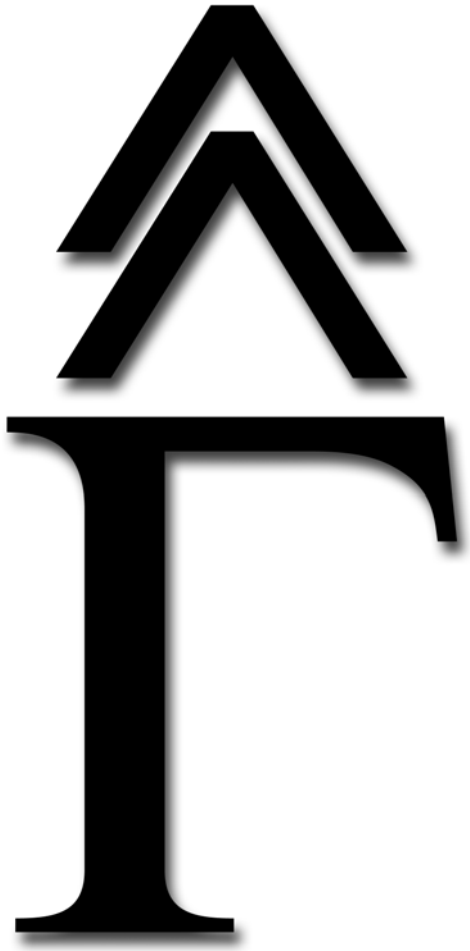
GAMMA

Basics Module



Basics Module

Table of Contents

1 Introduction

The first module supplied with GAMMA is called the ***Basics*** module. It doesn't have much to do with magnetic resonance simulations. It supplies functionality that is commonly used in GAMMA programs, such as facile input/output of parameters from/to ASCII files and facile input/output of parameters from a console window. This module does supply one class related to magnetic resonance, *spin isotopes*. These objects set up spin Hilbert spaces that are used in quantum mechanical based calculations.

Contents of the module are as follows.

- ***Gconstants*** - This provides GAMMA programs with knowledge of many physical constants used in MR simulations. The constants defined herein include π, $\boldsymbol{h}$, μ_e, μ_B, γ_{1H}, γ_e, and $\boldsymbol{g}$. Thus one can use expression *GAMMA1H* in a program for the proton gyromagnetic ratio (γ_{1H}) and *MU_E* for the electron magnetic moment (μ_e).
- ***Gutils*** - This gives GAMMA users the ability to build programs that run interactively, requesting input parameters from the user. The input may be supplied on the command line (for non-interactive use) when the program is invoked, or the program can wait for input values from the user. Default values may be specified so that the user may just hit return to input a reasonable parameter.
- ***Isotope*** - This class provides working spin isotopes. GAMMA users may declare spins in their programs and these intrinsically know their spin quantum numbers and gyromagnetic ratios. Additional "spins" may be added for special simulations, e.g. Ho3+ for electons with higher spin quantum values. Single Isotopes are not often used since GAMMA *spin systems* allow for the simultaneous manipluation of a collection of (interacting) spins.
- ***SinglePar*** - This class defines a single parameter in GAMMA to facilitate I/O of values from/to external ASCII files. This is not often used directly because GAMMA Parameter Sets can deal with muliple parameters including the ability to read/write entire ASCII files. Furthermore, GAMMA objects usually handle their own I/O to external files (through use of SinglePar and ParameterSet).
- ***ParameterSet*** - This class allows users to read and write sets of values from/to external ASCII files. Essentially the parameter set will perform all of the required parsing. If an external ASCII file is read into a parameter set the user may easily obtain any of the parameters therein for use in a program.
- ***StringCut*** - This module provides users with the ability to parse strings. Its utility is limited and largely available with Regex, howver Regex is not part of ANSI C++ (yet).

2 Constants

2.1 Overview

This file contains all of the GAMMA defined constants. These are values commonly used in multiple areas of magnetic resonance.

2.2 Available Consants

2.3 Standard Constants

2.3.1 PI

Usage:

```
#include <Basics/Gconstants.h>
extern const double PI
```

Description:

This value is π. If previously defined in the C++ compiler, the definition herein will not be used.

Return Value: double

Examples:

```
#include <gamma.h>
int main()
  { double dpi = PI; }                    // The value of pi
```

See Also: PIx2

2.3.2 PIx2

Usage:

```
#include <Basics/Gconstants.h>
extern const double PIx2
```

Description:

This value is 2 times π. That is $PIx2 = 2\pi$.

Return Value: double

Example:

```
#include <gamma.h>
int main()
  {
  double twopi = 2*PI;                    // Two time the value of pi
  double zero = twopi-PIx2;               // This should be zero
  }
```

See Also: PI

2.3.3 HUGE

Usage:

```
#include <Basics/Gconstants.h>
define HUGE HUGE_VAL
```

Description:

This value is a large number, hopefully no numbers on the system will be larger than it. If previously defined, the definition herein will not be used. Newer systems use HUGE_VAL, so HUGE and HUGE_VAL will work in GAMMA. The value is nice to use as a starting value when looking for maxima & minima.

Example:

```
#include <gamma.h>
int main()
  {
  double twopi = 2*PI;                  // Two time the value of pi
  double hval = -HUGE;                  // A huge negative nubmer
  double m = max(hvalue,twopi);         // This should be 2*pi also
  }
```

2.3.4 PLANCK

Usage:

#include <Basics/Gconstants.h>

const double PLANCK

Description:

This value is Plancks constant, h. In GAMMA this is defined as ***PLANCK = 6.6260755 x 10^{-34} J-sec***.

Example:

```
#include <gamma.h>
int main()
  { double h = PLANCK; }                // Set h to Planck's constant
```

2.3.5 HBAR

Usage:

#include <Basics/Gconstants.h>

const double HBAR

Description:

This value is Plancks constant divided by 2π, h. ***HBAR = h/(2π) = 1.05457266 x 10^{-34} J-sec***.

Example:

```
#include <gamma.h>
int main()
  { double hbar = HBAR); }              // This should be Plancks constant / (2*Pi)
```

2.4 Angle & Frequency

2.4.1 DEG2RAD

Usage:

#include <Basics/Gconstants.h>
extern const double DEG2RAD

Description:

This value is the factor needed to convert an angle in degrees to an angle in radians. **DEG2RAD = π/180 radians/degree**.

Example:

```
#include <gamma.h>
int main()
  {
  double Adeg = 90;                 // A 90 degree angle
  double Arad = Adeg*DEG2RAD;       // The same angle in radians
  }
```

See Also: RAD2DEG

2.4.2 RAD2DEG

Usage:

#include <Basics/Gconstants.h>
extern const double RAD2DEG

Description:

This value is the factor needed to convert an angle in radians to an angle in degrees. **RAD2DEG = 180/π degrees/radians.**

Example:

```
#include <gamma.h>
int main()
  {
  double Arad = PI;                 // A 180 degree angle in radians
  double Adeg = Arad*RAD2DEG;       // The same angle in degrees
  }
```

See Also: DEG2RAD

2.4.3 HZ2RAD

Usage:

#include <Basics/Gconstants.h>
extern const double HZ2RAD

Description:

This value is the factor needed to convert a frequency in Hz to a frequency in radians/sec. **HZ2RAD = 2π radians/cycle.**

Example:

```
#include <gamma.h>
int main()
  {
  double Fdeg = 127;                  // A frequency of 127 Hz
  double Frad = Fdeg*HZ2RAD;          // The same frequency in radians/sec
  }
```

See Also: RAD2HZ

2.4.4 RAD2HZ

Usage:

```
#include <Basics/Gconstants.h>
extern const double RAD2HZ
```

Description:

This value is the factor needed to convert a frequency in radians/sec to a frequency in Hz. ***RAD2Hz =1/(2π) cycles/radian***.

