /* RF pulse flip angle */
tpwrlf = getval("tpwrlf");       /* RF pulse fine power */
tpwrl = getval("tpwrl");         /* RF pulse coarse power */
```

```

/* retrieve pre-pulse variable values */
getstr("fsat",fsat); /* RFATSAT flag */
getstr("fsatpat",fsatpat); /* FATSAT RF-pulse shape */
tpwrfsat = getval("tpwrfsat"); /* coarse power setting FATSAT pulse */
tpwrfsatf = getval("tpwrfsatf"); /* fine power setting FATSAT pulse */
fsatfreq = getval("fsatfreq"); /*fat offset frequency */
pfsat = getval("pfsat"); /* FATSAT pulse duration */
fsatflip = getval("fsatflip"); /* FATSAT pulse duration */
gcrushfs = getval("gcrushfs"); /* FATSAT crusher amplitude */
tcrushfs = getval("tcrushfs"); /* FATSAT crusher duration */

```

The creation of pre-pulse follow the example of the fatsat pulse blow. In this example the FATSAT crusher gradient is created from the inputs gcrushfs and tcrushfs describing the FATSAT crusher amplitude and duration, respectively. The FATSAT crusher gradient will have the shape of a half-sine as defined in the structure member fatsat.shape. This section of pulse sequence code is only executed if the FATSAT flag, fsat, is set. In all other cases this section is omitted. The variable fatsatTime contains the entire duration of the fatsat pre-pulse duration, consisting of the RF pulse duration, pfsat, for the FATSAT pulse and the FATSAT crusher duration.

```

/*****/
/* Setup pre-pulses *****/
if (fsat[0] == "y")
{
    fsat.amplitude = gcrushfs; /* assign FATSAT crusher amplitude [G/cm] */
    fsat.duration = tcrushfs; /* assign FATSAT crusher duration [s] */
    fatsat.shape = SINE; /* assign gradient waveform shape - half-sine */
    calcGeneric(&fatsat); /* create the FATSAT crusher gradient */
    if (fatsat.error) abort_message("Gradient library error --> see text window for more information. \n");
}

```

The "displayGeneric" function is only inserted for demonstration and monitoring purposes and can be omitted in the actual sequence or called within an if-loop when an error occurs or the debug variable is set to true.

```

if (debug) displayGeneric(&fatsat);
if (fatsat.error) abort_message("Gradient library error --> see text window for more information. \n");
fatsatTime = fatsat.duration +pfsat;
}

```

Calculate the RF-pulse power settings for the excitation pulse and the fatsat pulse. In order to do so, the RF structures defined above are proloaded with the respective values and the power settings are than calculated.

```

/*****/
/* Calculate RF power *****/
rfPulse.flip = flip1; /* assign excitaion pulse flip angle */
rfPulse.rfDuration = p1; /* assign excitaion pulse duration */

```

```
strcpy(rfPulse.pulseName,p1pat);          /* assign excitaion pulse name */
calcPower(&rfPulse,rfcoil);              /* calculate excitaion pulse power */
```

The "displayRf" function is only inserted for demonstration and monitoring purposes and can be omitted in the actual sequence or called within an if-loop when an error occurs or the debug variable is set to true. An example would be "if (debug) displayRf(&rfPulse);" which would only display the structure content in debug mode or "if (rfPulse.error) displayRf(&rfPulse);" which would only display the content of the structure if an error occurred. Note that if the slice structure is to be displayed the aforementioned statement needs to be placed before the "abort-message" or it will never be executed.

```
if (debug) displayRf(&rfPulse);          /* look at the structure and display if debug flag is set */
                                          /* not necessary in sequence code */
```

The RF-power calculation "calcPower" is followed by an error checking statement, that aborts the sequence with an error message if the Shaped Gradient Library created an error. The user / pulse sequence programmer can of course use other error handling methods like evaluation of the error code and re-calculation of the slice select gradient to obtain corrected input values.

