MDAPI Specification (draft)

Version 0.9

David J. Hardy

September 17, 2004

Contents

Chapter 1

Overview

1.1 Purpose

The MDAPI (molecular dynamics applications programming interface) improves the design of molecular dynamics (MD) software for the simulation of biomolecules by enabling the separation of source code into a front end and a computational engine. The computational engine includes all of the code that performs the numerical computation for each time step of an MD simulation. The front end contains everything else that is needed for performing an MD simulation, specifically, the setup of the simulation, the analysis of the results, and all data file processing.

The MDAPI provides a well-defined, extensible interface that allows front ends to inter-operate with engines. User-friendly front ends can be developed that support a variety of data file formats without concern for introducing any side effects to the numerical portions of the code. Highly optimized engines can be developed without having to worry about all of the supporting tasks that are necessary for performing an MD simulation. Specialized engines can be created to solve particular problems or test new computational methods, all driven by the same front end.

1.2 Features

The MDAPI is a lightweight C programming interface with an object-oriented design. It compiles into a C library (tested on Linux and Solaris) to enable linking to either C or C++ programs, and it should be easily ported to any modern platform. There is no explicit support for Fortran engines, although it should be straightforward to provide.

The features supported by the MDAPI design are:

- predefined data types for describing the CHARMM force field :::ref:::, topology, and trajectory of a system of biomolecules;
- communication between front end and engine using data arrays referenced by string identifiers;
- front end control over multiple engines (or multiple instances of the same engine) all running simultaneously;
- nonblocking calling semantics for routines that are not designed to return immediately;

- front end callback routines for data communication during a running simulation;
- engine-defined data types that extend the predefined data types;
- error reporting and handling.

The MDAPI actually provides two interfaces, one for the front end to initialize and control the MD simulation, and the other for the computational engine to make its data and integration methods available to the front end. The front end has routines to initialize the engine, write to the engine data arrays, run the simulation for some number of steps, and read from the engine data arrays to obtain the simulation results. The front end operates on an engine object that is used for calls through the interface. Similarly, the engine is provided with a handle to the front end object to be used for its interface calls. The engine is required to provide an initialization routine (constructor), a cleanup routine (destructor), and a "run" routine that performs the numerical integration of the molecular system.

The front end interface has been designed, with its most basic features, to allow simple and straightforward access to an engine. There are also more advanced features, such as the use of asynchronous control within event loops, callbacks, and error handling, that can provide high performance access to an engine and facilitate the development of advanced GUI front ends.

The engine interface has been designed to provide minimal interference with the important computational aspects of molecular dynamics. Most of the interface calls are required for initialization, during which the engine offers the front end access to data arrays and possibly defines new data types. The "run" routine that contains the numerical integration loop must also be supplimented with interface calls to increment the step counter and process front end callback routines.

The MDAPI provides an interface layer that attempts to take on as much responsibility as possible in order to ease development of front ends and engines. The MDAPI collects the data arrays and types offered by the engine and presents them in a consistent manner to the front end, performing necessary memory management for dynamically allocated data buffers. Similarly, the MDAPI collects callback routines registered by the front end and presents them in a simplified manner to the engine. By using string identifiers to reference data arrays, rather than having hard-coded variable names, and also by permitting engine-defined data types, the interface is extensible beyond basic MD simulations to more specialized applications. Inter-operability between front ends and engines is enabled by specifying a standard naming convention for the fundamental data arrays and types needed for basic MD simulations. The interface to the front end provides routines that allow the investigation of all data arrays and types offered by the engine. The interface to the engine provides routines that allow specific knowledge of all data requirements for callbacks and permit asynchronous control of callback processing to improve performance.

1.3 Current Implementation

The current implementation of MDAPI is single-threaded and blocking, with the asynchronous control routines stubbed. The idea has been to finalize a simplest version of MDAPI that still conforms to the interface specification detailed in this document. A more advanced implementation of the MDAPI is expected to follow that provides the following enhancements:

- dynamic loading of engines (`dlopen`),
- multi-threaded execution (Posix threads),
- remote communication between front end and engine (TCP/IP sockets).

The current implementation will form the basis for future enhancements.

MDAPI is being specifically tailored to the design needs of NAMD :::ref:::, with planned deployment into NAMD 3. The long range vision is for VMD :::ref::: to use the MDAPI to provide front end support for controlling a full-featured NAMD computational engine, as well as special-purpose engines for method development and other scientific investigation.

1.4 Remainder of Document

This document presents aspects of the MDAPI in an order well-suited to familiarizing a C programmer as quickly as possible with the interface syntax and semantics. Chapter 2 provides a very brief summary of molecular dynamics and the potential energy function, then presents the predefined data types and how they are used for representing a molecular system. Chapter 3 discusses the front end interface, and Chapter 4 discusses the engine interface, both presented in a tutorial format. Chapter 5 provides the standard conventions that should be followed by front end and engine implementations to achieve inter-operability for basic MD simulations. Chapter 6 contains the complete reference for the MDAPI, including all defined constants, types, and function prototypes, along with the semantics of every interface routine.

The MDAPI uses the following code conventions.

- All names defined by the MDAPI begin with `MD_` to avoid name space conflicts.
- Constants and macro names are always capitalized with underscores used to separate words, e.g. `MD_ERR_ACCESS`, `MD_SIZEOF`.
- Predefined data types are mostly implemented as C `typedef`ed `struct`s, with each word in the name capitalized, e.g. `MD_BondPrm`, `MD_Attrib`.
- Function names are lowercase after the prefix with underscores used to separate words, e.g. `MD_init`, `MD_type_memberlist`.

Anyone who is interested in the use and development of the MDAPI is welcome to contact its author David Hardy by email to `dhardy@ks.uiuc.edu` regarding any questions, suggestions, or bug reports.

Chapter 2

Molecular Representation

This chapter begins with a brief summary of molecular dynamics (MD) and the potential energy function used for biomolecules. This is followed by an introduction to the MDAPI predefined data types, first discussing how they represent individual potential energy terms and bonded connections between atoms, then describing how arrays of the predefined data types work together to represent the entire molecular system.

