MPP–2006–46
hep–ph/0605049

SUSY Les Houches Accord 2 I/O made easy

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

Dec 19, 2006

Abstract

A library for reading and writing data in the SUSY Les Houches Accord 2 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 original SUSY Les Houches Accord [1] (SLHA1 in the following) has standardized and significantly simplified the exchange of MSSM input and output parameters between such disparate applications as spectrum calculators and event generators. Meanwhile, agreement has been reached also about the encoding of many extensions of the MSSM which has led to a preliminary SLHA2 document [2].

While the SLHA specifications 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. The data structures and subroutines are set up such that only very few changes are necessary when upgrading from the SLHALib 1 [3].

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 complex 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 `SLHANewDecay`, `SLHAFindDecay`, `SLHAAddDecay`, `SLHAGetDecay`, and `SLHADecayTable` 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 1–10 list the preprocessor variables defined in `SLHA.h` which follow closely the definition of the Accord. Note that preprocessor symbols are case sensitive. On the downside, there is no way to guard against out-of-range indices, not even with compiler flags. This is because the preprocessor has no such checks and the compiler cannot determine *a posteriori* whether the single index it sees addresses the ‘right’ part of the array.

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* [4]. Following is a list of common index conventions. This is only for a rough orientation: the actual indices and their ranges are always given explicitly 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

For each block B the offset into `slhadata` and the length are respectively defined as `Offset`B and `Length`B. The contents of the block can be addressed through the macro `Block`B`(`i`)`, where i runs from 1 to `Length`B.

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.

Block name	Array and length	Members	
SPINFO	`BlockSPInfo(n)`	`SPInfo_Severity`	
	`LengthSPInfo`	`SPInfo_NLines`	
		`SPInfo_Code(n)`	$n = 1 \ldots 15$
		`SPInfo_Text(i,n)`	do not address directly
DCINFO	`BlockDCInfo(n)`	`DCInfo_Severity`	
	`LengthDCInfo`	`DCInfo_NLines`	
		`DCInfo_Code(n)`	$n = 1 \ldots 15$
		`DCInfo_Text(i,n)`	do not address directly
MODSEL	`BlockModSel(n)`	`ModSel_Model`	
	`LengthModSel`	`ModSel_Content`	
		`ModSel_RPV`	
		`ModSel_CPV`	
		`ModSel_FV`	
		`ModSel_GridPts`	
		`ModSel_Qmax`	
		`ModSel_PDG(i)`	$i = 1 \ldots 5$
SMINPUTS	`BlockSMInputs(n)`	`SMInputs_AlfaMZ`	
	`LengthSMInputs`	`SMInputs_invAlfaMZ`	≡ `SMInputs_AlfaMZ`
		`SMInputs_GF`	
		`SMInputs_AlfasMZ`	
		`SMInputs_MZ`	
		`SMInputs_Mf(t,g)`	$t = 2 \ldots 4,\ g = 1 \ldots 3$
		`SMInputs_MfFlat(i)`	$i = 1 \ldots 9$
		`SMInputs_Me`	≡ `SMInputs_Mf(2,1)`
		`SMInputs_Mu`	≡ `SMInputs_Mf(3,1)`
		`SMInputs_Md`	≡ `SMInputs_Mf(4,1)`
		`SMInputs_Mmu`	≡ `SMInputs_Mf(2,2)`
		`SMInputs_Mc`	≡ `SMInputs_Mf(3,2)`
		`SMInputs_Ms`	≡ `SMInputs_Mf(4,2)`
		`SMInputs_Mtau`	≡ `SMInputs_Mf(2,3)`
		`SMInputs_Mt`	≡ `SMInputs_Mf(3,3)`
		`SMInputs_Mb`	≡ `SMInputs_Mf(4,3)`

Table 1: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array. The equivalence symbol (≡) indicates that the l.h.s. is just an alias for the r.h.s., not a new variable.