```
if (rfPulse.error) abort_message("Gradient library error --> see text window for more information. \n");
```

And the same for the FATSAT pulse. This part is only execute if the FATSAT flag is set and omitted in all other cases. The power calculation is equivalent to that of the excitation pulse above

```
if (fatsat[0] == 'y')
{
  fatsatPulse.flip          = fatsatflip;          /* assign flipangle for FATSAT RF pulse*/
  fatsatPulse.rfDuration = pfsat;                /* assign duration of FATSAT RF pulse */
  strcpy(fatsatPulse.pulseName,fsatpat);         /* assign RF pulse name for FATSAT RF pulse */
  calcPower(&fatsatPulse,rfcoil);                /* calculate FATSAT RF power */
  if (debug) displayRf(&fatsatPulse);
  if (fatsatPulse.error) abort_message("Gradient library error --> see text window for more information. \n");
}
```

To create a slice select gradient we need to modify at least the following structure members. In this example it is only the slice thickness. The init functions intentionally do not set the slice thickness, RF-pulse name, and RF-pulse duration to thk, p1, p1pat, respectively. This is done to avoid confusion, ensure consistency, and make pulse sequences easier to read. For example in a spin echo sequence we have two slice select gradients – one for the 90 degree RF-pulse and one for the 180 degree RF-pulse, which might well have different duration's, shapes, and slice thickness. Having an automated assignment for only one leads to confusion, and obscures the clarity and readability of the pulse sequence code.

```
/* ***** */
/* Calculate slice select Gradient ***** */
slice.thickness = thk;          /* [mm] explicitly assign slice thickness */
```

```
calcSlice(&slice, &rfPulse);          /* calculate and create slice select gradient */
```

The “displaySlice” function is only inserted for demonstration and monitoring purposes and can be omitted in the actual sequence or called within an if-loop when an error occurs or the debug variable is set to true. An example would be “if (debug) displaySlice(&slice);” which would only display the structure content in debug mode or “if (slice.error) displaySlice(&slice);” which would only display the content of the structure if an error occurred. Note that if the slice structure is to be displayed the aforementioned statement needs to be placed before the “abort-message” or it will never be executed.

```
if(debug) displaySlice(&slice);          /* look at the structure and display if debug flag is set */
                                         /* not necessary in sequence code */
```

The gradient creation “calcSlice” is followed by an error checking statement, that aborts the sequence with an error message if the Shaped Gradient Library created an error. The user / pulse sequence programmer can of course use other error handling methods like evaluation of the error code and re-calculation of the slice select gradient to obtain corrected input values.

```
if (slice.error) abort_message("Gradient library error --> see text window for more information. \n");
```

The slice refocus gradient is calculated using the output of the slice select gradient. It should be noted that the input into the refocus gradient is the 0th slice refocusing moment “slice.balancingMoment0” which is the fraction of the 0th moment of the slice select gradient that needs to be refocused. Do not use slice.moment0 which is the 0th moment of the entire slice select gradient.

The gradient creation “calcRefocus” is followed by an error checking statement, that aborts the sequence with an error message if the Shaped Gradient Library created an error. The user / pulse sequence programmer can of course use other error handling methods like evaluation of the error code and re-calculation of the refocus gradient to obtain corrected input values.

```
/* ***** */
/* Calculate slice refocus Gradient ***** */
refocus.balancingMoment0 = slice.m0ref;          /* assign moment to be refocused */
refocus.maxGrad = maxGrad3;                    /* reduce max allowed gradient for obliquing */
calcRefocus(&refocus);                          /* create refocus gradient */
```

The “displayRefocus” function is only inserted for demonstration and monitoring purposes and can be omitted in the actual sequence or called within an if-loop when an error occurs or the debug variable is set to true. An example would be “if (debug) displayRefocus(&refocus);” which would only display the value of the structure members in debug mode or “if (refocus.error) displayRefocus(&refocus);” which would only display the value of the structure members if an error occurred. Note that if the refocus structure is to be displayed the aforementioned statement needs to be placed before the “abort-message” or it will never be executed.

```
if (debug) displayRefocus(&refocus);          /* look at the structure and display if debug flag is set */
                                         /* not necessary in sequence code */
```

```

if (refocus.error) abort_message("Gradient library error --> see text window for more information. \n");

/*****
/* Calculate phase encode gradient *****/
phase.fov = lpe*10; /* assign FOV in phase direction */
phase.maxGrad= maxGrad3; /* reduce max allowed gradient for obliquing */
phase.steps = nv; /* assign number of points */
phase.calcFlag =SHORTEST_DURATION_FROM_MOMENT; /* flag to calculate shortest possible gradient */
calcPhase(&phase);
displayPhase(&phase); /* look at the structure and display if debug flag is set */
/* not necessary in sequence code */

if (phase.error) abort_message("Gradient library error --> see text window for more information. \n");