Example:

```
#include <gamma.h>
int main()
  {
  double Frad = 200;                  // A frequency of 200 /sec
  double Fdeg = Frad*RAD2HZ;          // The same frequency in Hz
  }
```

See Also: DEG2HZ

2.5 Field & Frequency Conversion

2.5.1 HZ2GAUSS

2.5.2 GAUSS2Hz

2.5.3 GHZ2GAUSS

2.5.4 GAUSS2GHz

Usage:

```
#include <Basics/Gconstants.h>
extern const double HZ2GAUSS
extern const double GHZ2GAUSS
extern const double GAUSS2HZ
extern const double GAUSS2GHZ
```

Description:

These values are the factors needed to convert a frequency and a matching field in Gauss for EPR.. ***HZ2GAUSS*** = ***7.1447751747 x 10^{-7} G/Hz***

Example:

```
#include <gamma.h>
int main()
  { double h2g = HZ2GAUSS; }          // The conversion factor
```

2.6 MR Constants

2.6.1 MU_E

Usage:

```
#include <Basics/Gconstants.h>
extern const double MU_E
```

Description:

This value is the electron magnetic moment, μ_e. ***MU_E = -9.2847701 x 10^{-24} J T^{-1}.***

Return Value: double

Example:

```
#include <gamma.h>
int main()
  { double mu = MU_E; }               // Set mu  to the electron magnetic moment.
```

2.6.2 BOHRMAG

Usage:

```
#include <Basics/Gconstants.h>
extern const double BOHRMAG
```

Description:

This value is the Bohr magneton, B_e. ***BOHRMAG = 9.2740154 x 10^{-24} J T^{-1}.***

Return Value: double

Example:

```
#include <gamma.h>
int main()
  { double Be = BOHRMAG; }              // Set Be  to the Bohr magneton.
```

2.6.3 GFREE

Usage:

```
#include <Basics/Gconstants.h>
extern const double GFREE
```

Description:

This value is the free electron g-factor, g. ***GFREE = 2.002319204386.***

Return Value: double

Example:

```
#include <gamma.h>
int main()
  { double g = GFREE; }                 // Set g  to the e- gyromagnetic ration.
```

2.6.4 GAMMAe

Usage:

```
#include <Basics/Gconstants.h>
extern const double GAMMAe
```

Description:

This value is the free electron gyromagnetic rato, γ_e. ***GAMMAe = 1.7608592 x 10^{11} T^{-1} s^{-1}.***

Return Value: double

Example:

```
#include <gamma.h>
int main()
  { double ge = GAMMAe; }               // Set ge to the e- gyromagnetic ratio.
```

2.6.5 GAMMA1H

Usage:

```
#include <Basics/Gconstants.h>
extern const double GAMMA1H
```

Description:

This value is the proton gyromagnetic rato, γ_{1H}. ***GAMMA1H = 26.7515255x10^{7}T^{-1}s^{-1}.***

Return Value: double

Example:

```
#include <gamma.h>
int main()
  { double gh = GAMMA1H; }                  // Set gh to the proton gyromagnetic ratio.
```

See Also: GAMMAe

2.6.6 DEFISO

Usage:

```
#include <Basics/Gconstants.h>
extern const string DEFISO
```

Description:

This value is what a default isotope is set to in GAMMA.

Return Value: string

Example:

```
#include <gamma.h>
int main()
  { string I = DEFISO;  }                   // Set I to the default isotope string
```

3 Utility Functions

3.1 Overview

The Basics module files named ***GUtils*** supply common functionality to GAMMA programs. In particular, they provide functions that allow users to build interactive GAMMA programs. They also supply functions that issue error messages in a standard format.

3.2 Available Functions

Command Line Input

Error Messaging

3.3 Command Line Input

3.3.1 query_parameter

Usage:

```
#include <Basics/Gutils.h>
void query_parameter(int argc, char* argv[], int par, const std::string& Q, std::string& V);
void query_parameter(int argc, char* argv[], int par, const std::string& Q, double&      V);
void query_parameter(int argc, char* argv[], int par, const std::string& Q, int&         V);
```

Description:

The function ***query_parameter*** is used to request a parameter from the user when the program executes. If the user supplies the parameter value on the command line when the program is run the query will be bypassed and the value supplied used. The function takes the number of arguments supplied on the command line, ***argc***, as well as a string of the arguements supplied on the command line, ***argv***. It will then look to see if the ***par***th value was supplied on the command line when the program was started. If that value was supplied, then it is used to set the value of ***V***. If the ***par***th value was not supplied on the command line the the string ***Q*** is written to the screen and the program will wait for the user to supply a value for ***V***. All overloads of the function ***query_parameter*** work identically but they allow for either integer, double, or string input. Note that when this function is used the user must *write main with the command line arguments*. That is, the program should begin with: int main(int argc, char* argv[]). Furthermore note that the 0^{th} argument on the command line will be the program executable name, thus making the next value to have the index of 1. These functions will pause program execution until a parameter value is specified.

Return Value: void

Examples:

```
#include <gamma.h>
int main(int argc, char* argv[])
  {
  int qn = 1;                                  // Start with index of 1 for input
  int iin;                                     // An integer value we will input
  query_parameter(argc,argv,qn++,              // Ask for the integer value if not
          "\n\tWhat Is The Integer? ", iin);// supplied as 1st one on command line
  string sin;                                  // A string value we will input
  query_parameter(argc,argv,qn++,              // Ask for the string value if not
          "\n\tWhat Is The String? ", sin);// supplied as 2nd one on command line
  }
```

If the above program were compiled to produce the executable named a.exe, then the command "./a.exe" would cause the program to request both the integer and the string. Were the command "./a.exe 7" issued then the program would automatically set iin to 7 and only request the string value from the user. Were the command "./a.exe 7 OK" issued then the program would not request information from the user but automatically set iin to 7 and sin to OK.

See Also: ask_set

3.3.2 ask_set

Usage:

```
#include <Basics/Gutils.h>
```

bool ask_set(int argc, char* argv[], int par, const std::string& Q, std::string& V);
bool ask_set(int argc, char* argv[], int par, const std::string& Q, double& V);
bool ask_set(int argc, char* argv[], int par, const std::string& Q, int& V);

Description:

The function ***ask_set*** is used to request a parameter from the user when the program executes. If the user supplies the parameter value on the command line when the program is run the query will be bypassed and the value supplied used. The function takes the number of arguments supplied on the command line, ***argc***, as well as a string of the arguements supplied on the command line, ***argv***. It will then look to see if the ***par***th value was supplied on the command line when the program was started. If that value was supplied, then it is used to set the value of ***V***. If the ***par***th value was not supplied on the command line the the string ***Q*** is written to the screen and the program will wait for the user to supply a value for ***V***. All overloads of the function ***ask_set*** work identically but they allow for either integer, double, or string input. Note that when this function is used the user must *write main with the command line arguments*. That is, the program should begin with: int main(int argc, char* argv[]). Furthermore note that the 0th argument on the command line will be the program executable name, thus making the next value to have the index of 1. The ***ask_set*** functions are very similar to the ***query_parameter*** functions except in this case a default value may be specified. That is, if **V** is set to a default value prior to the call to ***ask_set*** and that value is not supplied on the command line, the user may simply hit <RETURN> to use the default value of V. Also, these function will return true if the value is properly set, false if not.

Return Value: bool

Examples:

```
#include <gamma.h>
int main(int argc, char* argv[])
  {
  int qn = 1;                                  // Start with index of 1 for input
  int iin = 9;                                 // An integer value we will input
  ask_set(argc,argv,qn++,                      // Ask for the integer value if not
          "\n\tWhat Is The Integer? ", iin);// supplied as 1st one on command line
  string sin = "Anything";                     // A string value we will input
  ask_set(argc,argv,qn++,                      // Ask for the string value if not
          "\n\tWhat Is The String? ", sin);// supplied as 2nd one on command line
  }
```

If the above program were compiled to produce the executable named a.exe, then the command "./a.exe" would cause the program to request both the integer and the string. Were the command "./a.exe 7" issued then the program would automatically set iin to 7 and only request the string value from the user. If the user enters <RETURN> at that request sin will be set to "Anything". Were the command "./a.exe 7 OK" issued then the program would not request information from the user but automatically set iin to 7 and sin to OK.

