

CORSIKA add-on package IACT/ATMO:

Version 1.50 (November 2016)

Generated by Doxygen 1.8.8

Thu Mar 23 2017 15:34:49

Contents

1	Module Index	1
1.1	Modules	1
2	Data Structure Index	1
2.1	Data Structures	1
3	File Index	2
3.1	File List	2
4	Module Documentation	3
4.1	The listio program	3
4.1.1	Detailed Description	3
4.1.2	Function Documentation	3
4.2	The testio program	4
4.2.1	Detailed Description	4
4.2.2	Function Documentation	4
5	Data Structure Documentation	8
5.1	_struct_IO_BUFFER Struct Reference	8
5.1.1	Detailed Description	9
5.1.2	Field Documentation	9
5.2	_struct_IO_ITEM_HEADER Struct Reference	9
5.2.1	Detailed Description	10
5.2.2	Field Documentation	10
5.3	bunch Struct Reference	10
5.3.1	Detailed Description	11
5.4	camera_electronics Struct Reference	11
5.4.1	Field Documentation	11
5.5	compact_bunch Struct Reference	11
5.5.1	Detailed Description	12
5.6	detstruct Struct Reference	12
5.6.1	Detailed Description	13
5.7	ev_reg_entry Struct Reference	13
5.8	gridstruct Struct Reference	14
5.9	incpath Struct Reference	14
5.9.1	Detailed Description	15
5.10	linked_string Struct Reference	15
5.10.1	Detailed Description	15
5.11	mc_options Struct Reference	15
5.11.1	Detailed Description	16

5.11.2	Field Documentation	16
5.12	mc_run Struct Reference	16
5.12.1	Field Documentation	17
5.13	photo_electron Struct Reference	18
5.13.1	Detailed Description	18
5.13.2	Field Documentation	18
5.14	pm_camera Struct Reference	18
5.15	shower_extra_parameters Struct Reference	19
5.15.1	Detailed Description	19
5.15.2	Field Documentation	19
5.16	simulated_shower_parameters Struct Reference	20
5.16.1	Field Documentation	21
5.17	telescope_array Struct Reference	21
5.17.1	Detailed Description	22
5.17.2	Field Documentation	22
5.18	telescope_optics Struct Reference	23
5.19	test_struct Struct Reference	23
5.20	warn_specific_data Struct Reference	24
5.20.1	Detailed Description	24
5.20.2	Field Documentation	24
6	File Documentation	24
6.1	atmo.c File Reference	24
6.1.1	Detailed Description	26
6.1.2	Function Documentation	27
6.1.3	Variable Documentation	30
6.2	atmo.h File Reference	30
6.2.1	Detailed Description	31
6.2.2	Function Documentation	32
6.3	eventio.c File Reference	34
6.3.1	Detailed Description	39
6.3.2	Macro Definition Documentation	41
6.3.3	Function Documentation	41
6.4	fileopen.c File Reference	61
6.4.1	Detailed Description	62
6.4.2	Function Documentation	63
6.4.3	Variable Documentation	64
6.5	fileopen.h File Reference	64
6.5.1	Detailed Description	65
6.5.2	Function Documentation	65

6.6	iact.c File Reference	65
6.6.1	Detailed Description	71
6.6.2	Function Documentation	71
6.6.3	Variable Documentation	80
6.7	iact.h File Reference	80
6.7.1	Detailed Description	81
6.7.2	Function Documentation	81
6.8	initial.h File Reference	86
6.8.1	Detailed Description	87
6.9	io_basic.h File Reference	88
6.9.1	Detailed Description	94
6.9.2	Macro Definition Documentation	94
6.9.3	Function Documentation	94
6.10	io_simtel.c File Reference	111
6.10.1	Detailed Description	114
6.10.2	Function Documentation	114
6.10.3	Variable Documentation	126
6.11	listio.c File Reference	126
6.11.1	Detailed Description	126
6.12	mc_tel.h File Reference	127
6.12.1	Detailed Description	130
6.12.2	Function Documentation	130
6.13	sampling.h File Reference	142
6.13.1	Function Documentation	142
6.14	sim_skeleton.c File Reference	142
6.14.1	Detailed Description	144
6.14.2	Macro Definition Documentation	144
6.14.3	Function Documentation	145
6.15	straux.c File Reference	145
6.15.1	Detailed Description	146
6.15.2	Function Documentation	146
6.16	straux.h File Reference	147
6.16.1	Detailed Description	148
6.16.2	Function Documentation	148
6.17	testio.c File Reference	149
6.17.1	Detailed Description	150
6.18	warning.c File Reference	150
6.18.1	Detailed Description	152
6.18.2	Function Documentation	152
6.18.3	Variable Documentation	154

6.19 warning.h File Reference	154
6.19.1 Detailed Description	156
6.19.2 Function Documentation	156

1 Module Index

1.1 Modules

Here is a list of all modules:

The listio program	3
The testio program	4

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

_struct_IO_BUFFER The <code>IO_BUFFER</code> structure contains all data needed the manage the stuff	8
_struct_IO_ITEM_HEADER An <code>IO_ITEM_HEADER</code> is to access header info for an I/O block and as a handle to the I/O buffer	9
bunch Photons collected in bunches of identical direction, position, time, and wavelength	10
camera_electronics Parameters of the electronics of a telescope	11
compact_bunch The <code>compact_bunch</code> struct is equivalent to the <code>bunch</code> struct except that we try to use less memory	11
detstruct A structure describing a detector and linking its photons bunches to it	12
ev_reg_entry	13
gridstruct	14
incpath An element in a linked list of include paths	14
linked_string The <code>linked_string</code> is mainly used to keep CORSIKA input	15
mc_options Options of the simulation passed through to low-level functions	15
mc_run Basic parameters of the CORSIKA run	16

photo_electron	A photo-electron produced by a photon hitting a pixel	18
pm_camera	Parameters of a telescope camera (pixels, ...)	18
shower_extra_parameters	Extra shower parameters of unspecified nature	19
simulated_shower_parameters	Basic parameters of a simulated shower	20
telescope_array	Description of telescope position, array offsets and shower parameters	21
telescope_optics	Parameters describing the telescope optics	23
test_struct		23
warn_specific_data	A struct used to store thread-specific data	24

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

atmo.c	Use of tabulated atmospheric profiles and atmospheric refraction	24
atmo.h	Use of tabulated atmospheric profiles and atmospheric refraction	30
eventio.c	Basic functions for eventio data format	34
fileopen.c	Allow searching of files in declared include paths (fopen replacement)	61
fileopen.h	Function prototypes for fileopen.c	64
iact.c	CORSIKA interface for Imaging Atmospheric Cherenkov Telescopes etc	65
iact.h	Function declarations for CORSIKA IACT interface	80
initial.h	Identification of the system and including some basic include file	86
io_basic.h	Basic header file for eventio data format	88
io_simtel.c	Write and read CORSIKA blocks and simulated Cherenkov photon bunches	111

listio.c	
Main function for listing data consisting of eventio blocks	126
mc_tel.h	
Definitions and structures for CORSIKA Cherenkov light interface	127
sampling.h	142
sim_skeleton.c	
A (non-functional) skeleton program for reading CORSIKA IACT data	142
straux.c	
Check for abbreviations of strings and get words from strings	145
straux.h	
Check for abbreviations of strings and get words from strings	147
testio.c	
Test program for eventio data format	149
warning.c	
Pass warning messages to the screen or a usr function as set up	150
warning.h	
Pass warning messages to the screen or a usr function as set up	154

4 Module Documentation

4.1 The listio program

Functions

- int [main](#) (int argc, char **argv)
 Main function.

4.1.1 Detailed Description

4.1.2 Function Documentation

4.1.2.1 int main (int argc, char ** argv)

The main function of the listio program.

References [allocate_io_buffer\(\)](#), [fileopen\(\)](#), [find_io_block\(\)](#), [_struct_IO_BUFFER::input_file](#), [list_io_blocks\(\)](#), [list_io_sub_items\(\)](#), [_struct_IO_BUFFER::max_length](#), and [read_io_block\(\)](#).

4.2 The testio program

Data Structures

- struct `test_struct`

Typedefs

- typedef struct `test_struct` `TEST_DATA`

Functions

- int `datacmp` (`TEST_DATA` *data1, `TEST_DATA` *data2)
Compare elements of test data structures.
- int `main` (int argc, char **argv)
Main function for I/O test program.
- int `read_test1` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Read test data with single-element functions.
- int `read_test2` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Read test data with vector functions as far as possible.
- int `read_test3` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Read test data as a nested tree.
- void `syntax` (const char *prg)
Replacement for function missing on OS-9.
- int `write_test1` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Write test data with single-element functions.
- int `write_test2` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Write test data with vector functions as far as possible.
- int `write_test3` (`TEST_DATA` *data, `IO_BUFFER` *iobuf)
Write test data in nested items.

Variables

- static int `care_int`
- static int `care_long`
- static int `care_short`

4.2.1 Detailed Description

4.2.2 Function Documentation

4.2.2.1 int datacmp (`TEST_DATA` * data1, `TEST_DATA` * data2)

Compare elements of test data structures with the accuracy relevant to the I/O package.

Parameters

<i>data1</i>	first data structure
<i>data2</i>	second data structure

Returns

0 (something did not match), 1 (O.K.)

4.2.2.2 `int main (int argc, char ** argv)`

First writes a test data structure with the vector functions, then the same data structure with the single-element functions. The output file is then closed and reopened for reading. The first structure is then read with the single-element functions and the second with the vector functions (i.e. the other way as done for writing). The data from the file is compared with the original data, taking the relevant accuracy into account. Note that if an 'int' variable is written via '`put_short()`' and then read again via '`get_short()`' not only the upper two bytes (on a 32-bit machine) are lost but also the sign bit is propagated from bit 15 to the upper 16 bits. Similarly, if a 'long' variable is written via '`put_long()`' and later read via '`get_long()`' on a 64-bit-machine, not only the upper 4 bytes are lost but also the sign in bit 31 is propagated to the upper 32 bits.

References `allocate_io_buffer()`, `_struct_IO_BUFFER::byte_order`, `datacmp()`, `_struct_IO_BUFFER::extended`, `file_close()`, `fileopen()`, `find_io_block()`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::output_file`, `read_io_block()`, `read_test1()`, `read_test2()`, `read_test3()`, `syntax()`, `write_test1()`, `write_test2()`, and `write_test3()`.

4.2.2.3 `int read_test1 (TEST_DATA * data, IO_BUFFER * iobuf)`

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (ok), <0 (error as for `get_item_end()`)

References `get_count()`, `get_count16()`, `get_count32()`, `get_double()`, `get_int32()`, `get_item_begin()`, `get_item_end()`, `get_long()`, `get_long_string()`, `get_real()`, `get_scount()`, `get_scount16()`, `get_scount32()`, `get_sfloat()`, `get_short()`, `get_string()`, `get_uint32()`, `get_var_string()`, and `_struct_IO_ITEM_HEADER::type`.

4.2.2.4 `int read_test2 (TEST_DATA * data, IO_BUFFER * iobuf)`

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (ok), <0 (error as for `get_item_end()`)

References `get_count()`, `get_item_begin()`, `get_item_end()`, `get_long()`, `get_long_string()`, `get_scount()`, `get_scount16()`, `get_scount32()`, `get_sfloat()`, `get_string()`, `get_var_string()`, `get_vector_of_byte()`, `get_vector_of_double()`, `get_vector_of_int()`, `get_vector_of_int32()`, `get_vector_of_long()`, `get_vector_of_real()`, `get_vector_of_short()`, `get_vector_of_uint32()`, and `_struct_IO_ITEM_HEADER::type`.

4.2.2.5 `int read_test3 (TEST_DATA * data, IO_BUFFER * iobuf)`

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (ok), <0 (error as for `get_item_end()`)

References `get_count()`, `get_item_begin()`, `get_item_end()`, `get_long()`, `get_long_string()`, `get_scount()`, `get_scount16()`, `get_scount32()`, `get_sfloat()`, `get_string()`, `get_var_string()`, `get_vector_of_byte()`, `get_vector_of_double()`, `get_vector_of_int()`, `get_vector_of_int32()`, `get_vector_of_long()`, `get_vector_of_real()`, `get_vector_of_short()`, `get_vector_of_uint32()`, `next_subitem_type()`, `rewind_item()`, `search_sub_item()`, and `_struct_IO_ITEM_HEADER::type`.

4.2.2.6 int write_test1 (TEST_DATA * *data*, IO_BUFFER * *iobuf*)

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (O.K.), <0 (error as for [put_item_end\(\)](#))

References `_struct_IO_ITEM_HEADER::ident`, `put_count()`, `put_count16()`, `put_count32()`, `put_double()`, `put_int32()`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_long_string()`, `put_real()`, `put_scount()`, `put_scount16()`, `put_scount32()`, `put_sfloat()`, `put_short()`, `put_string()`, `put_uint32()`, `put_var_string()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

4.2.2.7 `int write_test2 (TEST_DATA * data, IO_BUFFER * iobuf)`

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (ok), <0 (error as for [put_item_end\(\)](#))

References `_struct_IO_ITEM_HEADER::ident`, `put_count()`, `put_count16()`, `put_count32()`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_long_string()`, `put_scount()`, `put_scount16()`, `put_scount32()`, `put_sfloat()`, `put_string()`, `put_var_string()`, `put_vector_of_byte()`, `put_vector_of_double()`, `put_vector_of_int()`, `put_vector_of_int32()`, `put_vector_of_long()`, `put_vector_of_real()`, `put_vector_of_short()`, `put_vector_of_uint32()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

4.2.2.8 `int write_test3 (TEST_DATA * data, IO_BUFFER * iobuf)`

Parameters

<i>data</i>	Pointer to test data structure
<i>iobuf</i>	Pointer to I/O buffer

Returns

0 (ok), <0 (error as for [put_item_end\(\)](#))

References `_struct_IO_ITEM_HEADER::ident`, `put_count()`, `put_count16()`, `put_count32()`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_long_string()`, `put_scount()`, `put_scount16()`, `put_scount32()`, `put_sfloat()`, `put_string()`, `put_var_string()`, `put_vector_of_byte()`, `put_vector_of_double()`, `put_vector_of_int()`, `put_vector_of_int32()`, `put_vector_of_long()`, `put_vector_of_real()`, `put_vector_of_short()`, `put_vector_of_uint32()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

5 Data Structure Documentation

5.1 `_struct_IO_BUFFER` Struct Reference

The `IO_BUFFER` structure contains all data needed the manage the stuff.

```
#include <io_basic.h>
```

Data Fields

- int `aux_count`
May be used for dedicated buffers.
- unsigned char * `buffer`
Pointer to allocated data space.
- long `buflen`
Usable length of data space.
- int `byte_order`
Set if block is not in internal byte order.
- BYTE * `data`
Position for next get.../put...
- int `data_pending`
Set to 1 when header is read but not the data.
- int `extended`
Set to 1 if you want to use the extension field always.
- FILE * `input_file`
For use of stream I/O for input.
- int `input_fileno`
For use of read() function for input.
- int `is_allocated`
Indicates if buffer is allocated by eventio.
- int `item_extension` [MAX_IO_ITEM_LEVEL]
Where the extension field was used.
- long `item_length` [MAX_IO_ITEM_LEVEL]
Length of each level of items.
- int `item_level`
Current level of nesting of items.
- long `item_start_offset` [MAX_IO_ITEM_LEVEL]
Where the item starts in buffer.
- long `max_length`
The maximum length for extending the buffer.
- long `min_length`
The initial and minimum length of the buffer.
- FILE * `output_file`
For use of stream I/O for output.
- int `output_fileno`
For use of write() function for output.
- long `r_remaining`
- int `regular`
1 if a regular file, 0 not known, -1 not regular
- long `sub_item_length` [MAX_IO_ITEM_LEVEL]
Length of its sub-items.

- `int sync_err_count`
Count of synchronization errors.
- `int sync_err_max`
Maximum accepted number of synchronisation errors.
- `int(* user_function)(unsigned char *, long, int)`
For use of special type of I/O.
- `long w_remaining`
Byte available for reading/writing.

5.1.1 Detailed Description

5.1.2 Field Documentation

5.1.2.1 `unsigned char* _struct_IO_BUFFER::buffer`

5.1.2.2 `long _struct_IO_BUFFER::buflen`

5.1.2.3 `int _struct_IO_BUFFER::byte_order`

5.1.2.4 `BYTE* _struct_IO_BUFFER::data`

5.1.2.5 `int _struct_IO_BUFFER::extended`

5.1.2.6 `FILE* _struct_IO_BUFFER::input_file`

5.1.2.7 `int _struct_IO_BUFFER::input_fileno`

5.1.2.8 `int _struct_IO_BUFFER::is_allocated`

It is 1 if buffer is allocated by eventio, 0 if buffer provided by user function (in which case the user should call `allocate_io_buffer` with the appropriate size; then the buffer always allocated in `allocate_io_buffer()` must be freed by the user function, replaced by its external buffer, and finally `is_allocated` set to 0).

5.1.2.9 `int _struct_IO_BUFFER::item_extension[MAX_IO_ITEM_LEVEL]`

5.1.2.10 `int _struct_IO_BUFFER::item_level`

5.1.2.11 `long _struct_IO_BUFFER::item_start_offset[MAX_IO_ITEM_LEVEL]`

5.1.2.12 `FILE* _struct_IO_BUFFER::output_file`

5.1.2.13 `int _struct_IO_BUFFER::output_fileno`

5.1.2.14 `int _struct_IO_BUFFER::sync_err_count`

5.1.2.15 `int _struct_IO_BUFFER::sync_err_max`

5.1.2.16 `int(* _struct_IO_BUFFER::user_function)(unsigned char *, long, int)`

5.1.2.17 `long _struct_IO_BUFFER::w_remaining`

The documentation for this struct was generated from the following file:

- `io_basic.h`

5.2 `_struct_IO_ITEM_HEADER` Struct Reference

An `IO_ITEM_HEADER` is to access header info for an I/O block and as a handle to the I/O buffer.

```
#include <io_basic.h>
```

Data Fields

- int [can_search](#)
Set to 1 if I/O block consist of sub-blocks only.
- long [ident](#)
Identity number.
- size_t [length](#)
Length of data field, for information only.
- int [level](#)
Tells how many levels deep we are nested now.
- unsigned long [type](#)
The type number telling the type of I/O block.
- int [use_extension](#)
Non-zero if the extension header field should be used.
- int [user_flag](#)
One more bit in the header available for user data.
- unsigned [version](#)
The version number used for the block.

5.2.1 Detailed Description

5.2.2 Field Documentation

5.2.2.1 int `_struct_IO_ITEM_HEADER::can_search`

5.2.2.2 long `_struct_IO_ITEM_HEADER::ident`

5.2.2.3 size_t `_struct_IO_ITEM_HEADER::length`

5.2.2.4 int `_struct_IO_ITEM_HEADER::level`

5.2.2.5 unsigned long `_struct_IO_ITEM_HEADER::type`

5.2.2.6 int `_struct_IO_ITEM_HEADER::use_extension`

5.2.2.7 int `_struct_IO_ITEM_HEADER::user_flag`

5.2.2.8 unsigned `_struct_IO_ITEM_HEADER::version`

The documentation for this struct was generated from the following file:

- [io_basic.h](#)

5.3 bunch Struct Reference

Photons collected in bunches of identical direction, position, time, and wavelength.

```
#include <mc_tel.h>
```

Data Fields

- float [ctime](#)
Arrival time (ns)
- float [cx](#)
- float [cy](#)
Direction cosines of photon direction.
- float [lambda](#)
Wavelength in nanometers or 0.
- float [photons](#)
Number of photons in bunch.
- float [x](#)
- float [y](#)
Arrival position relative to telescope (cm)
- float [zem](#)
Height of emission point above sea level (cm)

5.3.1 Detailed Description

The wavelength will normally be unspecified as produced by CORSIKA (lambda=0).

The documentation for this struct was generated from the following file:

- [mc_tel.h](#)

5.4 camera_electronics Struct Reference

Parameters of the electronics of a telescope.

Data Fields

- int [simulated](#)
Is 1 if the signal simulation was done.
- int [telescope](#)
Telescope sequence number.

5.4.1 Field Documentation

5.4.1.1 int camera_electronics::simulated

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.5 compact_bunch Struct Reference

The [compact_bunch](#) struct is equivalent to the bunch struct except that we try to use less memory.

```
#include <mc_tel.h>
```

Data Fields

- short `ctime`
*ctime*10 (0.1ns) after subtracting offset*
- short `cx`
- short `cy`
*cx,cy*30000*
- short `lambda`
(nm) or 0
- short `log_zem`
*log10(zem)*1000*
- short `photons`
*ph*100*
- short `x`
- short `y`
*x,y*10 (mm)*

5.5.1 Detailed Description

And that has a number of limitations: 1) Bunch sizes must be less than 327. 2) photon impact points in a horizontal plane through the centre of each detector sphere must be less than 32.7 m from the detector centre in both x and y coordinates. Thus, $\sec(z) * R < 32.7$ m is required, with 'z' being the zenith angle and 'R' the radius of the detector sphere. When accounting for multiple scattering and Cherenkov emission angles, the actual limit is reached even earlier than that. 3) Only times within 3.27 microseconds from the time, when the primary particle propagated with the speed of light would cross the altitude of the sphere centre, can be treated. For large zenith angle observations this limits horizontal core distances to about 1000 m. For efficiency reasons, no checks are made on these limits.

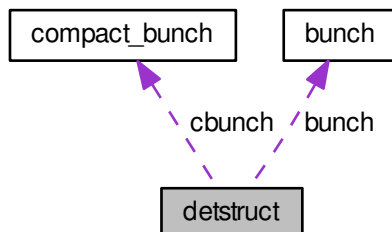
The documentation for this struct was generated from the following file:

- [mc_tel.h](#)

5.6 detstruct Struct Reference

A structure describing a detector and linking its photons bunches to it.

Collaboration diagram for detstruct:



Data Fields

- int **available_bunch**
- int **bits**
- struct [bunch](#) * **bunch**
- struct [compact_bunch](#) * **cbunch**
- int **dclass**
- double **dx**
- double **dy**
- char **ext_fname** [60]
- int **external_bunches**
- int **geo_type**
- int **iarray**
- int **idet**
- int **next_bunch**
- double **photons**
- double **r**
- double **r0**
- double **sampling_area**
- int **sens_type**
- int **shrink_cycle**
- int **shrink_factor**
- double **x**
- double **x0**
- double **y**
- double **y0**
- double **z0**

5.6.1 Detailed Description

The documentation for this struct was generated from the following file:

- [iact.c](#)

5.7 ev_reg_entry Struct Reference

Data Fields

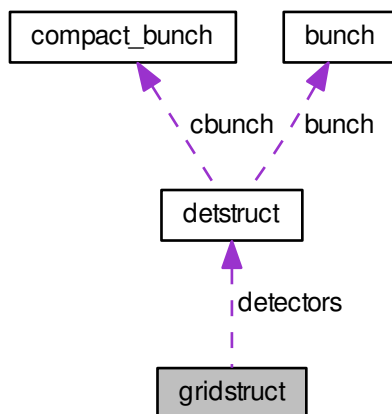
- char * [description](#)
Optional longer description of the data block.
- char * [name](#)
The data block name (short)
- unsigned long [type](#)
The data block type number.

The documentation for this struct was generated from the following file:

- [io_basic.h](#)

5.8 gridstruct Struct Reference

Collaboration diagram for gridstruct:



Data Fields

- struct `detstruct` ** `detectors`
- int `idet`
- int `ndet`

The documentation for this struct was generated from the following file:

- [iact.c](#)

5.9 incpath Struct Reference

An element in a linked list of include paths.

Collaboration diagram for incpath:



Data Fields

- struct `incpath` * `next`
The next element.

- `char * path`

The path name.

5.9.1 Detailed Description

The documentation for this struct was generated from the following file:

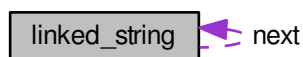
- [fileopen.c](#)

5.10 linked_string Struct Reference

The [linked_string](#) is mainly used to keep CORSIKA input.

```
#include <mc_tel.h>
```

Collaboration diagram for `linked_string`:



Data Fields

- struct [linked_string](#) * **next**
- `char * text`

5.10.1 Detailed Description

The documentation for this struct was generated from the following file:

- [mc_tel.h](#)

5.11 mc_options Struct Reference

Options of the simulation passed through to low-level functions.

Data Fields

- int [always_with_await](#)
Make MCEvent output always with area weights.
- const `char * input_fname`
Input file name.
- long [iobuf_max](#)
Size limit for I/O buffers.

5.11.1 Detailed Description

5.11.2 Field Documentation

5.11.2.1 int mc_options::always_with_aweight

5.11.2.2 const char* mc_options::input_fname

5.11.2.3 long mc_options::iobuf_max

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.12 mc_run Struct Reference

Basic parameters of the CORSIKA run.

Data Fields

- double [area](#)
Area over which offsets may be scattered [m^2].
- int [atmosphere](#)
Model atmosphere number.
- double [bfield_bx](#)
 x' component of B field [mT].
- double [bfield_bz](#)
z component of B field [mT].
- double [bfield_rot](#)
rotation angle mag.
- double [bunchsize](#)
Cherenkov bunch size (nominal).
- int [corsika_version](#)
*CORSIKA version number * 1000.*
- double [e_max](#)
Upper limit of simulated energies [TeV].
- double [e_min](#)
Lower limit of simulated energies [TeV].
- double [height](#)
Height of observation level [m].
- int [high_E_detail](#)
More details on high E interaction model (version etc.)
- int [high_E_model](#)
High energy interaction model flags.
- int [iact_options](#)
Option flags in CORSIKA (VOLUMEDET etc.)
- int [low_E_detail](#)
More details on low E interaction model (version etc.)
- int [low_E_model](#)
Low energy interaction model flags.
- int [num_arrays](#)
Number of arrays simulated.

- int `num_showers`
Number of showers simulated in run.
- double `phi_max`
Upper limit of azimuth angle [degrees].
- double `phi_min`
Lower limit of azimuth angle [degrees].
- double `radius`
Radius within which cores are thrown at random.
- double `radius1`
Distance parameter 1 in CSCAT [m].
- double `radius2`
Distance parameter 2 in CSCAT [m].
- int `run`
Run number.
- double `slope`
Spectral index of power-law spectrum.
- double `start_depth`
Atmospheric depth where primary particle started.
- double `theta_max`
Upper limit of zenith angle [degrees].
- double `theta_min`
Lower limit of zenith angle [degrees].
- double `viewcone_max`
Maximum of VIEWCONE range [degrees].
- double `viewcone_min`
Minimum of VIEWCONE range [degrees].
- double `wlen_max`
Upper limit of Cherenkov wavelength range [nm].
- double `wlen_min`
Lower limit of Cherenkov wavelength range [nm].

5.12.1 Field Documentation

5.12.1.1 double `mc_run::area`

5.12.1.2 int `mc_run::atmosphere`

5.12.1.3 double `mc_run::bfield_bx`

5.12.1.4 double `mc_run::bfield_bz`

5.12.1.5 double `mc_run::bfield_rot`

N -> geogr. N [degrees].

5.12.1.6 double `mc_run::bunchsize`

5.12.1.7 int `mc_run::corsika_version`

5.12.1.8 int `mc_run::high_E_model`

5.12.1.9 int `mc_run::low_E_model`

5.12.1.10 int `mc_run::num_arrays`

5.12.1.11 `int mc_run::num_showers`

5.12.1.12 `double mc_run::radius`

[m]

5.12.1.13 `int mc_run::run`

5.12.1.14 `double mc_run::start_depth`

5.12.1.15 `double mc_run::viewcone_max`

5.12.1.16 `double mc_run::viewcone_min`

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.13 photo_electron Struct Reference

A photo-electron produced by a photon hitting a pixel.

```
#include <mc_tel.h>
```

Data Fields

- `double atime`
The time [ns] when the photon hit the pixel.
- `int lambda`
The wavelength of the photon.
- `int pixel`
The pixel that was hit.

5.13.1 Detailed Description

5.13.2 Field Documentation

5.13.2.1 `double photo_electron::atime`

5.13.2.2 `int photo_electron::lambda`

5.13.2.3 `int photo_electron::pixel`

The documentation for this struct was generated from the following file:

- [mc_tel.h](#)

5.14 pm_camera Struct Reference

Parameters of a telescope camera (pixels, ...)

Data Fields

- `int telescope`
Telescope sequence number.

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.15 shower_extra_parameters Struct Reference

Extra shower parameters of unspecified nature.

```
#include <mc_tel.h>
```

Data Fields

- float * [fparam](#)
Space for extra floats, at least of size nparam.
- long [id](#)
May identify to the user what the parameters should mean.
- int * [iparam](#)
Space for extra integer parameters, at least of size nparam.
- int [is_set](#)
May be reset after writing the parameter block and must thus be set to 1 for each shower for which the extra parameters should get recorded.
- size_t [nfparam](#)
Number of extra floating-point parameters.
- size_t [niparam](#)
Number of extra integer parameters.
- double [weight](#)
To be used if the weight of a shower may change during processing, e.g.

5.15.1 Detailed Description

Useful for things to be used like in the event header but which may only become available while processing a shower. Should be initialized with the `init_shower_extra_parameters(int ni_max, int nf_max)` function.

5.15.2 Field Documentation

5.15.2.1 float* shower_extra_parameters::fparam

5.15.2.2 long shower_extra_parameters::id

5.15.2.3 int* shower_extra_parameters::iparam

5.15.2.4 int shower_extra_parameters::is_set

5.15.2.5 size_t shower_extra_parameters::nfparam

5.15.2.6 size_t shower_extra_parameters::niparam

5.15.2.7 double shower_extra_parameters::weight

when shower processing can be aborted depending on how quickly the electromagnetic component builds up and the remaining showers may have a larger weight to compensate for that. For backwards compatibility this should be set to 1.0 when no additional weight is needed.

The documentation for this struct was generated from the following file:

- [mc_tel.h](#)

5.16 simulated_shower_parameters Struct Reference

Basic parameters of a simulated shower.

Data Fields

- double [altitude](#)
Shower direction altitude above horizon.
- int [array](#)
Array number = shower usage number.
- double [aweight](#)
Area weight to be used with non-uniform.
- double [azimuth](#)
Shower direction azimuth [deg].
- double [clongi](#) [1071]
Vertical profile of Cherenkov light emission.
- double [cmax](#)
*Depth of maximum of Cherenkov light emission [g/cm**2].*
- double [core_dist_3d](#)
Distance of core from reference point.
- double [elongi](#) [1071]
Vertical profile of all electrons+positrons.
- double [emax](#)
Depth of shower maximum from positrons and electrons.
- double [energy](#)
Shower energy [TeV].
- double [h1int](#)
Height a.s.l.
- int [have_longi](#)
Indicates if vertical profiles were found in data.
- double [hmax](#)
Height of shower maximum (from xmax above) [m] a.s.l.
- int [particle](#)
Primary particle type [CORSIKA code].
- int [shower](#)
Shower number.
- double [step_longi](#)
*Step size of vertical profiles [g/cm**2].*
- double [tel_core_dist_3d](#) [MAX_TEL]
Offset of telescopes from shower axis.
- double [x0](#)
Atmospheric depth where particle was started [g/cm^2].
- double [xcore](#)
- double [xlongi](#) [1071]
Vertical profile of all particles.
- double [xmax](#)
*Depth of shower maximum from all particles [g/cm**2].*
- double [ycore](#)
- double [zcore](#)
Shower core position [m].

5.16.1 Field Documentation

5.16.1.1 double simulated_shower_parameters::aweight

core-position sampling [m^2].

5.16.1.2 double simulated_shower_parameters::clongi[1071]

5.16.1.3 double simulated_shower_parameters::elongi[1071]

5.16.1.4 double simulated_shower_parameters::h1int

of first interaction [m].

5.16.1.5 int simulated_shower_parameters::have_longi

If not, the followig numbers should be all zeroes.

5.16.1.6 double simulated_shower_parameters::hmax

5.16.1.7 double simulated_shower_parameters::step_longi

5.16.1.8 double simulated_shower_parameters::x0

5.16.1.9 double simulated_shower_parameters::xlongi[1071]

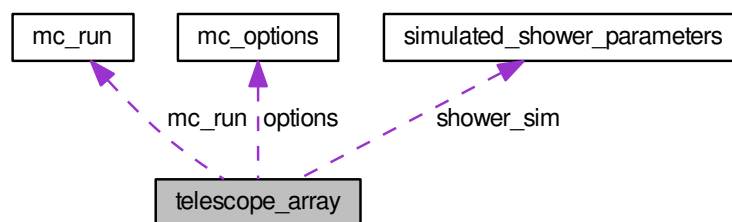
The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.17 telescope_array Struct Reference

Description of telescope position, array offets and shower parameters.

Collaboration diagram for telescope_array:



Data Fields

- double [altitude](#)
Nominal altitude angle of telescope system [deg].
- double [aweight](#) [MAX_ARRAY]
Area weight for non-uniformly distributed core offsets.
- double [azimuth](#)

- *Nominal azimuth angle of telescope system [deg].*
- int **event**
- int **max_tel**
 - *Maximum number of telescopes acceptable (MAX_TEL)*
- struct **mc_run mc_run**
- int **narray**
 - *Number of arrays with random shifts per shower.*
- int **ntel**
 - *Number of telescopes simulated per array.*
- double **obs_height**
 - *Height of observation level [cm].*
- struct **mc_options options**
 - *File names etc.*
- double **refpos** [3]
 - *Reference position with respect to obs.*
- double **rtel** [MAX_TEL]
 - *Radius of spheres enclosing telescopes [cm].*
- int **run**
- struct **simulated_shower_parameters shower_sim**
- double **source_altitude**
 - *Altitude of assumed source.*
- double **source_azimuth**
 - *Azimuth of assumed source.*
- double **toff**
 - *Time offset from first interaction to the moment.*
- int **with_await**
 - *Is 1 if input data came with weights.*
- double **xoff** [MAX_ARRAY]
 - *X offsets of the randomly shifted arrays [cm].*
- double **xel** [MAX_TEL]
 - *X positions of telescopes ([cm] -> north)*
- double **yoff** [MAX_ARRAY]
 - *Y offsets of the randomly shifted arrays [cm].*
- double **yel** [MAX_TEL]
 - *Y positions of telescopes ([cm] -> west)*
- double **zel** [MAX_TEL]
 - *Z positions of telescopes ([cm] -> up)*

5.17.1 Detailed Description

5.17.2 Field Documentation

5.17.2.1 double telescope_array::altitude

5.17.2.2 double telescope_array::await[MAX_ARRAY]

(may be 0 when older data with uniform offsets are read). [cm²]

5.17.2.3 double telescope_array::azimuth

5.17.2.4 struct mc_options telescope_array::options

5.17.2.5 double telescope_array::refpos[3]

level [cm]

5.17.2.6 double telescope_array::source_altitude

5.17.2.7 double telescope_array::source_azimuth

5.17.2.8 double telescope_array::toff

when the extrapolated primary flying with the vacuum speed of light would be at the observation level.

