MPP–2004–102
hep–ph/0408283

SUSY Les Houches Accord I/O made easy

T. Hahn
Max-Planck-Institut für Physik
Föhringer Ring 6, D–80805 Munich, Germany

August 25, 2004

Abstract

A library for reading and writing data in the SUSY Les Houches Accord format is presented. The implementation is in native Fortran 77. The data are contained in a single array conveniently indexed by preprocessor statements.

1 Introduction

The SUSY Les Houches Accord (SLHA) has standardized and significantly simplified the exchange of input and output parameters of SUSY models between such disparate applications as spectrum calculators and event generators.

While the SLHA specifications [1] include the precise formats for Fortran I/O, it is nevertheless not entirely straightforward to read or write a file in SLHA format. The present library provides the user with simple routines to read and write files in SLHA format, as well as a few utility routines. One thing the library does not do is modify the numbers, which means there is no routine to compute, say, a particular quantity at a new scale.

Sect. 2 describes the organization of the data structures, Sect. 3 gives the reference information for the library routines, Sect. 4 shows the usage in some examples, Sect. 5 contains download and build instructions, and Sect. 6 summarizes.

2 Data structures

The SLHA library is written in Fortran 77. All routines operate on a double-precision array, `slhadata`, which is about the simplest conceivable data format for this purpose in Fortran. For convenience of use, this array is accessed via preprocessor statements, so the user never needs to memorize any actual indices for the `slhadata` array. A file containing the preprocessor definitions must thus be included.

The `slhadata` array consists of a 'static' part containing the information from SLHA `BLOCK` sections and a 'dynamic' part containing the information from SLHA `DECAY` sections. The static part is indexed by preprocessor variables defined in `SLHA.h`, the dynamic part is accessed through the `SLHAGetDecay`, `SLHANewDecay`, and `SLHAAddDecay` functions and subroutines (see Sect. 3).

In addition, descriptive names for the PDG codes of the particles are declared in `PDG.h`. These are needed e.g. to access the decay information.

2.1 SLHA blocks

The explicit indexing of the `slhadata` need not (and should not) be done by the user. Rather, the members of the SLHA data structure are accessed through preprocessor variables. Tables 1, 2, 3, and 4 list the preprocessor variables defined in `SLHA.h` which follow closely the definition of the Accord [1]. Note that preprocessor symbols are case sensitive.

As far as there is overlap, the names for the block members have been chosen similar to the ones used in the MSSM model file of *FeynArts* [2]. The following index conventions are employed in the Tables:

$t = 1 \dots 4$ (s)fermion type:
- 1 = (s)neutrinos,
- 2 = isospin-down (s)leptons,
- 3 = isospin-up (s)quarks,
- 4 = isospin-down (s)quarks

$g = 1 \dots 3$ (s)fermion generation

$s = 1 \dots 2$ number of sfermion mass-eigenstate, in the absence of mixing 1 = L, 2 = R

$c = 1 \dots 2$ number of chargino mass-eigenstate

$n = 1 \dots 4$ number of neutralino mass-eigenstate

Matrices have a "`Flat`" array superimposed for convenience, in Fortran's standard column-major convention, e.g. `USf(1,1)` ≡ `USfFlat(1)`, `USf(2,1)` ≡ `USfFlat(2)`, `USf(1,2)` ≡ `USfFlat(3)`, `USf(2,2)` ≡ `USfFlat(4)`. This makes it possible to e.g. copy such a matrix with just a single do-loop.

2.2 PDG particle identifiers

`PDG.h` defines the human-readable versions of the PDG codes listed in Table 5. These are needed e.g. to access the decay information. At run time, the subroutine `SLHAPDGName` can be used to translate a PDG code into a particle name (see Sect. 3.9).

