Types

MuJoCo defines a large number of types:

- Two primitive types.
- C enum types used to define categorical values. These can be classified as:
    - Enums used in mjModel.
    - Enums used in mjData.
    - Enums for abstract visualization.
    - Enums used by the openGL renderer.
    - Enums used by the mjUI user interface package.
    - Enums used by engine plugins.
    - Enums used for procedural model manipulation.

  Note that the API does not use these enum types directly. Instead it uses ints, and the documentation/comments state that certain ints correspond to certain enum types. This is because we want the API to be compiler-independent, and the C standard does not dictate how many bytes must be used to represent an enum type. Nevertheless, for improved readiblity, we recommend using these types when calling API functions which take them as arguments.

- C struct types. These can be classified as:
    - Main structs:
        - mjModel.
        - mjOption (embedded in mjModel).
        - mjData.
    - Auxiliary struct types, also used by the engine.
    - Structs for collecting simulation statistics.
    - Structs for abstract visualization.
    - Structs used by the openGL renderer.
    - Structs used by the UI framework.
    - Structs used for procedural model manipulation.
    - Structs used by engine plugins.
- Several function types for user-defined callbacks.
- Notes regarding specific data structures that require detailed description.

stable

Primitive types

The two types below are defined in mjtnum.h.

mjtNum

This is the floating-point type used throughout the simulator. When using the default build configuration, `mjtNum` is defined as `double`. If the symbol `mjUSESINGLE` is defined, `mjtNum` is defined as `float`.

Currently only the double-precision version of MuJoCo is distributed, although the entire code base works with single-precision as well. We may release the single-precision version in the future, but the double-precision version will always be available. Thus it is safe to write user code assuming double precision. However, our preference is to write code that works with either single or double precision. To this end we provide math utility functions that are always defined with the correct floating-point type.

Note that changing `mjUSESINGLE` in `mjtnum.h` will not change how the library was compiled, and instead will result in numerous link errors. In general, the header files distributed with precompiled MuJoCo should never be changed by the user.

```
// floating point data type and minval
#ifndef mjUSESINGLE
  typedef double mjtNum;
  #define mjMINVAL    1E-15       // minimum value in any denominator
#else
  typedef float mjtNum;
  #define mjMINVAL    1E-15f
#endif
```

mjtByte

Byte type used to represent boolean variables.

```
typedef unsigned char mjtByte;
```

Enum types

All enum types use the `mjt` prefix.

stable

Model

The enums below are defined in mjmodel.h.

mjtDisableBit

Constants which are powers of 2. They are used as bitmasks for the field `disableflags` of mjOption. At runtime this field is `m->opt.disableflags`. The number of these constants is given by `mjNDISABLE` which is also the length of the global string array mjDISABLESTRING with text descriptions of these flags.

```
typedef enum mjtDisableBit_ {     // disable default feature bitflags
  mjDSBL_CONSTRAINT   = 1<<0,     // entire constraint solver
  mjDSBL_EQUALITY     = 1<<1,     // equality constraints
  mjDSBL_FRICTIONLOSS = 1<<2,     // joint and tendon frictionloss constraints
  mjDSBL_LIMIT        = 1<<3,     // joint and tendon limit constraints
  mjDSBL_CONTACT      = 1<<4,     // contact constraints
  mjDSBL_PASSIVE      = 1<<5,     // passive forces
  mjDSBL_GRAVITY      = 1<<6,     // gravitational forces
  mjDSBL_CLAMPCTRL    = 1<<7,     // clamp control to specified range
  mjDSBL_WARMSTART    = 1<<8,     // warmstart constraint solver
  mjDSBL_FILTERPARENT = 1<<9,     // remove collisions with parent body
  mjDSBL_ACTUATION    = 1<<10,    // apply actuation forces
  mjDSBL_REFSAFE      = 1<<11,    // integrator safety: make ref[0]>=2*timestep
  mjDSBL_SENSOR       = 1<<12,    // sensors
  mjDSBL_MIDPHASE     = 1<<13,    // mid-phase collision filtering
  mjDSBL_EULERDAMP    = 1<<14,    // implicit integration of joint damping in Euler integrat
  mjDSBL_AUTORESET    = 1<<15,    // automatic reset when numerical issues are detected
  mjDSBL_NATIVECCD    = 1<<16,    // native convex collision detection

  mjNDISABLE          = 17        // number of disable flags
} mjtDisableBit;
```

mjtEnableBit

Constants which are powers of 2. They are used as bitmasks for the field `enableflags` of mjOption. At runtime this field is `m->opt.enableflags`. The number of these constants is given by `mjNENABLE` which is also the length of the global string array mjENABLESTRING with text descriptions of these flags.

```
typedef enum mjtEnableBit_ {      // enable optional feature bitflags
  mjENBL_OVERRIDE     = 1<<0,     // override contact parameters
  mjENBL_ENERGY       = 1<<1,     // energy computation
  mjENBL_FWDINV       = 1<<2,     // record solver statistics
  mjENBL_INVDISCRETE  = 1<<3,     // discrete-time inverse dynamics
                                  // experimental features:
  mjENBL_MULTICCD     = 1<<4,     // multi-point convex collision detection
  mjENBL_ISLAND       = 1<<5,     // constraint island discovery

  mjNENABLE           = 6         // number of enable flags
} mjtEnableBit;
```

stable

mjtJoint

Primitive joint types. These values are used in `m->jnt_type`. The numbers in the comments indicate how many positional coordinates each joint type has. Note that ball joints and rotational components of free joints are represented as unit quaternions - which have 4 positional coordinates but 3 degrees of freedom each.

```
typedef enum mjtJoint_ {          // type of degree of freedom
  mjJNT_FREE          = 0,        // global position and orientation (quat)       (7)
  mjJNT_BALL,                     // orientation (quat) relative to parent        (4)
  mjJNT_SLIDE,                    // sliding distance along body-fixed axis       (1)
  mjJNT_HINGE                     // rotation angle (rad) around body-fixed axis  (1)
} mjtJoint;
```

mjtGeom

Geometric types supported by MuJoCo. The first group are “official” geom types that can be used in the model. The second group are geom types that cannot be used in the model but are used by the visualizer to add decorative elements. These values are used in `m->geom_type` and `m->site_type`.

```
typedef enum mjtGeom_ {           // type of geometric shape
  // regular geom types
  mjGEOM_PLANE        = 0,        // plane
  mjGEOM_HFIELD,                  // height field
  mjGEOM_SPHERE,                  // sphere
  mjGEOM_CAPSULE,                 // capsule
  mjGEOM_ELLIPSOID,               // ellipsoid
  mjGEOM_CYLINDER,                // cylinder
  mjGEOM_BOX,                     // box
  mjGEOM_MESH,                    // mesh
  mjGEOM_SDF,                     // signed distance field

  mjNGEOMTYPES,                   // number of regular geom types

  // rendering-only geom types: not used in mjModel, not counted in mjNGEOMTYPES
  mjGEOM_ARROW        = 100,      // arrow
  mjGEOM_ARROW1,                  // arrow without wedges
  mjGEOM_ARROW2,                  // arrow in both directions
  mjGEOM_LINE,                    // line
  mjGEOM_LINEBOX,                 // box with line edges
  mjGEOM_FLEX,                    // flex
  mjGEOM_SKIN,                    // skin
  mjGEOM_LABEL,                   // text label
  mjGEOM_TRIANGLE,                // triangle

  mjGEOM_NONE         = 1001      // missing geom type
} mjtGeom;
```

stable

mjtCamLight

Dynamic modes for cameras and lights, specifying how the camera/light position and orientation are computed. These values are used in `m->cam_mode` and `m->light_mode`.

```
typedef enum mjtCamLight_ {        // tracking mode for camera and light
  mjCAMLIGHT_FIXED     = 0,        // pos and rot fixed in body
  mjCAMLIGHT_TRACK,                // pos tracks body, rot fixed in global
  mjCAMLIGHT_TRACKCOM,             // pos tracks subtree com, rot fixed in body
  mjCAMLIGHT_TARGETBODY,           // pos fixed in body, rot tracks target body
  mjCAMLIGHT_TARGETBODYCOM         // pos fixed in body, rot tracks target subtree com
} mjtCamLight;
```

mjtTexture

Texture types, specifying how the texture will be mapped. These values are used in `m->tex_type`.

```
typedef enum mjtTexture_ {         // type of texture
  mjTEXTURE_2D         = 0,        // 2d texture, suitable for planes and hfields
  mjTEXTURE_CUBE,                  // cube texture, suitable for all other geom types
  mjTEXTURE_SKYBOX                 // cube texture used as skybox
} mjtTexture;
```

mjtTextureRole

Texture roles, specifying how the renderer should interpret the texture. Note that the MuJoCo built-in renderer only uses RGB textures. These values are used to store the texture index in the material's array `m->mat_texid`.

```
typedef enum mjtTextureRole_ {     // role of texture map in rendering
  mjTEXROLE_USER       = 0,        // unspecified
  mjTEXROLE_RGB,                   // base color (albedo)
  mjTEXROLE_OCCLUSION,             // ambient occlusion
  mjTEXROLE_ROUGHNESS,             // roughness
  mjTEXROLE_METALLIC,              // metallic
  mjTEXROLE_NORMAL,                // normal (bump) map
  mjTEXROLE_OPACITY,               // transparency
  mjTEXROLE_EMISSIVE,              // light emission
  mjTEXROLE_RGBA,                  // base color, opacity
  mjTEXROLE_ORM,                   // occlusion, roughness, metallic
  mjNTEXROLE
} mjtTextureRole;
```

mjtIntegrator

Numerical integrator types. These values are used in `m->opt.integrato`

stable

```
typedef enum mjtIntegrator_ {      // integrator mode
  mjINT_EULER          = 0,        // semi-implicit Euler
  mjINT_RK4,                      // 4th-order Runge Kutta
  mjINT_IMPLICIT,                 // implicit in velocity
  mjINT_IMPLICITFAST              // implicit in velocity, no rne derivative
} mjtIntegrator;
```

mjtCone

Available friction cone types. These values are used in `m->opt.cone`.

```
typedef enum mjtCone_ {           // type of friction cone
  mjCONE_PYRAMIDAL     = 0,       // pyramidal
  mjCONE_ELLIPTIC                 // elliptic
} mjtCone;
```

mjtJacobian

Available Jacobian types. These values are used in `m->opt.jacobian`.

```
typedef enum mjtJacobian_ {       // type of constraint Jacobian
  mjJAC_DENSE          = 0,       // dense
  mjJAC_SPARSE,                   // sparse
  mjJAC_AUTO                      // dense if nv<60, sparse otherwise
} mjtJacobian;
```

mjtSolver

Available constraint solver algorithms. These values are used in `m->opt.solver`.

```
typedef enum mjtSolver_ {         // constraint solver algorithm
  mjSOL_PGS            = 0,       // PGS    (dual)
  mjSOL_CG,                       // CG     (primal)
  mjSOL_NEWTON                    // Newton (primal)
} mjtSolver;
```

mjtEq

Equality constraint types. These values are used in `m->eq_type`.

```
typedef enum mjtEq_ {             // type of equality constraint
  mjEQ_CONNECT        = 0,        // connect two bodies at a point (ball joint)
  mjEQ_WELD,                      // fix relative position and orientation of two bodies
  mjEQ_JOINT,                     // couple the values of two scalar joints with cubic
  mjEQ_TENDON,                    // couple the lengths of two tendons with cubic
  mjEQ_FLEX,                      // fix all edge lengths of a flex
  mjEQ_DISTANCE                   // unsupported, will cause an error if used
} mjtEq;
```

mjtWrap

stable

Tendon wrapping object types. These values are used in `m->wrap_type`.

```
typedef enum mjtWrap_ {           // type of tendon wrap object
  mjWRAP_NONE         = 0,        // null object
  mjWRAP_JOINT,                   // constant moment arm
  mjWRAP_PULLEY,                  // pulley used to split tendon
  mjWRAP_SITE,                    // pass through site
  mjWRAP_SPHERE,                  // wrap around sphere
  mjWRAP_CYLINDER                 // wrap around (infinite) cylinder
} mjtWrap;
```

mjtTrn

Actuator transmission types. These values are used in `m->actuator_trntype`.

```
typedef enum mjtTrn_ {            // type of actuator transmission
  mjTRN_JOINT         = 0,        // force on joint
  mjTRN_JOINTINPARENT,            // force on joint, expressed in parent frame
  mjTRN_SLIDERCRANK,              // force via slider-crank linkage
  mjTRN_TENDON,                   // force on tendon
  mjTRN_SITE,                     // force on site
  mjTRN_BODY,                     // adhesion force on a body's geoms

  mjTRN_UNDEFINED     = 1000      // undefined transmission type
} mjtTrn;
```

mjtDyn

Actuator dynamics types. These values are used in `m->actuator_dyntype`.

```
typedef enum mjtDyn_ {            // type of actuator dynamics
  mjDYN_NONE          = 0,        // no internal dynamics; ctrl specifies force
  mjDYN_INTEGRATOR,               // integrator: da/dt = u
  mjDYN_FILTER,                   // linear filter: da/dt = (u-a) / tau
  mjDYN_FILTEREXACT,              // linear filter: da/dt = (u-a) / tau, with exact integrat
  mjDYN_MUSCLE,                   // piece-wise linear filter with two time constants
  mjDYN_USER                      // user-defined dynamics type
} mjtDyn;
```

mjtGain

Actuator gain types. These values are used in `m->actuator_gaintype`.

```
typedef enum mjtGain_ {           // type of actuator gain
  mjGAIN_FIXED        = 0,        // fixed gain
  mjGAIN_AFFINE,                  // const + kp*length + kv*velocity
  mjGAIN_MUSCLE,                  // muscle FLV curve computed by mju_muscleGain()
  mjGAIN_USER                     // user-defined gain type
} mjtGain;
```

stable

mjtBias

Actuator bias types. These values are used in `m->actuator_biastype`.

```
typedef enum mjtBias_ {           // type of actuator bias
  mjBIAS_NONE         = 0,        // no bias
  mjBIAS_AFFINE,                  // const + kp*length + kv*velocity
  mjBIAS_MUSCLE,                  // muscle passive force computed by mju_muscleBias()
  mjBIAS_USER                     // user-defined bias type
} mjtBias;
```

mjtObj

MuJoCo object types. These are used, for example, in the support functions mj_name2id and mj_id2name to convert between object names and integer ids.

```
typedef enum mjtObj_ {            // type of MujoCo object
  mjOBJ_UNKNOWN       = 0,        // unknown object type
  mjOBJ_BODY,                     // body
  mjOBJ_XBODY,                    // body, used to access regular frame instead of i-frame
  mjOBJ_JOINT,                    // joint
  mjOBJ_DOF,                      // dof
  mjOBJ_GEOM,                     // geom
  mjOBJ_SITE,                     // site
  mjOBJ_CAMERA,                   // camera
  mjOBJ_LIGHT,                    // light
  mjOBJ_FLEX,                     // flex
  mjOBJ_MESH,                     // mesh
  mjOBJ_SKIN,                     // skin
  mjOBJ_HFIELD,                   // heightfield
  mjOBJ_TEXTURE,                  // texture
  mjOBJ_MATERIAL,                 // material for rendering
  mjOBJ_PAIR,                     // geom pair to include
  mjOBJ_EXCLUDE,                  // body pair to exclude
  mjOBJ_EQUALITY,                 // equality constraint
  mjOBJ_TENDON,                   // tendon
  mjOBJ_ACTUATOR,                 // actuator
  mjOBJ_SENSOR,                   // sensor
  mjOBJ_NUMERIC,                  // numeric
  mjOBJ_TEXT,                     // text
  mjOBJ_TUPLE,                    // tuple
  mjOBJ_KEY,                      // keyframe
  mjOBJ_PLUGIN,                   // plugin instance

  mjNOBJECT,                      // number of object types

  // meta elements, do not appear in mjModel
  mjOBJ_FRAME         = 100,      // frame
  mjOBJ_DEFAULT,                  // default
  mjOBJ_MODEL                     // entire model

} mjtObj;
```

stable

mjtConstraint

Constraint types. These values are not used in mjModel, but are used in the mjData field `d->efc_type` when the list of active constraints is constructed at each simulation time step.

```
typedef enum mjtConstraint_ {      // type of constraint
  mjCNSTR_EQUALITY    = 0,         // equality constraint
  mjCNSTR_FRICTION_DOF,            // dof friction
  mjCNSTR_FRICTION_TENDON,         // tendon friction
  mjCNSTR_LIMIT_JOINT,             // joint limit
  mjCNSTR_LIMIT_TENDON,            // tendon limit
  mjCNSTR_CONTACT_FRICTIONLESS,    // frictionless contact
  mjCNSTR_CONTACT_PYRAMIDAL,       // frictional contact, pyramidal friction cone
  mjCNSTR_CONTACT_ELLIPTIC         // frictional contact, elliptic friction cone
} mjtConstraint;
```

mjtConstraintState

These values are used by the solver internally to keep track of the constraint states.

```
typedef enum mjtConstraintState_ {  // constraint state
  mjCNSTRSTATE_SATISFIED = 0,       // constraint satisfied, zero cost (limit, contact)
  mjCNSTRSTATE_QUADRATIC,           // quadratic cost (equality, friction, limit, contact)
  mjCNSTRSTATE_LINEARNEG,           // linear cost, negative side (friction)
  mjCNSTRSTATE_LINEARPOS,           // linear cost, positive side (friction)
  mjCNSTRSTATE_CONE                 // squared distance to cone cost (elliptic contact)
} mjtConstraintState;
```

mjtSensor

Sensor types. These values are used in `m->sensor_type`.

```
typedef enum mjtSensor_ {          // type of sensor
  // common robotic sensors, attached to a site
  mjSENS_TOUCH        = 0,         // scalar contact normal forces summed over sensor zone
  mjSENS_ACCELEROMETER,            // 3D linear acceleration, in local frame
  mjSENS_VELOCIMETER,              // 3D linear velocity, in local frame
  mjSENS_GYRO,                     // 3D angular velocity, in local frame
  mjSENS_FORCE,                    // 3D force between site's body and its parent body
  mjSENS_TORQUE,                   // 3D torque between site's body and its parent body
  mjSENS_MAGNETOMETER,             // 3D magnetometer
  mjSENS_RANGEFINDER,              // scalar distance to nearest geom or site along z-axis
  mjSENS_CAMPROJECTION,            // pixel coordinates of a site in the camera image

  // sensors related to scalar joints, tendons, actuators
  mjSENS_JOINTPOS,                 // scalar joint position (hinge and sli
  mjSENS_JOINTVEL,                 // scalar joint velocity (hinge and sli
  mjSENS_TENDONPOS,                // scalar tendon position
  mjSENS_TENDONVEL,                // scalar tendon velocity
```

stable

```
  mjSENS_ACTUATORPOS,             // scalar actuator position
  mjSENS_ACTUATORVEL,             // scalar actuator velocity
  mjSENS_ACTUATORFRC,             // scalar actuator force
  mjSENS_JOINTACTFRC,             // scalar actuator force, measured at the joint
  mjSENS_TENDONACTFRC,            // scalar actuator force, measured at the tendon

  // sensors related to ball joints
  mjSENS_BALLQUAT,                // 4D ball joint quaternion
  mjSENS_BALLANGVEL,              // 3D ball joint angular velocity

  // joint and tendon limit sensors, in constraint space
  mjSENS_JOINTLIMITPOS,           // joint limit distance-margin
  mjSENS_JOINTLIMITVEL,           // joint limit velocity
  mjSENS_JOINTLIMITFRC,           // joint limit force
  mjSENS_TENDONLIMITPOS,          // tendon limit distance-margin
  mjSENS_TENDONLIMITVEL,          // tendon limit velocity
  mjSENS_TENDONLIMITFRC,          // tendon limit force

  // sensors attached to an object with spatial frame: (x)body, geom, site, camera
  mjSENS_FRAMEPOS,                // 3D position
  mjSENS_FRAMEQUAT,               // 4D unit quaternion orientation
  mjSENS_FRAMEXAXIS,              // 3D unit vector: x-axis of object's frame
  mjSENS_FRAMEYAXIS,              // 3D unit vector: y-axis of object's frame
  mjSENS_FRAMEZAXIS,              // 3D unit vector: z-axis of object's frame
  mjSENS_FRAMELINVEL,             // 3D linear velocity
  mjSENS_FRAMEANGVEL,             // 3D angular velocity
  mjSENS_FRAMELINACC,             // 3D linear acceleration
  mjSENS_FRAMEANGACC,             // 3D angular acceleration

  // sensors related to kinematic subtrees; attached to a body (which is the subtree root)
  mjSENS_SUBTREECOM,              // 3D center of mass of subtree
  mjSENS_SUBTREELINVEL,           // 3D linear velocity of subtree
  mjSENS_SUBTREEANGMOM,           // 3D angular momentum of subtree

  // sensors for geometric distance; attached to geoms or bodies
  mjSENS_GEOMDIST,                // signed distance between two geoms
  mjSENS_GEOMNORMAL,              // normal direction between two geoms
  mjSENS_GEOMFROMTO,              // segment between two geoms

  // global sensors
  mjSENS_E_POTENTIAL,             // potential energy
  mjSENS_E_KINETIC,               // kinetic energy
  mjSENS_CLOCK,                   // simulation time

  // plugin-controlled sensors
  mjSENS_PLUGIN,                  // plugin-controlled

  // user-defined sensor
```

stable

```
  mjSENS_USER                     // sensor data provided by mjcb_sensor callback
} mjtSensor;
```

mjtStage

These are the compute stages for the skipstage parameters of mj_forwardSkip and mj_inverseSkip.

```
typedef enum mjtStage_ {          // computation stage
  mjSTAGE_NONE        = 0,        // no computations
  mjSTAGE_POS,                    // position-dependent computations
  mjSTAGE_VEL,                    // velocity-dependent computations
  mjSTAGE_ACC                     // acceleration/force-dependent computations
} mjtStage;
```

mjtDataType

These are the possible sensor data types, used in `mjData.sensor_datatype`.

```
typedef enum mjtDataType_ {       // data type for sensors
  mjDATATYPE_REAL     = 0,        // real values, no constraints
  mjDATATYPE_POSITIVE,            // positive values; 0 or negative: inactive
  mjDATATYPE_AXIS,                // 3D unit vector
  mjDATATYPE_QUATERNION           // unit quaternion
} mjtDataType;
```

mjtSameFrame

Types of frame alignment of elements with their parent bodies. Used as shortcuts during mj_kinematics in the last argument to mj_local2Global.

```
typedef enum mjtSameFrame_ {      // frame alignment of bodies with their children
  mjSAMEFRAME_NONE    = 0,        // no alignment
  mjSAMEFRAME_BODY,               // frame is same as body frame
  mjSAMEFRAME_INERTIA,            // frame is same as inertial frame
  mjSAMEFRAME_BODYROT,            // frame orientation is same as body orientation
  mjSAMEFRAME_INERTIAROT          // frame orientation is same as inertia orientation
} mjtSameFrame;
```

