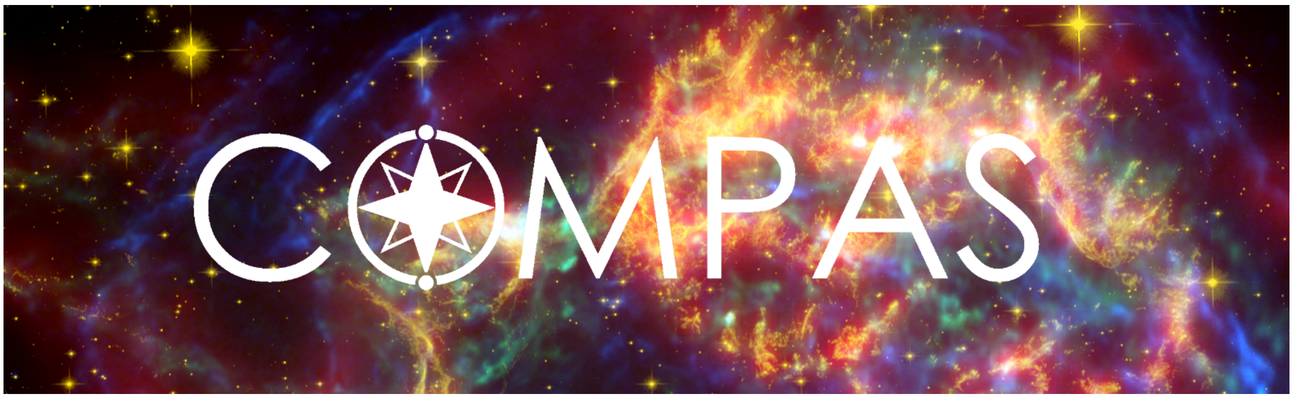
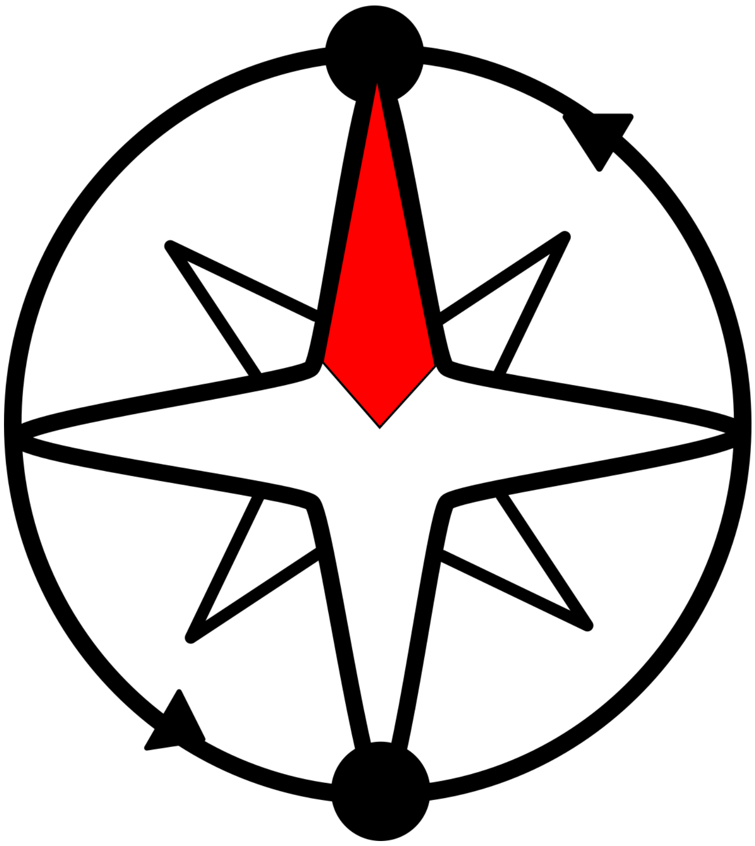


Compact Object Mergers: Population Astrophysics and Statistics

COMPAS is a platform for the exploration of populations of compact binaries formed through isolated binary evolution (Stevenson et al., 2017; Vigna-Gómez et al., 2018; Barrett et al., 2018; Neijssel et al., 2019; Broekgaarden et al., 2019; Stevenson et al., 2019; Chattopadhyay et al., 2020). The COMPAS population synthesis code is flexible, fast and modular, allowing rapid simulation of binary star evolution. The complete COMPAS suite includes the population synthesis code together with a collection of tools for sophisticated statistical treatment of the synthesised populations.



Visit http://compas.science for more information.

Contents

Revision History

Date	Version	Description	Author
2012-2019		Original COMPAS manual	Team COMPAS
1 September 2019	0.1	Initial draft	Jeff Riley
20 September 2019	0.2	Updated compilation requirements	Jeff Riley
1 October 2019	0.3	Added (minimal) CHE documentation	Jeff Riley
21 October 2019	0.4	Added Grids documentation	
		Added Programming Style and Conventions	
		Added Appendices A-E	
		Reformatted for User Guide, Developer Guide etc.	Jeff Riley
18 December 2019	0.5	Updated Grids documentation for kick values	Jeff Riley
20 December 2019	0.6	Updated Appendices A, B, D, E	Jeff Riley
23 January 2020	0.7	Updated Grids documentation for kick values	Jeff Riley
31 March 2020	1.0	Updated for Beta release	Jeff Riley
22 April 2020	1.1	Updated to reflect pre2ndSN -> preSN variable changes	Simon Stevenson
27 April 2020	1.2	Updated to reflect SSE Grid file changes	Jeff Riley

User Guide

Installation instructions for COMPAS and its dependencies are shown in the COMPAS Getting Started guide provided in the COMPAS suite. Please refer to that guide for dependency requirements etc.

COMPAS Input

COMPAS provides wide-ranging functionality and affords users much flexibility in determining how the synthesis and evolution of stars (single or binary) is conducted. Users configure COMPAS functionality and provide initial conditions via the use of program options, described in Appendix A – Program Options. Initial conditions can also be provided via Grid files, described below.

A convenient method of managing the many program options provided by COMPAS (Appendix A – Program Options) is the example Python script provided in the COMPAS suite. The example script provided is pythonSubmitDefault.py – users should copy and modify their copy of the script to match their experimental requirements. Refer to the Getting Started guide for more details.

Grids

Grid functionality allows users to specify a grid of initial values for both Single Star Evolution (SSE) and Binary Star Evolution (BSE).

For SSE, users can supply a text file that contains initial mass values and, optionally, metallicity and supernova kick related values, and COMPAS will evolve individual stars with those initial values (one star per record).

For BSE, users can supply a text file that contains initial mass and metallicity values for the binary constituent stars, as well as the initial separation or orbital period, and eccentricity, of the binary (one binary star per record of initial values).

Grid File Format

Grid files are comma-separated text files, with column headers denoting the meaning of the data in the column (with an exception for SSE Grid files – see below for details).

Grid files may contain comments. Comments are denoted by the hash/pound character ('#'). The hash character and any text following it on the line in which the hash character appears is ignored. The hash character can appear anywhere on a line - if it is the first character then the entire line is a comment and ignored, or it can follow valid characters on a line, in which case the characters before the hash are processed, but the hash character and any text following it is ignored. Blank lines are ignored.

Notwithstanding the exception for SSE Grid files mentioned above, the first non-comment, non-blank line in a Grid file must be the header record. The header record is a comma-separated list of strings that denote the meaning of the data in each of the columns in the file.

Data records follow the header record. Data records, with an exception for SSE data files described below, are comma-separated lists of non-negative floating-point numbers. Any data field that contains a negative number, or characters that do not convert to floating-point numbers, is considered an error and will cause processing of the Grid file to be abandoned – an error message will be displayed.

Data records are expected to contain the same number of columns as the header record. If a data record contains more columns than the header record, data beyond the number of columns in the header record is ignored. If a data record contains fewer columns than the header record, missing data values (by position) are set equal to 0.0 – a warning message will be displayed.

BSE Grid File

Header Record

The BSE Grid file header record must be a comma-separated list of strings taken from the following list (case is not significant):

Header String	**Column meaning**
Mass_1	mass value to be assigned to the primary star, in $M_\odot$
Mass_2	mass value to be assigned to the secondary star, in $M_\odot$
Metallicity_1	metallicity value to be assigned to the primary star
Metallicity_2	metallicity value to be assigned to the secondary star
Separation	separation of the stars – the semi-major axis value to be assigned to the binary, in *AU*
Eccentricity	eccentricity value to be assigned to the binary
Period	orbital period value to be assigned to the binary, in *days*
Kick_Velocity_Random_1	value to be used as the kick velocity magnitude random number, used to draw the kick velocity, for the primary star should it undergo a supernova event. This must be a floating-point number in the range [0.0, 1.0). If this column is present in the grid file, the kick velocity for the primary star, should it undergo a supernova event, will be drawn from the appropriate distribution at the time of the SN event. This column is used in preference to the *Kick_Velocity_1* column if both are present in the grid file.
Kick_Velocity_1	value to be used as the (drawn) kick velocity for the primary star should it undergo a supernova event, in *kms^-1*. If both this column and *Kick_Velocity_Random_1* are present in the grid file, *Kick_Velocity_Random_1* will be used in preference to *Kick_Velocity_1*.
Kick_Theta_1	value to be as the angle between the orbital plane and the 'z' axis of the supernova vector for the primary star should it undergo a supernova event, in *radians*
Kick_Phi_1	value to be used as the angle between 'x' and 'y', both in the orbital plane of the supernova vector, for the primary star should it undergo a supernova event, in *radians*
Kick_Mean_Anomaly_1	value to be used as the mean anomaly at the instant of the supernova for the primary star should it undergo a supernova event – should be uniform in [0, 2pi)
Kick_Velocity_Random_2	value to be used as the kick velocity magnitude random number, used to draw the kick velocity, for the secondary star should it undergo a supernova event. This must be a floating-point number in the range [0.0, 1.0). If this column is present in the grid file, the kick velocity for the secondary star, should it undergo a supernova event, will be drawn from the appropriate distribution at the time of the SN event. This column is used in preference to the *Kick_Velocity_2* column if both are present in the grid file.
Kick_Velocity_2	value to be used as the (drawn) kick velocity for the secondary star should it undergo a supernova event, in *kms^-1*. If both this column and *Kick_Velocity_Random_2* are present in the grid file, *Kick_Velocity_Random_2* will be used in preference to *Kick_Velocity_2*.
Kick_Theta_2	value to be as the angle between the orbital plane and the 'z' axis of the supernova vector for the secondary star should it undergo a supernova event, in *radians*
Kick_Phi_2	value to be used as the angle between 'x' and 'y', both in the orbital plane of the supernova vector, for the secondary star should it undergo a supernova event, in *radians*
Kick_Mean_Anomaly_2	value to be used as the mean anomaly at the instant of the supernova for the secondary star should it undergo a supernova event – should be uniform in [0, 2pi)

All header strings in **bold** in the table above are required in the header record, with the exception of Separation and Period: one of Separation and Period *must* be present, but both *may* be present.

The header strings in *italics* in the table above (the kick-related header strings) are not mandatory. However, if one of the kick-related header strings is present, then *all must* be present.

The order of the columns in the BSE Grid file is not significant.

Data Record

See the general description of data records above.

As for the header record, only one of Separation and Period is required to be present, but both may be present. The period may be used to calculate the separation of the binary. If the separation is present it is used as the value for the semi-major axis of the binary, regardless of whether the period is present (Separation has precedence over Period). If the period is present, but separation is not, the separation is calculated form the masses of the stars and the period given.

Also as for the header record, the kick-related values are not mandatory, but if one of the kick-related values is given, then all must be given.

SSE Grid File

Header Record

The SSE Grid file header record must be a comma-separated list of strings taken from the following list (case is not significant):

Header String	**Column meaning**
Mass	mass value () to be assigned to the star ($M_\odot$)
Metallicity	metallicity value to be assigned to the star
Kick_Velocity_Random	value to be used as the kick velocity magnitude random number, used to draw the kick velocity, for the star should it undergo a supernova event. This must be a floating-point number in the range [0.0, 1.0). If this column is present in the grid file, the kick velocity for the star, should it undergo a supernova event, will be drawn from the appropriate distribution at the time of the SN event. This column is used in preference to the Kick_Velocity column if both are present in the grid file.
Kick_Velocity	value to be used as the (drawn) kick velocity for the star should it undergo a supernova event, in kms^-1. If both this column and Kick_Velocity_Random are present in the grid file, Kick_Velocity_Random will be used in preference to Kick_Velocity.

The SSE Grid file is only required to list Mass values for each star, with Metallicity and Kick values being optional. If the Metallicity column is omitted, the metallicity value assigned to the star is the user-specified value for metallicity via the program options (OPTIONS→Metallicity()).

If the Metallicity column is omitted from the SSE Grid file, the header is optional: if there is only one column of data in the SSE Grid file it is assumed to be the Mass column, and no header is required (though may be present). If the Metallicity column header is present, the Mass column header is required.

The order of the columns in the SSE Grid file is not significant.

Data Record

See the general description of data records above. As for the header record, only Mass is required to be present, but Metallicity may also be present. If the Metallicity is omitted, the metallicity value assigned to the star is the user-specified value for metallicity via the program options (OPTIONS→Metallicity()).

COMPAS Output

Summary and status information during the evolution of stars is written to stdout; how much is written depends upon the value of the *quiet* program option.

Detailed information is written to log files (described below). All COMPAS output files are created inside a container directory, specified by the *output-container* program option. If BSE Detailed Output log files are created (see the *detailedOutput* program option) they will be created inside a containing directory named 'Detailed_Output' within the COMPAS output container directory. Also created in the COMPAS container directory is a filed named 'Run_Details' in which COMPAS records some details of the run (COMPAS version, start time, program option values etc.).

COMPAS defines several standard log files that may be produced depending upon the simulation type (Single Star Evolution (SSE), or Binary Star Evolution (BSE), and the value of various program options. The standard log files are:

- SSE Parameters log file
- BSE System Parameters log file
- BSE Detailed Output log file
- BSE Double Compact Objects log file
- BSE Common Envelopes log file
- BSE Supernovae log file
- BSE Pulsar Evolution log file

Standard Log File Record Specifiers

Each standard log file has an associated log file record specifier that defines what data are to be written to the log files. Each record specifier is a list of known properties that are to be written as the log record for the log file associated with the record specifier. Default record specifiers for each of the standard log files are shown in Appendix E – Default Log File Record Specifications. The standard log file record specifiers can be defined by the user at run-time (see Standard Log File Record Specification below).

When specifying known properties, the property name must be prefixed with the property type. The current list of valid property types available for use is:

- STAR_PROPERTY
- STAR_1_PROPERTY
- STAR_2_PROPERTY
- SUPERNOVA_PROPERTY
- COMPANION_PROPERTY
- BINARY_PROPERTY
- PROGRAM_OPTION

The stellar property types (all types except BINARY_PROPERTY AND PROGRAM_OPTION) must be paired with properties from the stellar property list, the binary property type BINARY_PROPERTY with properties from the binary property list, and the program option type PROGRAM_OPTION with properties from the program option property list.

Standard Log File Record Specification

The standard log file record specifiers can be changed at run-time by supplying a definitions file via the *logfile-definitions* program option.

The syntax of the definitions file is fairly simple. The definitions file is expected to contain zero or more log file record specifications, as explained below.

For the following specification:

`::=`	means "expands to" or "is defined as"
`{ x }`	means (possible) repetition: x may appear zero or more times
`[ x ]`	means x is optional: x may appear, or not
`<name>`	is a term (expression)
`"abc"`	means literal string "abc"
`|`	means "or"
`#`	indicates the start of a comment

Logfile Definitions File specification:

```
<def_file>   ::= { <rec_spec> }

<rec_spec>   ::= <rec_name> <op> "{" { [ <props_list> ] } "}" <spec_delim>

<rec_name>   ::= "SSE_PARMS_REC"        |                      # SSE only
                 "BSE_SYSPARMS_REC"     |                      # BSE only
                 "BSE_DCO_REC"          |                      # BSE only
                 "BSE_SNE_REC"          |                      # BSE only
                 "BSE_CEE_REC"          |                      # BSE only
                 "BSE_PULSARS_REC"      |                      # BSE only
                 "BSE_DETAILED_REC"                            # BSE only

<op>         ::= "=" | "+=" | "-="

<props_list> ::= <prop_spec> [ <prop_delim> <props_list> ]

<prop_spec>  ::= <prop_type> "::" <prop_name> <delim>

<spec_delim> ::= " " | EOL

<prop_delim> ::= "," | <spec_delim>

<prop_type>  ::= "STAR_PROPERTY"       |                       # SSE only
                 "STAR_1_PROPERTY"     |                       # BSE only
                 "STAR_2_PROPERTY"     |                       # BSE only
                 "SUPERNOVA_PROPERTY"  |                       # BSE only
                 "COMPANION_PROPERTY"  |                       # BSE only
                 "BINARY_PROPERTY"     |                       # BSE only
                 "PROGRAM_OPTION"                              # SSE or BSE

<prop_name>  ::= valid property name for specified property type
                 (see definitions in constants.h)
```

The file may contain comments. Comments are denoted by the hash/pound character ('#'). The hash character and any text following it on the line in which the hash character appears is ignored by the parser. The hash character can appear anywhere on a line - if it is the first character then the entire line is a comment and ignored by the parser, or it can follow valid symbols on a line, in which case the symbols before the hash character are parsed and interpreted by the parser.

A log file specification record is initially set to its default value (see Appendix E – Default Log File Record Specifications). The definitions file informs the code as to the modifications to the default values the user wants. This means that the definitions log file is not mandatory, and if the definitions file is not present, or contains no valid record specifiers, the log file record definitions will remain at their default values.

The assignment operator given in a record specification (`<op>` in the file specification above) can be one of "`=`", "`+=`", and "`-=`". The meanings of these are:

> "`=`" means that the record specifier should be assigned the list of properties specified in the braced-list following the "`=`" operator. The value of the record specifier prior to the assignment is discarded, and the new value set as described.
>
> "`+=`" means that the list of properties specified in the braced-list following the "`+=`" operator should be appended to the existing value of the record specifier. Note that the new properties are appended to the existing list, so will appear at the end of the list (properties are printed in the order they appear in the list).
>
> "`-=`" means that the list of properties specified in the braced-list following the "`-=`" operator should be subtracted from the existing value of the record specifier.

Example Log File Definitions File entries:

```
SSE_PARMS_REC = { STAR_PROPERTY::RANDOM_SEED,
                  STAR_PROPERTY::RADIUS, STAR_PROPERTY::MASS,
                  STAR_PROPERTY::LUMINOSITY }


BSE_PULSARS_REC += { STAR_1_PROPERTY::LUMINOSITY
                     STAR_2_PROPERTY::CORE_MASS
                     BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRIME_RSOL
                     COMPANION_PROPERTY::RADIUS }

BSE_PULSARS_REC -= { SUPERNOVA_PROPERTY::TEMPERATURE }

BSE_PULSARS_REC += { PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_NS,
                     BINARY_PROPERTY::ORBITAL_VELOCITY }
```

A full example Log File Record Specifications File is shown in Appendix F – Example Log File Record Specifications File.

The record specifications in the definitions file are processed individually in the sequence they appear in the file, and are cumulative: for record specifications pertaining to the same record name, the output of earlier specifications is input to later specifications.