Block name	Array and length	Members	
MINPAR	BlockMinPar(n)	MinPar_Q	
	LengthMinPar	MinPar_M0	
		MinPar_Lambda	$\equiv$ MinPar_M0
		MinPar_M12	
		MinPar_Mmess	$\equiv$ MinPar_M12
		MinPar_M32	$\equiv$ MinPar_M12
		MinPar_TB	
		MinPar_signMUE	
		MinPar_A	
		MinPar_N5	$\equiv$ MinPar_A
		MinPar_cgrav	
EXTPAR	BlockExtPar(n)	ExtPar_Q	
	LengthExtPar	ExtPar_M1	
		ExtPar_M2	
		ExtPar_M3	
		ExtPar_Af(t)	$t = 2 \ldots 4$
		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_MSS(g,q)	$g = 1 \ldots 3, \; q = 1 \ldots 5$
		ExtPar_MSL(g)	$\equiv$ ExtPar_MSS(g,1)
		ExtPar_MSE(g)	$\equiv$ ExtPar_MSS(g,2)
		ExtPar_MSQ(g)	$\equiv$ ExtPar_MSS(g,3)
		ExtPar_MSU(g)	$\equiv$ ExtPar_MSS(g,4)
		ExtPar_MSD(g)	$\equiv$ ExtPar_MSS(g,5)
		ExtPar_N5(g)	$g = 1 \ldots 3$
		ExtPar_lambda	
		ExtPar_kappa	
		ExtPar_Alambda	
		ExtPar_Akappa	
		ExtPar_MUEeff	

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

Block name	Array and length	Members	
MASS	BlockMass(n)	Mass_Mf(t,g)	$t = 1 \ldots 4,\ g = 1 \ldots 3$
	LengthMass	Mass_MfFlat(i)	$i = 1 \ldots 12$
		Mass_MSf(s,t,g)	$s = 1 \ldots 2,\ t = 1 \ldots 4,\ g = 1 \ldots 3$
		Mass_MSfFlat(i)	$i = 1 \ldots 24$
		Mass_MZ	
		Mass_MW	
		Mass_Mh0	
		Mass_MHH	
		Mass_MA0	
		Mass_MHp	
		Mass_MH1	$\equiv$ Mass_Mh0
		Mass_MH2	$\equiv$ Mass_MHH
		Mass_MH3	
		Mass_MA1	$\equiv$ Mass_MA0
		Mass_MA2	
		Mass_MNeu(n)	$n = 1 \ldots 5$
		Mass_MCha(c)	$c = 1 \ldots 2$
		Mass_MGl	
		Mass_MGrav	
NMIX	BlockNMix(n)	NMix_ZNeu(n_1,n_2)	$n_1, n_2 = 1 \ldots 4$
	LengthNMix	NMix_ZNeuFlat(i)	$i = 1 \ldots 16$
UMIX	BlockUMix(n)	UMix_UCha(c_1,c_2)	$c_1, c_2 = 1 \ldots 2$
	LengthUMix	UMix_UChaFlat(i)	$i = 1 \ldots 4$
VMIX	BlockVMix(n)	VMix_VCha(c_1,c_2)	$c_1, c_2 = 1 \ldots 2$
	LengthVMix	VMix_VChaFlat(i)	$i = 1 \ldots 4$

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

Block name	Array and length	Members	
	BlockSfMix(n)	SfMix_USf(s_1,s_2,t)	$s_1, s_2 = 1 \ldots 2,\ t = 2 \ldots 4$
	LengthSfMix	SfMix_USfFlat(i,t)	$i = 1 \ldots 4,\ t = 2 \ldots 4$
STAUMIX	BlockStauMix(n)	StauMix_USf(s_1,s_2)	$\equiv$ SfMix_USf(s_1,s_2,2)
	LengthStauMix	StauMix_USfFlat(i)	$\equiv$ SfMix_USfFlat(i,2)
STOPMIX	BlockStopMix(n)	StopMix_USf(s_1,s_2)	$\equiv$ SfMix_USf(s_1,s_2,3)
	LengthStopMix	StopMix_USfFlat(i)	$\equiv$ SfMix_USfFlat(i,3)
SBOTMIX	BlockSbotMix(n)	SbotMix_USf(s_1,s_2)	$\equiv$ SfMix_USf(s_1,s_2,4)
	LengthSbotMix	SbotMix_USfFlat(i)	$\equiv$ SfMix_USfFlat(i,4)
ALPHA	BlockAlpha(n)	Alpha_Alpha	
	LengthAlpha		
HMIX	BlockHMix(n)	HMix_Q	
	LengthHMix	HMix_MUE	
		HMix_TB	
		HMix_VEV	
		HMix_MA02	
GAUGE	BlocktGauge(n)	Gauge_Q	
	LengthGauge	Gauge_g1	
		Gauge_g2	
		Gauge_g3	
MSOFT	BlockMSoft(n)	MSoft_Q	
	LengthMSoft	MSoft_M1	
		MSoft_M2	
		MSoft_M3	
		MSoft_MHu2	
		MSoft_MHd2	
		MSoft_MSS(g,q)	$g = 1 \ldots 3,\ q = 1 \ldots 5$
		MSoft_MSL(g)	$\equiv$ MSoft_MSS(g,1)
		MSoft_MSE(g)	$\equiv$ MSoft_MSS(g,2)
		MSoft_MSQ(g)	$\equiv$ MSoft_MSS(g,3)
		MSoft_MSU(g)	$\equiv$ MSoft_MSS(g,4)
		MSoft_MSD(g)	$\equiv$ MSoft_MSS(g,5)