Data

The enums below are defined in mjdata.h.

mjtState

State component elements as integer bitflags and several convenient combinations of these flags. Used by mj_getState, mj_setState and mj_stateSize.

stable

```
typedef enum mjtState_ {          // state elements
  mjSTATE_TIME          = 1<<0,   // time
  mjSTATE_QPOS          = 1<<1,   // position
  mjSTATE_QVEL          = 1<<2,   // velocity
  mjSTATE_ACT           = 1<<3,   // actuator activation
  mjSTATE_WARMSTART     = 1<<4,   // acceleration used for warmstart
  mjSTATE_CTRL          = 1<<5,   // control
  mjSTATE_QFRC_APPLIED  = 1<<6,   // applied generalized force
  mjSTATE_XFRC_APPLIED  = 1<<7,   // applied Cartesian force/torque
  mjSTATE_EQ_ACTIVE     = 1<<8,   // enable/disable constraints
  mjSTATE_MOCAP_POS     = 1<<9,   // positions of mocap bodies
  mjSTATE_MOCAP_QUAT    = 1<<10,  // orientations of mocap bodies
  mjSTATE_USERDATA      = 1<<11,  // user data
  mjSTATE_PLUGIN        = 1<<12,  // plugin state

  mjNSTATE              = 13,     // number of state elements

  // convenience values for commonly used state specifications
  mjSTATE_PHYSICS       = mjSTATE_QPOS | mjSTATE_QVEL | mjSTATE_ACT,
  mjSTATE_FULLPHYSICS   = mjSTATE_TIME | mjSTATE_PHYSICS | mjSTATE_PLUGIN,
  mjSTATE_USER          = mjSTATE_CTRL | mjSTATE_QFRC_APPLIED | mjSTATE_XFRC_APPLIED |
                          mjSTATE_EQ_ACTIVE | mjSTATE_MOCAP_POS | mjSTATE_MOCAP_QUAT |
                          mjSTATE_USERDATA,
  mjSTATE_INTEGRATION   = mjSTATE_FULLPHYSICS | mjSTATE_USER | mjSTATE_WARMSTART
} mjtState;
```

mjtWarning

Warning types. The number of warning types is given by `mjNWARNING` which is also the length of the array `mjData.warning`.

```
typedef enum mjtWarning_ {    // warning types
  mjWARN_INERTIA      = 0,    // (near) singular inertia matrix
  mjWARN_CONTACTFULL,         // too many contacts in contact list
  mjWARN_CNSTRFULL,           // too many constraints
  mjWARN_VGEOMFULL,           // too many visual geoms
  mjWARN_BADQPOS,             // bad number in qpos
  mjWARN_BADQVEL,             // bad number in qvel
  mjWARN_BADQACC,             // bad number in qacc
  mjWARN_BADCTRL,             // bad number in ctrl

  mjNWARNING                  // number of warnings
} mjtWarning;
```

mjtTimer

Timer types. The number of timer types is given by `mjNTIMER` which is also the length of the array `mjData.timer`, as well as the length of the string array mjTIMERSTRING with timer names.

stable

```
typedef enum mjtTimer_ {      // internal timers
  // main api
  mjTIMER_STEP        = 0,   // step
  mjTIMER_FORWARD,           // forward
  mjTIMER_INVERSE,           // inverse

  // breakdown of step/forward
  mjTIMER_POSITION,          // fwdPosition
  mjTIMER_VELOCITY,          // fwdVelocity
  mjTIMER_ACTUATION,         // fwdActuation
  mjTIMER_CONSTRAINT,        // fwdConstraint
  mjTIMER_ADVANCE,           // mj_Euler, mj_implicit

  // breakdown of fwdPosition
  mjTIMER_POS_KINEMATICS,    // kinematics, com, tendon, transmission
  mjTIMER_POS_INERTIA,       // inertia computations
  mjTIMER_POS_COLLISION,     // collision detection
  mjTIMER_POS_MAKE,          // make constraints
  mjTIMER_POS_PROJECT,       // project constraints

  // breakdown of mj_collision
  mjTIMER_COL_BROAD,         // broadphase
  mjTIMER_COL_NARROW,        // narrowphase

  mjNTIMER                   // number of timers
} mjtTimer;
```

Visualization

The enums below are defined in mjvisualize.h.

mjtCatBit

These are the available categories of geoms in the abstract visualizer. The bitmask can be used in the function mjr_render to specify which categories should be rendered.

```
typedef enum mjtCatBit_ {         // bitflags for mjvGeom category
  mjCAT_STATIC        = 1,        // model elements in body 0
  mjCAT_DYNAMIC       = 2,        // model elements in all other bodies
  mjCAT_DECOR         = 4,        // decorative geoms
  mjCAT_ALL           = 7         // select all categories
} mjtCatBit;
```

mjtMouse

These are the mouse actions that the abstract visualizer recognizes. It is up to the user to intercept mouse events and translate them into these actions, as illustrated in simulate.cc.

```
typedef enum mjtMouse_ {          // mouse interaction mode
  mjMOUSE_NONE        = 0,        // no action
  mjMOUSE_ROTATE_V,               // rotate, vertical plane
  mjMOUSE_ROTATE_H,               // rotate, horizontal plane
  mjMOUSE_MOVE_V,                 // move, vertical plane
  mjMOUSE_MOVE_H,                 // move, horizontal plane
  mjMOUSE_ZOOM,                   // zoom
  mjMOUSE_SELECT                  // selection
} mjtMouse;
```

mjtPertBit

These bitmasks enable the translational and rotational components of the mouse perturbation. For the regular mouse, only one can be enabled at a time. For the 3D mouse (SpaceNavigator) both can be enabled simultaneously. They are used in `mjvPerturb.active`.

```
typedef enum mjtPertBit_ {        // mouse perturbations
  mjPERT_TRANSLATE    = 1,        // translation
  mjPERT_ROTATE       = 2         // rotation
} mjtPertBit;
```

mjtCamera

These are the possible camera types, used in `mjvCamera.type`.

```
typedef enum mjtCamera_ {         // abstract camera type
  mjCAMERA_FREE       = 0,        // free camera
  mjCAMERA_TRACKING,              // tracking camera; uses trackbodyid
  mjCAMERA_FIXED,                 // fixed camera; uses fixedcamid
  mjCAMERA_USER                   // user is responsible for setting OpenGL camera
} mjtCamera;
```

mjtLabel

These are the abstract visualization elements that can have text labels. Used in `mjvOption.label`.

```
typedef enum mjtLabel_ {          // object labeling
  mjLABEL_NONE        = 0,        // nothing
  mjLABEL_BODY,                   // body labels
  mjLABEL_JOINT,                  // joint labels
  mjLABEL_GEOM,                   // geom labels
  mjLABEL_SITE,                   // site labels
  mjLABEL_CAMERA,                 // camera labels
  mjLABEL_LIGHT,                  // light labels
  mjLABEL_TENDON,                 // tendon labels
  mjLABEL_ACTUATOR,               // actuator labels
  mjLABEL_CONSTRAINT,             // constraint labels
  mjLABEL_FLEX,                   // flex labels
  mjLABEL_SKIN,                   // skin labels
  mjLABEL_SELECTION,              // selected object
  mjLABEL_SELPNT,                 // coordinates of selection point
```

stable

```
  mjLABEL_CONTACTPOINT,           // contact information
  mjLABEL_CONTACTFORCE,           // magnitude of contact force
  mjLABEL_ISLAND,                 // id of island

  mjNLABEL                        // number of label types
} mjtLabel;
```

mjtFrame

These are the MuJoCo objects whose spatial frames can be rendered. Used in `mjvOption.frame`.

```
typedef enum mjtFrame_ {          // frame visualization
  mjFRAME_NONE        = 0,        // no frames
  mjFRAME_BODY,                   // body frames
  mjFRAME_GEOM,                   // geom frames
  mjFRAME_SITE,                   // site frames
  mjFRAME_CAMERA,                 // camera frames
  mjFRAME_LIGHT,                  // light frames
  mjFRAME_CONTACT,                // contact frames
  mjFRAME_WORLD,                  // world frame

  mjNFRAME                        // number of visualization frames
} mjtFrame;
```

mjtVisFlag

These are indices in the array `mjvOption.flags`, whose elements enable/disable the visualization of the corresponding model or decoration element.

```
typedef enum mjtVisFlag_ {        // flags enabling model element visualization
  mjVIS_CONVEXHULL    = 0,        // mesh convex hull
  mjVIS_TEXTURE,                  // textures
  mjVIS_JOINT,                    // joints
  mjVIS_CAMERA,                   // cameras
  mjVIS_ACTUATOR,                 // actuators
  mjVIS_ACTIVATION,               // activations
  mjVIS_LIGHT,                    // lights
  mjVIS_TENDON,                   // tendons
  mjVIS_RANGEFINDER,              // rangefinder sensors
  mjVIS_CONSTRAINT,               // point constraints
  mjVIS_INERTIA,                  // equivalent inertia boxes
  mjVIS_SCLINERTIA,               // scale equivalent inertia boxes with mass
  mjVIS_PERTFORCE,                // perturbation force
  mjVIS_PERTOBJ,                  // perturbation object
  mjVIS_CONTACTPOINT,             // contact points
  mjVIS_ISLAND,                   // constraint islands
  mjVIS_CONTACTFORCE,             // contact force
  mjVIS_CONTACTSPLIT,             // split contact force into normal and tangent
  mjVIS_TRANSPARENT,              // make dynamic geoms more transparent
```

stable

```
  mjVIS_AUTOCONNECT,                // auto connect joints and body coms
  mjVIS_COM,                        // center of mass
  mjVIS_SELECT,                     // selection point
  mjVIS_STATIC,                     // static bodies
  mjVIS_SKIN,                       // skin
  mjVIS_FLEXVERT,                   // flex vertices
  mjVIS_FLEXEDGE,                   // flex edges
  mjVIS_FLEXFACE,                   // flex element faces
  mjVIS_FLEXSKIN,                   // flex smooth skin (disables the rest)
  mjVIS_BODYBVH,                    // body bounding volume hierarchy
  mjVIS_FLEXBVH,                    // flex bounding volume hierarchy
  mjVIS_MESHBVH,                    // mesh bounding volume hierarchy
  mjVIS_SDFITER,                    // iterations of SDF gradient descent

  mjNVISFLAG                        // number of visualization flags
} mjtVisFlag;
```

mjtRndFlag

These are indices in the array `mjvScene.flags`, whose elements enable/disable OpenGL rendering effects.

```
typedef enum mjtRndFlag_ {          // flags enabling rendering effects
  mjRND_SHADOW        = 0,          // shadows
  mjRND_WIREFRAME,                  // wireframe
  mjRND_REFLECTION,                 // reflections
  mjRND_ADDITIVE,                   // additive transparency
  mjRND_SKYBOX,                     // skybox
  mjRND_FOG,                        // fog
  mjRND_HAZE,                       // haze
  mjRND_SEGMENT,                    // segmentation with random color
  mjRND_IDCOLOR,                    // segmentation with segid+1 color
  mjRND_CULL_FACE,                  // cull backward faces

  mjNRNDFLAG                        // number of rendering flags
} mjtRndFlag;
```

mjtStereo

These are the possible stereo rendering types. They are used in `mjvScene.stereo`.

```
typedef enum mjtStereo_ {           // type of stereo rendering
  mjSTEREO_NONE       = 0,          // no stereo; use left eye only
  mjSTEREO_QUADBUFFERED,            // quad buffered; revert to side-by-side if no hardware su
  mjSTEREO_SIDEBYSIDE               // side-by-side
} mjtStereo;
```

stable

Rendering

The enums below are defined in mjrender.h.

mjtGridPos

These are the possible grid positions for text overlays. They are used as an argument to the function mjr_overlay.

```
typedef enum mjtGridPos_ {          // grid position for overlay
  mjGRID_TOPLEFT      = 0,          // top left
  mjGRID_TOPRIGHT,                  // top right
  mjGRID_BOTTOMLEFT,                // bottom left
  mjGRID_BOTTOMRIGHT,               // bottom right
  mjGRID_TOP,                       // top center
  mjGRID_BOTTOM,                    // bottom center
  mjGRID_LEFT,                      // left center
  mjGRID_RIGHT                      // right center
} mjtGridPos;
```

mjtFramebuffer

These are the possible framebuffers. They are used as an argument to the function mjr_setBuffer.

```
typedef enum mjtFramebuffer_ {      // OpenGL framebuffer option
  mjFB_WINDOW         = 0,          // default/window buffer
  mjFB_OFFSCREEN                    // offscreen buffer
} mjtFramebuffer;
```

mjtDepthMap

These are the depth mapping options. They are used as a value for the `readPixelDepth` attribute of the mjrContext struct, to control how the depth returned by mjr_readPixels is mapped from `znear` to `zfar`.

```
typedef enum mjtDepthMap_ {         // depth mapping for `mjr_readPixels`
  mjDEPTH_ZERONEAR    = 0,          // standard depth map; 0: znear, 1: zfar
  mjDEPTH_ZEROFAR     = 1           // reversed depth map; 1: znear, 0: zfar
} mjtDepthMap;
```

mjtFontScale

These are the possible font sizes. The fonts are predefined bitmaps stored in the dynamic library at three different sizes.

```
typedef enum mjtFontScale_ {        // font scale, used at context creation
  mjFONTSCALE_50      = 50,         // 50% scale, suitable for low-res rendering
  mjFONTSCALE_100     = 100,        // normal scale, suitable in the absence of DPI scaling
  mjFONTSCALE_150     = 150,        // 150% scale
  mjFONTSCALE_200     = 200,        // 200% scale
  mjFONTSCALE_250     = 250,        // 250% scale
  mjFONTSCALE_300     = 300         // 300% scale
} mjtFontScale;
```

mjtFont

These are the possible font types.

```
typedef enum mjtFont_ {              // font type, used at each text operation
  mjFONT_NORMAL       = 0,           // normal font
  mjFONT_SHADOW,                     // normal font with shadow (for higher contrast)
  mjFONT_BIG                         // big font (for user alerts)
} mjtFont;
```

User Interface

The enums below are defined in mjui.h.

mjtButton

Mouse button IDs used in the UI framework.

```
typedef enum mjtButton_ {            // mouse button
  mjBUTTON_NONE = 0,                 // no button
  mjBUTTON_LEFT,                     // left button
  mjBUTTON_RIGHT,                    // right button
  mjBUTTON_MIDDLE                    // middle button
} mjtButton;
```

mjtEvent

Event types used in the UI framework.

```
typedef enum mjtEvent_ {             // mouse and keyboard event type
  mjEVENT_NONE = 0,                  // no event
  mjEVENT_MOVE,                      // mouse move
  mjEVENT_PRESS,                     // mouse button press
  mjEVENT_RELEASE,                   // mouse button release
  mjEVENT_SCROLL,                    // scroll
  mjEVENT_KEY,                       // key press
  mjEVENT_RESIZE,                    // resize
  mjEVENT_REDRAW,                    // redraw
  mjEVENT_FILESDROP                  // files drop
} mjtEvent;
```

mjtItem

Item types used in the UI framework.

```
typedef enum mjtItem_ {              // UI item type
  mjITEM_END = -2,                   // end of definition list (not an item)
  mjITEM_SECTION = -1,               // section (not an item)
  mjITEM_SEPARATOR = 0,              // separator
  mjITEM_STATIC,                     // static text
  mjITEM_BUTTON,                     // button
```

stable

```
  // the rest have data pointer
  mjITEM_CHECKINT,                // check box, int value
  mjITEM_CHECKBYTE,               // check box, mjtByte value
  mjITEM_RADIO,                   // radio group
  mjITEM_RADIOLINE,               // radio group, single line
  mjITEM_SELECT,                  // selection box
  mjITEM_SLIDERINT,               // slider, int value
  mjITEM_SLIDERNUM,               // slider, mjtNum value
  mjITEM_EDITINT,                 // editable array, int values
  mjITEM_EDITNUM,                 // editable array, mjtNum values
  mjITEM_EDITFLOAT,               // editable array, float values
  mjITEM_EDITTXT,                 // editable text

  mjNITEM                         // number of item types
} mjtItem;
```

mjtSection

State of a UI section.

```
typedef enum mjtSection_ {        // UI section state
  mjSECT_CLOSED = 0,              // closed state (regular section)
  mjSECT_OPEN,                    // open state (regular section)
  mjSECT_FIXED                    // fixed section: always open, no title
} mjtSection;
```

Spec

The enums below are defined in mjspec.h.

mjtGeomInertia

Type of inertia inference.

```
typedef enum mjtGeomInertia_ {     // type of inertia inference
  mjINERTIA_VOLUME = 0,            // mass distributed in the volume
  mjINERTIA_SHELL,                 // mass distributed on the surface
} mjtGeomInertia;
```

mjtBuiltin

Type of built-in procedural texture.

```
typedef enum mjtBuiltin_ {         // type of built-in procedural texture
  mjBUILTIN_NONE = 0,              // no built-in texture
  mjBUILTIN_GRADIENT,              // gradient: rgb1->rgb2
  mjBUILTIN_CHECKER,               // checker pattern: rgb1, rgb2
  mjBUILTIN_FLAT                   // 2d: rgb1; cube: rgb1-up, rgb2-side, rgb3-down
} mjtBuiltin;
```

stable

mjtMark

Mark type for procedural textures.

```
typedef enum mjtMark_ {            // mark type for procedural textures
  mjMARK_NONE = 0,                 // no mark
  mjMARK_EDGE,                     // edges
  mjMARK_CROSS,                    // cross
  mjMARK_RANDOM                    // random dots
} mjtMark;
```

mjtLimited

Type of limit specification.

```
typedef enum mjtLimited_ {         // type of limit specification
  mjLIMITED_FALSE = 0,             // not limited
  mjLIMITED_TRUE,                  // limited
  mjLIMITED_AUTO,                  // limited inferred from presence of range
} mjtLimited;
```

mjtAlignFree

Whether to align free joints with the inertial frame.

```
typedef enum mjtAlignFree_ {       // whether to align free joints with the inertial frame
  mjALIGNFREE_FALSE = 0,           // don't align
  mjALIGNFREE_TRUE,                // align
  mjALIGNFREE_AUTO,                // respect the global compiler flag
} mjtAlignFree;
```

mjtInertiaFromGeom

Whether to infer body inertias from child geoms.

```
typedef enum mjtInertiaFromGeom_ { // whether to infer body inertias from child geoms
  mjINERTIAFROMGEOM_FALSE = 0,     // do not use; inertial element required
  mjINERTIAFROMGEOM_TRUE,          // always use; overwrite inertial element
  mjINERTIAFROMGEOM_AUTO           // use only if inertial element is missing
} mjtInertiaFromGeom;
```

mjtOrientation

Type of orientation specifier.

```
typedef enum mjtOrientation_ {     // type of orientation specifier
  mjORIENTATION_QUAT = 0,          // quaternion
  mjORIENTATION_AXISANGLE,         // axis and angle
  mjORIENTATION_XYAXES,            // x and y axes
  mjORIENTATION_ZAXIS,             // z axis (minimal rotation)
```

stable

```
  mjORIENTATION_EULER,             // Euler angles
} mjtOrientation;
```

Plugins

The enums below are defined in mjplugin.h. See Engine plugins for details.

mjtPluginCapabilityBit

Capabilities declared by an engine plugin.

```
typedef enum mjtPluginCapabilityBit_ {
  mjPLUGIN_ACTUATOR = 1<<0,       // actuator forces
  mjPLUGIN_SENSOR   = 1<<1,       // sensor measurements
  mjPLUGIN_PASSIVE  = 1<<2,       // passive forces
  mjPLUGIN_SDF      = 1<<3,       // signed distance fields
} mjtPluginCapabilityBit;
```

Struct types

The three central struct types for physics simulation are mjModel, mjOption (embedded in mjModel) and mjData. An introductory discussion of these strucures can be found in the Overview.