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.18 telescope_optics Struct Reference

Parameters describing the telescope optics.

Data Fields

- int [telescope](#)
Telescope sequence number.

The documentation for this struct was generated from the following file:

- [sim_skeleton.c](#)

5.19 test_struct Struct Reference

Data Fields

- uint8_t **bvar** [2]
- uint16_t **cnt16var** [4]
- uint32_t **cnt32var** [6]
- size_t **cntvar** [8]
- size_t **cntzvar** [6]
- double **dvar** [2]
- double **fvar** [2]
- double **hvar** [2]
- int16_t **i16var** [2]
- int32_t **i32var** [2]
- int8_t **i8var** [2]
- int **ilvar** [2]
- int **isvar** [2]
- long **lvar** [2]
- size_t **nbvar**
- int16_t **scnt16var** [10]
- int32_t **scnt32var** [12]
- ssize_t **scntvar** [14]
- ssize_t **scntzvar** [12]
- char **str16var** [10]
- char **str32var** [10]
- char **strvvar** [10]
- short **svar** [3]
- uint16_t **u16var** [2]
- uint32_t **u32var** [2]

- `uint8_t u8var` [2]

The documentation for this struct was generated from the following file:

- [testio.c](#)

5.20 warn_specific_data Struct Reference

A struct used to store thread-specific data.

Data Fields

- `char *(* aux_function)(void)`
- `int buffered`
- `void(* log_function)(const char *, const char *, int, int)`
- `FILE * logfile`
- `const char * logfname`
The name of the log file.
- `char output_buffer` [2048]
- `void(* output_function)(const char *)`
- `int recursive`
- `char saved_logfname` [256]
- `int warninglevel`
- `int warningmode`

5.20.1 Detailed Description

5.20.2 Field Documentation

5.20.2.1 `const char* warn_specific_data::logfname`

Used only when opening the file.

The documentation for this struct was generated from the following file:

- [warning.c](#)

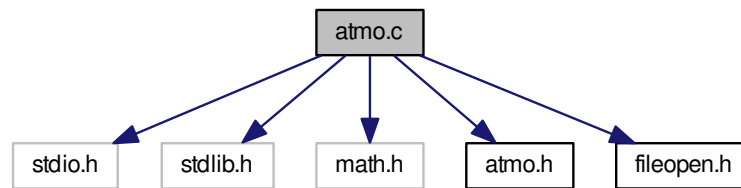
6 File Documentation

6.1 atmo.c File Reference

Use of tabulated atmospheric profiles and atmospheric refraction.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "atmo.h"
#include "fileopen.h"
```

Include dependency graph for atmo.c:



Macros

- `#define FAST_INTERPOLATION 1`
- `#define MAX_FAST_PROFILE 10000`
- `#define MAX_PROFILE 120`
- `#define UNUSED(x) UNUSED_ ## x`

Functions

- static double `atm_exp_fit` (double h1, double h2, double *ap, double *bp, double *cp, double *s0, int *npp)
Fit one atmosphere layer by an exponential density model.
- void `atmfit` (int *nlp, double *hlay, double *a_{atm}, double *b_{atm}, double *c_{atm})
Fit the tabulated density profile for CORSIKA EGS part.
- void `atmset` (int *iatmo, double *obslev)
Set number of atmospheric model profile to be used.
- static double `fn_rho` (double h, int nl, double *hl, double *UNUSED(a), double *b, double *c)
Corresponding to CORSIKA built-in function RHOF; only used to show fit results.
- static double `fn_thick` (double h, int nl, double *hl, double *a, double *b, double *c)
Corresponding to CORSIKA built-in function THICK; only used to show fit results.
- double `heighx` (double *thick)
*Altitude [cm] as a function of atmospheric thickness [g/cm**2].*
- static void `init_atmosphere` ()
Initialize atmospheric profiles.
- static void `init_corsika_atmosphere` ()
Take the atmospheric profile from CORSIKA built-in functions.
- static void `init_fast_interpolation` ()
An alternate interpolation method (which requires that the table is sufficiently fine-grained and equidistant) has to be initialized first.
- static void `init_refraction_tables` ()
Initialize tables needed for atmospheric refraction.
- static void `interp` (double x, double *v, int n, int *ipl, double *rpl)
Linear interpolation with binary search algorithm.
- void `raybnd` (double *zem, `cors_dbl_t` *u, `cors_dbl_t` *v, double *w, `cors_dbl_t` *dx, `cors_dbl_t` *dy, `cors_dbl_t` *dt)
Calculate the bending of light due to atmospheric refraction.
- double `refidx` (double *height)
Index of refraction as a function of altitude [cm].

- double `rhofx_` (double *height)
Density of the atmosphere as a function of altitude.
- double `rpol` (double *x, double *y, int n, double xp)
Linear interpolation with binary search algorithm.
- static double `sum_log_dev_sq` (double a, double b, double c, int np, double *h, double *t, double *rho)
- static double `sum_log_dev_sq` (double a, double b, double c, int np, double *h, double *t, double *UNUSE↵
D(rho))
Measure of deviation of model layers from tables.
- double `thickx_` (double *height)
*Atmospheric thickness [g/cm**2] as a function of altitude.*

Variables

- int `atmosphere`
The atmospheric profile number, 0 for built-in.
- static double `bottom_of_atmosphere` = 0.
- static double `etadsn`
About the same as in CORSIKA Cherenkov function.
- static double `fast_h_fac`
- static double `fast_p_alt` [MAX_FAST_PROFILE]
- static double `fast_p_log_n1` [MAX_FAST_PROFILE]
- static double `fast_p_log_rho` [MAX_FAST_PROFILE]
- static double `fast_p_log_thick` [MAX_FAST_PROFILE]
- static int `num_prof`
- static double `obs_level_refidx`
- static double `obs_level_thick`
- static double `observation_level`
Altitude [cm] of observation level.
- static double `p_alt` [MAX_PROFILE]
- static double `p_bend_ray_hori_a` [MAX_PROFILE]
- static double `p_bend_ray_time0` [MAX_PROFILE]
- static double `p_bend_ray_time_a` [MAX_PROFILE]
- static double `p_log_alt` [MAX_PROFILE]
- static double `p_log_n1` [MAX_PROFILE]
- static double `p_log_rho` [MAX_PROFILE]
- static double `p_log_thick` [MAX_PROFILE]
- static double `p_rho` [MAX_PROFILE]
- static double `top_of_atmosphere` = 112.83e5

6.1.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2016/07/26 12:52:42 \$

Version

CVS \$Revision: 1.18 \$

This file provides code for use of external atmospheric models (in the form of text-format tables) with the CORSIKA program. Six atmospheric models as implemented in the MODTRAN program and as tabulated in MODTRAN documentation (F.X. Kneizys et al. 1996, 'The MODTRAN 2/3 Report and LOWTRAN 7 Model', Phillips Laboratory, Hanscom AFB, MA 01731-3010, U.S.A.) are provided as separate files (atmprof1.dat ... atmprof6.dat). User-provided atmospheric models should be given model numbers above 6.

Note that for the Cherenkov part and the hadronic (and muon) part of CORSIKA the table values are directly interpolated but the electron/positron/gamma part (derived from EGS) uses special layers (at present 4 with exponential density decrease and the most upper layer with constant density). Parameters of these layers are fitted to tabulated values but not every possible atmospheric model fits very well with an exponential profile. You are advised to check that the fit matches tabulated values to sufficient precision in the altitude ranges of interest to you. Try to adjust layer boundary altitudes in case of problems. The propagation of light without refraction (as implemented in CORSIKA, unless using the CURVED option) and with refraction (as implemented by this software) assumes a plane-parallel atmosphere.

6.1.2 Function Documentation

6.1.2.1 void atmfit_ (int * nlp, double * hlay, double * aatm, double * batm, double * catm)

Fitting of the tabulated atmospheric density profile by piecewise exponential parts as used in CORSIKA. The fits are constrained by fixing the atmospheric thicknesses at the boundaries to the values obtained from the table. Note that not every atmospheric profile can be fitted well by the CORSIKA piecewise models (4*exponential + 1*constant density). In particular, the tropical model is known to be a problem. Setting the boundary heights manually might help. The user is advised to check at least once that the fitted layers represent the tabulated atmosphere sufficiently well, at least at the altitudes most critical for the observations (usually at observation level and near shower maximum but depending on the user's emphasis, this may vary).

Fit all layers (except the uppermost) by exponentials and (if *nlp > 0) try to improve fits by adjusting layer boundaries. The uppermost layer has constant density up to the 'edge' of the atmosphere.

This function may be called from CORSIKA.

Parameters (all pointers since function is called from Fortran):

Parameters

<i>nlp</i>	Number of layers (or negative of that if boundaries set manually)
<i>hlay</i>	Vector of layer (lower) boundaries.
<i>aatm,batm,catm</i>	Parameters as used in CORSIKA.

References atm_exp_fit(), fn_rho(), fn_thick(), rho_fx_(), and thickx_().

6.1.2.2 void atmset_ (int * iatmo, double * obslev)

The atmospheric model is initialized first before the interpolating functions can be used. For efficiency reasons, the functions rho_fx_(), thickx_(), ... don't check if the initialisation was done.

This function is called if the 'ATMOSPHERE' keyword is present in the CORSIKA input file.

The function may be called from CORSIKA to initialize the atmospheric model via 'CALL ATMSET(IATMO,OBSLEV)' or such.

Parameters

<i>iatmo</i>	(pointer to) atmospheric profile number; negative for CORSIKA built-in profiles.
--------------	--

<i>obslev</i>	(pointer to) altitude of observation level [cm]
---------------	---

Returns

(none)

References `init_atmosphere()`, `refidx_()`, and `thickx_()`.

6.1.2.3 `static double fn_rhof (double h, int nl, double * hl, double * UNUSEDa, double * b, double * c)` [static]

6.1.2.4 `static double fn_thick (double h, int nl, double * hl, double * a, double * b, double * c)` [static]

6.1.2.5 `double heighx_ (double * thick)`

This function can be called from Fortran code as HEIGHX(THICK).

Parameters

<i>thick</i>	(pointer to) atmospheric thickness [g/cm**2]
--------------	--

Returns

altitude [cm]

References `rpol()`.

6.1.2.6 `static void init_atmosphere ()` [static]

Internal function for initialising both external and CORSIKA built-in atmospheric profiles. If any CORSIKA built-in profile should be used, it simply calls `init_corsika_atmosphere()`.

Otherwise, atmospheric models are read in from text-format tables. The supplied models 1-6 are based on output of the MODTRAN program. For the interpolation of relevant parameters (density, thickness, index of refraction, ...) all parameters are transformed such that linear interpolation can be easily used.

References `fileopen()`, `init_corsika_atmosphere()`, `init_fast_interpolation()`, and `init_refraction_tables()`.

6.1.2.7 `static void init_corsika_atmosphere ()` [static]

For use of the refraction bending corrections together with the CORSIKA built-in atmospheres, the atmosphere tables are constructed from the CORSIKA RHOF and THICK functions. Note that the refraction index in this case is without taking the effect of water vapour into account.

References `heigh_()`, `init_fast_interpolation()`, `init_refraction_tables()`, `rhof_()`, and `thick_()`.

6.1.2.8 `static void init_refraction_tables ()` [static]

Initialize the correction tables used for the refraction bending of the light paths. It is called once after the atmospheric profile has been defined.

References `observation_level`, `refidx_()`, and `thickx_()`.

6.1.2.9 `static void interp (double x, double * v, int n, int * ipl, double * rpl)` [static]

Linear interpolation between data point in sorted (i.e. monotonic ascending or descending) order. This function determines between which two data points the requested coordinate is and where between them. If the given coordinate is outside the covered range, the value for the corresponding edge is returned.

A binary search algorithm is used for fast interpolation.

Parameters

<i>x</i>	Input: the requested coordinate
<i>v</i>	Input: tabulated coordinates at data points
<i>n</i>	Input: number of data points
<i>ipl</i>	Output: the number of the data point following the requested coordinate in the given sorting (1 ≤ <i>ipl</i> ≤ <i>n</i> -1)
<i>rpl</i>	Output: the fraction (x-v[<i>ipl</i> -1])/(v[<i>ipl</i>]-v[<i>ipl</i> -1]) with 0 ≤ <i>rpl</i> ≤ 1

6.1.2.10 void raybnd_ (double * zem, cors_dbl_t * u, cors_dbl_t * v, double * w, cors_dbl_t * dx, cors_dbl_t * dy, cors_dbl_t * dt)

Path of light through the atmosphere including the bending by refraction. This function assumes a plane-parallel atmosphere. Coefficients for corrections from straight-line propagation to refraction-bent path are numerically evaluated when the atmospheric model is defined. Note that while the former mix of double/float data types may appear odd, it was determined by the variables present in older CORSIKA to save conversions. With CORSIKA 6.0 all parameters are of double type.

This function may be called from FORTRAN as CALL RAYBND(ZEM,U,V,W,DX,DY,DT)

Parameters

<i>zem</i>	Altitude of emission above sea level [cm]
<i>u</i>	Initial/Final direction cosine along X axis (updated)
<i>v</i>	Initial/Final direction cosine along Y axis (updated)
<i>w</i>	Initial/Final direction cosine along Z axis (updated)
<i>dx</i>	Position in CORSIKA detection plane [cm] (updated)
<i>dy</i>	Position in CORSIKA detection plane [cm] (updated)
<i>dt</i>	Time of photon [ns]. Input: emission time. Output: time of arrival in CORSIKA detection plane.

References observation_level, refidx_(), rhofx_(), rpol(), and thickx_().

6.1.2.11 double refidx_ (double * height)

This function can be called from Fortran code as REFIDX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

index of refraction

References rpol().

6.1.2.12 double rhofx_ (double * height)

This function can be called from Fortran code as RHOFX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

density [g/cm**3]

References rpol().

6.1.2.13 `double rpol (double * x, double * y, int n, double xp)`

Linear interpolation between data point in sorted (i.e. monotonic ascending or descending) order. The resulting interpolated value is returned as a return value.

This function calls [interp\(\)](#) to find out where to interpolate.

Parameters

<i>x</i>	Input: Coordinates for data table
<i>y</i>	Input: Corresponding values for data table
<i>n</i>	Input: Number of data points
<i>xp</i>	Input: Coordinate of requested value

Returns

Interpolated value

References [interp\(\)](#).

6.1.2.14 `double thickx_ (double * height)`

This function can be called from Fortran code as THICKX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

thickness [g/cm**2]

References [rpol\(\)](#).

6.1.3 Variable Documentation

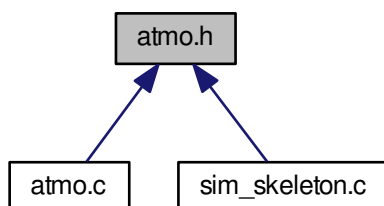
6.1.3.1 `double etadsn [static]`

(but doesn't need to be the same).

6.2 `atmo.h` File Reference

Use of tabulated atmospheric profiles and atmospheric refraction.

This graph shows which files directly or indirectly include this file:



Macros

- `#define CORSIKA_VERSION 6900`

Typedefs

- `typedef double cors_dbl_t`

Functions

- void `atmfit_` (int *nlp, double *hlay, double *aatm, double *batm, double *catm)
Fit the tabulated density profile for CORSIKA EGS part.
- void `atmset_` (int *iatmo, double *obslev)
Set number of atmospheric model profile to be used.
- double `heigh_` (double *thick)
The CORSIKA built-in function for the height as a function of overburden.
- double `heighx_` (double *thick)
*Altitude [cm] as a function of atmospheric thickness [g/cm**2].*
- void `raybnd_` (double *zem, `cors_dbl_t` *u, `cors_dbl_t` *v, double *w, `cors_dbl_t` *dx, `cors_dbl_t` *dy, `cors_dbl_t` *dt)
Calculate the bending of light due to atmospheric refraction.
- double `refidx_` (double *height)
Index of refraction as a function of altitude [cm].
- double `rhof_` (double *height)
The CORSIKA built-in density lookup function.
- double `rhofx_` (double *height)
Density of the atmosphere as a function of altitude.
- double `rpol` (double *x, double *y, int n, double xp)
Linear interpolation with binary search algorithm.
- double `thick_` (double *height)
The CORSIKA built-in function for vertical atmospheric thickness (overburden).
- double `thickx_` (double *height)
*Atmospheric thickness [g/cm**2] as a function of altitude.*

6.2.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2016/07/07 14:05:58 \$

Version

CVS \$Revision: 1.6 \$

6.2.2 Function Documentation

6.2.2.1 void atmfit_ (int * *nlp*, double * *hlay*, double * *aatm*, double * *batm*, double * *catm*)

Fitting of the tabulated atmospheric density profile by piecewise exponential parts as used in CORSIKA. The fits are constrained by fixing the atmospheric thicknesses at the boundaries to the values obtained from the table. Note that not every atmospheric profile can be fitted well by the CORSIKA piecewise models (4*exponential + 1*constant density). In particular, the tropical model is known to be a problem. Setting the boundary heights manually might help. The user is advised to check at least once that the fitted layers represent the tabulated atmosphere sufficiently well, at least at the altitudes most critical for the observations (usually at observation level and near shower maximum but depending on the user's emphasis, this may vary).

Fit all layers (except the uppermost) by exponentials and (if *nlp > 0) try to improve fits by adjusting layer boundaries. The uppermost layer has constant density up to the 'edge' of the atmosphere.

This function may be called from CORSIKA.

Parameters (all pointers since function is called from Fortran):

Parameters

<i>nlp</i>	Number of layers (or negative of that if boundaries set manually)
<i>hlay</i>	Vector of layer (lower) boundaries.
<i>aatm, batm, catm</i>	Parameters as used in CORSIKA.

References atm_exp_fit(), fn_rhof(), fn_thick(), rhofx_(), and thickx_().

6.2.2.2 void atmset_ (int * *iatmo*, double * *obslev*)

The atmospheric model is initialized first before the interpolating functions can be used. For efficiency reasons, the functions [rhofx_\(\)](#), [thickx_\(\)](#), ... don't check if the initialisation was done.

This function is called if the 'ATMOSPHERE' keyword is present in the CORSIKA input file.

The function may be called from CORSIKA to initialize the atmospheric model via 'CALL ATMSET(IATMO,OBSLEV)' or such.

Parameters

<i>iatmo</i>	(pointer to) atmospheric profile number; negative for CORSIKA built-in profiles.
<i>obslev</i>	(pointer to) altitude of observation level [cm]

Returns

(none)

6.2.2.3 double heigh_ (double * *thick*)

6.2.2.4 double heighx_ (double * *thick*)

This function can be called from Fortran code as HEIGHX(THICK).

Parameters

<i>thick</i>	(pointer to) atmospheric thickness [g/cm**2]
--------------	--

Returns

altitude [cm]

References rpol().

6.2.2.5 void raybnd_ (double * zem, cors_dbl_t * u, cors_dbl_t * v, double * w, cors_dbl_t * dx, cors_dbl_t * dy, cors_dbl_t * dt)

Path of light through the atmosphere including the bending by refraction. This function assumes a plane-parallel atmosphere. Coefficients for corrections from straight-line propagation to refraction-bent path are numerically evaluated when the atmospheric model is defined. Note that while the former mix of double/float data types may appear odd, it was determined by the variables present in older CORSIKA to save conversions. With CORSIKA 6.0 all parameters are of double type.

This function may be called from FORTRAN as CALL RAYBND(ZEM,U,V,W,DX,DY,DT)

Parameters

<i>zem</i>	Altitude of emission above sea level [cm]
<i>u</i>	Initial/Final direction cosine along X axis (updated)
<i>v</i>	Initial/Final direction cosine along Y axis (updated)
<i>w</i>	Initial/Final direction cosine along Z axis (updated)
<i>dx</i>	Position in CORSIKA detection plane [cm] (updated)
<i>dy</i>	Position in CORSIKA detection plane [cm] (updated)
<i>dt</i>	Time of photon [ns]. Input: emission time. Output: time of arrival in CORSIKA detection plane.

References observation_level, refidx_(), rhofx_(), rpol(), and thickx_().

6.2.2.6 double refidx_ (double * height)

This function can be called from Fortran code as REFIDX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

index of refraction

6.2.2.7 double rhof_ (double * height)

6.2.2.8 double rhofx_ (double * height)

This function can be called from Fortran code as RHOFX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

density [g/cm**3]

References rpol().

6.2.2.9 double rpol (double * x, double * y, int n, double xp)

Linear interpolation between data point in sorted (i.e. monotonic ascending or descending) order. The resulting interpolated value is returned as a return value.

This function calls [interp\(\)](#) to find out where to interpolate.

Parameters

x	Input: Coordinates for data table
y	Input: Corresponding values for data table
n	Input: Number of data points
xp	Input: Coordinate of requested value

Returns

Interpolated value

References `interp()`.

6.2.2.10 `double thick_ (double * height)`

6.2.2.11 `double thickx_ (double * height)`

This function can be called from Fortran code as THICKX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

thickness [g/cm**2]

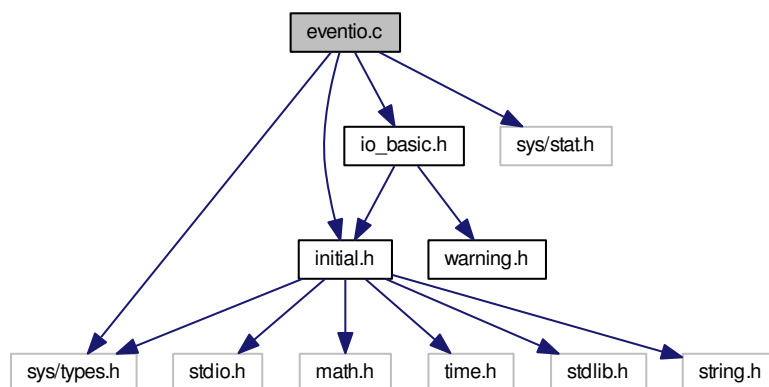
References `rpol()`.

6.3 eventio.c File Reference

Basic functions for eventio data format.

```
#include "initial.h"
#include "io_basic.h"
#include <sys/types.h>
#include <sys/stat.h>
```

Include dependency graph for eventio.c:



Macros

- `#define IO_BUFFER_MINIMUM_SIZE 32L`
- `#define NO_FOREIGN_PROTOTYPES 1`
- `#define READ_BYTES(fd, buf, nb)`

Functions

- `IO_BUFFER * allocate_io_buffer (size_t buflen)`
Dynamic allocation of an I/O buffer.
- `int append_io_block_as_item (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header, BYTE *buffer, long length)`
Append data from one I/O block into another one.
- `int copy_item_to_io_block (IO_BUFFER *iobuf2, IO_BUFFER *iobuf, const IO_ITEM_HEADER *item_header)`
Copy a sub-item to another I/O buffer as top-level item.
- `double dbl_from_sfloat (const uint16_t *snum)`
Convert from the internal representation of an OpenGL 16-bit floating point number back to normal floating point representation.
- `void dbl_to_sfloat (double dnum, uint16_t *snum)`
Convert a double to the internal representation of a 16 bit floating point number as specified in the OpenGL 3.1 standard, also called a half-float.
- `const char * eventio_registered_description (unsigned long type)`
Extract the optional description for a given type number, if available.
- `const char * eventio_registered_typename (unsigned long type)`
Extract the name for a given type number, if available.
- `int extend_io_buffer (IO_BUFFER *iobuf, unsigned next_byte, long increment)`
Extend the dynamically allocated I/O buffer.
- `struct ev_reg_entry * find_ev_reg (unsigned long t)`
This optionally available function is implemented externally.
- `int find_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header)`
Find the beginning of the next I/O data block in the input.
- `void fltp_to_sfloat (const float *fnum, uint16_t *snum)`
Convert a float to the internal representation of a 16 bit floating point number as specified in the OpenGL 3.1 standard, also called a half-float.
- `void free_io_buffer (IO_BUFFER *iobuf)`
Free an I/O buffer that has been allocated at run-time.
- `uintmax_t get_count (IO_BUFFER *iobuf)`
Get an unsigned integer of unspecified length from an I/O buffer.
- `uint16_t get_count16 (IO_BUFFER *iobuf)`
Get an unsigned 16 bit integer of unspecified length from an I/O buffer.
- `uint32_t get_count32 (IO_BUFFER *iobuf)`
Get an unsigned 32 bit integer of unspecified length from an I/O buffer.
- `double get_double (IO_BUFFER *iobuf)`
Get a double from the I/O buffer.
- `int32_t get_int32 (IO_BUFFER *iobuf)`
Read a four byte integer from an I/O buffer.
- `int get_item_begin (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header)`
Begin reading an item.
- `int get_item_end (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header)`
End reading an item.
- `long get_long (IO_BUFFER *iobuf)`

- Get 4-byte integer from I/O buffer and return as a long int.*

 - int [get_long_string](#) (char *s, int nmax, [IO_BUFFER](#) *iobuf)

Get a long string of ASCII characters from an I/O buffer.
- double [get_real](#) ([IO_BUFFER](#) *iobuf)

Get a floating point number (as written by [put_real](#)) from the I/O buffer.
- intmax_t [get_scount](#) ([IO_BUFFER](#) *iobuf)

Get a signed integer of unspecified length from an I/O buffer.
- int16_t [get_scount16](#) ([IO_BUFFER](#) *iobuf)

Shortened version of [get_scount](#) for up to 16 bits of data.
- int32_t [get_scount32](#) ([IO_BUFFER](#) *iobuf)

Shortened version of [get_scount](#) for up to 32 bits of data.
- double [get_sfloat](#) ([IO_BUFFER](#) *iobuf)

Get a 16-bit float from an I/O buffer and expand it to a double.
- int [get_short](#) ([IO_BUFFER](#) *iobuf)

Get a two-byte integer from an I/O buffer.
- int [get_string](#) (char *s, int nmax, [IO_BUFFER](#) *iobuf)

Get a string of ASCII characters from an I/O buffer.
- uint16_t [get_uint16](#) ([IO_BUFFER](#) *iobuf)

Get one unsigned short from an I/O buffer.
- uint32_t [get_uint32](#) ([IO_BUFFER](#) *iobuf)

Get a four-byte unsigned integer from an I/O buffer.
- int [get_var_string](#) (char *s, int nmax, [IO_BUFFER](#) *iobuf)

Get a string of ASCII characters from an I/O buffer.
- void [get_vector_of_byte](#) (BYTE *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of bytes from an I/O buffer.
- void [get_vector_of_double](#) (double *dvec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of floating point numbers as 'doubles' from an I/O buffer.
- void [get_vector_of_float](#) (float *fvec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of floating point numbers as 'floats' from an I/O buffer.
- void [get_vector_of_int](#) (int *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of (small) integers from I/O buffer.
- void [get_vector_of_int32](#) (int32_t *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of 32 bit integers from I/O buffer.
- void [get_vector_of_int_scount](#) (int *vec, int num, [IO_BUFFER](#) *iobuf)

Get an array of ints as [scount32](#) data from an I/O buffer.
- void [get_vector_of_long](#) (long *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of 4-byte integers as long int from I/O buffer.
- void [get_vector_of_real](#) (double *dvec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of floating point numbers as 'doubles' from an I/O buffer.
- void [get_vector_of_short](#) (short *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of short integers from I/O buffer.
- void [get_vector_of_uint16](#) (uint16_t *uval, int num, [IO_BUFFER](#) *iobuf)

Get a vector of unsigned shorts from an I/O buffer.
- void [get_vector_of_uint16_scount_differential](#) (uint16_t *vec, int num, [IO_BUFFER](#) *iobuf)

Get an array of [uint16_t](#) as differential [scount](#) data from an I/O buffer.
- void [get_vector_of_uint32](#) (uint32_t *vec, int num, [IO_BUFFER](#) *iobuf)

Get a vector of 32 bit integers from I/O buffer.
- void [get_vector_of_uint32_scount_differential](#) (uint32_t *vec, int num, [IO_BUFFER](#) *iobuf)

Get an array of [uint32_t](#) as differential [scount](#) data from an I/O buffer.
- int [list_io_blocks](#) ([IO_BUFFER](#) *iobuf, int verbosity)

Show the top-level item of an I/O block on standard output.

- int `list_sub_items` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header, int maxlevel, int verbosity)
Display the contents of sub-items on standard output.
- long `next_subitem_ident` (`IO_BUFFER` *iobuf)
Reads the header of a sub-item and return the identifier of it.
- long `next_subitem_length` (`IO_BUFFER` *iobuf)
Reads the header of a sub-item and return the length of it.
- int `next_subitem_type` (`IO_BUFFER` *iobuf)
Reads the header of a sub-item and return the type of it.
- void `put_count` (uintmax_t n, `IO_BUFFER` *iobuf)
Put an unsigned integer of unspecified length to an I/O buffer.
- void `put_count16` (uint16_t n, `IO_BUFFER` *iobuf)
Shortened version of put_count for up to 16 bits of data.
- void `put_count32` (uint32_t n, `IO_BUFFER` *iobuf)
Shortened version of put_count for up to 32 bits of data.
- void `put_double` (double dnum, `IO_BUFFER` *iobuf)
Put a 'double' as such into an I/O buffer.
- void `put_int32` (int32_t num, `IO_BUFFER` *iobuf)
Write a four-byte integer to an I/O buffer.
- int `put_item_begin` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Begin putting another (sub-) item into the output buffer.
- int `put_item_begin_with_flags` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header, int user_flag, int extended)
Begin putting another (sub-) item into the output buffer.
- int `put_item_end` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
End of putting an item into the output buffer.
- void `put_long` (long num, `IO_BUFFER` *iobuf)
Put a four-byte integer taken from a 'long' into an I/O buffer.
- int `put_long_string` (const char *s, `IO_BUFFER` *iobuf)
Put a long string of ASCII characters into an I/O buffer.
- void `put_real` (double dnum, `IO_BUFFER` *iobuf)
Put a 4-byte floating point number into an I/O buffer.
- void `put_scount` (intmax_t n, `IO_BUFFER` *iobuf)
Put a signed integer of unspecified length to an I/O buffer.
- void `put_scount16` (int16_t n, `IO_BUFFER` *iobuf)
Shorter version of put_scount for up to 16 bytes of data.
- void `put_scount32` (int32_t n, `IO_BUFFER` *iobuf)
Shorter version of put_scount for up to 32 bytes of data.
- void `put_sfloat` (double dnum, `IO_BUFFER` *iobuf)
Put a 16-bit float to an I/O buffer.
- void `put_short` (int num, `IO_BUFFER` *iobuf)
Put a two-byte integer on an I/O buffer.
- int `put_string` (const char *s, `IO_BUFFER` *iobuf)
Put a string of ASCII characters into an I/O buffer.
- void `put_uint32` (uint32_t num, `IO_BUFFER` *iobuf)
Put a four-byte integer into an I/O buffer.
- int `put_var_string` (const char *s, `IO_BUFFER` *iobuf)
Put a string of ASCII characters into an I/O buffer.
- void `put_vector_of_byte` (const BYTE *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of bytes into an I/O buffer.
- void `put_vector_of_double` (const double *dvec, int num, `IO_BUFFER` *iobuf)
Put a vector of doubles into an I/O buffer.

- void `put_vector_of_float` (const float *fvec, int num, `IO_BUFFER` *iobuf)
Put a vector of floats as IEEE 'float' numbers into an I/O buffer.
- void `put_vector_of_int` (const int *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of integers (range -32768 to 32767) into I/O buffer.
- void `put_vector_of_int32` (const int32_t *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 32 bit integers into I/O buffer.
- void `put_vector_of_int_scount` (const int *vec, int num, `IO_BUFFER` *iobuf)
Put an array of ints as scount32 data into an I/O buffer.
- void `put_vector_of_long` (const long *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of long int as 4-byte integers into an I/O buffer.
- void `put_vector_of_real` (const double *dvec, int num, `IO_BUFFER` *iobuf)
Put a vector of doubles as IEEE 'float' numbers into an I/O buffer.
- void `put_vector_of_short` (const short *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 2-byte integers on an I/O buffer.
- void `put_vector_of_uint16` (const uint16_t *uval, int num, `IO_BUFFER` *iobuf)
Put a vector of unsigned shorts into an I/O buffer.
- void `put_vector_of_uint16_scount_differential` (uint16_t *vec, int num, `IO_BUFFER` *iobuf)
Put an array of uint16_t as differential scount data into an I/O buffer.
- void `put_vector_of_uint32` (const uint32_t *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 32 bit integers into I/O buffer.
- void `put_vector_of_uint32_scount_differential` (uint32_t *vec, int num, `IO_BUFFER` *iobuf)
Put an array of uint16_t as differential scount data into an I/O buffer.
- int `read_io_block` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Read the data of an I/O block from the input.
- int `remove_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Remove an item from an I/O buffer.
- int `reset_io_block` (`IO_BUFFER` *iobuf)
Reset an I/O block to its empty status.
- int `rewind_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Go back to the beginning of an item.
- int `search_sub_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header, `IO_ITEM_HEADER` *sub_↵
item_header)
Search for an item of a specified type.
- void `set_eventio_registry_hook` (struct `ev_reg_entry` *(*fptr)(unsigned long t))
A single function for setting a registry search function is the interface between the eventio core code and any code implementing the registry.
- int `skip_io_block` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Skip the data of an I/O block from the input.
- int `skip_subitem` (`IO_BUFFER` *iobuf)
When the next sub-item is of no interest, it can be skipped.
- int `unget_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Go back to the beginning of an item being read.
- int `unput_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Undo writing at the present level.
- int `write_io_block` (`IO_BUFFER` *iobuf)
Write an I/O block to the block's output.

Variables

- static EVREGSEARCH `find_ev_reg_ptr`
We just keep a pointer to such a function in the core eventio code.
- static const char * `none` = ""
----- `find_ev_reg` -----

6.3.1 Detailed Description

Author

Konrad Bernloehr

Date

1991 to 2010

CVS \$Date: 2016/11/10 13:04:29 \$

Version

CVS \$Revision: 1.49 \$

===== General comments to eventio.c =====

'eventio.c' provides an interface for an (almost) machine-independent way to write and read event data, configuration data and Monte Carlo data. Byte ordering of the data is unimportant and data written in both byte orders are correctly read on any supported architecture. Usually the data is written to/read from a file (or separate files for different data types) to be opened before calling any eventio function. Other ways to 'save' data (e.g. into memory or via dedicated networking procedures) can easily be incorporated by assigning an input and/or output function to an I/O buffer instead of a file handle or pointer. The data structure is designed to allow reading of a mixture of different types of items from a single file. For this purpose, 'items' (see below) should not be interspersed with low-level material and, therefore, low-level functions should not be called from anywhere outside eventio.c.

An 'item' has the following structure:

Component	Type	Content	Description
-----	----	-----	-----
sync-tag	long	0xD41F8A37	Signature of start of any item (only for top item, not for sub-items).
type/version	long	...	Item type (bits 0 to 15), a single bit (16) of user data, extension flag (bit 17), reserved bits (18 to 19), and version of this item type (bits 20 to 31).
ident	long	...	Unique identification number of the item or -1.
length	long	...	No. of bytes following for this item (bits 0 to 29) and a flag indicating whether the item consists entirely of sub-items with known length (bit 30). Bit 31 must be 0. The bytes needed to pad the item to the next 4-byte boundary are included in the length.
[extension	long	...	Only present if the extension flag in the type/version field is set. At this time, bits 0 to 11 will extend the length field (as bits 30 to 41). Note that 32-bit machines will not be able to take advantage of more than the one or two of those bits. Bits 12 to 31 are reserved and must be 0.]
data	Item data (may consist of elementary data and of sub-items)

Field 'sync-tag':

The sync-tag is used to check that input is still synchronized. In the case of a synchronisation failure, all data should be skipped up to the next occurrence of that byte combination or its reverse. The byte ordering of the sync-tag defines also the byte ordering of all data in the item. Only byte orders 0-1-2-3 and 3-2-1-0 are

accepted at present.

Field 'type/version':

This field consists of a type number in bits 0 to 15 (values 0 to 65535), a single bit of user data (user flag), reserved bits 17 to 19 (must be 0), and an item version number in bits 20 to 31 (values 0 to 4095). Whenever the format of an item changes in a way which is incompatible with older reading software the version number has to be increased. This ways the reading software can be adapted to multiple format versions.

Field 'ident':

Items of the same type can be distinguished if an identification number is supplied. Negative values are interpreted as 'no ident supplied'.

Field 'length':

Each item and sub-item must have the number of bytes in its data area, including padding bytes, in bits 0 to 30 of this field. If an item consists entirely of sub-items and no atomic data, it can be searched for a specific type of sub-item without having to 'decode' (read from the buffer) any of the sub-items. Such an item is kind of a directory of sub-items and is marked by setting bit 30 of the length field on. The longest possible item length is thus $(2^{30} - 1)$. Note that the length field specifies the length of the rest of the item but not the sync-tag, type/version number, and length fields. All (sub-) items are padded to make the total length a multiple of 4 bytes and the no. of padded bytes must be included in 'length'.

Field 'extension':

This field is not present by default but requires the extension flag (as indicated in bit 17 of the 'type/version' field). Writing of data with the 'extension' flag can be forced by setting the 'extended' element of the I/O buffer to 1 in advance. It can also be activated on a per-item basis but complications would arise once a sub-item goes beyond the 1 GB limit and any of its ancestors does not have the extension activated. Only bits 0-11 are used at this time (as bits 30-41 of the item length). On 32-bit systems only one or two of these bits would be usable (depending if lengths are counted with signed or unsigned integers). All other bits are reserved and must be set to 0 for now. Data written with the extension field will not be readable with pre-mid-2007 versions of eventio. Apart from that and within the limitations of the host architecture, reading of data with or without extension field is completely transparent to the application.

Data:

Data of an item may be either sub-items or atomic data. An item may even consist of a mixture of both but in that case the sub-items are not accessible via 'directory' functions and can be processed only when the item data is 'decoded' by its corresponding 'read_...' function.

The beginning of the data field is aligned on a 4-byte boundary to allow efficient access to data if the byte order needs not to be changed and if the data itself obeys the required alignment.

The 'atomic' data types are kept as close as possible to internal data types. This data is only byte-aligned unless all atomic data of an item obeys a 2-byte or 4-byte alignment. Note that the ANSI C internal type `int32_t` typically corresponds to both 'int' and 'long' on 32-bit machines but to 'int' only on 64-bit machines and to 'long' only on 16-bit systems. Use the `int32_t`/`uint32_t` etc. types where the same length of internal variables is required. 64-bit integer data are also implemented in eventio but not available on all systems.

Type	Int. type	Size (bytes)	Comments
byte	<code>[u]int8_t</code>	1	Character or very short integer.
count	<code>uintmax_t</code>	1 to 9	Unsigned. Larger numbers need more bytes.

scount	intmax_t	1 to 9	Signed. Larger numbers need more bytes.
short	[u]int16_t	2	Short integer (signed or unsigned).
long	[u]int32_t	4	Long integer (signed or unsigned).
int64	[u]int64_t	8	Caution: not available on all systems.
string	-	2+length	Preceded by 2-byte length of string.
long str.	-	4+length	Preceded by 4-byte length of string.
var str.	-	(1-5)+length	Preceded by length of string as 'count'.
real	float	4	32-bit IEEE floating point number with the same byte order as a long integer.
double	double	8	64-bit IEEE floating point number.
sfloat	-	2	16-bit OpenGL floating point number

The byte-ordering of integers in input data is defined by that of the sync-tag (magic number) preceding top-level items. Therefore, the byte-ordering in a top-level item may differ from the ordering in a previous item. For output data the default ordering is so far to have the least-significant bytes first. This is the natural byte order on Mips R3000 and higher (under Ultrix), DEC Alpha, VAX, and Intel (80)x86 CPUs but the inverse of the natural byte order on Motorola 680x0, RS6000, PowerPC, and Sparc CPUs. The ordering may change without notice and without changing version numbers. Except for performance considerations, the byte-ordering should not be relevant as long as only the 0-1-2-3 and 3-2-1-0 orders are considered, and byte ordering of floating point numbers is the same as for long integers. Byte ordering for writing may be changed during run-time with the 'byte_order' element of the I/O buffer structure. Note that on CPUs with non-IEEE floating point format like VAX writing and reading of floating point numbers is likely to be less efficient than on IEEE-format CPUs.

Note that if an 'int' variable is written via 'put_short()' and then read again via 'get_short()' not only the upper two bytes (on a 32-bit machine) are lost but also the sign bit is propagated from bit 15 to the upper 16 bits. Similarly, if a 'long' variable is written via 'put_long()' and later read via 'get_long()' on a 64-bit-machine, not only the upper 4 bytes are lost but also the sign in bit 31 is propagated to the upper 32 bits.

Do not modify this file to include project-specific things!

6.3.2 Macro Definition Documentation

6.3.2.1 #define READ_BYTES(fd, buf, nb)

Value:

```
((int)fd==0) ? \
(ssize_t) fread((void *)buf, (size_t)1, (size_t)nb, stdin) : \
read(fd, (void *)buf, (size_t)nb) )
```

6.3.3 Function Documentation

6.3.3.1 IO_BUFFER* allocate_io_buffer (size_t buflen)

Dynamic allocation of an I/O buffer. The actual length of the buffer is passed as an argument. The buffer descriptor is initialized.

Parameters

<i>buflen</i>	The length of the actual buffer in bytes. A safety margin of 4 bytes is added.
---------------	--

Returns

Pointer to I/O buffer or NULL if allocation failed.

References `_struct_IO_BUFFER::aux_count`, `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::extended`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::is_allocated`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_BUFFER::max_length`, `_struct_IO_BUFFER::min_length`, `_struct_IO_BUFFER::output_file`, `_struct_IO_BUFFER::output_fileno`, `_struct_IO_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_BUFFER::sync_err_count`, `_struct_IO_BUFFER::sync_err_max`, `_struct_IO_BUFFER::user_function`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.2 `int append_io_block_as_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, BYTE * buffer, long length)`

Append the data from a complete i/o block as an additional subitem to another i/o block.

Parameters

<i>iobuf</i>	The target I/O buffer descriptor, must be 'opened' for 'writing', i.e. <code>'put_item_begin()'</code> must be called.
<i>item_header</i>	Item header of the item in iobuf which is currently being filled.
<i>buffer</i>	Data to be filled in. Must be all data from an I/O buffer, including the 4 signature bytes.
<i>length</i>	The length of buffer in bytes.

Returns

0 (o.k.), -1 (error), -2 (not enough memory etc.)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.3 `int copy_item_to_io_block (IO_BUFFER * iobuf2, IO_BUFFER * iobuf, const IO_ITEM_HEADER * item_header)`

Parameters

<i>iobuf2</i>	Target I/O buffer descriptor.
<i>iobuf</i>	Source I/O buffer descriptor.
<i>item_header</i>	Header for the item in iobuf that should be copied to iobuf2.

Returns

0 (o.k.), -1 (error), -2 (not enough memory etc.)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `reset_io_block()`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.4 `void dbl_to_sfloat (double dnum, uint16_t * snum)`

This is done via an intermediate float representation.

Parameters

<i>dnum</i>	The number to be converted.
<i>snum</i>	Pointer for the resulting representation, as stored in an unsigned 16-bit integer (1 bit sign, 5 bits exponent, 10 bits mantissa).

References `flt_to_sfloat()`.

6.3.3.5 `const char* eventio_registered_typename (unsigned long type)`

This functions using the stored function pointer are now in the core eventio code.

References `find_ev_reg()`, `ev_reg_entry::name`, and `none`.

6.3.3.6 `int extend_io_buffer (IO_BUFFER * iobuf, unsigned next_byte, long increment)`

Extend the dynamically allocated I/O buffer and if an item has been started and the argument 'next_byte' is smaller than 256 that argument will be appended as the next byte to the buffer.

Parameters

<i>iobuf</i>	The I/O buffer descriptor
<i>next_byte</i>	The value of the next byte or ≥ 256
<i>increment</i>	The no. of bytes by which to increase the buffer beyond the current point. If there is remaining space for writing, the buffer is extended by less than 'increment'.

Returns

`next_byte` (modulo 256) if successful, -1 for failure

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::is_allocated`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::max_length`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.7 `int find_io_block (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Read byte for byte from the input file specified for the I/O buffer and look for the sync-tag (magic number in little-endian or big-endian byte order. As long as the input is properly synchronized this sync-tag should be found in the first four bytes. Otherwise, input data is skipped until the next sync-tag is found. After the sync tag 10 more bytes (item type, version number, and length field) are read. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	An item header structure to be filled in.

Returns

0 (O.k.), -1 (error), or -2 (end-of-file)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `get_item_begin()`, `get_long()`, `get_uint32()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::sync_err_count`, `_struct_IO_BUFFER::sync_err_max`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_BUFFER::user_function`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.8 `void flt_to_sfloat (const float * fnum, uint16_t * snum)`

Both input and output come as pointers to avoid extra conversions.

Parameters

<i>fnum</i>	Pointer to the number to be converted.
<i>snum</i>	Pointer for the resulting representation, as stored in an unsigned 16-bit integer (1 bit sign, 5 bits exponent, 10 bits mantissa).

6.3.3.9 void free_io_buffer (IO_BUFFER * iobuf)

Free an I/O buffer that has been allocated at run-time (e.g. by a call to allocate_io_buf()).

Parameters

<i>iobuf</i>	The buffer descriptor to be de-allocated.
--------------	---

Returns

(none)

References `_struct_IO_BUFFER::buffer`, and `_struct_IO_BUFFER::is_allocated`.

6.3.3.10 uintmax_t get_count (IO_BUFFER * iobuf)

Get an unsigned integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. Even though the scheme in principle allows for arbitrary length data, the current implementation is limited for data of up to 64 bits. On systems with `uintmax_t` shorter than 64 bits, the result could be clipped unnoticed. It could also be clipped unnoticed in the application calling this function.

6.3.3.11 uint16_t get_count16 (IO_BUFFER * iobuf)

Get an unsigned 16 bit integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. This is a shorter version of `get_count`, for efficiency reasons.

6.3.3.12 uint32_t get_count32 (IO_BUFFER * iobuf)

Get an unsigned 32 bit integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. This is a shorter version of `get_count`, for efficiency reasons.

References `_struct_IO_BUFFER::data`.

6.3.3.13 double get_double (IO_BUFFER * iobuf)

Get a double-precision floating point number (as written by `put_double`) from the I/O buffer. The current implementation is only for machines using IEEE format internally.

Parameters

<i>iobuf</i>	– The I/O buffer descriptor;
--------------	------------------------------

Returns

The floating point number.

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.3.3.14 int32_t get_int32 (IO_BUFFER * iobuf)

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see [put_short\(\)](#)).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.3.3.15 int get_item_begin (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)

Reads the header of an item.

Reads the header of an item. If a specific item type is requested but a different type is found and the length of that item is known, the item is skipped.

Parameters

<i>iobuf</i>	The input buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error), -2 (end-of-buffer) or -3 (wrong item type).

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `get_long()`, `get_uint32()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.16 `int get_item_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Finish reading an item. The pointer in the I/O buffer is at the end of the item after this call, if succesful.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.17 `long get_long (IO_BUFFER * iobuf)`

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.3.3.18 `int get_long_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a long string of ASCII characters with leading count of bytes from an I/O buffer. Strings can be up to $2^{31}-1$ bytes long (assuming you have so much memory).

To work properly with strings longer than 32k, a machine with `sizeof(int) > 2` is actually required.

NOTE: the `nmax` count does account also for the trailing zero byte which will be appended.

References `_struct_IO_BUFFER::data`, `get_int32()`, and `get_vector_of_byte()`.

6.3.3.19 `double get_real (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	The I/O buffer descriptor;
--------------	----------------------------

Returns

The floating point number.

References `get_int32()`, and `get_long()`.

6.3.3.20 `intmax_t get_scount (IO_BUFFER * iobuf)`

Get a signed integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. Even though the scheme in principle allows for arbitrary length data, the current implementation is limited for data of up to 64 bits. On systems with `intmax_t` shorter than 64 bits, the result could be clipped unnoticed.

References `get_count()`.

6.3.3.21 `int get_short (IO_BUFFER * iobuf)`

Get a two-byte integer with least significant byte first. Should be machine-independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.3.3.22 `int get_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a string of ASCII characters with leading count of bytes (stored with 16 bits) from an I/O buffer.

NOTE: the `nmax` count does now account for the trailing zero byte which will be appended. This was different in an earlier version of this function where one additional byte had to be available for the trailing zero byte.

References `_struct_IO_BUFFER::data`, `get_short()`, and `get_vector_of_byte()`.

6.3.3.23 `uint16_t get_uint16 (IO_BUFFER * iobuf)`

Get one unsigned short (16-bit unsigned int) from an I/O buffer. The function should be used where sign propagation is of concern.

Parameters

<i>iobuf</i>	The output buffer descriptor.
--------------	-------------------------------

Returns

The value obtained from the I/O buffer.

References `get_vector_of_uint16()`.

6.3.3.24 `uint32_t get_uint32 (IO_BUFFER * iobuf)`

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.3.3.25 `int get_var_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a string of ASCII characters with leading count of bytes (stored with variable length) from an I/O buffer.

NOTE: the `nmax` count does also account for the trailing zero byte which will be appended.

References `_struct_IO_BUFFER::data`, `get_count()`, and `get_vector_of_byte()`.

6.3.3.26 `void get_vector_of_byte (BYTE * vec, int num, IO_BUFFER * iobuf)`

Parameters

<i>vec</i>	– Byte data vector.
<i>num</i>	– Number of bytes to get.
<i>iobuf</i>	– I/O buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::data`.**6.3.3.27 void get_vector_of_uint16 (uint16_t * uval, int num, IO_BUFFER * iobuf)**

Get a vector of unsigned shorts from an I/O buffer with least significant byte first. The values are in the range 0 to 65535. The function should be used where sign propagation is of concern.

Parameters

<i>uval</i>	The vector where the values should be loaded.
<i>num</i>	The number of elements to load.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.**6.3.3.28 void get_vector_of_uint16_scount_differential (uint16_t * vec, int num, IO_BUFFER * iobuf)**

For optimization reasons it is assumed that the data has been written in a consistent way and is complete. Only minimal tests for remaining data in the buffer are made - while the slower generic version would check for each extracted number.

References `_struct_IO_BUFFER::data`, and `get_scount32()`.**6.3.3.29 void get_vector_of_uint32_scount_differential (uint32_t * vec, int num, IO_BUFFER * iobuf)**

For optimization reasons it is assumed that the data has been written in a consistent way and is complete. Only minimal tests for remaining data in the buffer are made - while the slower generic version would check for each extracted number.

References `_struct_IO_BUFFER::data`, and `get_scount32()`.**6.3.3.30 int list_io_blocks (IO_BUFFER * iobuf, int verbosity)**

List type, version, ident, and length) of the top item of all I/O blocks in input file onto standard output.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>verbosity</i>	Try showing type name at ≥ 1 , description at ≥ 2 .

Returns

0 (O.k.), -1 (error)

References `_struct_IO_BUFFER::byte_order`, `eventio_registered_description()`, `eventio_registered_typename()`, `find_io_block()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `skip_io_block()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::user_flag`, and `_struct_IO_ITEM_HEADER::version`.

6.3.3.31 `int list_sub_items (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, int maxlevel, int verbosity)`

Display the contents (item types, versions, idents and lengths) of sub-items on standard output.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of the item from which to show contents.
<i>maxlevel</i>	The maximum nesting depth to show contents (counted from the top-level item on).
<i>verbosity</i>	Try showing type name at ≥ 1 , description at ≥ 2 .

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `eventio_registered_description()`, `eventio_registered_typename()`, `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `list_sub_items()`, `search_sub_item()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, and `_struct_IO_ITEM_HEADER::version`.

6.3.3.32 long next_subitem_ident (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::data`, `get_item_begin()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `unset_item()`.

6.3.3.33 long next_subitem_length (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::data`, `get_item_begin()`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_ITEM_HEADER::type`, and `unset_item()`.

6.3.3.34 int next_subitem_type (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `get_long()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, and `_struct_IO_BUFFER::item_start_offset`.

6.3.3.35 void put_count (uintmax_t *n*, IO_BUFFER * *iobuf*)

Put an unsigned integer of unspecified length in a way similar to the UTF-8 character encoding to an I/O buffer. The byte order resulting in the buffer is independent of the host byte order or the byte order in action for the I/O buffer, starting with as many leading bits in the first byte as extension bytes needed after the first byte. While the scheme in principle allows for values of arbitrary length, the implementation is limited to 64 bits.

Parameters

<i>n</i>	The number to be saved. Even on systems with 64-bit integers, this must not exceed 2**32-1 with the current implementation.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `put_vector_of_byte()`.**6.3.3.36** `void put_count16 (uint16_t n, IO_BUFFER * iobuf)`

Returns

(none)

References `put_vector_of_byte()`.**6.3.3.37** `void put_count32 (uint32_t n, IO_BUFFER * iobuf)`

Returns

(none)

References `put_vector_of_byte()`.**6.3.3.38** `void put_double (double dnum, IO_BUFFER * iobuf)`

Put a 'double' (floating point) number in a specific but (almost) machine-independent format into an I/O buffer. This implementation requires the machine to use IEEE double-precision floating point numbers. Only byte order conversion is done.

Parameters

<i>dnum</i>	The number to be put into the I/O buffer.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.39 `void put_int32 (int32_t num, IO_BUFFER * iobuf)`

Write a four-byte integer with least significant bytes first. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.40 `int put_item_begin (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When putting another item to the output buffer which may be either a top item or a sub-item, `put_item_begin()` initializes the buffer (for a top item) and puts the item header on the buffer.

Parameters

<i>iobuf</i>	The output buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.) or -1 (error)

References `put_item_begin_with_flags()`.

6.3.3.41 `int put_item_begin_with_flags (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, int user_flag, int extended)`

This is identical to `put_item_begin()` except for taking a third and fourth argument, a user flag to be included in the header data, and a flag indicating that the header extension should be used. In `put_item_begin()` these flags are forced to 0 (false) for backwards compatibility.

Parameters

<i>iobuf</i>	The output buffer descriptor.
<i>item_header</i>	The item header descriptor.
<i>flag</i>	The user flag (0 or 1).

Returns

0 (O.k.) or -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::extended`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `put_long()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.42 `int put_item_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When finished with putting an item to the output buffer, check for errors and do housekeeping.

Parameters

<i>iobuf</i>	The output buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.) or -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `put_uint32()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_BUFFER::w_remaining`, and `write_io_block()`.

6.3.3.43 `void put_long (long num, IO_BUFFER * iobuf)`

Write a four-byte integer with least significant bytes first. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.44 `int put_long_string (const char * s, IO_BUFFER * iobuf)`

Put a long string of ASCII characters with leading count of bytes into an I/O buffer. This is expected to work properly for strings of more than 32k only on machines with `sizeof(int) > 2` because 16-bit machines may not be able to represent lengths of long strings (as obtained with `strlen`).

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_int32()`, `put_short()`, and `put_vector_of_byte()`.

6.3.3.45 `void put_real (double dnum, IO_BUFFER * iobuf)`

Put a 'double' (floating point) number in a specific but (almost) machine-independent format into an I/O buffer. Not the full precision of a 'double' is saved but a 32 bit IEEE floating point number is written (with the same byte ordering as long integers). On machines with other floating point format than IEEE the input number is converted to a IEEE number first. An optimized (machine- specific) version should compute the output data by shift and add operations rather than by `log()`, divide, and multiply operations on such non-IEEE-format machines (implemented for VAX only).

Parameters

<i>dnum</i>	The number to be put into the I/O buffer.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

(none)

References `put_int32()`, and `put_long()`.

6.3.3.46 `void put_scount (intmax_t n, IO_BUFFER * iobuf)`

Put a signed integer of unspecified length in a way similar to the UTF-8 character encoding to an I/O buffer. The byte order resulting in the buffer is independent of the host byte order or the byte order in action for the I/O buffer, starting with as many leading bits in the first byte as extension bytes needed after the first byte. While the scheme in principle allows for values of arbitrary length, the implementation is limited to 32 bits. To allow an efficient representation of negative numbers, the sign bit is stored in the least significant bit. Portability of data across machines with different `intmax_t` sizes and the need to represent also the most negative number ($-(2^{31})$, $-(2^{63})$, or $-(2^{127})$, depending on CPU type and compiler) is achieved by putting the number's modulus minus 1 into the higher bits.

Parameters

<i>n</i>	The number to be saved. It can be in the range from $-(2^{63})$ to $2^{63}-1$ on systems with 64 bit integers (intrinsic or through the compiler) and from $-(2^{31})$ to $2^{31}-1$ on pure 32 bit systems.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `put_count()`.

6.3.3.47 `void put_scount16 (int16_t n, IO_BUFFER * iobuf)`

Apart from efficiency, the data can be read with identical results through `get_scount16` or `get_scount`.

Returns

(none)

References `put_count()`.**6.3.3.48 void put_scount32 (int32_t n, IO_BUFFER * iobuf)**

Apart from efficiency, the data can be read with identical results through `get_scount32` or `get_scount`.

Returns

(none)

References `put_count()`.**6.3.3.49 void put_short (int num, IO_BUFFER * iobuf)**

Put a two-byte integer on an I/O buffer with least significant byte first. Should be machine independent as long as 'short' and 'unsigned short' are 16-bit integers, the two's complement is used for negative numbers, and the '>>' operator does a logical shift with unsigned short. Although the 'num' argument is a 4-byte integer on most machines, the value should be in the range -32768 to 32767.

Parameters

<i>num</i>	The number to be saved. Should fit into a short integer and will be truncated otherwise.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.50 int put_string (const char * s, IO_BUFFER * iobuf)

Put a string of ASCII characters with leading count of bytes (stored with 16 bits) into an I/O buffer.

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_short()`, and `put_vector_of_byte()`.**6.3.3.51 void put_uint32 (uint32_t num, IO_BUFFER * iobuf)**

Write a four-byte integer with least significant bytes first. Should be machine independent (see [put_short\(\)](#)).

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.52 int put_var_string (const char * s, IO_BUFFER * iobuf)

Put a string of ASCII characters with leading count of bytes (stored with variable length) into an I/O buffer. Note that storing strings of 32k or more length will not work on systems with `sizeof(int)==2`.

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_count()`, and `put_vector_of_byte()`.

6.3.3.53 `void put_vector_of_byte (const BYTE * vec, int num, IO_BUFFER * iobuf)`

Parameters

<i>vec</i>	Byte data vector.
<i>num</i>	Number of bytes to be put.
<i>iobuf</i>	I/O buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.54 `void put_vector_of_double (const double * dvec, int num, IO_BUFFER * iobuf)`

Put a vector of 'double' floating point numbers as IEEE 'double' numbers into an I/O buffer.

References `put_double()`.

6.3.3.55 `void put_vector_of_int (const int * vec, int num, IO_BUFFER * iobuf)`

Put a vector of integers (with actual values in the range -32768 to 32767) into an I/O buffer. This may be replaced by a more efficient but machine-dependent version later.

References `put_short()`.

6.3.3.56 `void put_vector_of_short (const short * vec, int num, IO_BUFFER * iobuf)`

Put a vector of 2-byte integers on an I/O buffer. This may be replaced by a more efficient but machine-dependent version later. May be called by a number of elements equal to 0. In this case, nothing is done.

References `put_short()`.

6.3.3.57 `void put_vector_of_uint16 (const uint16_t * uval, int num, IO_BUFFER * iobuf)`

Put a vector of unsigned shorts into an I/O buffer with least significant byte first. The values are in the range 0 to 65535. The function should be used where sign propagation is of concern.

Parameters

<i>uval</i>	The vector of values to be saved.
<i>num</i>	The number of elements to save.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.58 `int read_io_block (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

This function is called for reading data after an I/O data block has been found (with `find_io_block`) on input. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error), -2 (end-of-file), -3 (block skipped because it is too large)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data_pending`, `extend_io_buffer()`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `skip_io_block()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::user_function`.

6.3.3.59 `int remove_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

If writing an item has already started and then some condition was found to remove the item again, this is the function for it. The item to be removed should be the last one written, since anything following it will be forgotten too.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item to be removed.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `put_uint32()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.60 `int reset_io_block (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
--------------	----------------------------

Returns

0 (O.k.), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::min_length`, `_struct_IO_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.61 `int rewind_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When reading from an I/O buffer, go back to the beginning of the data area of an item. This is typically used when searching for different types of sub-blocks but processing should not depend on the relative order of them.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
--------------	------------------------

<i>item_header</i>	Header of item last read.
--------------------	---------------------------

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.62 `int search_sub_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, IO_ITEM_HEADER * sub_item_header)`

Search for an item of a specified type, starting at the current position in the I/O buffer. After successful action the buffer data pointer points to the beginning of the header of the first item of that type. If no such item is found, it points right after the end of the item of the next higher level.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	The header of the item within which we search.
<i>sub_item_header</i>	To be filled with what we found.

Returns

0 (O.k., sub-item was found), -1 (error), -2 (no such sub-item), -3 (cannot skip sub-items),

References `_struct_IO_BUFFER::buffer`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `get_item_begin()`, `get_item_end()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.63 `void set_eventio_registry_hook (struct ev_reg_entry *(*)(unsigned long t) fptr)`

This search function is also responsible for initializing the registry. By default, no such registry is used.

Parameters

<i>fptr</i>	A pointer to the registry search function.
-------------	--

References `find_ev_reg()`.

6.3.3.64 `int skip_io_block (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Skip the data of an I/O block from the input (after the block's header was read). This is the alternative to [read_io_block\(\)](#) after having found an I/O block with `find_io_block` but realizing that this is a type of block you don't know how to read or simply not interested in. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error) or -2 (end-of-file)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::user_function`.

6.3.3.65 int skip_subitem (IO_BUFFER * *iobuf*)

Parameters

<i>iobuf</i>	I/O buffer descriptor.
--------------	------------------------

Returns

0 (ok), -1 (error)

References `get_item_begin()`, `get_item_end()`, and `_struct_IO_ITEM_HEADER::type`.

6.3.3.66 `int unget_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When reading from an I/O buffer, go back to the beginning of an item (more precisely: its header) currently being read.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.67 `int unput_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When writing to an I/O buffer, revert anything yet written at the present level. If the buffer was extended, the last length is kept.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_ITEM_HEADER::use_extension`, and `_struct_IO_BUFFER::w_remaining`.

6.3.3.68 `int write_io_block (IO_BUFFER * iobuf)`

The complete I/O block is written to the output destination, which can be raw I/O (through `write`), buffered I/O (through `fwrite`) or user-defined I/O (through a user function). All items must have been closed before.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
--------------	----------------------------

Returns

0 (O.k.), -1 (error), -2 (item has no data)

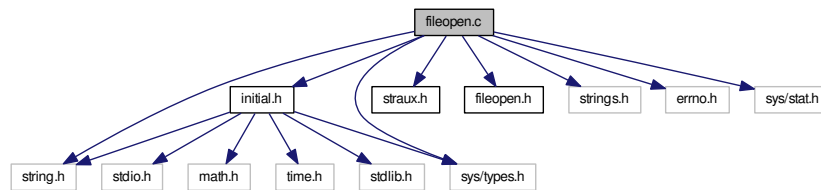
References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::output_file`, `_struct_IO_BUFFER::output_fileno`, `_struct_IO_BUFFER::user_function`, and `_struct_IO_BUFFER::w_remaining`.

6.4 fileopen.c File Reference

Allow searching of files in declared include paths (fopen replacement).

```
#include "initial.h"
#include "straux.h"
#include "fileopen.h"
#include <string.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Include dependency graph for fileopen.c:



Data Structures

- struct [incpath](#)
An element in a linked list of include paths.

Functions

- void [addexepath](#) (const char *name)
Add a path to the list of execution paths, if not already there.
- void [addpath](#) (const char *name)
Add a path to the list of include paths, if not already there.
- static FILE * [cmp_popen](#) (const char *fname, const char *mode, int compression)
Helper function for opening a compressed file through a fifo.
- void [disable_permissive_pipes](#) ()
Disable the permissive execution of pipes.
- void [enable_permissive_pipes](#) ()
Enable the permissive execution of pipes.
- static FILE * [exe_popen](#) (const char *fname, const char *mode)
Helper function for opening a pipe from or to a given program.
- int [fileclose](#) (FILE *f)
Close a file or fifo but not if it is one of the standard streams.
- FILE * [fileopen](#) (const char *fname, const char *mode)
Search for a file in the include path list and open it if possible.
- static FILE * [fopenx](#) (const char *fname, const char *mode)
- static void [freeexepath](#) ()
Free a whole list of execution path elements.
- static void [freepath](#) ()
Free a whole list of include path elements.
- void [initexepath](#) (const char *default_exe_path)

- void `initpath` (const char *default_path)
Init the path list, with default_path as the only entry.
- void `listpath` (char *buffer, size_t bufsize)
Show the list of include paths.
- static FILE * `popenx` (const char *fname, const char *mode)
- void `set_permissive_pipes` (int p)
Enable or disable the permissive execution of pipes.
- static FILE * `ssh_popen` (const char *fname, const char *mode, int compression)
Helper function for opening a file on a remote SSH server.
- static FILE * `uri_popen` (const char *fname, const char *mode, int compression)
Helper function for opening a file with a URI (`http://` etc.).

Variables

- static int `parallel` = 0
- static int `permissive_pipes` = 0
Allow any execution pipe command if this variable is non-zero.
- static struct `incpath` * `root_exe_path` = NULL
The starting element for execution paths.
- static struct `incpath` * `root_path` = NULL
The starting element of include paths.
- static int `verbose` = 0
Use to decide if open/close success/failure is reported.

6.4.1 Detailed Description

The functions provided in this file provide an enhanced replacement `fileopen()` for the C standard library's `fopen()` function. The enhancements are in several areas:

- Where possible files are opened such that more than 2 gigabytes of data can be accessed on 32-bit systems when suitably compiled. This also works with software where a `'-D_FILE_OFFSET_BITS=64'` at compile-time cannot be used (of which ROOT is an infamous example).
- For reading files, a list of paths can be configured before the the first `fileopen()` call and all files without absolute paths will be searched in these paths. Writing always strictly follows the given file name and will not search in the path list.
- Files compressed with `gzip` or `bzip2` can be handled on the fly. Files with corresponding file name extensions (`.gz` and `.bz2`) will be automatically decompressed when reading or compressed when writing (in a pipe, i.e. without producing temporary copies).
- In the same way, files compressed with `lzop` (for extension `.lzo`), `lzma` (for extension `.lzma`) as well as `xz` (for extension `@.xz`) and `lz4` (for extension `.lz4`) are handled on the fly. No check is made if these programs are installed.
- URIs (uniform resource identifiers) starting with `http:`, `https:`, or `ftp:` will also be opened in a pipe, with optional decompression, depending on the ending of the URI name. You can therefore easily process files located on a web or ftp server. Access is limited to reading.
- Files on any SSH server where you can login without a password can be read as `'ssh://user:filepath'` where filepath can be an absolute path (starting with `'/'`) or one relative to the users home directory.
- Input and output can also be from/to a user-defined program. Restrictions apply there which prevent execution of any program by default. Either a list of accepted execution paths has to be set up beforehand with `initexepath()/addexepath()` or permissive mode can be enabled, allowing execution of any given program.

Author

Konrad Bernloehr

Date

Nov. 2000

CVS \$Date: 2016/06/24 16:10:48 \$

Version

CVS \$Revision: 1.24 \$

6.4.2 Function Documentation**6.4.2.1 void addexepath (const char * *name*)**

The path name is always copied to a newly allocated memory location. This path name can actually be a colon-separated list, as for `initexepath()`.

