Particle Properties

A `Particle` corresponds to one entry/slot in the event record. Its properties therefore is a mix of ones belonging to a particle-as-such, like its identity code or four-momentum, and ones related to the event-as-a-whole, like which mother it has.

What is stored for each particle is

- the identity code,
- the status code,
- two mother indices,
- two daughter indices,
- a colour and an anticolour index,
- the four-momentum and mass,
- the scale at which the particle was produced (optional),
- the polarization/spin/helicity of the particle (optional),
- the production vertex and proper lifetime (optional),
- a pointer to the particle kind in the particle data table, and
- a pointer to the event the particle belongs to.

From these, a number of further quantities may be derived.

Basic output methods

The following member functions can be used to extract the most important information:

int Particle::id()
the identity of a particle, according to the PDG particle codes [Yao06].

int Particle::status()
status code. The status code includes information on how a particle was produced, i.e. where in the program execution it was inserted into the event record, and why. It also tells whether the particle is still present or not. It does not tell how a particle disappeared, whether by a decay, a shower branching, a hadronization process, or whatever, but this is implicit in the status code of its daughter(s). The basic scheme is:

- status = +- (10 * i + j)
- + : still remaining particles
- - : decayed/branched/fragmented/... and not remaining
- i = 1 - 9 : stage of event generation inside PYTHIA
- i = 10 -19 : reserved for future expansion
- i >= 20 : free for add-on programs
- j = 1 - 9 : further specification

In detail, the list of used or foreseen status codes is:

- 11 - 19 : beam particles
  - 11 : the event as a whole
  - 12 : incoming beam
  - 13 : incoming beam-inside-beam (e.g. *gamma* inside *e*)
  - 14 : outgoing elastically scattered
  - 15 : outgoing diffractively scattered
- 21 - 29 : particles of the hardest subprocess
  - 21 : incoming
  - 22 : intermediate (intended to have preserved mass)
  - 23 : outgoing
  - 24 : outgoing, nonperturbatively kicked out in diffraction
- 31 - 39 : particles of subsequent subprocesses
  - 31 : incoming
  - 32 : intermediate (intended to have preserved mass)
  - 33 : outgoing
  - 34 : incoming that has already scattered
- 41 - 49 : particles produced by initial-state-showers
  - 41 : incoming on spacelike main branch
  - 42 : incoming copy of recoiler
  - 43 : outgoing produced by a branching
  - 44 : outgoing shifted by a branching
  - 45 : incoming rescattered parton, with changed kinematics owing to ISR in the mother system (cf. status 34)
  - 46 : incoming copy of recoiler when this is a rescattered parton (cf. status 42)
  - 47 : a *W* or *Z* gauge boson produced in the shower evolution
- 51 - 59 : particles produced by final-state-showers
  - 51 : outgoing produced by parton branching
  - 52 : outgoing copy of recoiler, with changed momentum
  - 53 : copy of recoiler when this is incoming parton, with changed momentum
  - 54 : copy of a recoiler, when in the initial state of a different system from the radiator
  - 55 : copy of a recoiler, when in the final state of a different system from the radiator
  - 56 : a *W* or *Z* gauge boson produced in a shower branching (special case of 51)
- 61 - 69 : particles produced by beam-remnant treatment
  - 61 : incoming subprocess particle with primordial *kT* included