mjModel

This is the main data structure holding the MuJoCo model. It is treated as constant by the simulator. Some specific details regarding datastructures in mjModel can be found below in Notes.

```
struct mjModel_ {
  // ------------------------------- sizes

  // sizes needed at mjModel construction
  int nq;                         // number of generalized coordinates = dim(qpos)
  int nv;                         // number of degrees of freedom = dim(qvel)
  int nu;                         // number of actuators/controls = dim(ctrl)
  int na;                         // number of activation states = dim(act)
  int nbody;                      // number of bodies
  int nbvh;                       // number of total bounding volumes in all bodies
  int nbvhstatic;                 // number of static bounding volumes (aabb stored in mjMod
  int nbvhdynamic;                // number of dynamic bounding volumes (aabb stored in mjDa
  int njnt;                       // number of joints
  int ngeom;                      // number of geoms
  int nsite;                      // number of sites
  int ncam;                       // number of cameras
```

stable

```
  int nlight;                     // number of lights
  int nflex;                      // number of flexes
  int nflexnode;                  // number of dofs in all flexes
  int nflexvert;                  // number of vertices in all flexes
  int nflexedge;                  // number of edges in all flexes
  int nflexelem;                  // number of elements in all flexes
  int nflexelemdata;              // number of element vertex ids in all flexes
  int nflexelemedge;              // number of element edge ids in all flexes
  int nflexshelldata;             // number of shell fragment vertex ids in all flexes
  int nflexevpair;                // number of element-vertex pairs in all flexes
  int nflextexcoord;              // number of vertices with texture coordinates
  int nmesh;                      // number of meshes
  int nmeshvert;                  // number of vertices in all meshes
  int nmeshnormal;                // number of normals in all meshes
  int nmeshtexcoord;              // number of texcoords in all meshes
  int nmeshface;                  // number of triangular faces in all meshes
  int nmeshgraph;                 // number of ints in mesh auxiliary data
  int nmeshpoly;                  // number of polygons in all meshes
  int nmeshpolyvert;              // number of vertices in all polygons
  int nmeshpolymap;               // number of polygons in vertex map
  int nskin;                      // number of skins
  int nskinvert;                  // number of vertices in all skins
  int nskintexvert;               // number of vertiex with texcoords in all skins
  int nskinface;                  // number of triangular faces in all skins
  int nskinbone;                  // number of bones in all skins
  int nskinbonevert;              // number of vertices in all skin bones
  int nhfield;                    // number of heightfields
  int nhfielddata;                // number of data points in all heightfields
  int ntex;                       // number of textures
  int ntexdata;                   // number of bytes in texture rgb data
  int nmat;                       // number of materials
  int npair;                      // number of predefined geom pairs
  int nexclude;                   // number of excluded geom pairs
  int neq;                        // number of equality constraints
  int ntendon;                    // number of tendons
  int nwrap;                      // number of wrap objects in all tendon paths
  int nsensor;                    // number of sensors
  int nnumeric;                   // number of numeric custom fields
  int nnumericdata;               // number of mjtNums in all numeric fields
  int ntext;                      // number of text custom fields
  int ntextdata;                  // number of mjtBytes in all text fields
  int ntuple;                     // number of tuple custom fields
  int ntupledata;                 // number of objects in all tuple fields
  int nkey;                       // number of keyframes
  int nmocap;                     // number of mocap bodies
  int nplugin;                    // number of plugin instances
  int npluginattr;                // number of chars in all plugin config
  int nuser_body;                 // number of mjtNums in body_user
  int nuser_jnt;                  // number of mjtNums in jnt_user
  int nuser_geom;                 // number of mjtNums in geom_user
```

```
int nuser_site;                 // number of mjtNums in site_user
int nuser_cam;                  // number of mjtNums in cam_user
int nuser_tendon;               // number of mjtNums in tendon_user
int nuser_actuator;             // number of mjtNums in actuator_user
int nuser_sensor;               // number of mjtNums in sensor_user
int nnames;                     // number of chars in all names
int npaths;                     // number of chars in all paths

// sizes set after mjModel construction
int nnames_map;                 // number of slots in the names hash map
int nM;                         // number of non-zeros in sparse inertia matrix
int nB;                         // number of non-zeros in sparse body-dof matrix
int nC;                         // number of non-zeros in sparse reduced dof-dof matrix
int nD;                         // number of non-zeros in sparse dof-dof matrix
int nJmom;                      // number of non-zeros in sparse actuator_moment matrix
int ntree;                      // number of kinematic trees under world body
int ngravcomp;                  // number of bodies with nonzero gravcomp
int nemax;                      // number of potential equality-constraint rows
int njmax;                      // number of available rows in constraint Jacobian (legacy
int nconmax;                    // number of potential contacts in contact list (legacy)
int nuserdata;                  // number of mjtNums reserved for the user
int nsensordata;                // number of mjtNums in sensor data vector
int npluginstate;               // number of mjtNums in plugin state vector

size_t narena;                  // number of bytes in the mjData arena (inclusive of stack
size_t nbuffer;                 // number of bytes in buffer

// ------------------------------- options and statistics

mjOption opt;                   // physics options
mjVisual vis;                   // visualization options
mjStatistic stat;               // model statistics

// ------------------------------- buffers

// main buffer
void*     buffer;               // main buffer; all pointers point in it    (nbuffer)

// default generalized coordinates
mjtNum*   qpos0;                // qpos values at default pose              (nq x 1)
mjtNum*   qpos_spring;          // reference pose for springs               (nq x 1)

// bodies
int*      body_parentid;        // id of body's parent                      (nbody x 1)
int*      body_rootid;          // id of root above body                    (nbody x 1)
int*      body_weldid;          // id of body that this body is welded to   (nbody x 1)
int*      body_mocapid;         // id of mocap data; -1: none
int*      body_jntnum;          // number of joints for this body           (nbody x 1)
int*      body_jntadr;          // start addr of joints; -1: no joints      (nbody x 1)
int*      body_dofnum;          // number of motion degrees of freedom      (nbody x 1)
```

stable

```
int*      body_dofadr;          // start addr of dofs; -1: no dofs          (nbody x 1)
int*      body_treeid;          // id of body's kinematic tree; -1: static  (nbody x 1)
int*      body_geomnum;         // number of geoms                          (nbody x 1)
int*      body_geomadr;         // start addr of geoms; -1: no geoms        (nbody x 1)
mjtByte*  body_simple;          // 1: diag M; 2: diag M, sliders only       (nbody x 1)
mjtByte*  body_sameframe;       // same frame as inertia (mjtSameframe)     (nbody x 1)
mjtNum*   body_pos;             // position offset rel. to parent body      (nbody x 3)
mjtNum*   body_quat;            // orientation offset rel. to parent body   (nbody x 4)
mjtNum*   body_ipos;            // local position of center of mass         (nbody x 3)
mjtNum*   body_iquat;           // local orientation of inertia ellipsoid   (nbody x 4)
mjtNum*   body_mass;            // mass                                     (nbody x 1)
mjtNum*   body_subtreemass;     // mass of subtree starting at this body    (nbody x 1)
mjtNum*   body_inertia;         // diagonal inertia in ipos/iquat frame     (nbody x 3)
mjtNum*   body_invweight0;      // mean inv inert in qpos0 (trn, rot)       (nbody x 2)
mjtNum*   body_gravcomp;        // antigravity force, units of body weight  (nbody x 1)
mjtNum*   body_margin;          // MAX over all geom margins                (nbody x 1)
mjtNum*   body_user;            // user data                                (nbody x nuser
int*      body_plugin;          // plugin instance id; -1: not in use       (nbody x 1)
int*      body_contype;         // OR over all geom contypes                (nbody x 1)
int*      body_conaffinity;     // OR over all geom conaffinities           (nbody x 1)
int*      body_bvhadr;          // address of bvh root                      (nbody x 1)
int*      body_bvhnum;          // number of bounding volumes               (nbody x 1)

// bounding volume hierarchy
int*      bvh_depth;            // depth in the bounding volume hierarchy   (nbvh x 1)
int*      bvh_child;            // left and right children in tree          (nbvh x 2)
int*      bvh_nodeid;           // geom or elem id of node; -1: non-leaf    (nbvh x 1)
mjtNum*   bvh_aabb;             // local bounding box (center, size)        (nbvhstatic x

// joints
int*      jnt_type;             // type of joint (mjtJoint)                 (njnt x 1)
int*      jnt_qposadr;          // start addr in 'qpos' for joint's data    (njnt x 1)
int*      jnt_dofadr;           // start addr in 'qvel' for joint's data    (njnt x 1)
int*      jnt_bodyid;           // id of joint's body                       (njnt x 1)
int*      jnt_group;            // group for visibility                     (njnt x 1)
mjtByte*  jnt_limited;          // does joint have limits                   (njnt x 1)
mjtByte*  jnt_actfrclimited;    // does joint have actuator force limits    (njnt x 1)
mjtByte*  jnt_actgravcomp;      // is gravcomp force applied via actuators  (njnt x 1)
mjtNum*   jnt_solref;           // constraint solver reference: limit       (njnt x mjNREF
mjtNum*   jnt_solimp;           // constraint solver impedance: limit       (njnt x mjNIMP
mjtNum*   jnt_pos;              // local anchor position                    (njnt x 3)
mjtNum*   jnt_axis;             // local joint axis                         (njnt x 3)
mjtNum*   jnt_stiffness;        // stiffness coefficient                    (njnt x 1)
mjtNum*   jnt_range;            // joint limits                             (njnt x 2)
mjtNum*   jnt_actfrcrange;      // range of total actuator force            (njnt x 2)
mjtNum*   jnt_margin;           // min distance for limit detection         (njnt x 1)
mjtNum*   jnt_user;             // user data

// dofs
int*      dof_bodyid;           // id of dof's body                         (nv x 1)
```

stable

```
int*      dof_jntid;          // id of dof's joint                     (nv x 1)
int*      dof_parentid;       // id of dof's parent; -1: none          (nv x 1)
int*      dof_treeid;         // id of dof's kinematic tree            (nv x 1)
int*      dof_Madr;           // dof address in M-diagonal             (nv x 1)
int*      dof_simplenum;      // number of consecutive simple dofs     (nv x 1)
mjtNum*   dof_solref;         // constraint solver reference:frictionloss (nv x mjNREF)
mjtNum*   dof_solimp;         // constraint solver impedance:frictionloss (nv x mjNIMP)
mjtNum*   dof_frictionloss;   // dof friction loss                     (nv x 1)
mjtNum*   dof_armature;       // dof armature inertia/mass             (nv x 1)
mjtNum*   dof_damping;        // damping coefficient                   (nv x 1)
mjtNum*   dof_invweight0;     // diag. inverse inertia in qpos0        (nv x 1)
mjtNum*   dof_M0;             // diag. inertia in qpos0                (nv x 1)

// geoms
int*      geom_type;          // geometric type (mjtGeom)              (ngeom x 1)
int*      geom_contype;       // geom contact type                     (ngeom x 1)
int*      geom_conaffinity;   // geom contact affinity                 (ngeom x 1)
int*      geom_condim;        // contact dimensionality (1, 3, 4, 6)   (ngeom x 1)
int*      geom_bodyid;        // id of geom's body                     (ngeom x 1)
int*      geom_dataid;        // id of geom's mesh/hfield; -1: none    (ngeom x 1)
int*      geom_matid;         // material id for rendering; -1: none   (ngeom x 1)
int*      geom_group;         // group for visibility                  (ngeom x 1)
int*      geom_priority;      // geom contact priority                 (ngeom x 1)
int*      geom_plugin;        // plugin instance id; -1: not in use    (ngeom x 1)
mjtByte*  geom_sameframe;     // same frame as body (mjtSameframe)     (ngeom x 1)
mjtNum*   geom_solmix;        // mixing coef for solref/imp in geom pair (ngeom x 1)
mjtNum*   geom_solref;        // constraint solver reference: contact  (ngeom x mjNRE
mjtNum*   geom_solimp;        // constraint solver impedance: contact  (ngeom x mjNIM
mjtNum*   geom_size;          // geom-specific size parameters         (ngeom x 3)
mjtNum*   geom_aabb;          // bounding box, (center, size)          (ngeom x 6)
mjtNum*   geom_rbound;        // radius of bounding sphere             (ngeom x 1)
mjtNum*   geom_pos;           // local position offset rel. to body    (ngeom x 3)
mjtNum*   geom_quat;          // local orientation offset rel. to body (ngeom x 4)
mjtNum*   geom_friction;      // friction for (slide, spin, roll)      (ngeom x 3)
mjtNum*   geom_margin;        // detect contact if dist<margin         (ngeom x 1)
mjtNum*   geom_gap;           // include in solver if dist<margin-gap  (ngeom x 1)
mjtNum*   geom_fluid;         // fluid interaction parameters          (ngeom x mjNFL
mjtNum*   geom_user;          // user data                             (ngeom x nuser
float*    geom_rgba;          // rgba when material is omitted         (ngeom x 4)

// sites
int*      site_type;          // geom type for rendering (mjtGeom)     (nsite x 1)
int*      site_bodyid;        // id of site's body                     (nsite x 1)
int*      site_matid;         // material id for rendering; -1: none   (nsite x 1)
int*      site_group;         // group for visibility                  (nsite x 1)
mjtByte*  site_sameframe;     // same frame as body (mjtSameframe)     (nsite x 1)
mjtNum*   site_size;          // geom size for rendering
mjtNum*   site_pos;           // local position offset rel. to body
mjtNum*   site_quat;          // local orientation offset rel. to body (nsite x 4)
mjtNum*   site_user;          // user data                             (nsite x nuser
```

stable

```
  float*    site_rgba;          // rgba when material is omitted         (nsite x 4)

  // cameras
  int*      cam_mode;           // camera tracking mode (mjtCamLight)     (ncam x 1)
  int*      cam_bodyid;         // id of camera's body                    (ncam x 1)
  int*      cam_targetbodyid;   // id of targeted body; -1: none          (ncam x 1)
  mjtNum*   cam_pos;            // position rel. to body frame            (ncam x 3)
  mjtNum*   cam_quat;           // orientation rel. to body frame         (ncam x 4)
  mjtNum*   cam_poscom0;        // global position rel. to sub-com in qpos0 (ncam x 3)
  mjtNum*   cam_pos0;           // global position rel. to body in qpos0  (ncam x 3)
  mjtNum*   cam_mat0;           // global orientation in qpos0            (ncam x 9)
  int*      cam_orthographic;   // orthographic camera; 0: no, 1: yes     (ncam x 1)
  mjtNum*   cam_fovy;           // y field-of-view (ortho ? len : deg)    (ncam x 1)
  mjtNum*   cam_ipd;            // inter-pupilary distance                (ncam x 1)
  int*      cam_resolution;     // resolution: pixels [width, height]     (ncam x 2)
  float*    cam_sensorsize;     // sensor size: length [width, height]    (ncam x 2)
  float*    cam_intrinsic;      // [focal length; principal point]        (ncam x 4)
  mjtNum*   cam_user;           // user data                              (ncam x nuser_

  // lights
  int*      light_mode;         // light tracking mode (mjtCamLight)      (nlight x 1)
  int*      light_bodyid;       // id of light's body                     (nlight x 1)
  int*      light_targetbodyid; // id of targeted body; -1: none          (nlight x 1)
  mjtByte*  light_directional;  // directional light                      (nlight x 1)
  mjtByte*  light_castshadow;   // does light cast shadows                (nlight x 1)
  float*    light_bulbradius;   // light radius for soft shadows          (nlight x 1)
  mjtByte*  light_active;       // is light on                            (nlight x 1)
  mjtNum*   light_pos;          // position rel. to body frame            (nlight x 3)
  mjtNum*   light_dir;          // direction rel. to body frame           (nlight x 3)
  mjtNum*   light_poscom0;      // global position rel. to sub-com in qpos0 (nlight x 3)
  mjtNum*   light_pos0;         // global position rel. to body in qpos0  (nlight x 3)
  mjtNum*   light_dir0;         // global direction in qpos0              (nlight x 3)
  float*    light_attenuation;  // OpenGL attenuation (quadratic model)   (nlight x 3)
  float*    light_cutoff;       // OpenGL cutoff                          (nlight x 1)
  float*    light_exponent;     // OpenGL exponent                        (nlight x 1)
  float*    light_ambient;      // ambient rgb (alpha=1)                  (nlight x 3)
  float*    light_diffuse;      // diffuse rgb (alpha=1)                  (nlight x 3)
  float*    light_specular;     // specular rgb (alpha=1)                 (nlight x 3)

  // flexes: contact properties
  int*      flex_contype;       // flex contact type                      (nflex x 1)
  int*      flex_conaffinity;   // flex contact affinity                  (nflex x 1)
  int*      flex_condim;        // contact dimensionality (1, 3, 4, 6)    (nflex x 1)
  int*      flex_priority;      // flex contact priority                  (nflex x 1)
  mjtNum*   flex_solmix;        // mix coef for solref/imp in contact pair (nflex x 1)
  mjtNum*   flex_solref;        // constraint solver reference: contact   (nflex x mjNREF
  mjtNum*   flex_solimp;        // constraint solver impedance: contact
  mjtNum*   flex_friction;      // friction for (slide, spin, roll)
  mjtNum*   flex_margin;        // detect contact if dist<margin          (nflex x 1)
  mjtNum*   flex_gap;           // include in solver if dist<margin-gap   (nflex x 1)
```