For each record specification:

- Properties requested to be added to an existing record specification that already exist in that record specification are ignored.   Properties will not appear in a record specification twice.

- Properties requested to be subtracted from an existing record specification that do not exist in that record specification are ignored.

Note that neither of those circumstances will cause a parse error for the definitions file – in both cases the user’s intent is satisfied.

Standard Log File Format

Each standard log file consists three header records followed by data records. Header records and data records are delimiter separated fields, with the delimiter being that specified by the `logfile-delimiter` program option (COMMA, TAB or SPACE), and the fields as specified by the log file record specifier.

The header records for all files are:

Header record 1: Column Data Type Names
Header record 2: Column Units (where applicable)
Header record 3: Column Headings

Column Data Type Names are taken from the set {BOOL, INT, FLOAT, STRING}, where

BOOL	the data value will be a boolean value. Boolean data values will be recorded in the log file in either numerical format (1 or 0, where 1 = TRUE and 0 = FALSE), or string format ('TRUE' or 'FALSE'), depending upon the value of the `print-bool-as-string` program option.
INT	the data value will be an integer number
FLOAT	the data value will be a floating-point number
STRING	the data value will be a text string

Column Units is a string indicating the units of the corresponding data values (e.g. 'Msol*AU^2*yr^-1', 'Msol', 'AU', etc.). The Column Units value may be blank where units are not applicable, or may be one of:

'Count'	the data value is the total of a counted entity
'State'	the data value describes a state (e.g. 'Unbound' state is 'TRUE' or 'FALSE')
'Event'	the data value describes an event status (e.g. 'Simultaneous_RLOF' is 'TRUE')

Column Headings are string labels that describe the corresponding data values. The heading strings for stellar properties of constituent stars of a binary will have appropriate identifiers appended. That is, heading strings for:

STAR_1_PROPERTY::*properties* will have "_1" appended
STAR_2_PROPERTY::*properties* will have "_2" appended
SUPERNOVA_PROPERTY::*properties* will have "_SN" appended
COMPANION_PROPERTY::*properties* will have "_CP" appended

Developer Guide

Team COMPAS welcomes the active involvement of colleagues and others interested in the field in the ongoing development and improvement of the COMPAS software. We hope this Developer Guide helps anyone interested in contributing to the COMPAS software. We expect this guide to be a living document and improve along with the improvements made to the software.

SINGLE STAR EVOLUTION

Class Hierarchy

The main class for single star evolution is the **Star** class.

The Star class is a wrapper that abstracts away the details of the star and the evolution. Internally the Star class maintains a pointer to an object representing the star being evolved, with that object being an instance of one of the following classes:

MS_lte_07
MS_gt_07
CH
HG
FGB
CHeB
EAGB
TPAGB
HeMS
HeHG
HeGB
HeWD
COWD
ONeWD
NS
BH
MR

which track the phases from Hurley et al. 2000, with the exception of the CH class for Chemically Homogeneous stars, which is not described in Hurley et al. 2000.

Three other SSE classes are defined:

BaseStar
MainSequence
GiantBranch

These extra classes are included to allow inheritance of common functionality.

The BaseStar class is the main class for the underlying star object held by the Star class. The BaseStar class defines all member variables, and many member functions that provide common functionality. Similarly, the MainSequence and GiantBranch classes provide repositories for common functionality for main sequence and giant branch stars respectively.

The inheritance chain follows the phases described in Hurley et al. 2000 (again, with the exception of the CH, not described by Hurley et al. 2000), and is as follows:

```
Star
  BaseStar → MainSequence → ( MS_lte_07   )
                            ( MS_gt_07    ) → CH
                            ( GiantBranch ) → HG → FGB → CheB → EAGB →

     → TPAGB → HeMS → HeHG → HeGB → HeWD → COWD → OneWD → NS → BH → MR
```

CH (Chemically Homogeneous) stars inherit from MS_gt_07 because (in this implementation) they are just (large) main sequence stars that have a static radius.

HG (Hertzsprung Gap) stars inherit from the GiantBranch because they share the Giant Branch Parameters described in Hurley et al. 2000, section 5.2.

Each class in the inheritance chain has its own set of member functions that calculate various attributes of the star according to the phase the class represents (using the equations and parameters from Hurley et al. 2000 where applicable).

Evolution Model

The stellar evolution model is driven by the Evolve() function in the Star class which evolves the star through its entire lifetime by doing the following:

```
DO:

    1. calculate time step
           calculate the giant branch parameters (as necessary)
           calculate the timescales
           choose time step

    2. save the state of the underlying star object

    3. DO:
           a) evolve a single time step

           b) if too much change
                  revert to the saved state
                  reduce the size of the time step

       UNTIL timestep not reduced

    4. resolve any mass loss
           a) update initial mass (mass0)
           b) update age after mass loss
           c) apply mass transfer rejuvenation factor

    5. evolve to the next stellar type if necessary

WHILE the underlying star object is not one of: { HeWD, COWD, ONeWD, NS, BH, MR }
```

Evolving the star through a single time step (step 3a above) is driven by the UpdateAttributesAndAgeOneTimestep() function in the BaseStar class which does the following:

```
1.  check if the star should be a massless remnant

2.  check if the star is a supernova

if evolution on the phase should be performed
3.      evolve the star on the phase – update stellar attributes
4.      check if the star should evolve off the current phase to a different stellar type
else
5.  ready the star for the next time step
```

Evolving the star on its current phase, and off the current phase and preparing to evolve to a different stellar type, is handled by two functions in the BaseStar class: EvolveOnPhase() and ResolveEndOfPhase().

The EvolveOnPhase() function does the following:

1. Calculate Tau

2. Calculate CO Core Mass
3. Calculate Core Mass
4. Calculate He Core Mass

5. Calculate Luminosity
6. Calculate Radius

7. Calculate Envelope Mass

8. Calculate Perturbation Mu
9. Perturb Luminosity and Radius

10. Calculate Temperature

11. Resolve possible envelope loss

Each of the calculations in the EvolveOnPhase() function is performed in the context of the star evolving on its current phase. Each of the classes implements their own version of the calculations (via member functions) – some may inherit functions from the inheritance chain, while others might just return the value unchanged if the calculation is not relevant to their stellar type.

The ResolveEndOfPhase() function does the following:

1. Resolve possible envelope loss

2. Calculate Tau

3. Calculate CO Core Mass
4. Calculate Core Mass
5. Calculate He Core Mass

6. Calculate Luminosity
7. Calculate Radius

8. Calculate Envelope Mass

9. Calculate Perturbation Mu
10. Perturb Luminosity and Radius

11. Calculate Temperature

12. Evolve star to next phase

Each of the calculations in the ResolveEndOfPhase() function is performed in the context of the star evolving off its current phase to the next phase.

The remainder of the code (in general terms) supports these main driver functions.

BINARY STAR EVOLUTION

Class Hierarchy

The main class for single star evolution is the **BinaryStar** class. The BinaryStar class is a wrapper that abstracts away the details of the binary star and the evolution. Internally the BinaryStar class maintains a pointer to an object representing the binary star being evolved, with that object being an instance of the BaseBinaryStar class.

The BaseBinaryStar class is the main class for the underlying binary star object held by the BinaryStar class. The BaseBinaryStar class defines all member variables that pertain specifically to a binary star, and many member functions that provide binary-star specific functionality. Internally the BaseBinaryStar class maintains pointers to the two BinaryConstituentStar class objects that constitute the binary star.

The BinaryConstituentStar class inherits from the Star class, so objects instantiated from the BinaryConstituentStar class inherit the characteristics of the Star class, particularly the stellar evolution model. The BinaryConstituentStar class defines member variables and functions that pertain specifically to a constituent star of a binary system but that do not (generally) pertain to single stars that are not part of a binary system (there are some functions that are defined in the BaseStar class and its derived classes that deal with binary star attributes and behaviour – in some cases the stellar attributes that are required to make these calculations reside in the BaseStar class so it is easier and cleaner to define the functions there).

The inheritance chain is as follows:

```
BinaryStar → BaseBinaryStar

(Star → )       BinaryConstituentStar (star1)
(Star → )       BinaryConstituentStar (star2)
```

Evolution Model

The binary evolution model is driven by the Evolve() function in the BaseBinaryStar class which evolves the star through its entire lifetime by doing the following:

```
if (touching OR secondary too small)
    STOP = true
else
    calculate initial time step
    STOP = false

DO WHILE !STOP AND !max iterations:

    evolve a single time step
        evolve each constituent star a single time step (see SSE evolution)

    if  (unbound OR touching OR Massless Remnant)
        STOP = true
    else
        evaluate the binary
            calculate lambdas if necessary
            calculate zetas if necessary

            calculate mass transfer
            calculate winds mass loss

            if common envelope
                resolve common envelope
            else if supernova
                resolve supernova
            else
                resolve mass changes

            evaluate supernovae
            resolve tides
            calculate total energy and angular momentum
            update magnetic field and spin: both constituent stars

        if (unbound OR touching OR merger)
            STOP = true
        else
            if NS+BH
                resolve coalescence

                if AIS exploratory phase
                    calculate DCO Hit

                STOP = true
            else
                if (WD+WD OR max time)
                    STOP = true
                else
                    if NOT max iterations
                        calculate new time step
```

OBJECT IDENTIFIERS

All objects (instantiations of a class) are assigned unique object identifiers of type OBJECT_ID (unsigned long int - see constants.h for the typedef). In the original COMPAS code, all binary star objects were assigned unique object ids – this is just an extension of that so that all objects created in the COMPAS code are assigned unique ids. The purpose of the unique object id is to aid in object tracking and debugging.

As well as unique object ids, all objects are assigned an object type (of type OBJECT_TYPE – see constants.h for the enum class declaring OBJECT_TYPE), and a stellar type where applicable (of type STELLAR_TYPE – see constants.h for the enum class declaring STELLAR_TYPE).

Objects should expose the following functions:

```
OBJECT_ID       ObjectId() const      { return m_ObjectId; }
OBJECT_TYPE     ObjectType() const    { return m_ObjectType; }
STELLAR_TYPE    StellarType() const   { return m_StellarType; }
```

If any of the functions are not applicable to the object, then they must return "*::NONE (all objects should implement ObjectId() correctly).

Any object that uses the Errors service (i.e. the SHOW_* macros) ***must*** expose these functions: the functions are called by the SHOW_* macros (the Errors service is described later in this document).

SERVICES

A number of services have been provided to help simplify the code. These are:

- Program Options
- Random Numbers
- Logging and Debugging
- Error Handling

The code for each service is encapsulated in a singleton object (an instantiation of the relevant class). The singleton design pattern allows the definition of a class that can only be instantiated once, and that instance effectively exists as a global object available to all the code without having to be passed around as a parameter. Singletons are a little anti-OO, but provided they are used judiciously are not necessarily a bad thing, and can be very useful in certain circumstances.

PROGRAM OPTIONS

A Program Options service is provided encapsulated in a singleton object (an instantiation of the Options class).

The Options class member variables are private, and public getter functions have been created for the program options currently used in the code.

The Options service can be accessed by referring to the Options::Instance() object. For example, to retrieve the value of the "quiet" program option, call the Quiet() getter function:

```
bool quiet = Options::Instance()→Quiet();
```

Since that could become unwieldy, there is a convenience macro to access the Options service. The macro just defines "OPTIONS" as "Options::Instance()", so retrieving the value of the "quiet" program option can be written as:

```
bool quiet = OPTIONS→Quiet();
```

The Options service must be initialised before use. Initialise the Options service by calling the Initialise() function:

```
COMMANDLINE_STATUS programStatus = OPTIONS->Initialise(argc, argv);
```

(see constants.h for details of the COMMANDLINE_STATUS type)

Random Numbers

A Random Number service is provided, with the gsl Random Number Generator encapsulated in a singleton object (an instantiation of the Rand class).

The Rand class member variables are private, and public functions have been created for random number functionality required by the code.

The Rand service can be accessed by referring to the Rand::Instance() object. For example, to generate a uniform random floating point number in the range [0, 1), call the Random() function:

```
double u = Rand::Instance()→Random();
```

Since that could become unwieldy, there is a convenience macro to access the Rand service. The macro just defines “RAND” as “Rand::Instance()”, so calling the Random() function can be written as:

```
double u = RAND→Random();
```

The Rand service must be initialised before use. Initialise the Rand service by calling the Initialise() function:

RAND->Initialise();

Dynamically allocated memory associated with the gsl random number generator should be returned to the system by calling the Free() function:

RAND→Free();

before exiting the program.

The Rand service provides the following public member functions:

void Initialise()

Initialises the gsl random number generator. If the environment variable GSL_RNG_SEED exists, the gsl random number generator is seeded with the value of the environment variable, otherwise it is seeded with the current time.

void Free()

Frees any dynamically allocated memory.

unsigned long int Seed(const unsigned long p_Seed)

Sets the seed for the gsl random number generator to p_Seed. The return value is the seed.

unsigned long int DefaultSeed()

Returns the gsl default seed (gsl_rng_default_seed)

double Random(void)

Returns a random floating point number uniformly distributed in the range [0.0, 1.0)

double Random(const double p_Lower, const double p_Upper)

Returns a random floating point number uniformly distributed in the range [p_Lower, p_Upper), where p_Lower <= p_Upper.

(p_Lower and p_Upper will be swapped if p_Lower > p_Upper as passed)

double RandomGaussian(const double p_Sigma)

Returns a Gaussian random variate, with mean 0.0 and standard deviation p_Sigma

int RandomInt(const int p_Lower, const int p_Upper)

Returns a random integer number uniformly distributed in the range [p_Lower, p_Upper), where p_Lower <= p_Upper.

(p_Lower and p_Upper will be swapped if p_Lower > p_Upper as passed)

int RandomInt(const int p_Upper)

Returns a random integer number uniformly distributed in the range [0, p_Upper), where 0 <= p_Upper. Returns 0 if p_Upper < 0.

LOGGING & DEBUGGING

A logging and debugging service is provided encapsulated in a singleton object (an instantiation of the Log class).

The logging functionality was first implemented when the Single Star Evolution code was refactored, and the base-level of logging was sufficient for the needs of the SSE code. Refactoring the Binary Star Evolution code highlighted the need for expanded logging functionality. To provide for the logging needs of the BSE code, new functionality was added almost as a wrapper around the original, base-level logging functionality. Some of the original base-level logging functionality has almost been rendered redundant by the new functionality implemented for BSE code, but it remains (almost) in its entirety because it may still be useful in some circumstances.

When the base-level logging functionality was created, debugging functionality was also provided, as well as a set of macros to make debugging and the issuing of warning messages easier. A set of logging macros was also provided to make logging easier. The debug macros are still useful, and their use is encouraged (rather than inserting print statements using std::cout or std::cerr).

When the BSE code was refactored, some rudimentary error handling functionality was also provided in the form of the Errors service - an attempt at making error handling easier. Some of the functionality provided by the Errors service supersedes the DBG_WARN* macros provided as part of the Log class, but the DBG_WARN* macros are still useful in some circumstances (and in fact are still used in various places in the code). The LOG* macros are somewhat less useful, but remain in case the original base-level logging functionality (that which underlies the expanded logging functionality) is used in the future (as mentioned above, it could still be useful in some circumstances). The Errors service is described later in this document.

The expanded logging functionality introduces Standard Log Files - described later in this document.

Base-Level Logging

The Log class member variables are private, and public functions have been created for logging and debugging functionality required by the code.

The Log service can be accessed by referring to the Log::Instance() object. For example, to stop the logging service, call the Stop() function:

```
Log::Instance()→Stop();
```

Since that could become unwieldy, there is a convenience macro to access the Log service. The macro just defines “LOGGING” as “Log::Instance()”, so calling the Stop() function can be written as:

```
LOGGING→Stop();
```

The Log service must be initialised before logging and debugging functionality can be used. Initialise logging by calling the Start() function:

```
LOGGING→Start(
        outputPath,          - location of logfiles
        containerName,       - directory to be created at p_LogBasePath to hold all log files
        logfilePrefix,       - prefix for logfile names (can be blank)
        logLevel,            - logging level (integer) (see below)
        logClasses,          - array of enabled logging classes (strings) (see below)
        debugLevel,          - debug level (integer) (see below)
        debugClasses,        - array of enabled debug classes (strings) (see below)
        debugToLogfile,      - flag (boolean) indicating whether debug statements should also be
                               written to log file
        errorsToLogfile,     - flag (boolean) indicating whether error messages should also be
                               written to log file
        delimiter            - string (usually single character) to be used as the default field
                               delimiter in log file records
)
```

Start() returns nothing (void function).

The Log service should be stopped before exiting the program – this ensures all open log files are flushed to disk and closed properly. Stop logging by calling the Stop() function:

```
LOGGING→Stop(
        objectStats          - { number of objects (stars or binaries) requested, count created }
)
```

Stop() flushes any closes any open log files.
Stop() returns nothing (void function).

The Log service provides the following public member functions:

```
    void Start(
        outputPath,          - location of logfiles- the directory in which log files will be created
        containerName,       - directory to be created at p_LogBasePath to hold all log files
        logfilePrefix,       - prefix for logfile names (can be blank)
        logLevel,            - logging level (integer) (see below)
        logClasses,          - array of enabled logging classes (strings) (see below)
        debugLevel,          - debug level (integer) (see below)
        debugClasses,        - array of enabled debug classes (strings) (see below)
        debugToLogfile,      - flag (boolean) indicating whether debug statements should also be
                               written to log file
        errorsToLogfile,     - flag (boolean) indicating whether error messages should also be
                               written to log file
        delimiter            - string (usually single character) to be used as the default field
                               delimiter in log file records
    )
```

Initialises the logging and debugging service. Logging parameters are set per the program options specified (using default values if no options are specified by the user). The log file container directory is created. If a directory with the name as given by the *containerName* parameter already exists, a version number will be appended to the directory name. The Run_Details file is created within the logfile container directory. Log files to which debug statements and error messages will be created and opened if required.

void Stop(

objectStats	- { number of objects (stars or binaries) requested, count created }

)

Stops the logging and debugging service. All open log files are flushed to disk and closed (including and Standard Log Files open - see description of Standard Log Files later in this document). The Run_Details file is populated and closed.

bool Enabled()

Returns a boolean indicating whether the Log service is enabled – true indicates the Log service is enable and available; false indicates the Log service is not enable and so not available.

int Open(

logFileName,	- the name of the log file to be created and opened. This should be the filename only – the path, prefix and extensions are added by the logging service. If the file already exists, the logging service will append a version number to the name if necessary (see *append* parameter below).
append,	- flag (boolean) indicating whether the file should be opened in append mode (i.e. existing data is preserved) and new records written to the file appended, or whether a new file should be opened (with version number if necessary).
timeStamps,	- flag (boolean) indicating whether timestamps should be written with each log file record.
labels	- flag (boolean) indicating whether a label should be written with each log record. This is useful when different types of logging data is being written to the same log file file.
delimiter	- (optional) string (usually single character) to be used as the field delimiter in this log file. If *delimiter* is not provided the default delimiter is used (as parameter to Start()).

)

Opens a log file. If the append parameter is true and a file name *logFilename* exists, the existing file will be opened and the existing contents retained, otherwise a new file will be created and opened (not a Standard Log File - see description of Standard Log Files later in this document).

The log file container directory is created at the path specified by the *outputPath* parameter passed to the Start() function. New log files are created in the logfile container directory. BSE

Detailed log files are created in the Detailed_Output directory, which is created in the log file container directory if required.