    - 62 : outgoing subprocess particle with primordial *kT* included
    - 63 : outgoing beam remnant
- 71 - 79 : partons in preparation of hadronization process
    - 71 : copied partons to collect into contiguous colour singlet
    - 72 : copied recoiling singlet when ministring collapses to one hadron and momentum has to be reshuffled
    - 73 : combination of very nearby partons into one
    - 74 : combination of two junction quarks (+ nearby gluons) to a diquark
    - 75 : gluons split to decouple a junction-antijunction pair
    - 76 : partons with momentum shuffled to decouple a junction-antijunction pair
    - 77 : temporary opposing parton when fragmenting first two strings in to junction (should disappear again)
    - 78 : temporary combined diquark end when fragmenting last string in to junction (should disappear again)
- 81 - 89 : primary hadrons produced by hadronization process
    - 81 : from ministring into one hadron
    - 82 : from ministring into two hadrons
    - 83, 84 : from normal string (the difference between the two is technical, whether fragmented off from the top of the string system or from the bottom, useful for debug only)
    - 85, 86 : primary produced hadrons in junction fragmentation of the first two string legs in to the junction, in order of treatment
- 91 - 99 : particles produced in decay process, or by Bose-Einstein effects
    - 91 : normal decay products
    - 92 : decay products after oscillation *B0* ↔ *B0bar* or *B_s0* ↔ *B_s0bar*
    - 93, 94 : decay handled by external program, normally or with oscillation
    - 99 : particles with momenta shifted by Bose-Einstein effects (not a proper decay, but bookkept as an *1* → *1* such, happening after decays of short-lived resonances but before decays of longer-lived particles)
- 101 - 109 : particles in the handling of R-hadron production and decay, i.e. long-lived (or stable) particles containing a very heavy flavour
    - 101 : when a string system contains two such long-lived particles, the system is split up by the production of a new q-qbar pair (bookkept as decay chains that seemingly need not conserve flavour etc., but do when considered together)
    - 102 : partons rearranged from the long-lived particle end to prepare for fragmentation from this end
    - 103 : intermediate "half-R-hadron" formed when a colour octet particle (like the gluino) has been fragmented on one side, but not yet on the other
    - 104 : an R-hadron

    - 105 : partons or particles formed together with the R-hadron during the fragmentation treatment
    - 106 : subdivision of an R-hadron into its flavour content, with momentum split accordingly, in preparation of the decay of the heavy new particle, if it is unstable
    - 107 : two temporary leftover gluons joined into one in the formation of a gluino-gluon R-hadron.
- 111 - 199 : reserved for future expansion
- 201 - : free to be used by anybody

Note: a clarification on the role of the "hardest" vs. the "subsequent" subprocesses, the 20'ies and 30'ies status code series, respectively. Most events contain exactly one "hardest" $2 \to n$ interaction, and then an arbitrary number of "subsequent" softer $2 \to 2$ ones generated by the MPI framework. In the `SoftQCD:nonDiffractive` event class also the "hardest" is generated by the MPI machinery, and can be arbitrarily soft, but still with 20'ies codes. Diffractive systems span a broad mass range, where the higher masses admit a perturbative description with "hard" and "subsequent" subprocesses, like for nondiffractive events. A double diffractive event can contain up to two such "hardest" interactions, one per diffractive system. A nonperturbative diffractive system does not contain any $2 \to n$ subprocesses, but there is a kicked-out quark or gluon with status 24, combined with a beam remnant of one or two partons with status 63, that together define the mass and longitudinal axis of the diffractive system, for use in the subsequent hadronization. An event may also contain two 20'ies perturbative subcollisions if you use the Second Hard Process generation machinery.

int Particle::mother1()
int Particle::mother2()
the indices in the event record where the first and last mothers are stored, if any. There are six allowed combinations of `mother1` and `mother2`:

1. `mother1 = mother2 = 0`: for lines 0 - 2, where line 0 represents the event as a whole, and 1 and 2 the two incoming beam particles;
2. `mother1 = mother2 > 0`: the particle is a "carbon copy" of its mother, but with changed momentum as a "recoil" effect, e.g. in a shower;
3. `mother1 > 0, mother2 = 0`: the "normal" mother case, where it is meaningful to speak of one single mother to several products, in a shower or decay;
4. `mother1 < mother2`, both > 0, for `abs(status) = 81 - 86`: primary hadrons produced from the fragmentation of a string spanning the range from `mother1` to `mother2`, so that all partons in this range should be considered mothers; and analogously for `abs(status) = 101 - 106`, the formation of R-hadrons;
5. `mother1 < mother2`, both > 0, except case 4: particles with two truly different mothers, in particular the particles emerging from a hard $2 \to n$ interaction.
6. `mother2 < mother1`, both > 0: particles with two truly different mothers, notably for the special case that two nearby partons are joined together into a status 73 or 74 new parton, in the $g + q \to q$ case the q is made first mother to simplify flavour tracing.

Note 1: in backwards evolution of initial-state showers, the mother may well appear below the daughter in the event record.