See Also: query_parameter

3.4 Error Messaging

3.4.1 GAMMAerror

Usage:

```
#include <Basics/Gutils.h>
void GAMMAerror(const std::string& hdr, const std::string& msg, int noret=0);
void GAMMAerror(const std::string& hdr, int eidx, int noret=0);
void GAMMAerror(const std::string& hdr, int eidx, const std::string& pname, int noret=0);
```

Description:

The function ***GAMMAerror*** is used to issue common error messages. It is not often employed directly by GAMMA users. Rather it is invoked indirectly from within GAMMA supplied classes and functions. The function will first output the string ***hdr*** which typically indicates where the error arose (e.g. a class name). It will then issue either a standard error message with index ***eidx*** or a supplied error message ***msg***. Some error mesages may include an additional string ***pname*** which defines the error more specifically. The value ***noret*** is used to indicate whether an additional linefeed should be added following the error message output. Any non-zero value will supress the additional linefeed, the default is to include one.

Return Value: ***void***

Example:

```
#include <gamma.h>
int main()
  {
  string hdr("MyClass");
  string msg("This messed up");
  GAMMAerror(hdr,msg);
  }
```

3.5 Description

The Basics module files named ***GUtils*** supply common functionality to GAMMA programs. In particular, they provide functions that allow users to build interactive GAMMA programs. They also supply functions that issue error messages in a standard format.

3.5.1 Standard Errors

The table below lists the supplied error messages returned from the GAMMAerror function which takes only an error index, ***eidx***.

Table 1: GAMMA Error Messages

Index	Message	Index	Message
0	Program Aborting.....	**9**	Problems During Construction
1	Problems With Input File Stream	**10**	Cannot Access Internal Component
2	Problems With Output File Stream	**11**	Cannot Read From Input FileStream
3	Can't Construct From Parameter Set	**12**	End of File Reached
4	Cannot Construct From Input File	**13**	Cannot Open ASCII File For Read
5	Cannot Write To Parameter File	**14**	Cannot Read From Input File
6	Cannot Write To Output FileStream	**15**	Cannot Write To Output File
7	Cannot Produce Any Output	**x**	Unknown Error
8	Problems With Parameter Set		

The table below lists the supplied error messages returned from the GAMMAerror function which takes both an error index, ***eidx****, and an additonal string* ***pname****.*

Table 2: GAMMA Error Messages

Index	Message	Index	Message
1	Problems With File pname	**4**	Use Of Deprecated Function pname
2	Cannot Read Parameter pname	**5**	Please Use Class Member Function panme
3	Problems With Parameter Set	**x**	Unknown Error - pname

4 Isotope

4.1 Overview

The class ***Isotope*** defines the basic physical attributes of a spin isotope. These essential quantities are the isotope symbol, its associated spin angular momentum and gyromagnetic ratio. Functions are provided for simplified access to many isotope quantities as well as construction of new isotopes. Note that typical GAMMA programs will utilize spin systems having a set of such isotopes rather than the individual isotopes defined by this class.

4.2 Isotope Functions

Algebraic Operators

Basic Functions

I/O Functions

Existence And Addition

4.3 Algebraic Operators

4.3.1 Isotope

Usage:

#include <Basics/Isotope.h>
void Isotope::Isotope(const Isotope& data)
void Isotope::Isotope(string symbol)
void Isotope::Isotope(const Isotope &I)

Description:

The function *Isotope* is used to create a new isotope.

1. Isotope() -Without arguments the function creates a default isotope (a proton.)
2. Isotope(string) - Called with a string, the function will create a Isotope of the size indicated. By default, all Isotopes are set to be protons (spin I = 1/2). All other Isotope parameters are left unassigned although the appropriate array space for storage of all Isotope parameters will be allocated
3. Isotope(const Isotope &I) - Makes an Isotope which is a copy of the given Isotope I.

Return Value:

Isotope returns no parameters. It is used strictly to create a Isotope.

Examples:

```
#include <gamma.h>
int main()
  {
  Isotope I;                    // A default Isotope, I, which will be a proton.
  Isotope N14(”14N);            // An Isotope of 14N called N14.
  Isotope N(N14);               // An Isotope called N which is a duplicate of N14
  }
```

See Also: =

4.3.2 =

Usage:

#include <Basics/Isotope.h>
void Isotope::operator = (const Isotope& I)

Description:

The unary operator = (the assignment operator) is allows for the setting of one Isotope to another Isotope. If the Isotope exists it will be overwritten by the assigned Isotope.

Return Value: Void

Examples:

```
#include <gamma.h>
int main()
  {
```

```
Isotope I(string("13C"));          // Set Isotope I as a carbon 13.
Isotope I1;                        // Set a second Isotope default (1H)
I1 = I;                            // Now I1 is a carbon 13
}
```

See Also: Isotope, read

4.3.3 ==

Usage:

```
#include <Basics/Isotope.h>
bool Isotope::operator == (const Isotope& I) const
```

Description:

The unary operator == (the equality operator) will test two Isotopes for their equality. If the two Isotopes are identical the function returns true. If the two isotopes are not equal the function returns false.

Return Value: bool

Example:

```
#include <gamma.h>
int main()
  {
  Isotope H;                       // Define a default Isotope called H.
  Isotope C("13C");                // Define Isotope C containing carbon-13.
  if(H == C)                       // See if they the same isotope (they are not).
    cout << "The two Isotopes are identical";
  }
```

See Also: !=

4.3.4 !=

Usage:

```
#include <Basics/Isotope.h>
bool Isotope::operator != (const Isotope& I) const
```

Description:

The unary operator != (the inequality operator) can be used to test whether two Isotopes are equal. The function returns TRUE if the two Isotopes being compared are not equal and FALSE if they are identical.

Return Value: bool

Example:

```
#include <gamma.h>
int main()
  {
  Isotope I1("1H"), I2("13C");     // Define two Isotopes, I1 and I2.
  if(I1 != I2)                     // see if they are not the same.
    cout << "The two Isotopes are different";
  }
```

See Also: ==

4.4 Basic Functions

4.4.1 qn

Usage:

#include <Basics/Isotope.h>
double Isotope::qn () const

Description:

The function *qn* is used to obtain the quantum number I (spin angular momentum) carried by an Isotope. These are output in units of hbar, the default (1H) being 0.5.

Return Value: Double

Example:

```
#include <gamma.h>
int main()
  {
  Isotope X("131Xe");                  //  Isotope of 131Xe.
  cout << X.qn ();                     // Prints I value,1.5, of 131Xe.
  }
```

See Also: momentum, isotope

4.4.2 HS

Usage:

#include <Basics/Isotope.h>
int Isotope::HS ( ) const

Description:

The function *HS* is used to access the dimension of the spin Hilbert space associated with the individual Isotope. The spin Hilbert space size is related to the spin quantum number as,

$$HS = 2I+1$$

where I is the associated spin quantum number.

Return Value: Integer

Example:

```
#include <gamma.h>
int main()
  {
  int i;
  Isotope X("13C");                    // Define Isotope of 13C.
  cout << X.HS();                      // Output Hilbert Space size (2 for I=1/2);
  }
```

See Also:

4.4.3 momentum

Usage:

#include <Basics/Isotope.h>
string Isotope::momentum ( ) const

Description:

The function *momentum* is used to obtain, in string format, the spin angular momentum carried by an *Isotope*. This function is used for formatted output.

Return Value: string

Example:

```
#include <gamma.h>
int main()
  {
  Isotope T("3H");                  // Make Isotope T (tritium).
  cout << T.momentum();             // Prints momentum 1/2 of Isotope T.
  }
```

See Also: qn, isotope

4.4.4 symbol

Usage:

#include <Basics/Isotope.h>
const string Isotope::symbol( ) const

Description:

The function *symbol* is used to obtain, in string format, the isotope type. This function is typically used for formatted output.

Return Value: string

Example:

```
#include <gamma.h>
int main()
  {
  Isotope S("23Na");                // Define Isotope of sodium.
  cout << S.symbol(2);              // Prints isotope type 23Na.
  }
```

See Also: isotope, element

4.4.5 name

Usage:

#include <Basics/Isotope.h>
const string& Isotope::name() const

Description:

name is used to specify or retrieve a Isotope name.

Return Value: string

Example):

```
#include <gamma.h>
int main()
  {
  Isotope H("3H");                    // Set isotope of tritium
  cout << H.name();                   // Output "Hydrogen" to screen
  }
```

See Also:

4.4.6 element

Usage:

#include <Basics/Isotope.h>

const string Isotope::element ( ) const

Description:

The function *element* is used to obtain the name of the element assigned to a spin via the function *isotope*. It does not alter the Isotope and will return a blank if the spin has not previously been assigned. The function is commonly used for formatted output.