The filename is prefixed by the *logfilePrefix* parameter passed to the Start() function.

The file extension is based on the *delimiter* parameter passed to the Start() function: if the delimiter is SPACE, the file extension is ".txt"; if the delimiter is TAB the file extension is".tsv"; if the delimiter is COMMA the file extension is ".csv".

If a file with the name as given by the *logFilename* parameter already exists, and the *append* parameter is false, a version number will be appended to the filename before the extension (this functionality is largely redundant since the implementation of the log file container directory).

The log file identifier (integer) is returned to the caller - a value of -1 indicates the log file was not opened successfully. Multiple log files can be open simultaneously – referenced by the identifier returned.

bool Close(

logFileId,	- the identifier of the log file to be closed (as returned by Open())

)

Closes the log file specified by the *logFileId* parameter. If the log file specified by the *logFileId* parameter is open, it is flushed to disk and closed. The function returns a boolean indicating whether the file was closed successfully.

bool Write(

logFileId,	- the identifier of the log file to be written
logClass,	- string specifying the log class to be associated with the record to be written. Can be blank.
logLevel,	- integer specifying the log level to be associated with the record to be written.
logString,	- the string to be written to the log file.

)

Writes an unformatted record to the specified log file. If the Log service is enabled and the specified log file is active, and the log class and log level passed are enabled (see discussion of log classes and levels), the string is written to the file.

The function returns a boolean indicating whether the record was written successfully. If an error occurred the log file will be disabled.

bool Put(

logFileId,	- the identifier of the log file to be written
logClass,	- string specifying the log class to be associated with the record to be written. Can be blank.
logLevel,	- integer specifying the log level to be associated with the record to be written.
logString,	- the string to be written to the log file.

)

Writes a minimally formatted record to the specified log file. If the Log service is enabled and the specified log file is active, and the log class and log level passed are enabled (see discussion of log classes and levels), the string is written to the file.

If labels are enabled for the log file, a label will be prepended to the record. The label text will be the *logClass* parameter.

If timestamps are enabled for the log file, a formatted timestamp is prepended to the record. The timestamp format is *yyyymmdd hh:mm:ss*.

The function returns a boolean indicating whether the record was written successfully. If an error occurred the log file will be disabled.

bool Debug(

debugClass,	- string specifying the debug class to be associated with the record to be written. Can be blank.
debugLevel,	- integer specifying the debug level to be associated with the record to be written.
debugString,	- the string to be written to stdout (and optionally to file)

)

Writes *debugString* to stdout and, if logging is active and so configured (via program option debug-to-file), writes *debugString* to the debug log file.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the debug log file, the log file will be disabled.

bool DebugWait(

debugClass,	- string specifying the debug class to be associated with the record to be written. Can be blank.
debugLevel,	- integer specifying the debug level to be associated with the record to be written.
debugString,	- the string to be written to stdout (and optionally to file)

)

Writes *debugString* to stdout and, if logging is active and so configured (via program option debug-to-file), writes *debugString* to the debug log file, then waits for user input.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the debug log file, the log file will be disabled.

bool Error(

errorString,	- the string to be written to stdout (and optionally to file)

)

Writes *errorString* to stdout and, if logging is active and so configured (via program option errors-to-file), writes *errorString* to the error log file, then waits for user input.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the error log file, the log file will be disabled.

void Squawk(

squawkString,	- the string to be written to stderr

)

Writes *squawkString* to stderr.

void Say(

sayClass,	- string specifying the log class to be associated with the record to be written. Can be blank.
sayLevel,	- integer specifying the log level to be associated with the record to be written.
sayString,	- the string to be written to stdout

)

Writes *sayString* to stdout.

The filename to which debug records are written when Start() parameter “debugToLogfile” is true is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR. Currently the name is ‘Debug_Log’.

Extended Logging

The Logging service was extended to support standard log files for Binary Star Evolution (SSE also uses the extended logging). The standard log files defined are:

- SSE Parameters log file
- BSE System Parameters log file
- BSE Detailed Output log file
- BSE Double Compact Objects log file
- BSE Common Envelopes log file
- BSE Supernovae log file
- BSE Pulsar Evolution log file

The Logging service maintains information about each of the standard log files, and will handle creating, opening, writing and closing the files. For each execution of the COMPAS program that evolves binary stars, one (and only one) of each of the log file listed above will be created, except for the Detailed Output log in which case there will be one log file created for each binary star evolved.

The Log service provides the following public member functions specifically for managing standard log files:

void LogSingleStarParameters(Star, Id**)**
void LogBinarySystemParameters(Binary**)**
void LogDetailedOutput(Binary, Id**)**
void LogDoubleCompactObject(Binary**)**
void LogCommonEnvelope(Binary**)**
void LogSupernovaDetails(Binary**)**
void LogPulsarEvolutionParameters(Binary**)**

Each of the BSE functions is passed a pointer to the binary star for which details are to be logged, and in the case of the Detailed Output log file, an integer identifier (typically the loop index of the binary star) that is appended to the log file name.

The SSE function is passed a pointer to the single star for which details are to be logged, and an integer identifier (typically the loop index of the star) that is appended to the log file name.

Each of the functions listed above will, if necessary, create and open the appropriate log file. Internally the Log service opens (creates first if necessary) once at first use, and keeps the files open for the life of the program.

The Log service provides a further two functions to manage standard log files:

bool CloseStandardFile(LogFile**)**

Flushes and closes the specified standard log file. The function returns a boolean indicating whether the log file was closed successfully.

bool CloseAllStandardFiles()

Flushes and closes all currently open standard log files. The function returns a boolean I indicating whether all standard log files were closed successfully.

Standard log file names are supplied via program options, with default values declared in constants.h.

Logging & Debugging Macros

Logging Macros

The following macros are provide for logging:

LOG(id, …)
Writes log record to log file specified by "id". Use:

LOG(id, string)	writes "string" to log file specified by "id"
LOG(id, level, string)	writes "string" to log file specified by "id" if "level" is <= "id" in Start()

LOG(id, class, level, string)	writes "string" to log file specified by "id"
	if "class" is in "logClasses" in Start() and
	if "level" is <= "logLevel" in Start()

default "class" is ""; default "level" is 0

Examples:

```
LOG(SSEfileId, "This is a log record");
LOG(OutputFile2Id, "The value of x is " << x << " km");
LOG(MyLogfileId, 2, "Log string");
LOG(SSEfileId, "CHeB", 4, "This is a CHeB only log record");
```

LOG_ID(id, …)
Writes log record prepended with calling function name to log file. Use:

LOG_ID(id)	writes name of calling function to log file specified by “id”
LOG_ID(id, string)	writes “string” prepended with name of calling function to log file specified by “id”
LOG_ID(id, level, string)	writes “string” prepended with name of calling function to log file specified by “id” if “level” is <= “logLevel” in Start()

LOG_ID(id, class, level, string)	writes “string” prepended with name of calling function to log file specified by “id”
	if “class” is in “logClasses” in Start() and
	if “level” is <= “logLevel” in Start()

default “class” is “”; default “level” is 0

Examples:

```
LOG_ID(Outf1Id)
LOG_ID(Outf2Id, “This is a log record”);
LOG_ID(MyLogfileId, “The value of x is “ << x << “ km”);
LOG_ID(OutputFile2Id, 2, “Log string”);
LOG_ID(CHeBfileId, “CHeB”, 4, “This is a CHeB only log record”);
```

LOG_IF(id, cond, …)
Writes log record to log file if the condition given by “cond” is met. Use:

LOG_IF(id, cond, string)	writes “string” to log file specified by “id” if “cond” is true

LOG_IF(id, cond, level, string)	writes “string” to log file specified by “id”
	if “cond” is true and
	if “level” is <= “logLevel” in Start()

LOG_IF(id, cond, class, level, string)	writes “string” to log file specified by “id”
	if “cond” is true and
	if “class” is in “logClasses” in Start() and
	if “level” is <= “logLevel” in Start()

“cond” is any logical statement and is required; default “class” is “”; default “level” is 0

Examples:

```
LOG_IF(MyLogfileId, a > 1.0, “This is a log record”);
LOG(SSEfileId, (b == c && a > x), “The value of x is “ << x << “ km”);
LOG(CHeBfileId, flag, 2, “Log string”);
LOG(SSEfileId, (x >= y), “CHeB”, 4, “This is a CHeB only log record”);
```

LOG_ID_IF(id, ...)
Writes log record prepended with calling function name to log file if the condition given by "cond" is met. Use: see LOG_ID(id, …) and LOG_IF(id, cond, …) above.

The logging macros described above are provided in a verbose variant. The verbose macros function the same way as their non-verbose counterparts, with the added functionality that the log records written to the log file will be reflected on stdout as well. The verbose logging macros are:

LOGV(id, ...)
LOGV_ID(id, ...)
LOGV_IF(id, cond, ...)
LOGV_ID_IF(id, cond, …)

A further four macros are provided that allow writing directly to stdout rather than the log file. These are:

SAY(…)
SAY_ID(…)
SAY_IF(cond, …)
SAY_ID_IF(cond, ...)

The SAY macros function the same way as their LOG counterparts, but write directly to stdout instead of the log file. The SAY macros honour the logging classes and level.

Debugging Macros

A similar set of macros is also provided for debugging purposes.

The debugging macros write directly to stdout rather than the log file, but their output can also be written to the log file if desired (see the debugToLogfile parameter of Start(), and the debug-to-file program option described above).

A major difference between the logging macros and the debugging macros is that the debugging macros can be defined away. The debugging macro definitions are enclosed in an #ifdef enclosure, and are only present in the source code if #DEBUG is defined. This means that if #DEBUG is not defined (#undef), all debugging statements using the debugging macros will be removed from the source code by the preprocessor before the source is compiled. Un-defining #DEBUG not only prevents bloat of unused code in the executable, it improves performance. Many of the functions in the code are called hundreds of thousands, if not millions, of times as the stellar evolution proceeds. Even if the debugging classes and debugging level are set so that no debug statement is displayed, just checking the debugging level every time a function is called increases the run-time of the program. The suggested us is to enable the debugging macros (#define DEBUG) while developing new code, and disable them (#undef DEBUG) to produce a production version of the executable.

The debugging macros provided are:

DBG(…)	analgous to the LOG(…) macro
DBG_ID(…)	analgous to the LOG_ID(…) macro
DBG_IF(cond, …)	analgous to the LOG_IF(…) macro
DBG_ID_IF(cond, …)	analgous to the LOG_ID_IF(…) macro

Two further debugging macros are provided:

DBG_WAIT(…)
DBG_WAIT_IF(cond, …)

The DBG_WAIT macros function in the same way as their non-wait counterparts (DBG(…) and DBG_IF(cond, …) with the added functionality that they will pause execution of the program and wait for user input before proceeding.

A set of macros for printing warning message is also provided. These are the DBG_WARN macros:

DBG_WARN(...)	analgous to the LOG(…) macro
DBG_WARN_ID(...)	analgous to the LOG_ID(…) macro
DBG_WARN_IF(...)	analgous to the LOG_IF(…) macro
DBG_WARN_ID_IF(...)	analgous to the LOG_ID_IF(…) macro

The DBG_WARN macros write to stdout via the SAY macro, so honour the logging classes and level, and are not written to the debug or errors files.

Note that the "id" parameter of the "LOG" macros (to specify the logfileId) is not required for the DBG macros (the filename to which debug records are written is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR).

ERROR HANDLING

An error handling service is provided encapsulated in a singleton object (an instantiation of the Errors class).

The Errors service provides global error handling functionality. Following is a brief description of the Errors service (full documentation coming soon...):

Errors are defined in the error catalog in constants.h (see ERROR_CATALOG). It could be useful to move the catalog to a file so it can be changed without changing the code, or even have multiple catalogs provided for internationalisation – a task for later.

Errors defined in the error catalog have a scope and message text. The scope is used to determine when/if an error should be printed.

The current values for scope are:

NEVER	the error will not be printed
ALWAYS	the error will always be printed
FIRST	the error will be printed only on the first time it is encountered anywhere in the program
FIRST_IN_OBJECT_TYPE	the error will be printed only on the first time it is encountered anywhere in objects of the same type (e.g. Binary Star objects)
FIRST_IN_STELLAR_TYPE	the error will be printed only on the first time it is encountered anywhere in objects of the same stellar type (e.g. HeWD Star obejcts)
FIRST_IN_OBJECT_ID	the error will be printed only on the first time it is encountered anywhere in an object instance
FIRST_IN_FUNCTION	the error will be printed only on the first time it is encountered anywhere in the same function of an object instance (i.e. will print more than once if encountered in the same function name in different objects)

The Errors service provides methods to print both warnings and errors - essentially the same thing, but warning messages are prefixed with "WARNING:", whereas error messages are prefixed with "ERROR:".

Errors and warnings are printed by using the macros defined in ErrorsMacros.h. They are:

Error macros:

SHOW_ERROR(error_number)
Prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_ERROR(error_number, error_string)
Prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

SHOW_ERROR_IF(cond, error_number)
If "cond" is TRUE, prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_ERROR_IF(cond, error_number, error_string)
If "cond" is TRUE, prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

Warning macros:

SHOW_WARN(error_number)
Prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_WARN(error_number, error_string)
Prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

SHOW_WARN_IF(cond, error_number)
If "cond" is TRUE, prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_WARN_IF(cond, error_number, error_string)
If "cond" is TRUE, prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

Error and warning message always contain:

> The object id of the calling object
> The object type of the calling object
> The stellar type of the calling object (will be "NONE" if the calling object is not a star-type object)
> The function name of the calling function

Any object that uses the Errors service (i.e. the SHOW_* macros) must expose the following functions:

```
OBJECT_ID        ObjectId() const      { return m_ObjectId; }
OBJECT_TYPE      ObjectType() const    { return m_ObjectType; }
STELLAR_TYPE     StellarType() const   { return m_StellarType; }
```

These functions are called by the SHOW_* macros. If any of the functions are not applicable to the object, then they must return "*::NONE (all objects should implement ObjectId() correctly).

The filename to which error records are written when Start() parameter “errorsToLogfile” is true is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR. Currently the name is ‘Error_Log’.

FLOATING-POINT COMPARISONS

Floating-point comparisons are inherently problematic. Testing floating-point numbers for equality, or even inequality, is fraught with problems due to the internal representation of floating-point numbers: floating-point numbers are stored with a fixed number of binary digits, which limits their precision and accuracy. The problems with floating-point comparisons are even more evident if one or both of the numbers being compared are the results of (perhaps several) floating-point operations (rather than comparing constants).

To avoid the problems associated with floating-point comparisons it is (almost always) better to do any such comparisons with a tolerance rather than an absolute comparison. To this end, a floating-point comparison function has been provided, and (almost all of) the floating-point comparisons in the code have been changed to use that function. The function uses both an absolute tolerance and a relative tolerance, which are both declared in constants.h. Whether the function uses a tolerance or not can be changed by #define-ing or #undef-ing the "COMPARE_WITH_TOLERANCE" flag in constants.h (so the change is a compile-time change, not run-time).

The compare function is defined in utils.h and is implemented as follows:

```
static int Compare(const double p_X, const double p_Y) {
#ifdef COMPARE_WITH_TOLERANCE
   return (fabs(p_X - p_Y) <= max(FLOAT_TOLERANCE_ABSOLUTE,
                                  FLOAT_TOLERANCE_RELATIVE *
                                  max(fabs(p_X),
                                      fabs(p_Y)))) ? 0 : (p_X < p_Y ? -1 : 1);
#else
    return (p_X == p_Y) ? 0 : (p_X < p_Y ? -1 : 1);
#endif
```

If COMPARE_WITH_TOLERANCE is defined, p_X and p_Y are compared with tolerance values, whereas if COMPARE_WITH_TOLERANCE is not defined the comparison is an absolute comparison.

The function returns an integer indicating the result of the comparison:

-1	indicates that p_X is considered to be less than p_Y
0	indicates p_X and p_Y are considered to be equal
+1	indicates that p_X is considered to be greater than p_Y

The comparison is done using both an absolute tolerance and a relative tolerance. The tolerances can be defined to be the same number, or different numbers. If the relative tolerance is defined as 0.0, the comparison is done using the absolute tolerance only, and if the absolute tolerance is defined as 0.0 the comparison is done with the relative tolerance only.

Absolute tolerances are generally more effective when the numbers being compared are small – so using an absolute tolerance of (say) 0.0000005 is generally effective when comparing single-digit numbers (or so), but is less effective when comparing numbers in the thousands or millions. For comparisons of larger numbers a relative tolerance is generally more effective (the actual tolerance is wider because the relative tolerance is multiplied by the larger absolute value of the numbers being compared).

There is a little overhead in the comparisons even when the tolerance comparison is disabled, but it shouldn't be prohibitive.

CONSTANTS FILE – constants.h

As well as plain constant values, many distribution and prescription identifiers are declared in constants.h. These are mostly declared as enum classes, with each enum class having a corresponding map of labels. The benefit is that the values of a particular (e.g.) prescription are limited to the values declared in the enum class, rather than any integer value, so the compiler will complain if an incorrect value is inadvertently used to reference that prescription.

For example, the Common Envelope Lambda Prescriptions are declared in constants.h thus:

```
enum class CE_LAMBDA_PRESCRIPTION: int {
     FIXED, LOVERIDGE, NANJING, KRUCKOW, DEWI
};

const std::unordered_map<CE_LAMBDA_PRESCRIPTION, std::string>
CE_LAMBDA_PRESCRIPTION_LABEL = {
    { CE_LAMBDA_PRESCRIPTION::FIXED,     "LAMBDA_FIXED" },
    { CE_LAMBDA_PRESCRIPTION::LOVERIDGE, "LAMBDA_LOVERIDGE" },
    { CE_LAMBDA_PRESCRIPTION::NANJING,   "LAMBDA_NANJING" },
    { CE_LAMBDA_PRESCRIPTION::KRUCKOW,   "LAMBDA_KRUCKOW" },
    { CE_LAMBDA_PRESCRIPTION::DEWI,      "LAMBDA_DEWI" }
};
```

Note that the values allowed for variables of type `CE_LAMBDA_PRESCRIPTION` are limited to `FIXED`, `LOVERIDGE`, `NANJING`, `KRUCKOW` and `DEWI` – anything else will cause a compiler error.

The unordered map `CE_LAMBDA_PRESCRIPTION_LABEL` is indexed by `CE_LAMBDA_PRESCRIPTION` and declares a string label for each `CE_LAMBDA_PRESCRIPTION`. The strings declared in `CE_LAMBDA_PRESCRIPTION_LABEL` are used by the Options service to match user input to the required `CE_LAMBDA_PRESCRIPTION`. These strings can also be used if an English description of the value of a variable is required: instead of just printing an integer value that maps to a `CE_LAMBDA_PRESCRIPTION`, the string label associated with the prescription can be printed.