Note 2: the `motherList()` method below returns a vector of all the mothers, providing a uniform representation for all six cases.

int Particle::daughter1()
int Particle::daughter2()
the indices in the event record where the first and last daughters are stored, if any. There are five allowed combinations of `daughter1` and `daughter2`:

1. `daughter1 = daughter2 = 0`: there are no daughters (so far);
2. `daughter1 = daughter2 > 0`: the particle has a "carbon copy" as its sole daughter, but with changed momentum as a "recoil" effect, e.g. in a shower;
3. `daughter1 > 0, daughter2 = 0`: each of the incoming beams has only (at most) one daughter, namely the initiator parton of the hardest interaction; further, in a *2 → 1* hard interaction, like *q qbar → Z^0*, or in a clustering of two nearby partons, the initial partons only have this one daughter;
4. `daughter1 < daughter2`, both > 0: the particle has a range of decay products from `daughter1` to `daughter2`;
5. `daughter2 < daughter1`, both > 0: the particle has two separately stored decay products (e.g. in backwards evolution of initial-state showers).

Note 1: in backwards evolution of initial-state showers, the daughters may well appear below the mother in the event record.
Note 2: the mother-daughter relation normally is reciprocal, but not always. An example is hadron beams (indices 1 and 2), where each beam remnant and the initiator of each multiparton interaction has the respective beam as mother, but the beam itself only has the initiator of the hardest interaction as daughter.
Note 3: the `daughterList()` method below returns a vector of all the daughters, providing a uniform representation for all five cases. With this method, also all the daughters of the beams are caught, with the initiators of the basic process given first, while the rest are in no guaranteed order (since they are found by a scanning of the event record for particles with the beam as mother, with no further information).

int Particle::col()
int Particle::acol()
the colour and anticolour tags, Les Houches Accord [Boo01] style (starting from tag 101 by default, see below).
Note: in the preliminary implementation of colour sextets (exotic BSM particles) that exists since PYTHIA 8.150, a negative anticolour tag is interpreted as an additional positive colour tag, and vice versa.

double Particle::px()
double Particle::py()
double Particle::pz()
double Particle::e()
the particle four-momentum components.

Vec4 Particle::p()
the particle four-momentum vector, with components as above.

double Particle::m()
the particle mass, stored with a minus sign (times the absolute value) for spacelike virtual particles.

double Particle::scale()
the scale at which a parton was produced, which can be used to restrict its radiation to lower scales in subsequent steps of the shower evolution. Note that scale is linear in momenta, not quadratic (i.e. *Q*, not *Q^2*).

double Particle::pol()
the polarization/spin/helicity of a particle. Currently Pythia does not use this variable for any internal operations, so its meaning is not uniquely defined. The LHA standard sets `SPINUP` to be the cosine of the angle between the spin vector and the 3-momentum of the decaying particle in the lab frame, i.e. restricted to be between +1 and -1. A more convenient choice could be the same quantity in the rest frame of the particle production, either the hard subprocess or the mother particle of a decay. Unknown or unpolarized particles should be assigned the value 9.

double Particle::xProd()
double Particle::yProd()
double Particle::zProd()
double Particle::tProd()
the production vertex coordinates, in mm or mm/c.

Vec4 Particle::vProd()
The production vertex four-vector. Note that the components of a `Vec4` are named `px(), py(), pz() and e()` which of course then should be reinterpreted as above.

double Particle::tau()
the proper lifetime, in mm/c. It is assigned for all hadrons with positive nominal *tau*, *tau_0 > 0*, because it can be used by PYTHIA to decide whether a particle should or should not be allowed to decay, e.g. based on the decay vertex distance to the primary interaction vertex.

Input methods

The same method names as above are also overloaded in versions that set values. These have an input argument of the same type as the respective output above, and are of type `void`.

There are also a few alternative methods for input:

void Particle::statusPos()
void Particle::statusNeg()
sets the status sign positive or negative, without changing the absolute value.

void Particle::statusCode(int code)
changes the absolute value but retains the original sign.

void Particle::mothers(int mother1, int mother2)
sets both mothers in one go.

void Particle::daughters(int daughter1, int daughter2)
sets both daughters in one go.

void Particle::cols(int col, int acol)
sets both colour and anticolour in one go.

void Particle::p(double px, double py, double pz, double e)
sets the four-momentum components in one go.

void Particle::vProd(double xProd, double yProd, double zProd, double tProd)
sets the production vertex components in one go.