/*****
/* Calculate readout gradient *****/
readout.acqTime = at; /* assign acquisition time */
readout.fov = lro*CM_TO_MM /* assign FOV in frequency direction */
readout.numPointsFreq = np/2.0; /* assign number of points in freq direction */
calcReadout(&readout);
displayReadout(&readout); /* look at the structure and display if debug flag is set*/
/* not necessary in sequence code */

if (readout.error) abort_message("Gradient library error --> see text window for more information. \n");
/*****
/* Calculate dephase gradient *****/
dephase.balancingMoment0=readout.m0ref; /* assign dephase moment */
dephase.maxGrad= maxGrad3; /* reduce max allowed gradient for obliquing */
calcDephase(&dephase);
displayDephase(&dephase); /* look at the structure and display if debug flag is set */
/* not necessary in sequence code */

if (dephase.error) abort_message("Gradient library error --> see text window for more information. \n");

```

Determine the longest duration of the three gradients (refocus, phase, and dephase) and recalculate the the gradients forcing the longest duration. After recalculation of each of the gradients the error flag is evaluated.

```

/*****
/* determine longest duration and adjust gradients */
/*****
maxGradTime = MAX(refocus.duration,dephase.duration);
maxGradTime = MAX(maxGradTime,phase.duration);
maxRampTime = MAX(refocus.tramp,dephase.tramp);
maxRampTime = MAX(maxRampTime,phase.tramp);

/*****

```

```

/* Recalculate grad. duration *****/
phase.duration = maxGradTime; /* assign duration */
refocus.duration = maxGradTime; /* assign duration */
dephase.duration = maxGradTime; /* assign duration */
phase.tramp = maxRampTime; /* assign ramp time */
refocus.tramp = maxRampTime; /* assign ramp time */
dephase.tramp = maxRampTime; /* assign ramp time */
phase.calcFlag = AMPLITUDE_FROM_MOMENT_DURATION; /*set calc flag to recalculate gradient*/
dephase.calcFlag = AMPLITUDE_FROM_MOMENT_DURATION; /*set calc flag to recalculate gradient*/
refocus.calcFlag = AMPLITUDE_FROM_MOMENT_DURATION; /*set calc flag to recalculate gradient*/
calcPhase(&phase); /* recalculate phase encode gradient for given duration */
calcRefocus(&refocus); /* recalculate refocus gradient for given duration */
calcDephase(&dephase); /* recalculate dephase gradient for given duration */
if (debug) /* display structures if debug flag is set */
{
    displayPhase(&phase); /* display phase encode structure members */
    displayRefocus(&refocus); /* display refocus gradient structure members */
    displayDephase(&dephase); /* display dephase gradient structure members */
}
if (phase.error || refocus.error || dephase.error)
{
    abort_message("Gradient library error --> see text window for more information. \n");
}

/* *****/
/* Calculate crusher *****/
crusher.amplitude= gspoil;
crusher.duration= tspoil;
crusher.maxGrad= maxGrad3; /* reduce max allowed gradient for obliquing */
calcGeneric(&crusher);
if (debug) displayGeneric(&crusher); /* look at the structure and display if debug flag is set*/
/* not necessary in sequence code */

if (crusher.error) abort_message("Gradient library error --> see text window for more information. \n");
maxRewindTime = crusher.duration;

If phase rewinding is selected, a phase rewind gradient is calculated. This gradient could be the same as the phase encode gradient except with opposite polarity. In this example however, we are playing simultaneously with the phase rewind gradient also the crusher gradients. As described before in order to play different gradients simultaneously they need to have the same duration and hence we can not use the phase encode gradient with opposite polarity for phase rewinding.

/* *****/
/* Calculate phase rewind *****/
if (rewind[0] == 'y')

```

```

{
phase_rewind.fov = lpe*10;          /* assign FOV in phase direction */
phase_rewind.steps = nv;           /* assign number of points */
phase_rewind.maxGrad= maxGrad3;    /* reduce max allowed gradient for obliquing */
phase_rewind.calcFlag =SHORTEST_DURATION_FROM_MOMENT; /* flag to calculate shortest possible gradient */
if (debug) displayPhase(&phase_rewind); /* look at the structure and display if debug flag is set*/
/* not necessary in sequence code */

calcPhase(&phase_rewind);
if (phase_rewind.error) abort_message("Gradient library error --> see text window for more information. \n");
}

```

Now we need to determine which gradient (phase or crusher) is longer and recalculate the other one. This only needs to be done if we do a phase rewind. Otherwise we only have a crusher gradient.