Stellar types are also declared in constants.h via an enum class and associate label map. This allows stellar types to be referenced using symbolic names rather than an ordinal number. The stellar types enum class is STELLAR_TYPE, and is declared as:

```
enum class STELLAR_TYPE: int {
    MS_LTE_07,
    MS_GT_07,
    HERTZSPRUNG_GAP,
    FIRST_GIANT_BRANCH,
    CORE_HELIUM_BURNING,
    EARLY_ASYMPTOTIC_GIANT_BRANCH,
    THERMALLY_PULSING_ASYMPTOTIC_GIANT_BRANCH,
    NAKED_HELIUM_STAR_MS,
    NAKED_HELIUM_STAR_HERTZSPRUNG_GAP,
    NAKED_HELIUM_STAR_GIANT_BRANCH,
    HELIUM_WHITE_DWARF,
    CARBON_OXYGEN_WHITE_DWARF,
    OXYGEN_NEON_WHITE_DWARF,
    NEUTRON_STAR,
    BLACK_HOLE,
    MASSLESS_REMNANT,
    CHEMICALLY_HOMOGENEOUS,
    STAR,
    BINARY_STAR,
    NONE
};
```

Ordinal numbers can still be used to reference the stellar types, and because of the order of definition in the enum class the ordinal numbers match those given in Hurley et al. 2000.

The label map STELLAR_TYPE_LABEL can be used to print text descriptions of the stellar types, and is declared as:

```
const std::unordered_map<STELLAR_TYPE, std::string> STELLAR_TYPE_LABEL = {
 { STELLAR_TYPE::MS_LTE_07,                                 "Main_Sequence_<=_0.7" },
 { STELLAR_TYPE::MS_GT_07,                                  "Main_Sequence_>_0.7" },
 { STELLAR_TYPE::HERTZSPRUNG_GAP,                           "Hertzsprung_Gap" },
 { STELLAR_TYPE::FIRST_GIANT_BRANCH,                        "First_Giant_Branch" },
 { STELLAR_TYPE::CORE_HELIUM_BURNING,                       "Core_Helium_Burning" },
 { STELLAR_TYPE::EARLY_ASYMPTOTIC_GIANT_BRANCH,             "Early_Asymptotic_Giant_Branch" },
 { STELLAR_TYPE::THERMALLY_PULSING_ASYMPTOTIC_GIANT_BRANCH, "Thermally_Pulsing_Asymptotic_Giant_Branch" },
 { STELLAR_TYPE::NAKED_HELIUM_STAR_MS,                      "Naked_Helium_Star_MS" },
 { STELLAR_TYPE::NAKED_HELIUM_STAR_HERTZSPRUNG_GAP,         "Naked_Helium_Star_Hertzsprung_Gap" },
 { STELLAR_TYPE::NAKED_HELIUM_STAR_GIANT_BRANCH,            "Naked_Helium_Star_Giant_Branch" },
 { STELLAR_TYPE::HELIUM_WHITE_DWARF,                        "Helium_White_Dwarf" },
 { STELLAR_TYPE::CARBON_OXYGEN_WHITE_DWARF,                 "Carbon-Oxygen_White_Dwarf" },
 { STELLAR_TYPE::OXYGEN_NEON_WHITE_DWARF,                   "Oxygen-Neon_White_Dwarf" },
 { STELLAR_TYPE::NEUTRON_STAR,                              "Neutron_Star" },
 { STELLAR_TYPE::BLACK_HOLE,                                "Black_Hole" },
 { STELLAR_TYPE::MASSLESS_REMNANT,                          "Massless_Remnant" },
 { STELLAR_TYPE::CHEMICALLY_HOMOGENEOUS,                    "Chemically_Homogeneous" },
 { STELLAR_TYPE::STAR,                                      "Star" },
 { STELLAR_TYPE::BINARY_STAR,                               "Binary_Star" },
 { STELLAR_TYPE::NONE,                                      "Not_a_Star!" }
};
```

Programming Style and Conventions

Everyone has their own preferences and style, and the nature of a project such as COMPAS will reflect that. However, there is a need to suggest some guidelines for programming style, naming conventions etc. Following is a description of some of the elements of programming style and naming conventions used to develop COMPAS v2. These may evolve over time.

Object-Oriented Programming

COMPAS is written in C++, an object-oriented programming (OOP) language, and OOP concepts and conventions should apply throughout the code. There are many texts and web pages devoted to understanding C++ and OOP – following is a brief description of the key OOP concepts:

Abstraction

For any entity, product, or service, the goal of abstraction is to handle the complexity of the implementation by hiding details that don't need to be known in order to use, or consume, the entity, product, or service. In the OOP paradigm, hiding details in this way enables the consumer to implement more complex logic on top of the provided abstraction without needing to understand the hidden implementation details and complexity. (There is no suggestion that consumers shouldn't understand the implementation details, but they shouldn't need to in order to consume the entity, product, or service).

Abstraction in C++ is achieved via the use of *objects* – an object is an instance of a *class*, and typically corresponds to a real-world object or entity (in COMPAS, usually a star or binary star). An object maintains the state of an object (via class member variables), and provides all necessary means of changing the state of the object (by exposing public class member functions (methods)). A class may expose public functions to allow consumers to determine the value of class member variables ("getters"), and to se the value of class member variables ("setters").

Encapsulation

Encapsulation binds together the data and functions that manipulate the data in an attempt to keep both safe from outside interference and accidental misuse. An encapsulation paradigm that does not allow calling code to access internal object data and permits access through functions only is a strong form of abstraction. C++ allows developers to enforce access restrictions explicitly by defining class member variables and functions as *private*, *protected*, or *public*. These keywords are used throughout COMPAS to enforce encapsulation.

There are very few circumstances in which a consumer should change the value of a class member variable directly (via the use of a setter function) – almost always consumers should present new situational information to an object (via a public member function), and allow the object to respond to the new information. For example, in COMPAS, there should be almost no reason for a consumer of a star object to directly change (say) the radius of the star – the consumer should inform the star object of new circumstances or events, and allow the star object to respond to those events (perhaps changing the value of the radius of the star). Changing a single class member variable directly introduces the possibility that related class member variables (e.g. other attributes of stars) will not be changed accordingly. Moreover, developers changing the code in the future should, in almost all cases, expect that the state of an object is maintained consistently by the object, and that there should be no unexpected side-effects caused by calling non class-member functions. In short, changing the state of an object outside the object is potentially unsafe and should be avoided where possible.

Inheritance

Inheritance allows classes to be arranged in a hierarchy that represents *is-a-type-of* relationships. All *non-private* class member variables and functions of the parent (base) class are available to the child (derived) class (and, therefore, child classes of the child class). This allows easy re-use of the same procedures and data definitions, in addition to describing real-world relationships in an intuitive way. C++ allows multiple inheritance – a class may inherit from multiple parent classes.

Derived classes can define additional class member variables (using the *private*, *protected*, and *public* access restrictions), which will be available to any descendent classes (subject to inheritance rules), but will only be available to ancestor classes via the normal access methods (getters and setters).

Polymorphism

Polymorphism means *having many forms*. In OOP, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Following the discussion above regarding inheritance, in the OOP paradigm, and C++ specifically, derived classes can override methods defined by ancestor classes, allowing a derived class to implement functions specific to its circumstances. This means that a call to a class member function will cause a different function to be executed depending on the type of object that invokes the function. Descendent classes of a class that has overridden a base class member function inherit the overridden function (but can override it themselves).

COMPAS makes heavy use of inheritance and polymorphism, especially for the implementation of the different stellar types.

Programming Style

The goal of coding to a suggested style is readability and maintainability – if many developers implement code in COMPAS with their own coding style, readability and maintainability will be more difficult than if a consistent style is used throughout the code. Strict adherence isn't really necessary, but it will make it easier on all COMPAS developers if the coding style is consistent throughout.

Comments

An old, but good, rule-of-thumb is that any file that contains computer code should be about one-third code, one-third comments, and one-third white space. Adhering to this rule-of-thumb just makes the code a bit easier on the eye, and provides some description (at least of the intention) of the implementation.

Braces

The placement of braces in C++ code (actually, any code that uses braces to enclose scope) is a contentious issue, with many developers having long-held, often dogmatic preferences. COMPAS (so far) uses the K&R style ("the one true brace style") - the style used in the original Unix kernel and Kernighan and Ritchie's book *The C Programming Language*.

The K&R style puts the opening brace on the same line as the control statement:

```
while (x == y) {
    something();
    somethingelse();
}
```

Note also the space between the keyword *while* and the opening parenthesis, and the closing parenthesis and the opening brace.

Indentation

There is ongoing debate in the programming community as to whether indentation should be achieved using spaces or tabs (strange, but true…). The use of spaces is more common. COMPAS (so far) has a mix of both – whatever is convenient (pragmatism is your friend...).

COMPAS uses an indentation size of 4 spaces.

Function Parameters

In most cases, function parameters should be input only – meaning that the values of function parameters should not be changed by the function. Anything that needs to be changed and returned to the caller should be returned as a functional return. There are a few exceptions to this in COMPAS – all were done for performance reasons, and are documented in the code.

To avoid unexpected side-effects, developers should expect (in most cases) that any variables they pass to a function will remain unchanged – all changes should be returned as a functional return.

Performance & Optimisation

In general COMPAS developers should code for performance – within reason. Bear in mind that many functions will be called many, many thousands of times (in some cases, millions) in one execution of the program.

- Avoid calculating values inside loops that could be calculated once outside the loop.
- Try to use constants where possible.
- Use multiplication in preference to functions such as *pow()* and *sqrt()* (note that *pow()* is very expensive computationally; *sqrt()* is expensive, but much less expensive than *pow()*).
- Don't optimise to the point that readability and maintainability is compromised. Bear in mind that most compilers are good at optimising, and are very forgiving of less-than-optimally-written code (though they are not miracle workers...).

Naming Conventions

COMPAS (so far) uses the following naming conventions:

- all variable names should be in camelCase – don’t use underscore_to_separate_words
- function names should be in camelCase, beginning with an uppercase letter. Function names should be descriptive.
- class member variable names are prefixed with “m_”, and the character immediately following the prefix should be uppercase (in most cases – sometimes, for well-known names or words that are always written in lowercase, lowercase might be used)
- local variable names are just camelCase, beginning with a lowercase letter (again, with the caveat that sometimes, for well-known names or words that are always written in uppercase, uppercase might be used)
- function parameter names are prefixed with “p_”, and the character immediately following the prefix should be uppercase (again, with the caveat that sometimes, for well-known names or words that are always written in lowercase, lowercase might be used)

COMPILATION & REQUIREMENTS

Please refer to the COMPAS Getting Started guide.

Appendix A – Program Options

-h [--help]
Prints COMPAS help.

-v [--version]
Print COMPAS version string.

--allow-rlof-at-birth
Allow binaries that have one or both stars in RLOF at birth to evolve as overcontact systems.
Default = FALSE

--allow-touching-at-birth
Allow binaries that are touching at birth to be included in the sampling.
Default = FALSE

--alwaysStableCaseBBBCFlag
Choose case BB/BC mass transfer to be always stable.
Default = FALSE

--angularMomentumConservationDuringCircularisation
Conserve angular momentum when binary is circularised when entering a Mass Transfer episode.
Default = FALSE

--circulariseBinaryDuringMassTransfer
Circularise binary when it enters a Mass Transfer episode.
Default = FALSE

--common-envelope-allow-main-sequence-survive
Allow main sequence donors to survive common envelope evolution.
Default = FALSE

--debug-to-file
Write debug statements to file.
Default = FALSE

--detailedOutput
Print detailed output to file.
Default = FALSE

--errors-to-file
Write error messages to file.
Default = FALSE

--evolve-pulsars
Evolve pulsar properties of Neutron Stars.
Default = FALSE

--evolve-unbound-systems
Continue evolving stars even if the binary is disrupted.
Default = FALSE

--forceCaseBBBCStabilityFlag

Force case BB/BC mass transfer to always be only stable or unstable, ignoring usual stability criteria.
Default = FALSE

--lambda-calculation-every-timeStep
Calculate all values of lambda at each timestep.
Default = FALSE

--massTransfer
Enable mass transfer.
Default = TRUE

--pair-instability-supernovae
Enable pair instability supernovae (PISN).
Default = FALSE

--populationDataPrinting
Print details of population.
Default = FALSE

--print-bool-as-string
Print boolean properties as 'TRUE' or 'FALSE'.
Default = FALSE

--pulsational-pair-instability
Enable mass loss due to pulsational-pair-instability (PPI).
Default = FALSE

--quiet
Suppress printing to stdout.
Default = FALSE

--revised-energy-formalism-Nandez-Ivanova
Enable revised energy formalism of Nandez & Ivanova.
Default = FALSE

--single-star
Evolve single star(s).
Default = FALSE

--use-mass-loss
Enable mass loss.
Default = FALSE

--zeta-calculation-every-timestep
Calculate all values of MT zetas at each timestep.
Default = FALSE

--random-seed
Random seed.
Default = 0

--debug-level
Determines which print statements are displayed for debugging.
Default = 0

--log-level
Determines which print statements are included in the logfile.
Default = 0

--maximum-number-timestep-iterations
Maximum number of timesteps to evolve binary.
Default = 99999

-n [--number-of-binaries]
Specify the number of binaries to simulate.
Default = 10

--single-star-mass-steps
Specify the number of mass steps for single star evolution.
Default = 100

--common-envelope-alpha
Common Envelope efficiency alpha.
Default = 1.0

--common-envelope-alpha-thermal
Thermal energy contribution to the total envelope binding energy.
Defined such that lambda = alpha_th * lambda_b + (1.0 -alpha_th) * lambda_g.
Default = 1.0

--common-envelope-lambda
Common Envelope lambda.
Default = 0.1

--common-envelope-lambda-multiplier
Multiply lambda by some constant.
Default = 1.0

--common-envelope-mass-accretion-constant
Value of mass accreted by NS/BH during common envelope evolution if assuming all NS/BH accrete same amount of mass.
Used when --common-envelope-mass-accretion-prescription = CONSTANT, ignored otherwise.
Default = 0.0

--common-envelope-mass-accretion-max
Maximum amount of mass accreted by NS/BHs during common envelope evolution (Msol).
Default = 0.1

--common-envelope-mass-accretion-min
Minimum amount of mass accreted by NS/BHs during common envelope evolution (Msol).
Default = 0.04

--common-envelope-recombination-energy-density
Recombination energy density (ergs/g).
Default = 1.5E+13

--common-envelope-slope-Kruckow
Common Envelope slope for Kruckow lambda.
Default = -0.8

--eccentricity-max
Maximum eccentricity to generate.
Default = 1.0

--eccentricity-min
Minimum eccentricity to generate.
Default = 0.0

--eddington-accretion-factor
Multiplication factor for Eddington accretion for NS & BH, i.e. >1 is super-eddington and 0 is no accretion.
Default = 1.0

--fix-dimensionless-kick-velocity
Fix dimensionless kick velocity uk to this value.
Default = n/a (not used if option not present)

--initial-mass-max
Maximum mass (in Msol) to generate using given IMF.
Default = 100.0

--initial-mass-min
Minimum mass (in Msol) to generate using given IMF.
Default = 8.0

--initial-mass-power
Single power law power to generate primary mass using given IMF.
Default = -2.3

--kick-direction-power
Power for power law kick direction distribution.
Default = 0.0 = isotropic, +ve = polar, -ve = in plane

--kick-scaling-factor
Arbitrary factor used to scale kicks.
Default = 1.0

--kick-velocity-max
Maximum drawn kick velocity in km s^-1.
Must be > 0 if using --kick-velocity-distribution=FLAT
Default = -1.0

--kick-velocity-sigma-CCSN-BH
Sigma for chosen kick velocity distribution for black holes.
Default = 250.0 km s^-1

--kick-velocity-sigma-CCSN-NS
Sigma for chosen kick velocity distribution for neutron stars.
Default = 250.0 km s^-1

--kick-velocity-sigma-ECSN
Sigma for chosen kick velocity distribution for ECSN.
Default = 30.0 km s^-1

--kick-velocity-sigma-USSN
Sigma for chosen kick velocity distribution for USSN.
Default = 30.0 km s^-1

--luminous-blue-variable-multiplier
Multiplicative constant for LBV mass loss.
Default = 1.5, use 10 for Mennekens & Vanbeveren 2014

--mass-ratio-max
Maximum mass ratio m2/m1 to generate.
Default = 1.0

--mass-ratio-min
Minimum mass ratio m2/m1 to generate.
Default = 0.0

--mass-transfer-fa
Mass Transfer fraction accreted, when the FIXED mass transfer prescription is used.
Default = 1.0 (fully conservative)

--mass-transfer-jloss
Specific angular momentum with which the non-accreted system leaves the system.
Used when --mass-transfer-angular-momentum-loss-prescription = ARBITRARY, ignored otherwise.
Default = 1.0

--mass-transfer-thermal-limit-C
Mass Transfer Thermal rate factor for the accretor.
Default = 10.0

--maximum-evolution-time
Maximum time to evolve binaries (Myr).
Default = 13700.0

--maximum-mass-donor-Nandez-Ivanova
Maximum donor mass allowed for the revised common envelope formalism of Nandez & Ivanova (Msol).
Default = 2.0

--maximum-neutron-star-mass
Maximum mass of a neutron star (Msol).
Default = 3.0

-z [--metallicity]
Metallicity.
Default 0.02 (Zsol)

--minimum-secondary-mass
Minimum mass of secondary to generate (Msol).
Default = 0.0

--neutrino-mass-loss-bh-formation-value
Amount of mass lost in neutrinos during BH formation (either as fraction or in solar masses, depending on neutrino-mass-loss-bh-formation setting).
Default = 0.1

--orbital-period-max
Maximum period to generate (days)
Default = 1000.0

--orbital-period-min
Minimum period to generate (days).
Default = 1.1

--PISN-lower-limit
Minimum core mass for PISN (Msol).
Default = 60.0

--PISN-upper-limit
Maximum core mass for PISN (Msol).
Default = 135.0

--PPI-lower-limit
Minimum core mass for PPI (Msol).
Default = 35.0

--PPI-upper-limit
Maximum core mass for PPI (Msol).
Default = 60.0

--pulsar-birth-magnetic-field-distribution-max
Maximum (log10) pulsar birth magnetic field.
Default = 13.0

--pulsar-birth-magnetic-field-distribution-min
Minimum (log10) pulsar birth magnetic field.
Default = 11.0

--pulsar-birth-spin-period-distribution-max
Maximum pulsar birth spin period (ms).
Default = 100.0
--pulsar-birth-spin-period-distribution-min
Minimum pulsar birth spin period (ms).