Block name	Offset and length	Members	
MODSEL	OffsetModSel	ModSel_Model	
	LengthModSel	ModSel_Content	
		ModSel_GridPts	
		ModSel_Qmax	
		ModSel_PDG(i)	$i = 1 \dots 5$
SMINPUTS	OffsetSMInputs	SMInputs_AlfaMZ	
	LengthSMInputs	SMInputs_GF	
		SMInputs_AlfasMZ	
		SMInputs_MZ	
		SMInputs_Mf(t)	$t = 2 \dots 4$
		SMInputs_Mtau	≡ SMInputs_Mf(2)
		SMInputs_Mt	≡ SMInputs_Mf(3)
		SMInputs_Mb	≡ SMInputs_Mf(4)
MINPAR	OffsetMinPar	MinPar_Q	
	LengthMinPar	MinPar_M0	
		MinPar_Lambda	≡ MinPar_M0
		MinPar_M12	
		MinPar_Mmess	≡ MinPar_M12
		MinPar_M32	≡ MinPar_M12
		MinPar_TB	
		MinPar_signMUE	
		MinPar_A	
		MinPar_N5	≡ MinPar_A
		MinPar_cgrav	

Table 1: Preprocessor variables defined in SLHA.h to access the slhadata array.

Block name	Offset and length	Members	
EXTPAR	OffsetExtPar	ExtPar_Q	
	LengthExtPar	ExtPar_M1	
		ExtPar_M2	
		ExtPar_M3	
		ExtPar_Af(t)	$t = 2 \dots 4$
		ExtPar_Atau	$\equiv$ ExtPar_Af(2)
		ExtPar_At	$\equiv$ ExtPar_Af(3)
		ExtPar_Ab	$\equiv$ ExtPar_Af(4)
		ExtPar_MHu2	
		ExtPar_MHd2	
		ExtPar_MUE	
		ExtPar_MA02	
		ExtPar_TB	
		ExtPar_MSL(g)	$g = 1 \dots 3$
		ExtPar_MSE(g)	$g = 1 \dots 3$
		ExtPar_MSQ(g)	$g = 1 \dots 3$
		ExtPar_MSU(g)	$g = 1 \dots 3$
		ExtPar_MSD(g)	$g = 1 \dots 3$
		ExtPar_N5(g)	$g = 1 \dots 3$
MASS	OffsetMass	Mass_Mf(t,g)	$t = 1 \dots 4$,
	LengthMass		$g = 1 \dots 3$
		Mass_MSf(s,t,g)	$s = 1 \dots 2$,
			$t = 1 \dots 4$,
			$g = 1 \dots 3$
		Mass_MZ	
		Mass_MW	
		Mass_Mh0	
		Mass_MHH	
		Mass_MA0	
		Mass_MHp	
		Mass_MNeu(n)	$n = 1 \dots 4$
		Mass_MCha(c)	$c = 1 \dots 2$
		Mass_MGl	
		Mass_MGrav	

Table 2: Preprocessor variables defined in SLHA.h to access the slhadata array (cont'd).

Block name	Offset and length	Members	
NMIX	OffsetNMix LengthNMix	NMix_ZNeu(n_1,n_2) NMix_ZNeuFlat(i)	$n_1, n_2 = 1 \ldots 4$ $i = 1 \ldots 16$
UMIX	OffsetUMix LengthUMix	UMix_UCha(c_1,c_2) UMix_UChaFlat(i)	$c_1, c_2 = 1 \ldots 2$ $i = 1 \ldots 4$
VMIX	OffsetVMix LengthVMix	VMix_VCha(c_1,c_2) VMix_VChaFlat(i)	$c_1, c_2 = 1 \ldots 2$ $i = 1 \ldots 4$
		SfMix_USf(s_1,s_2,t) SfMix_USfFlat(i,t)	$s_1, s_2 = 1 \ldots 2$, $t = 2 \ldots 4$ $i = 1 \ldots 4$, $t = 2 \ldots 4$
STAUMIX	OffsetStauMix LengthStauMix	StauMix_USf(s_1,s_2) StauMix_USfFlat(i)	$\equiv$ SfMix_USf(s_1,s_2,2) $\equiv$ SfMix_USfFlat(i,2)
STOPMIX	OffsetStopMix LengthStopMix	StopMix_USf(s_1,s_2) StopMix_USfFlat(i)	$\equiv$ SfMix_USf(s_1,s_2,3) $\equiv$ SfMix_USfFlat(i,3)
SBOTMIX	OffsetSbotMix LengthSbotMix	SbotMix_USf(s_1,s_2) SbotMix_USfFlat(i)	$\equiv$ SfMix_USf(s_1,s_2,4) $\equiv$ SfMix_USfFlat(i,4)
ALPHA	OffsetAlpha LengthAlpha	Alpha_Alpha	
HMIX	OffsetHMix LengthHMix	HMix_Q HMix_MUE HMix_TB HMix_VEV HMix_MA02	
GAUGE	OffsetGauge LengthGauge	Gauge_Q Gauge_g1 Gauge_g2 Gauge_g3	
MSOFT	OffsetMSoft LengthMSoft	MSoft_Q MSoft_M1 MSoft_M2 MSoft_M3 MSoft_MHu2 MSoft_MHd2 MSoft_MSL(g) MSoft_MSE(g) MSoft_MSQ(g) MSoft_MSU(g) MSoft_MSD(g)	 $g = 1 \ldots 3$ $g = 1 \ldots 3$ $g = 1 \ldots 3$ $g = 1 \ldots 3$ $g = 1 \ldots 3$