stable

```
mjtByte*  flex_internal;        // internal flex collision enabled        (nflex x 1)
int*      flex_selfcollide;     // self collision mode (mjtFlexSelf)      (nflex x 1)
int*      flex_activelayers;    // number of active element layers, 3D only (nflex x 1)

// flexes: other properties
int*      flex_dim;             // 1: lines, 2: triangles, 3: tetrahedra  (nflex x 1)
int*      flex_matid;           // material id for rendering              (nflex x 1)
int*      flex_group;           // group for visibility                   (nflex x 1)
int*      flex_interp;          // interpolation (0: vertex, 1: nodes)    (nflex x 1)
int*      flex_nodeadr;         // first node address                     (nflex x 1)
int*      flex_nodenum;         // number of nodes                        (nflex x 1)
int*      flex_vertadr;         // first vertex address                   (nflex x 1)
int*      flex_vertnum;         // number of vertices                     (nflex x 1)
int*      flex_edgeadr;         // first edge address                     (nflex x 1)
int*      flex_edgenum;         // number of edges                        (nflex x 1)
int*      flex_elemadr;         // first element address                  (nflex x 1)
int*      flex_elemnum;         // number of elements                     (nflex x 1)
int*      flex_elemdataadr;     // first element vertex id address        (nflex x 1)
int*      flex_elemedgeadr;     // first element edge id address          (nflex x 1)
int*      flex_shellnum;        // number of shells                       (nflex x 1)
int*      flex_shelldataadr;    // first shell data address               (nflex x 1)
int*      flex_evpairadr;       // first evpair address                   (nflex x 1)
int*      flex_evpairnum;       // number of evpairs                      (nflex x 1)
int*      flex_texcoordadr;     // address in flex_texcoord; -1: none     (nflex x 1)
int*      flex_nodebodyid;      // node body ids                          (nflexnode x 1
int*      flex_vertbodyid;      // vertex body ids                        (nflexvert x 1
int*      flex_edge;            // edge vertex ids (2 per edge)           (nflexedge x 2
int*      flex_elem;            // element vertex ids (dim+1 per elem)    (nflexelemdata
int*      flex_elemtexcoord;    // element texture coordinates (dim+1)    (nflexelemdata
int*      flex_elemedge;        // element edge ids                       (nflexelemedge
int*      flex_elemlayer;       // element distance from surface, 3D only (nflexelem x 1
int*      flex_shell;           // shell fragment vertex ids (dim per frag) (nflexshelldat
int*      flex_evpair;          // (element, vertex) collision pairs      (nflexevpair x
mjtNum*   flex_vert;            // vertex positions in local body frames  (nflexvert x 3
mjtNum*   flex_vert0;           // vertex positions in qpos0 on [0, 1]^d  (nflexvert x 3
mjtNum*   flex_node;            // node positions in local body frames    (nflexnode x 3
mjtNum*   flex_node0;           // Cartesian node positions in qpos0      (nflexnode x 3
mjtNum*   flexedge_length0;     // edge lengths in qpos0                  (nflexedge x 1
mjtNum*   flexedge_invweight0;  // edge inv. weight in qpos0              (nflexedge x 1
mjtNum*   flex_radius;          // radius around primitive element        (nflex x 1)
mjtNum*   flex_stiffness;       // finite element stiffness matrix        (nflexelem x 2
mjtNum*   flex_damping;         // Rayleigh's damping coefficient         (nflex x 1)
mjtNum*   flex_edgestiffness;   // edge stiffness                         (nflex x 1)
mjtNum*   flex_edgedamping;     // edge damping                           (nflex x 1)
mjtByte*  flex_edgeequality;    // is edge equality constraint defined    (nflex x 1)
mjtByte*  flex_rigid;           // are all verices in the same body       (nflex x 1)
mjtByte*  flexedge_rigid;       // are both edge vertices in same body      stable
mjtByte*  flex_centered;        // are all vertex coordinates (0,0,0)
mjtByte*  flex_flatskin;        // render flex skin with flat shading     (nflex x 1)
int*      flex_bvhadr;          // address of bvh root; -1: no bvh        (nflex x 1)
```

```
int*      flex_bvhnum;          // number of bounding volumes              (nflex x 1)
float*    flex_rgba;            // rgba when material is omitted           (nflex x 4)
float*    flex_texcoord;        // vertex texture coordinates              (nflextexcoord

// meshes
int*      mesh_vertadr;         // first vertex address                    (nmesh x 1)
int*      mesh_vertnum;         // number of vertices                      (nmesh x 1)
int*      mesh_faceadr;         // first face address                      (nmesh x 1)
int*      mesh_facenum;         // number of faces                         (nmesh x 1)
int*      mesh_bvhadr;          // address of bvh root                     (nmesh x 1)
int*      mesh_bvhnum;          // number of bvh                           (nmesh x 1)
int*      mesh_normaladr;       // first normal address                    (nmesh x 1)
int*      mesh_normalnum;       // number of normals                       (nmesh x 1)
int*      mesh_texcoordadr;     // texcoord data address; -1: no texcoord  (nmesh x 1)
int*      mesh_texcoordnum;     // number of texcoord                      (nmesh x 1)
int*      mesh_graphadr;        // graph data address; -1: no graph        (nmesh x 1)
float*    mesh_vert;            // vertex positions for all meshes         (nmeshvert x 3
float*    mesh_normal;          // normals for all meshes                  (nmeshnormal x
float*    mesh_texcoord;        // vertex texcoords for all meshes         (nmeshtexcoord
int*      mesh_face;            // vertex face data                        (nmeshface x 3
int*      mesh_facenormal;      // normal face data                        (nmeshface x 3
int*      mesh_facetexcoord;    // texture face data                       (nmeshface x 3
int*      mesh_graph;           // convex graph data                       (nmeshgraph x
mjtNum*   mesh_scale;           // scaling applied to asset vertices       (nmesh x 3)
mjtNum*   mesh_pos;             // translation applied to asset vertices   (nmesh x 3)
mjtNum*   mesh_quat;            // rotation applied to asset vertices      (nmesh x 4)
int*      mesh_pathadr;         // address of asset path for mesh; -1: none (nmesh x 1)
int*      mesh_polynum;         // number of polygons per mesh             (nmesh x 1)
int*      mesh_polyadr;         // first polygon address per mesh          (nmesh x 1)
mjtNum*   mesh_polynormal;      // all polygon normals                     (nmeshpoly x 3
int*      mesh_polyvertadr;     // polygon vertex start address            (nmeshpoly x 1
int*      mesh_polyvertnum;     // number of vertices per polygon          (nmeshpoly x 1
int*      mesh_polyvert;        // all polygon vertices                    (nmeshpolyvert
int*      mesh_polymapadr;      // first polygon address per vertex        (nmeshvert x 1
int*      mesh_polymapnum;      // number of polygons per vertex           (nmeshvert x 1
int*      mesh_polymap;         // vertex to polygon map                   (nmeshpolymap

// skins
int*      skin_matid;           // skin material id; -1: none              (nskin x 1)
int*      skin_group;           // group for visibility                    (nskin x 1)
float*    skin_rgba;            // skin rgba                               (nskin x 4)
float*    skin_inflate;         // inflate skin in normal direction        (nskin x 1)
int*      skin_vertadr;         // first vertex address                    (nskin x 1)
int*      skin_vertnum;         // number of vertices                      (nskin x 1)
int*      skin_texcoordadr;     // texcoord data address; -1: no texcoord  (nskin x 1)
int*      skin_faceadr;         // first face address                      (nskin x 1)
int*      skin_facenum;         // number of faces
int*      skin_boneadr;         // first bone in skin
int*      skin_bonenum;         // number of bones in skin                 (nskin x 1)
float*    skin_vert;            // vertex positions for all skin meshes    (nskinvert x 3
float*    skin_texcoord;        // vertex texcoords for all skin meshes     (nskintexvert
int*      skin_face;            // triangle faces for all skin meshes       (nskinface x 3
int*      skin_bonevertadr;     // first vertex in each bone                (nskinbone x 1
int*      skin_bonevertnum;     // number of vertices in each bone          (nskinbone x 1
float*    skin_bonebindpos;     // bind pos of each bone                    (nskinbone x 3
float*    skin_bonebindquat;    // bind quat of each bone                   (nskinbone x 4
int*      skin_bonebodyid;      // body id of each bone                     (nskinbone x 1
int*      skin_bonevertid;      // mesh ids of vertices in each bone        (nskinbonevert
float*    skin_bonevertweight;  // weights of vertices in each bone         (nskinbonevert
int*      skin_pathadr;         // address of asset path for skin; -1: none (nskin x 1)

// height fields
mjtNum*   hfield_size;          // (x, y, z_top, z_bottom)                  (nhfield x 4)
int*      hfield_nrow;          // number of rows in grid                   (nhfield x 1)
int*      hfield_ncol;          // number of columns in grid                (nhfield x 1)
int*      hfield_adr;           // address in hfield_data                   (nhfield x 1)
float*    hfield_data;          // elevation data                           (nhfielddata x
int*      hfield_pathadr;       // address of hfield asset path; -1: none   (nhfield x 1)

// textures
int*      tex_type;             // texture type (mjtTexture)                (ntex x 1)
int*      tex_height;           // number of rows in texture image          (ntex x 1)
int*      tex_width;            // number of columns in texture image       (ntex x 1)
int*      tex_nchannel;         // number of channels in texture image      (ntex x 1)
int*      tex_adr;              // start address in tex_data                (ntex x 1)
mjtByte*  tex_data;             // pixel values                             (ntexdata x 1)
int*      tex_pathadr;          // address of texture asset path; -1: none  (ntex x 1)

// materials
int*      mat_texid;            // indices of textures; -1: none            (nmat x mjNTEX
mjtByte*  mat_texuniform;       // make texture cube uniform                (nmat x 1)
float*    mat_texrepeat;        // texture repetition for 2d mapping        (nmat x 2)
float*    mat_emission;         // emission (x rgb)                         (nmat x 1)
float*    mat_specular;         // specular (x white)                       (nmat x 1)
float*    mat_shininess;        // shininess coef                           (nmat x 1)
float*    mat_reflectance;      // reflectance (0: disable)                 (nmat x 1)
float*    mat_metallic;         // metallic coef                            (nmat x 1)
float*    mat_roughness;        // roughness coef                           (nmat x 1)
float*    mat_rgba;             // rgba                                     (nmat x 4)

// predefined geom pairs for collision detection; has precedence over exclude
int*      pair_dim;             // contact dimensionality                   (npair x 1)
int*      pair_geom1;           // id of geom1                              (npair x 1)
int*      pair_geom2;           // id of geom2                              (npair x 1)
int*      pair_signature;       // body1 << 16 + body2                      (npair x 1)
mjtNum*   pair_solref;          // solver reference: contact normal         (npair x mjNREF
mjtNum*   pair_solreffriction;  // solver reference: contact friction
mjtNum*   pair_solimp;          // solver impedance: contact
mjtNum*   pair_margin;          // detect contact if dist<margin            (npair x 1)
mjtNum*   pair_gap;             // include in solver if dist<margin-gap     (npair x 1)
```

```
mjtNum*   pair_friction;        // tangent1, 2, spin, roll1, 2              (npair x 5)

// excluded body pairs for collision detection
int*      exclude_signature;    // body1 << 16 + body2                      (nexclude x 1)

// equality constraints
int*      eq_type;              // constraint type (mjtEq)                  (neq x 1)
int*      eq_obj1id;            // id of object 1                           (neq x 1)
int*      eq_obj2id;            // id of object 2                           (neq x 1)
int*      eq_objtype;           // type of both objects (mjtObj)            (neq x 1)
mjtByte*  eq_active0;           // initial enable/disable constraint state  (neq x 1)
mjtNum*   eq_solref;            // constraint solver reference              (neq x mjNREF)
mjtNum*   eq_solimp;            // constraint solver impedance              (neq x mjNIMP)
mjtNum*   eq_data;              // numeric data for constraint              (neq x mjNEQDA

// tendons
int*      tendon_adr;           // address of first object in tendon's path (ntendon x 1)
int*      tendon_num;           // number of objects in tendon's path       (ntendon x 1)
int*      tendon_matid;         // material id for rendering                (ntendon x 1)
int*      tendon_group;         // group for visibility                     (ntendon x 1)
mjtByte*  tendon_limited;       // does tendon have length limits           (ntendon x 1)
mjtByte*  tendon_actfrclimited; // does tendon have actuator force limits   (ntendon x 1)
mjtNum*   tendon_width;         // width for rendering                      (ntendon x 1)
mjtNum*   tendon_solref_lim;    // constraint solver reference: limit       (ntendon x mjN
mjtNum*   tendon_solimp_lim;    // constraint solver impedance: limit       (ntendon x mjN
mjtNum*   tendon_solref_fri;    // constraint solver reference: friction    (ntendon x mjN
mjtNum*   tendon_solimp_fri;    // constraint solver impedance: friction    (ntendon x mjN
mjtNum*   tendon_range;         // tendon length limits                     (ntendon x 2)
mjtNum*   tendon_actfrcrange;   // range of total actuator force            (ntendon x 2)
mjtNum*   tendon_margin;        // min distance for limit detection         (ntendon x 1)
mjtNum*   tendon_stiffness;     // stiffness coefficient                    (ntendon x 1)
mjtNum*   tendon_damping;       // damping coefficient                      (ntendon x 1)
mjtNum*   tendon_armature;      // inertia associated with tendon velocity  (ntendon x 1)
mjtNum*   tendon_frictionloss;  // loss due to friction                     (ntendon x 1)
mjtNum*   tendon_lengthspring;  // spring resting length range              (ntendon x 2)
mjtNum*   tendon_length0;       // tendon length in qpos0                   (ntendon x 1)
mjtNum*   tendon_invweight0;    // inv. weight in qpos0                     (ntendon x 1)
mjtNum*   tendon_user;          // user data                                (ntendon x nus
float*    tendon_rgba;          // rgba when material is omitted            (ntendon x 4)

// list of all wrap objects in tendon paths
int*      wrap_type;            // wrap object type (mjtWrap)               (nwrap x 1)
int*      wrap_objid;           // object id: geom, site, joint             (nwrap x 1)
mjtNum*   wrap_prm;             // divisor, joint coef, or site id          (nwrap x 1)

// actuators
int*      actuator_trntype;     // transmission type (mjtTrn)
int*      actuator_dyntype;     // dynamics type (mjtDyn)
int*      actuator_gaintype;    // gain type (mjtGain)                      (nu x 1)
int*      actuator_biastype;    // bias type (mjtBias)                      (nu x 1)
```

stable

```
int*      actuator_trnid;       // transmission id: joint, tendon, site     (nu x 2)
int*      actuator_actadr;      // first activation address; -1: stateless  (nu x 1)
int*      actuator_actnum;      // number of activation variables           (nu x 1)
int*      actuator_group;       // group for visibility                     (nu x 1)
mjtByte*  actuator_ctrllimited; // is control limited                       (nu x 1)
mjtByte*  actuator_forcelimited;// is force limited                         (nu x 1)
mjtByte*  actuator_actlimited;  // is activation limited                    (nu x 1)
mjtNum*   actuator_dynprm;      // dynamics parameters                      (nu x mjNDYN)
mjtNum*   actuator_gainprm;     // gain parameters                          (nu x mjNGAIN)
mjtNum*   actuator_biasprm;     // bias parameters                          (nu x mjNBIAS)
mjtByte*  actuator_actearly;    // step activation before force             (nu x 1)
mjtNum*   actuator_ctrlrange;   // range of controls                        (nu x 2)
mjtNum*   actuator_forcerange;  // range of forces                          (nu x 2)
mjtNum*   actuator_actrange;    // range of activations                     (nu x 2)
mjtNum*   actuator_gear;        // scale length and transmitted force       (nu x 6)
mjtNum*   actuator_cranklength; // crank length for slider-crank            (nu x 1)
mjtNum*   actuator_acc0;        // acceleration from unit force in qpos0    (nu x 1)
mjtNum*   actuator_length0;     // actuator length in qpos0                 (nu x 1)
mjtNum*   actuator_lengthrange; // feasible actuator length range           (nu x 2)
mjtNum*   actuator_user;        // user data                                (nu x nuser_ac
int*      actuator_plugin;      // plugin instance id; -1: not a plugin     (nu x 1)

// sensors
int*      sensor_type;          // sensor type (mjtSensor)                  (nsensor x 1)
int*      sensor_datatype;      // numeric data type (mjtDataType)          (nsensor x 1)
int*      sensor_needstage;     // required compute stage (mjtStage)        (nsensor x 1)
int*      sensor_objtype;       // type of sensorized object (mjtObj)       (nsensor x 1)
int*      sensor_objid;         // id of sensorized object                  (nsensor x 1)
int*      sensor_reftype;       // type of reference frame (mjtObj)         (nsensor x 1)
int*      sensor_refid;         // id of reference frame; -1: global frame  (nsensor x 1)
int*      sensor_dim;           // number of scalar outputs                 (nsensor x 1)
int*      sensor_adr;           // address in sensor array                  (nsensor x 1)
mjtNum*   sensor_cutoff;        // cutoff for real and positive; 0: ignore  (nsensor x 1)
mjtNum*   sensor_noise;         // noise standard deviation                 (nsensor x 1)
mjtNum*   sensor_user;          // user data                                (nsensor x nus
int*      sensor_plugin;        // plugin instance id; -1: not a plugin     (nsensor x 1)

// plugin instances
int*      plugin;               // globally registered plugin slot number   (nplugin x 1)
int*      plugin_stateadr;      // address in the plugin state array        (nplugin x 1)
int*      plugin_statenum;      // number of states in the plugin instance  (nplugin x 1)
char*     plugin_attr;          // config attributes of plugin instances    (npluginattr x
int*      plugin_attradr;       // address to each instance's config attrib (nplugin x 1)

// custom numeric fields
int*      numeric_adr;          // address of field in numeric_data         (nnumeric x 1)
int*      numeric_size;         // size of numeric field
mjtNum*   numeric_data;         // array of all numeric fields

// custom text fields
```

```
int*      text_adr;          // address of text in text_data           (ntext x 1)
int*      text_size;         // size of text field (strlen+1)          (ntext x 1)
char*     text_data;         // array of all text fields (0-terminated) (ntextdata x 1

// custom tuple fields
int*      tuple_adr;         // address of text in text_data           (ntuple x 1)
int*      tuple_size;        // number of objects in tuple             (ntuple x 1)
int*      tuple_objtype;     // array of object types in all tuples    (ntupledata x
int*      tuple_objid;       // array of object ids in all tuples      (ntupledata x
mjtNum*   tuple_objprm;      // array of object params in all tuples   (ntupledata x

// keyframes
mjtNum*   key_time;          // key time                               (nkey x 1)
mjtNum*   key_qpos;          // key position                           (nkey x nq)
mjtNum*   key_qvel;          // key velocity                           (nkey x nv)
mjtNum*   key_act;           // key activation                         (nkey x na)
mjtNum*   key_mpos;          // key mocap position                     (nkey x nmocap
mjtNum*   key_mquat;         // key mocap quaternion                   (nkey x nmocap
mjtNum*   key_ctrl;          // key control                            (nkey x nu)

// names
int*      name_bodyadr;      // body name pointers                     (nbody x 1)
int*      name_jntadr;       // joint name pointers                    (njnt x 1)
int*      name_geomadr;      // geom name pointers                     (ngeom x 1)
int*      name_siteadr;      // site name pointers                     (nsite x 1)
int*      name_camadr;       // camera name pointers                   (ncam x 1)
int*      name_lightadr;     // light name pointers                    (nlight x 1)
int*      name_flexadr;      // flex name pointers                     (nflex x 1)
int*      name_meshadr;      // mesh name pointers                     (nmesh x 1)
int*      name_skinadr;      // skin name pointers                     (nskin x 1)
int*      name_hfieldadr;    // hfield name pointers                   (nhfield x 1)
int*      name_texadr;       // texture name pointers                  (ntex x 1)
int*      name_matadr;       // material name pointers                 (nmat x 1)
int*      name_pairadr;      // geom pair name pointers                (npair x 1)
int*      name_excludeadr;   // exclude name pointers                  (nexclude x 1)
int*      name_eqadr;        // equality constraint name pointers      (neq x 1)
int*      name_tendonadr;    // tendon name pointers                   (ntendon x 1)
int*      name_actuatoradr;  // actuator name pointers                 (nu x 1)
int*      name_sensoradr;    // sensor name pointers                   (nsensor x 1)
int*      name_numericadr;   // numeric name pointers                  (nnumeric x 1)
int*      name_textadr;      // text name pointers                     (ntext x 1)
int*      name_tupleadr;     // tuple name pointers                    (ntuple x 1)
int*      name_keyadr;       // keyframe name pointers                 (nkey x 1)
int*      name_pluginadr;    // plugin instance name pointers          (nplugin x 1)
char*     names;             // names of all objects, 0-terminated     (nnames x 1)
int*      names_map;         // internal hash map of names             (nnames_map x
```

stable

```
// paths
char*     paths;             // paths to assets, 0-terminated          (npaths x 1)
```

```
  // compilation signature
  uint64_t  signature;            // also held by the mjSpec that compiled this model
};
typedef struct mjModel_ mjModel;
```

mjOption

This is the data structure with simulation options. It corresponds to the MJCF element option. One instance of it is embedded in mjModel.

```
struct mjOption_ {                // physics options
  // timing parameters
  mjtNum timestep;                // timestep
  mjtNum apirate;                 // update rate for remote API (Hz)

  // solver parameters
  mjtNum impratio;                // ratio of friction-to-normal contact impedance
  mjtNum tolerance;               // main solver tolerance
  mjtNum ls_tolerance;            // CG/Newton linesearch tolerance
  mjtNum noslip_tolerance;        // noslip solver tolerance
  mjtNum ccd_tolerance;           // convex collision solver tolerance

  // physical constants
  mjtNum gravity[3];              // gravitational acceleration
  mjtNum wind[3];                 // wind (for lift, drag and viscosity)
  mjtNum magnetic[3];             // global magnetic flux
  mjtNum density;                 // density of medium
  mjtNum viscosity;               // viscosity of medium

  // override contact solver parameters (if enabled)
  mjtNum o_margin;                // margin
  mjtNum o_solref[mjNREF];        // solref
  mjtNum o_solimp[mjNIMP];        // solimp
  mjtNum o_friction[5];           // friction

  // discrete settings
  int integrator;                 // integration mode (mjtIntegrator)
  int cone;                       // type of friction cone (mjtCone)
  int jacobian;                   // type of Jacobian (mjtJacobian)
  int solver;                     // solver algorithm (mjtSolver)
  int iterations;                 // maximum number of main solver iterations
  int ls_iterations;              // maximum number of CG/Newton linesearch iterations
  int noslip_iterations;          // maximum number of noslip solver iterations
  int ccd_iterations;             // maximum number of convex collision solver iterations
  int disableflags;               // bit flags for disabling standard features
  int enableflags;                // bit flags for enabling optional feat
  int disableactuator;            // bit flags for disabling actuators by

  // sdf collision settings
```

stable

```
  int sdf_initpoints;             // number of starting points for gradient descent
  int sdf_iterations;             // max number of iterations for gradient descent
};
typedef struct mjOption_ mjOption;
```

mjData

This is the main data structure holding the simulation state. It is the workspace where all functions read their modifiable inputs and write their outputs.

```
struct mjData_ {
  // constant sizes
  size_t  narena;            // size of the arena in bytes (inclusive of the stack)
  size_t  nbuffer;           // size of main buffer in bytes
  int     nplugin;           // number of plugin instances

  // stack pointer
  size_t  pstack;            // first available byte in stack
  size_t  pbase;             // value of pstack when mj_markStack was last called

  // arena pointer
  size_t  parena;            // first available byte in arena

  // memory utilization statistics
  size_t  maxuse_stack;                     // maximum stack allocation in bytes
  size_t  maxuse_threadstack[mjMAXTHREAD];  // maximum stack allocation per thread in byte
  size_t  maxuse_arena;                     // maximum arena allocation in bytes
  int     maxuse_con;                       // maximum number of contacts
  int     maxuse_efc;                       // maximum number of scalar constraints

  // solver statistics
  mjSolverStat  solver[mjNISLAND*mjNSOLVER];  // solver statistics per island, per iteration
  int           solver_niter[mjNISLAND];      // number of solver iterations, per island
  int           solver_nnz[mjNISLAND];        // number of nonzeros in Hessian or efc_AR, pe
  mjtNum        solver_fwdinv[2];             // forward-inverse comparison: qfrc, efc

  // diagnostics
  mjWarningStat warning[mjNWARNING];          // warning statistics
  mjTimerStat   timer[mjNTIMER];              // timer statistics

  // variable sizes
  int     ncon;              // number of detected contacts
  int     ne;                // number of equality constraints
  int     nf;                // number of friction constraints
  int     nl;                // number of limit constraints
  int     nefc;              // number of constraints
  int     nJ;                // number of non-zeros in constraint Jacobia
  int     nA;                // number of non-zeros in constraint inverse inertia matrix
  int     nisland;           // number of detected constraint islands
```