Default = 0.0

--pulsar-magnetic-field-decay-massscale
Mass scale on which magnetic field decays during accretion (Msol).
Default = 0.025

--pulsar-magnetic-field-decay-timescale
Timescale on which magnetic field decays (Myr).
Default = 1000.0

--pulsar-minimum-magnetic-field
log10 of the minimum pulsar magnetic field (Gauss).
Default = 8.0

--semi-major-axis-max
Maximum semi-major axis to generate (AU).
Default = 1000.0

--semi-major-axis-min
Minimum semi-major axis to generate (AU).
Default = 0.1

--single-star-mass-max
Maximum mass for single star evolution (Msol).
Default = 100.0

--single-star-mass-min
Minimum mass for single star evolution (Msol).
Default = 5.0

--wolf-rayet-multiplier
Multiplicative constant for WR winds.
Default = 1.0

--zeta-adiabatic-arbitrary
Value of logarithmic derivative of radius with respect to mass, zeta adiabatic.
Default = 1.0E+04

--zeta-thermal-arbitrary
Value of logarithmic derivative of radius with respect to mass, zeta thermal.
Default = 1.0E+04

--zeta-radiative-giant-star
Value of logarithmic derivative of radius with respect to mass, zeta for radiative-envelope giant-like stars, including Hertzsprung gap stars.
Default = 6.5

--zeta-main-sequence
Value of logarithmic derivative of radius with respect to mass, zeta on the main sequence.
Default = 2.0

--black-hole-kicks
Black hole kicks relative to NS kicks (options: FULL, REDUCED, ZERO, FALLBACK).
Default = FALLBACK

--chemically-homogeneous-evolution
Chemically Homogeneous Evolution mode (options: NONE, OPTIMISTIC, PESSIMISTIC).
Default = NONE

--common-envelope-lambda-prescription
CE lambda prescription.
(options:
LAMBDA_FIXED, LAMBDA_LOVERIDGE, LAMBDA_NANJING, LAMBDA_KRUCKOW, LAMBDA_DEWI)
Default = LAMBDA_NANJING

--common-envelope-mass-accretion-prescription
Assumption about whether NS/BHs can accrete mass during common envelope evolution.
(options: ZERO, CONSTANT, UNIFORM, MACLEOD)
Default = ZERO

--envelope-state-prescription
Prescription for determining whether the envelope of the star is convective or radiative.
(options: LEGACY, HURLEY, FIXED_TEMPERATURE)
Default = LEGACY

--stellar-zeta-prescription
Prescription for stellar zeta.
(options: STARTRACK, SOBERMAN, HURLEY, ARBITRARY)
Default = SOBERMAN

-e [--eccentricity-distribution]
Initial eccentricity distribution, e.
(options:
ZERO, FIXED, FLAT, THERMALISED, GELLER+2013, THERMAL, DUQUENNOYMAYOR1991, SANA2012, IMPORTANCE).
Default = ZERO

--fryer-supernova-engine
Supernova engine type if using Fryer et al. 2012 fallback prescription.
(options: DELAYED, RAPID).
Default = DELAYED

--grid
Grid filename.
Default = ‘’

-i [--initial-mass-function]
Initial mass function, (options: SALPETER, POWERLAW, UNIFORM, KROUPA)
Default = KROUPA

--kick-direction
Natal kick direction distribution.
(options: ISOTROPIC, INPLANE, PERPENDICULAR, POWERLAW, WEDGE, POLES)
Default = ISOTROPIC

--kick-velocity-distribution
Natal kick velocity distribution.
(options:

ZERO, FIXED, FLAT, MAXWELLIAN, MAXWELL, MUELLER2016, MUELLER2016MAXWELLIAN, BRAYELDRIDGE, MULELRMANDEL)
Default = MAXWELLIAN

--logfile-BSE-common-envelopes
Filename for BSE Common Envelopes logfile.
Default = 'BSE_Common_Envelopes'

--logfile-BSE-detailed-output
Filename for BSE Detailed Output logfile.
Default = 'BSE_Detailed_Output'

--logfile-BSE-double-compact-objects
Filename for BSE Double Compact Objects logfile.
Default = 'BSE_Double_Compact_Objects'

--logfile-BSE-pulsar-evolution
Filename for BSE Pulsar Evolution logfile.
Default = 'BSE_Pulsar_Evolution'

--logfile-BSE-supernovae
Filename for BSE Supernovae logfile.
Default = 'BSE_Supernovae'

--logfile-BSE-system-parameters
Filename for BSE System Parameters logfile.
Default = 'BSE_System_Parameters'

--logfile-definitions
Filename for logfile record definitions.
Default = ''

--logfile-delimiter
Field delimiter for logfile records.
Default = TAB

--logfile-name-prefix
Prefix for logfile names.
Default = ''

--logfile-SSE-parameters
Filename for SSE Parameters logfile.
Default = 'SSE_Parameters'

--mass-loss-prescription
Mass loss prescription, (options: NONE, HURLEY, VINK).
Default = VINK

-q [--mass-ratio-distribution]
Initial mass ratio distribution for q=m2/m1.
(options: FLAT, DUQUENNOYMAYOR1991, SANA2012).
Default = FLAT

--mass-transfer-accretion-efficiency-prescription
Mass Transfer Accretion Efficiency prescription, (options: THERMAL, FIXED, CENTRIFUGAL).
Default = ISOTROPIC

--mass-transfer-angular-momentum-loss-prescription
Mass Transfer Angular Momentum Loss prescription.
(options: JEANS, ISOTROPIC, CIRCUMBINARY, ARBITRARY)
Default = ISOTROPIC

--mass-transfer-prescription
Mass Transfer prescription, (options: HURLEY, BELCZYNSKI, NONE).
Default = HURLEY

--mass-transfer-rejuvenation-prescription
Mass Transfer Rejuvenation prescription, (options: NONE, STARTRACK).
Default = NONE

--mass-transfer-thermal-limit-accretor
Mass Transfer Thermal Accretion limit multiplier, (options: CFACTOR, ROCHELOBE).
Default = CFACTOR

--neutrino-mass-loss-bh-formation
Assumption about neutrino mass loss during BH formation.
(options: FIXED_FRACTION, FIXED_MASS).
Default = FIXED_FRACTION

--neutron-star-equation-of-state
Neutron star equation of state, (options: SSE, ARP3).
Default = SSE

-c [--output-container]
Container (directory) name for output files.
Default = 'COMPAS_Output'

-o [--outputPath]
Path to which output is saved (i.e. directory in which the output container is created).
Default = CWD ('.')

--pulsar-birth-magnetic-field-distribution
Pulsar Birth Magnetic Field distribution.
(options: ZERO, FIXED, FLATINLOG, UNIFORM, LOGNORMAL).
Default = ZERO

--pulsar-birth-spin-period-distribution
Pulsar Birth Spin Period distribution, (options: ZERO, FIXED, UNIFORM, NORMAL).
Default = ZERO

--pulsational-pair-instability-prescription
Pulsational Pair Instability prescription, (options: COMPAS, STARTRACK, MARCHANT).

Default = COMPAS

--remnant-mass-prescription
Choose remnant mass prescription.
(options: POSTITNOTE, HURLEY2000, BELCZYNSKI2002, FRYER2012, MULLER2016, MULLER2016MAXWEL-LIAN,MULLERMANDEL)
Default = FRYER2012

--rotational-velocity-distribution
Initial rotational velocity distribution, (options: ZERO, HURLEY, VLTFLAMES).
Default = ZERO

-a [--semi-major-axis-distribution]
Initial semi-major axis distribution, a.
(options: FLATINLOG, CUSTOM, DUQUENNOYMAYOR1991, SANA2012)
Default = FLATINLOG

--debug-classes
Debug classes enabled.
Default = ‘’ (None)

--log-classes
Logging classes enabled.
Default = ‘’ (None)

Appendix B – Log File Record Specification: Stellar Properties

As described in **Standard Log File Record Specifiers**, when specifying known properties in a log file record specification record, the property name must be prefixed with the property type.

The current list of valid stellar property types available for use is:

• STAR_PROPERTY	for SSE
• STAR_1_PROPERTY	for the primary star of a binary for BSE
• STAR_2_PROPERTY	for the secondary star of a binary for BSE
• SUPERNOVA_PROPERTY	for the exploding star in a supernova for BSE
• COMPANION_PROPERTY	for the companion of the exploding star in a supernova for BSE

For example, to specify the property TEMPERATURE for an individual star being evolved for SSE, use:

```
STAR_PROPERTY::TEMPERATURE
```

To specify the property TEMPERATURE for the primary star in a binary star being evolved for BSE, use:

```
STAR_1_PROPERTY::TEMPERATURE
```

To specify the property TEMPERATURE for the supernova star in a binary star being evolved for BSE, use:

```
SUPERNOVA_PROPERTY::TEMPERATURE
```

Following is the list of stellar properties available for inclusion in log file record specifiers.

Stellar Properties

AGE
Header string: Age
Data type: DOUBLE
COMPAS variable: BaseStar::m_Age
Effective age (changes with mass loss/gain) (Myr)

ANGULAR_MOMENTUM
Header string: Ang_Momentum
Data type: DOUBLE
COMPAS variable: BaseStar::m_AngularMomentum
Angular momentum (Msol AU^2 yr-1)

BINDING_ENERGY_AT_COMMON_ENVELOPE
Header string: Binding_Energy@CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.bindingEnergy
Absolute value of the envelope binding energy (ergs) at the onset of unstable RLOF, used for calculating post-CE separation.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

BINDING_ENERGY_FIXED

Header string: BE_Fixed
Data type: DOUBLE
COMPAS variable: BaseStar::m_BindingEnergies.fixed
Absolute value of the envelope binding energy (ergs) calculated using a fixed lambda parameter.
Calculated using lambda = m_Lambdas.fixed

BINDING_ENERGY_KRUCKOW
Header string: BE_Kruckow
Data type: DOUBLE
COMPAS variable: BaseStar::m_BindingEnergies.kruckow
Absolute value of the envelope binding energy (ergs) calculated using the fit by Vigna-Gomez et al. (2018) to Kruckow et al. (2016).
Calculated using alpha = OPTIONS->CommonEnvelopeSlopeKruckow()

BINDING_ENERGY_LOVERIDGE
Header string: BE_Loveridge
Data type: DOUBLE
COMPAS variable: BaseStar::m_BindingEnergies.loveridge
Absolute value of the envelope binding energy (ergs) calculated as per Loveridge et al. (2001).
Calculated using lambda = m_Lambdas.loveridge

BINDING_ENERGY_LOVERIDGE_WINDS
Header string: BE_Loveridge_Winds
Data type: DOUBLE
COMPAS variable: BaseStar::m_BindingEnergies.loveridgeWinds
Absolute value of the envelope binding energy (ergs) calculated as per Webbink 1984 & Loveridge et al. 2011 including winds.
Calculated using lambda = m_Lambdas.loveridgeWinds

BINDING_ENERGY_NANJING
Header string: BE_Nanjing
Data type: DOUBLE
COMPAS variable: BaseStar::m_BindingEnergies.nanjing
Absolute value of the envelope binding energy (ergs) calculated as per Xu & Li, 2010.
Calculated using lambda = m_Lambdas.nanjing

BINDING_ENERGY_POST_COMMON_ENVELOPE
Header string: Binding_Energy>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.bindingEnergy
Absolute value of the binding energy (ergs) immediately after CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

BINDING_ENERGY_PRE_COMMON_ENVELOPE
Header string: Binding_Energy<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.bindingEnergy
Absolute value of the envelope binding energy (ergs) at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

CHEMICALLY_HOMOGENEOUS_MAIN_SEQUENCE
Header string: CH_on_MS
Data type: BOOL

COMPAS variable: BaseStar::m_CHE
Flag to indicate whether the star evolved as a CH star for its entire MS lifetime.
TRUE indicates star evolved as CH star for entire MS lifetime
FALSE indicates star spun down and switched from CH to a normal MS

CO_CORE_MASS
Header string: Mass_CO_Core
Data type: DOUBLE
COMPAS variable: BaseStar::m_COCoreMass
Carbon-Oxygen core mass (Msol).

CO_CORE_MASS_AT_COMMON_ENVELOPE
Header string: Mass_CO_Core@CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.COCoreMass
Carbon-Oxygen core mass (Msol) at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

CO_CORE_MASS_AT_COMPACT_OBJECT_FORMATION
Header string: : Mass_CO_Core@CO
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.COCoreMassAtCOFormation
Carbon-Oxygen core mass (Msol) immediately prior to a supernova.

CORE_MASS
Header string: Mass_Core
Data type: DOUBLE
COMPAS variable: BaseStar::m_CoreMass
Core mass (Msol)

CORE_MASS_AT_COMMON_ENVELOPE
Header string: Mass_Core@CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.CoreMass
Core mass (Msol) at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

CORE_MASS_AT_COMPACT_OBJECT_FORMATION
Header string: Mass_Core@CO
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.CoreMassAtCOFormation
Core mass (Msol) immediately prior to a supernova .

DRAWN_KICK_VELOCITY
Header string: Drawn_Kick_Velocity
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.drawnKickVelocity
Magnitude of natal kick velocity (kms^-1) without accounting for fallback (km s^-1).
Supplied by user in grid file or drawn from distribution (default).
This value is used to calculate the actual kick velocity

DT
Header string: dT
Data type: DOUBLE
COMPAS variable: BaseStar::m_Dt
Current timestep (Myr).

DYNAMICAL_TIMESCALE
Header string: Tau_Dynamical
Data type: DOUBLE
COMPAS variable: BaseStar::m_DynamicalTimescale
Dynamical time (Myr) .

DYNAMICAL_TIMESCALE_POST_COMMON_ENVELOPE
Header string: Tau_Dynamical>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.dynamicalTimescale
Dynamical time (Myr) immediately following common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

DYNAMICAL_TIMESCALE_PRE_COMMON_ENVELOPE
Header string: Tau_Dynamical<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.dynamicalTimescale
Dynamical timescale (Myr) immediately prior to common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

ECCENTRIC_ANOMALY
Header string: Eccentric_Anomaly
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.eccentricAnomaly
Eccentric anomaly calculated using Kepler's equation.

ENV_MASS
Header string: Mass_Env
Data type: DOUBLE
COMPAS variable: BaseStar::m_EnvMass
Envelope mass (Msol) calculated using Hurley et al. 2000.

ERROR
Header string: Error
Data type: INT
COMPAS variable: <derived from BaseStar::m_Error>
Error number (if error condition exists, else 0).

EXPERIENCED_CCSN
Header string: Experienced_CCSN

Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star exploded as a core-collapse supernova at any time prior to the current timestep.

EXPERIENCED_ECSN
Header string: Experienced_ECSN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past >
Flag to indicate whether the star exploded as an electron-capture supernova at any time prior to the current timestep.

EXPERIENCED_PISN
Header string: Experienced_PISN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star exploded as a pair-instability supernova at any time prior to the current timestep.

EXPERIENCED_PPISN
Header string: Experienced_PPISN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star exploded as a pulsational pair-instability supernova at any time prior to the current timestep.

EXPERIENCED_RLOF
Header string: Experienced_RLOF
Data type: BOOL
COMPAS variable: <derived from BinaryConstituentStar::m_RLOFDetails.experiencedRLOF>
Flag to indicate whether the star has overflowed its Roche Lobe at any time prior to the current timestep.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

EXPERIENCED_SN_TYPE
Header string: Experienced_SN_Type
Data type: INT
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
The type of supernova event experienced by the star prior to the current timestep.
Printed as one of {NONE = 0, CCSN = 1, ECSN = 2, PISN = 4, PPISN = 8, USSN = 16}
(see section **Supernova events/states** below for explanation)

EXPERIENCED_USSN
Header string: Experienced_USSN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star exploded as an ultra-stripped supernova at any time prior to the current timestep.

FALLBACK_FRACTION
Header string: Fallback_Fraction
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.fallbackFraction
Fallback fraction during a supernova.

HE_CORE_MASS
Header string: Mass_He_Core
Data type: DOUBLE
COMPAS variable: BaseStar::m_HeCoreMass
Helium core mass (Msol).

HE_CORE_MASS_AT_COMMON_ENVELOPE
Header string: Mass_He_Core@CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.HeCoreMass
Helium core mass (Msol) at the onset of unstable RLOF leading to the CE..
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

HE_CORE_MASS_AT_COMPACT_OBJECT_FORMATION
Header string: Mass_He_Core@CO
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.HeCoreMassAtCOFormation
Helium core mass (Msol) at immediately prior to a of supernova.

HYDROGEN_POOR
Header string: Hydrogen_Poor
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.hydrogenContent>
Flag to indicate if star is hydrogen poor.

HYDROGEN_RICH
Header string: Hydrogen_Rich
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.hydrogenContent>
Flag to indicate if star is hydrogen rich.

ID
Header string: ID
Data type: UNSIGNED LONG INT
COMPAS variable: BaseStar::m_ObjectId
Unique object identifier for c++ object – used in debugging to identify objects.

INITIAL_STELLAR_TYPE
Header string: Stellar_Type@ZAMS
Data type: INT
COMPAS variable: BaseStar::m_InitialStellarType
Stellar type at zero age main-sequence (per Hurley at al. 2000).
Note that this property has the same header string as INITIAL_STELLAR_TYPE_NAME below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

INITIAL_STELLAR_TYPE_NAME
Header string: Stellar_Type@ZAMS

Data type: STRING
COMPAS variable: <derived from BaseStar::m_InitialStellarType>
Stellar type at zero age main-sequence name (per Hurley at al. 2000).
e.g. “First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as INITIAL_STELLAR_TYPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

IS_CCSN
Header string: CCSN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
Flag to indicate whether the star is currently a core-collapse supernova.

IS_ECSN
Header string: ECSN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
Flag to indicate whether the star is currently an electron-capture supernova.

IS_PISN
Header string: PISN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
Flag to indicate whether the star is currently a pair-instability supernova.

IS_PPISN
Header string: PPISN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
Flag to indicate whether the star is currently a pulsational pair-instability supernova.

IS_RLOF
Header string: RLOF
Data type: BOOL
COMPAS variable: <derived from BinaryConstituentStar::m_RLOFDetails.isRLOF>
Flag to indicate whether the star is currently undergoing Roche Lobe overflow.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

IS_USSN
Header string: USSN
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
Flag to indicate whether the star is currently an ultra-stripped supernova.

KICK_VELOCITY
Header string: Applied_Kick_Velocity
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.kickVelocity
Magnitude of natal kick velocity (kms^-1) received during a supernova.
Calculate using the drawn kick velocity (see BaseStar::CalculateSNKickVelocity() for details)

LAMBDA_AT_COMMON_ENVELOPE
Header string: Lambda@CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.lambda
Common-envelope lambda parameter calculated at the unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

LAMBDA_DEWI
Header string: Dewi
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.dewi
Common envelope lambda parameter calculated as per Dewi & Tauris (2000) using the fit from Appendix A of Claeys et al. (2014).

LAMBDA_FIXED
Header string: Lambda_Fixed
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.fixed
Universal common envelope lambda parameter specified by the user (program option --common-envelope-lambda).

LAMBDA_KRUCKOW
Header string: Kruckow
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.kruckow
Common envelope lambda parameter calculated as per Kruckow et al. (2016) with the alpha exponent set by OPTIONS->CommonEnvelopeSlopeKruckow().
From Kruckow et al. 2016 (arXiv:1610.04417), fig 1.
Spectrum fit to the region bounded by the upper and lower limits as shown in Kruckow+ 2016

LAMBDA_KRUCKOW_BOTTOM
Header string: Kruckow_Bottom
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.kruckowBottom
Common envelope lambda parameter calculated as per Kruckow et al. (2016) with the alpha exponent set to -1.0.
From Kruckow et al. 2016 (arXiv:1610.04417), fig 1.
Spectrum fit to the region bounded by the upper and lower limits as shown in Kruckow+ 2016

LAMBDA_KRUCKOW_MIDDLE
Header string: Kruckow_Middle
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.kruckowMiddle
Common envelope lambda parameter calculated as per Kruckow et al. (2016) with the alpha exponent set to -4/5.
From Kruckow et al. 2016 (arXiv:1610.04417), fig 1.
Spectrum fit to the region bounded by the upper and lower limits as shown in Kruckow+ 2016

LAMBDA_KRUCKOW_TOP
Header string: Kruckow_Top
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.kruckowTop
Common envelope lambda parameter as per Kruckow et al. (2016) with the alpha exponent set to -2/3.
From Kruckow et al. 2016 (arXiv:1610.04417), fig 1.
Spectrum fit to the region bounded by the upper and lower limits as shown in Kruckow+ 2016

LAMBDA_LOVERIDGE
Header string: Loveridge
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.loveridge
Common envelope lambda parameter calculated per Webbink 1984 & Loveridge et al. 2011.