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

Block name	Array and length	Members	
	BlockAf(n)	Af_Q(t)	$t = 2 \ldots 4$
	LengthAf	Af_Af(g_1,g_2,t)	$g_1, g_2 = 1 \ldots 3,\ t = 2 \ldots 4$
		Af_AfFlat(i,t)	$i = 1 \ldots 9,\ t = 2 \ldots 4$
AE	BlockAe(n)	Ae_Q	$\equiv$ Af_Q(2)
	LengthAe	Ae_Af(g_1,g_2)	$\equiv$ Af_Af(g_1,g_2,2)
		Ae_AfFlat(i)	$\equiv$ Af_AfFlat(i,2)
		Ae_Atau	$\equiv$ Ae_Af(3,3)
AU	BlockAu(n)	Au_Q	$\equiv$ Af_Q(3)
	LengthAu	Au_Af(g_1,g_2)	$\equiv$ Af_Af(g_1,g_2,3)
		Au_AfFlat(i)	$\equiv$ Af_AfFlat(i,3)
		Au_At	$\equiv$ Au_Af(3,3)
AD	BlockAd(n)	Ad_Q	$\equiv$ Af_Q(4)
	LengthAd	Ad_Af(g_1,g_2)	$\equiv$ Af_Af(g_1,g_2,4)
		Ad_AfFlat(i)	$\equiv$ Af_AfFlat(i,4)
		Ad_Ab	$\equiv$ Ad_Af(3,3)
	BlockYf(n)	Yf_Q(t)	$t = 2 \ldots 4$
	LengthYf	Yf_Af(g_1,g_2,t)	$g_1, g_2 = 1 \ldots 3,\ t = 2 \ldots 4$
		Yf_AfFlat(i,t)	$i = 1 \ldots 9,\ t = 2 \ldots 4$
YE	BlockYe(n)	Ye_Q	$\equiv$ Yf_Q(2)
	LengthYe	Ye_Yf(g_1,g_2)	$\equiv$ Yf_Yf(g_1,g_2,2)
		Ye_YfFlat(i)	$\equiv$ Yf_YfFlat(i,2)
		Ye_Atau	$\equiv$ Ye_Yf(3,3)
YU	BlockYu(n)	Yu_Q	$\equiv$ Yf_Q(3)
	LengthYu	Yu_Yf(g_1,g_2)	$\equiv$ Yf_Yf(g_1,g_2,3)
		Yu_YfFlat(i)	$\equiv$ Yf_YfFlat(i,3)
		Yu_At	$\equiv$ Yu_Yf(3,3)
YD	BlockYd(n)	Yd_Q	$\equiv$ Yf_Q(4)
	LengthYd	Yd_Yf(g_1,g_2)	$\equiv$ Yf_Yf(g_1,g_2,4)
		Yd_YfFlat(i)	$\equiv$ Yf_YfFlat(i,4)
		Yd_Ab	$\equiv$ Yd_Yf(3,3)