Return: Pointer to a string containing element label.

Example:

```
#include <gamma.h>
int main()
  {
  Isotope I("51V");                   // Set Isotope I to Vanadium 51.
  cout << I.element();                // Prints V to standard output.
  }
```

See Also: isotope, symbol.

4.4.7 number

Usage:

#include <Basics/Isotope.h>

int Isotope::number ( ) const

Description:

The function *number* is used to obtain the atomic number carried by an *Isotope*. This function is used for formatted output.

Return Value: integer

Example:

```
#include <gamma.h>
```

```
int main()
  {
  Isotope I(string("51V");              // Set Isotope I to Vanadium 51.
  cout << I.number();                    // Prints 51 to standard output.
  }
```

See Also: qn, isotope

4.4.8 mass

Usage:

```
#include <Basics/Isotope.h>
int Isotope::mass( ) const
```

Description:

The function *mass* is used to obtain the atomic mass carried by an *Isotope*. This function is used for formatted output.

Return Value: integer.

Example:

```
#include <gamma.h>
int main()
  {
  Isotope X(131Xe);                     // Define Isotope of Xenon.
  cout << X.mass();                     // Prints mass of 131Xe in amu.
  }
```

See Also: qn, isotope

4.4.9 weight

Usage:

```
#include <Basics/Isotope.h>
double Isotope::weight ( ) const
```

Description:

The function *weight* is used to obtain the atomic weight carried by an *Isotope*.

Return Value: double

Example:

```
#include <gamma.h>
int main()
  {
  Isotope X(129Xe);                     // Define Isotope of Xenon.
  cout << X.weight();                   // Prints weight of 129Xe in g/mole.
  }
```

See Also: qn, isotope

4.4.10 receptivity

Usage:

#include <Basics/Isotope.h>
double Isotope::receptivity ( ) const

Description:

The function *number* is used to obtain, in string format, the spin angular momentum carried by an *Isotope*. This function is used for formatted output.

Return Value: double

Example:

```
#include <gamma.h>
int main()
  {
  Isotope B("10B");                    // Define Isotope of boron.
  cout << B.receptivity();             // Output boron receptivity.
  }
```

See Also: qn, isotope

4.4.11 relative_frequency

Usage:

#include <Basics/Isotope.h>
double Isotope::relative frequency( ) const

Description:

The function *number* is used to obtain, in string format, the spin angular momentum carried by an *Isotope*. This function is used for formatted output.

Return Value: double

Example:

```
#include <gamma.h>
int main()
  {
  Isotope Ne("21Ne");                  // Define Isotope Ne of Neon-21.
  cout << Ne.relative_frequency();     // Prints relative frequency of Neon-21.
  }
```

See Also: qn, isotope

4.4.12 gamma

Usage:

#include <Basics/Isotope.h>
double Isotope::gamma () const

Description:

The function *gamma* will return a value for the gyromagnetic ratio of an *isotope*. Gamma values are used to set relative frequencies and in some relaxation computations. In GAMMA, gyromagnetic ratios are maintained in SI units, rad T^{-1} sec^{-1}. For example, the returned value for ^{19}F is 2.51719e+08 and that of a proton 2.67519e+08.

Return Value: Double

Example:

```
#include <gamma.h>
int main()
  {
  Isotope H("1H");                    // Define Isotope H containing a proton.
  cout << H.gamma();                  // Prints gyromagnetic ratio spin 2, 2.67519e+08.
  }
```

See Also: isotope.

4.4.13 electron

Usage:

```
#include <Basics/Isotope.h>
bool Isotope::electron () const
```

Description:

The function *electron* will return *true* if the isotope is an electron and *false* if it is a nucleus.

Return Value: bool

Example:

```
#include <gamma.h>
int main()
  {
  Isotope H("1H");                    // Define Isotope H as a proton
  Isotope E("e-");                    // Define Isotope E as an electron.
  if (H.electron())  cout << "\n\tH is an electron!!!!";
  else               cout << "\n\tH is a nucleus.";
  if(!E.electron())  cout << "\n\tE is a nucleus!!!!";
  else               cout << "\n\tE is an electron.";
  }
```

See Also: isotope.

4.5 I/O Functions

4.5.1 write

Usage:

```
#include <Basics/Isotope.h>
void Isotope::write(const string& filename);
```

Description:

The function *write* is enables one to specifically write the Isotope out to a readable ASCII file. Isotope parameters are written in the GAMMA parameter format and readable by the function *read*.

Return Value: Void

Example:

```
#include <gamma.h>
int main()
  {
  Isotope I(19F);                    // Isotope of fluorine.
  sys.write("test.sys");             // Write information of Isotope to file test.sys.
  }
```

See Also: read

4.5.2 read

Usage:

#include <Basics/Isotope.h>

void Isotope::read (const string& filename);

Description:

The function *read* enables one to read in the Isotope from an external file. The file is assumed to be written in the standard parameter set format (see the Section Parameter Files later in this Chapter) utilizing the standard parameter names associated with a Isotope. Typically, the file being read was generated from a previous simulation through the use of the analogous Isotope function *write*. Alternatively, one can use an editor to construct a parameter set file for the Isotope.

Return Value: void

Example(s):

```
#include <gamma.h>
int main()
  {
  Isotope I;                         // An empty isotope
  I.read("test.sys");                // Read in the Isotope from file test.sys.
  }
```

See Also: write

4.5.3 print

Usage:

#include <Basics/Isotope.h>

ostream& Isotope::print(ostream&) const

Description:

The function *print* is used to print out all the current information stored in a Isotope object. This includes the number of Isotopes and spin isotope types.

Return Value:

None.

Example:

```
#include <gamma.h>
int main()
  {
  Isotope Mg("25Mg");                  // Define Isotope Mg containing a magnesium-25.
  Mg.print( );                         // Print out information on magnesium-25.
  }
```

4.5.4 <<

Usage:

#include <Basics/Isotope.h>
ostream& operator<< (ostream&, Isotope&)

Description:

The operator << is used for standard output of a Isotope. This includes the number of Isotopes and spin isotope types.

Return Value:

None.

Example:

```
#include <gamma.h>
int main()
  {
  Isotope Mg("25Mg");                  // Define Isotope Mg containing a magnesium-25.
  cout <<Mg;                           // Write the Isotope magnesium-25 to standard output.
  }
```

See Also: write, read, print

4.6 Existence And Addition

0.0.1 seek

0.0.2 exists

0.0.3 known

Usage:

#include <Basics/Isotope.h>
int Isotope::seek(const IsotopeData& ID) const
bool Isotope::exists(const string& ID) const
static int Isotope::known(const string& ID) const

Description:

The functions *seek*, *exists*, and *known* all check to see if a particular isotope is known to GAMMA. The isotope sought is indicated either by *ID*, either isotope data or a string for the isotope symbol. The *seek* function will return the index of the isotope in the isotope list, a value of *-1* is returned if the isotope is not found. Both *exists* and *know* will return true if the isotope is known to GAMMA or false if not. Note that known is identical to exists but does not require an instance of class Isotope for use.

Return Value: bool or integer

Example:

```
#include <gamma.h>
int main()
  {
  string S("25Mg");
  if(Isotope::known(S))
    cout << "\n\tGAMMA knows about " << S;
  Isotope MG(S);
  if(MG.exists("37Mg"))
    cout << "\n\tGAMMA knows nothing about 37Mg";
  }
```

4.6.1 AddIsotope

Usage:

#include <Basics/Isotope.h>

bool Isotope::AddIsotope(const IsotopeData& ID, int warn=2) const

Description:

The functions *AddIsotope* will add the isotope *ID* into GAMMA. The the function will return true if the isotope has been successfully added or false if not. The flag warn indicates how to handle failures: 0=do nothing, 1=issue warnings, >1= issue warnings, stop execution.