stable

```
// global properties
mjtNum  time;              // simulation time
mjtNum  energy[2];         // potential, kinetic energy

//-------------------- end of info header

// buffers
void*   buffer;            // main buffer; all pointers point in it            (nbuffer by
void*   arena;             // arena+stack buffer                               (narena byt

//-------------------- main inputs and outputs of the computation

// state
mjtNum* qpos;              // position                                         (nq x 1)
mjtNum* qvel;              // velocity                                         (nv x 1)
mjtNum* act;               // actuator activation                              (na x 1)
mjtNum* qacc_warmstart;    // acceleration used for warmstart                  (nv x 1)
mjtNum* plugin_state;      // plugin state                                     (npluginsta

// control
mjtNum* ctrl;              // control                                          (nu x 1)
mjtNum* qfrc_applied;      // applied generalized force                        (nv x 1)
mjtNum* xfrc_applied;      // applied Cartesian force/torque                   (nbody x 6)
mjtByte* eq_active;        // enable/disable constraints                       (neq x 1)

// mocap data
mjtNum* mocap_pos;         // positions of mocap bodies                        (nmocap x 3
mjtNum* mocap_quat;        // orientations of mocap bodies                     (nmocap x 4

// dynamics
mjtNum* qacc;              // acceleration                                     (nv x 1)
mjtNum* act_dot;           // time-derivative of actuator activation           (na x 1)

// user data
mjtNum* userdata;          // user data, not touched by engine                 (nuserdata

// sensors
mjtNum* sensordata;        // sensor data array                                (nsensordat

// plugins
int*       plugin;         // copy of m->plugin, required for deletion         (nplugin x
uintptr_t* plugin_data;    // pointer to plugin-managed data structure         (nplugin x

//-------------------- POSITION dependent

// computed by mj_fwdPosition/mj_kinematics
mjtNum* xpos;              // Cartesian position of body frame
mjtNum* xquat;             // Cartesian orientation of body frame              (nbody x 4)
mjtNum* xmat;              // Cartesian orientation of body frame              (nbody x 9)
```

```
mjtNum* xipos;             // Cartesian position of body com                   (nbody x 3)
mjtNum* ximat;             // Cartesian orientation of body inertia            (nbody x 9)
mjtNum* xanchor;           // Cartesian position of joint anchor               (njnt x 3)
mjtNum* xaxis;             // Cartesian joint axis                             (njnt x 3)
mjtNum* geom_xpos;         // Cartesian geom position                          (ngeom x 3)
mjtNum* geom_xmat;         // Cartesian geom orientation                       (ngeom x 9)
mjtNum* site_xpos;         // Cartesian site position                          (nsite x 3)
mjtNum* site_xmat;         // Cartesian site orientation                       (nsite x 9)
mjtNum* cam_xpos;          // Cartesian camera position                        (ncam x 3)
mjtNum* cam_xmat;          // Cartesian camera orientation                     (ncam x 9)
mjtNum* light_xpos;        // Cartesian light position                         (nlight x 3
mjtNum* light_xdir;        // Cartesian light direction                        (nlight x 3

// computed by mj_fwdPosition/mj_comPos
mjtNum* subtree_com;       // center of mass of each subtree                   (nbody x 3)
mjtNum* cdof;              // com-based motion axis of each dof (rot:lin)      (nv x 6)
mjtNum* cinert;            // com-based body inertia and mass                  (nbody x 10

// computed by mj_fwdPosition/mj_flex
mjtNum* flexvert_xpos;     // Cartesian flex vertex positions                  (nflexvert
mjtNum* flexelem_aabb;     // flex element bounding boxes (center, size)       (nflexelem
int*    flexedge_J_rownnz; // number of non-zeros in Jacobian row              (nflexedge
int*    flexedge_J_rowadr; // row start address in colind array                (nflexedge
int*    flexedge_J_colind; // column indices in sparse Jacobian                (nflexedge
mjtNum* flexedge_J;        // flex edge Jacobian                               (nflexedge
mjtNum* flexedge_length;   // flex edge lengths                                (nflexedge

// computed by mj_fwdPosition/mj_tendon
int*    ten_wrapadr;       // start address of tendon's path                   (ntendon x
int*    ten_wrapnum;       // number of wrap points in path                    (ntendon x
int*    ten_J_rownnz;      // number of non-zeros in Jacobian row              (ntendon x
int*    ten_J_rowadr;      // row start address in colind array                (ntendon x
int*    ten_J_colind;      // column indices in sparse Jacobian                (ntendon x
mjtNum* ten_J;             // tendon Jacobian                                  (ntendon x
mjtNum* ten_length;        // tendon lengths                                   (ntendon x
int*    wrap_obj;          // geom id; -1: site; -2: pulley                    (nwrap x 2)
mjtNum* wrap_xpos;         // Cartesian 3D points in all paths                 (nwrap x 6)

// computed by mj_fwdPosition/mj_transmission
mjtNum* actuator_length;   // actuator lengths                                 (nu x 1)
int*    moment_rownnz;     // number of non-zeros in actuator_moment row       (nu x 1)
int*    moment_rowadr;     // row start address in colind array                (nu x 1)
int*    moment_colind;     // column indices in sparse Jacobian                (nJmom x 1)
mjtNum* actuator_moment;   // actuator moments                                 (nJmom x 1)

// computed by mj_fwdPosition/mj_crb
mjtNum* crb;               // com-based composite inertia and mass
mjtNum* qM;                // total inertia (sparse)

// computed by mj_fwdPosition/mj_factorM
```

stable

```
mjtNum* qLD;              // L'*D*L factorization of M (sparse)            (nM x 1)
mjtNum* qLDiagInv;        // 1/diag(D)                                     (nv x 1)

// computed by mj_collisionTree
mjtNum*  bvh_aabb_dyn;    // global bounding box (center, size)            (nbvhdynami
mjtByte* bvh_active;      // was bounding volume checked for collision     (nbvh x 1)

//-------------------- POSITION, VELOCITY dependent

// computed by mj_fwdVelocity
mjtNum* flexedge_velocity; // flex edge velocities                         (nflexedge
mjtNum* ten_velocity;     // tendon velocities                             (ntendon x
mjtNum* actuator_velocity; // actuator velocities                          (nu x 1)

// computed by mj_fwdVelocity/mj_comVel
mjtNum* cvel;             // com-based velocity (rot:lin)                  (nbody x 6)
mjtNum* cdof_dot;         // time-derivative of cdof (rot:lin)             (nv x 6)

// computed by mj_fwdVelocity/mj_rne (without acceleration)
mjtNum* qfrc_bias;        // C(qpos,qvel)                                  (nv x 1)

// computed by mj_fwdVelocity/mj_passive
mjtNum* qfrc_spring;      // passive spring force                          (nv x 1)
mjtNum* qfrc_damper;      // passive damper force                          (nv x 1)
mjtNum* qfrc_gravcomp;    // passive gravity compensation force            (nv x 1)
mjtNum* qfrc_fluid;       // passive fluid force                           (nv x 1)
mjtNum* qfrc_passive;     // total passive force                           (nv x 1)

// computed by mj_sensorVel/mj_subtreeVel if needed
mjtNum* subtree_linvel;   // linear velocity of subtree com                (nbody x 3)
mjtNum* subtree_angmom;   // angular momentum about subtree com            (nbody x 3)

// computed by mj_Euler or mj_implicit
mjtNum* qH;               // L'*D*L factorization of modified M            (nM x 1)
mjtNum* qHDiagInv;        // 1/diag(D) of modified M                       (nv x 1)

// computed by mj_resetData
int*    B_rownnz;         // body-dof: non-zeros in each row               (nbody x 1)
int*    B_rowadr;         // body-dof: address of each row in B_colind     (nbody x 1)
int*    B_colind;         // body-dof: column indices of non-zeros         (nB x 1)
int*    M_rownnz;         // inertia: non-zeros in each row                (nv x 1)
int*    M_rowadr;         // inertia: address of each row in M_colind      (nv x 1)
int*    M_colind;         // inertia: column indices of non-zeros          (nM x 1)
int*    mapM2M;           // index mapping from M (legacy) to M (CSR)      (nM x 1)
int*    C_rownnz;         // reduced dof-dof: non-zeros in each row        (nv x 1)
int*    C_rowadr;         // reduced dof-dof: address of each row in C_colind (nv x 1)
int*    C_colind;         // reduced dof-dof: column indices of non-ze
int*    mapM2C;           // index mapping from M to C
int*    D_rownnz;         // dof-dof: non-zeros in each row                (nv x 1)
int*    D_rowadr;         // dof-dof: address of each row in D_colind      (nv x 1)
```

```
int*    D_diag;            // dof-dof: index of diagonal element               (nv x 1)
int*    D_colind;          // dof-dof: column indices of non-zeros             (nD x 1)
int*    mapM2D;            // index mapping from M to D                        (nD x 1)
int*    mapD2M;            // index mapping from D to M                        (nM x 1)

// computed by mj_implicit/mj_derivative
mjtNum* qDeriv;            // d (passive + actuator - bias) / d qvel           (nD x 1)

// computed by mj_implicit/mju_factorLUSparse
mjtNum* qLU;               // sparse LU of (qM - dt*qDeriv)                    (nD x 1)

//-------------------- POSITION, VELOCITY, CONTROL/ACCELERATION dependent

// computed by mj_fwdActuation
mjtNum* actuator_force;    // actuator force in actuation space                (nu x 1)
mjtNum* qfrc_actuator;     // actuator force                                   (nv x 1)

// computed by mj_fwdAcceleration
mjtNum* qfrc_smooth;       // net unconstrained force                          (nv x 1)
mjtNum* qacc_smooth;       // unconstrained acceleration                       (nv x 1)

// computed by mj_fwdConstraint/mj_inverse
mjtNum* qfrc_constraint;   // constraint force                                 (nv x 1)

// computed by mj_inverse
mjtNum* qfrc_inverse;      // net external force; should equal:
                           // qfrc_applied + J'*xfrc_applied + qfrc_actuator   (nv x 1)

// computed by mj_sensorAcc/mj_rnePostConstraint if needed; rotation:translation format
mjtNum* cacc;              // com-based acceleration                           (nbody x 6)
mjtNum* cfrc_int;          // com-based interaction force with parent          (nbody x 6)
mjtNum* cfrc_ext;          // com-based external force on body                 (nbody x 6)

//-------------------- arena-allocated: POSITION dependent

// computed by mj_collision
mjContact* contact;        // array of all detected contacts                   (ncon x 1)

// computed by mj_makeConstraint
int*    efc_type;          // constraint type (mjtConstraint)                  (nefc x 1)
int*    efc_id;            // id of object of specified type                   (nefc x 1)
int*    efc_J_rownnz;      // number of non-zeros in constraint Jacobian row   (nefc x 1)
int*    efc_J_rowadr;      // row start address in colind array                (nefc x 1)
int*    efc_J_rowsuper;    // number of subsequent rows in supernode           (nefc x 1)
int*    efc_J_colind;      // column indices in constraint Jacobian            (nJ x 1)
int*    efc_JT_rownnz;     // number of non-zeros in constraint Jacobian row T (nv x 1)
int*    efc_JT_rowadr;     // row start address in colind array
int*    efc_JT_rowsuper;   // number of subsequent rows in supernode
int*    efc_JT_colind;     // column indices in constraint Jacobian          T (nJ x 1)
mjtNum* efc_J;             // constraint Jacobian                              (nJ x 1)
```

stable

```
  mjtNum* efc_JT;            // constraint Jacobian transposed                   (nJ x 1)
  mjtNum* efc_pos;           // constraint position (equality, contact)          (nefc x 1)
  mjtNum* efc_margin;        // inclusion margin (contact)                       (nefc x 1)
  mjtNum* efc_frictionloss;  // frictionloss (friction)                          (nefc x 1)
  mjtNum* efc_diagApprox;    // approximation to diagonal of A                   (nefc x 1)
  mjtNum* efc_KBIP;          // stiffness, damping, impedance, imp'              (nefc x 4)
  mjtNum* efc_D;             // constraint mass                                  (nefc x 1)
  mjtNum* efc_R;             // inverse constraint mass                          (nefc x 1)
  int*    tendon_efcadr;     // first efc address involving tendon; -1: none     (ntendon x

  // computed by mj_island
  int*    dof_island;        // island id of this dof; -1: none                  (nv x 1)
  int*    island_dofnum;     // number of dofs in island                         (nisland x
  int*    island_dofadr;     // start address in island_dofind                   (nisland x
  int*    island_dofind;     // island dof indices; -1: none                     (nv x 1)
  int*    dof_islandind;     // dof island indices; -1: none                     (nv x 1)
  int*    efc_island;        // island id of this constraint                     (nefc x 1)
  int*    island_efcnum;     // number of constraints in island                  (nisland x
  int*    island_efcadr;     // start address in island_efcind                   (nisland x
  int*    island_efcind;     // island constraint indices                        (nefc x 1)

  // computed by mj_projectConstraint (PGS solver)
  int*    efc_AR_rownnz;     // number of non-zeros in AR                        (nefc x 1)
  int*    efc_AR_rowadr;     // row start address in colind array                (nefc x 1)
  int*    efc_AR_colind;     // column indices in sparse AR                      (nA x 1)
  mjtNum* efc_AR;            // J*inv(M)*J' + R                                  (nA x 1)

  //-------------------- arena-allocated: POSITION, VELOCITY dependent

  // computed by mj_fwdVelocity/mj_referenceConstraint
  mjtNum* efc_vel;           // velocity in constraint space: J*qvel             (nefc x 1)
  mjtNum* efc_aref;          // reference pseudo-acceleration                    (nefc x 1)

  //-------------------- arena-allocated: POSITION, VELOCITY, CONTROL/ACCELERATION dependent

  // computed by mj_fwdConstraint/mj_inverse
  mjtNum* efc_b;             // linear cost term: J*qacc_smooth - aref           (nefc x 1)
  mjtNum* efc_force;         // constraint force in constraint space             (nefc x 1)
  int*    efc_state;         // constraint state (mjtConstraintState)            (nefc x 1)

  // thread pool pointer
  uintptr_t threadpool;

  // compilation signature
  uint64_t  signature;       // also held by the mjSpec that compiled the model
};
typedef struct mjData_ mjData;
```

Auxiliary

These struct types are used in the engine and their names are prefixed with `mj`. mjVisual and mjStatistic are embedded in mjModel, mjContact is embedded in mjData, and mjVFS is a library-level struct used for loading assets.

mjVisual

This is the data structure with abstract visualization options. It corresponds to the MJCF element visual. One instance of it is embedded in mjModel.

```
struct mjVisual_ {                // visualization options
  struct {                        // global parameters
    int orthographic;             // is the free camera orthographic (0: no, 1: yes)
    float fovy;                   // y field-of-view of free camera (orthographic ? length :
    float ipd;                    // inter-pupilary distance for free camera
    float azimuth;                // initial azimuth of free camera (degrees)
    float elevation;              // initial elevation of free camera (degrees)
    float linewidth;              // line width for wireframe and ray rendering
    float glow;                   // glow coefficient for selected body
    float realtime;               // initial real-time factor (1: real time)
    int   offwidth;               // width of offscreen buffer
    int   offheight;              // height of offscreen buffer
    int   ellipsoidinertia;       // geom for inertia visualization (0: box, 1: ellipsoid)
    int   bvactive;               // visualize active bounding volumes (0: no, 1: yes)
  } global;

  struct {                        // rendering quality
    int   shadowsize;             // size of shadowmap texture
    int   offsamples;             // number of multisamples for offscreen rendering
    int   numslices;              // number of slices for builtin geom drawing
    int   numstacks;              // number of stacks for builtin geom drawing
    int   numquads;               // number of quads for box rendering
  } quality;

  struct {                        // head light
    float ambient[3];             // ambient rgb (alpha=1)
    float diffuse[3];             // diffuse rgb (alpha=1)
    float specular[3];            // specular rgb (alpha=1)
    int   active;                 // is headlight active
  } headlight;

  struct {                        // mapping
    float stiffness;              // mouse perturbation stiffness (space->force)
    float stiffnessrot;           // mouse perturbation stiffness (space->torque)
    float force;                  // from force units to space units
    float torque;                 // from torque units to space units
    float alpha;                  // scale geom alphas when transparency
    float fogstart;               // OpenGL fog starts at fogstart * mjMc
    float fogend;                 // OpenGL fog ends at fogend * mjModel.stat.extent
    float znear;                  // near clipping plane = znear * mjModel.stat.extent
    float zfar;                   // far clipping plane = zfar * mjModel.stat.extent
    float haze;                   // haze ratio
    float shadowclip;             // directional light: shadowclip * mjModel.stat.extent
    float shadowscale;            // spot light: shadowscale * light.cutoff
    float actuatortendon;         // scale tendon width
  } map;

  struct {                        // scale of decor elements relative to mean body size
    float forcewidth;             // width of force arrow
    float contactwidth;           // contact width
    float contactheight;          // contact height
    float connect;                // autoconnect capsule width
    float com;                    // com radius
    float camera;                 // camera object
    float light;                  // light object
    float selectpoint;            // selection point
    float jointlength;            // joint length
    float jointwidth;             // joint width
    float actuatorlength;         // actuator length
    float actuatorwidth;          // actuator width
    float framelength;            // bodyframe axis length
    float framewidth;             // bodyframe axis width
    float constraint;             // constraint width
    float slidercrank;            // slidercrank width
    float frustum;                // frustum zfar plane
  } scale;

  struct {                        // color of decor elements
    float fog[4];                 // fog
    float haze[4];                // haze
    float force[4];               // external force
    float inertia[4];             // inertia box
    float joint[4];               // joint
    float actuator[4];            // actuator, neutral
    float actuatornegative[4];    // actuator, negative limit
    float actuatorpositive[4];    // actuator, positive limit
    float com[4];                 // center of mass
    float camera[4];              // camera object
    float light[4];               // light object
    float selectpoint[4];         // selection point
    float connect[4];             // auto connect
    float contactpoint[4];        // contact point
    float contactforce[4];        // contact force
    float contactfriction[4];     // contact friction force
    float contacttorque[4];       // contact torque
    float contactgap[4];          // contact point in gap
    float rangefinder[4];         // rangefinder ray
    float constraint[4];          // constraint
    float slidercrank[4];         // slidercrank
    float crankbroken[4];         // used when crank must be stretched/broken
    float frustum[4];             // camera frustum
    float bv[4];                  // bounding volume
    float bvactive[4];            // active bounding volume
  } rgba;
};
typedef struct mjVisual_ mjVisual;
```

mjStatistic

This is the data structure with model statistics precomputed by the compiler or set by the user. It corresponds to the MJCF element statistic. One instance of it is embedded in mjModel.

```
struct mjStatistic_ {             // model statistics (in qpos0)
  mjtNum meaninertia;             // mean diagonal inertia
  mjtNum meanmass;                // mean body mass
  mjtNum meansize;                // mean body size
  mjtNum extent;                  // spatial extent
  mjtNum center[3];               // center of model
};
typedef struct mjStatistic_ mjStatistic;
```

mjContact

This is the data structure holding information about one contact. `mjData.contact` is a preallocated array of mjContact data structures, populated at runtime with the contacts found by the collision detector. Additional contact information is then filled-in by the simulator.

```
struct mjContact_ {                // result of collision detection functions
  // contact parameters set by near-phase collision function
  mjtNum  dist;                    // distance between nearest points; neg: penetration
  mjtNum  pos[3];                  // position of contact point: midpoint between geoms
  mjtNum  frame[9];                // normal is in [0-2], points from geom[0] to geom[1]

  // contact parameters set by mj_collideGeoms
  mjtNum  includemargin;           // include if dist<includemargin=margin-gap
  mjtNum  friction[5];             // tangent1, 2, spin, roll1, 2
  mjtNum  solref[mjNREF];          // constraint solver reference, normal direction
  mjtNum  solreffriction[mjNREF];  // constraint solver reference, friction directions
  mjtNum  solimp[mjNIMP];          // constraint solver impedance

  // internal storage used by solver
  mjtNum  mu;                      // friction of regularized cone, set by mj_makeConstraint
  mjtNum  H[36];                   // cone Hessian, set by mj_constraintUpdate

  // contact descriptors set by mj_collideXXX
  int     dim;                     // contact space dimensionality: 1, 3,
  int     geom1;                   // id of geom 1; deprecated, use geom[0]
  int     geom2;                   // id of geom 2; deprecated, use geom[1]
```

stable

```
  int     geom[2];                 // geom ids; -1 for flex
  int     flex[2];                 // flex ids; -1 for geom
  int     elem[2];                 // element ids; -1 for geom or flex vertex
  int     vert[2];                 // vertex ids;  -1 for geom or flex element

  // flag set by mj_setContact or mj_instantiateContact
  int     exclude;                 // 0: include, 1: in gap, 2: fused, 3: no dofs

  // address computed by mj_instantiateContact
  int     efc_address;             // address in efc; -1: not included
};
typedef struct mjContact_ mjContact;
```

mjResource

A resource is an abstraction of a file in a filesystem. The name field is the unique name of the resource while the other fields are populated by a resource provider.

```
struct mjResource_ {
  char* name;                                   // name of resource (filename, etc)
  void* data;                                   // opaque data pointer
  char timestamp[512];                          // timestamp of the resource
  const struct mjpResourceProvider* provider;   // pointer to the provider
};
typedef struct mjResource_ mjResource;
```

mjVFS

This is the data structure of the virtual file system. It can only be constructed programmatically, and does not have an analog in MJCF.

```
struct mjVFS_ {                               // virtual file system for loading from memory
  void* impl_;                                // internal pointer to VFS memory
};
typedef struct mjVFS_ mjVFS;
```

mjLROpt

Options for configuring the automatic actuator length-range computation.

```
struct mjLROpt_ {                 // options for mj_setLengthRange()
  // flags
  int mode;                       // which actuators to process (mjtLRMode)
  int useexisting;                // use existing length range if available
  int uselimit;                   // use joint and tendon limits if available

  // algorithm parameters
  mjtNum accel;                   // target acceleration used to compute
  mjtNum maxforce;                // maximum force; 0: no limit
  mjtNum timeconst;               // time constant for velocity reduction; min 0.01
  mjtNum timestep;                // simulation timestep; 0: use mjOption.timestep
  mjtNum inttotal;                // total simulation time interval
  mjtNum interval;                // evaluation time interval (at the end)
  mjtNum tolrange;                // convergence tolerance (relative to range)
};
typedef struct mjLROpt_ mjLROpt;
```

mjTask

This is a representation of a task to be run asynchronously inside of an mjThreadPool . It is created in the mju_threadPoolEnqueue method of the mjThreadPool and is used to join the task at completion.

```
struct mjTask_ {        // a task that can be executed by a thread pool.
  mjfTask func;         // pointer to the function that implements the task
  void* args;           // arguments to func
  volatile int status;  // status of the task
};
typedef struct mjTask_ mjTask;
```

mjThreadPool

This is the data structure of the threadpool. It can only be constructed programmatically, and does not have an analog in MJCF. In order to enable multi-threaded calculations, a pointer to an existing mjThreadPool should be assigned to the `mjData.threadpool`.

```
struct mjThreadPool_ {
  int nworker;  // number of workers in the pool
};
typedef struct mjThreadPool_ mjThreadPool;
```

Sim statistics

These structs are all embedded in mjData, and collect simulation-related statistics.

mjWarningStat

This is the data structure holding information about one warning type. `mjData.warning` is a preallocated array of mjWarningStat data structures, one for each warning type.

```
struct mjWarningStat_ {      // warning statistics
  int     lastinfo;          // info from last warning
  int     number;            // how many times was warning raised
};
typedef struct mjWarningStat_ mjWarningStat;
```

mjTimerStat

This is the data structure holding information about one timer. `mjData.timer` is a preallocated array of mjTimerStat data structures, one for each timer type.

```
struct mjTimerStat_ {         // timer statistics
  mjtNum  duration;           // cumulative duration
  int     number;             // how many times was timer called
};
typedef struct mjTimerStat_ mjTimerStat;
```

mjSolverStat

This is the data structure holding information about one solver iteration. `mjData.solver` is a preallocated array of mjSolverStat data structures, one for each iteration of the solver, up to a maximum of mjNSOLVER. The actual number of solver iterations is given by `mjData.solver_iter`.

```
struct mjSolverStat_ {        // per-iteration solver statistics
  mjtNum  improvement;        // cost reduction, scaled by 1/trace(M(qpos0))
  mjtNum  gradient;           // gradient norm (primal only, scaled)
  mjtNum  lineslope;          // slope in linesearch
  int     nactive;            // number of active constraints
  int     nchange;            // number of constraint state changes
  int     neval;              // number of cost evaluations in line search
  int     nupdate;            // number of Cholesky updates in line search
};
typedef struct mjSolverStat_ mjSolverStat;
```