Table 5: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members	
RVLAMBDAIN	BlockRVLambdaIn(n)	RVLambdaIn_lambda(i,j,k)	$i,j,k = 1\dots3$
	LengthRVLambdaIn	RVLambdaIn_lambdaFlat(i)	$i = 1\dots27$
RVLAMBDA	BlockRVLambda(n)	RVLambda_Q	
	LengthRVLambda	RVLambda_lambda(i,j,k)	$i,j,k = 1\dots3$
		RVLambda_lambdaFlat(i)	$i = 1\dots27$
RVLAMBDAPIN	BlockRVLambdaPIn(n)	RVLambdaPIn_lambdaP(i,j,k)	$i,j,k = 1\dots3$
	LengthRVLambdaPIn	RVLambdaPIn_lambdaPFlat(i)	$i = 1\dots27$
RVLAMBDAP	BlockRVLambdaP(n)	RVLambdaP_Q	
	LengthRVLambdaP	RVLambdaP_lambdaP(i,j,k)	$i,j,k = 1\dots3$
		RVLambdaP_lambdaPFlat(i)	$i = 1\dots27$
RVLAMBDAPPIN	BlockRVLambdaPPIn(n)	RVLambdaPPIn_lambdaPP(i,j,k)	$i,j,k = 1\dots3$
	LengthRVLambdaPPIn	RVLambdaPPIn_lambdaPPFlat(i)	$i = 1\dots27$
RVLAMBDAPP	BlockRVLambdaPP(n)	RVLambdaPP_Q	
	LengthRVLambdaPP	RVLambdaPP_lambdaPP(i,j,k)	$i,j,k = 1\dots3$
		RVLambdaPP_lambdaPPFlat(i)	$i = 1\dots27$
RVAIN	BlockRVAIn(n)	RVAIn_A(i,j,k)	$i,j,k = 1\dots3$
	LengthRVAIn	RVAIn_AFlat(i)	$i = 1\dots27$
RVA	BlockRVA(n)	RVA_Q	
	LengthRVA	RVA_A(i,j,k)	$i,j,k = 1\dots3$
		RVA_AFlat(i)	$i = 1\dots27$
RVAPIN	BlockRVAPIn(n)	RVAPIn_AP(i,j,k)	$i,j,k = 1\dots3$
	LengthRVAPIn	RVAPIn_APFlat(i)	$i = 1\dots27$
RVAP	BlockRVAP(n)	RVAP_Q	
	LengthRVAP	RVAP_AP(i,j,k)	$i,j,k = 1\dots3$
		RVAP_APFlat(i)	$i = 1\dots27$
RVAPPIN	BlockRVAPPIn(n)	RVAPPIn_APP(i,j,k)	$i,j,k = 1\dots3$
	LengthRVAPPIn	RVAPPIn_APPFlat(i)	$i = 1\dots27$
RVAPP	BlockRVAPP	RVAPP_Q	
	LengthRVAPP	RVAPP_APP(i,j,k)	$i,j,k = 1\dots3$
		RVAPP_APPFlat(i)	$i = 1\dots27$