LAMBDA_LOVERIDGE_WINDS
Header string: Loveridge_Winds
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.loveridgeWinds
Common envelope lambda parameter calculated per Webbink 1984 & Loveridge et al. 2011 including winds.
Note: currently this evaluates the same as BINDING_ENERGY_LOVERIDGE

LAMBDA_NANJING
Header string: Lambda_Nanjing
Data type: DOUBLE
COMPAS variable: BaseStar::m_Lambdas.nanjing
Common envelope lambda parameter calculated as per Xu & Li (2010).
From X.-J. Xu and X.-D. Li arXiv:1004.4957 (v1, 28Apr2010) as implemented in STARTRACK

LBV_PHASE_FLAG
Header string: LBV_Phase_Flag
Data type: BOOL
COMPAS variable: BaseStar::m_LBVphaseFlag
Flag to indicate if the star ever entered the luminous blue variable phase.

LUMINOSITY
Header string: Luminosity
Data type: DOUBLE
COMPAS variable: BaseStar::m_Luminosity
Luminosity (Lsol).

LUMINOSITY_POST_COMMON_ENVELOPE
Header string: Luminosity>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.luminosity
Luminosity (Lsol) immediately following common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

LUMINOSITY_PRE_COMMON_ENVELOPE
Header string: Luminosity<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.luminosity
Luminosity (Lsol) at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

MASS
Header string: Mass
Data type: DOUBLE
COMPAS variable: BaseStar::m_Mass
Mass (Msol).

MASS_0
Header string: Mass_0
Data type: DOUBLE
COMPAS variable: BaseStar::m_Mass0
Effective initial mass (Msol).

MASS_LOSS_DIFF
Header string: dmWinds
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_MassLossDiff
The amount of mass lost due to winds (Msol)
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

MASS_TRANSFER_CASE_INITIAL
Header string: MT_Case
Data type: INT
COMPAS variable: BinaryConstituentStar::m_MassTransferCase
Indicator of mass transfer type when first RLOF occurs, if any (one of {NONE, A, B, C, D})
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

MASS_TRANSFER_DIFF
Header string: dmMT
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_MassTransferDiff
Amount of mass (Msol) accreted or donated during a mass transfer episode.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

MDOT
Header string: Mdot
Data type: DOUBLE
COMPAS variable: BaseStar::m_Metallicity
Mass loss rate (Msol yr^-1).
MEAN_ANOMALY
Header string: SN_Kick_Mean_Anomaly

Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.meanAnomaly
Mean anomaly of supernova kick, supplied by user in grid file, default = random number between 0 and 2pi.
See https://en.wikipedia.org/wiki/Mean_anomaly for explanation

METALLICITY
Header string: Metallicity@ZAMS
Data type: DOUBLE
COMPAS variable: BaseStar::m_Metallicity
Metallicity.

MZAMS
Header string: Mass@ZAMS
Data type: DOUBLE
COMPAS variable: BaseStar::m_MZAMS
ZAMS Mass (Msol).

NUCLEAR_TIMESCALE
Header string: Tau_Nuclear
Data type: DOUBLE
COMPAS variable: BaseStar::m_NuclearTimescale
Nuclear timescale (Myr).

NUCLEAR_TIMESCALE_POST_COMMON_ENVELOPE
Header string: Tau_Nuclear>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.nuclearTimescale
Nuclear timescale (Myr) immediately following common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

NUCLEAR_TIMESCALE_PRE_COMMON_ENVELOPE
Header string: Tau_Nuclear<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.nuclearTimescale
Nuclear timescale (Myr) at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

OMEGA
Header string: Omega
Data type: DOUBLE
COMPAS variable: BaseStar::m_Omega
Angular frequency (yr^-1).

OMEGA_BREAK
Header string: Omega_Break
Data type: DOUBLE
COMPAS variable: BaseStar::m_OmegaBreak
Break-up angular frequency (yr^-1).

OMEGA_ZAMS
Header string: Omega@ZAMS
Data type: DOUBLE
COMPAS variable: BaseStar::m_OmegaZAMS

Angular frequency at ZAMS (yr^-1).

ORBITAL_ENERGY_POST_SUPERNOVA
Header string: Orbital_Energy>SN
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_PostSNeOrbitalEnergy
Absolute value of orbital energy immediately following supernova event (Msol AU^2 yr^-2).
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

ORBITAL_ENERGY_PRE_SUPERNOVA
Header string: Orbital_Energy<SN
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_PreSNeOrbitalEnergy
Orbital energy immediately prior to supernova event (Msol AU^2 yr^-2).
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

PULSAR_MAGNETIC_FIELD
Header string: Pulsar_Mag_Field
Data type: DOUBLE
COMPAS variable: BaseStar::m_PulsarDetails.magneticField
Pulsar magnetic field strength (G).

PULSAR_SPIN_DOWN_RATE
Header string: Pulsar_Spin_Down
Data type: DOUBLE
COMPAS variable: BaseStar::m_PulsarDetails.spinDownRate
Pulsar spin-down rate.

PULSAR_SPIN_FREQUENCY
Header string: Pulsar_Spin_Freq
Data type: DOUBLE
COMPAS variable: BaseStar::m_PulsarDetails.spinFrequency
Pulsar spin angular frequency (rads s^-1).

PULSAR_SPIN_PERIOD
Header string: Pulsar_Spin_Period
Data type: DOUBLE
COMPAS variable: BaseStar::m_PulsarDetails.spinPeriod
Pulsar spin period (ms).

RADIAL_EXPANSION_TIMESCALE
Header string: Tau_Radial
Data type: DOUBLE
COMPAS variable: BaseStar::m_RadialExpansionTimescale
e-folding time (Myr) of stellar radius.

RADIAL_EXPANSION_TIMESCALE_POST_COMMON_ENVELOPE
Header string: Tau_Radial>CE
Data type: DOUBLE
COMPAS variable: : BinaryConstituentStar::m_CEDetails.postCEE.radialExpansionTimescale
e-folding time (Myr) of stellar radius immediately following common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

RADIAL_EXPANSION_TIMESCALE_PRE_COMMON_ENVELOPE
Header string: Tau_Radial<CE
Data type: DOUBLE
COMPAS variable: : BinaryConstituentStar::m_CEDetails.preCEE.radialExpansionTimescale
e-folding time (Myr) of stellar radius at the onset of unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

RADIUS
Header string: Radius
Data type: DOUBLE
COMPAS variable: BaseStar::m_Radius
Radius (Rsol).

RANDOM_SEED
Header string: SEED
Data type: UNSIGNED LONG
COMPAS variable: BaseStar::m_RandomSeed
Seed for random number generator for this star.

RECYCLED_NEUTRON_STAR
Header string: Recycled_NS
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the object was a recycled neutron star at any time prior to the current timestep (was a neutron star accreting mass).

RLOF_ONTO_NS
Header string: RLOF->NS
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star transferred mass to a neutron star at any time prior to the current timestep.

RUNAWAY
Header string: Runaway
Data type: BOOL
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.past>
Flag to indicate whether the star was unbound by a supernova event at any time prior to the current timestep.
(Unbound after supernova event and not a WD, NS, BH or MR)

RZAMS
Header string: R@ZAMS
Data type: DOUBLE
COMPAS variable: BaseStar::m_RZAMS
ZAMS Radius (Rsol)

SN_TYPE
Header string: SN_Type
Data type: INT
COMPAS variable: <derived from BaseStar::m_SupernovaDetails.events.current>
The type of supernova event currently being experienced by the star.
Printed as one of {NONE = 0, CCSN = 1, ECSN = 2, PISN = 4, PPISN = 8, USSN = 16}
(see section **Supernova events/states** below for explanation)

STELLAR_TYPE
Header string: Stellar_Type
Data type: INT
COMPAS variable: BaseStar::m_StellarType
Stellar type (per Hurley at al. 2000)
Note that this property has the sam.e header string as STELLAR_TYPE_NAME below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_NAME
Header string: Stellar_Type
Data type: STRING
COMPAS variable: <derived from BasteStar::m_StellarType>
Stellar type name (per Hurley at al. 2000).
e.g. “First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_PREV
Header string: Stellar_Type_Prev
Data type: INT
COMPAS variable: BaseStar::m_StellarTypePrev
Stellar type (per Hurley at al. 2000) at previous timestep.
Note that this property has the same header string as STELLAR_TYPE_PREV_NAME below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_PREV_NAME
Header string: Stellar_Type_Prev
Data type: STRING
COMPAS variable: <derived from BaseStar::m_StellarTypePrev>
Stellar type name (per Hurley at al. 2000) at previous timestep.
e.g. “First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE_PREV above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SUPERNOVA_KICK_VELOCITY_MAGNITUDE_RANDOM_NUMBER
Header string: SN_Kick_Magnitude_Random_Number

Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.kickVelocityRandom
Random number for drawing the supernova kick velocity magnitude (if required).
Either supplied by user in grid file, default = drawn from uniform random distribution 0..1

SUPERNOVA_PHI
Header string: SN_Kick_Phi
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.phi
Angle between 'x' and 'y', both in the orbital plane of supernovae vector (rad).
Either supplied by user in grid file, default = drawn from uniform random distribution 0..2pi

SUPERNOVA_THETA
Header string: SN_Kick_Theta
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.theta
Angle between the orbital plane and the 'z' axis of supernovae vector (rad).
Either supplied by user in grid file, default = drawn from specified distribution (option – kick_direction)

TEMPERATURE
Header string: Teff
Data type: DOUBLE
COMPAS variable: BaseStar::m_Temperature
Effective temperature (Tsol).

TEMPERATURE_POST_COMMON_ENVELOPE
Header string: Teff>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.temperature
Effective temperature (Tsol) immediately following common envelope event.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

TEMPERATURE_PRE_COMMON_ENVELOPE
Header string: Teff<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.temperature
Effective temperature (Tsol) at the unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

THERMAL_TIMESCALE
Header string: Tau_Thermal
Data type: DOUBLE
COMPAS variable: BaseStar::m_ThermalTimescale
Thermal timescale (Myr).

THERMAL_TIMESCALE_POST_COMMON_ENVELOPE
Header string: Tau_Thermal>CE
Data type: DOUBLE
COMPAS variable: : BinaryConstituentStar::m_CEDetails.postCEE.thermalTimescale
Thermal timescale immediately following common envelope event (Myr).
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

THERMAL_TIMESCALE_PRE_COMMON_ENVELOPE
Header string: Tau_Thermal<CE
Data type: DOUBLE
COMPAS variable: : BinaryConstituentStar::m_CEDetails.preCEE.thermalTimescale
Thermal timescale (Myr) at the onset of the unstable RLOF leading to the CE.
Applies only to constituent stars of a binary system (i.e. does not apply to SSE star)

TIME
Header string: Time
Data type: DOUBLE
COMPAS variable: BaseStar::m_Time
Time since ZAMS (Myr).

TIMESCALE_MS
Header string: tMS
Data type: DOUBLE
COMPAS variable: BaseStar::m_Timescales[tMS]
Main Sequence timescale (Myr).

TOTAL_MASS_AT_COMPACT_OBJECT_FORMATION
Header string: Mass_Total@CO
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.totalMassAtCOFormation
Total mass of the star at the beginning of a supernova event (Msol).

TRUE_ANOMALY
Header string: True_Anomaly(psi)
Data type: DOUBLE
COMPAS variable: BaseStar::m_SupernovaDetails.trueAnomaly
True anomaly (rad) calculated using Kepler's equation .
See https://en.wikipedia.org/wiki/True_anomaly for explanation

ZETA_HURLEY
Header string: Zeta_Hurley
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.hurley
Adiabatic exponent calculated per Hurley et al. (2002) using core mass.

ZETA_HURLEY_HE
Header string: Zeta_Hurley_He
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.hurleyHe
Adiabatic exponent calculated per (Hurley et al. 2002) using He core mass.

ZETA_NUCLEAR
Header string: Zeta_Nuclear
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.nuclear
Nuclear timescale mass-radius exponent.

ZETA_SOBERMAN
Header string: Zeta_Soberman
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.soberman
Adiabatic exponent calculated per Soberman, Phinney, van den Heuvel (1997) using core mass.

ZETA_SOBERMAN_HE
Header string: Zeta_Soberman_He
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.sobermanHe
Adiabatic exponent calculated per Soberman, Phinney, van den Heuvel (1997) using He-core mass.

ZETA_THERMAL
Header string: Zeta_Thermal
Data type: DOUBLE
COMPAS variable: BaseStar::m_Zetas.thermal
Thermal timescale mass-radius exponent.

Supernova events/states

Supernova events/states, both current ("is") and past ("experienced"), are stored within COMPAS as bit maps. That means different values can be ORed or ANDed into the bit map, so that various events or states can be set concurrently.

The values shown below for the SN_EVENT type are powers of 2 so that they can be used in a bit map and manipulated with bit-wise logical operators. Any of the individual supernova event/state types that make up the SN_EVENT type can be set independently of any other event/state.

```
enum class SN_EVENT: int {
   NONE            = 0,
   CCSN            = 1,
   ECSN            = 2,
   PISN            = 4,
   PPISN           = 8,
   USSN            = 16,
   RUNAWAY         = 32,
   RECYCLED_NS     = 64,
   RLOF_ONTO_NS = 128
};

const COMPASUnorderedMap<SN_EVENT, std::string> SN_EVENT_LABEL = {
   { SN_EVENT::NONE,                "No Supernova" },
   { SN_EVENT::CCSN,                "Core Collapse Supernova" },
   { SN_EVENT::ECSN,                "Electron Capture Supernova" },
   { SN_EVENT::PISN,                "Pair Instability Supernova" },
   { SN_EVENT::PPISN,               "Pulsational Pair Instability Supernova" },
   { SN_EVENT::USSN,                "Ultra Stripped Supernova" },
   { SN_EVENT::RUNAWAY,             "Runaway Companion" },
   { SN_EVENT::RECYCLED_NS,"Recycled Neutron Star" },
   { SN_EVENT::RLOF_ONTO_NS,        "Donated Mass to Neutron Star through RLOF" }
};
```

A convenience function (shown below) is provided in utils.cpp to interpret the bit map. Given an SN_EVENT bitmap (current or past), it returns (in priority order):

SN_EVENT::CCSN	iff CCSN bit is set and USSN bit is not set
SN_EVENT::ECSN	iff ECSN bit is set
SN_EVENT::PISN	iff PISN bit is set
SN_EVENT::PPISN	iff PPISN bit is set
SN_EVENT::USSN	iff USSN bit is set
SN_EVENT::NONE	otherwise

```
/*
 * Returns a single SN type based on the SN_EVENT parameter passed
 *
 * Returns (in priority order):
 *
 *    SN_EVENT::CCSN  iff CCSN  bit is set and USSN bit is not set
 *    SN_EVENT::ECSN  iff ECSN  bit is set
 *    SN_EVENT::PISN  iff PISN  bit is set
 *    SN_EVENT::PPISN iff PPISN bit is set
 *    SN_EVENT::USSN  iff USSN  bit is set
 *    SN_EVENT::NONE  otherwise
 *
 *
 * @param   [IN]    p_SNEvent      SN_EVENT mask to check for SN event type
 * @return                                    SN_EVENT
 */
SN_EVENT SNEventType(const SN_EVENT p_SNEvent) {

    if ((p_SNEvent & (SN_EVENT::CCSN | SN_EVENT::USSN)) == SN_EVENT::CCSN )
       return SN_EVENT::CCSN;

    if ((p_SNEvent & SN_EVENT::ECSN ) == SN_EVENT::ECSN    ) return SN_EVENT::ECSN;
    if ((p_SNEvent & SN_EVENT::PISN ) == SN_EVENT::PISN    ) return SN_EVENT::PISN;
    if ((p_SNEvent & SN_EVENT::PPISN) == SN_EVENT::PPISN   ) return SN_EVENT::PPISN;
    if ((p_SNEvent & SN_EVENT::USSN ) == SN_EVENT::USSN    ) return SN_EVENT::USSN;

    return SN_EVENT::NONE;
}
```

Appendix C – Log File Record Specification: Binary Properties

As described in **Standard Log File Record Specifiers**, when specifying known properties in a log file record specification record, the property name must be prefixed with the property type.

Currently there is a single binary property type available for use: BINARY_PROPERTY.

For example, to specify the property SEMI_MAJOR_AXIS_PRE_COMMON_ENVELOPE for a binary star being evolved in BSE, use:

BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRE_COMMON_ENVELOPE

Following is the list of binary properties available for inclusion in log file record specifiers.

Binary Properties

CIRCULARIZATION_TIMESCALE
Header string: Tau_Circ
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CircularizationTimescale
Tidal circularisation timescale (Myr).

COMMON_ENVELOPE_ALPHA
Header string: CE_Alpha
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.alpha
Common envelope alpha (efficiency) parameter
User-supplied via command line option --common-envelope-alpha.

COMMON_ENVELOPE_AT_LEAST_ONCE
Header string: CEE
Data type: BOOL
COMPAS variable: <derived from BaseBinaryStar::m_CEDetails.CEEcount>
Flag to indicate if there has been at least one common envelope event.

COMMON_ENVELOPE_EVENT_COUNT
Header string: CE_Event_Count
Data type: UNSIGNED INT
COMPAS variable: BaseBinaryStar::m_CEDetails.CEEcount
The number of common envelope events.

DIMENSIONLESS_KICK_VELOCITY
Header string: Kick_Velocity(uK)
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_uK
Dimensionless kick velocity supplied by user (see option --fix-dimensionless-kick-velocity).

DOUBLE_CORE_COMMON_ENVELOPE
Header string: Double_Core_CE
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_CEDetails.doubleCoreCE
Flag to indicate double-core common envelope.
DT

Header string: dT
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_Dt
Current timestep (Myr).

ECCENTRICITY
Header string: Eccentricity
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_Eccentricity
Orbital eccentricity.

ECCENTRICITY_AT_DCO_FORMATION
Header string: Eccentricity@DCO
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_EccentricityAtDCOFormation
Orbital eccentricity at DCO formation.

ECCENTRICITY_INITIAL
Header string: Eccentricity@ZAMS
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_EccentricityInitial
Orbital eccentricity at ZAMS.
Supplied by user via grid file or sampled from distribution (see –eccentricity-distribution option) (default)

ECCENTRICITY_POST_COMMON_ENVELOPE
Header string: Eccentricity>CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.postCEE.eccentricity
Eccentricity immediately following common envelope event.

ECCENTRICITY_PRE_SUPERNOVA
Header string: Eccentricity<SN
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_EccentricityPreSN
Current timestep (Myr)
Eccentricity of the binary immediately prior to supernova event

ECCENTRICITY_PRE_COMMON_ENVELOPE
Header string: Eccentricity
Data type: DOUBLE<CE
COMPAS variable: BaseBinaryStar::m_CEDetails.preCEE.eccentricity
Eccentricity at the onset of RLOF leading to the CE.

ECCENTRICITY_PRIME
Header string: Eccentricity
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_EccentricityPrime
Post-SN eccentricity.

ERROR
Header string: Error
Data type: INT
COMPAS variable: <derived from BaseBinaryStar::m_Error>
Error number (if error condition exists, else 0).