Visualisation

The names of these struct types are prefixed with `mjv`.

mjvPerturb

This is the data structure holding information about mouse perturbations.

```
struct mjvPerturb_ {               // object selection and perturbation
  int      select;                 // selected body id; non-positive: none
  int      flexselect;             // selected flex id; negative: none
  int      skinselect;             // selected skin id; negative: none
  int      active;                 // perturbation bitmask (mjtPertBit)
  int      active2;                // secondary perturbation bitmask (mjtPertBit)
  mjtNum   refpos[3];              // reference position for selected object
  mjtNum   refquat[4];             // reference orientation for selected object
  mjtNum   refselpos[3];           // reference position for selection point
  mjtNum   localpos[3];            // selection point in object coordinates
  mjtNum   localmass;              // spatial inertia at selection point
  mjtNum   scale;                  // relative mouse motion-to-space scali
};
typedef struct mjvPerturb_ mjvPerturb;
```

stable

mjvCamera

This is the data structure describing one abstract camera.

```
struct mjvCamera_ {                 // abstract camera
  // type and ids
  int       type;                   // camera type (mjtCamera)
  int       fixedcamid;             // fixed camera id
  int       trackbodyid;            // body id to track

  // abstract camera pose specification
  mjtNum    lookat[3];              // lookat point
  mjtNum    distance;               // distance to lookat point or tracked body
  mjtNum    azimuth;                // camera azimuth (deg)
  mjtNum    elevation;              // camera elevation (deg)

  // orthographic / perspective
  int       orthographic;           // 0: perspective; 1: orthographic
};
typedef struct mjvCamera_ mjvCamera;
```

mjvGLCamera

This is the data structure describing one OpenGL camera.

```
struct mjvGLCamera_ {               // OpenGL camera
  // camera frame
  float     pos[3];                 // position
  float     forward[3];             // forward direction
  float     up[3];                  // up direction

  // camera projection
  float     frustum_center;         // hor. center (left,right set to match aspect)
  float     frustum_width;          // width (not used for rendering)
  float     frustum_bottom;         // bottom
  float     frustum_top;            // top
  float     frustum_near;           // near
  float     frustum_far;            // far

  // orthographic / perspective
  int       orthographic;           // 0: perspective; 1: orthographic
};
typedef struct mjvGLCamera_ mjvGLCamera;
```

mjvGeom

This is the data structure describing one abstract visualization geom – which could correspond to a model geom or to a decoration element constructed b

stable

```
struct mjvGeom_ {                   // abstract geom
  // type info
  int      type;                   // geom type (mjtGeom)
  int      dataid;                 // mesh, hfield or plane id; -1: none
  int      objtype;                // mujoco object type; mjOBJ_UNKNOWN for decor
  int      objid;                  // mujoco object id; -1 for decor
  int      category;               // visual category
  int      matid;                  // material id; -1: no textured material
  int      texcoord;               // mesh or flex geom has texture coordinates
  int      segid;                  // segmentation id; -1: not shown

  // spatial transform
  float    size[3];                // size parameters
  float    pos[3];                 // Cartesian position
  float    mat[9];                 // Cartesian orientation

  // material properties
  float    rgba[4];                // color and transparency
  float    emission;               // emission coef
  float    specular;               // specular coef
  float    shininess;              // shininess coef
  float    reflectance;            // reflectance coef

  char     label[100];             // text label

  // transparency rendering (set internally)
  float    camdist;                // distance to camera (used by sorter)
  float    modelrbound;            // geom rbound from model, 0 if not model geom
  mjtByte  transparent;            // treat geom as transparent
};
typedef struct mjvGeom_ mjvGeom;
```

mjvLight

This is the data structure describing one OpenGL light.

```
struct mjvLight_ {                 // OpenGL light
  float    pos[3];                 // position rel. to body frame
  float    dir[3];                 // direction rel. to body frame
  float    attenuation[3];         // OpenGL attenuation (quadratic model)
  float    cutoff;                 // OpenGL cutoff
  float    exponent;               // OpenGL exponent
  float    ambient[3];             // ambient rgb (alpha=1)
  float    diffuse[3];             // diffuse rgb (alpha=1)
  float    specular[3];            // specular rgb (alpha=1)
  mjtByte  headlight;              // headlight
  mjtByte  directional;            // directional light
  mjtByte  castshadow;             // does light cast shadows
  float    bulbradius;             // bulb radius for soft shadows
};
typedef struct mjvLight_ mjvLight;
```

stable

mjvOption

This structure contains options that enable and disable the visualization of various elements.

```
struct mjvOption_ {                  // abstract visualization options
  int      label;                    // what objects to label (mjtLabel)
  int      frame;                    // which frame to show (mjtFrame)
  mjtByte  geomgroup[mjNGROUP];      // geom visualization by group
  mjtByte  sitegroup[mjNGROUP];      // site visualization by group
  mjtByte  jointgroup[mjNGROUP];     // joint visualization by group
  mjtByte  tendongroup[mjNGROUP];    // tendon visualization by group
  mjtByte  actuatorgroup[mjNGROUP];  // actuator visualization by group
  mjtByte  flexgroup[mjNGROUP];      // flex visualization by group
  mjtByte  skingroup[mjNGROUP];      // skin visualization by group
  mjtByte  flags[mjNVISFLAG];        // visualization flags (indexed by mjtVisFlag)
  int      bvh_depth;                // depth of the bounding volume hierarchy to be visuali
  int      flex_layer;               // element layer to be visualized for 3D flex
};
typedef struct mjvOption_ mjvOption;
```

mjvScene

This structure contains everything needed to render the 3D scene in OpenGL.

```
struct mjvScene_ {                // abstract scene passed to OpenGL renderer
  // abstract geoms
  int      maxgeom;               // size of allocated geom buffer
  int      ngeom;                 // number of geoms currently in buffer
  mjvGeom* geoms;                 // buffer for geoms (ngeom)
  int*     geomorder;             // buffer for ordering geoms by distance to camera (ngeom)

  // flex data
  int      nflex;                 // number of flexes
  int*     flexedgeadr;           // address of flex edges (nflex)
  int*     flexedgenum;           // number of edges in flex (nflex)
  int*     flexvertadr;           // address of flex vertices (nflex)
  int*     flexvertnum;           // number of vertices in flex (nflex)
  int*     flexfaceadr;           // address of flex faces (nflex)
  int*     flexfacenum;           // number of flex faces allocated (nflex)
  int*     flexfaceused;          // number of flex faces currently in use (nflex)
  int*     flexedge;              // flex edge data (2*nflexedge)
  float*   flexvert;              // flex vertices (3*nflexvert)
  float*   flexface;              // flex faces vertices (9*sum(flexfacenum))
  float*   flexnormal;            // flex face normals (9*sum(flexfacenum))
  float*   flextexcoord;          // flex face texture coordinates (6*sum(flexfacenum))
  mjtByte  flexvertopt;           // copy of mjVIS_FLEXVERT mjvOption fla
  mjtByte  flexedgeopt;           // copy of mjVIS_FLEXEDGE mjvOption fla
  mjtByte  flexfaceopt;           // copy of mjVIS_FLEXFACE mjvOption flag
  mjtByte  flexskinopt;           // copy of mjVIS_FLEXSKIN mjvOption flag
```

stable

```
  // skin data
  int       nskin;                  // number of skins
  int*      skinfacenum;            // number of faces in skin (nskin)
  int*      skinvertadr;            // address of skin vertices (nskin)
  int*      skinvertnum;            // number of vertices in skin (nskin)
  float*    skinvert;               // skin vertex data (3*nskinvert)
  float*    skinnormal;             // skin normal data (3*nskinvert)

  // OpenGL lights
  int       nlight;                 // number of lights currently in buffer
  mjvLight lights[mjMAXLIGHT];      // buffer for lights (nlight)

  // OpenGL cameras
  mjvGLCamera camera[2];            // left and right camera

  // OpenGL model transformation
  mjtByte   enabletransform;        // enable model transformation
  float     translate[3];           // model translation
  float     rotate[4];              // model quaternion rotation
  float     scale;                  // model scaling

  // OpenGL rendering effects
  int       stereo;                 // stereoscopic rendering (mjtStereo)
  mjtByte   flags[mjNRNDFLAG];      // rendering flags (indexed by mjtRndFlag)

  // framing
  int       framewidth;             // frame pixel width; 0: disable framing
  float     framergb[3];            // frame color
};
typedef struct mjvScene_ mjvScene;
```

mjvSceneState

This structure contains the portions of mjModel and mjData that are required for various `mjv_*` functions.

```
struct mjvSceneState_ {
  int nbuffer;                      // size of the buffer in bytes
  void* buffer;                     // heap-allocated memory for all arrays in this struct
  int maxgeom;                      // maximum number of mjvGeom supported by this state obje
  mjvScene scratch;                 // scratch space for vis geoms inserted by the user and p

  // fields in mjModel that are necessary to re-render a scene
  struct {
    int nv;
    int nu;
    int na;
    int nbody;
    int nbvh;
```

stable

```
  int nbvhstatic;
  int njnt;
  int ngeom;
  int nsite;
  int ncam;
  int nlight;
  int nmesh;
  int nskin;
  int nflex;
  int nflexvert;
  int nflextexcoord;
  int nskinvert;
  int nskinface;
  int nskinbone;
  int nskinbonevert;
  int nmat;
  int neq;
  int ntendon;
  int ntree;
  int nwrap;
  int nsensor;
  int nnames;
  int npaths;
  int nsensordata;
  int narena;

  mjOption opt;
  mjVisual vis;
  mjStatistic stat;

  int* body_parentid;
  int* body_rootid;
  int* body_weldid;
  int* body_mocapid;
  int* body_jntnum;
  int* body_jntadr;
  int* body_dofnum;
  int* body_dofadr;
  int* body_geomnum;
  int* body_geomadr;
  mjtNum* body_iquat;
  mjtNum* body_mass;
  mjtNum* body_inertia;
  int* body_bvhadr;
  int* body_bvhnum;

  int* bvh_depth;
  int* bvh_child;
  int* bvh_nodeid;
  mjtNum* bvh_aabb;
  int* jnt_type;
  int* jnt_bodyid;
  int* jnt_group;

  int* geom_type;
  int* geom_bodyid;
  int* geom_contype;
  int* geom_conaffinity;
  int* geom_dataid;
  int* geom_matid;
  int* geom_group;
  mjtNum* geom_size;
  mjtNum* geom_aabb;
  mjtNum* geom_rbound;
  float* geom_rgba;

  int* site_type;
  int* site_bodyid;
  int* site_matid;
  int* site_group;
  mjtNum* site_size;
  float* site_rgba;

  int* cam_orthographic;
  mjtNum* cam_fovy;
  mjtNum* cam_ipd;
  int* cam_resolution;
  float* cam_sensorsize;
  float* cam_intrinsic;

  mjtByte* light_directional;
  mjtByte* light_castshadow;
  float* light_bulbradius;
  mjtByte* light_active;
  float* light_attenuation;
  float* light_cutoff;
  float* light_exponent;
  float* light_ambient;
  float* light_diffuse;
  float* light_specular;

  mjtByte* flex_flatskin;
  int* flex_dim;
  int* flex_matid;
  int* flex_group;
  int* flex_interp;
  int* flex_nodeadr;
  int* flex_nodenum;
  int* flex_nodebodyid;
  int* flex_vertadr;
  int* flex_vertnum;
  int* flex_elem;
  int* flex_elemtexcoord;
  int* flex_elemlayer;
  int* flex_elemadr;
  int* flex_elemnum;
  int* flex_elemdataadr;
  int* flex_shell;
  int* flex_shellnum;
  int* flex_shelldataadr;
  int* flex_texcoordadr;
  int* flex_bvhadr;
  int* flex_bvhnum;
  mjtByte* flex_centered;
  mjtNum* flex_node;
  mjtNum* flex_radius;
  float* flex_rgba;
  float* flex_texcoord;

  int* hfield_pathadr;

  int* mesh_bvhadr;
  int* mesh_bvhnum;
  int* mesh_texcoordadr;
  int* mesh_graphadr;
  int* mesh_pathadr;

  int* skin_matid;
  int* skin_group;
  float* skin_rgba;
  float* skin_inflate;
  int* skin_vertadr;
  int* skin_vertnum;
  int* skin_texcoordadr;
  int* skin_faceadr;
  int* skin_facenum;
  int* skin_boneadr;
  int* skin_bonenum;
  float* skin_vert;
  int* skin_face;
  int* skin_bonevertadr;
  int* skin_bonevertnum;
  float* skin_bonebindpos;
  float* skin_bonebindquat;
  int* skin_bonebodyid;
  int* skin_bonevertid;
  float* skin_bonevertweight;
  int* skin_pathadr;
  int* tex_pathadr;

  int* mat_texid;
  mjtByte* mat_texuniform;
  float* mat_texrepeat;
  float* mat_emission;
  float* mat_specular;
  float* mat_shininess;
  float* mat_reflectance;
  float* mat_metallic;
  float* mat_roughness;
  float* mat_rgba;

  int* eq_type;
  int* eq_obj1id;
  int* eq_obj2id;
  int* eq_objtype;
  mjtNum* eq_data;

  int* tendon_num;
  int* tendon_matid;
  int* tendon_group;
  mjtByte* tendon_limited;
  mjtByte* tendon_actfrclimited;
  mjtNum* tendon_width;
  mjtNum* tendon_range;
  mjtNum* tendon_actfrcrange;
  mjtNum* tendon_stiffness;
  mjtNum* tendon_damping;
  mjtNum* tendon_frictionloss;
  mjtNum* tendon_lengthspring;
  float* tendon_rgba;

  int* actuator_trntype;
  int* actuator_dyntype;
  int* actuator_trnid;
  int* actuator_actadr;
  int* actuator_actnum;
  int* actuator_group;
  mjtByte* actuator_ctrllimited;
  mjtByte* actuator_actlimited;
  mjtNum* actuator_ctrlrange;
  mjtNum* actuator_actrange;
  mjtNum* actuator_cranklength;

  int* sensor_type;
  int* sensor_objid;
  int* sensor_adr;

  int* name_bodyadr;
  int* name_jntadr;
  int* name_geomadr;
  int* name_siteadr;
  int* name_camadr;
  int* name_lightadr;
  int* name_eqadr;
  int* name_tendonadr;
  int* name_actuatoradr;
  char* names;
  char* paths;
} model;

// fields in mjData that are necessary to re-render a scene
struct {
  mjWarningStat warning[mjNWARNING];

  int nefc;
  int ncon;
  int nisland;

  mjtNum time;

  mjtNum* act;

  mjtNum* ctrl;
  mjtNum* xfrc_applied;
  mjtByte* eq_active;

  mjtNum* sensordata;

  mjtNum* xpos;
  mjtNum* xquat;
  mjtNum* xmat;
  mjtNum* xipos;
  mjtNum* ximat;
  mjtNum* xanchor;
  mjtNum* xaxis;
  mjtNum* geom_xpos;
  mjtNum* geom_xmat;
  mjtNum* site_xpos;
  mjtNum* site_xmat;
  mjtNum* cam_xpos;
  mjtNum* cam_xmat;
  mjtNum* light_xpos;
  mjtNum* light_xdir;

  mjtNum* subtree_com;

  int* ten_wrapadr;
  int* ten_wrapnum;
```

stable

```
    int* wrap_obj;
    mjtNum* ten_length;
    mjtNum* wrap_xpos;

    mjtNum* bvh_aabb_dyn;
    mjtByte* bvh_active;
    int* island_dofadr;
    int* island_dofind;
    int* dof_island;
    int* efc_island;
    int* tendon_efcadr;

    mjtNum* flexvert_xpos;

    mjContact* contact;
    mjtNum* efc_force;
    void* arena;
  } data;
};
typedef struct mjvSceneState_ mjvSceneState;
```

mjvFigure

This structure contains everything needed to render a 2D plot in OpenGL. The buffers for line points etc. are preallocated, and the user has to populate them before calling the function mjr_figure with this data structure as an argument.

```
struct mjvFigure_ {                  // abstract 2D figure passed to OpenGL renderer
  // enable flags
  int     flg_legend;                // show legend
  int     flg_ticklabel[2];          // show grid tick labels (x,y)
  int     flg_extend;                // automatically extend axis ranges to fit data
  int     flg_barplot;               // isolated line segments (i.e. GL_LINES)
  int     flg_selection;             // vertical selection line
  int     flg_symmetric;             // symmetric y-axis

  // style settings
  float   linewidth;                 // line width
  float   gridwidth;                 // grid line width
  int     gridsize[2];               // number of grid points in (x,y)
  float   gridrgb[3];                // grid line rgb
  float   figurergba[4];             // figure color and alpha
  float   panergba[4];               // pane color and alpha
  float   legendrgba[4];             // legend color and alpha
  float   textrgb[3];                // text color
  float   linergb[mjMAXLINE][3];     // line colors
  float   range[2][2];               // axis ranges; (min>=max) automatic
  char    xformat[20];               // x-tick label format for sprintf
  char    yformat[20];               // y-tick label format for sprintf
  char    minwidth[20];              // string used to determine min y-tick width
```

stable

```
  // text labels
  char    title[1000];            // figure title; subplots separated with 2+ spaces
  char    xlabel[100];            // x-axis label
  char    linename[mjMAXLINE][100];  // line names for legend

  // dynamic settings
  int     legendoffset;           // number of lines to offset legend
  int     subplot;                // selected subplot (for title rendering)
  int     highlight[2];           // if point is in legend rect, highlight line
  int     highlightid;            // if id>=0 and no point, highlight id
  float   selection;              // selection line x-value

  // line data
  int     linepnt[mjMAXLINE];     // number of points in line; (0) disable
  float   linedata[mjMAXLINE][2*mjMAXLINEPNT];  // line data (x,y)

  // output from renderer
  int     xaxispixel[2];          // range of x-axis in pixels
  int     yaxispixel[2];          // range of y-axis in pixels
  float   xaxisdata[2];           // range of x-axis in data units
  float   yaxisdata[2];           // range of y-axis in data units
};
typedef struct mjvFigure_ mjvFigure;
```

Rendering

The names of these struct types are prefixed with `mjr`.

mjrRect

This structure specifies a rectangle.

```
struct mjrRect_ {                 // OpenGL rectangle
  int left;                       // left (usually 0)
  int bottom;                     // bottom (usually 0)
  int width;                      // width (usually buffer width)
  int height;                     // height (usually buffer height)
};
typedef struct mjrRect_ mjrRect;
```

mjrContext

This structure contains the custom OpenGL rendering context, with the ids of all OpenGL resources uploaded to the GPU.

```
struct mjrContext_ {                // custom OpenGL context
  // parameters copied from mjVisual
  float lineWidth;                  // line width for wireframe rendering
  float shadowClip;                 // clipping radius for directional lights
```

stable

```
float shadowScale;                // fraction of light cutoff for spot lights
float fogStart;                   // fog start = stat.extent * vis.map.fogstart
float fogEnd;                     // fog end = stat.extent * vis.map.fogend
float fogRGBA[4];                 // fog rgba
int shadowSize;                   // size of shadow map texture
int offWidth;                     // width of offscreen buffer
int offHeight;                    // height of offscreen buffer
int offSamples;                   // number of offscreen buffer multisamples

// parameters specified at creation
int fontScale;                    // font scale
int auxWidth[mjNAUX];             // auxiliary buffer width
int auxHeight[mjNAUX];            // auxiliary buffer height
int auxSamples[mjNAUX];           // auxiliary buffer multisamples

// offscreen rendering objects
unsigned int offFBO;              // offscreen framebuffer object
unsigned int offFBO_r;            // offscreen framebuffer for resolving multisamples
unsigned int offColor;            // offscreen color buffer
unsigned int offColor_r;          // offscreen color buffer for resolving multisamples
unsigned int offDepthStencil;     // offscreen depth and stencil buffer
unsigned int offDepthStencil_r;   // offscreen depth and stencil buffer for multisamples

// shadow rendering objects
unsigned int shadowFBO;           // shadow map framebuffer object
unsigned int shadowTex;           // shadow map texture

// auxiliary buffers
unsigned int auxFBO[mjNAUX];      // auxiliary framebuffer object
unsigned int auxFBO_r[mjNAUX];    // auxiliary framebuffer object for resolving
unsigned int auxColor[mjNAUX];    // auxiliary color buffer
unsigned int auxColor_r[mjNAUX];  // auxiliary color buffer for resolving

// materials with textures
int mat_texid[mjMAXMATERIAL*mjNTEXROLE]; // material texture ids (-1: no texture)
int mat_texuniform[mjMAXMATERIAL];       // uniform cube mapping
float mat_texrepeat[mjMAXMATERIAL*2];    // texture repetition for 2d mapping

// texture objects and info
int ntexture;                            // number of allocated textures
int textureType[mjMAXTEXTURE];           // type of texture (mjtTexture) (ntexture)
unsigned int texture[mjMAXTEXTURE];      // texture names

// displaylist starting positions
unsigned int basePlane;           // all planes from model
unsigned int baseMesh;            // all meshes from model
unsigned int baseHField;          // all height fields from model
unsigned int baseBuiltin;         // all builtin geoms, with quality fr
unsigned int baseFontNormal;      // normal font
unsigned int baseFontShadow;      // shadow font
```

stable

```
  unsigned int baseFontBig;         // big font

  // displaylist ranges
  int rangePlane;                   // all planes from model
  int rangeMesh;                    // all meshes from model
  int rangeHField;                  // all hfields from model
  int rangeBuiltin;                 // all builtin geoms, with quality from model
  int rangeFont;                    // all characters in font

  // skin VBOs
  int nskin;                        // number of skins
  unsigned int* skinvertVBO;        // skin vertex position VBOs (nskin)
  unsigned int* skinnormalVBO;      // skin vertex normal VBOs (nskin)
  unsigned int* skintexcoordVBO;    // skin vertex texture coordinate VBOs (nskin)
  unsigned int* skinfaceVBO;        // skin face index VBOs (nskin)

  // character info
  int charWidth[127];               // character widths: normal and shadow
  int charWidthBig[127];            // chacarter widths: big
  int charHeight;                   // character heights: normal and shadow
  int charHeightBig;                // character heights: big

  // capabilities
  int glInitialized;                // is OpenGL initialized
  int windowAvailable;              // is default/window framebuffer available
  int windowSamples;                // number of samples for default/window framebuffer
  int windowStereo;                 // is stereo available for default/window framebuffer
  int windowDoublebuffer;           // is default/window framebuffer double buffered

  // framebuffer
  int currentBuffer;                // currently active framebuffer: mjFB_WINDOW or mjFB_OFF

  // pixel output format
  int readPixelFormat;              // default color pixel format for mjr_readPixels

  // depth output format
  int readDepthMap;                 // depth mapping: mjDEPTH_ZERONEAR or mjDEPTH_ZEROFAR
};
typedef struct mjrContext_ mjrContext;
```