Table 6: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members	
RVKAPPAIN	`BlockRVKappaIn`(n) `LengthRVKappaIn`	`RVKappaIn_kappa`(i)	$i = 1 \dots 3$
RVKAPPA	`BlockRVKappa`(n) `LengthRVKappa`	`RVKappa_Q` `RVKappa_kappa`(i)	 $i = 1 \dots 3$
RVDIN	`BlockRVDIn`(n) `LengthRVDIn`	`RVDIn_D`(i)	$i = 1 \dots 3$
RVD	`BlockRVD`(n) `LengthRVD`	`RVD_Q` `RVD_D`(i)	 $i = 1 \dots 3$
RVSNVEVIN	`BlockRVSnVEVIn`(n) `LengthRVSnVEVIn`	`RVSnVEVIn_VEV`(i)	$i = 1 \dots 3$
RVSNVEV	`BlockRVSnVEV`(n) `LengthRVSnVEV`	`RVSnVEV_Q` `RVSnVEV_VEV`(i)	 $i = 1 \dots 3$
RVMLH1SQIN	`BlockRVMLH1SqIn`(n) `LengthRVMLH1SqIn`	`RVMLH1SqIn_MLH12`(i)	$i = 1 \dots 3$
RVMLH1SQ	`BlockRVMLH1Sq`(n) `LengthRVMLH1Sq`	`RVMLH1Sq_Q` `RVMLH1Sq_MLH12`(i)	 $i = 1 \dots 3$
RVNMIX	`BlockRVNMix`(n) `LengthRVNMix`	`RVNMix_ZNeu`(n_1,n_2) `RVNMix_ZNeuFlat`(i)	$n_1, n_2 = 1 \dots 7$ $i = 1 \dots 49$
RVUMIX	`BlockRVUMix`(n) `LengthRVUMix`	`RVUMix_UCha`(c_1,c_2) `RVUMix_UChaFlat`(i)	$c_1, c_2 = 1 \dots 5$ $i = 1 \dots 25$
RVVMIX	`BlockRVVMix`(n) `LengthRVVMix`	`RVVMix_VCha`(c_1,c_2) `RVVMix_VChaFlat`(i)	$c_1, c_2 = 1 \dots 5$ $i = 1 \dots 25$
RVHMIX	`BlockRVHMix`(n) `LengthRVHMix`	`RVUMix_UH`(h_1,h_2) `RVUMix_UHFlat`(i)	$h_1, h_2 = 1 \dots 5$ $i = 1 \dots 25$
RVAMIX	`BlockRVAMix`(n) `LengthRVAMix`	`RVAMix_UA`(h_1,h_2) `RVAMix_UAFlat`(i)	$h_1, h_2 = 1 \dots 5$ $i = 1 \dots 25$
RVLMIX	`BlockRVLMix`(n) `LengthRVLMix`	`RVLMix_CLep`(l_1,l_2) `RVLMix_CLepFlat`(i)	$l_1, l_2 = 1 \dots 8$ $i = 1 \dots 64$

Table 7: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members

VCKMINPUTS	`BlockVCKMInputs(`n`)`
	`LengthVCKMInputs`
		`VCKMInputs_theta12`
		`VCKMInputs_theta23`
		`VCKMInputs_theta13`
		`VCKMInputs_delta13`

VCKM	`BlockVCKM(`n`)`
	`LengthVCKM`
		`VCKM_Q`
		`VCKM_theta12`
		`VCKM_theta23`
		`VCKM_theta13`
		`VCKM_delta13`

Table 8: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

2.2 PDG particle identifiers

`PDG.h` defines the human-readable versions of the PDG codes listed in Table 11. 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. 3.12).