```

if (rewind[0] == 'Y')
{
maxRewindTime = MAX(phase_rewind.duration, crusher.duration);
maxRewindRampTime = MAX(phase_rewind.tramp, crusher.tramp);
if (phase_rewind.duration > crusher.duration)
{
/*****/
/* recalculate crusher duration */
crusher.duration      = phase.duration;
crusher.tramp         = maxRewindRampTime;
crusher.calcFlag      = MOMENT_FROM_DURATION_AMPLITUDE_RAMP;
calcGeneric(&crusher);
if (debug) displayGeneric(&crusher);
if (crusher.error) abort_message("Gradient library error --> see text window for more information. \n");
}
else
{
/*****/
/* recalculate phase rewind duration */
phase_rewind.duration = crusher.duration;
phase_rewind.tramp    = maxRewindRampTime;
phase_rewind.calcFlag = AMPLITUDE_FROM_MOMENT_DURATION_RAMP;
calcPhase(&phase_rewind);
if (debug) displayPhase(&phase_rewind);
if (crusher.error) abort_message("Gradient library error --> see text window for more information. \n");
}
}
}

```

```

/*****/
/*  Min TE *****/
minTE = slice.duration/2.0 + readout.duration/2.0 + maxGradTime;
if (te < minTE)
{
  abort_message("TE too short.  Minimum TE= %.2fms\n",minTE*1000);
}
delay_90_acq = te - minTE;

/*****/
/*  Min TR *****/
minTR = (fatsatTime + slice.duration + readout.duration + maxGradTime + delay_90_acq + maxRewindTime) * ns;
if (tr < minTR)
{
  abort_message("TR too short.  Minimum TR= %.2fms\n",minTR*1000);
}

postDelay = (tr/ns) - minTR;
seqtime = tr;

/*****/
/* PULSE SEQUENCE *****/
/*****/
initval(nv/2.0,v7);

status(A);

/* Begin phase-encode loop *****/
peloop(seqcon[2],nv,v5,v6);

/* Begin multislice loop *****/
msloop(seqcon[1],ns,v11,v12);
rotate();

Play the FATSAT pulse at the beginning of the sequence if the fatsat flag is set to 'y'
/* FATSAT PULSE *****/
if (fatsat[0] == 'y')
{
  obspower(fatsatPulse.powerCoarse);          /* set RF-pulse coarse power */
  obspwrf(fatsatPulse.powerFine);            /* set RF-pulse fine power */
  obsoffset(resto+fatsatfreq);
  Play the FATSAT RF-pulse
  shapedpulse(fsatpat,pfsat,zero,rofl,rofl); /* play the shaped FATSAT RF-pulse */
}

```

```

obsoffset(resto);
Play the fatsat crusher gradient. Here we use the WAIT keyword since we do not like to play any other event during the crusher gradient plays out.
obls_shapedgradient(fatsat.name, fatsat.name, fatsat.name,
                    fatsat.duration,
                    fatsat.amp, fatsat.amp, fatsat.amp, 1, WAIT);
}

/* Relaxation delay *****/
position_offset_list(pss,slice.amplitude,ns,resto,OBSch,list_number,v12);
xgate(ticks);
Play the slice select gradient. Here we use the NOWAIT keyword since we like to play the shaped RF-pulse while the gradient is playing. The NOWAIT keyword allows execution of the consecutive commands without waiting the the gradient waveform to finish payout.
/* RF pulse *****/
obspower(rfPulse.powerCoarse);
obspwrf(rfPulse.powerFine);

obl_shapedgradient("", "", slice.name, slice.duration,
                  0.0, 0.0, slice.amp, NOWAIT); /* Play slice select gradient, do not wait for execution */
delay(slice.rfDelayFront); /* wait until gradient has ramped up and delay */
/* to center RF-pulse*/
shapedpulse(plpat, slice.rfDuration, v1, rof1, rof1); /* play shaped RF pulse */
obsoffset(resto);
delay(slice.rfDelayBack); /* wait until gradient has ramped down */