ID
Header string: ID
Data type: UNSIGNED LONG INT
COMPAS variable: BaseBinaryStar::m_ObjectId
Unique object identifier for c++ object – used in debugging to identify objects.

IMMEDIATE_RLOF_POST_COMMON_ENVELOPE
Header string: Immediate_RLOF>CE
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_RLOFDetails.immediateRLOFPostCEE
Flag to indicate if either star overflows its Roche lobe immediately following common envelope event.

LUMINOUS_BLUE_VARIABLE_FACTOR
Header string: LBV_Multiplier
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_LBVfactor
Luminous blue variable wind mass loss multiplier
User-supplied via option --luminous-blue-variable-multiplier.

MASS_1_FINAL
Header string: Core_Mass_1
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_Mass1Final
Mass of the primary star (Msol) after losing its envelope (assumes complete loss of envelope).

MASS_1_POST_COMMON_ENVELOPE
Header string: Mass_1>CE
Data type: DOUBE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.mass
Mass of the primary star (Msol) immediately following common envelope event.

MASS_1_PRE_COMMON_ENVELOPE
Header string: Mass_1<CE
Data type: DOUBE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.mass
Mass of the primary star (Msol) immediately prior to common envelope event.

MASS_2_FINAL
Header string: Core_Mass_2
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_Mass2Final
Mass of the secondary star (Msol) after losing its envelope (assumes complete loss of envelope).

MASS_2_POST_COMMON_ENVELOPE
Header string: Mass_2>CE
Data type: DOUBE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.mass
Mass of the secondary star (Msol) immediately following common envelope event.

MASS_2_PRE_COMMON_ENVELOPE
Header string: Mass_2<CE
Data type: DOUBE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.mass
Mass of the secondary star (Msol) immediately prior to common envelope event.

MASS_ENV_1
Header string: Mass_Env_1
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_MassEnv1
Envelope mass of the primary star (Msol).

MASS_ENV_2
Header string: Mass_Env_2
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_MassEnv2
Envelope mass of the secondary star (Msol).

MASSES_EQUILIBRATED
Header string: Equilibrated
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_MassesEquilibrated
Flag to indicate whether chemically homogeneous stars had masses equilibrated and orbit circularised due to Roche lobe overflow during evolution.

MASSES_EQUILIBRATED_AT_BIRTH
Header string: Equilibrated_At_Birth
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_MassesEquilibratedAtBirth
Flag to indicate whether stars had masses equilibrated and orbit circularised at birth due to Roche lobe overflow.

MASS_TRANSFER_TRACKER_HISTORY
Header string: MT_History
Data type: INT
COMPAS variable: <derived from BaseBinaryStar::m_MassTransferTrackerHistory>
Indicator of mass transfer history for the binary.
Will be printed as one of:

NO_MASS_TRANSFER	= 0
STABLE_FROM_1_TO_2	= 1
STABLE_FROM_2_TO_1	= 2
CE_FROM_1_TO_2	= 3
CE_FROM_2_TO_1	= 4
CE_DOUBLE_CORE	= 5
CE_BOTH_MS	= 6
CE_MS_WITH_CO	= 7

MERGES_IN_HUBBLE_TIME
Header string: Merges_Hubble_Time

Data type: BOOL
COMPAS variable: BaseBinaryStar::m_MergesInHubbleTime
Flag to indicate if the binary compact remnants merge within a Hubble time.

OPTIMISTIC_COMMON_ENVELOPE
Header string: Optimistic_CE
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_CEDetails.optimisticCE
Flag that returns TRUE if we have a Hertzsprung-gap star, and we allow it to survive the CE.

ORBITAL_VELOCITY
Header string: Orbital_Velocity
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_OrbitalVelocity
Orbital velocity (km s^-1).

ORBITAL_VELOCITY_PRE_SUPERNOVA
Header string: Orbital_Velocity<SN
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_OrbitalVelocityPreSN
Orbital velocity (km s^-1) immediately prior to supernova event.

RADIUS_1_POST_COMMON_ENVELOPE
Header string: Radius_1>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.radius
Radius of the primary star (Rsol) immediately following common envelope event.

RADIUS_1_PRE_COMMON_ENVELOPE
Header string: Radius_1<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.radius
Radius of the primary star (Rsol) at the onset of RLOF leading to the common-envelope episode.

RADIUS_2_POST_COMMON_ENVELOPE
Header string: Radius_2>CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.radius
Radius of the secondary star (Rsol) immediately following common envelope event.

RADIUS_2_PRE_COMMON_ENVELOPE
Header string: Radius_2<CE
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_CEDetails.preCEE.radius
Radius of the secondary star (Rsol)at the onset of RLOF leading to the common-envelope episode.

RANDOM_SEED
Header string: SEED
Data type: UNSIGNED LONG
COMPAS variable: BaseBinaryStar::m_RandomSeed
Seed for random number generator for this binary star.
(supplied by user via option—random-seed, default generated from system time)

ROCHE_LOBE_RADIUS_1
Header string: RocheLobe_1/a
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_RocheLobeRadius
Roche radius of the primary star (Rsol).

ROCHE_LOBE_RADIUS_2
Header string: RocheLobe_2/a
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_RocheLobeRadius
Roche radius of the secondary star (Rsol).

ROCHE_LOBE_RADIUS_1_POST_COMMON_ENVELOPE
Header string: RocheLobe_1>CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.postCEE.rocheLobe1to2
Roche radius of the primary star (Rsol) immediately following common envelope event.

ROCHE_LOBE_RADIUS_2_POST_COMMON_ENVELOPE
Header string: RocheLobe_2>CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.postCEE.rocheLobe2to1
Roche radius of the secondary star (Rsol) immediately following common envelope event.

ROCHE_LOBE_RADIUS_1_PRE_COMMON_ENVELOPE
Header string: RocheLobe_1<CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.preCEE.rocheLobe1to2
Roche radius of the primary star (Rsol) at the onset of RLOF leading to the common-envelope episode.

ROCHE_LOBE_RADIUS_2_PRE_COMMON_ENVELOPE
Header string: RocheLobe_2<CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.preCEE.rocheLobe2to1
Roche radius of the secondary star (Rsol) at the onset of RLOF leading to the common-envelope episode.

ROCHE_LOBE_TRACKER_1
Header string: Radius_1/RL
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_RocheLobeTracker
Ratio of the primary star's stellar radius to Roche radius (R/RL), evaluated at periapsis

ROCHE_LOBE_TRACKER_2
Header string: Radius_2/RL
Data type: DOUBLE
COMPAS variable: BinaryConstituentStar::m_RocheLobeTracker
Ratio of the secondary star's stellar radius to Roche radius (R/RL), evaluated at periapsis.

SECONDARY_TOO_SMALL_FOR_DCO
Header string: Secondary<<DCO
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_SecondaryTooSmallForDCO
Flag to indicate that the secondary star was born too small for the binary to evolve into a DCO.

SEMI_MAJOR_AXIS_AT_DCO_FORMATION
Header string: Separation@DCO
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SemiMajorAxisAtDCOFormation
Semi-major axis(AU) at DCO formation.

SEMI_MAJOR_AXIS_INITIAL
Header string: Separation@ZAMS
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SemiMajorAxisInitial
Semi-major axis (AU) at ZAMS.

SEMI_MAJOR_AXIS_POST_COMMON_ENVELOPE
Header string: Separation>CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.postCEE.semiMajorAxis
Semi-major axis (AU) immediately following common envelope event.

SEMI_MAJOR_AXIS_PRE_SUPERNOVA
Header string: Separation<SN
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SemiMajorAxisPreSN
Semi-major axis (AU) immediately prior to supernova event.
Note that this property has the same header string as SEMI_MAJOR_AXIS_PRE_SUPERNOVA_RSOL below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SEMI_MAJOR_AXIS_PRE_SUPERNOVA_RSOL
Header string: Separation<SN
Data type: DOUBLE
COMPAS variable: <derived from BaseBinaryStar::m_SemiMajorAxisPreSN>
Semi-major axis (Rsol) immediately prior to supernova event.
Note that this property has the same header string as SEMI_MAJOR_AXIS_PRE_SUPERNOVA above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SEMI_MAJOR_AXIS_PRE_COMMON_ENVELOPE
Header string: Separation<CE
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_CEDetails.preCEE.semiMajorAxis
Semi-major axis(AU) at the onset of RLOF leading to the common-envelope episode.

SEMI_MAJOR_AXIS_PRIME
Header string: Separation
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SemiMajorAxisPrime
Semi-major axis of the binary (AU).
Note that this property has the same header string as SEMI_MAJOR_AXIS_PRIME_RSOL below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SEMI_MAJOR_AXIS_PRIME_RSOL
Header string: Separation
Data type: DOUBLE
COMPAS variable: <derived from BaseBinaryStar::m_SemiMajorAxisPrime>
Semi-major axis of the binary (RSOL).
Note that this property has the same header string as SEMI_MAJOR_AXIS_PRIME above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SIMULTANEOUS_RLOF
Header string: Simultaneous_RLOF
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_RLOFDetails.simultaneousRLOF
Flag to indicate that both stars are undergoing RLOF.

STABLE_RLOF_POST_COMMON_ENVELOPE
Header string: Stable_RLOF>CE
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_RLOFDetails.stableRLOFPostCEE
Flag to indicate stable mass transfer after common envelope event.

STELLAR_MERGER
Header string: Merger
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_StellarMerger
Flag to indicate the stars merged (stars were touching) during evolution.

STELLAR_MERGER_AT_BIRTH
Header string: Merger_At_Birth
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_StellarMergerAtBirth
Flag to indicate the stars merged (stars were touching) at birth.

STELLAR_TYPE_1_POST_COMMON_ENVELOPE
Header string: Stellar_Type_1>CE
Data type: INT
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type (per Hurley at al. 2000) of the primary star immediately following common envelope event.
Note that this property has the same header string as STELLAR_TYPE_NAME_1_POST_COMMON_ENVELOPE below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_1_PRE_COMMON_ENVELOPE
Header string: Stellar_Type_1<CE
Data type: INT
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type (per Hurley at al. 2000) of the primary star at the onset of RLOF leading to the common-envelope episode.
Note that this property has the same header string as STELLAR_TYPE_NAME_1_PRE_COMMON_ENVELOPE below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_2_POST_COMMON_ENVELOPE
Header string: Stellar_Type_2>CE
Data type: INT
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type of the secondary star immediately following common envelope event.
Note that this property has the same header string as STELLAR_TYPE_NAME_2_POST_COMMON_ENVELOPE below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_2_PRE_COMMON_ENVELOPE
Header string: Stellar_Type_2<CE
Data type: INT
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type (per Hurley at al. 2000) of the secondary star at the onset of RLOF leading to the common-envelope episode.
Note that this property has the same header string as STELLAR_TYPE_NAME_2_PRE_COMMON_ENVELOPE below. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_NAME_1_POST_COMMON_ENVELOPE
Header string: Stellar_Type_1>CE
Data type: STRING
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type name (per Hurley at al. 2000) of the primary star immediately following common envelope event.
e.g. "First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE_1_POST_COMMON_ENVELOPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_NAME_1_PRE_COMMON_ENVELOPE
Header string: Stellar_Type_1<CE
Data type: STRING
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type name (per Hurley at al. 2000) of the primary star at the onset of RLOF leading to the common-envelope episode.
e.g. "First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE_1_PRE_COMMON_ENVELOPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_NAME_2_POST_COMMON_ENVELOPE
Header string: Stellar_Type_2>CE
Data type: STRING
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type name (per Hurley at al. 2000) of the secondary star immediately following common envelope event.
e.g. "First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE_2_POST_COMMON_ENVELOPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

STELLAR_TYPE_NAME_2_PRE_COMMON_ENVELOPE
Header string: Stellar_Type_2<CE
Data type: STRING
COMPAS variable: BinaryConstituentStar::m_CEDetails.postCEE.stellarType
Stellar type name (per Hurley at al. 2000) of the secondary star at the onset of RLOF leading to the common-envelope episode.
e.g. "First_Giant_Branch", "Core_Helium_Burning", "Early_Asymptotic_Giant_Branch", etc.
Note that this property has the same header string as STELLAR_TYPE_2_PRE_COMMON_ENVELOPE above. It is expected that one or the other is printed in any file, but not both. If both are printed then the file will contain two columns with the same header string.

SUPERNOVA_STATE
Header string: Supernova_State
Data type: INT
COMPAS variable: <derived from BaseBinaryStar::m_SupernovaState>
Indicates which star(s) went supernova.
Will be printed as one of:

0 = no supernova
1 = Star 1 is the supernova
2 = Star 2 is the supernova
3 = Both stars are supernovae

SYNCHRONIZATION_TIMESCALE
Header string: Tau_Sync
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SynchronizationTimescale
Tidal synchronisation timescale (Myr).

SYSTEMIC_VELOCITY
Header string: Systemic_Velocity
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_SystemicVelocity
Post-supernova systemic (center-of-mass) velocity (km s^-1).

TIME
Header string: Time
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_Time
Time (Myrs) since ZAMS.

TIME_TO_COALESCENCE
Header string: Coalescence_Time
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_TimeToCoalescence
Time between formation of double compact object and gravitational-wave driven merger (Myr).

TOTAL_ANGULAR_MOMENTUM_PRIME
Header string: Ang_Momentum_Total
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_TotalAngularMomentumPrime
Total angular momentum calculated using regular conservation of energy (Msol AU^2 yr^-1).

TOTAL_ENERGY_PRIME
Header string: Energy_Total
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_TotalAngularMomentumPrime
Total energy calculated using regular conservation of energy (Msol AU^2 yr^-1).

UNBOUND
Header string: Unbound
Data type: BOOL
COMPAS variable: BaseBinaryStar::m_Unbound
Flag to indicate the binary is unbound (or has become unbound after a supernova event).

WOLF_RAYET_FACTOR
Header string: WR_Multiplier
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_WolfRayetFactor
Multiplicative constant for Wolf-Rayet mass loss rate
User-supplied via option --wolf-rayet-multiplier.

ZETA_LOBE
Header string: Zeta_Lobe
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_ZetaLobe
The logarithmic derivative of Roche lobe radius with respect to donor mass for q = Md / Ma at the onset of the RLOF.

ZETA_STAR
Header string: Zeta_Star
Data type: DOUBLE
COMPAS variable: BaseBinaryStar::m_ZetaStar
Mass-radius exponent of the star at the onset of the RLOF. Calculated differently based on the value of several options:

    --common-envelope-zeta-prescription
    --forceCaseBBBCStabilityFlag,
    --alwaysStableCaseBBBCFlag
    --zeta-radiative-envelope-giant

Appendix D – Log File Record Specification: Program Options

As described in **Standard Log File Record Specifiers**, when specifying known properties in a log file record specification record, the property name must be prefixed with the property type.

Currently there is a single program option property type available for use: PROGRAM_OPTION.

For example, to specify the program option property RANDOM_SEED, use:

PROGRAM_OPTION::RANDOM_SEED

Following is the list of program option properties available for inclusion in log file record specifiers.

Program Option Properties

KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_BH
Header string: Sigma_Kick_CCSN_BH
Data type: DOUBLE
COMPAS variable: Options::kickVelocityDistributionSigmaCCSN_BH
Value of program option --kick-velocity-sigma-CCSN-BH

KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_NS
Header string: Sigma_Kick_CCSN_NS
Data type: DOUBLE
COMPAS variable: Options::kickVelocityDistributionSigmaCCSN_BH
Value of program option --kick-velocity-sigma-CCSN-NS

KICK_VELOCITY_DISTRIBUTION_SIGMA_FOR_ECSN
Header string: Sigma_Kick_ECSN
Data type: DOUBLE
COMPAS variable: Options::kickVelocityDistributionSigmaForECSN
Value of program option --kick-velocity-sigma-ECSN

KICK_VELOCITY_DISTRIBUTION_SIGMA_FOR_USSN
Header string: Sigma_Kick_USSN
Data type: DOUBLE
COMPAS variable: Options::kickVelocityDistributionSigmaForUSSN
Value of program option --kick-velocity-sigma-USSN

RANDOM_SEED
Header string: SEED_(ProgramOption)
Data type: UNSIGNED LONG INT
COMPAS variable: Options::randomSeed
Value of program option –random-seed

Appendix E – Default Log File Record Specifications

Following are the default log file record specifications for each of the standard log files. These specifications can be overridden by the use of a log file specifications file via the `logfile-definitions` program option.

SSE Parameters

```
const ANY_PROPERTY_VECTOR SSE_PARAMETERS_REC = {
   STAR_PROPERTY::AGE,
   STAR_PROPERTY::DT,
   STAR_PROPERTY::TIME,
   STAR_PROPERTY::STELLAR_TYPE,
   STAR_PROPERTY::METALLICITY,
   STAR_PROPERTY::MASS_0,
   STAR_PROPERTY::MASS,
   STAR_PROPERTY::RADIUS,
   STAR_PROPERTY::RZAMS,
   STAR_PROPERTY::LUMINOSITY,
   STAR_PROPERTY::TEMPERATURE,
   STAR_PROPERTY::CORE_MASS,
   STAR_PROPERTY::CO_CORE_MASS,
   STAR_PROPERTY::HE_CORE_MASS,
   STAR_PROPERTY::MDOT,
   STAR_PROPERTY::TIMESCALE_MS
};
```

BSE System Parameters

```
const ANY_PROPERTY_VECTOR BSE_SYSTEM_PARAMETERS_REC = {
   BINARY_PROPERTY::ID,
   BINARY_PROPERTY::RANDOM_SEED,
   STAR_1_PROPERTY::MZAMS,
   STAR_2_PROPERTY::MZAMS,
   BINARY_PROPERTY::SEMI_MAJOR_AXIS_INITIAL,
   BINARY_PROPERTY::ECCENTRICITY_INITIAL,
   STAR_1_PROPERTY::SUPERNOVA_KICK_VELOCITY_MAGNITUDE_RANDOM_NUMBER,
   STAR_1_PROPERTY::SUPERNOVA_THETA,
   STAR_1_PROPERTY::SUPERNOVA_PHI,
   STAR_1_PROPERTY::MEAN_ANOMALY,
   STAR_2_PROPERTY::SUPERNOVA_KICK_VELOCITY_MAGNITUDE_RANDOM_NUMBER,
   STAR_2_PROPERTY::SUPERNOVA_THETA,
   STAR_2_PROPERTY::SUPERNOVA_PHI,
   STAR_2_PROPERTY::MEAN_ANOMALY,
   STAR_1_PROPERTY::OMEGA_ZAMS,
   STAR_2_PROPERTY::OMEGA_ZAMS,
   PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_NS,
   PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_BH,
   PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_FOR_ECSN,
   PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_FOR_USSN,
   BINARY_PROPERTY::LUMINOUS_BLUE_VARIABLE_FACTOR,
   BINARY_PROPERTY::WOLF_RAYET_FACTOR,
   BINARY_PROPERTY::COMMON_ENVELOPE_ALPHA,
   STAR_1_PROPERTY::METALLICITY,
   STAR_2_PROPERTY::METALLICITY,
   BINARY_PROPERTY::UNBOUND,
   BINARY_PROPERTY::STELLAR_MERGER,
   BINARY_PROPERTY::STELLAR_MERGER_AT_BIRTH,
   STAR_1_PROPERTY::INITIAL_STELLAR_TYPE,
   STAR_1_PROPERTY::STELLAR_TYPE,
   STAR_2_PROPERTY::INITIAL_STELLAR_TYPE,
   STAR_2_PROPERTY::STELLAR_TYPE,
   BINARY_PROPERTY::ERROR
};
```