Block name	Array and length	Members	
	BlockMSS2In(n)	MSS2In_MSS2(g_1,g_2,q)	$g_1, g_2 = 1 \ldots 3,\ q = 1 \ldots 5$
	LengthMSS2In	MSS2In_MSS2Flat(i,q)	$i = 1 \ldots 9,\ q = 1 \ldots 5$
MSL2IN	BlockMSL2In(n)	MSL2In_MSL2(g_1,g_2)	$\equiv$ MSS2In_MSS2(g_1,g_2,1)
	LengthMSL2In	MSL2In_MSL2Flat(i)	$\equiv$ MSS2In_MSS2Flat(i,1)
MSE2IN	BlockMSE2In(n)	MSE2In_MSE2(g_1,g_2)	$\equiv$ MSS2In_MSS2(g_1,g_2,2)
	LengthMSE2In	MSE2In_MSE2Flat(i)	$\equiv$ MSS2In_MSS2Flat(i,2)
MSQ2IN	BlockMSQ2In(n)	MSQ2In_MSQ2(g_1,g_2)	$\equiv$ MSS2In_MSS2(g_1,g_2,3)
	LengthMSQ2In	MSQ2In_MSQ2Flat(i)	$\equiv$ MSS2In_MSS2Flat(i,3)
MSU2IN	BlockMSU2In(n)	MSU2In_MSU2(g_1,g_2)	$\equiv$ MSS2In_MSS2(g_1,g_2,4)
	LengthMSU2In	MSU2In_MSU2Flat(i)	$\equiv$ MSS2In_MSS2Flat(i,4)
MSD2IN	BlockMSD2In(n)	MSD2In_MSD2(g_1,g_2)	$\equiv$ MSS2In_MSS2(g_1,g_2,5)
	LengthMSD2In	MSD2In_MSD2Flat(i)	$\equiv$ MSS2In_MSS2Flat(i,5)
	BlockMSS2(n)	MSS2_Q(q)	$q = 1 \ldots 5$
	LengthMSS2	MSS2_MSS2(g_1,g_2,q)	$g_1, g_2 = 1 \ldots 3,\ q = 1 \ldots 5$
		MSS2_MSS2Flat(i,q)	$i = 1 \ldots 9,\ q = 1 \ldots 5$
MSL2	BlockMSL2(n)	MSL2_Q	$\equiv$ MSS2_Q(1)
	LengthMSL2	MSL2_MSL2(g_1,g_2)	$\equiv$ MSS2_MSS2(g_1,g_2,1)
		MSL2_MSL2Flat(i)	$\equiv$ MSS2_MSS2Flat(i,1)
MSE2	BlockMSE2(n)	MSE2_Q	$\equiv$ MSS2_Q(2)
	LengthMSE2	MSE2_MSE2(g_1,g_2)	$\equiv$ MSS2_MSS2(g_1,g_2,2)
		MSE2_MSE2Flat(i)	$\equiv$ MSS2_MSS2Flat(i,2)
MSQ2	BlockMSQ2(n)	MSQ2_Q	$\equiv$ MSS2_Q(3)
	LengthMSQ2	MSQ2_MSQ2(g_1,g_2)	$\equiv$ MSS2_MSS2(g_1,g_2,3)
		MSQ2_MSQ2Flat(i)	$\equiv$ MSS2_MSS2Flat(i,3)
MSU2	BlockMSU2(n)	MSU2_Q	$\equiv$ MSS2_Q(4)
	LengthMSU2	MSU2_MSU2(g_1,g_2)	$\equiv$ MSS2_MSS2(g_1,g_2,4)
		MSU2_MSU2Flat(i)	$\equiv$ MSS2_MSS2Flat(i,4)
MSD2	BlockMSD2(n)	MSD2_Q	$\equiv$ MSS2_Q(5)
	LengthMSD2	MSD2_MSD2(g_1,g_2)	$\equiv$ MSS2_MSS2(g_1,g_2,5)
		MSD2_MSD2Flat(i)	$\equiv$ MSS2_MSS2Flat(i,5)
	BlockASfMix(n)	ASfMix_UASf(s_1,s_2,t)	$s_1, s_2 = 1 \ldots 6,\ t = 1 \ldots 4$
	LengthASfMix	ASfMix_UASfFlat(i,t)	$i = 1 \ldots 36,\ t = 1 \ldots 4$
SNMIX	BlockSnMix(n)	SnMix_UASf(s_1,s_2)	$\equiv$ ASfMix_UASf(s_1,s_2,1)
	LengthSnMix	SnMix_UASfFlat(i)	$\equiv$ ASfMix_UASfFlat(i,1)
SLMIX	BlockSlMix(n)	SlMix_UASf(s_1,s_2)	$\equiv$ ASfMix_UASf(s_1,s_2,2)
	LengthSlMix	SlMix_UASfFlat(i)	$\equiv$ ASfMix_UASfFlat(i,2)
USQMIX	BlockUSqMix(n)	USqMix_UASf(s_1,s_2)	$\equiv$ ASfMix_UASf(s_1,s_2,3)
	LengthUSqMix	USqMix_UASfFlat(i)	$\equiv$ ASfMix_UASfFlat(i,3)
DSQMIX	BlockDSqMix(n)	DSqMix_UASf(s_1,s_2)	$\equiv$ ASfMix_UASf(s_1,s_2,4)
	LengthDSqMix	DSqMix_UASfFlat(i)	$\equiv$ ASfMix_UASfFlat(i,4)

Table 9: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members	
CVHMIX	`BlockCVHMix(`n`)`	`CVHMix_UH(`h_1`,`h_2`)`	$h_1, h_2 = 1 \dots 4$
	`LengthCVHMix`	`CVHMix_UHFlat(`i`)`	$i = 1 \dots 16$
NMNMIX	`BlockNMNMix(`n`)`	`NMNMix_ZNeu(`n_1`,`n_2`)`	$n_1, n_2 = 1 \dots 5$
	`LengthNMNMix`	`NMNMix_ZNeuFlat(`i`)`	$i = 1 \dots 25$
NMHMIX	`BlockNMHMix(`n`)`	`NMUMix_UH(`h_1`,`h_2`)`	$h_1, h_2 = 1 \dots 3$
	`LengthNMHMix`	`NMUMix_UHFlat(`i`)`	$i = 1 \dots 9$
NMAMIX	`BlockNMAMix`	`NMAMix_UA(`h_1`,`h_2`)`	$h_1, h_2 = 1 \dots 3$
	`LengthNMAMix`	`NMAMix_UAFlat(`i`)`	$i = 1 \dots 9$