Play phase encode, refocus, and slice encode gradient simultaneously. The increments for the two static gradients, refocus and dephase are 0.0 and their real-time variable is zero. Note that the refocus and dephase gradients are inverted since all gradient waveforms are positive.
/* Phase encode, refocus, and dephase gradient *****/
pe3_shapedgradient(phase.name, phase.duration,
                  -dephase.amp, phase.amp, -refocus.amp,
                  0.0, -phase.increment, 0.0,
                  zero, v6, zero);

Wait for the respective time between the gradients and the readout gradient to adjust for to the selected TE
delay(delay_90_acq); /* wait for time separating the gradients and readout */

Play the readout gradient. Here again we use the NOWAIT keyword since we like to control the acquisition while the gradient waveform is playing.
/* Readout gradient and acquisition *****/
obl_shapedgradient(readout.name, "", "", readout.duration, readout.amp, 0.0, 0.0, 1, NOWAIT);
delay(readout.atDelayFront); /* wait for readout gradient to ramp up */

```

```

acquire(np,1.0/readout.bandwidth);
obsoffset(resto);
delay(readout.atDelayBack);

/* and delay to center echo / fid */
/* wait for readout gradient to ramp down */

Play either the phase rewind gradient with the crusher gradients or the crusher alone. Note that the phase rewind is inverted with respect to the phase encode gradient.
/* Rewind / crusher gradient *****/
if (rewind[0] == 'y')
{
    pe3_shapedgradient(phase_rewind.name, phase_rewind.duration,
                      crusher.amp, -phase_rewind.amp, crusher.amp,
                      0.0, phase_rewind.increment, 0.0,
                      zero, v6, zero);
}
else
{
    obl_shapedgradient(crusher.name, "", crusher.name, crusher.duration,
                      crusher.amp, 0.0, crusher.amp, 1, WAIT);
}

delay(postDelay);
endmsloop(seqcon[1],v12);
endpeloop(seqcon[2],v6);
}

/* wait until TR is over */
/* end of the slice select loop */
/* end of the phase encode loop */

```

Annotated Example of Simple Pulse Sequence Using Wrapper Functions

The following is an annotated example of a simple gradient-echo (GE) pulse sequence using the Shaped Gradient Library and the wrapper functions. All annotations are printed in *bold italic* style, while the pulse sequence itself is printed in the font Courier.

```
/*
Gradient echo test sequence for demonstration of SGL usage
with wrapper functions
    - multislice
    - pre-pulse
    - phase rewind
    - crusher
*/
#include <standard.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "sgl.c"
#include "add_on.h"
#include "sglwrappers.h"

Load the slew rate limiter with the minimum values to desable the slew rate limiter and take advantegous of the sjhaped gradients..
pre_fidsequence()
{
    delay(0.1);
    sgl_download_master_decc_values();
}

pulsesequence()
{
    /* Internal variable declarations */
    char    rewind[MAXSTR];
    int     list_number = 0;
    double  debug;

    /* sequence timing variables */
    double  seqtime;
    double  tref , trewind, tramp;

```

```

double  minTE, te_delay, minTR, tr_delay;
double  fatsatTime;

/* crusher pulse variables */
double  gspoil, tspoil;

/* RF pulse variables */
double  pl, tpwrlf, tpwr1;
char    plpat[MAXSTR];

/* pre pulse variables */
char    fsat[MAXSTR],fsatpat[MAXSTR];          /* Fat suppression flag, FATSAT pulse pattern*/
double  tpwrfsat,tpwrfsatf;                  /* FATSAT power setting coarse/fine */
double  pfsat,fsatfrq;                       /* FATSAT duration, FATSAT frequency */
double  gcrushfs,tcrushfs;                   /* FATSAT crusher amplitude / duration */

GENERIC_GRADIENT_T      fatsat;              /* define FATSAT crusher gradient structure */

/* Initialize paramter *****/
initparms_sis();
loop_check();

/***** Initialize variables *****/
tref      = 0.0;
trewind   = 0.0;
tramp     = 0.0;
fatsatTime = 0.0;