User Interface

For a high-level description of the UI framework, see User Interface. The names of these struct types are prefixed with `mjui`, except for the main mjUI struct itself.

mjuiState

This C struct represents the global state of the window, keyboard and mouse, input event descriptors, and all window rectangles (including the visible UI rectangles). There

stable

is only one `mjuiState` per application, even if there are multiple UIs. This struct would normally be defined as a global variable.

```
struct mjuiState_ {                // mouse and keyboard state
  // constants set by user
  int nrect;                       // number of rectangles used
  mjrRect rect[mjMAXUIRECT];       // rectangles (index 0: entire window)
  void* userdata;                  // pointer to user data (for callbacks)

  // event type
  int type;                        // (type mjtEvent)

  // mouse buttons
  int left;                        // is left button down
  int right;                       // is right button down
  int middle;                      // is middle button down
  int doubleclick;                 // is last press a double click
  int button;                      // which button was pressed (mjtButton)
  double buttontime;               // time of last button press

  // mouse position
  double x;                        // x position
  double y;                        // y position
  double dx;                       // x displacement
  double dy;                       // y displacement
  double sx;                       // x scroll
  double sy;                       // y scroll

  // keyboard
  int control;                     // is control down
  int shift;                       // is shift down
  int alt;                         // is alt down
  int key;                         // which key was pressed
  double keytime;                  // time of last key press

  // rectangle ownership and dragging
  int mouserect;                   // which rectangle contains mouse
  int dragrect;                    // which rectangle is dragged with mouse
  int dragbutton;                  // which button started drag (mjtButton)

  // files dropping (only valid when type == mjEVENT_FILESDROP)
  int dropcount;                   // number of files dropped
  const char** droppaths;          // paths to files dropped
};
typedef struct mjuiState_ mjuiState;
```

mjuiThemeSpacing

stable

This structure defines the spacing of UI items in the theme.

```
struct mjuiThemeSpacing_ {        // UI visualization theme spacing
  int total;                      // total width
  int scroll;                     // scrollbar width
  int label;                      // label width
  int section;                    // section gap
  int cornersect;                 // corner radius for section
  int cornersep;                  // corner radius for separator
  int itemside;                   // item side gap
  int itemmid;                    // item middle gap
  int itemver;                    // item vertical gap
  int texthor;                    // text horizontal gap
  int textver;                    // text vertical gap
  int linescroll;                 // number of pixels to scroll
  int samples;                    // number of multisamples
};
typedef struct mjuiThemeSpacing_ mjuiThemeSpacing;
```

mjuiThemeColor

This structure defines the colors of UI items in the theme.

```
struct mjuiThemeColor_ {          // UI visualization theme color
  float master[3];                // master background
  float thumb[3];                 // scrollbar thumb
  float secttitle[3];             // section title
  float secttitle2[3];            // section title: bottom color
  float secttitleuncheck[3];      // section title with unchecked box
  float secttitleuncheck2[3];     // section title with unchecked box: bottom color
  float secttitlecheck[3];        // section title with checked box
  float secttitlecheck2[3];       // section title with checked box: bottom color
  float sectfont[3];              // section font
  float sectsymbol[3];            // section symbol
  float sectpane[3];              // section pane
  float separator[3];             // separator title
  float separator2[3];            // separator title: bottom color
  float shortcut[3];              // shortcut background
  float fontactive[3];            // font active
  float fontinactive[3];          // font inactive
  float decorinactive[3];         // decor inactive
  float decorinactive2[3];        // inactive slider color 2
  float button[3];                // button
  float check[3];                 // check
  float radio[3];                 // radio
  float select[3];                // select
  float select2[3];               // select pane
  float slider[3];                // slider
  float slider2[3];               // slider color 2
  float edit[3];                  // edit
  float edit2[3];                 // edit invalid
  float cursor[3];                // edit cursor
};
typedef struct mjuiThemeColor_ mjuiThemeColor;
```

mjuiItem

This structure defines one UI item.

```
struct mjuiItemSingle_ {          // check and button-related
  int modifier;                   // 0: none, 1: control, 2: shift; 4: alt
  int shortcut;                   // shortcut key; 0: undefined
};


struct mjuiItemMulti_ {                  // static, radio and select-related
  int nelem;                             // number of elements in group
  char name[mjMAXUIMULTI][mjMAXUINAME];  // element names
};


struct mjuiItemSlider_ {          // slider-related
  double range[2];                // slider range
  double divisions;               // number of range divisions
};


struct mjuiItemEdit_ {            // edit-related
  int nelem;                      // number of elements in list
  double range[mjMAXUIEDIT][2];   // element range (min>=max: ignore)
};


struct mjuiItem_ {                // UI item
  // common properties
  int type;                       // type (mjtItem)
  char name[mjMAXUINAME];         // name
  int state;                      // 0: disable, 1: enable, 2+: use predicate
  void *pdata;                    // data pointer (type-specific)
  int sectionid;                  // id of section containing item
  int itemid;                     // id of item within section
  int userid;                     // user-supplied id (for event handling)

  // type-specific properties
  union {
    struct mjuiItemSingle_ single;  // check and button
    struct mjuiItemMulti_ multi;    // static, radio and select
    struct mjuiItemSlider_ slider;  // slider
    struct mjuiItemEdit_ edit;      // edit
  };

  // internal
```

stable

```
  mjrRect rect;                 // rectangle occupied by item
  int skip;                     // item skipped due to closed separator
};
typedef struct mjuiItem_ mjuiItem;
```

mjuiSection

This structure defines one section of the UI.

```
struct mjuiSection_ {           // UI section
  // properties
  char name[mjMAXUINAME];       // name
  int state;                    // section state (mjtSection)
  int modifier;                 // 0: none, 1: control, 2: shift; 4: alt
  int shortcut;                 // shortcut key; 0: undefined
  int checkbox;                 // 0: none, 1: unchecked, 2: checked
  int nitem;                    // number of items in use
  mjuiItem item[mjMAXUIITEM];   // preallocated array of items

  // internal
  mjrRect rtitle;               // rectangle occupied by title
  mjrRect rcontent;             // rectangle occupied by content
  int lastclick;                // last mouse click over this section
};
typedef struct mjuiSection_ mjuiSection;
```

mjuiDef

This structure defines one entry in the definition table used for simplified UI construction. It contains everything needed to define one UI item. Some translation is performed by the helper functions, so that multiple mjuiDefs can be defined as a static table.

```
struct mjuiDef_ {               // table passed to mjui_add()
  int type;                     // type (mjtItem); -1: section
  char name[mjMAXUINAME];       // name
  int state;                    // state
  void* pdata;                  // pointer to data
  char other[mjMAXUITEXT];      // string with type-specific properties
  int otherint;                 // int with type-specific properties
};
typedef struct mjuiDef_ mjuiDef;
```

mjUI

This C struct represents an entire UI. The same application could have multiple UIs, for example on the left and the right of the window. This would normally b global variable. As explained earlier, it contains static allocation for a m of supported UI sections (mjuiSection) each with a maximum number of supported

stable

items (mjuiItem). It also contains the color and spacing themes, enable/disable callback, virtual window descriptor, text edit state, mouse focus. Some of these fields are set only once when the UI is initialized, others change at runtime.

```
struct mjUI_ {                   // entire UI
  // constants set by user
  mjuiThemeSpacing spacing;      // UI theme spacing
  mjuiThemeColor color;          // UI theme color
  mjfItemEnable predicate;       // callback to set item state programmatically
  void* userdata;                // pointer to user data (passed to predicate)
  int rectid;                    // index of this ui rectangle in mjuiState
  int auxid;                     // aux buffer index of this ui
  int radiocol;                  // number of radio columns (0 defaults to 2)

  // UI sizes (framebuffer units)
  int width;                     // width
  int height;                    // current height
  int maxheight;                 // height when all sections open
  int scroll;                    // scroll from top of UI

  // mouse focus and count
  int mousesect;                 // 0: none, -1: scroll, otherwise 1+section
  int mouseitem;                 // item within section
  int mousehelp;                 // help button down: print shortcuts
  int mouseclicks;               // number of mouse clicks over UI
  int mousesectcheck;            // 0: none, otherwise 1+section

  // keyboard focus and edit
  int editsect;                  // 0: none, otherwise 1+section
  int edititem;                  // item within section
  int editcursor;                // cursor position
  int editscroll;                // horizontal scroll
  char edittext[mjMAXUITEXT];    // current text
  mjuiItem* editchanged;         // pointer to changed edit in last mjui_event

  // sections
  int nsect;                     // number of sections in use
  mjuiSection sect[mjMAXUISECT]; // preallocated array of sections
};
typedef struct mjUI_ mjUI;
```

Model Editing

The structs below are defined in mjspec.h and, with the exception of the top level mjSpec struct, begin with the `mjs` prefix. For more details, see the Moc chapter.

stable

mjSpec

Model specification.

```
typedef struct mjSpec_ {          // model specification
  mjsElement* element;            // element type
  mjString* modelname;            // model name

  // compiler data
  mjsCompiler compiler;           // compiler options
  mjtByte strippath;              // automatically strip paths from mesh files
  mjString* meshdir;              // mesh and hfield directory
  mjString* texturedir;           // texture directory

  // engine data
  mjOption option;                // physics options
  mjVisual visual;                // visual options
  mjStatistic stat;               // statistics override (if defined)

  // sizes
  size_t memory;                  // number of bytes in arena+stack memory
  int nemax;                      // max number of equality constraints
  int nuserdata;                  // number of mjtNums in userdata
  int nuser_body;                 // number of mjtNums in body_user
  int nuser_jnt;                  // number of mjtNums in jnt_user
  int nuser_geom;                 // number of mjtNums in geom_user
  int nuser_site;                 // number of mjtNums in site_user
  int nuser_cam;                  // number of mjtNums in cam_user
  int nuser_tendon;               // number of mjtNums in tendon_user
  int nuser_actuator;             // number of mjtNums in actuator_user
  int nuser_sensor;               // number of mjtNums in sensor_user
  int nkey;                       // number of keyframes
  int njmax;                      // (deprecated) max number of constraints
  int nconmax;                    // (deprecated) max number of detected contacts
  size_t nstack;                  // (deprecated) number of mjtNums in mjData stack

  // global data
  mjString* comment;              // comment at top of XML
  mjString* modelfiledir;         // path to model file

  // other
  mjtByte hasImplicitPluginElem;  // already encountered an implicit plugin sensor/actuator
} mjSpec;
```

mjsElement

Special type corresponding to any element. This struct is the first member of all other elements; in the low-level C++ implementation, it is not included as a member but via class inheritance. Inclusion via inheritance allows the compiler to `stat:` `mjsElement` to the correct C++ object class. Unlike all other attributes of the structs

stable

below, which are user-settable by design, modifying the contents of an `mjsElement` is not allowed and leads to undefined behavior.

```
typedef struct mjsElement_ {       // element type, do not modify
  mjtObj elemtype;                 // element type
  uint64_t signature;              // compilation signature
} mjsElement;
```

mjsCompiler

Compiler options.

```
typedef struct mjsCompiler_ {      // compiler options
  mjtByte autolimits;              // infer "limited" attribute based on range
  double boundmass;                // enforce minimum body mass
  double boundinertia;             // enforce minimum body diagonal inertia
  double settotalmass;             // rescale masses and inertias; <=0: ignore
  mjtByte balanceinertia;          // automatically impose A + B >= C rule
  mjtByte fitaabb;                 // meshfit to aabb instead of inertia box
  mjtByte degree;                  // angles in radians or degrees
  char eulerseq[3];                // sequence for euler rotations
  mjtByte discardvisual;           // discard visual geoms in parser
  mjtByte usethread;               // use multiple threads to speed up compiler
  mjtByte fusestatic;              // fuse static bodies with parent
  int inertiafromgeom;             // use geom inertias (mjtInertiaFromGeom)
  int inertiagrouprange[2];        // range of geom groups used to compute inertia
  mjtByte saveinertial;            // save explicit inertial clause for all bodies to XML
  int alignfree;                   // align free joints with inertial frame
  mjLROpt LRopt;                   // options for lengthrange computation
} mjsCompiler;
```

mjsBody

Body specification.

```
typedef struct mjsBody_ {          // body specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* childclass;            // childclass name

  // body frame
  double pos[3];                   // frame position
  double quat[4];                  // frame orientation
  mjsOrientation alt;              // frame alternative orientation

  // inertial frame
  double mass;                     // mass
  double ipos[3];                  // inertial frame position
  double iquat[4];                 // inertial frame orientation
  double inertia[3];               // diagonal inertia (in i-frame)
```

stable

```
  mjsOrientation ialt;             // inertial frame alternative orientation
  double fullinertia[6];           // non-axis-aligned inertia matrix

  // other
  mjtByte mocap;                   // is this a mocap body
  double gravcomp;                 // gravity compensation
  mjDoubleVec* userdata;           // user data
  mjtByte explicitinertial;        // whether to save the body with explicit inertial clause
  mjsPlugin plugin;                // passive force plugin
  mjString* info;                  // message appended to compiler errors
} mjsBody;
```

mjsFrame

Frame specification.

```
typedef struct mjsFrame_ {         // frame specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* childclass;            // childclass name
  double pos[3];                   // position
  double quat[4];                  // orientation
  mjsOrientation alt;              // alternative orientation
  mjString* info;                  // message appended to compiler errors
} mjsFrame;
```

mjsJoint

Joint specification.

```
typedef struct mjsJoint_ {         // joint specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjtJoint type;                   // joint type

  // kinematics
  double pos[3];                   // anchor position
  double axis[3];                  // joint axis
  double ref;                      // value at reference configuration: qpos0
  int align;                       // align free joint with body com (mjtAlignFree)

  // stiffness
  double stiffness;                // stiffness coefficient
  double springref;                // spring reference value: qpos_spring
  double springdamper[2];          // timeconst, dampratio

  // limits
  int limited;                     // does joint have limits (mjtLimited)
  double range[2];                 // joint limits
  double margin;                   // margin value for joint limit detection
  mjtNum solref_limit[mjNREF];     // solver reference: joint limits
  mjtNum solimp_limit[mjNIMP];     // solver impedance: joint limits
  int actfrclimited;               // are actuator forces on joint limited (mjtLimited)
  double actfrcrange[2];           // actuator force limits

  // dof properties
  double armature;                 // armature inertia (mass for slider)
  double damping;                  // damping coefficient
  double frictionloss;             // friction loss
  mjtNum solref_friction[mjNREF];  // solver reference: dof friction
  mjtNum solimp_friction[mjNIMP];  // solver impedance: dof friction

  // other
  int group;                       // group
  mjtByte actgravcomp;             // is gravcomp force applied via actuators
  mjDoubleVec* userdata;           // user data
  mjString* info;                  // message appended to compiler errors
} mjsJoint;
```

mjsGeom

Geom specification.

```
typedef struct mjsGeom_ {          // geom specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjtGeom type;                    // geom type

  // frame, size
  double pos[3];                   // position
  double quat[4];                  // orientation
  mjsOrientation alt;              // alternative orientation
  double fromto[6];                // alternative for capsule, cylinder, box, ellipsoid
  double size[3];                  // type-specific size

  // contact related
  int contype;                     // contact type
  int conaffinity;                 // contact affinity
  int condim;                      // contact dimensionality
  int priority;                    // contact priority
  double friction[3];              // one-sided friction coefficients: slide, roll, spin
  double solmix;                   // solver mixing for contact pairs
  mjtNum solref[mjNREF];           // solver reference
  mjtNum solimp[mjNIMP];           // solver impedance
  double margin;                   // margin for contact detection
  double gap;                      // include in solver if dist < margin-gap

  // inertia inference
  double mass;                     // used to compute density
  double density;                  // used to compute mass and inertia from volume or surfac
  mjtGeomInertia typeinertia;      // selects between surface and volume inertia
```

stable

```
  // fluid forces
  mjtNum fluid_ellipsoid;          // whether ellipsoid-fluid model is active
  mjtNum fluid_coefs[5];           // ellipsoid-fluid interaction coefs

  // visual
  mjString* material;              // name of material
  float rgba[4];                   // rgba when material is omitted
  int group;                       // group

  // other
  mjString* hfieldname;            // heightfield attached to geom
  mjString* meshname;              // mesh attached to geom
  double fitscale;                 // scale mesh uniformly
  mjDoubleVec* userdata;           // user data
  mjsPlugin plugin;                // sdf plugin
  mjString* info;                  // message appended to compiler errors
} mjsGeom;
```

mjsSite

Site specification.

```
typedef struct mjsSite_ {          // site specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // frame, size
  double pos[3];                   // position
  double quat[4];                  // orientation
  mjsOrientation alt;              // alternative orientation
  double fromto[6];                // alternative for capsule, cylinder, box, ellipsoid
  double size[3];                  // geom size

  // visual
  mjtGeom type;                    // geom type
  mjString* material;              // name of material
  int group;                       // group
  float rgba[4];                   // rgba when material is omitted

  // other
  mjDoubleVec* userdata;           // user data
  mjString* info;                  // message appended to compiler errors
} mjsSite;
```

mjsCamera

Camera specification.

stable

```
typedef struct mjsCamera_ {        // camera specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // extrinsics
  double pos[3];                   // position
  double quat[4];                  // orientation
  mjsOrientation alt;              // alternative orientation
  mjtCamLight mode;                // tracking mode
  mjString* targetbody;            // target body for tracking/targeting

  // intrinsics
  int orthographic;                // is camera orthographic
  double fovy;                     // y-field of view
  double ipd;                      // inter-pupilary distance
  float intrinsic[4];              // camera intrinsics (length)
  float sensor_size[2];            // sensor size (length)
  float resolution[2];             // resolution (pixel)
  float focal_length[2];           // focal length (length)
  float focal_pixel[2];            // focal length (pixel)
  float principal_length[2];       // principal point (length)
  float principal_pixel[2];        // principal point (pixel)

  // other
  mjDoubleVec* userdata;           // user data
  mjString* info;                  // message appended to compiler errors
} mjsCamera;
```

mjsLight

Light specification.

```
typedef struct mjsLight_ {         // light specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // frame
  double pos[3];                   // position
  double dir[3];                   // direction
  mjtCamLight mode;                // tracking mode
  mjString* targetbody;            // target body for targeting

  // intrinsics
  mjtByte active;                  // is light active
  mjtByte directional;             // is light directional or spot
  mjtByte castshadow;              // does light cast shadows
  double bulbradius;               // bulb radius, for soft shadows
  float attenuation[3];            // OpenGL attenuation (quadratic model)
  float cutoff;                    // OpenGL cutoff
  float exponent;                  // OpenGL exponent
  float ambient[3];                // ambient color
  float diffuse[3];                // diffuse color
```

stable

```
  float specular[3];               // specular color

  // other
  mjString* info;                  // message appended to compiler errorsx
} mjsLight;
```

mjsFlex

Flex specification.

```
typedef struct mjsFlex_ {          // flex specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // contact properties
  int contype;                     // contact type
  int conaffinity;                 // contact affinity
  int condim;                      // contact dimensionality
  int priority;                    // contact priority
  double friction[3];              // one-sided friction coefficients: slide, roll, spin
  double solmix;                   // solver mixing for contact pairs
  mjtNum solref[mjNREF];           // solver reference
  mjtNum solimp[mjNIMP];           // solver impedance
  double margin;                   // margin for contact detection
  double gap;                      // include in solver if dist<margin-gap

  // other properties
  int dim;                         // element dimensionality
  double radius;                   // radius around primitive element
  mjtByte internal;                // enable internal collisions
  mjtByte flatskin;                // render flex skin with flat shading
  int selfcollide;                 // mode for flex self collision
  int activelayers;                // number of active element layers in 3D
  int group;                       // group for visualizatioh
  double edgestiffness;            // edge stiffness
  double edgedamping;              // edge damping
  float rgba[4];                   // rgba when material is omitted
  mjString* material;              // name of material used for rendering
  double young;                    // Young's modulus
  double poisson;                  // Poisson's ratio
  double damping;                  // Rayleigh's damping
  double thickness;                // thickness (2D only)

  // mesh properties
  mjStringVec* nodebody;           // node body names
  mjStringVec* vertbody;           // vertex body names
  mjDoubleVec* node;               // node positions
  mjDoubleVec* vert;               // vertex positions
  mjIntVec* elem;                  // element vertex ids
  mjFloatVec* texcoord;            // vertex texture coordinates
```

stable

```
  mjIntVec* elemtexcoord;          // element texture coordinates

  // other
  mjString* info;                  // message appended to compiler errors
} mjsFlex;
```