2.1 Molecular Modeling

2.1.1 Molecular dynamics and the potential energy function

A molecular dynamics simulation computes numerically the time evolution of Newton's equations of motion for a system of N atoms in 3-dimensional space (i.e. a system of $3N$ equations),

$$\mathbf{F}_i(\mathbf{r}(t)) = m_i \frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{r}_i(t), \quad \text{for } i = 1, 2, \ldots, N,$$

where $\mathbf{r}_i$ is the position of the ith atom in the system, with m_i its mass and $\mathbf{F}_i$ its force as a function of all of the atomic positions. This second-order system is typically recast as a system of $6N$ first-order ODEs,

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{v}_i(t) &= \frac{1}{m_i}\mathbf{F}_i(\mathbf{r}(t)) \\ \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{r}_i(t) &= \mathbf{v}_i(t), \end{aligned}$$

and solved as an initial value problem, which requires initial positions $\mathbf{r}_i = (x_i, y_i, z_i)^\mathsf{T}$ and velocities $\mathbf{v}_i = (\dot{x}_i, \dot{y}_i, \dot{z}_i)^\mathsf{T}$ at time $t = 0$. The numerical integration method of choice for solving this system is the leapfrog (velocity-Verlet) integrator,

$$\begin{aligned} \mathbf{v}_i^{(k+1/2)} &= \mathbf{v}_i^{(k)} + \frac{\Delta t}{2}\frac{\mathbf{f}_i^{(k)}}{m_i}, \\ \mathbf{r}_i^{(k+1)} &= \mathbf{r}_i^{(k)} + \Delta t \mathbf{v}_i^{(k+1/2)}, \\ \mathbf{f}_i^{(k+1)} &= \mathbf{F}_i(\mathbf{r}^{(k+1)}), \\ \mathbf{v}_i^{(k+1)} &= \mathbf{v}_i^{(k+1/2)} + \frac{\Delta t}{2}\frac{\mathbf{f}_i^{(k+1)}}{m_i}, \end{aligned}$$

shown here propagating the ith atom in the system from step number k to step number $k+1$ using a time step of size Δt. Although many other numerical integration methods are possible, the best in practice are variants of leapfrog, which has the virtue of being an explicit, second-order accurate method needing only one force evaluation per time step. For modeling biomolecules of scientific interest, N is in the range of $10,000$ to $1,000,000$ atoms, and numerical stability requires that $\Delta t \approx 1$fs, where 1 femtosecond $= 10^{-15}$ seconds. This means that the relatively short, on biological timescales, simulation of one nanosecond needs around a million time steps.

Evaluating the force is the most computationally demanding part of molecular dynamics. The force is the negative gradient of a scalar potential energy function,

$$\mathbf{F}(\mathbf{r}) = -\nabla U(\mathbf{r}),$$

and, for systems of biomolecules, this potential function involves the summing,

$$U(\mathbf{r}) = \sum U_{\text{bonded}}(\mathbf{r}) + \sum U_{\text{nonbonded}}(\mathbf{r}),$$

over a large number of bonded and nonbonded terms. The bonded potential terms involve 2–, 3–, and 4–body interactions of covalently bonded atoms, with $O(N)$ terms in the summation. The nonbonded potential terms involve interactions between all pairs of atoms (usually excluding pairs of atoms already involved in a bonded term), with $O(N^2)$ terms in the summation, although fast evaluation techniques are used to compute good approximations to their contribution to the potential with $O(N)$ or $O(N \log N)$ computational cost.

2.1.2 Bonded potential energy terms

The bonded potential terms involve 2–, 3–, and 4–body interactions of covalently bonded atoms.

The 2–body spring bond potential describes the harmonic vibrational motion between an (i, j)–pair of covalently bonded atoms,

$$U_{\text{bond}} = k(r_{ij} - r_0)^2,$$

where $r_{ij} = \|\mathbf{r}_j - \mathbf{r}_i\|$ gives the distance between the atoms, r_0 is the equilibrium distance, and k is the spring constant.

The 3–body angular bond potential describes the angular vibrational motion occurring between an (i, j, k)–triple of covalently bonded atoms,

$$U_{\text{angle}} = k_\theta(\theta - \theta_0)^2 + k_{\text{ub}}(r_{ik} - r_{\text{ub}})^2,$$

where, in the first term, θ is the angle in radians between vectors $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ and $\mathbf{r}_{kj} = \mathbf{r}_j - \mathbf{r}_k$, θ_0 is the equilibrium angle, and k_θ is the angle constant. The second term is the Urey–Bradley term used to describe a (noncovalent) spring between the outer i and k atoms, active when constant $k_{\text{ub}} \neq 0$, where, like the spring bond, $r_{ik} = \|\mathbf{r}_k - \mathbf{r}_i\|$ gives the distance between the pair of atoms and r_{ub} is the equilibrium distance.

The 4–body torsion angle (also known as dihedral angle) potential describes the angular spring between the planes formed by the first three and last three atoms of a consecutively bonded (i, j, k, l)–quadruple of atoms,

$$U_{\text{tors}} = \begin{cases} k(1 + \cos(n\psi + \phi)) & \text{if } n > 0, \\ k(\psi - \phi)^2 & \text{if } n = 0, \end{cases}$$

where ψ is the angle in radians between the (i, j, k)–plane and the (j, k, l)–plane. The integer constant n is nonnegative and indicates the periodicity. For $n > 0$, ϕ is the phase shift angle and k is the multiplicative constant. For $n = 0$, ϕ acts as an equilibrium angle and the units of k change to potential/rad^2. A given (i, j, k, l)–quadruple of atoms might contribute multiple terms to the potential, each with its own parameterization. The use of multiple terms for a torsion angle allows for complex angular variation of the potential, effectively a truncated Fourier series.

2.1.3 Nonbonded potential energy terms

The nonbonded potential terms involve interactions between all (i, j)–pairs of atoms, usually excluding pairs of atoms already involved in a bonded term. Even using a fast evaluation methods the cost of computing the nonbonded potentials dominates the work required for each time step of an MD simulation.

The Lennard–Jones potential accounts for the weak dipole attraction between distant atoms and the hard core repulsion as atoms become close,

$$U_{\mathrm{LJ}} = (-E_{\min}) \left[\left(\frac{R_{\min}}{r_{ij}} \right)^{12} - 2 \left(\frac{R_{\min}}{r_{ij}} \right)^{6} \right],$$

where $r_{ij} = \|\mathbf{r}_j - \mathbf{r}_i\|$ gives the distance between the pair of atoms. The parameter $E_{\min} = U_{\mathrm{LJ}}(R_{\min})$ is the minimum of the potential term ($E_{\min} < 0$, which means that $-E_{\min}$ is the well-depth). The Lennard–Jones potential approaches 0 rapidly as r_{ij} increases, so it is usually truncated (smoothly shifted) to 0 past a cutoff radius, requiring $O(N)$ computational cost.