References `addpath()`, `root_exe_path`, and `root_path`.

6.4.2.2 void addpath (const char * *name*)

The path name is always copied to a newly allocated memory location. This path name can actually be a colon-separated list, as for `initpath()`. Also environment variables (indicated by starting with '\$', e.g. "\$HOME") are accepted (and may expand into colon-separated list) but no mixed expansion (like "\$HOME/bin").

References `getword()`, `incpath::next`, `incpath::path`, `root_path`, and `verbose`.

6.4.2.3 static FILE* cmp_popen (const char * *fname*, const char * *mode*, int *compression*) [static]

References `verbose`.

6.4.2.4 void disable_permissive_pipes (void)**6.4.2.5 void enable_permissive_pipes (void)****6.4.2.6 static FILE* exe_popen (const char * *fname*, const char * *mode*) [static]**

References `incpath::next`, `incpath::path`, and `verbose`.

6.4.2.7 int fileclose (FILE * *f*)

References `verbose`.

6.4.2.8 FILE* fileopen (const char * *fname*, const char * *mode*)

References `cmp_popen()`, `exe_popen()`, `initpath()`, `incpath::next`, `incpath::path`, `root_path`, `ssh_popen()`, `uri_popen()`, and `verbose`.

6.4.2.9 static void freeexepath () [static]

References `incpath::next`, and `incpath::path`.

6.4.2.10 static void freepath () [static]

References `incpath::next`, and `incpath::path`.

6.4.2.11 void initpath (const char * *default_path*)

References `addpath()`, `freepath()`, `getword()`, and `verbose`.

6.4.2.12 void listpath (char * *buffer*, size_t *bufsize*)

References incpath::next, and incpath::path.

6.4.2.13 void set_permissive_pipes (int *p*)

References disable_permissive_pipes(), and enable_permissive_pipes().

6.4.2.14 static FILE * uri_popen (const char * *fname*, const char * *mode*, int *compression*) [static]

References verbose.

6.4.3 Variable Documentation

6.4.3.1 int permissive_pipes = 0 [static]

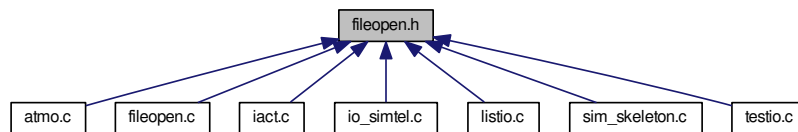
6.4.3.2 struct incpath* root_exe_path = NULL [static]

6.4.3.3 struct incpath* root_path = NULL [static]

6.5 fileopen.h File Reference

Function prototypes for [fileopen.c](#).

This graph shows which files directly or indirectly include this file:



Functions

- void [addexepath](#) (const char **name*)
Add a path to the list of execution paths, if not already there.
- void [addpath](#) (const char **name*)
Add a path to the list of include paths, if not already there.
- void [disable_permissive_pipes](#) (void)
Disable the permissive execution of pipes.
- void [enable_permissive_pipes](#) (void)
Enable the permissive execution of pipes.
- int [fileclose](#) (FILE **f*)
Close a file or fifo but not if it is one of the standard streams.
- FILE * [fileopen](#) (const char **fname*, const char **mode*)
Search for a file in the include path list and open it if possible.
- void [initexepath](#) (const char **default_path*)
- void [initpath](#) (const char **default_path*)
Init the path list, with default_path as the only entry.
- void [listpath](#) (char **buffer*, size_t *bufsize*)
Show the list of include paths.
- void [set_permissive_pipes](#) (int *p*)
Enable or disable the permissive execution of pipes.

6.5.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2014/06/23 09:34:45 \$

Version

CVS \$Revision: 1.7 \$

6.5.2 Function Documentation

6.5.2.1 void addexepath (const char * *name*)

The path name is always copied to a newly allocated memory location. This path name can actually be a colon-separated list, as for `initexepath()`.

References `addpath()`, `root_exe_path`, and `root_path`.

6.5.2.2 void addpath (const char * *name*)

The path name is always copied to a newly allocated memory location. This path name can actually be a colon-separated list, as for `initpath()`. Also environment variables (indicated by starting with '\$', e.g. "\$HOME") are accepted (and may expand into colon-separated list) but no mixed expansion (like "\$HOME/bin").

References `getword()`, `incpath::next`, `incpath::path`, `root_path`, and `verbose`.

6.5.2.3 void disable_permissive_pipes (void)

6.5.2.4 void enable_permissive_pipes (void)

6.5.2.5 int fileclose (FILE * *f*)

References `verbose`.

6.5.2.6 FILE* fopen (const char * *fname*, const char * *mode*)

References `cmp_popen()`, `exe_popen()`, `initpath()`, `incpath::next`, `incpath::path`, `root_path`, `ssh_popen()`, `uri_popen()`, and `verbose`.

6.5.2.7 void initpath (const char * *default_path*)

References `addpath()`, `freepath()`, `getword()`, and `verbose`.

6.5.2.8 void listpath (char * *buffer*, size_t *bufsize*)

References `incpath::next`, and `incpath::path`.

6.5.2.9 void set_permissive_pipes (int *p*)

References `disable_permissive_pipes()`, and `enable_permissive_pipes()`.

6.6 iact.c File Reference

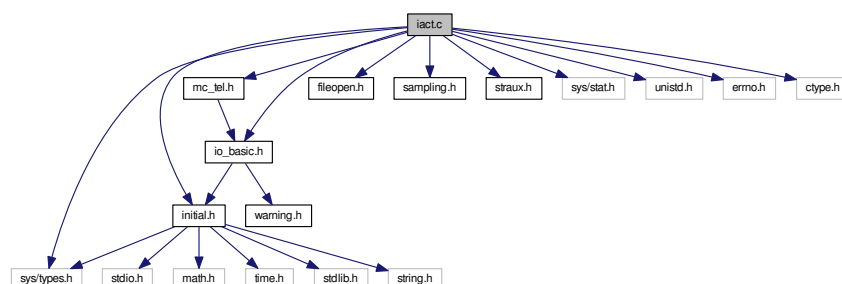
CORSIKA interface for Imaging Atmospheric Cherenkov Telescopes etc.

```

#include "initial.h"
#include "io_basic.h"
#include "mc_tel.h"
#include "fileopen.h"
#include "sampling.h"
#include "straux.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>

```

Include dependency graph for iact.c:



Data Structures

- struct [detstruct](#)

A structure describing a detector and linking its photons bunches to it.

- struct [gridstruct](#)

Macros

- `#define CORSIKA_VERSION 6900 /* the version that should be matched */`
- `#define EXTERNAL_STORAGE 1`
Enable external temporary bunch storage.
- `#define EXTRA_MEM 0`
- `#define EXTRA_MEM_1 EXTRA_MEM`
- `#define EXTRA_MEM_10 EXTRA_MEM`
- `#define EXTRA_MEM_11 EXTRA_MEM`
- `#define EXTRA_MEM_12 EXTRA_MEM`
- `#define EXTRA_MEM_2 EXTRA_MEM`
- `#define EXTRA_MEM_3 EXTRA_MEM`
- `#define EXTRA_MEM_4 EXTRA_MEM`
- `#define EXTRA_MEM_5 EXTRA_MEM`
- `#define EXTRA_MEM_6 EXTRA_MEM`
- `#define EXTRA_MEM_7 EXTRA_MEM`
- `#define EXTRA_MEM_8 EXTRA_MEM`
- `#define EXTRA_MEM_9 EXTRA_MEM`
- `#define GRID_SIZE 1000`
unit: cm
- `#define HAVE_EVENTIO_FUNCTIONS 1`
- `#define IACT_ATMO_VERSION "1.50 (2016-10-12)"`

- #define `INTERNAL_LIMIT` 100000
Start external storage after so many bunches.
- #define `max(a, b)` ((a)>(b)?(a):(b))
- #define `MAX_ARRAY_SIZE` 1000 /* Use the same limit as in CORSIKA itself. */
Maximum number of telescopes (or other detectors) per array.
- #define `MAX_BUNCHES` 5000000
- #define `MAX_CLASS` 1
- #define `MAX_IO_BUFFER` 200000000
- #define `min(a, b)` ((a)<(b)?(a):(b))
- #define `NBUNCH` 5000
Memory allocation step size for bunches.
- #define `PIPE_OUTPUT` 1
- #define `PRMPAR_SIZE` 17
- #define `square(x)` ((x)*(x))
- #define `UNKNOWN_LONG_DIST` 1

Typedefs

- typedef double `cors_dbl_t`
*Type for CORSIKA numbers which are currently REAL*8.*
- typedef float `cors_real_t`
*Type for CORSIKA floating point numbers remaining REAL*4.*

Functions

- static int `compact_photon_hit` (struct `detstruct` *det, double x, double y, double cx, double cy, double sx, double sy, double photons, double ctime, double zem, double lambda)
Store a photon bunch for a given telescope in compact format.
- static void `cross_prod` (double *v1, double *v2, double *v3)
Cross (outer) product of two 3-D vectors v1, v2 into 3-D vector v3.
- static int `expand_env` (char *fname, size_t maxlen)
Expanding environment variables ourselves rather than passing that on a shell later, so that we can still check characters after expansion.
- void `extprim_setup` (const char *text)
Placeholder function for activating and setting up user-defined (external to CORSIKA) controlled over types, spectra, and angular distribution of primaries.
- void `extprm_` (`cors_dbl_t` *type, `cors_dbl_t` *eprim, double *thetap, double *phip)
Placeholder function for external shower-by-shower setting of primary type, energy, and direction.
- void `get_iact_stats` (long *sb, double *ab, double *ap)
- void `get_impact_offset` (`cors_real_t` evth[273], `cors_dbl_t` prmpar[PRMPAR_SIZE])
Approximate impact offset of primary due to geomagnetic field.
- double `heigh_` (double *thickness)
The CORSIKA built-in function for the height as a function of overburden.
- static void `iact_param` (const char *text0)
Processing of IACT module specific parameters in Corsika input.
- double `iact_rndm` (int dummy)
- static int `in_detector` (struct `detstruct` *det, double x, double y, double sx, double sy)
Check if a photon bunch hits a particular telescope volume.
- static void `ioerrorcheck` ()
- static int `is_off` (char *word)
- static int `is_on` (char *word)
- static int `Nint_f` (double x)

- Nearest integer function.*

 - static double `norm3` (double *v)
- Norm of a 3-D vector.*

 - static void `norm_vec` (double *v)
- Normalize a 3-D vector.*

 - static int `photon_hit` (struct `detstruct` *det, double x, double y, double cx, double cy, double sx, double sy, double photons, double ctime, double zem, double lambda)
- Store a photon bunch for a given telescope in long format.*

 - double `refidx_` (double *height)
- Index of refraction as a function of altitude [cm].*

 - double `rhof_` (double *height)
- The CORSIKA built-in density lookup function.*

 - void `rmmard_` (double *, int *, int *)
 - static double `rndm` (int dummy)
- Random number interface using sequence 4 of CORSIKA.*

 - void `sample_offset` (const char *sampling_fname, double core_range, double theta, double phi, double thetaref, double phiref, double offax, double E, int primary, double *xoff, double *yoff, double *sampling_area)
- Get uniformly sampled or importance sampled offset of array with respect to core, in the plane perpendicular to the shower axis.*

 - static int `set_random_systems` (double theta, double phi, double thetaref, double phiref, double offax, double E, int primary, int volflag)
- Randomly scatter each array of detectors in given area.*

 - void `telasu_` (int *n, `cors_dbl_t` *dx, `cors_dbl_t` *dy)
- Setup how many times each shower is used.*

 - void `telend_` (`cors_real_t` evte[273])
- End of event.*

 - void `televt_` (`cors_real_t` evth[273], `cors_dbl_t` prmpar[PRMPAR_SIZE])
- Start of new event.*

 - void `telfil_` (const char *name0)
- Define the output file for photon bunches hitting the telescopes.*

 - void `telinf_` (int *itel, double *x, double *y, double *z, double *r, int *exists)
- Return information about configured telescopes back to CORSIKA.*

 - void `telling_` (int *type, double *data, int *ndim, int *np, int *nthick, double *thickstep)
- Write CORSIKA 'longitudinal' (vertical) distributions.*

 - void `tellni_` (const char *line, int *llength)
- Keep a record of CORSIKA input lines.*

 - int `telout_` (`cors_dbl_t` *bsize, `cors_dbl_t` *wt, `cors_dbl_t` *px, `cors_dbl_t` *py, `cors_dbl_t` *pu, `cors_dbl_t` *pv, `cors_dbl_t` *ctime, `cors_dbl_t` *zem, `cors_dbl_t` *lambda)
- Check if a photon bunch hits one or more simulated detector volumes.*

 - void `telrne_` (`cors_real_t` rune[273])
- Write run end block to the output file.*

 - void `telrnh_` (`cors_real_t` runh[273])
- Save aparameters from CORSIKA run header.*

 - void `telset_` (`cors_dbl_t` *x, `cors_dbl_t` *y, `cors_dbl_t` *z, `cors_dbl_t` *r)
- Add another telescope to the system (array) of telescopes.*

 - void `telshw_` ()
- Show what telescopes have actually been set up.*

 - void `telsmp_` (const char *name)
- Set the file name with parameters for importance sampling.*

Variables

- static double `airlightspeed` = 29.9792458/1.0002256
[cm/ns] at H=2200 m
- static double `all_bunches`
- static double `all_bunches_run`
- static double `all_photons`
- static double `all_photons_run`
- static double `Bfield` [3]
Magnetic field vector in detector coordinate system (with Bz positive if upwards)
- static double `bxplane` [3]
- static double `byplane` [3]
Spanning vectors of shower plane such that projection of B field is in bxplane direction.
- static double `core_range`
The maximum core offset of array centres in circular distribution.
- static double `core_range1`
The maximum core offsets in x,y for rectangular distribution.
- static double `core_range2`
- static struct `linked_string corsika_inputs` = { corsika_inputs_head, NULL }
- static char `corsika_inputs_head` [80]
- int `corsika_version` = (CORSIKA_VERSION)
The CORSIKA version actually running.
- static int `count_print_evt` = 0
- static int `count_print_tel` = 0
- static int `det_in_class` [MAX_CLASS]
- static struct `detstruct ** detector`
- static double `dmax` = 0.
Max.
- static int `do_print`
- static double `energy`
- static int `event_number`
- static double `first_int`
- static struct `gridstruct * grid`
- static int `grid_elements`
- static int `grid_nx`
- static int `grid_ny`
- static double `grid_x_high`
- static double `grid_x_low`
- static double `grid_y_high`
- static double `grid_y_low`
- static int `impact_correction` = 1
Correct impact position if non-zero.
- static double `impact_offset` [2]
Offset of impact position of charged primaries.
- static `IO_BUFFER * iobuf`
- static double `lambda1`
- static double `lambda2`
- static long `max_bunches` = MAX_BUNCHES
- static int `max_internal_bunches` = INTERNAL_LIMIT
The largest number of photon bunches kept in main memory before attempting to flush them to temporary files on disk.
- static size_t `max_io_buffer` = MAX_IO_BUFFER
The largest block size in the output data, which must hold all photons bunches of one array.

- static int **max_print_evt** = 100
- static int **max_print_tel** = 10
- static int **narray**
- static int * **ndet**
- static int **nevents**
- static int **nrun**
- static int **nsys** = 1
 - Number of arrays.*
- static int **ntel** = 0
 - Number of telescopes set up.*
- static double **obs_height**
- static double **off_axis**
- static char * **output_fname**
 - The name of the output file for eventio format data.*
- static double **phi_central**
- static double **phi_prim**
- static double **pprim** [3]
 - Momentum vector of primary particle.*
- static int **primary**
- static double **raise_tel**
 - Non-zero if any telescope has negative z.*
- static double **rmax** = 0.
 - Max.*
- static double **rtel** [MAX_ARRAY_SIZE]
- static char * **sampling_fname**
 - The name of the file providing parameters for importance sampling.*
- static int **skip_off2** = 1
- static int **skip_print** = 1
- static int **skip_print2** = 100
- static long **stored_bunches**
- static int **tel_individual** = 0
- static size_t **tel_split_threshold** = 10000000
- static int **televt_done**
- static double **theta_central**
 - The central value of the allowed ranges in theta and phi.*
- static double **theta_prim**
- static double **toffset**
- static int **use_compact_format** = 1
- static double **ush**
- static double **ushc**
- static double **vsh**
- static double **vshc**
- static double * **weight**
- int **with_extprim** = 0
- static double **wsh**
- static double **wshc**
- static double * **xoffset**
- static double **xtel** [MAX_ARRAY_SIZE]
 - Position and size definition of fiducial spheres.*
- static double * **yoffset**
- static double **ytel** [MAX_ARRAY_SIZE]
- static double **ztel** [MAX_ARRAY_SIZE]

6.6.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2016/10/13 15:50:54 \$

CVS \$Revision: 1.92 \$

Version 1.2.31 (for IACT/ATMO package version 1.50)

This file implements a CORSIKA interface for the simulation of (3-D) arrays of Cherenkov telescopes. A whole array may be simulated in multiple instances with random offsets of each instance. For full use of this software additional files are required which are available now on request from Konrad Bernloehr (e-mail: Konrad.Bernloehr@mpi-hd.mpg.de). These additional files should be included in the same add-on package to CORSIKA which includes this file. A fallback mechanism is included to use the normal CORSIKA output of Cherenkov photon bunches instead of the dedicated output functions from the unavailable files. However, this fallback mechanism has important drawbacks: information about positions of telescopes are completely lost and no photon bunches are collected in memory because the collected bunches would never be written out. For those reasons you are advised to obtain and use the additional software.

General comments on this file:

Routines provided in this file interface to recent versions of the CORSIKA air shower simulation program. Modifications to CORSIKA have been kept as simple as possible and the existing routines for production of Cherenkov light have been largely maintained. Setup of the telescope systems to be simulated is via the usual CORSIKA input file (the syntax of which has been extended by a few additional keywords). These telescope systems can be randomly scattered several times within a given area. All treatment whether a bunch of photons hits a telescope is done by the routines in this file. Photon bunches are kept in main memory until the end of the event. This might be a limitation when simulating large showers / many telescopes / many systems of telescopes on a computer with little memory. An option to store photon bunches in a temporary file has, therefore, been included. After the end of an event in CORSIKA all photon bunches (sorted by system and telescope) are written to a data file in the 'eventio' portable data format also used for CRT and HEGRA CT data. All CORSIKA run/event header/trailer blocks are also written to this file.

This version comes with sections for conditional compilation like EXTENDED_TELOUT CORSIKA is compiled with extended interface (IACTEXT option). Information about particles on ground is stored. IACTEXT For all practical purposes a synonym to EXTENDED_TELOUT. STORE_EMITTER Store information about all particles emitting light after the photon bunch. This duplicated the amount of data. Requires the IACTEXT option to be activated. MARK_DIRECT_LIGHT The Cherenkov photons bunches from the primary particle and its leading/major fragments are marked up with a non-zero wavelength (1: direct, 2, 3: major fragments). Requires the IACTEXT option to be activated. See the ALL_WL_RANDOM configuration parameter to sim_telarray. CORSIKA_SAVES_PHOTONS CORSIKA should save photons in its own format. IACT_NO_GRID For optimization of finding detector hits in normal simulations a detector is assigned to each grid cell onto which the detector throws a 'shadow', reducing the detector intersection checks a lot. In special shower simulations (example: a 10000 m high string of telescopes) or in technical simulations (e.g. using the IACT interface for artificial light sources, without CORSIKA) this may fail and explicit intersection of each photon bunch with each detector sphere needs to be checked. Note that these extensions may not have been tested in a long time. Use with care.

6.6.2 Function Documentation

6.6.2.1 `static int compact_photon_hit (struct detstruct * det, double x, double y, double cx, double cy, double sx, double sy, double photons, double ctime, double zem, double lambda) [static]`

Store a photon bunch in the bunch list for a given telescope. This bunch list is dynamically created and extended as required. This routine is using a more compact format than `photon_hit()`. This compact format is not appropriate when core distances of telescopes times sine of zenith angle exceed 1000 m.

Parameters

<i>det</i>	pointer to data structure of the detector hit.
<i>x</i>	X position in CORSIKA detection plane [cm]
<i>y</i>	Y position in CORSIKA detection plane [cm]
<i>cx</i>	Direction projection onto X axis
<i>cy</i>	Direction projection onto Y axis
<i>sx</i>	Slope with respect to X axis ($\text{atan}(sx) = \text{acos}(cx)$)
<i>sy</i>	Slope with respect to Y axis ($\text{atan}(sy) = \text{acos}(cy)$)
<i>photons</i>	Bunch size (sizes above 327 cannot be represented)
<i>ctime</i>	Arrival time of bunch in CORSIKA detection plane.
<i>zem</i>	Altitude of emission above sea level [cm]
<i>lambda</i>	Wavelength (0: undetermined, -1: converted to photo-electron)

Returns

0 (O.K.), -1 (failed to save photon bunch)

References compact_bunch::ctime, compact_bunch::cy, INTERNAL_LIMIT, compact_bunch::lambda, compact_bunch::log_zem, NBUNCH, compact_bunch::photons, and compact_bunch::y.

6.6.2.2 void extprim_setup (const char * text)**Parameters**

<i>text</i>	CORSIKA input card text following the 'IACT EXTPRIM' keywords. Could be parameter values or a file name.
-------------	--

6.6.2.3 void extprm_ (cors_dbl_t * type, cors_dbl_t * eprim, double * thetap, double * phip)

If primaries are to be generated following some special (non-power-law) spectrum, with a mixed composition, or with an angular distribution not achieved by built-in methods, a user-defined implementation of this function can be used to generate a mixture of primaries of any spectrum, composition, and angular distribution.

Be aware that this function would be called before televt_ and thus (at least for the first shower) before those run parameters not fitting into the run header are known.

Parameters

<i>type</i>	The type number of the primary to be used in CORSIKA (output).
<i>eprim</i>	The energy [GeV] to be used for the primary (output).
<i>thetap</i>	The zenith angle [rad] of the primary (output).
<i>phip</i>	The azimuth angle [rad] of the primary in the CORSIKA way (direction of movement from magnetic North counter-clockwise) (output).

6.6.2.4 void get_impact_offset (cors_real_t evth[273], cors_dbl_t prmpar[PRMPAR_SIZE])

Get the approximate impact offset of the primary particle due to deflection in the geomagnetic field. The approximation that the curvature radius is large compared to the distance travelled is used. The method is also not very accurate at large zenith angles where curvature of the atmosphere gets important. Therefore a zenith angle cut is applied and showers very close to the horizon are skipped. Only the offset at the lowest detection level is evaluated.

Parameters

<i>evth</i>	CORSIKA event header block
<i>prmpar</i>	CORSIKA primary particle block. We need it to get the particle's relativistic gamma factor (prmpar[2] or prmpar[1], depending on the CORSIKA version).

Returns

(none)

References heigh_().

6.6.2.5 double heigh_ (double * *thickness*)6.6.2.6 static void iact_param (const char * *text0*) [static]

Parameters

<i>text</i>	Text following the IACT keyword on the input line.
-------------	--

References extprim_setup(), getword(), telfil_(), and telsmp_().

6.6.2.7 static int in_detector (struct detstruct * *det*, double *x*, double *y*, double *sx*, double *sy*) [static]

Check if a photon bunch (or, similarly, a particle) hits a particular simulated telescope/detector.

Parameters

<i>x</i>	X position of photon position in CORSIKA detection level [cm]
<i>y</i>	Y position of photon position in CORSIKA detection level [cm]
<i>sx</i>	Slope of photon direction in X/Z plane.
<i>sy</i>	Slope of photon direction in Y/Z plane.

Returns

0 (does not hit), 1 (does hit)

6.6.2.8 static int Nint_f (double *x*) [static]6.6.2.9 static int photon_hit (struct detstruct * *det*, double *x*, double *y*, double *cx*, double *cy*, double *sx*, double *sy*, double *photons*, double *ctime*, double *zem*, double *lambda*) [static]

Store a photon bunch in the bunch list for a given telescope. It is kept in memory or temporary disk storage until the end of the event. This way, photon bunches are sorted by telescope. This bunch list is dynamically created and extended as required.

Parameters

<i>det</i>	pointer to data structure of the detector hit.
<i>x</i>	X position in CORSIKA detection plane [cm]
<i>y</i>	Y position in CORSIKA detection plane [cm]
<i>cx</i>	Direction projection onto X axis
<i>cy</i>	Direction projection onto Y axis
<i>sx</i>	Slope with respect to X axis ($\text{atan}(sx) = \text{acos}(cx)$)
<i>sy</i>	Slope with respect to Y axis ($\text{atan}(sy) = \text{acos}(cy)$)
<i>photons</i>	Bunch size
<i>ctime</i>	Arrival time of bunch in CORSIKA detection plane.
<i>zem</i>	Altitude of emission above sea level [cm]
<i>lambda</i>	Wavelength (0: undetermined, -1: converted to photo-electron)

Returns

0 (O.K.), -1 (failed to save photon bunch)

Note: With the EXTENDED_TELOUT every second call would have data of the emitting particle: the mass in *cx*, the charge in *cy*, the energy in *photons*, the time of emission in *zem*, and 9999 in *lambda*.

References fclose(), fopen(), bunch::lambda, and bunch::photons.

6.6.2.10 double refidx_ (double * *height*)

This function can be called from Fortran code as REFIDX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

index of refraction

6.6.2.11 `double rhof_ (double * height)`

6.6.2.12 `static double rndm (int dummy) [static]`

6.6.2.13 `void sample_offset (const char * sampling_fname, double core_range, double theta, double phi, double thetaref, double phiref, double offax, double E, int primary, double * xoff, double * yoff, double * sampling_area)`

Parameters

<i>sampling_fname</i>	Name of file with parameters, to be read on first call.
<i>core_range</i>	Maximum core distance as used in data format check [cm]. If not obeying this maximum distance, make sure to switch on the long data format manually.
<i>theta</i>	Zenith angle [radians]
<i>phi</i>	Shower azimuth angle in CORSIKA angle convention [radians].
<i>thetaref</i>	Reference zenith angle (e.g. of VIEWCONE centre) [radians].
<i>phiref</i>	Reference azimuth angle (e.g. of VIEWCONE centre) [radians].
<i>offax</i>	Angle between central direction (typically VIEWCONE centre) and the direction of the current primary [radians].
<i>E</i>	Energy of primary particle [GeV]
<i>primary</i>	Primary particle ID.
<i>xoff</i>	X offset [cm] to be generated.
<i>yoff</i>	Y offset [cm] to be generated.
<i>sampling_area</i>	Area weight of the generated sample (normalized to $\text{Pi} \times \text{core_range}^2$) [cm^2].

6.6.2.14 `static int set_random_systems (double theta, double phi, double thetaref, double phiref, double offax, double E, int primary, int vofflag) [static]`

The area containing the detectors is sub-divided into a rectangular grid and each detector with a (potential) intersection with a grid element is marked for that grid element. A detector can be marked for several grid elements unless completely inside one element. Checks which detector(s) is/are hit by a photon bunch (or, similarly, by a particle) is thus reduced to check only the detectors marked for the grid element which is hit by the photon bunch (or particle). The grid should be sufficiently fine-grained that there are usually not much more than one detector per element but finer graining than the detector sizes makes no sense.

Parameters

<i>theta</i>	Zenith angle of the shower following [radians].
<i>phi</i>	Shower azimuth angle in CORSIKA angle convention [radians].
<i>thetaref</i>	Reference zenith angle (e.g. of VIEWCONE centre) [radians].
<i>phiref</i>	Reference azimuth angle (e.g. of VIEWCONE centre) [radians].
<i>offax</i>	Angle between central direction (typically VIEWCONE centre) and the direction of the current primary [radians].
<i>E</i>	Primary particle energy in GeV (may be used in importance sampling).
<i>primary</i>	Primary particle ID (may be used in importance sampling).
<i>vofflag</i>	Set to 1 if CORSIKA was compiled with VOLUMEDET option, 0 otherwise.

Returns

0 (O.K.), -1 (error)

References `core_range`, `GRID_SIZE`, `Nint_f()`, `nsys`, `rndm()`, and `sample_offset()`.

6.6.2.15 void telasu_ (int * n, cors_dbl_t * dx, cors_dbl_t * dy)

Set up how many times the telescope system should be randomly scattered within a given area. Thus each telescope system (array) will see the same shower but at random offsets. Each shower is thus effectively used several times. This function is called according to the CSCAT keyword in the CORSIKA input file.

Parameters

<i>n</i>	The number of telescope systems
<i>dx</i>	Core range radius (if <i>dy</i> =0) or core x range
<i>dy</i>	Core y range (non-zero for rectangular, 0 for circular)

Returns

(none)

6.6.2.16 void telend_ (cors_real_t evte[273])

Write out all recorded photon bunches.

End of an event: write all stored photon bunches to the output data file, and the CORSIKA event end block as well.

Parameters

<i>evte</i>	CORSIKA event end block
-------------	-------------------------

Returns

(none)

References begin_write_tel_array(), end_write_tel_array(), shower_extra_parameters::is_set, _struct_IO_BUFFER::output_file, bunch::photons, write_tel_array_end(), write_tel_array_head(), write_tel_block(), write_tel_compact_photons(), and write_tel_photons().

6.6.2.17 void televt_ (cors_real_t evth[273], cors_dbl_t prmpar[PRMPAR_SIZE])

Save event parameters.

Start of new event: get parameters from CORSIKA event header block, create randomly scattered telescope systems in given area, and write their positions as well as the CORSIKA block to the data file.

Parameters

<i>evth</i>	CORSIKA event header block
<i>prmpar</i>	CORSIKA primary particle block

Returns

(none)

References atmosphere, clear_shower_extra_parameters(), core_range, core_range1, cross_prod(), get_impact_offset(), heigh_(), norm3(), norm_vec(), nsys, _struct_IO_BUFFER::output_file, refidx_(), rhof_(), set_random_systems(), write_tel_block(), write_tel_offset(), write_tel_offset_w(), and write_tel_pos().

6.6.2.18 void telfil_ (const char * name0)

This function is called when the 'TELFIL' keyword is present in the CORSIKA input file.

```
* The 'file name' parsed is actually decoded further:
*   Apart from the leading '+' or '|' or '+|' the TELFIL argument
*   may contain further bells and whistles:
*   If the supplied file name contains colons, they are assumed to
```



```

*   separate appended numbers with the following meaning:
*   #1: number of events for which the photons per telescope are shown
*   #2: number of events for which energy, direction etc. are shown
*   #3: every so often an event is shown (e.g. 10 -> every tenth event).
*   #4: every so often the event number is shown even if #1 and #2 ran out.
*   #5: offset for #4 (#4=100, #5=1: show events 1, 101, 201, ...)
*   #6: the maximum number of photon bunches before using external storage
*   #7: the maximum size of the output buffer in Megabytes.
*   Example: name = "iact.dat:5:15:10"
*           name becomes "iact.dat"
*           5 events are fully shown
*           15 events have energy etc. shown
*           Every tenth event is shown, i.e. 10,20,30,40,50 are fully shown
*           and events number 60,...,150 have their energies etc. shown.
*           After that every shower with event number divideable by 1000 is shown.
*   Note: No spaces inbetween! CORSIKA input processing truncates at blanks.
*

```

Parameters

<i>name</i>	Output file name. Note: A leading '+' means: use non-compact format A leading ' ' (perhaps after '+') means that the name will not be interpreted as the name of a data file but of a program to which the 'eventio' data stream will be piped (i.e. that program should read the data from its standard input.
-------------	---

Returns

(none)

References INTERNAL_LIMIT.

6.6.2.19 void telinf_ (int * itel, double * x, double * y, double * z, double * r, int * exists)

Parameters

<i>itel</i>	number of telescope in question
<i>x,y,z</i>	telescope position [cm]
<i>r</i>	radius of fiducial volume [cm]
<i>exists</i>	telescope exists

6.6.2.20 void telling_ (int * type, double * data, int * ndim, int * np, int * nthick, double * thickstep)

Write several kinds of vertical distributions to the output. These are kinds of histograms as a function of atmospheric depth. In CORSIKA, these are generally referred to as 'longitudinal' distributions.

```

*   There are three types of distributions:
*   type 1: particle distributions for
*           gammas, positrons, electrons, mu+, mu-,
*           hadrons, all charged, nuclei, Cherenkov photons.
*   type 2: energy distributions (with energies in GeV) for
*           gammas, positrons, electrons, mu+, mu-,
*           hadrons, all charged, nuclei, sum of all.
*   type 3: energy deposits (in GeV) for
*           gammas, e.m. ionisation, cut of e.m. particles,
*           muon ionisation, muon cut, hadron ionisation,
*           hadron cut, neutrinos, sum of all.
*           ('cut' accounting for low-energy particles dropped)
*

```

Note: Corsika can be extracted from CMZ sources with three options concerning the vertical profile of Cherenkov light: default = emission profile, INTCLONG = integrated light profile, NOCLONG = no Cherenkov profiles at all. If you know which kind you are using, you are best off by defining it for compilation of this file (either -DINTEGRATE↵D_LONG_DIST, -DEMISSION_LONG_DIST, or -DNO_LONG_DIST). By default, a run-time detection is attempted which should work well with some 99.99% of all air showers but may fail in some cases like non-interacting muons as primary particles etc.

Parameters

<i>type</i>	see above
<i>data</i>	set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	number of distributions (usually 9)
<i>nthick</i>	number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	step size in g/cm**2

Returns

(none)

References `heigh_()`, and `write_shower_longitudinal()`.6.6.2.21 `void tellni_ (const char * line, int * llength)`

Add a CORSIKA input line to a linked list of strings which will be written to the output file in eventio format right after the run header.

Parameters

<i>line</i>	input line (not terminated)
<i>llength</i>	maximum length of input lines (132 usually)

References `iact_param()`.6.6.2.22 `int telout_ (cors_dbl_t * bsize, cors_dbl_t * wt, cors_dbl_t * px, cors_dbl_t * py, cors_dbl_t * pu, cors_dbl_t * pv, cors_dbl_t * ctime, cors_dbl_t * zem, cors_dbl_t * lambda)`

A bunch of photons from CORSIKA is checked if they hit a telescope and in this case it is stored (in memory). This routine can alternatively trigger that the photon bunch is written by CORSIKA in its usual photons file.