Further output methods

In addition, a number of derived quantities can easily be obtained, but cannot be set, such as:

int Particle::idAbs()
the absolute value of the particle identity code.

int Particle::statusAbs()
the absolute value of the status code.

bool Particle::isFinal()
true for a remaining particle, i.e. one with positive status code, else false. Thus, after an event has been fully generated, it separates the final-state particles from intermediate-stage ones. (If used earlier in the generation process, a particle then considered final may well decay later.)

bool Particle::isRescatteredIncoming()
true for particles with a status code -34, -45, -46 or -54, else false. This singles out partons that have been created in a previous scattering but here are bookkept as belonging to the incoming state of another scattering.

bool Particle::hasVertex()
production vertex has been set; if false then production at the origin is assumed.

double Particle::m2()
squared mass, which can be negative for spacelike partons.

double Particle::mCalc()
double Particle::m2Calc()
(squared) mass calculated from the four-momentum; should agree with `m(), m2()` up to roundoff. Negative for spacelike virtualities.

double Particle::eCalc()
energy calculated from the mass and three-momentum; should agree with `e()` up to roundoff. For spacelike partons a positive-energy solution is picked. This need not be the correct one, so it is recommended not to use the method in such cases.

double Particle::pT()
double Particle::pT2()
(squared) transverse momentum.

double Particle::mT()
double Particle::mT2()

(squared) transverse mass. If *m_T^2* is negative, which can happen for a spacelike parton, then `mT()` returns *-sqrt(-m_T^2)*, by analogy with the negative sign used to store spacelike masses.

double Particle::pAbs()
double Particle::pAbs2()
(squared) three-momentum size.

double Particle::eT()
double Particle::eT2()
(squared) transverse energy, *eT = e * sin(theta) = e * pT / pAbs*.

double Particle::theta()
double Particle::phi()
polar and azimuthal angle.

double Particle::thetaXZ()
angle in the *(p_x, p_z)* plane, between *-pi* and *+pi*, with 0 along the *+z* axis

double Particle::pPos()
double Particle::pNeg()
E +- p_z.

double Particle::y()
double Particle::eta()
rapidity and pseudorapidity.

double Particle::xDec()
double Particle::yDec()
double Particle::zDec()
double Particle::tDec()
Vec4 Particle::vDec()
the decay vertex coordinates, in mm or mm/c. This decay vertex is calculated from the production vertex, the proper lifetime and the four-momentum assuming no magnetic field or other detector interference. It can be used to decide whether a decay should be performed or not, and thus is defined also for particles which PYTHIA did not let decay.

Not part of the `Particle` class proper, but obviously tightly linked, are the two methods

double m(const Particle& pp1, const Particle& pp2)
double m2(const Particle& pp1, const Particle& pp2)
the (squared) invariant mass of two particles.

Properties of the particle species

Each Particle contains a pointer to the respective `ParticleDataEntry` object in the particle data tables. This gives access to properties of the particle species as such. It is there mainly for convenience, and should be thrown if an event is written to disk, to avoid any problems of object persistency. Should an event later be read back in, the pointer will be recreated from the `id` code if the normal input methods are used. (Use the `Event::restorePtrs()` method if your persistency scheme bypasses the normal methods.) This pointer is used by the following member functions:

string Particle::name()
the name of the particle.

string Particle::nameWithStatus()
as above, but for negative-status particles the name is given in brackets to emphasize that they are intermediaries.

int Particle::spinType()
*2 *spin + 1* when defined, else 0.

double Particle::charge()
int Particle::chargeType()
charge, and three times it to make an integer.

bool Particle::isCharged()
bool Particle::isNeutral()
charge different from or equal to 0.

int Particle::colType()
0 for colour singlets, 1 for triplets, -1 for antitriplets and 2 for octets. (A preliminary implementation of colour sextets also exists, using 3 for sextets and -3 for antisextets.)

double Particle::m0()
the nominal mass of the particle, according to the data tables.

double Particle::mWidth()
double Particle::mMin()
double Particle::mMax()
the width of the particle, and the minimum and maximum allowed mass value for particles with a width, according to the data tables.

double Particle::mSel()
the mass of the particle, picked according to a Breit-Wigner distribution for particles with width. It is different each time called, and is therefore only used once per particle to set its mass `m()`.