BSE Detailed Output

```
const ANY_PROPERTY_VECTOR BSE_DETAILED_OUTPUT_REC = {
    BINARY_PROPERTY::ID,
    BINARY_PROPERTY::RANDOM_SEED,
    BINARY_PROPERTY::DT,
    BINARY_PROPERTY::TIME,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRIME_RSOL,
    BINARY_PROPERTY::ECCENTRICITY_PRIME,
    STAR_1_PROPERTY::MZAMS,
    STAR_2_PROPERTY::MZAMS,
    STAR_1_PROPERTY::MASS_0,
    STAR_2_PROPERTY::MASS_0,
    STAR_1_PROPERTY::MASS,
    STAR_2_PROPERTY::MASS,
    STAR_1_PROPERTY::ENV_MASS,
    STAR_2_PROPERTY::ENV_MASS,
    STAR_1_PROPERTY::CORE_MASS,
    STAR_2_PROPERTY::CORE_MASS,
    STAR_1_PROPERTY::HE_CORE_MASS,
    STAR_2_PROPERTY::HE_CORE_MASS,
    STAR_1_PROPERTY::CO_CORE_MASS,
    STAR_2_PROPERTY::CO_CORE_MASS,
    STAR_1_PROPERTY::RADIUS,
    STAR_2_PROPERTY::RADIUS,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_1,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_2,
    BINARY_PROPERTY::ROCHE_LOBE_TRACKER_1,
    BINARY_PROPERTY::ROCHE_LOBE_TRACKER_2,
    STAR_1_PROPERTY::OMEGA,
    STAR_2_PROPERTY::OMEGA,
    STAR_1_PROPERTY::OMEGA_BREAK,
    STAR_2_PROPERTY::OMEGA_BREAK,
    STAR_1_PROPERTY::INITIAL_STELLAR_TYPE,
    STAR_2_PROPERTY::INITIAL_STELLAR_TYPE,
    STAR_1_PROPERTY::STELLAR_TYPE,
    STAR_2_PROPERTY::STELLAR_TYPE,
    STAR_1_PROPERTY::AGE,
    STAR_2_PROPERTY::AGE,
    STAR_1_PROPERTY::LUMINOSITY,
    STAR_2_PROPERTY::LUMINOSITY,
    STAR_1_PROPERTY::TEMPERATURE,
    STAR_2_PROPERTY::TEMPERATURE,
    STAR_1_PROPERTY::ANGULAR_MOMENTUM,
    STAR_2_PROPERTY::ANGULAR_MOMENTUM,
    STAR_1_PROPERTY::DYNAMICAL_TIMESCALE,
    STAR_2_PROPERTY::DYNAMICAL_TIMESCALE,
    STAR_1_PROPERTY::THERMAL_TIMESCALE,
    STAR_2_PROPERTY::THERMAL_TIMESCALE,
    STAR_1_PROPERTY::NUCLEAR_TIMESCALE,
    STAR_2_PROPERTY::NUCLEAR_TIMESCALE,
    STAR_1_PROPERTY::ZETA_THERMAL,
    STAR_2_PROPERTY::ZETA_THERMAL,
    STAR_1_PROPERTY::ZETA_NUCLEAR,
    STAR_2_PROPERTY::ZETA_NUCLEAR,
    STAR_1_PROPERTY::ZETA_SOBERMAN,
    STAR_2_PROPERTY::ZETA_SOBERMAN,
    STAR_1_PROPERTY::ZETA_SOBERMAN_HE,
    STAR_2_PROPERTY::ZETA_SOBERMAN_HE,
    STAR_1_PROPERTY::ZETA_HURLEY,
    STAR_2_PROPERTY::ZETA_HURLEY,
    STAR_1_PROPERTY::ZETA_HURLEY_HE,
    STAR_2_PROPERTY::ZETA_HURLEY_HE,
    STAR_1_PROPERTY::MASS_LOSS_DIFF,
    STAR_2_PROPERTY::MASS_LOSS_DIFF,
    STAR_1_PROPERTY::MASS_TRANSFER_DIFF,
    STAR_2_PROPERTY::MASS_TRANSFER_DIFF,
    BINARY_PROPERTY::TOTAL_ANGULAR_MOMENTUM_PRIME,
    BINARY_PROPERTY::TOTAL_ENERGY_PRIME,
    STAR_1_PROPERTY::LAMBDA_NANJING,
    STAR_2_PROPERTY::LAMBDA_NANJING,
    STAR_1_PROPERTY::LAMBDA_LOVERIDGE,
    STAR_2_PROPERTY::LAMBDA_LOVERIDGE,
    STAR_1_PROPERTY::LAMBDA_KRUCKOW_TOP,
    STAR_2_PROPERTY::LAMBDA_KRUCKOW_TOP,
    STAR_1_PROPERTY::LAMBDA_KRUCKOW_MIDDLE,
    STAR_2_PROPERTY::LAMBDA_KRUCKOW_MIDDLE,
    STAR_1_PROPERTY::LAMBDA_KRUCKOW_BOTTOM,
    STAR_2_PROPERTY::LAMBDA_KRUCKOW_BOTTOM,
    STAR_1_PROPERTY::METALLICITY,
    STAR_2_PROPERTY::METALLICITY,
    BINARY_PROPERTY::MASS_TRANSFER_TRACKER_HISTORY,
    STAR_1_PROPERTY::PULSAR_MAGNETIC_FIELD,
    STAR_2_PROPERTY::PULSAR_MAGNETIC_FIELD,
    STAR_1_PROPERTY::PULSAR_SPIN_FREQUENCY,
    STAR_2_PROPERTY::PULSAR_SPIN_FREQUENCY,
    STAR_1_PROPERTY::PULSAR_SPIN_DOWN_RATE,
    STAR_2_PROPERTY::PULSAR_SPIN_DOWN_RATE,
    STAR_1_PROPERTY::RADIAL_EXPANSION_TIMESCALE,
    STAR_2_PROPERTY::RADIAL_EXPANSION_TIMESCALE
};
```

BSE Double Compact Objects

```
const ANY_PROPERTY_VECTOR BSE_DOUBLE_COMPACT_OBJECTS_REC = {
    BINARY_PROPERTY::ID,
    BINARY_PROPERTY::RANDOM_SEED,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_AT_DCO_FORMATION,
    BINARY_PROPERTY::ECCENTRICITY_AT_DCO_FORMATION,
    STAR_1_PROPERTY::MASS,
    STAR_1_PROPERTY::STELLAR_TYPE,
    STAR_2_PROPERTY::MASS,
    STAR_2_PROPERTY::STELLAR_TYPE,
    BINARY_PROPERTY::TIME_TO_COALESCENCE,
    BINARY_PROPERTY::TIME,
    STAR_1_PROPERTY::MASS_TRANSFER_CASE_INITIAL,
    STAR_2_PROPERTY::MASS_TRANSFER_CASE_INITIAL,
    BINARY_PROPERTY::MERGES_IN_HUBBLE_TIME,
    STAR_1_PROPERTY::RECYCLED_NEUTRON_STAR,
    STAR_2_PROPERTY::RECYCLED_NEUTRON_STAR,
};
```

BSE Common Envelopes

```
const ANY_PROPERTY_VECTOR BSE_COMMON_ENVELOPES_REC = {
    BINARY_PROPERTY::ID,
    BINARY_PROPERTY::RANDOM_SEED,
    BINARY_PROPERTY::TIME,
    STAR_1_PROPERTY::LAMBDA_AT_COMMON_ENVELOPE,
    STAR_2_PROPERTY::LAMBDA_AT_COMMON_ENVELOPE,
    STAR_1_PROPERTY::BINDING_ENERGY_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::BINDING_ENERGY_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::ECCENTRICITY_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::ECCENTRICITY_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_1_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_1_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_2_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::ROCHE_LOBE_RADIUS_2_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::MASS_1_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::MASS_ENV_1,
    BINARY_PROPERTY::MASS_1_FINAL,
    BINARY_PROPERTY::RADIUS_1_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::RADIUS_1_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::STELLAR_TYPE_1_PRE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::STELLAR_TYPE,
    STAR_1_PROPERTY::LAMBDA_FIXED,
    STAR_1_PROPERTY::LAMBDA_NANJING,
    STAR_1_PROPERTY::LAMBDA_LOVERIDGE,
    STAR_1_PROPERTY::LAMBDA_LOVERIDGE_WINDS,
    STAR_1_PROPERTY::LAMBDA_KRUCKOW,
    STAR_1_PROPERTY::BINDING_ENERGY_FIXED,
    STAR_1_PROPERTY::BINDING_ENERGY_NANJING,
    STAR_1_PROPERTY::BINDING_ENERGY_LOVERIDGE,
    STAR_1_PROPERTY::BINDING_ENERGY_LOVERIDGE_WINDS,
    STAR_1_PROPERTY::BINDING_ENERGY_KRUCKOW,
    BINARY_PROPERTY::MASS_2_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::MASS_ENV_2,
    BINARY_PROPERTY::MASS_2_FINAL,
    BINARY_PROPERTY::RADIUS_2_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::RADIUS_2_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::STELLAR_TYPE_2_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::STELLAR_TYPE,
    STAR_2_PROPERTY::LAMBDA_FIXED,
    STAR_2_PROPERTY::LAMBDA_NANJING,
    STAR_2_PROPERTY::LAMBDA_LOVERIDGE,
    STAR_2_PROPERTY::LAMBDA_LOVERIDGE_WINDS,
    STAR_2_PROPERTY::LAMBDA_KRUCKOW,
    STAR_2_PROPERTY::BINDING_ENERGY_FIXED,
    STAR_2_PROPERTY::BINDING_ENERGY_NANJING,
    STAR_2_PROPERTY::BINDING_ENERGY_LOVERIDGE,
    STAR_2_PROPERTY::BINDING_ENERGY_LOVERIDGE_WINDS,
    STAR_2_PROPERTY::BINDING_ENERGY_KRUCKOW,
    BINARY_PROPERTY::MASS_TRANSFER_TRACKER_HISTORY,
    BINARY_PROPERTY::STELLAR_MERGER,
    BINARY_PROPERTY::OPTIMISTIC_COMMON_ENVELOPE,
    BINARY_PROPERTY::COMMON_ENVELOPE_EVENT_COUNT,
    BINARY_PROPERTY::DOUBLE_CORE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::IS_RLOF,
    STAR_1_PROPERTY::LUMINOSITY_PRE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::TEMPERATURE_PRE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::DYNAMICAL_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::THERMAL_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_1_PROPERTY::NUCLEAR_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::IS_RLOF,
    STAR_2_PROPERTY::LUMINOSITY_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::TEMPERATURE_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::DYNAMICAL_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::THERMAL_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::NUCLEAR_TIMESCALE_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::ZETA_STAR_COMPARE,
    BINARY_PROPERTY::ZETA_RLOF_ANALYTIC,
    BINARY_PROPERTY::SYNCHRONIZATION_TIMESCALE,
    BINARY_PROPERTY::CIRCULARIZATION_TIMESCALE,
    STAR_1_PROPERTY::RADIAL_EXPANSION_TIMESCALE_PRE_COMMON_ENVELOPE,
    STAR_2_PROPERTY::RADIAL_EXPANSION_TIMESCALE_PRE_COMMON_ENVELOPE,
    BINARY_PROPERTY::IMMEDIATE_RLOF_POST_COMMON_ENVELOPE,
    BINARY_PROPERTY::SIMULTANEOUS_RLOF
};
```

BSE Supernovae

```
const ANY_PROPERTY_VECTOR BSE_SUPERNOVAE_REC = {
    BINARY_PROPERTY::ID,
    BINARY_PROPERTY::RANDOM_SEED,
    SUPERNOVA_PROPERTY::DRAWN_KICK_VELOCITY,
    SUPERNOVA_PROPERTY::KICK_VELOCITY,
    SUPERNOVA_PROPERTY::FALLBACK_FRACTION,
    BINARY_PROPERTY::ORBITAL_VELOCITY_PRE_SUPERNOVA,
    BINARY_PROPERTY::DIMENSIONLESS_KICK_VELOCITY,
    SUPERNOVA_PROPERTY::TRUE_ANOMALY,
    SUPERNOVA_PROPERTY::SUPERNOVA_THETA,
    SUPERNOVA_PROPERTY::SUPERNOVA_PHI,
    SUPERNOVA_PROPERTY::SN_TYPE,
    BINARY_PROPERTY::UNBOUND,
    SUPERNOVA_PROPERTY::TOTAL_MASS_AT_COMPACT_OBJECT_FORMATION,
    COMPANION_PROPERTY::MASS,
    SUPERNOVA_PROPERTY::CO_CORE_MASS_AT_COMPACT_OBJECT_FORMATION,
    SUPERNOVA_PROPERTY::MASS,
    SUPERNOVA_PROPERTY::EXPERIENCED_RLOF,
    SUPERNOVA_PROPERTY::STELLAR_TYPE,
    BINARY_PROPERTY::SUPERNOVA_STATE,
    SUPERNOVA_PROPERTY::STELLAR_TYPE_PREV,
    COMPANION_PROPERTY::STELLAR_TYPE_PREV,
    SUPERNOVA_PROPERTY::CORE_MASS_AT_COMPACT_OBJECT_FORMATION,
    SUPERNOVA_PROPERTY::HE_CORE_MASS_AT_COMPACT_OBJECT_FORMATION,
    BINARY_PROPERTY::TIME,
    BINARY_PROPERTY::ECCENTRICITY_PRE_SUPERNOVA,
    BINARY_PROPERTY::ECCENTRICITY,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRE_SUPERNOVA_RSOL,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRIME_RSOL,
    BINARY_PROPERTY::SYSTEMIC_VELOCITY,
    SUPERNOVA_PROPERTY::HYDROGEN_RICH,
    SUPERNOVA_PROPERTY::HYDROGEN_POOR,
    COMPANION_PROPERTY::RUNAWAY
};
```

BSE Pulsar Evolution

```
const ANY_PROPERTY_VECTOR BSE_PULSAR_EVOLUTION_REC = {
    BINARY_PROPERTY::ID,
    BINARY_PROPERTY::RANDOM_SEED,
    STAR_1_PROPERTY::MASS,
    STAR_2_PROPERTY::MASS,
    STAR_1_PROPERTY::STELLAR_TYPE,
    STAR_2_PROPERTY::STELLAR_TYPE,
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRIME_RSOL,
    BINARY_PROPERTY::MASS_TRANSFER_TRACKER_HISTORY,
    STAR_1_PROPERTY::PULSAR_MAGNETIC_FIELD,
    STAR_2_PROPERTY::PULSAR_MAGNETIC_FIELD,
    STAR_1_PROPERTY::PULSAR_SPIN_FREQUENCY,
    STAR_2_PROPERTY::PULSAR_SPIN_FREQUENCY,
    STAR_1_PROPERTY::PULSAR_SPIN_DOWN_RATE,
    STAR_2_PROPERTY::PULSAR_SPIN_DOWN_RATE,
    BINARY_PROPERTY::TIME,
    BINARY_PROPERTY::DT
};
```

Appendix F – Example Log File Record Specifications File

Following is an example log file record specifications file. COMPAS can be configured to use this file via the `logfile-definitions` program option.

This file (`COMPAS_Output_Definitions.txt`) is also delivered as part of the COMPAS github repository.

```
# sample standard log file specifications file

# the '#' character and anything following it on a single line is considered a comment
# (so, lines starting with '#' are comment lines)

# case is not significant
# specifications can span several lines
# specifications for the same log file are cumulative
# if a log file is not specified in this file, the default specification is used


# SSE Parameters

# start with the default SSE Parameters specification and add ENV_MASS

sse_parms_rec += { STAR_PROPERTY::ENV_MASS }

# take the updated SSE Parameters specification and add ANGULAR_MOMENTUM

sse_parms_rec += { STAR_PROPERTY::ANGULAR_MOMENTUM }

# take the updated SSE Parameters specification and subtract MASS_0 and MDOT

sse_parms_rec -= { STAR_PROPERTY::MASS_0, STAR_PROPERTY::MDOT }


# BSE System Parameters

bse_sysparms_rec = {                             # set the BSE System Parameters specification to:
    BINARY_PROPERTY::ID,                         # ID of the binary
    BINARY_PROPERTY::RANDOM_SEED,                # RANDOM_SEED for the binary
    STAR_1_PROPERTY::MZAMS,                      # MZAMS for Star1
    STAR_2_PROPERTY::MZAMS                       # MZAMS for Star2
}

# ADD to the BSE System Parameters specification:
# SEMI_MAJOR_AXIS_INITIAL for the binary
# ECCENTRICITY_INITIAL for the binary
# SUPERNOVA_THETA for Star1 and SUPERNOVA_PHI for Star1

bse_sysparms_rec += {
    BINARY_PROPERTY::SEMI_MAJOR_AXIS_INITIAL,
    BINARY_PROPERTY::ECCENTRICITY_INITIAL,
    STAR_1_PROPERTY::SUPERNOVA_THETA, STAR_1_PROPERTY::SUPERNOVA_PHI
}

bse_sysparms_rec += {                                # ADD to the BSE System Parameters specification:
    SUPERNOVA_PROPERTY::IS_ECSN,                     # IS_ECSN for the supernova star
    SUPERNOVA_PROPERTY::IS_SN,                       # IS_SN for the supernova star
    SUPERNOVA_PROPERTY::IS_USSN,                     # IS_USSN for the supernova star
    SUPERNOVA_PROPERTY::EXPERIENCED_PISN,            # EXPERIENCED_PISN for the supernova star
    SUPERNOVA_PROPERTY::EXPERIENCED_PPISN,           # EXPERIENCED_PPISN for the supernova star
    BINARY_PROPERTY::SURVIVED_SUPERNOVA_EVENT,       # SURVIVED_SUPERNOVA_EVENT for the binary
    SUPERNOVA_PROPERTY::MZAMS,                       # MZAMS for the supernova star
    COMPANION_PROPERTY::MZAMS                        # MZAMS for the companion star
}

# SUBTRACT from the BSE System Parameters specification:
# RANDOM_SEED for the binary
# ID for the binary
bse_sysparms_rec -= {                         # SUBTRACT from the BSE System Parameters specification:
    BINARY_PROPERTY::RANDOM_SEED,             # RANDOM_SEED for the binary
    BINARY_PROPERTY::ID                       # ID for the binary
}


# BSE Double Compas Objects
# set the BSE Double Compact Objects specifivation to MZAMS for Star1, and MZAMS for Star2

BSE_DCO_Rec = {STAR_1_PROPERTY::MZAMS,STAR_2_PROPERTY::MZAMS}

# set the BSE Double Compact Objects specification to empty - nothing will be printed
# (file will not be created)

BSE_DCO_Rec = {}


# BSE Supernovae

BSE_SNE_Rec = {}               # set spec empty - nothing will be printed (file will not be created)


# BSE Common Envelopes

BSE_CEE_Rec = {   }            # set spec empty - nothing will be printed (file will not be created)


# BSE Pulsars

# line ignored (comment).  BSE Pulsars specification will be default

# BSE_Pulsars_Rec= { STAR_1_PROPERTY::MASS, STAR_2_PROPERTY::MASS }


# BSE Detailed Output

BSE_Detailed_Rec={}            # set spec empty - nothing will be printed (file will not be created)
```