Note that this function should only be called for downward photons as there is no parameter that could indicate upwards photons.

The interface to this function can be modified by defining `EXTENDED_TELOUT`. Doing so requires to have a CORSIKA version with support for the `IACTEXT` option, and to actually activate that option. That could be useful when adding your own code to create some nice graphs or statistics that requires to know the emitting particle and its energy but would be of little help for normal use. Inconsistent usage of `EXTENDED_TELOUT` here and `IACTEXT` in CORSIKA will most likely lead to a crash.

Parameters

<i>bsize</i>	Number of photons (can be fraction of one)
<i>wt</i>	Weight (if thinning option is active)
<i>px</i>	x position in detection level plane
<i>py</i>	y position in detection level plane
<i>pu</i>	x direction cosine
<i>pv</i>	y direction cosine
<i>ctime</i>	arrival time in plane after first interaction
<i>zem</i>	height of emission above sea level
<i>lambda</i>	0. (if wavelength undetermined) or wavelength [nm]. If $\lambda < 0$, photons are already converted to photo-electrons (p.e.), i.e. we have p.e. bunches.
<i>temis</i>	Time of photon emission (only if CORSIKA extracted with <code>IACTEXT</code> option and this code compiled with <code>EXTENDED_TELOUT</code> defined).

<i>penergy</i>	Energy of emitting particle (under conditions as temis).
<i>amass</i>	Mass of emitting particle (under conditions as temis).
<i>charge</i>	Charge of emitting particle (under conditions as temis).

Returns

0 (no output to old-style CORSIKA file needed) 2 (detector hit but no eventio interface available or output should go to CORSIKA file anyway)

References compact_photon_hit(), GRID_SIZE, in_detector(), compact_bunch::lambda, photon_hit(), and compact_bunch::y.

6.6.2.23 void telrne_ (cors_real_t rune[273])

Parameters

<i>rune</i>	CORSIKA run end block
-------------	-----------------------

References fclose(), _struct_IO_BUFFER::output_file, and write_tel_block().

6.6.2.24 void telrnh_ (cors_real_t runh[273])

Get relevant parameters from CORSIKA run header block and write run header block to the data output file.

Parameters

<i>runh</i>	CORSIKA run header block
-------------	--------------------------

Returns

(none)

References allocate_io_buffer(), expand_env(), _struct_IO_BUFFER::extended, fopen(), max_io_buffer, _struct_IO_BUFFER::max_length, _struct_IO_BUFFER::output_file, output_fname, write_input_lines(), and write_tel_block().

6.6.2.25 void telset_ (cors_dbl_t * x, cors_dbl_t * y, cors_dbl_t * z, cors_dbl_t * r)

Set up another telescope for the simulated telescope system. No details of a telescope need to be known except for a fiducial sphere enclosing the relevant optics. Actually, the detector could as well be a non-imaging device.

This function is called for each TELESCOPE keyword in the CORSIKA input file.

Parameters

<i>x</i>	X position [cm]
<i>y</i>	Y position [cm]
<i>z</i>	Z position [cm]
<i>r</i>	radius [cm] within which the telescope is fully contained

Returns

(none)

References MAX_ARRAY_SIZE, ntel, and raise_tel.

6.6.2.26 void telshw_ (void)

This function is called by CORSIKA after the input file is read.

References ntel.

6.6.2.27 void telsmp_(const char * name)

Note that the TELSAMPLE parameter is not processed by CORSIKA itself and thus has to be specified through configuration lines like

```
IACT TELSAMPLE filename
*(IACT) TELSAMPLE filename
```

where the first form requires a CORSIKA patch and the second would work without that patch (but then only with uppercase file names).

6.6.3 Variable Documentation

6.6.3.1 double core_range [static]

6.6.3.2 double core_range1 [static]

6.6.3.3 int corsika_version = (CORSIKA_VERSION)

6.6.3.4 double dmax = 0. [static]

distance of telescopes in (x,y)

6.6.3.5 int impact_correction = 1 [static]

6.6.3.6 int max_internal_bunches = INTERNAL_LIMIT [static]

6.6.3.7 size_t max_io_buffer = MAX_IO_BUFFER [static]

6.6.3.8 char* output_fname [static]

6.6.3.9 double rmax = 0. [static]

radius of telescopes

6.6.3.10 char* sampling_fname [static]

6.6.3.11 double theta_central [static]

6.6.3.12 double xtel[MAX_ARRAY_SIZE] [static]

6.7 iact.h File Reference

Function declarations for CORSIKA IACT interface.

Macros

- #define PRMPAR_SIZE 17

Typedefs

- typedef double cors_dbl_t
*Type for CORSIKA numbers which are currently REAL*8.*
- typedef float cors_real_t
*Type for CORSIKA floating point numbers remaining REAL*4.*

Functions

- void `extprim_setup` (const char *text)
Placeholder function for activating and setting up user-defined (external to CORSIKA) controlled over types, spectra, and angular distribution of primaries.
- void `extprm_` (cors_dbl_t *type, cors_dbl_t *eprim, double *thetap, double *phip)
Placeholder function for external shower-by-shower setting of primary type, energy, and direction.
- void `get_iact_stats` (long *sb, double *ab, double *ap)
- void `get_impact_offset` (cors_real_t evth[273], cors_dbl_t prmpar[17])
- double `iact_rndm` (int dummy)
- void `ioerrorcheck` (void)
- double `refidx_` (double *height)
Index of refraction as a function of altitude [cm].
- double `rhof_` (double *height)
The CORSIKA built-in density lookup function.
- void `telasu_` (int *n, cors_dbl_t *dx, cors_dbl_t *dy)
Setup how many times each shower is used.
- void `telend_` (cors_real_t evte[273])
End of event.
- void `televt_` (cors_real_t evth[273], cors_dbl_t prmpar[17])
- void `telfil_` (const char *name)
Define the output file for photon bunches hitting the telescopes.
- void `telinf_` (int *itel, double *x, double *y, double *z, double *r, int *exists)
Return information about configured telescopes back to CORSIKA.
- void `telling_` (int *type, double *data, int *ndim, int *np, int *nthick, double *thickstep)
Write CORSIKA 'longitudinal' (vertical) distributions.
- void `tellni_` (const char *line, int *llength)
Keep a record of CORSIKA input lines.
- int `telout_` (cors_dbl_t *bsize, cors_dbl_t *wt, cors_dbl_t *px, cors_dbl_t *py, cors_dbl_t *pu, cors_dbl_t *pv, cors_dbl_t *ctime, cors_dbl_t *zem, cors_dbl_t *lambda)
Check if a photon bunch hits one or more simulated detector volumes.
- void `telrne_` (cors_real_t rune[273])
Write run end block to the output file.
- void `telrnh_` (cors_real_t runh[273])
Save aparameters from CORSIKA run header.
- void `telset_` (cors_dbl_t *x, cors_dbl_t *y, cors_dbl_t *z, cors_dbl_t *r)
Add another telescope to the system (array) of telescopes.
- void `telshw_` (void)
Show what telescopes have actually been set up.
- void `telcmp_` (const char *name)
Set the file name with parameters for importance sampling.

6.7.1 Detailed Description

Function declarations are only needed if linking a C or C++ program to the IACT interface. The CORSIKA Fortran code is not using it.

Author

Konrad Bernloehr

Date

CVS \$Date: 2016/09/16 16:04:14 \$

CVS \$Revision: 1.4 \$

6.7.2 Function Documentation

6.7.2.1 void extprim_setup (const char * text)

Parameters

<i>text</i>	CORSIKA input card text following the 'IACT EXTPRIM' keywords. Could be parameter values or a file name.
-------------	--

6.7.2.2 void extprm_ (cors_dbl_t * type, cors_dbl_t * eprim, double * thetap, double * phip)

If primaries are to be generated following some special (non-power-law) spectrum, with a mixed composition, or with an angular distribution not achieved by built-in methods, a user-defined implementation of this function can be used to generate a mixture of primaries of any spectrum, composition, and angular distribution.

Be aware that this function would be called before `televt_` and thus (at least for the first shower) before those run parameters not fitting into the run header are known.

Parameters

<i>type</i>	The type number of the primary to be used in CORSIKA (output).
<i>eprim</i>	The energy [GeV] to be used for the primary (output).
<i>thetap</i>	The zenith angle [rad] of the primary (output).
<i>phip</i>	The azimuth angle [rad] of the primary in the CORSIKA way (direction of movement from magnetic North counter-clockwise) (output).

6.7.2.3 double refidx_ (double * height)

This function can be called from Fortran code as REFIDX(HEIGHT).

Parameters

<i>height</i>	(pointer to) altitude [cm]
---------------	----------------------------

Returns

index of refraction

References `ropol()`.

6.7.2.4 double rhof_ (double * height)

6.7.2.5 void telasu_ (int * n, cors_dbl_t * dx, cors_dbl_t * dy)

Set up how many times the telescope system should be randomly scattered within a given area. Thus each telescope system (array) will see the same shower but at random offsets. Each shower is thus effectively used several times. This function is called according to the CSCAT keyword in the CORSIKA input file.

Parameters

<i>n</i>	The number of telescope systems
<i>dx</i>	Core range radius (if <code>dy==0</code>) or core x range
<i>dy</i>	Core y range (non-zero for rectangular, 0 for circular)

Returns

(none)

6.7.2.6 void telend_ (cors_real_t evte[273])

Write out all recorded photon bunches.

End of an event: write all stored photon bunches to the output data file, and the CORSIKA event end block as well.

Parameters

<i>evte</i>	CORSIKA event end block
-------------	-------------------------

Returns

(none)

References `begin_write_tel_array()`, `end_write_tel_array()`, `shower_extra_parameters::is_set`, `_struct_IO_BUFFER::output_file`, `bunch::photons`, `write_tel_array_end()`, `write_tel_array_head()`, `write_tel_block()`, `write_tel_compact_photons()`, and `write_tel_photons()`.

6.7.2.7 `void telfil_ (const char * name0)`

This function is called when the 'TELFIL' keyword is present in the CORSIKA input file.

```
* The 'file name' parsed is actually decoded further:
*   Apart from the leading '+' or '|' or '+|' the TELFIL argument
*   may contain further bells and whistles:
*   If the supplied file name contains colons, they are assumed to
*   separate appended numbers with the following meaning:
*   #1: number of events for which the photons per telescope are shown
*   #2: number of events for which energy, direction etc. are shown
*   #3: every so often an event is shown (e.g. 10 -> every tenth event).
*   #4: every so often the event number is shown even if #1 and #2 ran out.
*   #5: offset for #4 (#4=100, #5=1: show events 1, 101, 201, ...)
*   #6: the maximum number of photon bunches before using external storage
*   #7: the maximum size of the output buffer in Megabytes.
*   Example: name = "iact.dat:5:15:10"
*   name becomes "iact.dat"
*   5 events are fully shown
*   15 events have energy etc. shown
*   Every tenth event is shown, i.e. 10,20,30,40,50 are fully shown
*   and events number 60,...,150 have their energies etc. shown.
*   After that every shower with event number divideable by 1000 is shown.
*   Note: No spaces inbetween! CORSIKA input processing truncates at blanks.
*
```

Parameters

<i>name</i>	Output file name. Note: A leading '+' means: use non-compact format A leading ' ' (perhaps after '+') means that the name will not be interpreted as the name of a data file but of a program to which the 'eventio' data stream will be piped (i.e. that program should read the data from its standard input).
-------------	--

Returns

(none)

References `INTERNAL_LIMIT`.

6.7.2.8 `void telfil_ (int * itel, double * x, double * y, double * z, double * r, int * exists)`

Parameters

<i>itel</i>	number of telescope in question
<i>x,y,z</i>	telescope position [cm]
<i>r</i>	radius of fiducial volume [cm]
<i>exists</i>	telescope exists

6.7.2.9 `void tellng_ (int * type, double * data, int * ndim, int * np, int * nthick, double * thickstep)`

Write several kinds of vertical distributions to the output. These are kinds of histograms as a function of atmospheric depth. In CORSIKA, these are generally referred to as 'longitudinal' distributions.


```

*   There are three types of distributions:
*       type 1: particle distributions for
*               gammas, positrons, electrons, mu+, mu-,
*               hadrons, all charged, nuclei, Cherenkov photons.
*       type 2: energy distributions (with energies in GeV) for
*               gammas, positrons, electrons, mu+, mu-,
*               hadrons, all charged, nuclei, sum of all.
*       type 3: energy deposits (in GeV) for
*               gammas, e.m. ionisation, cut of e.m. particles,
*               muon ionisation, muon cut, hadron ionisation,
*               hadron cut, neutrinos, sum of all.
*               ('cut' accounting for low-energy particles dropped)
*

```

Note: Corsika can be extracted from CMZ sources with three options concerning the vertical profile of Cherenkov light: default = emission profile, INTCLONG = integrated light profile, NOCLONG = no Cherenkov profiles at all. If you know which kind you are using, you are best off by defining it for compilation of this file (either -DINTEGRATE↵D_LONG_DIST, -DEMISSION_LONG_DIST, or -DNO_LONG_DIST). By default, a run-time detection is attempted which should work well with some 99.99% of all air showers but may fail in some cases like non-interacting muons as primary particles etc.

Parameters

<i>type</i>	see above
<i>data</i>	set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	number of distributions (usually 9)
<i>nthick</i>	number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	step size in g/cm**2

Returns

(none)

References `heigh_()`, and `write_shower_longitudinal()`.

6.7.2.10 void tellni_ (const char * line, int * llength)

Add a CORSIKA input line to a linked list of strings which will be written to the output file in eventio format right after the run header.

Parameters

<i>line</i>	input line (not terminated)
<i>llength</i>	maximum length of input lines (132 usually)

References `iact_param()`.

6.7.2.11 int telout_ (cors_dbl_t * bsize, cors_dbl_t * wt, cors_dbl_t * px, cors_dbl_t * py, cors_dbl_t * pu, cors_dbl_t * pv, cors_dbl_t * ctime, cors_dbl_t * zem, cors_dbl_t * lambda)

A bunch of photons from CORSIKA is checked if they hit a a telescope and in this case it is stored (in memory). This routine can alternatively trigger that the photon bunch is written by CORSIKA in its usual photons file.

Note that this function should only be called for downward photons as there is no parameter that could indicate upwards photons.

The interface to this function can be modified by defining EXTENDED_TELOUT. Doing so requires to have a CO↵RSIKA version with support for the IACTEXT option, and to actually activate that option. That could be useful when adding your own code to create some nice graphs or statistics that requires to know the emitting particle and its energy but would be of little help for normal use. Inconsistent usage of EXTENDED_TELOUT here and IACTEXT in CORSIKA will most likely lead to a crash.

Parameters

<i>bsize</i>	Number of photons (can be fraction of one)
<i>wt</i>	Weight (if thinning option is active)
<i>px</i>	x position in detection level plane
<i>py</i>	y position in detection level plane
<i>pu</i>	x direction cosine
<i>pv</i>	y direction cosine
<i>ctime</i>	arrival time in plane after first interaction
<i>zem</i>	height of emission above sea level
<i>lambda</i>	0. (if wavelength undetermined) or wavelength [nm]. If $\lambda < 0$, photons are already converted to photo-electrons (p.e.), i.e. we have p.e. bunches.
<i>temis</i>	Time of photon emission (only if CORSIKA extracted with IACTEXT option and this code compiled with EXTENDED_TELOUT defined).
<i>penergy</i>	Energy of emitting particle (under conditions as temis).
<i>amass</i>	Mass of emitting particle (under conditions as temis).
<i>charge</i>	Charge of emitting particle (under conditions as temis).

Returns

0 (no output to old-style CORSIKA file needed) 2 (detector hit but no eventio interface available or output should go to CORSIKA file anyway)

References compact_photon_hit(), GRID_SIZE, in_detector(), compact_bunch::lambda, photon_hit(), and compact_bunch::y.

6.7.2.12 void telrne_ (cors_real_t rune[273])

Parameters

<i>rune</i>	CORSIKA run end block
-------------	-----------------------

References fclose(), _struct_IO_BUFFER::output_file, and write_tel_block().

6.7.2.13 void telrnh_ (cors_real_t runh[273])

Get relevant parameters from CORSIKA run header block and write run header block to the data output file.

Parameters

<i>runh</i>	CORSIKA run header block
-------------	--------------------------

Returns

(none)

References allocate_io_buffer(), expand_env(), _struct_IO_BUFFER::extended, fopen(), max_io_buffer, _struct_IO_BUFFER::max_length, _struct_IO_BUFFER::output_file, output_fname, write_input_lines(), and write_tel_block().

6.7.2.14 void telset_ (cors_dbl_t * x, cors_dbl_t * y, cors_dbl_t * z, cors_dbl_t * r)

Set up another telescope for the simulated telescope system. No details of a telescope need to be known except for a fiducial sphere enclosing the relevant optics. Actually, the detector could as well be a non-imaging device.

This function is called for each TELESCOPE keyword in the CORSIKA input file.

Parameters

<i>x</i>	X position [cm]
<i>y</i>	Y position [cm]
<i>z</i>	Z position [cm]
<i>r</i>	radius [cm] within which the telescope is fully contained

Returns

(none)

References MAX_ARRAY_SIZE, ntel, and raise_tel.

6.7.2.15 void telshw_ (void)

This function is called by CORSIKA after the input file is read.

References ntel.

6.7.2.16 void telsmp_ (const char * name)

Note that the TELSAMPLE parameter is not processed by CORSIKA itself and thus has to be specified through configuration lines like

```
IACT TELSAMPLE filename
*(IACT) TELSAMPLE filename
```

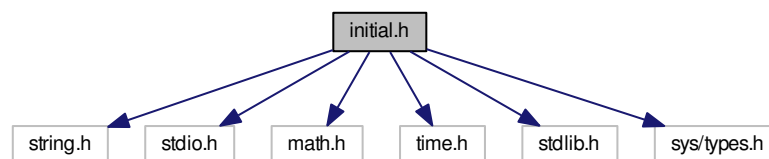
where the first form requires a CORSIKA patch and the second would work without that patch (but then only with uppercase file names).

6.8 initial.h File Reference

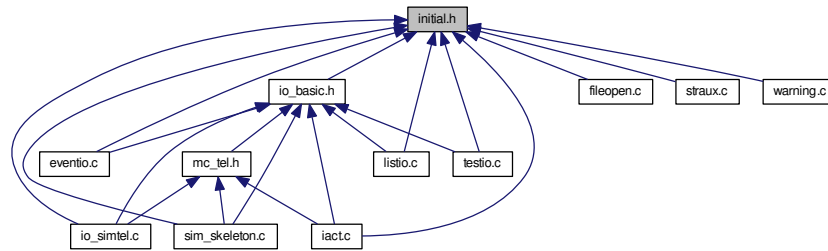
Identification of the system and including some basic include file.

```
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <sys/types.h>
```

Include dependency graph for initial.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define Abs(a) (((a)>=0)?(a):(-1*(a)))`
- `#define APPEND_BINARY "a"`
- `#define APPEND_TEXT "a"`
- `#define ARGLIST(a) a`
- `#define CONST_QUAL`
- `#define IEEE_FLOAT_FORMAT 1`
- `#define M_PI 3.14159265358979323846`
- `#define Max(a, b) ((a)>(b)?(a):(b))`
- `#define max(a, b) ((a)>(b)?(a):(b))`
- `#define Min(a, b) ((a)<(b)?(a):(b))`
- `#define min(a, b) ((a)<(b)?(a):(b))`
- `#define Nint(a) (((a)>=0.)?((long)(a+0.5)):((long)(a-0.5)))`
- `#define READ_BINARY "r"`
- `#define READ_TEXT "r"`
- `#define REGISTER register`
- `#define SEEK_CUR 1`
- `#define WRITE_BINARY "w"`
- `#define WRITE_TEXT "w"`

Typedefs

- `typedef short int16_t`
- `typedef int int32_t`
- `typedef char int8_t`
- `typedef long intmax_t`
- `typedef unsigned short uint16_t`
- `typedef unsigned int uint32_t`
- `typedef unsigned char uint8_t`
- `typedef unsigned long uintmax_t`

6.8.1 Detailed Description

Author

Konrad Bernloehr

Date

1991 to 2010

`$Date: 2016/11/24 13:07:43 $`**Version**`$Revision: 1.19 $`

This file identifies a range of supported operating systems and processor types. As a result, some preprocessor definitions are made. A basic set of system include files (which may vary from one system to another) are included. In addition, compatibility between different systems is improved, for example between K&R compiler systems and ANSI C compilers of various flavours.

Identification of the host operating system (not CPU):

Supported identifiers are

OS_MSDOS

OS_VAXVMS

OS_UNIX

+ variant identifiers like

OS_ULTRIX, OS_LYNX, OS_LINUX, OS_DECUNIX, OS_AIX, OS_HPUX,

OS_DARWIN (Mac OS X).

Note: ULTRIX may be on VAX or MIPS, LINUX on Intel or Alpha,

OS_LYNX on 68K or PowerPC.

OS_OS9

You might first reset all identifiers here.

Then set one or more identifiers according to the system.

Identification of the CPU architecture:

Supported CPU identifiers are

CPU_I86

CPU_X86_64

CPU_VAX

CPU_MIPS

CPU_ALPHA

CPU_68K

CPU_RS6000

CPU_PowerPC

CPU_HPPA

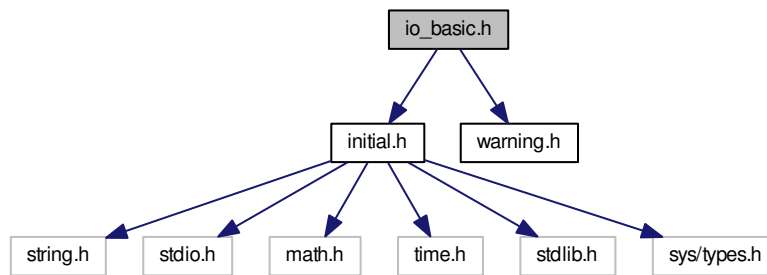
6.9 io_basic.h File Reference

Basic header file for eventio data format.

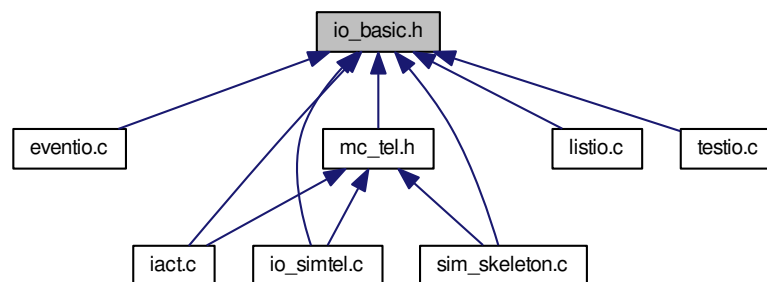
```
#include "initial.h"
```

```
#include "warning.h"
```

Include dependency graph for `io_basic.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [_struct_IO_BUFFER](#)

The `IO_BUFFER` structure contains all data needed to manage the stuff.

- struct [_struct_IO_ITEM_HEADER](#)

An `IO_ITEM_HEADER` is to access header info for an I/O block and as a handle to the I/O buffer.

- struct [ev_reg_entry](#)

Macros

- `#define COPY_BYTES(_target, _source, _num) memcpy(_target, _source, _num)`
- `#define COPY_BYTES_SWAB(_target, _source, _num) swab(_source, _target, _num)`
- `#define EVENTIO_EXTENSION_FLAG 2`
- `#define EVENTIO_USER_FLAG 1`
- `#define get_byte(p) ((p)->r_remaining >= 0 ? *(p)->data++ : -1)`
- `#define get_bytes(p, k) (((p)->r_remaining - k) >= 0 ? (*(p)->data += k) - k : -1)`
- `#define get_vector_of_int16 get_vector_of_short`
- `#define get_vector_of_uint8 get_vector_of_byte`
- `#define HAVE_EVENTIO_EXTENDED_LENGTH 1`
- `#define HAVE_EVENTIO_HEADER_LENGTH 1`

- #define **HAVE_EVENTIO_USER_FLAG** 1
- #define **IO_BUFFER_INITIAL_LENGTH** 32768L
- #define **IO_BUFFER_LENGTH_INCREMENT** 65536L
- #define **IO_BUFFER_MAXIMUM_LENGTH** 3000000L
- #define **MAX_IO_ITEM_LEVEL** 20
- #define **put_byte**(_c, _p)
- #define **put_vector_of_int16** [put_vector_of_short](#)
- #define **put_vector_of_uint8** [put_vector_of_byte](#)

Typedefs

- typedef unsigned char **BYTE**
- typedef struct [ev_reg_entry](#) *(* **EVREGSEARCH**)(unsigned long t)
- typedef struct [_struct_IO_BUFFER](#) **IO_BUFFER**
- typedef struct [_struct_IO_ITEM_HEADER](#) **IO_ITEM_HEADER**
- typedef int(* **IO_USER_FUNCTION**)(unsigned char *, long, int)

Functions

- [IO_BUFFER](#) * [allocate_io_buffer](#) (size_t buflen)
Dynamic allocation of an I/O buffer.
- int [append_io_block_as_item](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header, **BYTE** *_buffer, long length)
Append data from one I/O block into another one.
- int [copy_item_to_io_block](#) ([IO_BUFFER](#) *iobuf2, [IO_BUFFER](#) *iobuf, const [IO_ITEM_HEADER](#) *item_header)
Copy a sub-item to another I/O buffer as top-level item.
- double [dbl_from_sfloat](#) (const uint16_t *snum)
Convert from the internal representation of an OpenGL 16-bit floating point number back to normal floating point representation.
- void [dbl_to_sfloat](#) (double dnum, uint16_t *snum)
Convert a double to the internal representation of a 16 bit floating point number as specified in the OpenGL 3.1 standard, also called a half-float.
- const char * [eventio_registered_description](#) (unsigned long type)
Extract the optional description for a given type number, if available.
- const char * [eventio_registered_typename](#) (unsigned long type)
This functions using the stored function pointer are now in the core eventio code.
- int [extend_io_buffer](#) ([IO_BUFFER](#) *iobuf, unsigned next_byte, long increment)
Extend the dynamically allocated I/O buffer.
- struct [ev_reg_entry](#) * [find_ev_reg](#) (unsigned long t)
This optionally available function is implemented externally.
- int [find_io_block](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header)
Find the beginning of the next I/O data block in the input.
- void [flt_to_sfloat](#) (const float *fnum, uint16_t *snum)
Convert a float to the internal representation of a 16 bit floating point number as specified in the OpenGL 3.1 standard, also called a half-float.
- void [free_io_buffer](#) ([IO_BUFFER](#) *iobuf)
Free an I/O buffer that has been allocated at run-time.
- uintmax_t [get_count](#) ([IO_BUFFER](#) *iobuf)
Get an unsigned integer of unspecified length from an I/O buffer.
- uint16_t [get_count16](#) ([IO_BUFFER](#) *iobuf)

- Get an unsigned 16 bit integer of unspecified length from an I/O buffer.*

 - uint32_t [get_count32](#) (IO_BUFFER *iobuf)
- Get an unsigned 32 bit integer of unspecified length from an I/O buffer.*

 - double [get_double](#) (IO_BUFFER *iobuf)
- Get a double from the I/O buffer.*

 - int32_t [get_int32](#) (IO_BUFFER *iobuf)
- Read a four byte integer from an I/O buffer.*

 - int [get_item_begin](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header)

Begin reading an item.

 - int [get_item_end](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header)

End reading an item.
- long [get_long](#) (IO_BUFFER *iobuf)

Get 4-byte integer from I/O buffer and return as a long int.
- int [get_long_string](#) (char *s, int nmax, IO_BUFFER *iobuf)

Get a long string of ASCII characters from an I/O buffer.
- double [get_real](#) (IO_BUFFER *iobuf)

Get a floating point number (as written by put_real) from the I/O buffer.
- intmax_t [get_scount](#) (IO_BUFFER *iobuf)

Get a signed integer of unspecified length from an I/O buffer.
- int16_t [get_scount16](#) (IO_BUFFER *iobuf)

Shortened version of get_scount for up to 16 bits of data.
- int32_t [get_scount32](#) (IO_BUFFER *iobuf)

Shortened version of get_scount for up to 32 bits of data.
- double [get_sfloat](#) (IO_BUFFER *iobuf)

Get a 16-bit float from an I/O buffer and expand it to a double.
- int [get_short](#) (IO_BUFFER *iobuf)

Get a two-byte integer from an I/O buffer.
- int [get_string](#) (char *s, int nmax, IO_BUFFER *iobuf)

Get a string of ASCII characters from an I/O buffer.
- uint16_t [get_uint16](#) (IO_BUFFER *iobuf)

Get one unsigned short from an I/O buffer.
- uint32_t [get_uint32](#) (IO_BUFFER *iobuf)

Get a four-byte unsigned integer from an I/O buffer.
- int [get_var_string](#) (char *s, int nmax, IO_BUFFER *iobuf)

Get a string of ASCII characters from an I/O buffer.
- void [get_vector_of_byte](#) (BYTE *vec, int num, IO_BUFFER *iobuf)

Get a vector of bytes from an I/O buffer.
- void [get_vector_of_double](#) (double *vec, int num, IO_BUFFER *iobuf)

Get a vector of floating point numbers as 'doubles' from an I/O buffer.
- void [get_vector_of_float](#) (float *vec, int num, IO_BUFFER *iobuf)

Get a vector of floating point numbers as 'floats' from an I/O buffer.
- void [get_vector_of_int](#) (int *vec, int num, IO_BUFFER *iobuf)

Get a vector of (small) integers from I/O buffer.
- void [get_vector_of_int32](#) (int32_t *vec, int num, IO_BUFFER *iobuf)

Get a vector of 32 bit integers from I/O buffer.
- void [get_vector_of_int_scount](#) (int *vec, int num, IO_BUFFER *iobuf)

Get an array of ints as scount32 data from an I/O buffer.
- void [get_vector_of_long](#) (long *vec, int num, IO_BUFFER *iobuf)

Get a vector of 4-byte integers as long int from I/O buffer.
- void [get_vector_of_real](#) (double *vec, int num, IO_BUFFER *iobuf)

Get a vector of floating point numbers as 'doubles' from an I/O buffer.

- void [get_vector_of_short](#) (short *vec, int num, [IO_BUFFER](#) *iobuf)
Get a vector of short integers from I/O buffer.
- void [get_vector_of_uint16](#) (uint16_t *uval, int num, [IO_BUFFER](#) *iobuf)
Get a vector of unsigned shorts from an I/O buffer.
- void [get_vector_of_uint16_scount_differential](#) (uint16_t *vec, int num, [IO_BUFFER](#) *iobuf)
Get an array of uint16_t as differential scout data from an I/O buffer.
- void [get_vector_of_uint32](#) (uint32_t *vec, int num, [IO_BUFFER](#) *iobuf)
Get a vector of 32 bit integers from I/O buffer.
- void [get_vector_of_uint32_scount_differential](#) (uint32_t *vec, int num, [IO_BUFFER](#) *iobuf)
Get an array of uint32_t as differential scout data from an I/O buffer.
- int [list_io_blocks](#) ([IO_BUFFER](#) *iobuf, int verbosity)
Show the top-level item of an I/O block on standard output.
- int [list_sub_items](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header, int maxlevel, int verbosity)
Display the contents of sub-items on standard output.
- long [next_subitem_ident](#) ([IO_BUFFER](#) *iobuf)
Reads the header of a sub-item and return the identifier of it.
- long [next_subitem_length](#) ([IO_BUFFER](#) *iobuf)
Reads the header of a sub-item and return the length of it.
- int [next_subitem_type](#) ([IO_BUFFER](#) *iobuf)
Reads the header of a sub-item and return the type of it.
- void [put_count](#) (uintmax_t num, [IO_BUFFER](#) *iobuf)
Put an unsigned integer of unspecified length to an I/O buffer.
- void [put_count16](#) (uint16_t num, [IO_BUFFER](#) *iobuf)
Shortened version of put_count for up to 16 bits of data.
- void [put_count32](#) (uint32_t num, [IO_BUFFER](#) *iobuf)
Shortened version of put_count for up to 32 bits of data.
- void [put_double](#) (double d, [IO_BUFFER](#) *iobuf)
Put a 'double' as such into an I/O buffer.
- void [put_int32](#) (int32_t num, [IO_BUFFER](#) *iobuf)
Write a four-byte integer to an I/O buffer.
- int [put_item_begin](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header)
Begin putting another (sub-) item into the output buffer.
- int [put_item_begin_with_flags](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header, int user_flag, int extended)
Begin putting another (sub-) item into the output buffer.
- int [put_item_end](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *item_header)
End of putting an item into the output buffer.
- void [put_long](#) (long num, [IO_BUFFER](#) *iobuf)
Put a four-byte integer taken from a 'long' into an I/O buffer.
- int [put_long_string](#) (const char *s, [IO_BUFFER](#) *iobuf)
Put a long string of ASCII characters into an I/O buffer.
- void [put_real](#) (double d, [IO_BUFFER](#) *iobuf)
Put a 4-byte floating point number into an I/O buffer.
- void [put_scount](#) (intmax_t num, [IO_BUFFER](#) *iobuf)
Put a signed integer of unspecified length to an I/O buffer.
- void [put_scount16](#) (int16_t num, [IO_BUFFER](#) *iobuf)
Shorter version of put_scount for up to 16 bytes of data.
- void [put_scount32](#) (int32_t num, [IO_BUFFER](#) *iobuf)
Shorter version of put_scount for up to 32 bytes of data.
- void [put_sfloat](#) (double dnum, [IO_BUFFER](#) *iobuf)
Put a 16-bit float to an I/O buffer.