mjsMesh

Mesh specification.

```
typedef struct mjsMesh_ {          // mesh specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* content_type;          // content type of file
  mjString* file;                  // mesh file
  double refpos[3];                // reference position
  double refquat[4];               // reference orientation
  double scale[3];                 // rescale mesh
  mjtMeshInertia inertia;          // inertia type (convex, legacy, exact, shell)
  mjtByte smoothnormal;            // do not exclude large-angle faces from normals
  int maxhullvert;                 // maximum vertex count for the convex hull
  mjFloatVec* uservert;            // user vertex data
  mjFloatVec* usernormal;          // user normal data
  mjFloatVec* usertexcoord;        // user texcoord data
  mjIntVec* userface;              // user vertex indices
  mjIntVec* userfacetexcoord;      // user texcoord indices
  mjsPlugin plugin;                // sdf plugin
  mjString* info;                  // message appended to compiler errors
} mjsMesh;
```

mjsHField

Height field specification.

```
typedef struct mjsHField_ {        // height field specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* content_type;          // content type of file
  mjString* file;                  // file: (nrow, ncol, [elevation data])
  double size[4];                  // hfield size (ignore referencing geom size)
  int nrow;                        // number of rows
  int ncol;                        // number of columns
  mjFloatVec* userdata;            // user-provided elevation data
  mjString* info;                  // message appended to compiler errors
} mjsHField;
```

mjsSkin

stable

Skin specification.

```
typedef struct mjsSkin_ {          // skin specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* file;                  // skin file
  mjString* material;              // name of material used for rendering
  float rgba[4];                   // rgba when material is omitted
  float inflate;                   // inflate in normal direction
  int group;                       // group for visualization

  // mesh
  mjFloatVec* vert;                // vertex positions
  mjFloatVec* texcoord;            // texture coordinates
  mjIntVec* face;                  // faces

  // skin
  mjStringVec* bodyname;           // body names
  mjFloatVec* bindpos;             // bind pos
  mjFloatVec* bindquat;            // bind quat
  mjIntVecVec* vertid;             // vertex ids
  mjFloatVecVec* vertweight;       // vertex weights

  // other
  mjString* info;                  // message appended to compiler errors
} mjsSkin;
```

mjsTexture

Texture specification.

```
typedef struct mjsTexture_ {       // texture specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjtTexture type;                 // texture type

  // method 1: builtin
  int builtin;                     // builtin type (mjtBuiltin)
  int mark;                        // mark type (mjtMark)
  double rgb1[3];                  // first color for builtin
  double rgb2[3];                  // second color for builtin
  double markrgb[3];               // mark color
  double random;                   // probability of random dots
  int height;                      // height in pixels (square for cube and skybox)
  int width;                       // width in pixels
  int nchannel;                    // number of channels

  // method 2: single file
  mjString* content_type;          // content type of file
  mjString* file;                  // png file to load; use for all sides
  int gridsize[2];                 // size of grid for composite file; (1,1)-repeat
  char gridlayout[13];             // row-major: L,R,F,B,U,D for faces; . for unused
  // method 3: separate files
  mjStringVec* cubefiles;          // different file for each side of the cube

  // method 4: from buffer read by user
  mjByteVec* data;                  // texture data

  // flip options
  mjtByte hflip;                   // horizontal flip
  mjtByte vflip;                   // vertical flip

  // other
  mjString* info;                  // message appended to compiler errors
} mjsTexture;
```

mjsMaterial

Material specification.

```
typedef struct mjsMaterial_ {      // material specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjStringVec* textures;           // names of textures (empty: none)
  mjtByte texuniform;              // make texture cube uniform
  float texrepeat[2];              // texture repetition for 2D mapping
  float emission;                  // emission
  float specular;                  // specular
  float shininess;                 // shininess
  float reflectance;               // reflectance
  float metallic;                  // metallic
  float roughness;                 // roughness
  float rgba[4];                   // rgba
  mjString* info;                  // message appended to compiler errors
} mjsMaterial;
```

mjsPair

Pair specification.

```
typedef struct mjsPair_ {          // pair specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* geomname1;             // name of geom 1
  mjString* geomname2;             // name of geom 2

  // optional parameters: computed from geoms if not set by user
  int condim;                      // contact dimensionality
  mjtNum solref[mjNREF];           // solver reference, normal direction
  mjtNum solreffriction[mjNREF];   // solver reference, frictional directions
  mjtNum solimp[mjNIMP];           // solver impedance
  double margin;                   // margin for contact detection
```

stable

```
  double gap;                       // include in solver if dist<margin-gap
  double friction[5];               // full contact friction
  mjString* info;                   // message appended to errors
} mjsPair;
```

mjsExclude

Exclude specification.

```
typedef struct mjsExclude_ {       // exclude specification
  mjsElement* element;              // element type
  mjString* name;                   // name
  mjString* bodyname1;              // name of geom 1
  mjString* bodyname2;              // name of geom 2
  mjString* info;                   // message appended to errors
} mjsExclude;
```

mjsEquality

Equality specification.

```
typedef struct mjsEquality_ {      // equality specification
  mjsElement* element;              // element type
  mjString* name;                   // name
  mjtEq type;                       // constraint type
  double data[mjNEQDATA];           // type-dependent data
  mjtByte active;                   // is equality initially active
  mjString* name1;                  // name of object 1
  mjString* name2;                  // name of object 2
  mjtObj objtype;                   // type of both objects
  mjtNum solref[mjNREF];            // solver reference
  mjtNum solimp[mjNIMP];            // solver impedance
  mjString* info;                   // message appended to errors
} mjsEquality;
```

mjsTendon

Tendon specification.

```
typedef struct mjsTendon_ {        // tendon specification
  mjsElement* element;              // element type
  mjString* name;                   // name

  // stiffness, damping, friction, armature
  double stiffness;                 // stiffness coefficient
  double springlength[2];           // spring resting length; {-1, -1}: use qpos_spring
  double damping;                   // damping coefficient
  double frictionloss;              // friction loss
  mjtNum solref_friction[mjNREF];   // solver reference: tendon friction
  mjtNum solimp_friction[mjNIMP];   // solver impedance: tendon friction
  double armature;                  // inertia associated with tendon velocity
  // length range
  int limited;                     // does tendon have limits (mjtLimited)
  int actfrclimited;               // does tendon have actuator force limits
  double range[2];                 // length limits
  double actfrcrange[2];           // actuator force limits
  double margin;                   // margin value for tendon limit detection
  mjtNum solref_limit[mjNREF];     // solver reference: tendon limits
  mjtNum solimp_limit[mjNIMP];     // solver impedance: tendon limits

  // visual
  mjString* material;              // name of material for rendering
  double width;                    // width for rendering
  float rgba[4];                   // rgba when material is omitted
  int group;                       // group

  // other
  mjDoubleVec* userdata;           // user data
  mjString* info;                  // message appended to errors
} mjsTendon;
```

mjsWrap

Wrapping object specification.

```
typedef struct mjsWrap_ {          // wrapping object specification
  mjsElement* element;             // element type
  mjString* info;                  // message appended to errors
} mjsWrap;
```

mjsActuator

Actuator specification.

```
typedef struct mjsActuator_ {      // actuator specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // gain, bias
  mjtGain gaintype;                // gain type
  double gainprm[mjNGAIN];         // gain parameters
  mjtBias biastype;                // bias type
  double biasprm[mjNGAIN];         // bias parameters

  // activation state
  mjtDyn dyntype;                  // dynamics type
  double dynprm[mjNDYN];           // dynamics parameters
  int actdim;                      // number of activation variables
  mjtByte actearly;                // apply next activations to qfrc

  // transmission
```

stable

```
  mjtTrn trntype;                  // transmission type
  double gear[6];                  // length and transmitted force scaling
  mjString* target;                // name of transmission target
  mjString* refsite;               // reference site, for site transmission
  mjString* slidersite;            // site defining cylinder, for slider-crank
  double cranklength;              // crank length, for slider-crank
  double lengthrange[2];           // transmission length range
  double inheritrange;             // automatic range setting for position and intvelocity

  // input/output clamping
  int ctrllimited;                 // are control limits defined (mjtLimited)
  double ctrlrange[2];             // control range
  int forcelimited;                // are force limits defined (mjtLimited)
  double forcerange[2];            // force range
  int actlimited;                  // are activation limits defined (mjtLimited)
  double actrange[2];              // activation range

  // other
  int group;                       // group
  mjDoubleVec* userdata;           // user data
  mjsPlugin plugin;                // actuator plugin
  mjString* info;                  // message appended to compiler errors
} mjsActuator;
```

mjsSensor

Sensor specification.

```
typedef struct mjsSensor_ {        // sensor specification
  mjsElement* element;             // element type
  mjString* name;                  // name

  // sensor definition
  mjtSensor type;                  // type of sensor
  mjtObj objtype;                  // type of sensorized object
  mjString* objname;               // name of sensorized object
  mjtObj reftype;                  // type of referenced object
  mjString* refname;               // name of referenced object

  // user-defined sensors
  mjtDataType datatype;            // data type for sensor measurement
  mjtStage needstage;              // compute stage needed to simulate sensor
  int dim;                         // number of scalar outputs

  // output post-processing
  double cutoff;                   // cutoff for real and positive datatypes
  double noise;                    // noise stdev

  // other
  mjDoubleVec* userdata;           // user data
```

stable

```
  mjsPlugin plugin;                // sensor plugin
  mjString* info;                  // message appended to compiler errors
} mjsSensor;
```

mjsNumeric

Custom numeric field specification.

```
typedef struct mjsNumeric_ {       // custom numeric field specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjDoubleVec* data;               // initialization data
  int size;                        // array size, can be bigger than data size
  mjString* info;                  // message appended to compiler errors
} mjsNumeric;
```

mjsText

Custom text specification.

```
typedef struct mjsText_ {          // custom text specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjString* data;                  // text string
  mjString* info;                  // message appended to compiler errors
} mjsText;
```

mjsTuple

Tuple specification.

```
typedef struct mjsTuple_ {         // tuple specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  mjIntVec* objtype;               // object types
  mjStringVec* objname;            // object names
  mjDoubleVec* objprm;             // object parameters
  mjString* info;                  // message appended to compiler errors
} mjsTuple;
```

mjsKey

Keyframe specification.

```
typedef struct mjsKey_ {           // keyframe specification
  mjsElement* element;             // element type
  mjString* name;                  // name
  double time;                     // time
  mjDoubleVec* qpos;               // qpos
  mjDoubleVec* qvel;               // qvel
  mjDoubleVec* act;                // act
  mjDoubleVec* mpos;              // mocap pos
  mjDoubleVec* mquat;             // mocap quat
  mjDoubleVec* ctrl;              // ctrl
  mjString* info;                 // message appended to compiler errors
} mjsKey;
```

mjsDefault

Default specification.

```
typedef struct mjsDefault_ {      // default specification
  mjsElement* element;            // element type
  mjString* name;                 // class name
  mjsJoint* joint;                // joint defaults
  mjsGeom* geom;                  // geom defaults
  mjsSite* site;                  // site defaults
  mjsCamera* camera;              // camera defaults
  mjsLight* light;                // light defaults
  mjsFlex* flex;                  // flex defaults
  mjsMesh* mesh;                  // mesh defaults
  mjsMaterial* material;          // material defaults
  mjsPair* pair;                  // pair defaults
  mjsEquality* equality;          // equality defaults
  mjsTendon* tendon;              // tendon defaults
  mjsActuator* actuator;          // actuator defaults
} mjsDefault;
```

mjsPlugin

Plugin specification.

```
typedef struct mjsPlugin_ {       // plugin specification
  mjsElement* element;            // element type
  mjString* name;                 // instance name
  mjString* plugin_name;          // plugin name
  mjtByte active;                 // is the plugin active
  mjString* info;                 // message appended to compiler errors
} mjsPlugin;
```

mjsOrientation

Alternative orientation specifiers.

```
typedef struct mjsOrientation_ {  // alternative orientation specifiers
  mjtOrientation type;            // active orientation specifier
  double axisangle[4];            // axis and angle
  double xyaxes[6];               // x and y axes
  double zaxis[3];                // z axis (minimal rotation)
  double euler[3];                // Euler angles
} mjsOrientation;
```

stable

Array handles

C handles for C++ strings and vector types. When using from C, use the provided getters and setters.

```
#ifdef __cplusplus
  // C++: defined to be compatible with corresponding std types
  using mjString      = std::string;
  using mjStringVec   = std::vector<std::string>;
  using mjIntVec      = std::vector<int>;
  using mjIntVecVec   = std::vector<std::vector<int>>;
  using mjFloatVec    = std::vector<float>;
  using mjFloatVecVec = std::vector<std::vector<float>>;
  using mjDoubleVec   = std::vector<double>;
  using mjByteVec     = std::vector<std::byte>;
#else
  // C: opaque types
  typedef void mjString;
  typedef void mjStringVec;
  typedef void mjIntVec;
  typedef void mjIntVecVec;
  typedef void mjFloatVec;
  typedef void mjFloatVecVec;
  typedef void mjDoubleVec;
  typedef void mjByteVec;
#endif
```

Plugins

The names of these struct types are prefixed with `mjp`. See Engine plugins for more details.

mjpPlugin

This structure contains the definition of a single engine plugin. It mostly contains a set of callbacks, which are triggered by the compiler and the engine during various phases of the computation pipeline.

```
struct mjpPlugin_ {
  const char* name;                 // globally unique name identifying the plugin

  int nattribute;                   // number of configuration attributes
  const char* const* attributes;    // name of configuration attributes

  int capabilityflags;              // plugin capabilities: bitfield of mjtPluginCapabilityBit
  int needstage;                    // sensor computation stage (mjtStage)

  // number of mjtNums needed to store the state of a plugin instance (required)
  int (*nstate)(const mjModel* m, int instance);
```

stable

```
  // dimension of the specified sensor's output (required only for sensor plugins)
  int (*nsensordata)(const mjModel* m, int instance, int sensor_id);

  // called when a new mjData is being created (required), returns 0 on success or -1 on fai
  int (*init)(const mjModel* m, mjData* d, int instance);

  // called when an mjData is being freed (optional)
  void (*destroy)(mjData* d, int instance);

  // called when an mjData is being copied (optional)
  void (*copy)(mjData* dest, const mjModel* m, const mjData* src, int instance);

  // called when an mjData is being reset (required)
  void (*reset)(const mjModel* m, mjtNum* plugin_state, void* plugin_data, int instance);

  // called when the plugin needs to update its outputs (required)
  void (*compute)(const mjModel* m, mjData* d, int instance, int capability_bit);

  // called when time integration occurs (optional)
  void (*advance)(const mjModel* m, mjData* d, int instance);

  // called by mjv_updateScene (optional)
  void (*visualize)(const mjModel*m, mjData* d, const mjvOption* opt, mjvScene* scn, int ins

  // methods specific to actuators (optional)

  // updates the actuator plugin's entries in act_dot
  // called after native act_dot is computed and before the compute callback
  void (*actuator_act_dot)(const mjModel* m, mjData* d, int instance);

  // methods specific to signed distance fields (optional)

  // signed distance from the surface
  mjtNum (*sdf_distance)(const mjtNum point[3], const mjData* d, int instance);

  // gradient of distance with respect to local coordinates
  void (*sdf_gradient)(mjtNum gradient[3], const mjtNum point[3], const mjData* d, int insta

  // called during compilation for marching cubes
  mjtNum (*sdf_staticdistance)(const mjtNum point[3], const mjtNum* attributes);

  // convert attributes and provide defaults if not present
  void (*sdf_attribute)(mjtNum attribute[], const char* name[], const char* value[]);

  // bounding box of implicit surface
  void (*sdf_aabb)(mjtNum aabb[6], const mjtNum* attributes);
};
typedef struct mjpPlugin_ mjpPlugin;
```

mjpResourceProvider

This data structure contains the definition of a resource provider. It contains a set of callbacks used for opening and reading resources.

```
struct mjpResourceProvider {
  const char* prefix;               // prefix for match against a resource name
  mjfOpenResource open;             // opening callback
  mjfReadResource read;             // reading callback
  mjfCloseResource close;           // closing callback
  mjfGetResourceDir getdir;         // get directory callback (optional)
  mjfResourceModified modified;     // resource modified callback (optional)
  void* data;                       // opaque data pointer (resource invariant)
};
typedef struct mjpResourceProvider mjpResourceProvider;
```

Function types

MuJoCo callbacks have corresponding function types. They are defined in mjdata.h and in mjui.h. The actual callback functions are documented in the globals page.

Physics Callbacks

These function types are used by physics callbacks.

mjfGeneric

```
typedef void (*mjfGeneric)(const mjModel* m, mjData* d);
```

This is the function type of the callbacks mjcb_passive and mjcb_control.

mjfConFilt

```
typedef int (*mjfConFilt)(const mjModel* m, mjData* d, int geom1, int geom2);
```

This is the function type of the callback mjcb_contactfilter. The return value is 1: discard, 0: proceed with collision check.

mjfSensor

```
typedef void (*mjfSensor)(const mjModel* m, mjData* d, int stage);
```

This is the function type of the callback mjcb_sensor.

mjfTime

stable

```
typedef mjtNum (*mjfTime)(void);
```

This is the function type of the callback mjcb_time.

mjfAct

```
typedef mjtNum (*mjfAct)(const mjModel* m, const mjData* d, int id);
```

This is the function type of the callbacks mjcb_act_dyn, mjcb_act_gain and mjcb_act_bias.

mjfCollision

```
typedef int (*mjfCollision)(const mjModel* m, const mjData* d,
                            mjContact* con, int g1, int g2, mjtNum margin);
```

This is the function type of the callbacks in the collision table mjCOLLISIONFUNC.

UI Callbacks

These function types are used by the UI framework.

mjfItemEnable

```
typedef int (*mjfItemEnable)(int category, void* data);
```

This is the function type of the predicate function used by the UI framework to determine if each item is enabled or disabled.

Resource Provider Callbacks

These callbacks are used by resource providers.

mjfOpenResource

```
typedef int (*mjfOpenResource)(mjResource* resource);
```

This callback is for opeing a resource; returns zero on failure.

mjfReadResource

```
typedef int (*mjfReadResource)(mjResource* resource, const void** buffer);
```

This callback is for reading a resource. Returns number of bytes stored in buffer and returns -1 on error.

mjfCloseResource

```
typedef void (*mjfCloseResource)(mjResource* resource);
```

This callback is for closing a resource, and is responsible for freeing an memory.

stable

mjfGetResourceDir

```
typedef void (*mjfGetResourceDir)(mjResource* resource, const char** dir, int* ndir);
```

This callback is for returning the directory of a resource, by setting dir to the directory string with ndir being size of directory string.

mjfResourceModified

```
typedef int (*mjfResourceModified)(const mjResource* resource);
```

This callback is for checking if a resource was modified since it was last read. Returns positive value if the resource was modified since last open, 0 if resource was not modified, and negative value if inconclusive.

Notes

This section contains miscellaneous notes regarding data-structure conventions in MuJoCo struct types.

c-frame variables

mjData contains two arrays with the `c` prefix, which are used for internal calculations: `cdof` and `cinert`, both computed by mj_comPos. The `c` prefix means that quantities are with respect to the "c-frame", a frame at the center-of-mass of the local kinematic subtree (`mjData.subtree_com`), oriented like the world frame. This choice increases the precision of kinematic computations for mechanisms that are distant from the global origin.

`cdof`:

These 6D motion vectors (3 rotation, 3 translation) describe the instantaneous axis of a degree-of-freedom and are used by all Jacobian functions. The minimal computation required for analytic Jacobians is mj_kinematics followed by mj_comPos.

`cinert`:

These 10-vectors describe the inertial properties of a body in the c-frame and are used by the Composite Rigid Body algorithm (mj_crb). The 10 numbers are packed arrays of lengths (6, 3, 1) with semantics:

`cinert[0-5]`: Upper triangle of the body's inertia matrix.

stable

`cinert[6-8]`: Body mass multiplied by the body CoM's offset from the c-frame origin.

cinert[9]: Body mass.

Convex hulls

The convex hull descriptors are stored in mjModel:

```
int*      mesh_graphadr;     // graph data address; -1: no graph      (nmesh x 1)
int*      mesh_graph;        // convex graph data                     (nmeshgraph x 1)
```

If mesh N has a convex hull stored in mjModel (which is optional), then m->mesh_graphadr[N] is the offset of mesh N's convex hull data in m->mesh_graph. The convex hull data for each mesh is a record with the following format:

```
int numvert;
int numface;
int vert_edgeadr[numvert];
int vert_globalid[numvert];
int edge_localid[numvert+3*numface];
int face_globalid[3*numface];
```

Note that the convex hull contains a subset of the vertices of the full mesh. We use the nomenclature globalid to refer to vertex indices in the full mesh, and localid to refer to vertex indices in the convex hull. The meaning of the fields is as follows:

numvert

Number of vertices in the convex hull.

numface

Number of faces in the convex hull.

vert_edgeadr[numvert]

For each vertex in the convex hull, this is the offset of the edge record for that vertex in edge_localid.

vert_globalid[numvert]

For each vertex in the convex hull, this is the corresponding vertex index in the full mesh

edge_localid[numvert+3*numface]

This contains a sequence of edge records, one for each vertex in the convex hull. Each edge record is an array of vertex indices (in localid format) terminated with -1. For example, say the record for vertex 7 is: 3, 4, 5, 9, -1. This means that vertex 7 belongs to 4 edges, and the other ends of these edges are verti
this way every edge is represented twice, in the edge records of i
Note that for a closed triangular mesh (such as the convex hulls used here), the

stable

number of edges is `3*numface/2`. Thus when each edge is represented twice, we have `3*numface edges`. And since we are using the separator -1 at the end of each edge record (one separator per vertex), the length of `edge_localid` is `numvert+3*numface`.

`face_globalid[3*numface]`

For each face of the convex hull, this contains the indices of the three vertices in the full mesh

stable