/* retrieve RF pulse variables */
pl = getval("pl");                          /* grep rf pulse duration */
tpwr1 = getval("tpwr1");
tpwrlf = getval("tpwrlf");
getstr("rewind",rewind);
getstr("plpat",plpat);                       /* RF pulse shape */
getstr("fsat",fsat);                         /* FATSAT flag */
getstr("fsatpat",fsatpat);                   /* FATSAT RF-pulse shape */
tpwrfsat = getval("tpwrfsat");               /* coarse power setting FATSAT pulse */
tpwrfsatf = getval("tpwrfsatf");            /* fine power setting FATSAT pulse */
fsatfrq = getval("fsatfrq");                 /* fat offset freq */
pfsat = getval("pfsat");                     /* FATSAT duration */
gcrushfs = getval("gcrushfs");               /* FATSAT crusher amplitude */
tcrushfs = getval("tcrushfs");               /* FATSAT crusher duration */
gspoil = getval("gspoil");                   /* crusher amplitude */
tspoil = getval("tspoil");                   /* crusher duration */

```

```
debug      = getval("debug");                /* debug flag */
```

Initialize the required variables holding the structures used in the pulse sequence. Each gradient used in the sequence will have a variable carrying its structure.

```
/* **** */
/* Initialize gradient structures **** */
init_rf(&rf_pl,plpat,pl,getval("flip1"),rof1,rof2); /* initialize excitaion pulse */
init_rf(&rf_fsat,fsatpat,pfsat1,getval("fsatflip"),rof,rof2); /* initialize FATSAT RF pulse */
init_slice(&ss_grad,"gss",thk); /* initialize slice select gradient */
init_slice_refocus(&ssr_grad, "ssr"); /* initialize slice refocus gradient */
init_readout(&ro_grad,"ro",lro,np,sw); /* initialize readout gradient */
init_readout_refocus(&ror_grad,"ror"); /* initialize dephase gradient */
init_phase(&pe_grad,"pe",lpe,nv); /* initialize phase encode gradient */
init_phase(&per_grad,"per",lpe,nv); /* initialize phase rewind gradient */
init_generic(&crush_grad,"crush",gspoil,tspoil); /* initialize crusher gradient */
init_generic(&fatsat,"fatsat",gcrushfs,tcrushfs); /* initialize FATSAT crusher gradient */
```

Calculate the RF power settings for the excitaion pulse and the FATSAT pulse. Power for the FATSAT pulse is only calculated if the FATSAT flag is set. The display commands are not necessary in the pulse sequence and are only display the RF structure if the debug flag is set to "1".

```
/* **** */
/* Calculate RF power **** */
calc_rf(&rf_pl,"tpwr1","tpwr1f"); /* calculate excitation pulse power setting*/
if (debug) displayRf(&rf_pl); /* display RF structure when debug flag is set */
if (rf_pl.error) abort_message("Gradient library error --> see text window for more inforamtion. \n");

if (fatsat[0] == 'y')
{
  calc_rf(&rf_fatsat,"tpwrfsat","tpwrfsatf"); /* calculate FATSAT pulse power setting*/
  if (debug) displayRf(&rf_fatsat); /* display RF structure when debug flag is set */
  if (rf_fatsat.error) abort_message("Gradient library error --> see text window for more inforamtion. \n");
}
```

Calculate the gradients required for in the pulse sequence. Each gradient used in the sequence will have a variable carrying its structure.

```
/* **** */
/* Gradient Calculations **** */
calc_slice(&ss_grad, &rf_pl,WRITE,"gss"); /* calculate slice select gradient*/
calc_slice_refocus(&ssr_grad, &ss_grad, NOWRITE,"gssr"); /* calculate slice refocus gradient */
calc_readout(&ro_grad, WRITE, "gro","sw","at"); /* calculate readout gradient */
calc_readout_refocus(&ror_grad, &ro_grad, NOWRITE, "gror"); /* calculate dephase gradient */
calc_phase(&pe_grad, NOWRITE, "gpe","tpe"); /* calculate phase encode gradient */
```

Display the structure members is the debug flag is set.

```

/*****/
/* Display if debug flag is set *****/
if (debug)
{
displaySlice(&ss_grad);
displayRefocus(&ssr_grad);
displayReadout(&ro_grad);
displayDephase(&ror_grad);
displayPhase(&pe_grad);
}

```

Check ofr errors that might have occurred during the calculation and display the appropriate message to the user.