Table 3: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Offset and length	Members	
		Af_Q(t)	$t = 2 \ldots 4$
		Af_Af(t)	$t = 2 \ldots 4$
AE	OffsetAe	Ae_Q	$\equiv$ Af_Q(2)
	LengthAe	Ae_Atau	$\equiv$ Af_Af(2)
AU	OffsetAu	Au_Q	$\equiv$ Af_Q(3)
	LengthAu	Au_At	$\equiv$ Af_Af(3)
AD	OffsetAd	Ad_Q	$\equiv$ Af_Q(4)
	LengthAd	Ad_Ab	$\equiv$ Af_Af(4)
		Yf_Q(t)	$t = 2 \ldots 4$
		Yf_Af(t)	$t = 2 \ldots 4$
YE	OffsetYe	Ye_Q	$\equiv$ Yf_Q(2)
	LengthYe	Ye_Atau	$\equiv$ Yf_Yf(2)
YU	OffsetYu	Yu_Q	$\equiv$ Yf_Q(3)
	LengthYu	Yu_At	$\equiv$ Yf_Yf(3)
YD	OffsetYd	Yd_Q	$\equiv$ Yf_Q(4)
	LengthYd	Yd_Ab	$\equiv$ Yf_Yf(4)

Table 4: Preprocessor variables defined in SLHA.h to access the slhadata array (cont'd).

fermions	sfermions	
PDG_nu_e	PDG_snu_e1	PDG_snu_e2
PDG_electron	PDG_selectron1	PDG_selectron2
PDG_up	PDG_sup1	PDG_sup2
PDG_down	PDG_sdown1	PDG_sdown2
PDG_nu_mu	PDG_snu_mu1	PDG_snu_mu2
PDG_muon	PDG_smuon1	PDG_smuon2
PDG_charm	PDG_scharm1	PDG_scharm2
PDG_strange	PDG_sstrange1	PDG_sstrange2
PDG_nu_tau	PDG_snu_tau1	PDG_snu_tau2
PDG_tau	PDG_stau1	PDG_stau2
PDG_top	PDG_stop1	PDG_stop2
PDG_bottom	PDG_sbottom1	PDG_sbottom2

bosons	gauginos
PDG_h0	PDG_neutralino1
PDG_HH	PDG_neutralino2
PDG_A0	PDG_neutralino3
PDG_Hp	PDG_neutralino4
PDG_photon	PDG_chargino1
PDG_Z	PDG_chargino2
PDG_W	PDG_gluino
PDG_gluon	PDG_gravitino
PDG_graviton	

Table 5: The PDG codes defined in PDG.h.

3 Routines provided by the SLHA library

3.1 SLHAClear

```
      subroutine SLHAClear(slhadata)
      double precision slhadata(nslhadata)
```

This subroutine sets all data in the `slhadata` array given as argument to the value `invalid` (defined in `SLHA.h`). It is important that this is done before using `slhadata`, or else any kind of junk that happens to be in the memory occupied by `slhadata` will later on be interpreted as valid data.

3.2 SLHARead

```
      subroutine SLHARead(error, slhadata, filename, abort)
      integer error, abort
      double precision slhadata(nslhadata)
      character*(*) filename
```

This subroutine reads the data in SLHA format from `filename` into the `slhadata` array. If the specified file cannot be opened, the function issues an error message and returns `error = 1`. The `abort` flag governs what happens when superfluous text is read, i.e. text that cannot be interpreted as SLHA data. If `abort` is 0, a warning is printed and reading continues. Otherwise, reading stops at the offending line and `error = 2` is returned.