- void `put_short` (int num, `IO_BUFFER` *iobuf)
Put a two-byte integer on an I/O buffer.
- int `put_string` (const char *s, `IO_BUFFER` *iobuf)
Put a string of ASCII characters into an I/O buffer.
- void `put_uint32` (uint32_t num, `IO_BUFFER` *iobuf)
Put a four-byte integer into an I/O buffer.
- int `put_var_string` (const char *s, `IO_BUFFER` *iobuf)
Put a string of ASCII characters into an I/O buffer.
- void `put_vector_of_byte` (const BYTE *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of bytes into an I/O buffer.
- void `put_vector_of_double` (const double *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of doubles into an I/O buffer.
- void `put_vector_of_float` (const float *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of floats as IEEE 'float' numbers into an I/O buffer.
- void `put_vector_of_int` (const int *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of integers (range -32768 to 32767) into I/O buffer.
- void `put_vector_of_int32` (const int32_t *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 32 bit integers into I/O buffer.
- void `put_vector_of_int_scount` (const int *vec, int num, `IO_BUFFER` *iobuf)
Put an array of ints as scount32 data into an I/O buffer.
- void `put_vector_of_long` (const long *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of long int as 4-byte integers into an I/O buffer.
- void `put_vector_of_real` (const double *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of doubles as IEEE 'float' numbers into an I/O buffer.
- void `put_vector_of_short` (const short *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 2-byte integers on an I/O buffer.
- void `put_vector_of_uint16` (const uint16_t *uval, int num, `IO_BUFFER` *iobuf)
Put a vector of unsigned shorts into an I/O buffer.
- void `put_vector_of_uint16_scount_differential` (uint16_t *vec, int num, `IO_BUFFER` *iobuf)
Put an array of uint16_t as differential scount data into an I/O buffer.
- void `put_vector_of_uint32` (const uint32_t *vec, int num, `IO_BUFFER` *iobuf)
Put a vector of 32 bit integers into I/O buffer.
- void `put_vector_of_uint32_scount_differential` (uint32_t *vec, int num, `IO_BUFFER` *iobuf)
Put an array of uint16_t as differential scount data into an I/O buffer.
- int `read_io_block` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Read the data of an I/O block from the input.
- int `remove_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Remove an item from an I/O buffer.
- int `reset_io_block` (`IO_BUFFER` *iobuf)
Reset an I/O block to its empty status.
- int `rewind_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Go back to the beginning of an item.
- int `search_sub_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header, `IO_ITEM_HEADER` *sub_↵
item_header)
Search for an item of a specified type.
- void `set_eventio_registry_hook` (EVREGSEARCH fptr)
This function should be used to set the find_ev_reg_ptr function pointer.
- int `skip_io_block` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Skip the data of an I/O block from the input.
- int `skip_subitem` (`IO_BUFFER` *iobuf)
When the next sub-item is of no interest, it can be skipped.

- int `unget_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Go back to the beginning of an item being read.
- int `unput_item` (`IO_BUFFER` *iobuf, `IO_ITEM_HEADER` *item_header)
Undo writing at the present level.
- int `write_io_block` (`IO_BUFFER` *iobuf)
Write an I/O block to the block's output.

6.9.1 Detailed Description

Author

Konrad Bernloehr

Date

1991 to 2014

CVS \$Date: 2016/03/14 13:34:52 \$

Version

CVS \$Revision: 1.25 \$

Header file for structures and function prototypes for the basic eventio functions. Not to be used to declare any project-specific structures and prototypes! Declare any such things in 'io_project.h' or in separate header files.

6.9.2 Macro Definition Documentation

6.9.2.1 #define put_byte(_c, _p)

Value:

```
(--(_p)->w_remaining>=0 ? \
  (*( _p )->data++ = (BYTE) (_c)) : \
  (BYTE) extend_io_buffer (_p, (unsigned) (_c), \
    (IO_BUFFER_LENGTH_INCREMENT)))
```

6.9.3 Function Documentation

6.9.3.1 `IO_BUFFER*` `allocate_io_buffer` (`size_t` *buflen*)

Dynamic allocation of an I/O buffer. The actual length of the buffer is passed as an argument. The buffer descriptor is initialized.

Parameters

<i>buflen</i>	The length of the actual buffer in bytes. A safety margin of 4 bytes is added.
---------------	--

Returns

Pointer to I/O buffer or NULL if allocation failed.

References `_struct_IO_BUFFER::aux_count`, `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::extended`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::is_allocated`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_BUFFER::max_length`, `_struct_IO_BUFFER::min_length`, `_struct_IO_BUFFER::output_file`, `_struct_IO_BUFFER::output_fileno`, `_struct_IO_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_BUFFER::sync_err_count`, `_struct_IO_BUFFER::sync_err_max`, `_struct_IO_BUFFER::user_function`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.2 `int append_io_block_as_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, BYTE * buffer, long length)`

Append the data from a complete i/o block as an additional subitem to another i/o block.

Parameters

<i>iobuf</i>	The target I/O buffer descriptor, must be 'opened' for 'writing', i.e. 'put_item_begin()' must be called.
<i>item_header</i>	Item header of the item in iobuf which is currently being filled.
<i>buffer</i>	Data to be filled in. Must be all data from an I/O buffer, including the 4 signature bytes.
<i>length</i>	The length of buffer in bytes.

Returns

0 (o.k.), -1 (error), -2 (not enough memory etc.)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.3 `int copy_item_to_io_block (IO_BUFFER * iobuf2, IO_BUFFER * iobuf, const IO_ITEM_HEADER * item_header)`

Parameters

<i>iobuf2</i>	Target I/O buffer descriptor.
<i>iobuf</i>	Source I/O buffer descriptor.
<i>item_header</i>	Header for the item in iobuf that should be copied to iobuf2.

Returns

0 (o.k.), -1 (error), -2 (not enough memory etc.)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `reset_io_block()`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.4 `void dbl_to_sfloat (double dnum, uint16_t * snum)`

This is done via an intermediate float representation.

Parameters

<i>dnum</i>	The number to be converted.
<i>snum</i>	Pointer for the resulting representation, as stored in an unsigned 16-bit integer (1 bit sign, 5 bits exponent, 10 bits mantissa).

References `fltp_to_sfloat()`.

6.9.3.5 `const char* eventio_registered_typename (unsigned long type)`

This functions using the stored function pointer are now in the core eventio code.

References `find_ev_reg()`, `ev_reg_entry::name`, and `none`.

6.9.3.6 `int extend_io_buffer (IO_BUFFER * iobuf, unsigned next_byte, long increment)`

Extend the dynamically allocated I/O buffer and if an item has been started and the argument 'next_byte' is smaller than 256 that argument will be appended as the next byte to the buffer.

Parameters

<i>iobuf</i>	The I/O buffer descriptor
<i>next_byte</i>	The value of the next byte or ≥ 256
<i>increment</i>	The no. of bytes by which to increase the buffer beyond the current point. If there is remaining space for writing, the buffer is extended by less than 'increment'.

Returns

next_byte (modulo 256) if successful, -1 for failure

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::is_allocated`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::max_length`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.7 `int find_io_block (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Read byte for byte from the input file specified for the I/O buffer and look for the sync-tag (magic number in little-endian or big-endian byte order). As long as the input is properly synchronized this sync-tag should be found in the first four bytes. Otherwise, input data is skipped until the next sync-tag is found. After the sync tag 10 more bytes (item type, version number, and length field) are read. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	An item header structure to be filled in.

Returns

0 (O.k.), -1 (error), or -2 (end-of-file)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `get_item_begin()`, `get_long()`, `get_uint32()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::input_file`, `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::sync_err_count`, `_struct_IO_BUFFER::sync_err_max`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_BUFFER::user_function`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.8 `void fltp_to_sfloat (const float * fnum, uint16_t * snum)`

Both input and output come as pointers to avoid extra conversions.

Parameters

<i>fnum</i>	Pointer to the number to be converted.
<i>snum</i>	Pointer for the resulting representation, as stored in an unsigned 16-bit integer (1 bit sign, 5 bits exponent, 10 bits mantissa).

6.9.3.9 `void free_io_buffer (IO_BUFFER * iobuf)`

Free an I/O buffer that has been allocated at run-time (e.g. by a call to `allocate_io_buf()`).

Parameters

<i>iobuf</i>	The buffer descriptor to be de-allocated.
--------------	---

Returns

(none)

References `_struct_IO_BUFFER::buffer`, and `_struct_IO_BUFFER::is_allocated`.

6.9.3.10 `uintmax_t get_count (IO_BUFFER * iobuf)`

Get an unsigned integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. Even though the scheme in principle allows for arbitrary length data, the current implementation is limited for data of up to 64 bits. On systems with `uintmax_t` shorter than 64 bits, the result could be clipped unnoticed. It could also be clipped unnoticed in the application calling this function.

6.9.3.11 `uint16_t get_count16 (IO_BUFFER * iobuf)`

Get an unsigned 16 bit integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. This is a shorter version of `get_count`, for efficiency reasons.

6.9.3.12 `uint32_t get_count32 (IO_BUFFER * iobuf)`

Get an unsigned 32 bit integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. This is a shorter version of `get_count`, for efficiency reasons.

References `_struct_IO_BUFFER::data`.

6.9.3.13 `double get_double (IO_BUFFER * iobuf)`

Get a double-precision floating point number (as written by `put_double`) from the I/O buffer. The current implementation is only for machines using IEEE format internally.

Parameters

<i>iobuf</i>	– The I/O buffer descriptor;
--------------	------------------------------

Returns

The floating point number.

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.14 `int32_t get_int32 (IO_BUFFER * iobuf)`

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.15 `int get_item_begin (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Reads the header of an item.

Reads the header of an item. If a specific item type is requested but a different type is found and the length of that item is known, the item is skipped.

Parameters

<i>iobuf</i>	The input buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error), -2 (end-of-buffer) or -3 (wrong item type).

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::byte_order`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `get_long()`, `get_uint32()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.16 `int get_item_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Finish reading an item. The pointer in the I/O buffer is at the end of the item after this call, if succesful.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.17 `long get_long (IO_BUFFER * iobuf)`

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see [put_short\(\)](#)).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.18 `int get_long_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a long string of ASCII characters with leading count of bytes from an I/O buffer. Strings can be up to 2^{31-1} bytes long (assuming you have so much memory).

To work properly with strings longer than 32k, a machine with `sizeof(int) > 2` is actually required.

NOTE: the `nmax` count does account also for the trailing zero byte which will be appended.

References `_struct_IO_BUFFER::data`, `get_int32()`, and `get_vector_of_byte()`.

6.9.3.19 `double get_real (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	The I/O buffer descriptor;
--------------	----------------------------

Returns

The floating point number.

References `get_int32()`, and `get_long()`.

6.9.3.20 `intmax_t get_scount (IO_BUFFER * iobuf)`

Get a signed integer of unspecified length from an I/O buffer where it is encoded in a way similar to the UTF-8 character encoding. Even though the scheme in principle allows for arbitrary length data, the current implementation is limited for data of up to 64 bits. On systems with `intmax_t` shorter than 64 bits, the result could be clipped unnoticed.

References `get_count()`.

6.9.3.21 `int get_short (IO_BUFFER * iobuf)`

Get a two-byte integer with least significant byte first. Should be machine-independent (see [put_short\(\)](#)).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.22 `int get_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a string of ASCII characters with leading count of bytes (stored with 16 bits) from an I/O buffer.

NOTE: the *nmax* count does now account for the trailing zero byte which will be appended. This was different in an earlier version of this function where one additional byte had to be available for the trailing zero byte.

References `_struct_IO_BUFFER::data`, `get_short()`, and `get_vector_of_byte()`.

6.9.3.23 `uint16_t get_uint16 (IO_BUFFER * iobuf)`

Get one unsigned short (16-bit unsigned int) from an I/O buffer. The function should be used where sign propagation is of concern.

Parameters

<i>iobuf</i>	The output buffer descriptor.
--------------	-------------------------------

Returns

The value obtained from the I/O buffer.

References `get_vector_of_uint16()`.

6.9.3.24 `uint32_t get_uint32 (IO_BUFFER * iobuf)`

Read a four byte integer with little-endian or big-endian byte order from memory. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.25 `int get_var_string (char * s, int nmax, IO_BUFFER * iobuf)`

Get a string of ASCII characters with leading count of bytes (stored with variable length) from an I/O buffer.

NOTE: the *nmax* count does also account for the trailing zero byte which will be appended.

References `_struct_IO_BUFFER::data`, `get_count()`, and `get_vector_of_byte()`.

6.9.3.26 `void get_vector_of_byte (BYTE * vec, int num, IO_BUFFER * iobuf)`

Parameters

<i>vec</i>	– Byte data vector.
<i>num</i>	– Number of bytes to get.
<i>iobuf</i>	– I/O buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::data`.

6.9.3.27 `void get_vector_of_uint16 (uint16_t * uval, int num, IO_BUFFER * iobuf)`

Get a vector of unsigned shorts from an I/O buffer with least significant byte first. The values are in the range 0 to 65535. The function should be used where sign propagation is of concern.

Parameters

<i>uval</i>	The vector where the values should be loaded.
<i>num</i>	The number of elements to load.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, and `_struct_IO_BUFFER::data`.

6.9.3.28 void get_vector_of_uint16_scount_differential (uint16_t * vec, int num, IO_BUFFER * iobuf)

For optimization reasons it is assumed that the data has been written in a consistent way and is complete. Only minimal tests for remaining data in the buffer are made - while the slower generic version would check for each extracted number.

References `_struct_IO_BUFFER::data`, and `get_scount32()`.

6.9.3.29 void get_vector_of_uint32_scount_differential (uint32_t * vec, int num, IO_BUFFER * iobuf)

For optimization reasons it is assumed that the data has been written in a consistent way and is complete. Only minimal tests for remaining data in the buffer are made - while the slower generic version would check for each extracted number.

References `_struct_IO_BUFFER::data`, and `get_scount32()`.

6.9.3.30 int list_io_blocks (IO_BUFFER * iobuf, int verbosity)

List type, version, ident, and length) of the top item of all I/O blocks in input file onto standard output.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>verbosity</i>	Try showing type name at ≥ 1 , description at ≥ 2 .

Returns

0 (O.k.), -1 (error)

References `_struct_IO_BUFFER::byte_order`, `eventio_registered_description()`, `eventio_registered_typename()`, `find_io_block()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `skip_io_block()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::user_flag`, and `_struct_IO_ITEM_HEADER::version`.

6.9.3.31 int list_sub_items (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, int maxlevel, int verbosity)

Display the contents (item types, versions, idents and lengths) of sub-items on standard output.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of the item from which to show contents.
<i>maxlevel</i>	The maximum nesting depth to show contents (counted from the top-level item on).
<i>verbosity</i>	Try showing type name at ≥ 1 , description at ≥ 2 .

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `eventio_registered_description()`, `eventio_registered_typename()`, `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `list_sub_items()`, `search_sub_item()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, and `_struct_IO_ITEM_HEADER::version`.

6.9.3.32 long next_subitem_ident (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::data`, `get_item_begin()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `unset_item()`.

6.9.3.33 `long next_subitem_length (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::data`, `get_item_begin()`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_ITEM_HEADER::type`, and `unset_item()`.

6.9.3.34 `int next_subitem_type (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	The input buffer descriptor.
--------------	------------------------------

Returns

≥ 0 (O.k.), -1 (error), -2 (end-of-buffer).

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `get_long()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, and `_struct_IO_BUFFER::item_start_offset`.

6.9.3.35 `void put_count (uintmax_t n, IO_BUFFER * iobuf)`

Put an unsigned integer of unspecified length in a way similar to the UTF-8 character encoding to an I/O buffer. The byte order resulting in the buffer is independent of the host byte order or the byte order in action for the I/O buffer, starting with as many leading bits in the first byte as extension bytes needed after the first byte. While the scheme in principle allows for values of arbitrary length, the implementation is limited to 64 bits.

Parameters

<i>n</i>	The number to be saved. Even on systems with 64-bit integers, this must not exceed $2^{32}-1$ with the current implementation.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `put_vector_of_byte()`.

6.9.3.36 `void put_count16 (uint16_t n, IO_BUFFER * iobuf)`

Returns

(none)

References `put_vector_of_byte()`.

6.9.3.37 void put_count32 (uint32_t *n*, IO_BUFFER * *iobuf*)

Returns

(none)

References put_vector_of_byte().

6.9.3.38 void put_double (double *dnum*, IO_BUFFER * *iobuf*)

Put a 'double' (floating point) number in a specific but (almost) machine-independent format into an I/O buffer. This implementation requires the machine to use IEEE double-precision floating point numbers. Only byte order conversion is done.

Parameters

<i>dnum</i>	The number to be put into the I/O buffer.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

(none)

References _struct_IO_BUFFER::byte_order, _struct_IO_BUFFER::data, extend_io_buffer(), and _struct_IO_BUFFER::w_remaining.

6.9.3.39 void put_int32 (int32_t *num*, IO_BUFFER * *iobuf*)

Write a four-byte integer with least significant bytes first. Should be machine independent (see [put_short\(\)](#)).

References _struct_IO_BUFFER::byte_order, _struct_IO_BUFFER::data, extend_io_buffer(), and _struct_IO_BUFFER::w_remaining.

6.9.3.40 int put_item_begin (IO_BUFFER * *iobuf*, IO_ITEM_HEADER * *item_header*)

When putting another item to the output buffer which may be either a top item or a sub-item, [put_item_begin\(\)](#) initializes the buffer (for a top item) and puts the item header on the buffer.

Parameters

<i>iobuf</i>	The output buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.) or -1 (error)

References put_item_begin_with_flags().

6.9.3.41 int put_item_begin_with_flags (IO_BUFFER * *iobuf*, IO_ITEM_HEADER * *item_header*, int *user_flag*, int *extended*)

This is identical to [put_item_begin\(\)](#) except for taking a third and fourth argument, a user flag to be included in the header data, and a flag indicating that the header extension should be used. In [put_item_begin\(\)](#) these flags are forced to 0 (false) for backwards compatibility.

Parameters

<i>iobuf</i>	The output buffer descriptor.
--------------	-------------------------------

<i>item_header</i>	The item header descriptor.
<i>flag</i>	The user flag (0 or 1).

Returns

0 (O.k.) or -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, `_struct_IO_BUFFER::extended`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `put_long()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_ITEM_HEADER::user_flag`, `_struct_IO_ITEM_HEADER::version`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.42 `int put_item_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When finished with putting an item to the output buffer, check for errors and do housekeeping.

Parameters

<i>iobuf</i>	The output buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.) or -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::length`, `_struct_IO_ITEM_HEADER::level`, `put_uint32()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::use_extension`, `_struct_IO_BUFFER::w_remaining`, and `write_io_block()`.

6.9.3.43 `void put_long (long num, IO_BUFFER * iobuf)`

Write a four-byte integer with least significant bytes first. Should be machine independent (see `put_short()`).

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.44 `int put_long_string (const char * s, IO_BUFFER * iobuf)`

Put a long string of ASCII characters with leading count of bytes into an I/O buffer. This is expected to work properly for strings of more than 32k only on machines with `sizeof(int) > 2` because 16-bit machines may not be able to represent lengths of long strings (as obtained with `strlen`).

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_int32()`, `put_short()`, and `put_vector_of_byte()`.

6.9.3.45 `void put_real (double dnum, IO_BUFFER * iobuf)`

Put a 'double' (floating point) number in a specific but (almost) machine-independent format into an I/O buffer. Not the full precision of a 'double' is saved but a 32 bit IEEE floating point number is written (with the same byte ordering

as long integers). On machines with other floating point format than IEEE the input number is converted to a IEEE number first. An optimized (machine- specific) version should compute the output data by shift and add operations rather than by `log()`, divide, and multiply operations on such non-IEEE-format machines (implemented for VAX only).

Parameters

<i>dnum</i>	The number to be put into the I/O buffer.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

(none)

References put_int32(), and put_long().

6.9.3.46 void put_scount (intmax_t *n*, IO_BUFFER * *iobuf*)

Put a signed integer of unspecified length in a way similar to the UTF-8 character encoding to an I/O buffer. The byte order resulting in the buffer is independent of the host byte order or the byte order in action for the I/O buffer, starting with as many leading bits in the first byte as extension bytes needed after the first byte. While the scheme in principle allows for values of arbitrary length, the implementation is limited to 32 bits. To allow an efficient representation of negative numbers, the sign bit is stored in the least significant bit. Portability of data across machines with different intmax_t sizes and the need to represent also the most negative number ($-(2^{31})$, $-(2^{63})$, or $-(2^{127})$, depending on CPU type and compiler) is achieved by putting the number's modulus minus 1 into the higher bits.

Parameters

<i>n</i>	The number to be saved. It can be in the range from $-(2^{63})$ to $2^{63}-1$ on systems with 64 bit integers (intrinsic or through the compiler) and from $-(2^{31})$ to $2^{31}-1$ on pure 32 bit systems.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References put_count().

6.9.3.47 void put_scount16 (int16_t *n*, IO_BUFFER * *iobuf*)

Apart from efficiency, the data can be read with identical results through get_scount16 or get_scount.

Returns

(none)

References put_count().

6.9.3.48 void put_scount32 (int32_t *n*, IO_BUFFER * *iobuf*)

Apart from efficiency, the data can be read with identical results through get_scount32 or get_scount.

Returns

(none)

References put_count().

6.9.3.49 void put_short (int *num*, IO_BUFFER * *iobuf*)

Put a two-byte integer on an I/O buffer with least significant byte first. Should be machine independent as long as 'short' and 'unsigned short' are 16-bit integers, the two's complement is used for negative numbers, and the '>>' operator does a logical shift with unsigned short. Although the 'num' argument is a 4-byte integer on most machines, the value should be in the range -32768 to 32767.

Parameters

<i>num</i>	The number to be saved. Should fit into a short integer and will be truncated otherwise.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.50 int put_string (const char * s, IO_BUFFER * iobuf)

Put a string of ASCII characters with leading count of bytes (stored with 16 bits) into an I/O buffer.

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_short()`, and `put_vector_of_byte()`.

6.9.3.51 void put_uint32 (uint32_t num, IO_BUFFER * iobuf)

Write a four-byte integer with least significant bytes first. Should be machine independent (see [put_short\(\)](#)).

References `_struct_IO_BUFFER::byte_order`, `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.52 int put_var_string (const char * s, IO_BUFFER * iobuf)

Put a string of ASCII characters with leading count of bytes (stored with variable length) into an I/O buffer. Note that storing strings of 32k or more length will not work on systems with `sizeof(int)==2`.

Parameters

<i>s</i>	The null-terminated ASCII string.
<i>iobuf</i>	The I/O buffer descriptor.

Returns

Length of string

References `put_count()`, and `put_vector_of_byte()`.

6.9.3.53 void put_vector_of_byte (const BYTE * vec, int num, IO_BUFFER * iobuf)**Parameters**

<i>vec</i>	Byte data vector.
<i>num</i>	Number of bytes to be put.
<i>iobuf</i>	I/O buffer descriptor.

Returns

(none)

References `_struct_IO_BUFFER::data`, `extend_io_buffer()`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.54 void put_vector_of_double (const double * *dvec*, int *num*, IO_BUFFER * *iobuf*)

Put a vector of 'double' floating point numbers as IEEE 'double' numbers into an I/O buffer.

References put_double().

6.9.3.55 void put_vector_of_int (const int * *vec*, int *num*, IO_BUFFER * *iobuf*)

Put a vector of integers (with actual values in the range -32768 to 32767) into an I/O buffer. This may be relaced by a more efficient but machine-dependent version later.

References put_short().

6.9.3.56 void put_vector_of_short (const short * *vec*, int *num*, IO_BUFFER * *iobuf*)

Put a vector of 2-byte integers on an I/O buffer. This may be relaced by a more efficient but machine-dependent version later. May be called by a number of elements equal to 0. In this case, nothing is done.

References put_short().

6.9.3.57 void put_vector_of_uint16 (const uint16_t * *uval*, int *num*, IO_BUFFER * *iobuf*)

Put a vector of unsigned shorts into an I/O buffer with least significant byte first. The values are in the range 0 to 65535. The function should be used where sign propagation is of concern.

Parameters

<i>uval</i>	The vector of values to be saved.
<i>num</i>	The number of elements to save.
<i>iobuf</i>	The output buffer descriptor.

Returns

(none)

References _struct_IO_BUFFER::byte_order, _struct_IO_BUFFER::data, extend_io_buffer(), and _struct_IO_BUFFER::w_remaining.

6.9.3.58 int read_io_block (IO_BUFFER * *iobuf*, IO_ITEM_HEADER * *item_header*)

This function is called for reading data after an I/O data block has been found (with find_io_block) on input. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error), -2 (end-of-file), -3 (block skipped because it is too large)

References _struct_IO_BUFFER::buffer, _struct_IO_BUFFER::buflen, _struct_IO_BUFFER::data_pending, extend_io_buffer(), _struct_IO_BUFFER::input_file, _struct_IO_BUFFER::input_fileno, _struct_IO_BUFFER::item_extension, _struct_IO_BUFFER::item_length, _struct_IO_BUFFER::item_level, skip_io_block(), _struct_IO_ITEM_HEADER::type, and _struct_IO_BUFFER::user_function.

6.9.3.59 int remove_item (IO_BUFFER * *iobuf*, IO_ITEM_HEADER * *item_header*)

If writing an item has already started and then some condition was found to remove the item again, this is the function for it. The item to be removed should be the last one written, since anything following it will be forgotten too.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item to be removed.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `put_uint32()`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::use_extension`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.60 int reset_io_block (IO_BUFFER * iobuf)**Parameters**

<i>iobuf</i>	The I/O buffer descriptor.
--------------	----------------------------

Returns

0 (O.k.), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::min_length`, `_struct_IO_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.61 int rewind_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)

When reading from an I/O buffer, go back to the beginning of the data area of an item. This is typically used when searching for different types of sub-blocks but processing should not depend on the relative order of them.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.62 int search_sub_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header, IO_ITEM_HEADER * sub_item_header)

Search for an item of a specified type, starting at the current position in the I/O buffer. After successful action the buffer data pointer points to the beginning of the header of the first item of that type. If no such item is found, it points right after the end of the item of the next higher level.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
--------------	----------------------------

<i>item_header</i>	The header of the item within which we search.
<i>sub_item_↵ header</i>	To be filled with what we found.

Returns

0 (O.k., sub-item was found), -1 (error), -2 (no such sub-item), -3 (cannot skip sub-items),

References `_struct_IO_BUFFER::buffer`, `_struct_IO_ITEM_HEADER::can_search`, `_struct_IO_BUFFER::data`, `get_item_begin()`, `get_item_end()`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `↵` `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `↵` `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.63 `void set_eventio_registry_hook (EVREGSEARCH fptr)`

6.9.3.64 `int skip_io_block (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

Skip the data of an I/O block from the input (after the block's header was read). This is the alternative to `read_↵` `io_block()` after having found an I/O block with `find_io_block` but realizing that this is a type of block you don't know how to read or simply not interested in. The type of I/O (raw, buffered, or user-defined) depends on the settings of the I/O block.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
<i>item_header</i>	The item header descriptor.

Returns

0 (O.k.), -1 (error) or -2 (end-of-file)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data_pending`, `_struct_IO_BUFFER::input_file`, `↵` `_struct_IO_BUFFER::input_fileno`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_↵` `_BUFFER::regular`, `_struct_IO_BUFFER::sub_item_length`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_B_↵` `UFFER::user_function`.

6.9.3.65 `int skip_subitem (IO_BUFFER * iobuf)`

Parameters

<i>iobuf</i>	I/O buffer descriptor.
--------------	------------------------

Returns

0 (ok), -1 (error)

References `get_item_begin()`, `get_item_end()`, and `_struct_IO_ITEM_HEADER::type`.

6.9.3.66 `int unget_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When reading from an I/O buffer, go back to the beginning of an item (more precisely: its header) currently being read.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.67 `int unput_item (IO_BUFFER * iobuf, IO_ITEM_HEADER * item_header)`

When writing to an I/O buffer, revert anything yet written at the present level. If the buffer was extended, the last length is kept.

Parameters

<i>iobuf</i>	I/O buffer descriptor.
<i>item_header</i>	Header of item last read.

Returns

0 (ok), -1 (error)

References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::buflen`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::item_start_offset`, `_struct_IO_ITEM_HEADER::level`, `_struct_IO_ITEM_HEADER::use_extension`, and `_struct_IO_BUFFER::w_remaining`.

6.9.3.68 `int write_io_block (IO_BUFFER * iobuf)`

The complete I/O block is written to the output destination, which can be raw I/O (through `write`), buffered I/O (through `fwrite`) or user-defined I/O (through a user function). All items must have been closed before.

Parameters

<i>iobuf</i>	The I/O buffer descriptor.
--------------	----------------------------

Returns

0 (O.k.), -1 (error), -2 (item has no data)

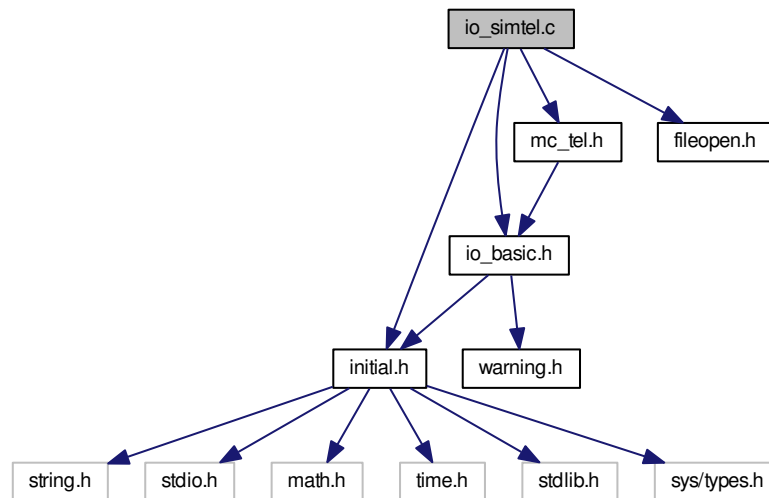
References `_struct_IO_BUFFER::buffer`, `_struct_IO_BUFFER::data`, `_struct_IO_BUFFER::item_extension`, `_struct_IO_BUFFER::item_length`, `_struct_IO_BUFFER::item_level`, `_struct_IO_BUFFER::output_file`, `_struct_IO_BUFFER::output_fileno`, `_struct_IO_BUFFER::user_function`, and `_struct_IO_BUFFER::w_remaining`.

6.10 `io_simtel.c` File Reference

Write and read CORSIKA blocks and simulated Cherenkov photon bunches.

```
#include "initial.h"
#include "io_basic.h"
#include "mc_tel.h"
#include "fileopen.h"
```

Include dependency graph for io_simtel.c:



Functions

- int [begin_read_tel_array](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int *array)
Begin reading data for one array of telescopes/detectors.
- int [begin_write_tel_array](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int array)
Begin writing data for one array of telescopes/detectors.
- int [clear_shower_extra_parameters](#) (struct [shower_extra_parameters](#) *ep)
Similar to [init_shower_extra_parameters\(\)](#) but without any attempts to re-allocate or resize buffers.
- int [end_read_tel_array](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih)
End reading data for one array of telescopes/detectors.
- int [end_write_tel_array](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih)
End writing data for one array of telescopes/detectors.
- struct [shower_extra_parameters](#) * [get_shower_extra_parameters](#) ()
- int [init_shower_extra_parameters](#) (struct [shower_extra_parameters](#) *ep, size_t ni_max, size_t nf_max)
Initialize, resize, clear shower extra parameters.
- int [print_camera_layout](#) (IO_BUFFER *iobuf)
Print the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.
- int [print_photo_electrons](#) (IO_BUFFER *iobuf)
List the the photoelectrons registered in a Cherenkov telescope camera.
- int [print_shower_extra_parameters](#) (IO_BUFFER *iobuf)
- int [print_tel_block](#) (IO_BUFFER *iobuf)
Print a CORSIKA header/trailer block of any type (see [mc_tel.h](#))
- int [print_tel_offset](#) (IO_BUFFER *iobuf)
Print offsets and weights of randomly scattered arrays with respect to shower core.
- int [print_tel_photons](#) (IO_BUFFER *iobuf)
Print bunches of Cherenkov photons for one telescope/detector.
- int [print_tel_pos](#) (IO_BUFFER *iobuf)
Print positions of telescopes/detectors within a system or array.

- int [read_camera_layout](#) (IO_BUFFER *iobuf, int max_pixels, int *itel, int *type, int *pixels, double *xp, double *yp)
Read the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.
- int [read_input_lines](#) (IO_BUFFER *iobuf, struct [linked_string](#) *list)
Read a block with several character strings (normally containing the text of the CORSIKA inputs file) into a linked list.
- int [read_photo_electrons](#) (IO_BUFFER *iobuf, int max_pixels, int max_pe, int *array, int *tel, int *npe, int *pixels, int *flags, int *pe_counts, int *tstart, double *t, double *a, int *photon_counts)
Read the photoelectrons registered in a Cherenkov telescope camera.
- int [read_shower_extra_parameters](#) (IO_BUFFER *iobuf, struct [shower_extra_parameters](#) *ep)
- int [read_shower_longitudinal](#) (IO_BUFFER *iobuf, int *event, int *type, double *data, int ndim, int *np, int *nthick, double *thickstep, int max_np)
Read CORSIKA shower longitudinal distributions.
- int [read_tel_array_end](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int *array)
End reading data for one array of telescopes/detectors.
- int [read_tel_array_head](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int *array)
Begin reading data for one array of telescopes/detectors.
- int [read_tel_block](#) (IO_BUFFER *iobuf, int type, real *data, int maxlen)
Read a CORSIKA header/trailer block of given type (see [mc_tel.h](#))
- int [read_tel_offset](#) (IO_BUFFER *iobuf, int max_array, int *narray, double *toff, double *xoff, double *yoff)
Read offsets of randomly scattered arrays with respect to shower core.
- int [read_tel_offset_w](#) (IO_BUFFER *iobuf, int max_array, int *narray, double *toff, double *xoff, double *yoff, double *weight)
Read offsets and weights of randomly scattered arrays with respect to shower core.
- int [read_tel_photons](#) (IO_BUFFER *iobuf, int max_bunches, int *array, int *tel, double *photons, struct [bunch](#) *bunches, int *nbunches)
Read bunches of Cherenkov photons for one telescope/detector.
- int [read_tel_pos](#) (IO_BUFFER *iobuf, int max_tel, int *ntel, double *x, double *y, double *z, double *r)
Read positions of telescopes/detectors within a system or array.
- int [write_camera_layout](#) (IO_BUFFER *iobuf, int itel, int type, int pixels, double *xp, double *yp)
Write the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.
- int [write_input_lines](#) (IO_BUFFER *iobuf, struct [linked_string](#) *list)
Write a linked list of character strings (normally containing the text of the CORSIKA inputs file) as a dedicated block.
- int [write_photo_electrons](#) (IO_BUFFER *iobuf, int array, int tel, int npe, int flags, int pixels, int *pe_counts, int *tstart, double *t, double *a, int *photon_counts)
Write the photo-electrons registered in a Cherenkov telescope camera.
- int [write_shower_extra_parameters](#) (IO_BUFFER *iobuf, struct [shower_extra_parameters](#) *ep)
- int [write_shower_longitudinal](#) (IO_BUFFER *iobuf, int event, int type, double *data, int ndim, int np, int nthick, double thickstep)
Write CORSIKA shower longitudinal distributions.
- int [write_tel_array_end](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int array)
End writing data for one array of telescopes/detectors.
- int [write_tel_array_head](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int array)
Begin writing data for one array of telescopes/detectors.
- int [write_tel_block](#) (IO_BUFFER *iobuf, int type, int num, real *data, int len)
Write a CORSIKA block as given type number (see [mc_tel.h](#)).
- int [write_tel_compact_photons](#) (IO_BUFFER *iobuf, int array, int tel, double photons, struct [compact_bunch](#) *cbunches, int nbunches, int ext_bunches, char *ext_fname)
Write all the photon bunches for one telescope to an I/O buffer.
- int [write_tel_offset](#) (IO_BUFFER *iobuf, int narray, double toff, double *xoff, double *yoff)
Write offsets of randomly scattered arrays with respect to shower core.
- int [write_tel_offset_w](#) (IO_BUFFER *iobuf, int narray, double toff, double *xoff, double *yoff, double *weight)
Write offsets and weights of randomly scattered arrays with respect to shower core.