double Particle::constituentMass()
will give the constituent masses for quarks and diquarks, else the same masses as with `m0()`.

double Particle::tau0()
the nominal lifetime *tau_0 > 0*, in mm/c, of the particle species. It is used to assign the actual lifetime *tau*.

bool Particle::mayDecay()
flag whether particle has been declared unstable or not, offering the main user switch to select which particle species to decay.

bool Particle::canDecay()
flag whether decay modes have been declared for a particle, so that it could be decayed, should that be requested.

bool Particle::doExternalDecay()
particles that are decayed by an external program.

bool Particle::isResonance()
particles where the decay is to be treated as part of the hard process, typically with nominal mass above 20 GeV (*W^+-, Z^0, t, ...*).

bool Particle::isVisible()
particles with strong or electric charge, or composed of ones having it, which thereby should be considered visible in a normal detector.

bool Particle::isLepton()
true for a lepton or an antilepton (including neutrinos).

bool Particle::isQuark()
true for a quark or an antiquark.

bool Particle::isGluon()
true for a gluon.

bool Particle::isDiquark()
true for a diquark or an antidiquark.

bool Particle::isParton()
true for a gluon, a quark or antiquark up to the b (but excluding top), and a diquark or antidiquark consisting of quarks up to the b.

bool Particle::isHadron()
true for a hadron (made up out of normal quarks and gluons, i.e. not for R-hadrons and other exotic states).

ParticleDataEntry& particleDataEntry()
a reference to the ParticleDataEntry.

Methods that may access the event the particle belongs to

A particle can be created on its own. When inserted into an event record, it obtains a pointer to that event-as-a-whole. It is then possible to use methods that do not make sense for a particle in isolation. These methods are listed below. Whenever the pointer to the event is not defined, these will return an appropriate "null" value, this being -1 for an integer, false for a bool, and empty for a vector, unless otherwise specified.

void Particle::index()
the index of the particle itself in the event record.

int Particle::statusHepMC()
returns the status code according to the HepMC conventions agreed in February 2009. This convention does not preserve the full information provided by the internal PYTHIA status code, as obtained by `Particle::status()`, but comes reasonably close. The allowed output values are:

- 0 : an empty entry, with no meaningful information and therefore to be skipped unconditionally;

- 1 : a final-state particle, i.e. a particle that is not decayed further by the generator (may also include unstable particles that are to be decayed later, as part of the detector simulation);
- 2 : a decayed Standard Model hadron or tau or mu lepton, excepting virtual intermediate states thereof (i.e. the particle must undergo a normal decay, not e.g. a shower branching);
- 3 : a documentation entry (not used in PYTHIA);
- 4 : an incoming beam particle;
- 11 - 200 : an intermediate (decayed/branched/...) particle that does not fulfill the criteria of status code 2, with a generator-dependent classification of its nature; in PYTHIA the absolute value of the normal status code is used.

Note: for a particle without a properly set pointer to its event, codes 1 and 4 can still be inferred from its status code, while everythg else is assigned code 0.

int Particle::iTopCopy()
int Particle::iBotCopy()
are used to trace carbon copies of the particle up to its top mother or down to its bottom daughter. If there are no such carbon copies, the index of the particle itself will be returned. A carbon copy is when the "same" particle appears several times in the event record, but with changed momentum owing to recoil effects.

int Particle::iTopCopyId()
int Particle::iBotCopyId()
also trace top mother and bottom daughter, but do not require carbon copies, only that one can find an unbroken chain, of mothers or daughters, with the same flavour `id` code. When it encounters ambiguities, say a *g* → *g g* branching or a *u u* → *u u* hard scattering, it will stop the tracing and return the current position. It can be confused by nontrivial flavour changes, e.g. a hard process *u d* → *d u* by *W^+-* exchange will give the wrong answer. These methods therefore are of limited use for common particles, in particular for the gluon, but should work well for "rare" particles.

vector<int> Particle::sisterList(bool traceTopBot = false)
returns a vector of all the sister indices of the particle, i.e. all the daughters of the first mother, except the particle itself. If the argument `traceTopBot = true` the particle is first traced up with `iTopCopy()` before its mother is found, and then all the particles in the `daughterList()` of this mother are traced down with `iBotCopy()`, omitting the original particle itself. The method is not meaningful for the 0 entry, with status code -11, and there returns an empty list.