Table 10: 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_H3`	`PDG_neutralino5`
`PDG_A2`	`PDG_chargino1`
`PDG_photon`	`PDG_chargino2`
`PDG_Z`	`PDG_gluino`
`PDG_W`	`PDG_gravitino`
`PDG_gluon`	
`PDG_graviton`	

Table 11: The PDG codes defined in `PDG.h`.

3 Routines provided by the SLHA library

The file `SLHA.h` must be included in every subroutine or function that uses SLHALib routines. It contains the necessary preprocessor definitions as well as external declarations for the SLHALib routines.

The basic data structure is the double complex array `slhadata` of length `nslhadata`. These names are hard-coded into the preprocessor definitions and may not be changed by the user. As a corollary, only one instance of the `slhadata` structure can be used in any one routine. This poses no serious limitation for most applications, however.

3.1 SLHAClear

```
subroutine SLHAClear(slhadata)
double complex 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 complex 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. `SLHARead` implicitly calls `SLHAClear` to clear the `slhadata` array before reading the file.

The blocks SPINFO and DCINFO are largely ignored when reading the file, as they are for human information only. Only the maximum of all message codes is kept in the `Severity` member of the block. Since the message codes increase with severity, this indicates the overall reliability of the corresponding data (spectrum or decay information). For example, if the `Severity` member is 4 (real errors), the Accord advises not to use the corresponding data. See also Sect. 3.4.

3.3 SLHAWrite

```
subroutine SLHAWrite(error, slhadata, filename)
integer error
double complex slhadata(nslhadata)
character*(*) filename
```

This subroutine writes the data in `slhadata` to `filename`.

3.4 SLHAInfo

```
subroutine SLHAInfo(slhablock, code, text)
double complex slhablock(*)
integer code
character*(*) text
```

This subroutine adds a message to one of the informational blocks, SPINFO or DCINFO. The block is most conveniently addressed through the `Block...` macros, for example

```
call SLHAInfo(BlockSPInfo(1), 4, "Error in computation")
```

Allowed codes are

- 1 = program name,
- 2 = program version,
- 3 = warning message,
- 4 = error message.

Messages are truncated at 80 characters.

3.5 SLHANewDecay

```
integer function SLHANewDecay(slhadata, width, parent_id)
double complex slhadata(nslhadata)
double precision 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 return value is an integer index which 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 SLHAFindDecay

```
integer function SLHAFindDecay(slhadata, parent_id)
double complex slhadata(nslhadata)
integer parent_id
```

This function also initiates the setting of decay information. Unlike `SLHANewDecay`, it requires that the decay of the `parent_id` particle exist and reshuffles the decay information in `slhadata` such that new channels can be added to this decay. If no decay matching `parent_id` is found, the return value is 0, otherwise it is the index needed to add decay modes with `SLHAAddDecay`.

3.7 SLHAAddDecay

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

This subroutine adds the decay mode

`(parent_id)` $\to$ `child1_id` `child2_id` `child3_id` `child4_id`

to the decay section previously initiated by `SLHANewDecay` or `SLHAFindDecay`. `decay` is the index obtained from the latter (which also set 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. 3.8), only the first `nchildren` of the `child`n`_id` are actually accessed and Fortran allows to omit the remaining ones in the invocation.

3.8 SLHAGetDecay

```
      double precision function SLHAGetDecay(slhadata, parent_id,
   &    nchildren, child1_id, child2_id, child3_id, child4_id)
      double complex slhadata(*)
      integer parent_id
      integer nchildren, child1_id, child2_id, child3_id, child4_id
```

This function extracts the decay

`parent_id` $\to$ `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.