- int [write_tel_photons](#) ([IO_BUFFER](#) *iobuf, int array, int tel, double photons, struct [bunch](#) *bunches, int nbunches, int ext_bunches, char *ext_fname)

Write all the photon bunches for one telescope to an I/O buffer.

- int [write_tel_pos](#) ([IO_BUFFER](#) *iobuf, int ntel, double *x, double *y, double *z, double *r)

Write positions of telescopes/detectors within a system or array.

Variables

- static struct
[shower_extra_parameters](#) [private_shower_extra_parameters](#)

There is one global (more precisely: static) block of extra shower parameters as, for example, used in the CORSIKA IACT interface.

6.10.1 Detailed Description

This file provides functions for writing and reading of CORSIKA header and trailer blocks, positions of telescopes/detectors, lists of simulated Cherenkov photon bunches before any detector simulation for the telescopes as well as of photoelectrons after absorption, telescope ray-tracing and quantum efficiency applied.

Author

Konrad Bernloehr

Date

1997 to 2010

CVS \$Date: 2017/03/23 14:32:40 \$

Version

CVS \$Revision: 1.31 \$

6.10.2 Function Documentation

6.10.2.1 int begin_read_tel_array ([IO_BUFFER](#) * iobuf, [IO_ITEM_HEADER](#) * ih, int * array)

Note: this function does not finish reading from the I/O block but after reading of the photons a call to [end_read_tel_array\(\)](#) is needed.

Parameters

<i>iobuf</i>	– I/O buffer descriptor
<i>ih</i>	– I/O item header (for item opened here)
<i>array</i>	– Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References [get_item_begin\(\)](#), [_struct_IO_ITEM_HEADER::ident](#), [_struct_IO_ITEM_HEADER::type](#), and [_struct_IO_ITEM_HEADER::version](#).

6.10.2.2 int begin_write_tel_array ([IO_BUFFER](#) * iobuf, [IO_ITEM_HEADER](#) * ih, int array)

Note: this function does not finish writing to the I/O block but after writing of the photons a call to [end_write_tel_array\(\)](#) is needed.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (for item opened here)
<i>array</i>	Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.3 `int clear_shower_extra_parameters (struct shower_extra_parameters * ep)`

Just clear contents.

Parameters

<i>ep</i>	Pointer to parameter block. A NULL value indicates that the static block is meant.
-----------	--

References `shower_extra_parameters::fparam`, `shower_extra_parameters::id`, `shower_extra_parameters::iparam`, `shower_extra_parameters::is_set`, `shower_extra_parameters::nfparam`, `shower_extra_parameters::niparam`, and `shower_extra_parameters::weight`.

6.10.2.4 `int end_read_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_end()`.

6.10.2.5 `int end_write_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `put_item_end()`.

6.10.2.6 `int init_shower_extra_parameters (struct shower_extra_parameters * ep, size_t ni_max, size_t nf_max)`

Parameters

<i>ep</i>	Pointer to parameter block. A NULL value indicates that the static block is meant.
<i>ni_max</i>	The number of integer parameters to be used.

<i>nf_max</i>	The number of float parameters to be used.
---------------	--

References shower_extra_parameters::fparam, shower_extra_parameters::id, shower_extra_parameters::iparam, shower_extra_parameters::is_set, shower_extra_parameters::nfparam, shower_extra_parameters::niparam, and shower_extra_parameters::weight.

6.10.2.7 int print_camera_layout (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_real(), get_short(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.8 int print_photo_electrons (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), get_short(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.9 int print_tel_block (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.10 int print_tel_offset (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.11 int print_tel_photons (IO_BUFFER * iobuf)

The data format may be either the more or less compact one.

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References bunch::ctime, bunch::cy, get_item_begin(), get_item_end(), get_long(), get_real(), get_short(), bunch::lambda, bunch::photons, compact_bunch::photons, _struct_IO_ITEM_HEADER::type, _struct_IO_ITEM_HEADER::version, bunch::y, and bunch::zem.

6.10.2.12 int print_tel_pos (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), ntel, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.13 int read_camera_layout (IO_BUFFER * iobuf, int max_pixels, int * itel, int * type, int * pixels, double * xp, double * yp)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_pixels</i>	The maximum number of pixels that can be stored in xp, yp.
<i>itel</i>	telescope number
<i>type</i>	camera type (hex/square)
<i>pixels</i>	number of pixels
<i>xp</i>	X positions of pixels
<i>yp</i>	Y position of pixels

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_short(), get_vector_of_real(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.14 int read_input_lines (IO_BUFFER * iobuf, struct linked_string * list)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>list</i>	starting point of linked list (on first call this should be a link to an empty list, i.e. the first element has text=NULL and next=NULL; on additional calls the new lines will be appended.)

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_string(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.10.2.15 int read_photo_electrons (IO_BUFFER * iobuf, int max_pixels, int max_pe, int * array, int * tel, int * npe, int * pixels, int * flags, int * pe_counts, int * tstart, double * t, double * a, int * photon_counts)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_pixels</i>	Maximum number of pixels which can be treated
<i>max_pe</i>	Maximum number of photo-electrons
<i>array</i>	Array number
<i>tel</i>	Telescope number
<i>npe</i>	The total number of photo-electrons read.
<i>pixels</i>	Number of pixels read.
<i>flags</i>	Bit 0: amplitudes available, bit 1: includes NSB p.e.
<i>pe_counts</i>	Numbers of photo-electrons in each pixel
<i>tstart</i>	Offsets in 't' at which data for each pixel starts
<i>t</i>	Time of arrival of photons at the camera.
<i>a</i>	Amplitudes of p.e. signals [mean p.e.] (optional, may be NULL).
<i>photon_counts</i>	Optional number of photons arriving at a pixel.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, `unset_item()`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.16 `int read_shower_longitudinal (IO_BUFFER * iobuf, int * event, int * type, double * data, int ndim, int * np, int * nthick, double * thickstep, int max_np)`

See [telling_\(\)](#) in [iact.c](#) for more detailed parameter description.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>event</i>	return event number
<i>type</i>	return 1 = particle numbers, 2 = energy, 3 = energy deposits
<i>data</i>	return set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	return number of distributions (usually 9)
<i>nthick</i>	return number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	return step size in g/cm**2
<i>max_np</i>	maximum number of distributions for which we have space.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.17 `int read_tel_array_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int * array)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.18 `int read_tel_array_head (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int * array)`

Note: this function does not finish reading from the I/O block but after reading of the photons a call to `end_read_tel_array()` is needed.

Parameters

<i>iobuf</i>	– I/O buffer descriptor
<i>ih</i>	– I/O item header (for item opened here)
<i>array</i>	– Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.19 `int read_tel_block (IO_BUFFER * iobuf, int type, real * data, int maxlen)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>type</i>	block type (see mc_tel.h)
<i>data</i>	area for data to be read
<i>maxlen</i>	maximum number of elements to be read

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `_struct_IO_ITEM_HEADER::type`, `ev_reg_entry::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.20 `int read_tel_offset (IO_BUFFER * iobuf, int max_array, int * narray, double * toff, double * xoff, double * yoff)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_array</i>	Maximum number of arrays that can be treated
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `read_tel_offset_w()`.

6.10.2.21 `int read_tel_offset_w (IO_BUFFER * iobuf, int max_array, int * narray, double * toff, double * xoff, double * yoff, double * weight)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_array</i>	Maximum number of arrays that can be treated
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays
<i>weight</i>	Area weight for uniform or importance sampled core offset. For old version data (uniformly sampled), 0.0 is returned.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.22 `int read_tel_photons (IO_BUFFER * iobuf, int max_bunches, int * array, int * tel, double * photons, struct bunch * bunches, int * nbunches)`

The data format may be either the more or less compact one.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_bunches</i>	maximum number of bunches that can be treated
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>bunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `bunch::ctime`, `bunch::cy`, `compact_bunch::cy`, `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `bunch::lambda`, `bunch::photons`, `_struct_IO_ITEM_HEADER::type`, `unget_item()`, `_struct_IO_ITEM_HEADER::version`, `bunch::y`, and `bunch::zem`.

6.10.2.23 `int read_tel_pos (IO_BUFFER * iobuf, int max_tel, int * ntel, double * x, double * y, double * z, double * r)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_tel</i>	maximum number of telescopes allowed
<i>ntel</i>	number of telescopes/detectors
<i>x</i>	X positions
<i>y</i>	Y positions
<i>z</i>	Z positions
<i>r</i>	radius of spheres including the whole devices

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.24 `int write_camera_layout (IO_BUFFER * iobuf, int itel, int type, int pixels, double * xp, double * yp)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>itel</i>	telescope number
<i>type</i>	camera type (hex/square)
<i>pixels</i>	number of pixels
<i>xp</i>	X positions of pixels
<i>yp</i>	Y position of pixels

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_short()`, `put_vector_of_↵`
`real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.25 `int write_input_lines (IO_BUFFER * iobuf, struct linked_string * list)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>list</i>	starting point of linked list

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_string()`, `_struct_↵`
`_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.26 `int write_photo_electrons (IO_BUFFER * iobuf, int array, int tel, int npe, int flags, int pixels, int * pe_counts, int
 * tstart, double * t, double * a, int * photon_counts)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>npe</i>	Total number of photo-electrons in the camera.
<i>pixels</i>	No. of pixels to be written
<i>flags</i>	Bit 0: amplitudes available, bit 1: includes NSB p.e., bit 2: also including no. of photons hitting each pixel.
<i>pe_counts</i>	Numbers of photo-electrons in each pixel
<i>tstart</i>	Offsets in 't' at which data for each pixel starts
<i>t</i>	Time of arrival of photons at the camera.
<i>a</i>	Amplitudes of p.e. signals [mean p.e.] (optional, may be NULL).
<i>photon_counts</i>	Optional number of photons arriving at a pixel (with flags bit 2 set)

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_short()`, `put_↵`
`vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.27 `int write_shower_longitudinal (IO_BUFFER * iobuf, int event, int type, double * data, int ndim, int np, int nthick,
 double thickstep)`

See [telling_\(\)](#) in [iact.c](#) for more detailed parameter description.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>event</i>	event number
<i>type</i>	1 = particle numbers, 2 = energy, 3 = energy deposits
<i>data</i>	set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	number of distributions (usually 9)
<i>nthick</i>	number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	step size in g/cm**2

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_↵short()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.28 `int write_tel_array_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int array)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `_struct_IO_ITEM_HEADER↵::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.29 `int write_tel_array_head (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int array)`

Note: this function does not finish writing to the I/O block but after writing of the photons a call to [end_write_tel_↵array\(\)](#) is needed.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (for item opened here)
<i>array</i>	Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `_struct_IO_ITEM_HEADER↵::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.30 `int write_tel_block (IO_BUFFER * iobuf, int type, int num, real * data, int len)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>type</i>	block type (see mc_tel.h)

<i>num</i>	Run or event number depending on type
<i>data</i>	Data as passed from CORSIKA
<i>len</i>	Number of elements to be written

Returns

0 (OK), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `_struct_IO_ITEM_HEADER::type`, `ev_reg_entry::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.31 `int write_tel_compact_photons (IO_BUFFER * iobuf, int array, int tel, double photons, struct compact_bunch * cbunches, int nbunches, int ext_bunches, char * ext_fname)`

Usually, calls to this function for each telescope in an array should be enclosed within calls to [begin_write_tel_array\(\)](#) and [end_write_tel_array\(\)](#). This routine writes the more compact format (16 bytes per bunch). The more compact format should usually be used to save memory and disk space.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>cbunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list
<i>ext_bunches</i>	number of elements in external file
<i>ext_fname</i>	name of external (temporary) file

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `compact_bunch::ctime`, `compact_bunch::cy`, `fclose()`, `fopen()`, `_struct_IO_ITEM_HEADER::ident`, `compact_bunch::lambda`, `compact_bunch::log_zem`, `compact_bunch::photons`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_short()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::version`, and `compact_bunch::y`.

6.10.2.32 `int write_tel_offset (IO_BUFFER * iobuf, int narray, double toff, double * xoff, double * yoff)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `write_tel_offset_w()`.

6.10.2.33 `int write_tel_offset_w (IO_BUFFER * iobuf, int narray, double toff, double * xoff, double * yoff, double * weight)`

With respect to the backwards-compatible non-weights version [write_tel_offset\(\)](#), this version adds a weight to each offset position which should be normalized in such a way that with uniform sampling it should be the area over which showers are thrown divided by the number of array in each shower. With importance sampling the same relation

should hold on average. So in either case, the average sum of weights for the different offsets in one shower equals just the area over which cores are randomized. This leaves the possibility to change the number of offsets from shower to shower.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays
<i>weight</i>	Area weight for uniform or importance sampled core offset.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.2.34 `int write_tel_photons (IO_BUFFER * iobuf, int array, int tel, double photons, struct bunch * bunches, int nbunches, int ext_bunches, char * ext_fname)`

Usually, calls to this function for each telescope in an array should be enclosed within calls to `begin_write_tel_array()` and `end_write_tel_array()`. This routine writes the less compact format (32 bytes per bunch).

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>bunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list
<i>ext_bunches</i>	number of elements in external file
<i>ext_fname</i>	name of external (temporary) file

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `bunch::ctime`, `bunch::cy`, `fclose()`, `fileopen()`, `_struct_IO_ITEM_HEADER::ident`, `bunch::lambda`, `bunch::photons`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_short()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::version`, `bunch::y`, and `bunch::zem`.

6.10.2.35 `int write_tel_pos (IO_BUFFER * iobuf, int ntel, double * x, double * y, double * z, double * r)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ntel</i>	number of telescopes/detectors
<i>x</i>	X positions
<i>y</i>	Y positions
<i>z</i>	Z positions
<i>r</i>	radius of spheres including the whole devices

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.10.3 Variable Documentation

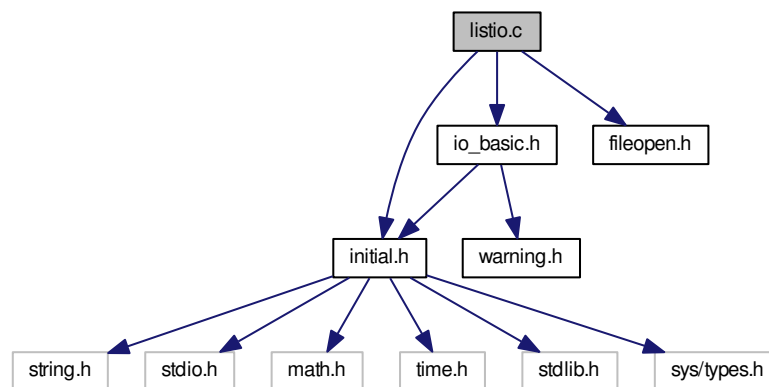
6.10.3.1 struct shower_extra_parameters private_shower_extra_parameters [static]

Get a pointer to this block.

6.11 listio.c File Reference

Main function for listing data consisting of eventio blocks.

```
#include "initial.h"
#include "io_basic.h"
#include "fileopen.h"
Include dependency graph for listio.c:
```



Functions

- int [main](#) (int argc, char **argv)
Main function.

6.11.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2014/06/01 11:33:05 \$

Version

CVS \$Revision: 1.14 \$

The item type, version, length and ident are displayed. With command line option '-s' all sub-items are shown as well. Input is from standard input by default, output to standard output.

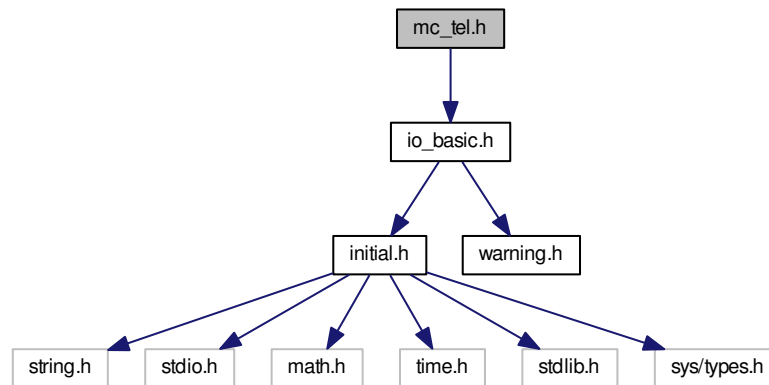
```
Syntax: listio [-s[n]] [-p] [filename]
List structure of eventio data files.
-s : also list contained (sub-) items
-sn: list sub-items up to depth n (n=0,1,...)
-p : show positions of items in the file
If no file name given, standard input is used.
```

6.12 mc_tel.h File Reference

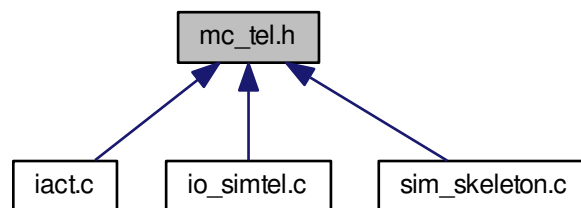
Definitions and structures for CORSIKA Cherenkov light interface.

```
#include "io_basic.h"
```

Include dependency graph for mc_tel.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [bunch](#)
Photons collected in bunches of identical direction, position, time, and wavelength.
- struct [compact_bunch](#)
The [compact_bunch](#) struct is equivalent to the [bunch](#) struct except that we try to use less memory.
- struct [linked_string](#)

The *linked_string* is mainly used to keep CORSIKA input.

- struct [photo_electron](#)

A photo-electron produced by a photon hitting a pixel.

- struct [shower_extra_parameters](#)

Extra shower parameters of unspecified nature.

Macros

- `#define MC_TEL_LOADED 2`
- `#define IO_TYPE_MC_BASE 1200`
- `#define IO_TYPE_MC_EVTE (IO_TYPE_MC_BASE+9)`
- `#define IO_TYPE_MC_EVTH (IO_TYPE_MC_BASE+2)`
- `#define IO_TYPE_MC_EXTRA_PARAM (IO_TYPE_MC_BASE+15)`
- `#define IO_TYPE_MC_INPUTCFG (IO_TYPE_MC_BASE+12)`
- `#define IO_TYPE_MC_LAYOUT (IO_TYPE_MC_BASE+6)`
- `#define IO_TYPE_MC_LONGI (IO_TYPE_MC_BASE+11)`
- `#define IO_TYPE_MC_PE (IO_TYPE_MC_BASE+8)`
- `#define IO_TYPE_MC_PHOTONS (IO_TYPE_MC_BASE+5)`
- `#define IO_TYPE_MC_RUNE (IO_TYPE_MC_BASE+10)`
- `#define IO_TYPE_MC_RUNH (IO_TYPE_MC_BASE+0)`
- `#define IO_TYPE_MC_TELARRAY (IO_TYPE_MC_BASE+4)`
- `#define IO_TYPE_MC_TELARRAY_END (IO_TYPE_MC_BASE+14)`
- `#define IO_TYPE_MC_TELARRAY_HEAD (IO_TYPE_MC_BASE+13)`
- `#define IO_TYPE_MC_TELOFF (IO_TYPE_MC_BASE+3)`
- `#define IO_TYPE_MC_TELPOS (IO_TYPE_MC_BASE+1)`
- `#define IO_TYPE_MC_TRIGTIME (IO_TYPE_MC_BASE+7)`

Typedefs

- typedef short **INT16**
- typedef int **INT32**
- typedef float **real**
- typedef unsigned short **UINT16**
- typedef unsigned int **UINT32**

Functions

- int [begin_read_tel_array](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *ih, int *array)
Begin reading data for one array of telescopes/detectors.
- int [begin_write_tel_array](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *ih, int array)
Begin writing data for one array of telescopes/detectors.
- int [clear_shower_extra_parameters](#) (struct [shower_extra_parameters](#) *ep)
Similar to [init_shower_extra_parameters\(\)](#) but without any attempts to re-allocate or resize buffers.
- int [end_read_tel_array](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *ih)
End reading data for one array of telescopes/detectors.
- int [end_write_tel_array](#) ([IO_BUFFER](#) *iobuf, [IO_ITEM_HEADER](#) *ih)
End writing data for one array of telescopes/detectors.
- struct [shower_extra_parameters](#) * [get_shower_extra_parameters](#) (void)
- int [init_shower_extra_parameters](#) (struct [shower_extra_parameters](#) *ep, size_t ni_max, size_t nf_max)
Initialize, resize, clear shower extra parameters.
- int [print_camera_layout](#) ([IO_BUFFER](#) *iobuf)
Print the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.

- int [print_photo_electrons](#) (IO_BUFFER *iobuf)
List the photoelectrons registered in a Cherenkov telescope camera.
- int [print_shower_extra_parameters](#) (IO_BUFFER *iobuf)
- int [print_tel_block](#) (IO_BUFFER *iobuf)
Print a CORSIKA header/trailer block of any type (see [mc_tel.h](#))
- int [print_tel_offset](#) (IO_BUFFER *iobuf)
Print offsets and weights of randomly scattered arrays with respect to shower core.
- int [print_tel_photons](#) (IO_BUFFER *iobuf)
Print bunches of Cherenkov photons for one telescope/detector.
- int [print_tel_pos](#) (IO_BUFFER *iobuf)
Print positions of telescopes/detectors within a system or array.
- int [read_camera_layout](#) (IO_BUFFER *iobuf, int max_pixels, int *itel, int *type, int *pixels, double *xp, double *yp)
Read the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.
- int [read_input_lines](#) (IO_BUFFER *iobuf, struct [linked_string](#) *list)
Read a block with several character strings (normally containing the text of the CORSIKA inputs file) into a linked list.
- int [read_photo_electrons](#) (IO_BUFFER *iobuf, int max_pixel, int max_pe, int *array, int *tel, int *npe, int *pixels, int *flags, int *pe_counts, int *tstart, double *t, double *a, int *photon_counts)
Read the photoelectrons registered in a Cherenkov telescope camera.
- int [read_shower_extra_parameters](#) (IO_BUFFER *iobuf, struct [shower_extra_parameters](#) *ep)
- int [read_shower_longitudinal](#) (IO_BUFFER *iobuf, int *event, int *type, double *data, int ndim, int *np, int *nthick, double *thickstep, int max_np)
Read CORSIKA shower longitudinal distributions.
- int [read_tel_array_end](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int *array)
End reading data for one array of telescopes/detectors.
- int [read_tel_array_head](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int *array)
Begin reading data for one array of telescopes/detectors.
- int [read_tel_block](#) (IO_BUFFER *iobuf, int type, real *data, int maxlen)
Read a CORSIKA header/trailer block of given type (see [mc_tel.h](#))
- int [read_tel_offset](#) (IO_BUFFER *iobuf, int max_array, int *narray, double *toff, double *xoff, double *yoff)
Read offsets of randomly scattered arrays with respect to shower core.
- int [read_tel_offset_w](#) (IO_BUFFER *iobuf, int max_array, int *narray, double *toff, double *xoff, double *yoff, double *weight)
Read offsets and weights of randomly scattered arrays with respect to shower core.
- int [read_tel_photons](#) (IO_BUFFER *iobuf, int max_bunches, int *array, int *tel, double *photons, struct [bunch](#) *bunches, int *nbunches)
Read bunches of Cherenkov photons for one telescope/detector.
- int [read_tel_pos](#) (IO_BUFFER *iobuf, int max_tel, int *ntel, double *x, double *y, double *z, double *r)
Read positions of telescopes/detectors within a system or array.
- int [write_camera_layout](#) (IO_BUFFER *iobuf, int itel, int type, int pixels, double *xp, double *yp)
Write the layout (pixel positions) of a camera used for converting from photons to photo-electrons in a pixel.
- int [write_input_lines](#) (IO_BUFFER *iobuf, struct [linked_string](#) *list)
Write a linked list of character strings (normally containing the text of the CORSIKA inputs file) as a dedicated block.
- int [write_photo_electrons](#) (IO_BUFFER *iobuf, int array, int tel, int npe, int pixels, int flags, int *pe_counts, int *tstart, double *t, double *a, int *photon_counts)
Write the photo-electrons registered in a Cherenkov telescope camera.
- int [write_shower_extra_parameters](#) (IO_BUFFER *iobuf, struct [shower_extra_parameters](#) *ep)
- int [write_shower_longitudinal](#) (IO_BUFFER *iobuf, int event, int type, double *data, int ndim, int np, int nthick, double thickstep)
Write CORSIKA shower longitudinal distributions.
- int [write_tel_array_end](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int array)
End writing data for one array of telescopes/detectors.

- int [write_tel_array_head](#) (IO_BUFFER *iobuf, IO_ITEM_HEADER *ih, int array)
Begin writing data for one array of telescopes/detectors.
- int [write_tel_block](#) (IO_BUFFER *iobuf, int type, int num, real *data, int len)
Write a CORSIKA block as given type number (see [mc_tel.h](#)).
- int [write_tel_compact_photons](#) (IO_BUFFER *iobuf, int array, int tel, double photons, struct [compact_bunch](#) *cbunches, int nbunches, int ext_bunches, char *ext_fname)
Write all the photon bunches for one telescope to an I/O buffer.
- int [write_tel_offset](#) (IO_BUFFER *iobuf, int narray, double toff, double *xoff, double *yoff)
Write offsets of randomly scattered arrays with respect to shower core.
- int [write_tel_offset_w](#) (IO_BUFFER *iobuf, int narray, double toff, double *xoff, double *yoff, double *weight)
Write offsets and weights of randomly scattered arrays with respect to shower core.
- int [write_tel_photons](#) (IO_BUFFER *iobuf, int array, int tel, double photons, struct [bunch](#) *bunches, int nbunches, int ext_bunches, char *ext_fname)
Write all the photon bunches for one telescope to an I/O buffer.
- int [write_tel_pos](#) (IO_BUFFER *iobuf, int ntel, double *x, double *y, double *z, double *r)
Write positions of telescopes/detectors within a system or array.

6.12.1 Detailed Description

This file contains definitions of data structures and of function prototypes as needed for the Cherenkov light extraction interfaced to the modified CORSIKA code.

Author

Konrad Bernloehr

Date

1997 to 2010

CVS \$Date: 2016/03/08 16:07:50 \$

Version

CVS \$Revision: 1.16 \$

6.12.2 Function Documentation

6.12.2.1 int begin_read_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int * array)

Note: this function does not finish reading from the I/O block but after reading of the photons a call to [end_read_tel_array\(\)](#) is needed.

Parameters

<i>iobuf</i>	– I/O buffer descriptor
<i>ih</i>	– I/O item header (for item opened here)
<i>array</i>	– Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References [get_item_begin\(\)](#), [_struct_IO_ITEM_HEADER::ident](#), [_struct_IO_ITEM_HEADER::type](#), and [_struct_IO_ITEM_HEADER::version](#).

6.12.2.2 int begin_write_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int array)

Note: this function does not finish writing to the I/O block but after writing of the photons a call to [end_write_tel_array\(\)](#) is needed.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (for item opened here)
<i>array</i>	Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.3 `int clear_shower_extra_parameters (struct shower_extra_parameters * ep)`

Just clear contents.

Parameters

<i>ep</i>	Pointer to parameter block. A NULL value indicates that the static block is meant.
-----------	--

References `shower_extra_parameters::fparam`, `shower_extra_parameters::id`, `shower_extra_parameters::iparam`, `shower_extra_parameters::is_set`, `shower_extra_parameters::nfparam`, `shower_extra_parameters::niparam`, and `shower_extra_parameters::weight`.

6.12.2.4 `int end_read_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_end()`.

6.12.2.5 `int end_write_tel_array (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `put_item_end()`.

6.12.2.6 `int init_shower_extra_parameters (struct shower_extra_parameters * ep, size_t ni_max, size_t nf_max)`

Parameters

<i>ep</i>	Pointer to parameter block. A NULL value indicates that the static block is meant.
<i>ni_max</i>	The number of integer parameters to be used.

<i>nf_max</i>	The number of float parameters to be used.
---------------	--

References shower_extra_parameters::fparam, shower_extra_parameters::id, shower_extra_parameters::iparam, shower_extra_parameters::is_set, shower_extra_parameters::nfparam, shower_extra_parameters::niparam, and shower_extra_parameters::weight.

6.12.2.7 int print_camera_layout (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_real(), get_short(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.8 int print_photo_electrons (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), get_short(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.9 int print_tel_block (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.10 int print_tel_offset (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.11 int print_tel_photons (IO_BUFFER * iobuf)

The data format may be either the more or less compact one.

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References bunch::ctime, bunch::cy, get_item_begin(), get_item_end(), get_long(), get_real(), get_short(), bunch::lambda, bunch::photons, compact_bunch::photons, _struct_IO_ITEM_HEADER::type, _struct_IO_ITEM_HEADER::version, bunch::y, and bunch::zem.

6.12.2.12 int print_tel_pos (IO_BUFFER * iobuf)

Parameters

<i>iobuf</i>	I/O buffer descriptor
--------------	-----------------------

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_real(), ntel, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.13 int read_camera_layout (IO_BUFFER * iobuf, int max_pixels, int * itel, int * type, int * pixels, double * xp, double * yp)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_pixels</i>	The maximum number of pixels that can be stored in xp, yp.
<i>itel</i>	telescope number
<i>type</i>	camera type (hex/square)
<i>pixels</i>	number of pixels
<i>xp</i>	X positions of pixels
<i>yp</i>	Y position of pixels

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_short(), get_vector_of_real(), _struct_IO_ITEM_HEADER::ident, _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.14 int read_input_lines (IO_BUFFER * iobuf, struct linked_string * list)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>list</i>	starting point of linked list (on first call this should be a link to an empty list, i.e. the first element has text=NULL and next=NULL; on additional calls the new lines will be appended.)

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References get_item_begin(), get_item_end(), get_long(), get_string(), _struct_IO_ITEM_HEADER::type, and _struct_IO_ITEM_HEADER::version.

6.12.2.15 int read_photo_electrons (IO_BUFFER * iobuf, int max_pixels, int max_pe, int * array, int * tel, int * npe, int * pixels, int * flags, int * pe_counts, int * tstart, double * t, double * a, int * photon_counts)

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_pixels</i>	Maximum number of pixels which can be treated
<i>max_pe</i>	Maximum number of photo-electrons
<i>array</i>	Array number
<i>tel</i>	Telescope number
<i>npe</i>	The total number of photo-electrons read.
<i>pixels</i>	Number of pixels read.
<i>flags</i>	Bit 0: amplitudes available, bit 1: includes NSB p.e.
<i>pe_counts</i>	Numbers of photo-electrons in each pixel
<i>tstart</i>	Offsets in 't' at which data for each pixel starts
<i>t</i>	Time of arrival of photons at the camera.
<i>a</i>	Amplitudes of p.e. signals [mean p.e.] (optional, may be NULL).
<i>photon_counts</i>	Optional number of photons arriving at a pixel.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, `unget_item()`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.16 `int read_shower_longitudinal (IO_BUFFER * iobuf, int * event, int * type, double * data, int ndim, int * np, int * nthick, double * thickstep, int max_np)`

See [telling_\(\)](#) in [iact.c](#) for more detailed parameter description.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>event</i>	return event number
<i>type</i>	return 1 = particle numbers, 2 = energy, 3 = energy deposits
<i>data</i>	return set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	return number of distributions (usually 9)
<i>nthick</i>	return number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	return step size in g/cm**2
<i>max_np</i>	maximum number of distributions for which we have space.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.17 `int read_tel_array_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int * array)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.18 `int read_tel_array_head (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int * array)`

Note: this function does not finish reading from the I/O block but after reading of the photons a call to `end_read_tel_array()` is needed.

Parameters

<i>iobuf</i>	– I/O buffer descriptor
<i>ih</i>	– I/O item header (for item opened here)
<i>array</i>	– Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `_struct_IO_ITEM_HEADER::ident`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.19 `int read_tel_block (IO_BUFFER * iobuf, int type, real * data, int maxlen)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>type</i>	block type (see mc_tel.h)
<i>data</i>	area for data to be read
<i>maxlen</i>	maximum number of elements to be read

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `_struct_IO_ITEM_HEADER::type`, `ev_reg_entry::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.20 `int read_tel_offset (IO_BUFFER * iobuf, int max_array, int * narray, double * toff, double * xoff, double * yoff)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_array</i>	Maximum number of arrays that can be treated
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `read_tel_offset_w()`.

6.12.2.21 `int read_tel_offset_w (IO_BUFFER * iobuf, int max_array, int * narray, double * toff, double * xoff, double * yoff, double * weight)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_array</i>	Maximum number of arrays that can be treated
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays
<i>weight</i>	Area weight for uniform or importance sampled core offset. For old version data (uniformly sampled), 0.0 is returned.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.22 `int read_tel_photons (IO_BUFFER * iobuf, int max_bunches, int * array, int * tel, double * photons, struct bunch * bunches, int * nbunches)`

The data format may be either the more or less compact one.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_bunches</i>	maximum number of bunches that can be treated
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>bunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `bunch::ctime`, `bunch::cy`, `compact_bunch::cy`, `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_short()`, `bunch::lambda`, `bunch::photons`, `_struct_IO_ITEM_HEADER::type`, `unget_item()`, `_struct_IO_ITEM_HEADER::version`, `bunch::y`, and `bunch::zem`.