The electrostatic potential is repulsive for atomic charges with the same sign and attractive for atomic charges with opposite signs,

$$U_{\mathrm{elec}} = \epsilon_{14} \frac{C q_i q_j}{\epsilon_0 r_{ij}},$$

where $r_{ij} = \|\mathbf{r}_j - \mathbf{r}_i\|$ gives the distance between the pair of atoms, and q_i and q_j are the charges on the respective atoms. Coulomb's constant C and the dielectric constant ϵ_0 are fixed for all electrostatic interactions. The parameter ϵ_{14} is a unitless scaling factor whose value is 1, except for a modified 1–4 interaction, where the pair of atoms is separated by a sequence of three covalent bonds (so that the atoms might also be involved in a torsion angle interaction), in which case $\epsilon_{14} = \varepsilon$, for a fixed constant $0 \leq \varepsilon \leq 1$. Although the electrostatic potential may be computed with a cutoff like the Lennard–Jones potential, the $1/r$ potential approaches 0 much more slowly than the $1/r^6$ potential, so neglecting the long range electrostatic terms can degrade qualitative results, especially for highly charged systems. There are other fast evaluation methods that approximate the contribution to the long range electrostatic terms that require $O(N)$ or $O(N \log N)$ computational cost, depending on the method.

2.2 Predefined Data Types

The MDAPI predefines a collection of data types to represent the molecular system. The *primary* data types provide storage for fundamental numeric quantities and fixed buffer length strings. These provide data type components with which larger *derived* types are constructed, useful for abstracting parameters for the potential terms discussed in the previous sections or representing the bond connections between atoms.

Engines are able to extend the available collection of data types by defining their own derived data types. This is done by agglomerating previously defined types, either primary or derived, as member fields of the new type, much like a C structure. More detailed information about this type definition process is available in Sec. 4.1.4. The front end has several routines available to dynamically discover new types, enabling use of an object of that type and efficient access to its members. More information about the front end type discovery routines is available in Sec. 3.3.3.

The purpose of abstracting types is to allow use of the MDAPI to be extended beyond providing communication of basic MD data, and doing so in a manner that requires no modification to the MDAPI specification. Towards this purpose, it is necessary to establish well-defined numeric types so that the interface can reliably provide byte offsets to type members and perform byte reordering on data arrays to facilitate communication between a front end and engine running on remote machines of different architectures.

The predefined data types are presented in the next two sections, followed by their use in representing a molecular system with the force field potentials discussed previously. The C type definitions for the predefined data types may all be found in the `mdtypes.h` header file.

2.2.1 Primary data types

The primary data types are used as components for constructing derived data types. The primary types include fundamental numeric quantities, some taken directly from C. Every type has a *name* that can be used in variable declarations and a *number* that is used to abstractly refer to and derive from that type.

The following table lists the nine primary types by name, numeric constant, number of bytes, and purpose.

name	*numeric constant*	*number of bytes*	*purpose*
`char`	`MD_CHAR`	1	single character
`int32`	`MD_INT32`	4 (32 bits)	integer
`float`	`MD_FLOAT`	4	single precision floating point
`double`	`MD_DOUBLE`	8	double precision floating point
`MD_Fvec`	`MD_FVEC`	12	3D vector, single precision
`MD_Dvec`	`MD_DVEC`	24	3D vection, double precision
`MD_Name`	`MD_NAME`	8	short string for atom names
`MD_String`	`MD_STRING`	64	medium-length string
`MD_Message`	`MD_MESSAGE`	512	long string

The `int32` type should be used as a replacement for C `int`, since `int32` guarantees a 4-byte (32-bit) integer, unlike C `int` whose length is architecture dependent. The definitions for the last five table entries are as follows.

```
enum {
  MD_NAME_SIZE = 8,
  MD_STRING_SIZE = 64,
  MD_MESSAGE_SIZE = 512
};

typedef struct MD_Fvec_tag { float  x, y, z; } MD_Fvec;
typedef struct MD_Dvec_tag { double x, y, z; } MD_Dvec;
typedef char MD_Name    [ MD_NAME_SIZE    ];
typedef char MD_String  [ MD_STRING_SIZE  ];
typedef char MD_Message [ MD_MESSAGE_SIZE ];
```

Note that the member components of `MD_Fvec` and `MD_Dvec` are `x`, `y`, and `z`. The purpose of the 3D vector types are to represent mathematical vector quantities, such as position, velocity, and force. Although they appear to be derived from `float` and `double`, respectively, they are treated by the MDAPI as primary for increased efficiency. The string types are meant to store C-style (nil-terminated) strings, which means that `MD_Name`, `MD_String`, and `MD_Message` can accomodate strings of length 7, 63, and 511, respectively.

There are two other related constants for `int32`.

```
enum {
  MD_INT32_MIN = -2147483648,
  MD_INT32_MAX = 2147483647
};
```

2.2.2 Derived data types for force field parameters

The MDAPI provides a collection of derived data types designed to represent the potential energy terms discussed previously. The parameterization of the force field (i.e. the constants in the potential energy terms) are determined based on the kinds of atoms involved in interactions.