```

/*****/
/* Check for errors *****/
if ((ss_grad.error) || (ssr_grad.error) || (ro_grad.error) ||
(ror_grad.error) || (pe_grad.error))
{
abort_message("Gradient library error --> see text window for more inforamtion. \n");
}

```

The creation of pre-pulse follow the example of the fatsat pulse blow. In this example the FATSAT crusher gradient is created from the inputs gcrushfs and tcrushfs describing the FATSAT crusher amplitude and duration, respectively. The FATSAT crusher gradient will have the shape of a half-sine as defined in the structure member fatsat.shape. This section of pulse sequence code is only executed if the FATSAT flag, fsat, is set. In all other cases this section is omitted. The variable fatsatTime contains the entire

```

/*****/
/* Setup pre-pulse *****/
if (fsat[0]=='Y')
{
fatsat.shape = SINE;
calc_generic(&fatsat,WRITE);

```

The "displayGeneric" function is only inserted for demonstration and monitoring purposes and can be omitted in the actual sequence or called within an if-loop when an error occurs or the debug variable is set to true.

```

if (debug) displayGeneric(&fatsat); /* look at the structure and display */
/* not necessary in sequence code */
if (fatsat.error) abort_message("Gradient library error --> see text window for more inforamtion. \n");
fatsatTime = fatsat.duration + pfsat;
}

```

Use the wrapper function to equalize / block the slice refocus, dephase end phase encode gradient. This is required for simultaneous ployout, since all waveforms need to have the same duration.

```

/*****/

```

```

/* Equalize Gradients *****/
tref = calc_sim_gradient(&ror_grad, &pe_grad, &ssr_grad, tpe, WRITE, "gror","gssr","gpe","tpe");

If phase rewinding is selected, a phase rewind gradient is calculated. This gradient could be the same as the phase encode gradient except with opposite polarity. In this example however, we are playing simultaneously with the phase rewind gradient also the crusher gradients. As described before in order to play different gradients simultaneously they need to have the same duration and hence we can not use the phase encode gradient with opposite polarity for phase rewinding.
/*****/
/* Calculate phase rewind gradient *****/
if (rewind[0] == 'y')
{
  calc_phase(&per_grad,WRITE,"","");
  calc_generic(&crush_grad,WRITE);
  /*****/
  /* Display if debug flag is set *****/
  if (debug)
  {
    displayPhase(&per_grad);
    displayGeneric(&crush_grad);
  }

  /*****/
  /* Check for errors *****/
  if ((per_grad.error) || (crush_grad.error) )
  {
    abort_message("Gradient library error --> see text window for more inforamtion. \n");
  }

Now we need to determine which gradient (phase or crusher) is longer and which one has the longest ramp time. recalculate the other one. This only needs to be done if we do a phase rewind. Otherwise we only have a crusher gradient.
/*****/
/* Equalize gradients *****/
trewind = MAX(per_grad.duration,crush_grad.duration);
tramp = MAX(per_grad.rampTime,crush_grad.rampTime);

per_grad.duration = crush_grad.duration = trewind;
per_grad.rampTime = crush_grad.rampTime = tramp;
per_grad.calcFlag = crush_grad.calcFlag = AMPLITUDE_FROM_MOMENT_DURATION_RAMP;

calc_phase(&per_grad,WRITE,"","");
calc_generic(&crush_grad,WRITE);
/*****/
/* Display if debug flag is set *****/
if (debug)

```

```

    {
    displayPhase(&per_grad);
    displayGeneric(&crush_grad);
    }

    /*****
    /* Check for errors *****/
    if ((per_grad.error) || (crush_grad.error) )
    {
        abort_message("Gradient library error --> see text window for more inforamtion. \n");
    }
}
And the case where we only have a crusher gradient.
else /* otherwise use only crusher */
{
    calc_generic(&crush_grad,WRITE);
    if (debug) displayGeneric(&crush_grad);
    if (crush_grad.error) abort_message("Gradient library error --> see text window for more inforamtion. \n");
}

/*****
/* Min TE *****/
minTE = ss_grad.duration/2.0 + ro_grad.duration/2.0 + tref;
if (te < minTE)
{
    abort_message("TE too short. Minimum TE= %.2fms\n",minTE*1000);
}
te_delay = te - minTE;