3.3 SLHAWrite

```
      subroutine SLHAWrite(error, slhadata,
     &    program, version, filename)
      integer error
      double precision slhadata(nslhadata)
      character*(*) program, version, filename
```

This subroutine writes the data in `slhadata` to `filename`. The name and version of the program that generates the output is given in `program` and `version`.

3.4 SLHAGetDecay

```
      double precision function SLHAGetDecay(slhadata, parent_id,
     &    nchildren, child1_id, child2_id, child3_id, child4_id)
      implicit none
      double precision slhadata(*)
```

```
      integer parent_id
      integer nchildren, child1_id, child2_id, child3_id, child4_id
```

This function extracts the decay

```
parent_id  →  child1_id  child2_id  child3_id  child4_id
```

from the `slhadata` array, or the value `invalid` (defined in `SLHA.h`) if no such decay can be found. The parent and child particles are given by their PDG identifiers (see Sect. 2.2). The return value is the total decay width if `nchildren = 0`, otherwise the branching ratio of the specified channel.

Note that only the first `nchildren` of the child$n$_id are actually accessed and Fortran allows to omit the remaining ones in the invocation (a strict syntax checker might issue a warning, though). Thus, for instance,

```
Zbb = SLHAGetDecay(slhadata, PDG_Z, 2, PDG_bottom, -PDG_bottom)
```

is a perfectly legitimate way to extract the $Z \to b\bar{b}$ decay.

3.5 SLHANewDecay

```
      integer function SLHANewDecay(slhadata, width, parent_id)
      double precision slhadata(nslhadata), width
      integer parent_id
```

This function initiates the setting of decay information for the particle specified by the `parent_id` PDG code, whose total decay width is given by `width`. The integer index it returns is needed to subsequently add individual decay modes with `SLHAAddDecay`. If the fixed-length array `slhadata` becomes full, a warning is printed and zero is returned. If a decay of the given particle is already present in `slhadata`, it is first removed.

3.6 SLHAAddDecay

```
      subroutine SLHAAddDecay(slhadata, br, decay,
     &   nchildren, child1_id, child2_id, child3_id, child4_id)
      double precision slhadata(nslhadata), br
      integer decay
      integer nchildren, child1_id, child2_id, child3_id, child4_id
```

This subroutine adds the decay mode

```
(parent_id)  →  child1_id  child2_id  child3_id  child4_id
```

to the decay section previously initiated by `SLHANewDecay`. `decay` is the index obtained from `SLHANewDecay` (which also sets the `parent_id`) and child$n$_id are the PDG codes of the final-state particles. The branching ratio is given in `br`. If the fixed-length array `slhadata` becomes full, a warning is printed and `decay` is set to zero.

If `decay` is zero, an overflow of `slhadata` in an earlier invocation is silently assumed and no action is performed. It is therefore sufficient to check for overflow only once, after setting all decay modes (unless, of course, one needs to pinpoint the exact location of the overflow).

As with `SLHAGetDecay` (see Sect. 3.4), only the first `nchildren` of the child$n$_id are actually accessed and Fortran allows to omit the remaining ones in the invocation.

3.7 SLHAExist

```
logical function SLHAExist(slhablock, length)
double precision slhablock(*)
integer length
```

This function tests whether a given SLHA block is not entirely empty, i.e. it returns `.TRUE.` if at least one member of the block is valid. The SLHA blocks are most conveniently accessed using the `Offset...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHAExist(slhadata(OffsetMass), LengthMass) ) ...
```

3.8 SLHAValid

```
logical function SLHAValid(slhablock, length)
double precision slhablock(*)
integer length
```

This function tests whether a given SLHA block consists entirely of valid data, i.e. it returns `.FALSE.` if at least one member of the block is invalid. The SLHA blocks are most conveniently accessed using the `Offset...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHAValid(slhadata(OffsetNMix), LengthNMix) ) ...
```

3.9 SLHAPDGName

```
subroutine SLHAPDGName(code, name)
integer code
character*(PDGLen) name
```

This subroutine translates a PDG code into a particle name. The sign of the PDG code is ignored, hence the same name is returned for a particle and its antiparticle. The maximum length of the name, `PDGLen`, is defined in `PDG.h`.