3.9 SLHADecayTable

```
      integer function SLHADecayTable(slhadata, parent_id,
     &    width, id, maxparticles, maxchannels)
      double complex slhadata(nslhadata)
      integer parent_id, maxparticles, maxchannels
      double precision width(maxchannels)
      integer id(0:maxparticles,maxchannels)
```

This function stores all decay channels for the particle identified by `parent_id` in the arrays `id` and `width`. Unlike `SLHAGetDecay`, one does not need to know the exact decay mode in order to extract information. The value 0 for `parent_id` serves as a wildcard and transfers the entire decay table contained in `slhadata`. `SLHADecayTable` returns the number of channels found. The two arrays can be read out rather straightforwardly:

For each channel c,

- $n =$ `id(0,`c`)` gives the number of participating particles, i.e. the number of decay products plus one.
- The PDG code of the decaying particle is in `id(1,`c`)`.
- The PDG codes of the decay products are in `id(2,`c`)` … `id(`n`,`c`)`.
- If $n = 1$, `width(`c`)` contains the decaying particle's total width in GeV.
- If $n > 1$, `width(`c`)` contains the branching ratio for the given decay.

3.10 SLHAExist

```
integer function SLHAExist(slhablock, length)
double complex slhablock(*)
integer length
```

This function tests whether a given SLHA block is not entirely empty. It returns 2 if the block has at least one complex member, 1 if the block has at least one real member (i.e. all imaginary parts zero), and 0 if the block has no valid members at all. The SLHA blocks are most conveniently accessed using the `Block...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHAExist(BlockMass(1), LengthMass) .ne. 0 ) ...
```

3.11 SLHAValid

```
integer function SLHAValid(slhablock, length)
double complex slhablock(*)
integer length
```

This function tests whether a given SLHA block consists entirely of valid data, i.e. it returns 0 if at least one member of the block is invalid. The SLHA blocks are most conveniently accessed using the `Block...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHAValid(BlockNMix(1), LengthNMix) .ne. 0 ) ...
```

3.12 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`.

3.13 Incompatible Changes

Two incompatible changes in the interface were necessary with respect to the SLHALib 1 [3], largely due to the fact that the SLHA2 allows complex entries:

- `slhadata` is now a double complex, not a double precision array.

- The `SLHAExist` function has become an integer function, as it now distinguishes *three* possible scenarios: no valid entries, only real entries, and complex entries.

- The `SLHAWrite` subroutine no longer has arguments for program name and version. Such informational messages can now be added with the much more general subroutine `SLHAInfo`.

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 complex slhadata(nslhadata)

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

        call SLHAInfo(BlockSPInfo(1), 1, "My Test Program")
        call SLHAInfo(BlockSPInfo(1), 2, "1.0")

        call SLHAWrite(error, slhadata, "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 complex 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 complex 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 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 g = 1, 3
          do t = 1, 4
            call SLHAAddDecay(slhadata, br(t,g), decay,
     &        2, ferm_id(t,g), -ferm_id(t,g))
          enddo
        enddo

        call SLHAInfo(BlockDCInfo(1), 1, "My Decay Calculator")
        call SLHAInfo(BlockDCInfo(1), 2, "3.1415")

        call SLHAWrite(error, slhadata, "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-2.1` and type

```
./configure
make
make install
```

Some simple demonstration programs are located in the `demo` subdirectory.

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`
ifort	Linux x86	`-extend_source`
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.

```
SLHALIB=$HOME/SLHALib-2.1/$HOSTTYPE
pgf77 -Mextend -I$SLHALIB/include myprogram.F -L$SLHALIB/lib -lSLHA
```

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

It is also possible to use the SLHALib in C and C++. In this case one needs to include the header file `CSLHA.h` in the program text. Compilation should be done using the `fcc` script, i.e. replace the invocation of the C compiler with `fcc`, as in

```
fcc -I$SLHALIB/include myprogram.c -L$SLHALIB/lib -lSLHA
```

The `fcc` script is installed together with the library and automatically adds the necessary libraries for linking with Fortran code.

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 complex 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.

Acknowledgements

I thank J. Guasch for providing a first implementation of some SLHA2 features and introducing useful routines.

References

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

[2] B.C. Allanach et al., hep-ph/0602198.

[3] T. Hahn, hep-ph/0408283.

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