bool Particle::isAncestor(int iAncestor)
traces the particle upwards through mother, grandmother, and so on, until either *iAncestor* is found or the top of the record is reached. Normally one unique mother is required, as is the case e.g. in decay chains or in parton showers, so that e.g. the tracing through a hard scattering would not work. For hadronization, first-rank hadrons are identified with the respective string endpoint quark, which may be useful e.g. for *b* physics, while higher-rank hadrons give `false`. Currently also ministrings that collapsed to one single hadron and junction topologies give `false`.

bool Particle::undoDecay()
removes the decay chain of the particle and thus restores it to its undecayed state. It is only intended for "normal" particle decay chains, and will return false in other cases, notably if the particle is coloured. The procedure would not work if non-local momentum shifts have been performed, such as

with a Bose-Einstein shift procedure (or for a dipole shower recoiler). As the decay products are erased from the event record, mother and daughter indices are updated to retain a correct history for the remaining particles.

Methods that perform operations

There are some further methods, some of them inherited from `Vec4`, to modify the properties of a particle. They are of little interest to the normal user.

void Particle::rescale3(double fac)
multiply the three-momentum components by `fac`.

void Particle::rescale4(double fac)
multiply the four-momentum components by `fac`.

void Particle::rescale5(double fac)
multiply the four-momentum components and the mass by `fac`.

void Particle::rot(double theta, double phi)
rotate three-momentum and production vertex by these polar and azimuthal angles.

void Particle::bst(double betaX, double betaY, double betaZ)
boost four-momentum and production vertex by this three-vector.

void Particle::bst(double betaX, double betaY, double betaZ, double gamma)
as above, but also input the *gamma* value, to reduce roundoff errors.

void Particle::bst(const Vec4& pBst)
boost four-momentum and production vertex by *beta = (px/e, py/e, pz/e)*.

void Particle::bst(const Vec4& pBst, double mBst)
as above, but also use *gamma> = e/m* to reduce roundoff errors.

void Particle::bstback(const Vec4& pBst)
void Particle::bstback(const Vec4& pBst, double mBst)
as above, but with sign of boost flipped.

void Particle::rotbst(const RotBstMatrix& M)
combined rotation and boost of the four-momentum and production vertex.

void Particle::offsetHistory(int minMother, int addMother, int minDaughter, int addDaughter))
add a positive offset to the mother and daughter indices, i.e. if `mother1` is above `minMother` then `addMother` is added to it, same with `mother2`, if `daughter1` is above `minDaughter` then `addDaughter` is added to it, same with `daughter2`.

void Particle::offsetCol(int addCol)
add a positive offset to colour indices, i.e. if `col` is positive then `addCol` is added to it, same with `acol`.

Constructors and operators

Normally a user would not need to create new particles. However, if necessary, the following constructors and methods may be of interest.

Particle::Particle()
constructs an empty particle, i.e. where all properties have been set 0 or equivalent.

Particle::Particle(int id, int status = 0, int mother1 = 0, int mother2 = 0, int daughter1 = 0, int daughter2 = 0, int col = 0, int acol = 0, double px = 0., double py = 0., double pz = 0., double e = 0., double m = 0., double scale = 0., double pol = 9.)
constructs a particle with the input properties provided, and non-provided ones set 0 (9 for `pol`).

Particle::Particle(int id, int status, int mother1, int mother2, int daughter1, int daughter2, int col, int acol, Vec4 p, double m = 0., double scale = 0., double pol = 9.)
constructs a particle with the input properties provided, and non-provided ones set 0 (9 for `pol`).

Particle::Particle(const Particle& pt)
constructs an particle that is a copy of the input one.

Particle& Particle::operator=(const Particle& pt)
copies the input particle.

void Particle::setEvtPtr(Event* evtPtr)
sets the pointer to the `Event` object the particle belongs to. This method is automatically called when a particle is appended to an event record. Also calls `setPDEPtr` below.

void Particle::setPDEPtr(ParticleDataEntry* pdePtr = 0)
sets the pointer to the `ParticleDataEntry` object of the particle, based on its current `id` code. If the particle belongs to an event there is no need to provide the input argument. As explained above, a valid `ParticleDataEntry` pointer is needed for the methods that provide information generic to the particle species.