4 Examples

Consider the following example program, which just copies one SLHA file to another:

```
        program copy_slha_file
        implicit none

#include "SLHA.h"

        integer error
        double precision slhadata(nslhadata)

        call SLHAClear(slhadata)

        call SLHARead(error, slhadata, "infile.slha", 0)
        if( error .ne. 0 ) stop "Read error"

        call SLHAWrite(error, slhadata,
     &    "My Test Program", "1.0", "outfile.slha")
        if( error .ne. 0 ) stop "Write error"
        end
```

Already in this simple program a couple of things can be seen:

- the file `SLHA.h` must be included in every function or subroutine that uses the SLHA routines and this must be done using the preprocessor `#include` (not Fortran's `include`), thus the program file should have the extension `.F` (capital F).
- `slhadata` must be declared as a double-precision array of length `nslhadata`.
- One should not continue with processing if a non-zero error flag is returned.

A more sensible application would add something to the `slhadata` before writing them out again. The next little program pretends to compute the fermionic Z decays (by calling a hypothetical subroutine `MyCalculation`) and adds them to `slhadata`:

```
        program compute_decays
        implicit none

#include "SLHA.h"
#include "PDG.h"

        integer error, decay, t, g
        double precision slhadata(nslhadata)
        double precision total_width, br(4,3)
```

```
      integer ferm_id(4,3)
      data ferm_id /
     &   PDG_nu_e, PDG_electron, PDG_up, PDG_down,
     &   PDG_nu_mu, PDG_muon, PDG_charm, PDG_strange,
     &   PDG_nu_tau, PDG_tau, PDG_top, PDG_bottom /

      call SLHAClear(slhadata)

      call SLHARead(error, slhadata, "infile.slha", 0)
      if( error .ne. 0 ) stop "Read error"

* compute the decays with parameters taken from the slhadata:
      call MyCalculation(SMInputs_MZ, MinPar_TB, ...,
     &   total_width, br)

      decay = SLHANewDecay(slhadata, total_width, PDG_Z)
      do t = 1, 4
        do g = 1, 3
          call SLHAAddDecay(slhadata, br(t,g), decay,
     &       2, ferm_id(t,g), -ferm_id(t,g))
        enddo
      enddo

      call SLHAWrite(error, slhadata,
     &   "My Test Program", "2.0", "outfile.slha")
      if( error .ne. 0 ) stop "Write error"
      end
```

Demonstrated here is the access of SLHA data (`SMInputs_MZ`, `MinPar_TB`) and the setting of decay information.

5 Building and Compiling

The SLHA library package can be downloaded as a gzipped tar archive from the Web site `http://www.feynarts.de/slha`. After unpacking the archive, change into the directory `SLHALib-1.0` and type

```
./configure
make
```

A simple demonstration program (`demo`, source code in `demo.F`) is built together with the library `libSLHA.a`.

Compiling a program that uses the SLHA library is in principle equally straightforward. The only tricky thing is that one has to relax Fortran's 72-column limit. This is because even lines perfectly within the 72-column range may become longer after the preprocessor's substitutions. While essentially every Fortran compiler offers such an option, the name is quite different. A glance at the man page should suffice to find out. Here are a few common choices:

Compiler	Platform/OS	Option name
g77	any	`-ffixed-line-length-none`
pgf77	Linux x86	`-Mextend`
f77	Tru64 Alpha	`-extend_source`
f77	SunOS, Solaris	`-e`
fort77	HP-UX	`+es`

To compile and link your program, add this option and `-I`*path* `-L`*path* `-lSLHA` to the compiler command line, where *path* is the location of the SLHA library, e.g.

```
pgf77 -Mextend -I$HOME/SLHALib-1.0 myprogram.F -L$HOME/SLHALib-1.0 -lSLHA
```

All externally visible symbols of the SLHA library start with the prefix `SLHA` and should thus pretty much avoid symbol conflicts.

6 Summary

The SLHA library presented here provides simple functions to read and write files in SLHA format. Data are kept in a single double-precision array and accessed through preprocessor variables. The library is written in native Fortran 77 and is easy to build. The source code is openly available at `http://www.feynarts.de/slha` and is distributed under the GNU Library General Public License.

The author welcomes any kind of feedback, in particular bug and performance reports, at hahn@feynarts.de.

References

[1] P. Skands et al., hep-ph/0311123.

[2] T. Hahn and C. Schappacher, *Comp. Phys. Commun.* **143** (2002) 54 [hep-ph/0105349].