Each atom has a *type*, for instance oxygen, hydrogen, or carbon. The `MD_AtomPrm` data type contains parameters for each type of atom present in the system.

```
typedef struct MD_AtomPrm_tag {
  double emin;     /* Lennard-Jones energy min (kcal/mol) */
  double rmin;     /* Lennard-Jones distance for emin (A) */
  double emin14;   /* modified 1-4 energy min (kcal/mol) */
  double rmin14;   /* modified 1-4 distance for emin14 (A) */
  MD_Name type;    /* string to identify atom type */
} MD_AtomPrm;
```

The `type` member is a short string indicating the type of the atom. For the CHARMM force field, this string is alphanumeric of length no longer than four characters. The `emin` and `rmin` members are parameters for determining the minimum energy $E_{\min}$ and corresponding distance $R_{\min}$ for this atom type when involved in a nonbonded Lennard–Jones interaction with another atom. The actual parameters for the potential term are determined by combining the `emin` and `rmin` values for both atom types involved in the interaction. For a pairwise interaction between atom types i and j, the minimum energy is obtained by negating the geometric average of the two `emin` values and the corresponding distance is obtained as the arithmetic average of the two `rmin` values,

$$E_{\min} = -\sqrt{\texttt{emin}_i \, \texttt{emin}_j}, \qquad R_{\min} = \frac{\texttt{rmin}_i + \texttt{rmin}_j}{2}.$$

Note the units on these quantities: `emin` is given in energy units kcal/mol and `rmin` is given in distance units Å (Angstroms). The `emin14` and `rmin14` members serve exactly the same purpose, except for determining parameters for a modified 1–4 interaction. A modified 1–4 interaction occurs if the pair of atoms is separated by a sequence of three covalent bonds, so that the atoms might also be involved in a torsion angle interaction. (There is typically an additional simulation parameter that enables the evaluation of modified 1–4 interactions.)

The `MD_BondPrm` data type contains parameters for each type of spring bond interaction occurring between a pair of covalently bonded atoms, where the type of spring bond is determined by the types of atoms involved in the bond.

```
typedef struct MD_BondPrm_tag {
  double k;        /* spring coefficient (kcal/mol/A^2) */
  double r0;       /* equilibrium length (A) */
  MD_Name type[2]; /* strings to identify atom types */
} MD_BondPrm;
```

The `type` member is a two-element short string array to label the two atom types comprising this bond. The spring coefficient `k` is given in units kcal/mol/Å^2, and the equilibrium length `r0` is given in distance units Å.

The `MD_AnglePrm` data type contains parameters for each type of angle interaction occurring between a sequence of three covalently bonded atoms, where the type of angle is determined by the types of atoms involved in the angle.

```
typedef struct MD_AnglePrm_tag {
```

```
  double k_theta;  /* coefficient for theta (kcal/mol/rad^2) */
  double theta0;   /* equilibrium angle (radians) */
  double k_ub;     /* coef for Urey-Bradley term (kcal/mol/A^2) */
  double r_ub;     /* equil length for Urey-Bradley term (A) */
  MD_Name type[3]; /* strings to identify atom types */
} MD_AnglePrm;
```

The `type` member is a three-element short string array to label the three atom types comprising this angle. The atom types are intended to be listed consecutively, corresponding to the sequence of covalent bonds. The angle coefficient `k_theta` is given in units kcal/mol/rad^2, and the equilibrium angle `theta0` is given in angular units radians. The Urey–Bradley coefficient `k_ub` is given in units kcal/mol/Å^2, and the equilibrium distance `r_ub` is given in distance units Å.

The `MD_TorsPrm` data type contains parameters for each type of torsion angle interaction occurring between a sequence of four covalently bonded atoms, where the type of the torsion angle is determined by the types of the atoms involved in the torsion angle.

```
typedef struct MD_TorsPrm_tag {
  double k_tor;    /* torsion coef (kcal/mol for n>0 OR kcal/mol/rad^2) */
  double phi;      /* phase shift (radians) */
  int32 n;         /* periodicity */
  int32 mult;      /* multiplicity of torsion */
  MD_Name type[4]; /* strings to identify atom types */
} MD_TorsPrm;
```

The `type` member is a four-element short string array to label the four atom types comprising this torsion angle. The atom types are intended to be listed consecutively, corresponding to the sequence of covalent bonds. The integer constant `n` is nonnegative, indicating the periodicity of the torsion angle. The torsion coefficient `k_tor` is given in units kcal/mol for `n` > 0 or units kcal/mol/rad^2 for `n` $= 0$. The phase shift angle `phi` is given in angular units radians. The `mult` member indicates the multiplicity of this torsion angle interaction. The value `mult` $= 1$ indicates that this type of torsion angle contains only one term (whose parameters are defined by the other members). For a torsion angle of multiplicity $m > 1$, the value `mult` $= m$ indicates that this type of torsion angle contains m terms. These additional terms are stored using consecutive elements in an array of `MD_TorsPrm`, the details of which will be discussed in Sec. 2.3.2.

The `MD_NbfixPrm` data type contains parameters that define correction constants for a Lennard–Jones interaction, overriding the usual rules for combining pairs of constants from two `MD_AtomPrm` data.

```
typedef struct MD_NbfixPrm_tag {
  double emin;     /* Lennard-Jones energy min (kcal/mol) */
  double rmin;     /* Lennard-Jones distance for emin (A) */
  double emin14;   /* modified 1-4 energy min (kcal/mol) */
  double rmin14;   /* modified 1-4 distance for emin14 (A) */
  int32 prm[2];    /* index MD_AtomPrm array */
  MD_Name type[2]; /* strings to identify atom types */
} MD_NbfixPrm;
```

The `type` member is a two-element short string array to label the two atom types. The `prm` member is a two-element `int32` array containing the indices to the array of `MD_AtomPrm` indicating the particular atom parameter elements that are being redefined (see Sec. 2.3 for details). The redefinition explicitly assigns

$$E_{\min} = \texttt{emin}, \qquad R_{\min} = \texttt{rmin}$$

for this type of Lennard–Jones interaction. As before, `emin` is given in energy units kcal/mol and `rmin` is given in distance units Å. The `emin14` and `rmin14` serve the similar purpose of overriding the modified 1–4 interaction parameter values.

2.2.3 Derived data types for molecular topology

The MDAPI provides a collection of derived data types designed to represent the molecular topology. These data types gather together parameters for each individual atom and define the connections between covalently bonded atoms. The following data types each contain `int32` members that refer either to the individual atoms in the system or to the force field parameter data types previously discussed. Sec. 2.3 describes the representation of the entire macromolecule using arrays of the force field parameter data types and the topology data types, and these `int32` members are indices into the arrays. However, for the presentation of the topology data types, simply accept the fact that every atom in the system has a unique identification number, as does every atom type (`MD_AtomPrm`), bond type (`MD_BondPrm`), angle type (`MD_AnglePrm`), and torsion type (`MD_TorsPrm`).