Return Value: bool

Example:

```
#include <gamma.h>
int main()
  {
  Isotope Iele("e-");                          /// Get e- (use some of its values)
  int hs        = 6;                           // For I=5/2
  string symb   = "Ce3+";                      // Isotope Symbol
  string name   = "Cerium";                    // Isotope Name
  string ele    = "Ce";                        // Element Designation
  int   atnum   = 58;                          // Atomic Number
  int   mass    = 140;                         // Atomic Mass
  double weight  = Iele.weight();              // Atomic weight (g/m)
  double recept  = Iele.receptivity();         // Receptivity
  double relfreq = Iele.relative_frequency();  // Relative frequency
  bool   elect   = true;                        // Its an electron
  IsotopeData ID(hs,symb,name,ele,atnum,mass,weight,recept,relfreq,elect);
  Isotope::AddIsotope(ID);                     // Add isotope to GAMMA
  Isotope I("Ce3+");                           // Here is a Ce3+ isotope
  cout << "\n\t\tNew Electron Spin:\n" << I;}
```

4.7 Description

An Isotope is a nucleus (or electron) which has spin angular momentum. Each object of class **Isotope** has an intrinsic **spin angular momentum** (I). This in turn sets the dimension of an associated spin **Hilbert space** (2I+1) which can be used in quantum mechanical calculations. Internally, an isotope is simply an index in a static list of isotopes maintained in GAMMA.

Class Isotope Components

Internal Name	Type	Usage
Isotopes	static vector<IsotopeData>*	Pointer to vector of Isotopes (Isotope List)
NIso	static int	Number of isotopes known in Γ[a]
RELFRQ1H	static double	Base Proton Relative Frequency
Iso	int	Index of isotope in vector Isotopes

a. The total number of isotopes here is internally set in GAMMA when the Isotopes list is initialized. Addition of isotopes will not change this value, NIso, even though the number in the list will increase.

Figure 0-1 **Contents of each Isotope variable. The Isotope list is shared by all Isotope variables. The index is where the isotope is stored in the Isotope list, the item being type IsotopeData.**

Thus, each Isotope is an integer index into a list of Isotopes that all Isotopes know of, and share. Isotopes of the same type point to the same list element, so they are absolutely identical. When a variable of type Isotope is declared in a GAMMA program, class Isotope automatically checks the static Isotope list to see if it exists. If found, the information regarding the isotope is assigned to it from the static list. In most other functions, the Isotope uses it's list element to retrieve information.

Class Isotope Functioning

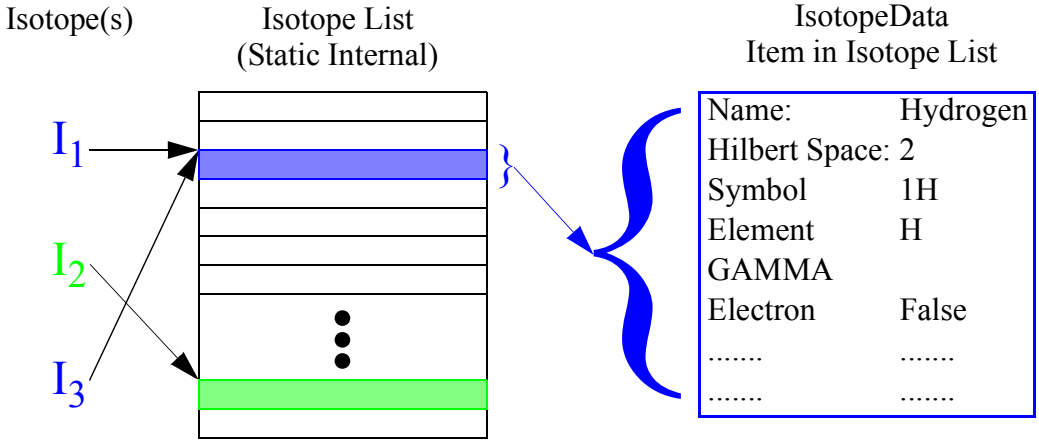


Figure 0-2 **Each Isotope has a Pix which points to an item in an internal list shared by all variables of type Isotope. Each list item is of type IsotopeData. In turn Isotope data contains the information specific to a particular isotope and functions which access this information. Many Isotopes may contain the same Pix, i.e. they may be the same isotope type.**

4.8 List of Available Isotopes

Available Isotopes in GAMMA

	Element	I		Element	I		Element	I
1H	Hydrogen	1/2	2H	Deuterium	1	3H	Tritium	1/2
3He	Helium	1/2	6Li	Lithium	1	7Li	Lithium	3/2
9Be	Beryllium	3/2	10B	Boron	3	11B	Boron	3/2
13C	Carbon	1/2	14N	Nitrogen	1	15N	Nitrogen	1/2
17O	Oxygen	5/2	19F	Fluorine	1/2	21Ne	Neon	3/2
23Na	Sodium	3/2	25Mg	Magnesium	5/2	27Al	Aluminum	5/2
29Si	Silicon	1/2	31P	Phosphorus	1/2	33S	Sulfur	3/2
35Cl	Chlorine	3/2	37Cl	Chlorine	3/2	39K	Potassium	3/2
41K	Potassium	3/2	43Ca	Calcium	7/2	45Sc	Scandium	7/2
47Ti	Titanium	5/2	49Ti	Titanium	7/2	50V	Vanadium	6
51V	Vanadium	7/2	53Cr	Chromium	3/2	55Mn	Manganese	5/2
57Fe	Iron	1/2	59Co	Cobalt	7/2	61Ni	Nickel	3/2
63Cu	Copper	3/2	65Cu	Copper	3/2	67Zn	Zinc	5/2
69Ga	Gallium	3/2	71Ga	Gallium	3/2	73Ge	Germanium	9/2
75As	Arsenic	3/2	77Se	Selenium	1/2	79Br	Bromine	3/2
81Br	Bromine	3/2	85Rb	Rubidium	5/2	87Rb	Rubidium	3/2
87Sr	Strontium	9/2	89Y	Yttrium	1/2	91Zr	Zirconium	5/2
93Nb	Niobium	9/2	95Mo	Molybdenum	5/2	97Mo	Molybdenum	5/2
99Tc	Technetium	9/2	99Ru	Ruthenium	5/2	101Ru	Ruthenium	5/2
103Rh	Rhodium	1/2	105Pd	Palladium	5/2	107	Silver	1/2
109Ag	Silver	1/2	111Cd	Cadmium	1/2	113Cd	Cadmium	1/2
113In	Indium	9/2	115In	Indium	9/2	117Sn	Tin	1/2
119Sn	Tin	1/2	121Sb	Antimony	5/2	123Sb	Antimony	7/2
123Te	Tellurium	1/2	125Te	Tellurium	1/2	127I	Iodine	5/2
129Xe	Xenon	1/2	131Xe	Xenon	3/2	133Cs	Cesium	7/2
135Ba	Barium	3/2	137Ba	Barium	3/2	138La	Lanthanum	5
139La	Lanthanum	7/2	141Pr	Praseodymium	5/2	143Nd	Neodymium	7/2
145Nd	Neodymium	7/2	147	Samarium	7/2	149Sm	Samarium	7/2
151Eu	Europium	5/2	153Eu	Europium	5/2	155Gd	Gadolinium	3/2
157Gd	Gadolinium	3/2	159Tb	Terbium	3/2	161Dy	Dysprosium	5/2
163Dy	Dysprosium	5/2	165Ho	Holmium	7/2	167Er	Erbium	7/2
169Tm	Thulium	1/2	171Yb	Ytterbium	1/2	173Yb	Ytterbium	5/2
175Lu	Lutetium	7/2	176Lu	Lutetium	7	177Hf	Hafnium	7/2
179Hf	Hafnium	9/2	181Ta	Tantalum	7/2	183W	Tungsten	1/2
185Re	Rhenium	5/2	187Re	Rhenium	5/2	187Os	Osmium	1/2
189Os	Osmium	3/2	191Ir	Iridium	3/2	193Ir	Iridium	3/2
195Pt	Platinum	1/2	197Au	Gold	3/2	199Hg	Mercury	1/2
201Hg	Mercury	3/2	203Tl	Thallium	1/2	205Tl	Thallium	1/2
207Pb	Lead	1/2	209Bi	Bismuth	9/2	235U	Uranium	7/2
e-	Electron	1/2						

5 Single Parameters

5.1 Overview

The class ***SinglePar*** defines a single "GAMMA" parameter. It is intended to facilitate importing/exporting parameters from/to ASCII files. This class forms the entry type of GAMMA parameter sets - lists of parameters that can be imported from/exported to ASCII files (see ParameterSet.) Direct I/O of single parameters should be performed though class ***ParameterSet***. Thus, objects of type ***SinglePar*** are normally found in GAMMA programs only when used in conjunction with objects of type ***ParameterSet***.