/*****
/* Min TR *****/
minTR = (fatsatTime+ ss_grad.duration + ro_grad.duration + tref + te_delay + trewind) * ns;
if (tr < minTR)
{
    abort_message("TR too short. Minimum TR= %.2fms\n",minTR*1000);
}

tr_delay = (tr/ns) - minTR;
seqtime = tr;

if (nv == 0)
{
    pe_grad.amplitude = pe_grad.increment = 0;
    per_grad.amplitude = per_grad.increment = 0;
}

```

```

}

/* PULSE SEQUENCE *****/
initval(nv/2.0,v7);

/* TTL trigger to scope sequence *****/

status(A);
obspower(tpwr1);
obspwrf(tpwr1f);

/* Begin phase-encode loop *****/
peloop(seqcon[2],nv,v5,v6);

/* Begin multislice loop *****/
msloop(seqcon[1],ns,v11,v12);
rotate();
/* FATSAT PULSE *****/
Play the FATSAT pulse at the beginning of the sequence if the fatsat flag is set to 'y'
if (fsat[0] == 'y')
{
obspower(tpwrfsat);
obspwrf(tpwrfsatf);
obsoffset(resto+fsatfrq);
Play the FATSAT RF-pulse
shapedpulse(fsatpat,pfsat,zero,rof1,rof1);
obsoffset(resto);
Play the fatsat crusher gradient. Here we use the WAIT keyword since we do not like to play any other event during the crusher gradient plays out.
obl_shapedgradient(fatsat.name,fatsat.name,fatsat.name,
fatsat.duration,
fatsat.mag,fatsat.mag,fatsat.mag,1,WAIT);
}

/* Relaxation delay *****/
position_offset_list(pss,ss_grad.amplitude,ns,resto,OBSch,list_number,v12);
xgate(ticks);

Play the slice select gradient. Here we use the NOWAIT keyword since we like to play the shaped RF-pulse while the gradient is playing. The NOWAIT keyword allows execution of the consecutive commands without waiting the the gradient waveform to finish payout.
/* RF pulse *****/
obl_shapedgradient("", "",ss_grad.name,ss_grad.duration,0,0,ss_grad.amp,1,NOWAIT);

```

```

delay(ss_grad.rfDelayFront);
shapedpulse(plpat,p1,v1,rof1,rof1);
obsoffset(resto);
delay(ss_grad.rfDelayBack);

```

Play phase encode, refocus, and slice encode gradient simultaneously. Note that the increments for the static gradients refocus and dephase are set to zero and their real-time variable is zero. Note that the refocus and dephase gradients are inverted since all gradient waveforms are positive.

```

/* Phase encode, refocus, and dephase gradient *****/
pe3_shapedgradient(pe_grad.name, pe_grad.duration,
                  -ror_grad.amp, pe_grad.amp, -ssr_grad.amp
                  0.0, -pe_grad.increment, 0.0,
                  zero, v6, zero);

```

Wait for the respective time between the gradients and the readout gradient to adjust for to the selected TE

```

delay(te_delay);

```

Play the readout gradient. Here again we use the NOWAIT keyword since we like to control the acquisition while the gradient waveform is playing.

```

/* Readout gradient and acquisition *****/
obl_shapedgradient(ro_grad.name, "", "", ro_grad.duration, ro_grad.amp, 0, 0, 1, NOWAIT);
delay(ro_grad.atDelayFront);
acquire(np, 1.0/sw);
obsoffset(resto);
delay(ro_grad.atDelayBack);

```

Play either the phase rewind gradient with the crusher gradients or the crusher alone. Here again we use the keyword WAIT, since we do not want to continue until the gradients have finished playout. Note that the phase rewind is inverted with respect to the phase encode gradient.

```

/* Rewind / crusher gradient *****/
if (rewind[0] == 'y')
{
    pe3_shapedgradient(per_grad.name, per_grad.duration,
                    -crush_grad.amp, -per_grad.amp, crush_grad.amp,
                    0.0, per_grad.increment, 0.0,
                    zero, v6, zero);
}
else
{
    obl_shapedgradient(crush_grad.name, crush_grad.name, crush_grad.name, crush_grad.duration,
                    crush_grad.amp, 0, crush_grad.amp, 1, WAIT);
}
delay(tr_delay);
endmsloop(seqcon[1], v12);

```

```
    endpeloop(seqcon[2],v6);  
}
```