Each individual atom in the system has associated parameters that are contained in the `MD_Atom` data type.

```
typedef struct MD_Atom_tag {
  double m;        /* mass (AMU) */
  double q;        /* charge (e) */
  int32 prm;       /* index MD_AtomPrm array */
  int32 notused;   /* padding */
  MD_Name name;    /* string to identify atom name */
  MD_Name type;    /* string to identify atom type name */
} MD_Atom;
```

The `name` member is a short string indicating the name of the atom. For the CHARMM force field, this name field labels an atom by an alphanumeric string of length no longer than four characters. The `type` member plays the same role as `name`, only for labeling the atom type. The `prm` member refers to the `MD_AtomPrm` parameters for this type of atom. The mass of the atom `m` is given in AMU (atomic mass units), and the charge `q` is given in e units (electron charge units). The `notused` member is provided to pad the `struct` for 8-byte word alignment, needed for the containment of word-aligned 8-byte numeric quantities `m` and `q`.

Covalent bonds between pairs of atoms are contained in the `MD_Bond` data type.

```
typedef struct MD_Bond_tag {
  int32 atom[2];   /* index MD_Atom array */
  int32 prm;       /* index MD_BondPrm array */
} MD_Bond;
```

The 2-element `atom` array identifies the atoms by number. The `prm` member identifies, also by number, the spring bond interaction parameters stored using data type `MD_BondPrm`.

Angular bonds between a sequence of three atoms are contained in the `MD_Angle` data type.

```
typedef struct MD_Angle_tag {
  int32 atom[3];   /* index MD_Atom array */
  int32 prm;       /* index MD_AnglePrm array */
} MD_Angle;
```

The 3-element `atom` array indentifies the atoms by number. The `prm` member identifies, also by number, the angle bond interaction parameters stored using data type `MD_AnglePrm`.

Torsion angles between a sequence of four atoms are contained in the `MD_Tors` data type.

```
typedef struct MD_Tors_tag {
  int32 atom[4];   /* index MD_Atom array */
  int32 prm;       /* index MD_TorsPrm array */
} MD_Tors;
```

The 4-element `atom` array identifies the atoms by number. The `prm` member identifies, also by number, the torsion angle interaction parameters stored using data type `MD_TorsPrm`.

Sometimes it is necessary to explicitly exclude pairs of atoms from nonbonded interactions. (This is as opposed to the implicit exclusions that result when, say, two atoms are covalently bonded.) Explicit excluded pairs of atoms are contained in the `MD_Excl` data type.

```
typedef struct MD_Excl_tag {
  int32 atom[2];   /* index MD_Atom array */
} MD_Excl;
```

The 2-element `atom` array identifies the excluded pair of atoms by number.

2.2.4 Summary of derived data types

The previous two sections discussed the derived data types, which contain information about the force field parameters and the molecular topology. The following table summarizes the collection of all ten derived data types, listing them by name, numeric constant, number of bytes, and purpose.

name	*numeric constant*	*number of bytes*	*purpose*
`MD_AtomPrm`	`MD_ATOMPRM`	40	parameters for each type of atom
`MD_BondPrm`	`MD_BONDPRM`	32	parameters for each type of bond
`MD_AnglePrm`	`MD_ANGLEPRM`	56	parameters for each type of angle
`MD_TorsPrm`	`MD_TORSPRM`	56	parameters for each type of torsion
`MD_NbfixPrm`	`MD_NBFIXPRM`	56	parameters to fix nonbonded constants
`MD_Atom`	`MD_ATOM`	40	parameters for each individual atom
`MD_Bond`	`MD_BOND`	12	bond connections between atoms
`MD_Angle`	`MD_ANGLE`	16	angle connections between atoms
`MD_Tors`	`MD_TORS`	20	torsion connections between atoms
`MD_Excl`	`MD_EXCL`	8	explicit nonbonded exclusions

2.2.5 `MD_SIZEOF()` macro and unit conversion constants

All defined data types, whether predefined by the MDAPI or created by the engine, are assigned a type number (`int32`). The `MD_SIZEOF()` macro takes as an argument the type number of a defined data type and returns in bytes the size of an instance of that type. This permits an elegant way to deal with a defined data type as a parameter, for instance generically allocating array space for an array of some given data type.

The `mdtypes.h` header file also defines several unit conversion constants, used as multiplicative factors to convert a quantity to a different set of units. Each unit conversion constant name, value, and purpose are listed in the following table.

named constant	*value*	*purpose*
MD_PICOSEC	0.001	convert time from fs to ps
MD_FEMTOSEC	1000	convert time from ps to fs
MD_ANGSTROM_FS	0.001	convert velocity from Å/ps to Å/fs
MD_ANGSTROM_PS	1000	convert velocity from Å/fs to Å/ps
MD_ENERGY_CONST	0.0004184	convert energy from kcal/mol to AMU $\times$ Å^2/fs^2
MD_FORCE_CONST	0.0004184	convert force from kcal/mol/Å to AMU $\times$ Å/fs^2
MD_KCAL_MOL	1/MD_ENERGY_CONST	convert energy from AMU $\times$ Å^2/fs^2 to kcal/mol
MD_KCAL_MOL_A	1/MD_FORCE_CONST	convert force from AMU $\times$ Å/fs^2 to kcal/mol/Å
MD_COULOMB	332.0636	Coulomb's constant C in units kcal $\times$ Å/mol/e^2
MD_PI	π	standard mathematical constant
MD_RADIANS	$\pi/180$	convert from angle degrees to radians
MD_DEGREES	$180/\pi$	convert from radians to angle degrees

2.3 Representing the Molecular System

The predefined data types presented in the previous section must be used together to represent the entire molecular system. This is accomplished by defining arrays of the data types. The molecular topology data types define the individual atoms in the system (MD_Atom), along with the connections between atoms: covalent bonds (MD_Bond), angle bonds (MD_Angle), dihedrals (MD_Tors), and impropers (MD_Tors). (Torsion angles are categorized by the CHARMM force field as either dihedrals or impropers.) Each of these contributes potential energy terms, where pairs of atoms contribute nonbonded terms and the bonds, angles, dihedrals, and impropers all contribute bonded terms. These terms all have constant parameters, many of which are determined by the types of the atoms involved in the interaction. The force field parameter data types provide these atom type dependent parameters: nonbonded interaction parameters (MD_AtomPrm), bond parameters (MD_BondPrm), angle parameters (MD_AnglePrm), dihedral parameters (MD_TorsPrm), and improper parameters (MD_TorsPrm). The two remaining derived data types provide corrections to the potential terms: explicit nonbonded exclusions (MD_Excl) that list excluded pairs of atoms and corrected nonbonded interaction parameters (MD_NbfixPrm) that fix pairs of constants (derived from MD_AtomPrm).