5.2 Available Functions

Constructors

Access Functions

Input

Output

5.3 Constructors

5.3.1 SinglePar

Usage:

#include <Basics/SinglePar.h>
void SinglePar::SinglePar()
void SinglePar::SinglePar(SinglePar &par)
void SinglePar::SinglePar(const string& pname, int ptype, const string& pdata, const string& pstate)
void SinglePar::SinglePar(const string& pname, const string& pdata, const string& pstate)
void SinglePar::SinglePar(const string& pname, int pdata, const string& pstate)
void SinglePar::SinglePar(const string& pname, double pdata, const string& pstate)

Description:

The function *SinglePar* is used to create a new parameter.

1. ***SinglePar()*** - Called without arguments the function creates an "empty" NULL parameter.
2. ***SinglePar(par)*** - Called with another parameter, an identical parameter is created
3. ***SinglePar(pname, ptype, pdata, pstate)*** - Creates a parameter having name ***pname*** of type ***ptype***. The data in string ***pdata*** will be parsed according the specified type ***ptype***. The parameter will also maintain a comment ***pstate***.
4. ***SinglePar(pname, pdata, pstate)*** - Creates a parameter having name ***pname***, data ***pdata***, and comment ***pstate***. The constructor is overloaded so that the data can either be an integer, double, or string and the parameter type set accordingly.

Return Value:

SinglePar returns no parameters. It is used strictly to create a SinglePar.

Examples:

```
#include <gamma.h>
int main()
  {
  SinglePar A;                              // Define all NULL parameter called A.
  SinglePar p1("p1", 0, "1,", "one");       // Create an integer parameter p1
  SinglePar p2("p2", 1, "1.0", "one double");// A double parameters
  SinglePar p3("p3", 2, "one", "one string");// A string parameter
  SinglePar p4(p3);                         // Identical to p3
  SinglePar p1i("p1", 1, "one");            // Same as earlier p1
  SinglePar p2d("p2", 1.0, "one double");   // Same as earlier p2
  Singlepar p3d("p3", "one", "one string);  // Same as earlier p3
  }
```

See Also: =, read

5.3.2 =

Usage:

#include <Basics/SinglePar.h>
void SinglePar::operator = (SinglePar &par)

Description:

The unary ***operator*** = (the assignment operator) is allows for the setting of one SinglePar to another SinglePar. If the parameter exists it will be overwritten by the assigned SinglePar.

Return Value: None

Examples:

```
#include <gamma.h>
int main()
  {
  SinglePar A;                    // Define all NULL parameter called A.
  SinglePar A2BX(4);              // Define parameter A2BX containing four spins.
  A = A2BX;                       // Set parameter A to be equal to current Four.
  }
```

See Also: SinglePar, read

5.4 Access Functions

5.4.1 name

Usage:

```
#include <Basics/SinglePar.h>
string SinglePar::name( )
void SinglePar::name(const string& Name)
```

Description:

The function ***name*** either sets or returns the parameter name.

Return Value: ***void or string***

Example:

```
#include <gamma.h>
int main()
  {
  SinglePar xyz;                          // An empty parameter
  xyz.name("XYZ");                        // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name();   // Print the parameter name.
  }
```

See Also: type, data, state

5.5 Input

5.5.1 read

Usage:

#include <Basics/SinglePar.h>
void SinglePar::read (ifstream& inp);

Description:

The function ***read*** enables one to read in the SinglePar from an input filestream. The input is assumed to be written in the standard parameter set format. Typically this is only used by class ParameterSet.

Return Value: None.

Example:

```
#include <gamma.h>
int main()
  {
  SinglePar par;                    // Define an empty SinglePar.
  par.read("test.par");             // Read in the SinglePar from file test.par.
  }
```

5.6 Output

5.6.1 print

Usage:

#include <Basics/SinglePar.h>
ostream& SinglePar::print(ostream&) const

Description:

The function *print* is used to print out all the current information stored in a parameter object.

Return Value: ostream

Example:

```
#include <gamma.h>
int main()
  {
  SinglePar par("test", 27.8, "try this");      // Make some double parameter
  par.print(cout );                             // Print out information in parameter
  }
```

5.6.2 <<

Usage:

#include <Basics/SinglePar.h>
ostream& operator<< (ostream&, SinglePar&)

Description:

The operator << is used for standard output of a parameter.

Return Value: ostream

Example(s):

```
#include <gamma.h>
int main()
  {
  CH3.SinglePar(3);                             // define parameter CH3 containing three spins.
  cout << CH3;                                  // write the parameter CH3 to standard output.
  }
```

See Also: write, read, print

5.6.3 write

Usage:

#include <Basics/SinglePar.h>
void SinglePar::write(ofstream& ofstr, int namelen=10) const;

Description:

The function ***write*** is enables one to specifically write the SinglePar out to an ASCII output file stream, ***ostr***. Parameters are written in the standard parameter set format and readable by the function *read*. The value ***namelen*** sets the minimum length of the column the parameter name will be written in.

Return Value:

None.

Example:

```
#include <gamma.h>
int main()
  {
  SinglePar par(3);                    // define SinglePar par containing three spins.
  par.write("test.par");               // write information in SinglePar par to file test.par.
  }
```

See Also: read

5.7 Description

The class ***SinglePar*** defines a single "GAMMA" parameter, meant to facilitate importing/exporting parameters from/to ASCII files. However, individual parameter I/O is only ***rarely*** performed. Rather, multiple parameters are read/written simultaneously through use of parameter sets. This class forms the entry type of GAMMA parameter sets - lists of parameters that can be imported from/exported to ASCII files (see ParameterSet.)

5.7.1 Internal Structure

Every object of type ***SinglePar*** contains a 1.) **Name (string)**, 2.) **Data (string)**, 3.) **Type (int)**, and 4.) **Comment (string)**. The internal structure of class ***SinglePar*** contains the quantities listed in the following table (names shown are also internal).

Table 3: : Internal Structure of Class SinglePar

Name	Description	Type
ParName	Name of the Parameter	string
ParType	Parameter Type	int
ParData	Parameter Value(s)	string
ParState	Comment Describing Parameter	string

Fig. 0-3 **Depiction of class Single contents, i.e. what each GAMMA defined single parameter contains. The integer ParType is used to indicate which type of value is associated with the parameter. These are currently: 0=integer, 1=float, 2=string, 3=coordinate, 4=tensor.**

The data contains the parameter value(s) and is stored in string format so that multiple data types are supported. How the data (as a string) is parsed depends upon the parameter type specified.

5.7.2 External Structure

A single parameter, **SinglePar**, must assume a particular format when written in an ASCII file. The parameter will exists on a single line (unless the data spans multiple lines) and contain four quantities as follows

PName (PType) : PValue - PStatement

Rules governing how this exists in the file are

1 The parameter name is 1st in the line, followed by the type, value, and statement (if any).

2 There are no columns, i.e. the 4 quantities may begin in any column that adheres to rule 1.

3 There must be at least 1 space or tab separating the four quantities.

4 The parameter name, PName, cannot have any spaces, tabs, parentheses, or hyphens in it.

5 The parameter type, PType, must be a single number surrounded by parentheses, no spaces.

6 A colon with at least one blank or tab on each side must reside between (PType) & PValue.

7 The parameter statment is optional. If present it must be preceded by a hyphen surrounded by at least one blank or tab on each side.

5.7.3 Single Parameter Examples

Each line in the table below represents a line in an ASCII file that defines a single parameter. Note that users may create parameters with any name, value, and statement. Any ASCII file editor can be used to place GAMMA parameters in a file. The ones shown below are simply representative of parameters commonly used in GAMMA simulations.