6.12.2.23 `int read_tel_pos (IO_BUFFER * iobuf, int max_tel, int * ntel, double * x, double * y, double * z, double * r)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>max_tel</i>	maximum number of telescopes allowed
<i>ntel</i>	number of telescopes/detectors
<i>x</i>	X positions
<i>y</i>	Y positions
<i>z</i>	Z positions
<i>r</i>	radius of spheres including the whole devices

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `get_item_begin()`, `get_item_end()`, `get_long()`, `get_real()`, `get_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.24 `int write_camera_layout (IO_BUFFER * iobuf, int itel, int type, int pixels, double * xp, double * yp)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>itel</i>	telescope number
<i>type</i>	camera type (hex/square)
<i>pixels</i>	number of pixels
<i>xp</i>	X positions of pixels
<i>yp</i>	Y position of pixels

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_short()`, `put_vector_of_↵`
`real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.25 `int write_input_lines (IO_BUFFER * iobuf, struct linked_string * list)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>list</i>	starting point of linked list

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_string()`, `_struct_↵`
`_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.26 `int write_photo_electrons (IO_BUFFER * iobuf, int array, int tel, int npe, int flags, int pixels, int * pe_counts, int
* tstart, double * t, double * a, int * photon_counts)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>npe</i>	Total number of photo-electrons in the camera.
<i>pixels</i>	No. of pixels to be written
<i>flags</i>	Bit 0: amplitudes available, bit 1: includes NSB p.e., bit 2: also including no. of photons hitting each pixel.
<i>pe_counts</i>	Numbers of photo-electrons in each pixel
<i>tstart</i>	Offsets in 't' at which data for each pixel starts
<i>t</i>	Time of arrival of photons at the camera.
<i>a</i>	Amplitudes of p.e. signals [mean p.e.] (optional, may be NULL).
<i>photon_counts</i>	Optional number of photons arriving at a pixel (with flags bit 2 set)

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_short()`, `put_↵`
`vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.27 `int write_shower_longitudinal (IO_BUFFER * iobuf, int event, int type, double * data, int ndim, int np, int nthick,
double thickstep)`

See [telling_\(\)](#) in [iact.c](#) for more detailed parameter description.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>event</i>	event number
<i>type</i>	1 = particle numbers, 2 = energy, 3 = energy deposits
<i>data</i>	set of (usually 9) distributions
<i>ndim</i>	maximum number of entries per distribution
<i>np</i>	number of distributions (usually 9)
<i>nthick</i>	number of entries actually filled per distribution (is 1 if called without LONGI being enabled).
<i>thickstep</i>	step size in g/cm**2

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_↵short()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.28 `int write_tel_array_end (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int array)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (as opened in begin_write_tel_array())

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `_struct_IO_ITEM_HEADER↵::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.29 `int write_tel_array_head (IO_BUFFER * iobuf, IO_ITEM_HEADER * ih, int array)`

Note: this function does not finish writing to the I/O block but after writing of the photons a call to [end_write_tel_↵array\(\)](#) is needed.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ih</i>	I/O item header (for item opened here)
<i>array</i>	Number of array

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `_struct_IO_ITEM_HEADER↵::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.30 `int write_tel_block (IO_BUFFER * iobuf, int type, int num, real * data, int len)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>type</i>	block type (see mc_tel.h)

<i>num</i>	Run or event number depending on type
<i>data</i>	Data as passed from CORSIKA
<i>len</i>	Number of elements to be written

Returns

0 (OK), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `_struct_IO_ITEM_HEADER::type`, `ev_reg_entry::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.31 `int write_tel_compact_photons (IO_BUFFER * iobuf, int array, int tel, double photons, struct compact_bunch * cbunches, int nbunches, int ext_bunches, char * ext_fname)`

Usually, calls to this function for each telescope in an array should be enclosed within calls to `begin_write_tel_array()` and `end_write_tel_array()`. This routine writes the more compact format (16 bytes per bunch). The more compact format should usually be used to save memory and disk space.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>cbunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list
<i>ext_bunches</i>	number of elements in external file
<i>ext_fname</i>	name of external (temporary) file

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `compact_bunch::ctime`, `compact_bunch::cy`, `fclose()`, `fopen()`, `_struct_IO_ITEM_HEADER::ident`, `compact_bunch::lambda`, `compact_bunch::log_zem`, `compact_bunch::photons`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_short()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::version`, and `compact_bunch::y`.

6.12.2.32 `int write_tel_offset (IO_BUFFER * iobuf, int narray, double toff, double * xoff, double * yoff)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `write_tel_offset_w()`.

6.12.2.33 `int write_tel_offset_w (IO_BUFFER * iobuf, int narray, double toff, double * xoff, double * yoff, double * weight)`

With respect to the backwards-compatible non-weights version `write_tel_offset()`, this version adds a weight to each offset position which should be normalized in such a way that with uniform sampling it should be the area over which showers are thrown divided by the number of array in each shower. With importance sampling the same relation

should hold on average. So in either case, the average sum of weights for the different offsets in one shower equals just the area over which cores are randomized. This leaves the possibility to change the number of offsets from shower to shower.

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>narray</i>	Number of arrays of telescopes/detectors
<i>toff</i>	Time offset (ns, from first interaction to ground)
<i>xoff</i>	X offsets of arrays
<i>yoff</i>	Y offsets of arrays
<i>weight</i>	Area weight for uniform or importance sampled core offset.

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.12.2.34 `int write_tel_photons (IO_BUFFER * iobuf, int array, int tel, double photons, struct bunch * bunches, int nbunches, int ext_bunches, char * ext_fname)`

Usually, calls to this function for each telescope in an array should be enclosed within calls to `begin_write_tel_array()` and `end_write_tel_array()`. This routine writes the less compact format (32 bytes per bunch).

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>array</i>	array number
<i>tel</i>	telescope number
<i>photons</i>	sum of photons (and fractions) in this device
<i>bunches</i>	list of photon bunches
<i>nbunches</i>	number of elements in bunch list
<i>ext_bunches</i>	number of elements in external file
<i>ext_fname</i>	name of external (temporary) file

Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `bunch::ctime`, `bunch::cy`, `fclose()`, `fileopen()`, `_struct_IO_ITEM_HEADER::ident`, `bunch::lambda`, `bunch::photons`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_real()`, `put_short()`, `_struct_IO_ITEM_HEADER::type`, `_struct_IO_ITEM_HEADER::version`, `bunch::y`, and `bunch::zem`.

6.12.2.35 `int write_tel_pos (IO_BUFFER * iobuf, int ntel, double * x, double * y, double * z, double * r)`

Parameters

<i>iobuf</i>	I/O buffer descriptor
<i>ntel</i>	number of telescopes/detectors
<i>x</i>	X positions
<i>y</i>	Y positions
<i>z</i>	Z positions
<i>r</i>	radius of spheres including the whole devices

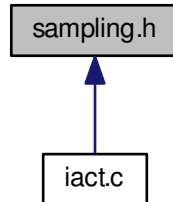
Returns

0 (o.k.), -1, -2, -3 (error, as usual in eventio)

References `_struct_IO_ITEM_HEADER::ident`, `put_item_begin()`, `put_item_end()`, `put_long()`, `put_vector_of_real()`, `_struct_IO_ITEM_HEADER::type`, and `_struct_IO_ITEM_HEADER::version`.

6.13 `sampling.h` File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void `sample_offset` (const char *`sampling_fname`, double `core_range`, double `theta`, double `phi`, double `thetaref`, double `phiref`, double `offax`, double `E`, int `primary`, double *`xoff`, double *`yoff`, double *`sampling_area`)

Get uniformly sampled or importance sampled offset of array with respect to core, in the plane perpendicular to the shower axis.

6.13.1 Function Documentation

6.13.1.1 void `sample_offset` (const char * `sampling_fname`, double `core_range`, double `theta`, double `phi`, double `thetaref`, double `phiref`, double `offax`, double `E`, int `primary`, double * `xoff`, double * `yoff`, double * `sampling_area`)

Parameters

<code>sampling_fname</code>	Name of file with parameters, to be read on first call.
<code>core_range</code>	Maximum core distance as used in data format check [cm]. If not obeying this maximum distance, make sure to switch on the long data format manually.
<code>theta</code>	Zenith angle [radians]
<code>phi</code>	Shower azimuth angle in CORSIKA angle convention [radians].
<code>thetaref</code>	Reference zenith angle (e.g. of VIEWCONE centre) [radians].
<code>phiref</code>	Reference azimuth angle (e.g. of VIEWCONE centre) [radians].
<code>offax</code>	Angle between central direction (typically VIEWCONE centre) and the direction of the current primary [radians].
<code>E</code>	Energy of primary particle [GeV]
<code>primary</code>	Primary particle ID.
<code>xoff</code>	X offset [cm] to be generated.
<code>yoff</code>	Y offset [cm] to be generated.
<code>sampling_area</code>	Area weight of the generated sample (normalized to $\text{Pi} \cdot \text{core_range}^2$) [cm ²].

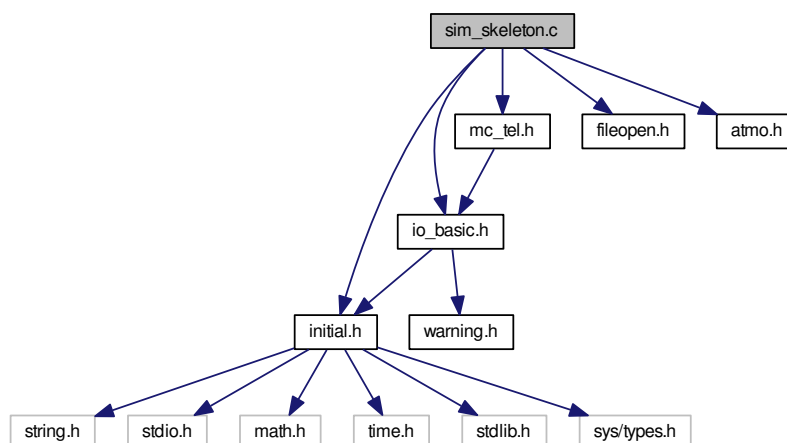
References `core_range`, `fclose()`, `fileopen()`, and `rndm()`.

6.14 `sim_skeleton.c` File Reference

A (non-functional) skeleton program for reading CORSIKA IACT data.

```
#include "initial.h"
#include "io_basic.h"
#include "mc_tel.h"
#include "fileopen.h"
#include "atmo.h"
```

Include dependency graph for sim_skeleton.c:



Data Structures

- struct [camera_electronics](#)
Parameters of the electronics of a telescope.
- struct [mc_options](#)
Options of the simulation passed through to low-level functions.
- struct [mc_run](#)
Basic parameters of the CORSIKA run.
- struct [pm_camera](#)
Parameters of a telescope camera (pixels, ...)
- struct [simulated_shower_parameters](#)
Basic parameters of a simulated shower.
- struct [telescope_array](#)
Description of telescope position, array offsets and shower parameters.
- struct [telescope_optics](#)
Parameters describing the telescope optics.

Macros

- `#define` [MAX_ARRAY](#) 100
The largest no.
- `#define` [MAX_BUNCHES](#) 2500000
Change the following limits as appropriate <<<<
- `#define` **MAX_PHOTOELECTRONS** 1000000 */** The largest number of photo-electrons. */*
- `#define` [MAX_PIXELS](#) 1024
The largest no.
- `#define` [MAX_TEL](#) 16

The largest no.

- `#define Nair(hkm) (1.+0.0002814*exp(-0.0947982*(hkm)-0.00134614*(hkm)*(hkm)))`

Refraction index of air as a function of height in km ($0\text{km} \leq h \leq 8\text{km}$)

Functions

- double **atmospheric_transmission** (int iwl, double zem, double airmass)
- void **atmset_** (int *iatmo, double *obslev)

Set number of atmospheric model profile to be used.

- double **find_max_pos** (double *y, int n)
- double **heigh_** (double *x)

The CORSIKA built-in function for the height as a function of overburden.

- double **line_point_distance** (double x1, double y1, double z1, double cx, double cy, double cz, double x, double y, double z)

Distance between a straight line and a point in space.

- int **main** (int argc, char **argv)

Main program of Cherenkov telescope simulation.

- double **RandFlat** (void)
- double **rhof_** (double *h)

The CORSIKA built-in density lookup function.

- double **thick_** (double *h)

The CORSIKA built-in function for vertical atmospheric thickness (overburden).

Variables

- static double **airlightspeed** = 29.9792458/1.0002256
- struct **linked_string corsika_inputs**

6.14.1 Detailed Description

This file contains a (non-functional) skeleton of the telescope simulation. It serves only as an illustration of the essential usage of CORSIKA related eventio functions to read CORSIKA data in eventio format and how some of the required values are extracted. Comment lines with '...' usually indicate that you should fill in relevant code yourself.

This file comes with no warranties.

If you want a working program using the same interfaces, look for `sim_telarray`.

6.14.2 Macro Definition Documentation

6.14.2.1 `#define MAX_ARRAY 100`

of arrays to be handled

6.14.2.2 `#define MAX_BUNCHES 2500000`

The largest no. of bunches that can be handled.

6.14.2.3 `#define MAX_PIXELS 1024`

of pixels per camers

6.14.2.4 `#define MAX_TEL 16`

of telescopes/array.

6.14.3 Function Documentation

6.14.3.1 void atmset_ (int * *iatmo*, double * *obslev*)

The atmospheric model is initialized first before the interpolating functions can be used. For efficiency reasons, the functions `rhofx_()`, `thickx_()`, ... don't check if the initialisation was done.

This function is called if the 'ATMOSPHERE' keyword is present in the CORSIKA input file.

The function may be called from CORSIKA to initialize the atmospheric model via 'CALL ATMSET(IATMO,OBSLEV)' or such.

Parameters

<i>iatmo</i>	(pointer to) atmospheric profile number; negative for CORSIKA built-in profiles.
<i>obslev</i>	(pointer to) altitude of observation level [cm]

Returns

(none)

References `init_atmosphere()`, `refidx_()`, and `thickx_()`.

6.14.3.2 double heigh_ (double * *thick*)

6.14.3.3 double line_point_distance (double *x1*, double *y1*, double *z1*, double *cx*, double *cy*, double *cz*, double *x*, double *y*, double *z*)

Parameters

<i>x1,y1,z1</i>	reference point on the line
<i>cx,cy,cz</i>	direction cosines of the line
<i>x,y,z</i>	point in space

Returns

distance

6.14.3.4 double rhof_ (double * *height*)

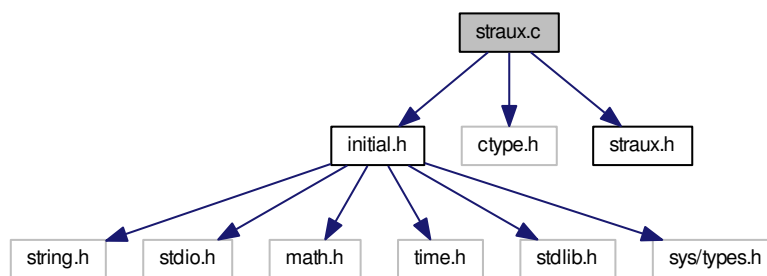
6.14.3.5 double thick_ (double * *height*)

6.15 straux.c File Reference

Check for abbreviations of strings and get words from strings.

```
#include "initial.h"
#include <ctype.h>
#include "straux.h"
```

Include dependency graph for `straux.c`:



Macros

- `#define NO_INITIAL_MACROS 1`

Functions

- `int abbrev (CONST char *s, CONST char *t)`
Compare strings `s` and `t`.
- `int getword (CONST char *s, int *spos, char *word, int maxlen, char blank, char endchar)`
*Copies a blank or `'\0'` or `< endchar >` delimited word from position `*spos` of the string `s` to the string `word` and increment `*spos` to the position of the first non-blank character after the word.*
- `int stricmp (CONST char *a, CONST char *b)`
Case independent comparison of character strings.

6.15.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2010/07/20 13:37:45 \$

Version

CVS \$Revision: 1.4 \$

6.15.2 Function Documentation

6.15.2.1 `int abbrev (CONST char * s, CONST char * t)`

`s` may be an abbreviation of `t`. Upper/lower case in `s` is ignored. `s` has to be at least as long as the leading upper case, digit, and `'_'` part of `t`.

Parameters

<i>s</i>	The string to be checked.
<i>t</i>	The test string with minimum part in upper case.

Returns

1 if *s* is an abbreviation of *t*, 0 if not.

6.15.2.2 `int getword (CONST char * s, int * spos, char * word, int maxlen, char blank, char endchar)`

The word must have a length less than or equal to *maxlen*.

Parameters

<i>s</i>	string with any number of words.
<i>spos</i>	position in the string where we start and end.
<i>word</i>	the extracted word.
<i>maxlen</i>	the maximum allowed length of word.
<i>blank</i>	has the same effect as ' ', i.e. end-of-word.
<i>endchar</i>	his terminates the whole string (as '\0').

Returns

-2 : Invalid string or NULL -1 : The word was longer than *maxlen* (without the terminating '\0'); 0 : There were no more words in the string *s*. 1 : ok, we have a word and there are still more of them in the string *s* 2 : ok, but this was the last word

6.15.2.3 `int stricmp (CONST char * a, CONST char * b)`

Parameters

<i>a,b</i>	– strings to be compared.
------------	---------------------------

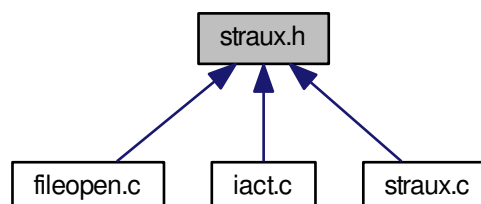
Returns

0 : strings are equal (except perhaps for case) >0 : *a* is lexically 'greater' than *b* <0 : *a* is lexically 'smaller' than *b*

6.16 `straux.h` File Reference

Check for abbreviations of strings and get words from strings.

This graph shows which files directly or indirectly include this file:



Macros

- #define **CONST** const

Functions

- int **abbrev** (CONST char *s, CONST char *t)
Compare strings s and t.
- int **getword** (CONST char *s, int *spos, char *word, int maxlen, char blank, char endchar)
*Copies a blank or '\0' or < endchar > delimited word from position *spos of the string s to the string word and increment *spos to the position of the first non-blank character after the word.*
- int **stricmp** (CONST char *a, CONST char *b)
Case independent comparison of character strings.

6.16.1 Detailed Description**Author**

Konrad Bernloehr

Date

CVS \$Date: 2010/07/20 13:37:45 \$

Version

CVS \$Revision: 1.2 \$

6.16.2 Function Documentation**6.16.2.1 int abbrev (CONST char * s, CONST char * t)**

s may be an abbreviation of t. Upper/lower case in s is ignored. s has to be at least as long as the leading upper case, digit, and '_' part of t.

Parameters

<i>s</i>	The string to be checked.
<i>t</i>	The test string with minimum part in upper case.

Returns

1 if s is an abbreviation of t, 0 if not.

6.16.2.2 int getword (CONST char * s, int * spos, char * word, int maxlen, char blank, char endchar)

The word must have a length less than or equal to maxlen.

Parameters

<i>s</i>	string with any number of words.
<i>spos</i>	position in the string where we start and end.

<i>word</i>	the extracted word.
<i>maxlen</i>	the maximum allowed length of word.
<i>blank</i>	has the same effect as ' ', i.e. end-of-word.
<i>endchar</i>	his terminates the whole string (as '\0').

Returns

-2 : Invalid string or NULL -1 : The word was longer than maxlen (without the terminating '\0'); 0 : There were no more words in the string s. 1 : ok, we have a word and there are still more of them in the string s 2 : ok, but this was the last word

6.16.2.3 int strcmp (CONST char * a, CONST char * b)

Parameters

<i>a,b</i>	– strings to be compared.
------------	---------------------------

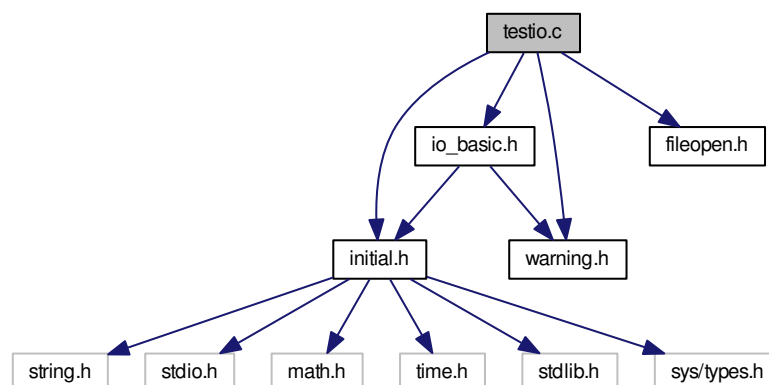
Returns

0 : strings are equal (except perhaps for case) >0 : a is lexically 'greater' than b <0 : a is lexically 'smaller' than b

6.17 testio.c File Reference

Test program for eventio data format.

```
#include "initial.h"
#include "warning.h"
#include "io_basic.h"
#include "fileopen.h"
Include dependency graph for testio.c:
```



Data Structures

- struct [test_struct](#)

Typedefs

- typedef struct [test_struct](#) TEST_DATA

Functions

- int `datacmp` (`TEST_DATA *data1`, `TEST_DATA *data2`)
Compare elements of test data structures.
- int `main` (int argc, char **argv)
Main function for I/O test program.
- int `read_test1` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Read test data with single-element functions.
- int `read_test2` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Read test data with vector functions as far as possible.
- int `read_test3` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Read test data as a nested tree.
- void `syntax` (const char *prg)
Replacement for function missing on OS-9.
- int `write_test1` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Write test data with single-element functions.
- int `write_test2` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Write test data with vector functions as far as possible.
- int `write_test3` (`TEST_DATA *data`, `IO_BUFFER *iobuf`)
Write test data in nested items.

Variables

- static int `care_int`
- static int `care_long`
- static int `care_short`

6.17.1 Detailed Description

Author

Konrad Bernloehr

Date

1994 to 2010

CVS \$Date: 2016/03/03 11:16:39 \$

Version

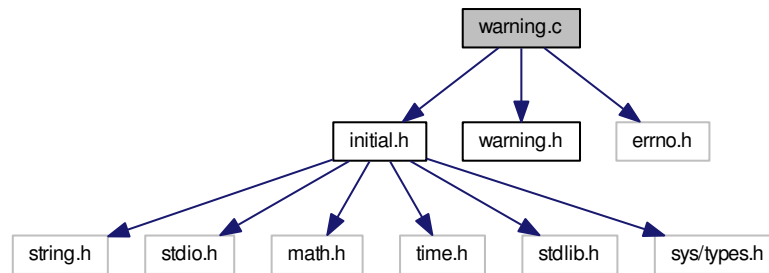
CVS \$Revision: 1.23 \$

6.18 warning.c File Reference

Pass warning messages to the screen or a usr function as set up.

```
#include "initial.h"
#include "warning.h"
#include <errno.h>
```

Include dependency graph for warning.c:



Data Structures

- struct [warn_specific_data](#)
A struct used to store thread-specific data.

Macros

- `#define __WARNING_MODULE 1`
- `#define get_warn_specific() (&warn_defaults)`

Functions

- void [flush_output](#) ()
Flush buffered output.
- void [set_aux_warning_function](#) (char *(*auxfunc)(void))
Set an auxilliary function for warnings.
- void [set_default_aux_warning_function](#) (char *(*auxfunc)(void))
- void [set_default_logging_function](#) (void(*user_function)(const char *, const char *, int, int))
- void [set_default_output_function](#) (void(*user_function)(const char *))
- int [set_default_warning](#) (int level, int mode)
- int [set_log_file](#) (const char *fname)
Set a new log file name and save it in local storage.
- void [set_logging_function](#) (void(*user_function)(const char *, const char *, int, int))
Set user-defined function for logging warnings and errors.
- void [set_output_function](#) (void(*user_function)(const char *))
Set a user-defined function as the function to be used for normal text output.
- int [set_warning](#) (int level, int mode)
Set a specific warning level and mode.
- void [warn_f_output_text](#) (const char *text)
Print a text string (without appending a newline etc.) on the screen or send it to a controlling process, depending on the setting of the output function.
- void [warn_f_warning](#) (const char *msgtext, const char *msgorigin, int msglevel, int msgno)
Issue a warning to screen or other configured target.
- void [warning_status](#) (int *plevel, int *pmode)
Inquire status of warning settings.

Variables

- static struct [warn_specific_data](#) **warn_defaults**

6.18.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2014/02/20 10:53:06 \$

Version

CVS \$Revision: 1.9 \$

One of the most import parameter for setting up the bevaviour is the warning level:

```
-----
Warning level: The lowest level of messages to be displayed
-----
Warning mode:
bit 0: display on screen (stderr),
bit 1: write to file,
bit 2: write with user-defined logging function.
bit 3: display origin if supplied.
bit 4: open log file for appending.
bit 5: call auxilliary function for time/date etc.
bit 6: use the auxilliary function output as origin string
      if no explicit origin was supplied.
bit 7: use syslog().
-----
```

6.18.2 Function Documentation

6.18.2.1 void flush_output (void)

Output is flushed, no matter if it is standard output or a special output function;

Returns

(none)

6.18.2.2 void set_aux_warning_function (char (*)(void) auxfunc)

This function may be used to insert time and date or origin etc. at the beginning of the warning text.

Parameters

<i>auxfunc</i>	– Pointer to a function taking no argument and returning a character string.
----------------	--

Returns

(none)

6.18.2.3 int set_log_file (const char * fname)

If there was a log file with a different name opened previously, close it.

Parameters

<i>fname</i>	New name of log file for warnings
--------------	-----------------------------------

Returns

0 (o.k.), -1 (error)

References warn_specific_data::logfname.

6.18.2.4 void set_logging_function (void(*) (const char *, const char *, int, int) user_function)

Set a user-defined function as the function to be used for logging warnings and errors. To enable usage of this function, bit 2 of the warning mode must be set and other bits reset, if logging to screen and/or disk file is no longer wanted.

Parameter userfunc: Pointer to a function taking two strings (the message text and the origin text, which may be NULL) and two integers (message level and message number).

Returns

(none)

6.18.2.5 void set_output_function (void(*) (const char *) user_function)

Such a function may be used to send output back to a remote control process via network.

Parameter userfunc: Pointer to a function taking a string (the text to be displayed) as argument.

Returns

(none)

References flush_output().

6.18.2.6 int set_warning (int level, int mode)

Parameters

<i>level</i>	Warnings with level below this are ignored.
<i>mode</i>	To screen, to file, with user function ...

Returns

0 if ok, -1 if level and/or mode could not be set.

6.18.2.7 void warn_f_output_text (const char * text)

Parameters

<i>text</i>	A text string to be displayed.
-------------	--------------------------------

Returns

(none)

6.18.2.8 void warn_f_warning (const char * msgtext, const char * msgorigin, int msglevel, int msgno)

Issue a warning to screen and/or file if the warning has a sufficiently large message 'level' (high enough severity). This function should best be called through the macros 'Information', 'Warning', and 'Error'. The name of this function has been changed from 'warning' to '_warning' to avoid trouble if you call 'warning' instead of 'Warning'. Now such a typo causes an error in the link step.

Parameters

<i>msgtext</i>	Warning or error text.
<i>msgorigin</i>	Optional origin (e.g. function name) or NULL.
<i>msglevel</i>	Level of message importance: negative: debugging if needed, 0-9: informative, 10-19: warning, 20-29: error.
<i>msgno</i>	Number of message or 0.

Returns

(none)

References warn_specific_data::logfname.

6.18.2.9 void warning_status (int * *plevel*, int * *pmode*)

Parameters

<i>plevel</i>	Pointer to variable for storing current level.
<i>pmode</i>	Pointer to store the current warning mode.

Returns

(none)

6.18.3 Variable Documentation

6.18.3.1 struct warn_specific_data warn_defaults [static]

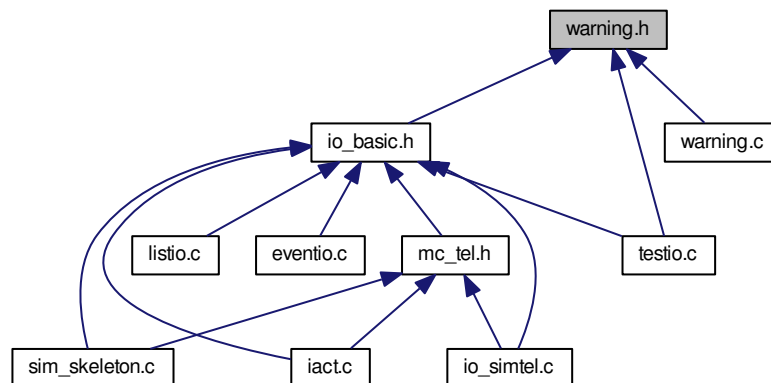
Initial value:

```
=
{
  0,
  1+8,
  "",
  "warning.log",
  "",
  0,
  NULL,
  NULL,
  NULL,
  NULL,
  0
}
```

6.19 warning.h File Reference

Pass warning messages to the screen or a usr function as set up.

This graph shows which files directly or indirectly include this file:



Macros

- `#define Error(string) warn_f_warning(string,WARNING_ORIGIN,20,0)`
- `#define Information(string) warn_f_warning(string,WARNING_ORIGIN,0,0)`
- `#define Output(string) warn_f_output_text(string)`
- `#define Warning(string) warn_f_warning(string,WARNING_ORIGIN,10,0)`
- `#define WARNING_ORIGIN (char *) NULL`

Functions

- void `flush_output` (void)
Flush buffered output.
- void `set_aux_warning_function` (char *(*auxfunc)(void))
Set an auxilliary function for warnings.
- void `set_default_aux_warning_function` (char *(*auxfunc)(void))
- void `set_default_logging_function` (void(*user_function)(const char *, const char *, int, int))
- void `set_default_output_function` (void(*user_function)(const char *))
- int `set_default_warning` (int level, int mode)
- int `set_log_file` (const char *fname)
Set a new log file name and save it in local storage.
- void `set_logging_function` (void(*user_function)(const char *, const char *, int, int))
Set user-defined function for logging warnings and errors.
- void `set_output_function` (void(*user_function)(const char *))
Set a user-defined function as the function to be used for normal text output.
- int `set_warning` (int level, int mode)
Set a specific warning level and mode.
- char * `warn_f_get_message_buffer` (void)
- void `warn_f_output_text` (const char *text)
Print a text string (without appending a newline etc.) on the screen or send it to a controlling process, depending on the setting of the output function.
- void `warn_f_warning` (const char *text, const char *origin, int level, int msgno)
Issue a warning to screen or other configured target.
- void `warning_status` (int *plevel, int *pmode)
Inquire status of warning settings.

6.19.1 Detailed Description

Author

Konrad Bernloehr

Date

CVS \$Date: 2010/07/20 13:37:45 \$

Version

CVS \$Revision: 1.5 \$

6.19.2 Function Documentation

6.19.2.1 void flush_output (void)

Output is flushed, no matter if it is standard output or a special output function;

Returns

(none)

6.19.2.2 void set_aux_warning_function (char (*)(void) auxfunc)

This function may be used to insert time and date or origin etc. at the beginning of the warning text.

Parameters

<i>auxfunc</i>	– Pointer to a function taking no argument and returning a character string.
----------------	--

Returns

(none)

6.19.2.3 int set_log_file (const char * fname)

If there was a log file with a different name opened previously, close it.

Parameters

<i>fname</i>	New name of log file for warnings
--------------	-----------------------------------

Returns

0 (o.k.), -1 (error)

References warn_specific_data::logfname.

6.19.2.4 void set_logging_function (void (*)(const char *, const char *, int, int) user_function)

Set a user-defined function as the function to be used for logging warnings and errors. To enable usage of this function, bit 2 of the warning mode must be set and other bits reset, if logging to screen and/or disk file is no longer wanted.

Parameter userfunc: Pointer to a function taking two strings (the message text and the origin text, which may be NULL) and two integers (message level and message number).

Returns

(none)

6.19.2.5 void set_output_function (void(*)(const char *) user_function)

Such a function may be used to send output back to a remote control process via network.

Parameter userfunc: Pointer to a function taking a string (the text to be displayed) as argument.

Returns

(none)

References flush_output().

6.19.2.6 int set_warning (int level, int mode)

Parameters

<i>level</i>	Warnings with level below this are ignored.
<i>mode</i>	To screen, to file, with user function ...

Returns

0 if ok, -1 if level and/or mode could not be set.

6.19.2.7 void warn_f_output_text (const char * text)

Parameters

<i>text</i>	A text string to be displayed.
-------------	--------------------------------

Returns

(none)

6.19.2.8 void warn_f_warning (const char * msgtext, const char * msgorigin, int msglevel, int msgno)

Issue a warning to screen and/or file if the warning has a sufficiently large message 'level' (high enough severity). This function should best be called through the macros 'Information', 'Warning', and 'Error'. The name of this function has been changed from 'warning' to '_warning' to avoid trouble if you call 'warning' instead of 'Warning'. Now such a typo causes an error in the link step.

Parameters

<i>msgtext</i>	Warning or error text.
<i>msgorigin</i>	Optional origin (e.g. function name) or NULL.
<i>msglevel</i>	Level of message importance: negative: debugging if needed, 0-9: informative, 10-19: warning, 20-29: error.
<i>msgno</i>	Number of message or 0.

Returns

(none)

References warn_specific_data::logfname.

6.19.2.9 void warning_status (int * plevel, int * pmode)

Parameters

<i>plevel</i>	Pointer to variable for storing current level.
<i>pmode</i>	Pointer to store the current warning mode.

Returns

(none)

Index

bunch, [10](#)

datacmp

 The testio program, [4](#)

detstruct, [12](#)

gridstruct, [14](#)

incpath, [14](#)

main

 The listio program, [3](#)

 The testio program, [4](#)

The listio program, [3](#)

 main, [3](#)

The testio program, [4](#)

 datacmp, [4](#)

 main, [4](#)