2.3.1 Required data arrays

The N atoms of the system are represented by an N-element array of MD_Atom (indexed $0 \ldots N-1$). The trajectory of the system is defined by corresponding N-element arrays of type either MD_Dvec or MD_Fvec that contain positions, velocities, and forces, each with the same atom ordering as the MD_Atom array. The other molecular topology and force field parameter data types also each have an associated array. The topology arrays associate force field parameters with their elements. This is accomplished using the prm member in the topology data types as the index into the corresponding force field parameter array.

The MDAPI provides to the front end controlled access to the engine data arrays, with the front end responsible for setting up the molecular system by initializing the data arrays and the engine responsible for evaluating the force field and integrating the system. The data arrays are accessed by string name identifiers, some of which are required to be present to enable front ends to inter-operate with engines. The following table presents the subset of the required data arrays that directly represent the molecular system, listed by name.

array name	*data type*	*purpose and properties*
`"atom"`	`MD_Atom`	parameters required for each atom defines ordering of atoms in system indexes `"atomprm"` array
`"pos"`	`MD_Dvec` or `MD_Fvec`	position for each atom uses same ordering as `"atom"`
`"vel"`	`MD_Dvec` or `MD_Fvec`	velocity for each atom uses same ordering as `"atom"`
`"force"`	`MD_Dvec` or `MD_Fvec`	force for each atom uses same ordering as `"atom"`
`"bond"`	`MD_Bond`	covalent bonds in system references atoms using `"atom"` ordering indexes `"bondprm"` array
`"angle"`	`MD_Angle`	angle bonds in system references atoms using `"atom"` ordering indexes `"angleprm"` array
`"dihed"`	`MD_Tors`	dihedral angles in system references atoms using `"atom"` ordering indexes `"dihedprm"` array
`"impr"`	`MD_Tors`	improper angles in system references atoms using `"atom"` ordering indexes `"imprprm"` array
`"excl"`	`MD_Excl`	excluded nonbonded pairwise interactions references atoms using `"atom"` ordering
`"atomprm"`	`MD_AtomPrm`	nonbonded atom parameters based on atom types
`"bondprm"`	`MD_BondPrm`	covalent bond parameters based on atom types involved in bond
`"angleprm"`	`MD_AnglePrm`	angle bond parameters based on atom types involved in angle
`"dihedprm"`	`MD_TorsPrm`	dihedral angle parameters based on atom types involved in dihedral
`"imprprm"`	`MD_TorsPrm`	improper angle parameters based on atom types involved in improper
`"nbfixprm"`	`MD_NbfixPrm`	corrections to nonbonded parameters references pairs of `"atomprm"` to be fixed

There is some complication involved with having arrays of molecular topology data types with elements that index arrays of force field parameters. The molecular topology data types could have instead been designed to include the storage of force field parameters, eliminating the need for most of the force field parameter data types except perhaps for the nonbonded Lennard–Jones interaction parameters. However, notice that the molecular topology data arrays have a $O(N)$ storage, requiring linear space in the number of atoms, whereas the force field parameters require only linear space in the number of atom *types*, which does not grow with the size of the system. In fact, the bound is a reasonably small constant when considering biological systems. The data structures presented here turn out to provide a significant savings in memory storage over the alternative of combining force field parameters with molecular topology data. Furthermore, the choice to separate force field parameters from molecular topology data matches the data file structuring used by other MD software packages.

2.3.2 Torsion angle multiplicity

Recall from Sec. 2.2.2 that a given torsion angle (either dihedral or improper) might contribute several terms to the potential energy, each with a different set of force field parameters. This is handled by having the particular `"dihed"` or `"impr"` torsion angle index a consecutive sequence of elements in the corresponding force field parameter array `"dihedprm"` or `"imprprm"`. The length of this sequence is controlled by the `mult` member of the `MD_TorsPrm` element, which should always give the count of the remaining elements in the parameter set, starting with itself.

For example, suppose that the `"dihedprm"` array has in its kth entry a set of dihedral parameters with multiplicity 1. This would mean (permitting an abuse of C language notation) that:

```
dihedprm[k] == { ... , mult == 1, ... }
```

Now suppose that the next array element begins a set of parameters for a dihedral interaction with multiplicity 4. This means that the parameters would be stored in the next four entries with:

```
dihedprm[k+1] == { ... , mult == 4, ... }
dihedprm[k+2] == { ... , mult == 3, ... }
dihedprm[k+3] == { ... , mult == 2, ... }
dihedprm[k+4] == { ... , mult == 1, ... }
```

In case the dihedral from the jth entry of the `MD_Tors "dihed"` array uses this particular multiplicity 4 parameter set, it would index the first entry of the `"dihedprm"` sequence:

```
dihed[j] == { ... , prm == k+1, ... }
```

2.3.3 Nonbonded exclusions

Nonbonded exclusions are vaguely expressed as "the exclusion of pairwise interactions involving atoms that already interact within some molecular bond." However, this notion must be defined more precisely when initializing an MD simulation.

Software packages for MD simulation generally define some *exclusion policy*. This concept goes beyond the scope of the molecular representation enabled by the `mdtypes.h` header file definitions. The engine would need this supplied as a parameter through some additional engine data array not already discussed. The NAMD software package defines the `"exclude"` parameter for use in simulation configuration files to be set to one of the following values:

`"none"` no nonbonded interactions are to be excluded based on atomic bonds

`"1-2"` exclude nonbonded interactions between atoms that are covalently bonded (in other words, pairs that are in a 2-atom sequence)

`"1-3"` exclude nonbonded interactions between atoms that are covalently bonded or that are both covalently bonded to the same atom (in other words, pairs that are in a 3-atom sequence)

`"1-4"` exclude nonbonded interactions between atoms that are covalently bonded or that are both covalently bonded to either the same atom or to a pair of covalently bonded atoms (in other words, pairs that are in a 4-atom sequence)

`"scaled1-4"` exclude `"1-3"` interactions and modify the interaction parameters between pairs of atoms that are the endpoints of a 4-atom sequence to (this would involve the use of `"emin14"` and `"rmin14"` members in `MD_AtomPrm` and `MD_NbfixPrm` data types along with using the ϵ_{14} scaling parameter in the electrostatic potential)