Name	(Type)	:	Value	-	Statement
Nspins	(0)	:	5	-	Number of Spins
Iso(0)	(2)	:	19F	-	Spin Isotope Type
Iso(3)	(2)	:	3H	-	Spin Isotope Type
J(0,1)	(1)	:	7.839	-	Coupling constant (Hz)
Omega	(1)	:	500.0	-	Proton frequency (MHz)
Coord(7)	(3)	:	(0.0, 1.2, -4.56)	-	Position of spin 7 (A)
Iso(3)	(2)	:	3H	-	Spin Isotope Type

Keep in mind that column alignment is not required nor is the statment ("- some words") necessary. Multiple parameters may be placed into the same ASCII file, and any text not in parameter format may coexist in the file. Row position in the file is on no consequence either unless SinglePar, rather than class ParameterSet, is being used to read the parameter from the file.

5.7.4 Parameter Input Example

As mentioned, ***SinglePar*** does not provide the best method of importing an object of type ***SinglePar*** from an external file. Class ***ParameterSet*** is superior in this regard. As such, this example will rely on the latter and not even use any objects of type ***SinglePar***. Consider an ASCII file named ***SinglePar.asc*** which contains the line

```
J(0,1)  (1)  :  7.839  - Coupling constant (Hz)
```

The line may be the only line in the file, or it may coexist with other text and other parameters. It can be in any row position. To read this parameter into a GAMMA program one might use the following code.

```
#include <gamma.h>
int main()
  {
  ParameterSet pset;                    // An empty parameter set.
  pset.read(“SinglePar.asc”);           // Read all parameters in file SinglePar.asc into pset
  string pname("J(0,1)");               // Name of parameter we want
  double J01;                           // Parameter value we want
  pset. getDouble(pname, J01);          // Set J01 to the double value of parameter J(0,1)
  }
```

As the reader might construe, if the external file has multiple parameters all of them will be read into the program. The code must simply request the desired parameter values from the parameter set. Furthermore, the "program" above can be readily generalized so that the user specifies the input ASCII file interactively. See the documentation on class ParameterSet for further details.

5.7.5 Parameter Output Example

As mentioned, ***SinglePar*** does not provide the best method of exporting an object of type ***SinglePar*** to an external file. Again, class ***ParameterSet*** is superior in this regard. However, the code below will write a single parameter into a specified external file.

```
#include <gamma.h>
int main()
  {
  SinglePar par("MyPar", 167, "Int Val");   // A single integer parameter with value 167.
  ofstream ofstr("MyFile.asc");             // Open a file named Myfile.asc for output
  pset.write(ofstr);                        // Write the parameter into the file
  ofstr.close();                            // Close the output file
  }
```

Obviously, one may write multiple parameters into the output file before it is closed. See the documentation on class ParameterSet for the best way to accomplish this.

6 Parameter Sets

6.1 Overview

The class ***Parameter.Set*** defines a set of GAMMA parameters. The class is intended to facilitate importing/exporting parameters from/to ASCII files. Each member of the parameter set is a single parameter (see SinglePar.)

6.2 Available Functions

Base Functions (Inherited)

Constructors

Auxiliary Functions

Access Functions

Input

Output

Interactive Functions

6.3 Base Functions (Inherited)

These are functions which are inherited through the base class of ***Parameter Set***, ***list***. The class ***list*** in now a defined type in ANSI C++ and should be part of any ANSI compatible compiler. See the C++ documentation of the class for a full function listing. Note that the iterator in the ***Parameter-Set*** list is an object of type ***SinglePar*** (single parameter) as defined in that class.

6.3.1 begin
6.3.2 end
6.3.3 front
6.3.4 back

Description:

These functions are used to traverse through the parameter set using iterators.

Return Value:

These either return an interator in the parameter set list, or they return a single parameter.

Examples:

```
#include <gamma.h>
int main()
  {
  }
```

See Also: none

6.3.5 push_back
6.3.6 pop_back
6.3.7 push_front
6.3.8 pop_front
6.3.9 insert
6.3.10 erase
6.3.11 clear

Description:

These functions are used to traverse through the parameter set using iterators.

Return Value:

These either return an interator in the parameter set list, or they return a single parameter.

Examples:

```
#include <gamma.h>
int main()
  {
  }
```

See Also: none

6.3.12 size

6.3.13 empty

6.3.14 ==

6.3.15 !=

Description:

These functions are used to traverse through the parameter set using iterators.

Return Value:

These either return an interator in the parameter set list, or they return a single parameter.

Examples:

```
#include <gamma.h>
int main()
  {
  }
```

See Also: none

6.4 Constructors

6.4.1 ParameterSet

Usage:

```
#include <Basics/ParamSet.h>
void ParameterSet::ParameterSet()
void ParameterSet::ParameterSet(ParameterSet &par)
void ParameterSet::ParameterSet(const string& pname, int ptype, const string& pdata, const string& pstate)
void ParameterSet::ParameterSet(const string& pname, const string& pdata, const string& pstate)
void ParameterSet::ParameterSet(const string& pname, int pdata, const string& pstate)
void ParameterSet::ParameterSet(const string& pname, double pdata, const string& pstate)
```

Description:

The function *ParameterSet* is used to create a new parameter.

1. ***ParameterSet()*** - Called without arguments the function creates an "empty" NULL parameter.
2. ***ParameterSet(par)*** - Called with another parameter, an identical parameter is created
3. ***ParameterSet(pname, ptype, pdata, pstate)*** - Creates a parameter having name ***pname*** of type ***ptype***. The data in string ***pdata*** will be parsed according the the specified type ***ptype***. The parameter will also maintain a comment ***pstate***.
4. ***ParameterSet(pname, pdata, pstate)*** - Creates a parameter having name ***pname***, data ***pdata***, and comment ***pstate***. The constructor is overloaded so that the data can either be an integer, double, or string and the parameter type set accordingly.

Return Value:

ParameterSet returns no parameters. It is used strictly to create a ParameterSet.

Examples:

```
#include <gamma.h>
int main()
  {
  ParameterSet A;                          // Define all NULL parameter called A.
  ParameterSet p1("p1", 0, "1,", "one");   // Create an integer parameter p1
  ParameterSet p2("p2", 1, "1.0", "one double");// A double parameters
  ParameterSet p3("p3", 2, "one", "one string");// A string parameter
  ParameterSet p4(p3);                     // Identical to p3
  ParameterSet p1i("p1", 1, "one");        // Same as earlier p1
  ParameterSet p2d("p2", 1.0, "one double");// Same as earlier p2
  ParameterSet p3d("p3", "one", "one string);// Same as earlier p3
  }
```

See Also: =, read

6.4.2 =

Usage:

#include <Basics/ParamSet.h>
void ParameterSet::operator = (ParameterSet &par)

Description:

The unary ***operator*** = (the assignment operator) is allows for the setting of one ParameterSet to another ParameterSet. If the parameter exists it will be overwritten by the assigned ParameterSet.

Return Value:

None, the function is void

Examples:

```
#include <gamma.h>
int main()
  {
  ParameterSet A;                 // Define all NULL parameter called A.
  ParameterSet A2BX(4);           // Define parameter A2BX containing four spins.
  A = A2BX;                       // Set parameter A to be equal to current Four.
  }
```

See Also: ParameterSet, read

6.5 Auxiliary Functions

6.5.1 contains

Usage:

#include <Basics/ParamSet.h>

int ParameterSet::contains(const string& pname) const

int ParameterSet::contains(const SinglePar& par) const

Description:

The function ***contains*** returns 0 or 1 depending on whether it contains a parameter with the name ***pname*** or the input parameter ***par***.

Return Value:

int

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet xyz;                    // An empty parameter
  xyz.name("XYZ");                     // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name(); // Print the parameter name.
  }
```

See Also: type, data, state

6.5.2 seek

Usage:

#include <Basics/ParamSet.h>

list<SinglePar>::const_iterator ParameterSet::seek(const string& pname) const

list<SinglePar>::const_iterator ParameterSet::contains(const SinglePar& par) const

Description:

The function ***seek*** returns an interator into the list if it contains a parameter with the name ***pname*** or the input parameter ***par***. The iterator will be the list ***end()*** if the parameter is not found.

Return Value: int

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet xyz;                    // An empty parameter
  xyz.name("XYZ");                     // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name(); // Print the parameter name.
  }
```

See Also: type, data, state

6.5.3 strip

Usage:

#include <Basics/ParamSet.h>

ParameterSet ParameterSet::strip(int indx) const

Description:

The function ***strip*** returns a parameter set which contains a sub-set of the paraemters in the original list. The subset will contain only those parameters whose names begin with ***[indx]***.