Notice that `"1-4"` and `"scaled1-4"` both include the other exclusion policies as subsets, and there is a nesting of exclusion policy subsets: `"1-3"` contains `"1-2"` which contains `"none"` (the empty set). Also notice that the exclusion policies are based entirely on sequences of covalent bonds (as is described by the `"bond"` array), *not* by the other bonded types. This means that when employing a `"1-3"` exclusion policy, any pairs of atoms appearing in the same `"angle"` array element would be excluded since an angle must involve a sequence of 3 covalently bonded atoms, however, there might be additional exclusions since there might exist a sequence of 3 covalently bonded atoms that do not also define an angle. The most used exclusion policy for modern MD simulation is `"scaled1-4"` because this best takes into account quantum mechanical effects by modifying the nonbonded parameterization for atoms that form the endpoints of torsion angles.

The `"excl"` array of data type `MD_Excl` is intended to specify additional nonbonded exclusions beyond those indicated by the exclusion policy. The engine receives a value for the exclusion policy parameter, then, based on the content of the `"bond"` array of type `MD_Bond`, builds an exclusion table. The exclusion table is supplemented by the content of the `"excl"` array. This means that it would be possible to define, say, a `"1-3"` exclusion policy explicitly through the `"excl"` array, it should not be necessary to do so. Note that defining a `"scaled1-4"` exclusion policy does require the use of an additional engine parameter. Although setting up an exclusion table entails quite a bit of work by the engine, there are additional libraries within the MDX framework that provide routines to automate this process. See Sec. **??** for details.

2.3.4 Initialization using the MDIO library

The front end is responsible for initializing the force field parameter and molecular topology arrays as well as providing initial positions and velocities. Fortunately, most of the effort in setting up these arrays, along with properly setting up the indexing of the force field parameter elements by the topology elements, is automated by high-level file reading routines using the MDIO library from MDX. These library routines presently read a very limited set of MD files, mostly a subset of the input files that NAMD supports, including:

- X-Plor force field parameter files,
- X-Plor protein structure files (PSF), and
- PDB coordinate files (`ATOM` and `HETATOM` records, ignoring all but the coordinate data).

More information about the MDIO library routines can be found in the `mdio.txt` documentation. Generating the data in these files can be done, at least in part, by external programs such as the `psfgen` utility included with NAMD. Creating these data files from scratch for a new system is beyond the capabilities of MDX.

Chapter 3

Guide to Using Front End Interface

The front end is responsible for reading and writing files, configuring and invoking the engine, and post-processing the results. To be able to use the front end interface, include the `mdfront.h` header file into the front end source code and link the executable to the MDAPI library. The details for building and linking to the current MDAPI implementation are presented in Sec. 1.3.

This chapter provides guidelines for using the front end interface, demonstrating basic and advanced features. Source code examples show how one might use the MDAPI data structures and functions. A complete presentation of the type definitions, function prototypes, and function calling semantics is deferred until chapter 6, with the material pertinent to the front end given in Secs. 6.1 and 6.2.

3.1 Brief Summary

The main responsibility of the front end is to setup the data used by the engine, then to run the simulation by calling `MD_run()` one or more times, and finally to perform any post-processing or output of results from the simulation. Most of the API routines deal with various ways of communicating data between the front end and the engine. There are alternative ways of communicating data, some of which improve performance and decrease memory usage.

The front end interface is object oriented. All functions operate on an *engine* object of type `MD_Engine`, with the leading argument a pointer to this object. The most basic functions, along with their general order of use, are:

- `MD_init()` — constructor (essential)
- `MD_idnum()` — provides ID number for a named data array (essential)
- `MD_write()` — write to an engine data array (some alternatives)
- `MD_run()` — run the simulation (essential)
- `MD_read()` — read from an engine data array (some alternatives)
- `MD_done()` — destructor (essential).

The interface allows running one engine with each `MD_Engine` object. It is possible for a single front end to run multiple simulations by managing several `MD_Engine` objects. (If the same engine code is being invoked for each `MD_Engine` object, then the engine code needs to be reentrant.)

There are some initial concepts that need clarification. In order to perform a simulation, an engine needs data (a lot of it) to be supplied by the front end. Since the data varies quite a bit depending on the type of simulation and computational methods employed, hard-coded variable names for the data arrays would be a poor choice. These would require every engine to provide all functionality desired (or at least define the entire set of variable names). Furthermore, the introduction of any new features or methods would then require a new version of the interface, as well as updates to all front ends and engines. Instead of using hard-coded variable names, data arrays are identified by string names. There is a core set of names that a front end can expect an engine to define (see Chap. 5) for the data that is common to most molecular dynamics simulations. This provides a basic level of interoperability between front ends and engines, while also enabling the development of more advanced front ends that can take advantage of features offered by specialized engines.

Advanced front ends cannot afford to yield control indefinitely (e.g. imagine a GUI application that is unresponsive to the user). At the same time, advanced implementations of the MDAPI layer are expected to connect a front end to a remote engine. For instance, the front end might be VMD running on a graphics workstation connected to the NAMD engine running remotely on a beowulf cluster. This requires allowing the front end to asynchronously invoke the engine. Interface routines that require a response from the engine are defined with nonblocking calling semantics, in which the routine returns immediately but is not necessarily finished with its task. There are interface routines provided to test whether a nonblocking call has finished or to wait for the call to finish. In order to simplify the MDAPI semantics, only one outstanding call to a nonblocking routine is allowed, and the call must finish before calling other interface routines. The following routines are defined with nonblocking semantics:

- `MD_init()`
- `MD_read()`/`MD_readsub()`
- `MD_direct()`
- `MD_update()`
- `MD_run()`

These are all routines that require data to be sent from the engine. Note that `MD_write()`/`MD_writesub()` do not need to block because they modify buffers local to the front end. The engine is not guaranteed to see the buffer modifications until the front end calls `MD_updata()` or `MD_run()`.