Return Value:

int

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet xyz;                       // An empty parameter
  xyz.name("XYZ");                        // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name();   // Print the parameter name.
  }
```

See Also: type, data, state

6.5.4 countpar

Usage:

#include <Basics/ParamSet.h>

int ParameterSet::countpar(const string& pname, idx0=0)

Description:

The function ***countpar*** returns of parameters in the set whose names are ***pname(#)***, beginning with ***pname(idx0)***, and continuously existing while the value is incremented by +1.

Return Value:

int

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet xyz;                       // An empty parameter
  xyz.name("XYZ");                        // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name();   // Print the parameter name.
  }
```

See Also: type, data, state

6.6 Access Functions

6.6.1 getInt

6.6.2 getDouble

6.6.3 getString

Usage:

#include <Basics/ParamSet.h>

int ParameterSet::getInt(const string& pname, int& value) const

int ParameterSet::getDouble(const string& pname, double& value) const

int ParameterSet::getString(const string& pname, string& value) const

Description:

The functions ***getInt, getDouble, and getString*** attempt to set the value ***value*** to that of parameter having name ***pname***. The function returns 0 or 1 depending on whether it contains a parameter with the name ***pname*** and successfully set the value ***value***.

Return Value:

int

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet xyz;                       // An empty parameter
  xyz.name("XYZ");                        // Set parameter name to XYZ
  cout << "\n\tName is " << xyz.name();   // Print the parameter name.
  }
```

See Also: none

6.7 Input

6.7.1 read

Usage:

#include <Basics/ParamSet.h>

int ParameterSet::read (const string& filein, int warn=0);

int ParameterSet::read (ifstream& inp, int warn=0);

Description:

The function ***read*** attempts to read in a ParameterSet from either a file named ***filein*** or from an input filestream ***inp***. The input is assumed to be written in the GAMMA standard parameter set format. The function returns 0 or 1 if it fails or succeeds. The value of ***warn*** set how the function should respond to a failure.

Return Value:

None.

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet par;                    // Define an empty ParameterSet.
  par.read("test.par");                // Read in the ParameterSet from file test.par.
  }
```

6.8 Output

6.8.1 print

Usage:

#include <Parameter.Set.h>
ostream& ParameterSet::print(ostream&) const

Description:

The function *print* is used to print out all the current information stored in a parameter set.

Return Value:

The output stream is returned.

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet par("test", 27.8, "try this"); // Make some double paramter
  par.print(cout );                           // Print out information in parameter
  }
```

6.8.2 <<

Usage:

#include <Basics/ParamSet.h>
ostream& operator<< (osteram&, ParameterSet&)

Description:

The operator << is used for standard output of a parameter.

Return Value:

None.

Example(s):

```
#include <gamma.h>
int main()
  {
  CH3.ParameterSet(3);                        // define parameter CH3 containing three spins.
  cout << CH3;                                // write the parameter CH3 to standard output.
  }
```

See Also: write, read, print

6.8.3 write

Usage:

```
#include <Basics/ParamSet.h>
void ParameterSet::write(ofstream& ofstr, int namelen=10) const;
```

Description:

The function ***write*** is enables one to specifically write the ParameterSet out to an ASCII output file stream, ***ostr***. Parameters are written in the standard parameter set format and readable by the function *read*. The value ***namelen*** sets the miminum length of the column the parameter name will be written in.

Return Value: None

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet par(3);              // define ParameterSet par containing three spins.
  par.write("test.par");            // write information in ParameterSet par to file test.par.
  }
```

See Also: read

6.9 Interactive Functions

6.9.1 ask_read

Usage:

#include <Basics/ParamSet.h>
void ParameterSet::write(ofstream& ofstr, int namelen=10) const;

Description:

The function ***write*** is enables one to specifically write the ParameterSet out to an ASCII output file stream, ***ostr***. Parameters are written in the standard parameter set format and readable by the function *read*. The value ***namelen*** sets the miminum length of the column the parameter name will be written in.

Return Value: None

Example:

```
#include <gamma.h>
int main()
  {
  ParameterSet par(3);              // define ParameterSet par containing three spins.
  par.write("test.par");            // write information in ParameterSet par to file test.par.
  }
```

See Also: read

6.10 Description

Every parameter of tyep ParameterSet contains a 1.) **Name (string)**, 2.) **Data (string)**, 3.) **Type (int)**, and 4.) **Comment (string)**. The data is stored in string format so that multiple data types are supported. How the data is parsed depends upon the parameter type.

6.11 Parameter Files

This section describes how an ASCII file may be constructed that is self readable by a ***ParameterSet***. The file can be created with an editor of the users choosing and is read with the ***ParameterSet*** member function “read”. Keep in mind that parameter ordering in the file is arbitrary. Some parameters are essential, some not. Also, other parameters are allowed in the file which do not relate to the parameter. See class ParameterSet for details.

parameter name: parName

A parameter name may be entered. It has no mathematical function in GAMMA but comes in handy when quickly scanning the input file or tagging some output file with the parameter. This parameter is optional and of type 2, a string parameter.

Number of spins: Nspins

It is essential that the number of spins be specified. A simple integer is input here. This is the first thing that will be read from the file so keep it somewhere near the top.

Isotope Types: Iso(i)

Spin isotopes are specified with this parameter. These are optional, spins for which there is no isotope specified will be set to protons. A complete lookup table of isotope information is internal to GAMMA and scanned upon parameter construction depending upon the isotope types. Isotopes are specified by the atomic number immediately followed (no blanks) by the atomic symbol.

parameter Parameters

Parameter	Assumed Units	Examples Parameter (Type) : Value - Statement
parName	none	parName (2) : Glycine - parameter Name
NSpins	none	Nspins (0) : 5 - Number of Spins
Iso(i)	none	Iso(0) (2) : 19F - Spin Isotope Type Iso(1) (2) : 3H - Spin Isotope Type Iso(3) (2) : 13C - Spin Isotope Type

Figure 0-4 **Parameter type 2 indicates a string parameter. Parameter type 0 indicates an integer parameter. Note that the first spin is spin zero and the last spin is spin N-1 in a parameter with N spins. The values need not be in any order in the file nor even following each other.**

By including these types of parameter statements (right column in previous table) in an ASCII file a GAMMA basic parameter can be set with the ***read*** function. For example, the code below reads “file.asc” parameters to set up a working parameter.

ASCII File Read To Set A Basic parameter

file.asc

```
parName     (3)  : HNC  - parameter name
NSpins      (0)  : 3    - Number of spins in the partem
Iso(0)      (2) : 2H    - Spin 0 set to deuteriumr
Iso(1)      (2) : 15N   - Spin 1 set to nitrogen 15
Iso(2)      (2) : 13C   - Spin 2 set to carbon 13
```

code.cc

```
ParameterSet par;
string filein("file.asc");
par.read(filein);.
```

Figure 0-5 **Specifying basic parameters using an external ASCII file.**

Remember, these ASCII files are read as GAMMA parameter set files so that they may contain additional lines of information and additional parameters. Things such as column spacing is not important - read about GAMMA parameters sets for full details.

Because ***ParameterSet*** contains very little information it is occasionally easier in a GAMMA program to simply code in all ***ParameterSet*** parameters directly. On the other hand, it is common that a user has many parameter files associated with more complex quantities (***ParameterSettem***, ***tensors***, etc.) which happens to contain ***ParameterSet*** information. The more complicated file can be used equally well, the needed values simply scanned amongst the other parameters present in the file. Below is another example ASCII file which contains only ParameterSet parameters.

Galactose

```
parName (2) : gal  - Name of the parameter (galactose)
NSpins  (0) : 8    - Number of Spins in the partem
Iso(0)  (2) : 1H   - Spin Isotope Type
Iso(1)  (2) : 1H   - Spin Isotope Type
Iso(2)  (2) : 1H   - Spin Isotope Type
Iso(3)  (2) : 1H   - Spin Isotope Type
Iso(4)  (2) : 1H   - Spin Isotope Type
Iso(5)  (2) : 1H   - Spin Isotope Type
Iso(6)  (2) : 1H   - Spin Isotope Type
Iso(7)  (2) : 1H   - Spin Isotope Type
```