The call to `MD_run()` hides the majority of the work done by the engine. In order to improve performance for long simulations, it is best to allow the engine to continue to run for a large number of steps. However, the front end will still need feedback during the simulation, including intermediate trajectories to log for later imaging and analysis and quantities to be monitored to ensure the stability of the simulation. An advanced front end might want to enable realtime imaging of the simulation or additional interactions between the user and the simulation, such as interactive molecular dynamics to allow the user to supply an external force to the system via a controller. The MDAPI enables the communication of data between the front end and engine during a running simulation through the use of callbacks. A callback is simply an entry point back into the front end (i.e. a function) that is called during `MD_run()`, supplying the front end with data from the simulation or providing the engine with new data.

There are predefined data types (in `mdtypes.h`) that facilitate the computation of forces and the integration of a system of atoms. An engine is also permitted to define new types (i.e. C structures) to be offered through

the MDAPI with routines that allow the front end to "see" the definition of a previously unknown type and access its members.

Most of the interface routines return zero on success or `MD_FAIL` on failure, in which case an error status value is set (similar to `errno` from the C library). Additional routines permit monitoring the error state and attempting error recovery.

3.2 Basic Features

The interface routines that are essential and most easily used are presented in this section.

3.2.1 Initialization and cleanup

The front end must allocate the engine object and then call its constructor before using any other interface routine.

```
MD_Engine *eng;

eng = (MD_Engine *) malloc(sizeof(MD_Engine));
MD_init(eng, "engine", 0, engine_init, engine_done);
MD_wait(eng);
```

The arguments to `MD_init()` listed after the engine object are the engine name, the flags, the engine initialization routine, and the engine cleanup routine. Note that the `MD_init()` and `MD_done()` routine perform initialization and cleanup for the interface layer, and these routines in turn invoke the constructor and destructor for the particular engine. The flags argument is passed on to the *engine_init* constructor, where the interpretation is defined by the engine.

The string is the name of the engine. For an implementation of MDAPI that supports dynamic loading of engines, the name should have the following form:

```
[[user@]hostname:]pathname
```

In this case, pass `NULL` in place of *engine_init* and *engine_done*. The MDAPI layer will then attempt to dynamically load the engine named *pathname* on the remote machine *hostname* logged in as *user*.

If *engine_init* and *engine_done* are not `NULL`, then these function pointers are invoked as engine constructor and destructor, with the name retained as just a label. In this case, the engine is already linked to the front end executable.

The call to `MD_wait()` demonstrates the simplest use of the synchronization routines, in which the front end blocks to wait for `MD_init()` to finish. More details regarding synchronization are presented in the following section.

After the front end has finished performing the simulation using the engine object, the destructor should be invoked.

```
MD_done(eng);
free(eng);
```

Remember to free the memory used by the engine object after calling the destructor. The call to `MD_done()` will invoke the *engine_done* cleanup routine that was established by `MD_init()`.

3.2.2 Synchronization

Front end interface routines that require a response from the engine, as opposed to those that can be handled by the MDAPI layer, are provided with nonblocking semantics. This enables advanced front end implementations to control a remote engine. The nonblocking routines begin a communication exchange with the engine and immediately return control to the front end. Synchronization is performed by calling `MD_test()` to test for completion of the communication exchange and `MD_wait()` to wait until the communication is received. After calling a nonblocking routine, no other front end interface routines (besides `MD_test()` and `MD_wait()`) may be called until communication has finished.

The `MD_test()` routine immediately evaluates to true (nonzero) if the communication has finished and otherwise returns false (zero). This can be used within event loops to poll the state of the engine object, so that the front end does not have yield control indefinitely for the engine to respond.

```
while (events) {
  if (MD_test(eng)) {
    /*** finished previous MDAPI routine, now call another ***/
  }

  /*** check other events ***/

}
```

Once `MD_test()` evaluates to true, it will continue to return true until another nonblocking MDAPI routine is called.

The `MD_wait()` routine blocks the front end, waiting for the communication exchange to finish. This is more straightforward to use in a simple front end that has no other responsibilities than to setup and run a simulation. The earlier example is repeated.

```
MD_init(eng, "engine", 0, engine_init, engine_done);
MD_wait(eng);
```

The `MD_wait()` routine, like most other MDAPI routines, indicates an error by returning `MD_FAIL` and success by returning zero. See the next section for details.

3.2.3 Error handling and recovery

The engine object maintains an error condition number to report an error, similar to the use of `errno` by the C standard library. The error condition number is available through the `MD_errnum()` routine.

```
errnum = MD_errnum(eng);
```

There are predefined error constants in `mdcommon.h`, (included automatically by including `mdfront.h`). The engine is also permitted to define its own error conditions that might occur while running the simulation through `MD_run()`.

Most of the front end interface routines return an integer value to indicate success or failure. These return values should always be checked in order to make the front end robust (although most of the code examples here will not do so). Generally, success is indicated by zero and failure is indicated by `MD_FAIL`. If the routine fails, then the internal error condition number has been set.

Simple use of the engine object might check for errors and report them.

```
if (MD_init(eng, "engine", 0, engine_init, engine_done)
    || MD_wait(eng)) {
  fprintf(stderr, "ERROR %d: %s\n", MD_errnum(eng), MD_errmsg(eng));
  exit(1);
}
```

The `MD_errmsg()` routine is used to return a text string description of the error.

Error recovery is also possible. An error condition is *fatal* if it is unrecoverable, in which case the use of the engine object should be terminated by calling `MD_done()`. If the error condition is nonfatal, then the internal error number can be reset with the `MD_reset()` routine and use of the engine object can continue. The following example shows detecting and attempting to reset the error condition after a failed call to `MD_setlen()`.

```
if (MD_setlen(eng, pos_id, n)) {
  errnum = MD_errnum(eng);
  if (MD_reset(eng)) {
    fprintf(stderr, "FATAL ERROR %d: %s\n", errnum, MD_errmsg(eng));
    MD_done(eng);
    return MD_FAIL;  /* return failure to previous routine */
  }

  /*** use errnum to deal with error and continue ***/

}
```

The `MD_reset()` routine returns zero if successful. Since this has the side effect of resetting the internal error condition, it is necessary to first obtain the error number in order to deal with a nonfatal error.

3.2.4 Accessing data arrays

An engine offers the front end access to one-dimensional data arrays used to communicate data between front end and engine. A single datam value (scalar) may be represented using an array of fixed length one. A multi-dimensional array can be represented by folding it into a single dimension. The front end is resposible for doing any necessary file I/O and then initializing the data arrays for the engine. During and perhaps also after the simulation, the front end will want to access the data arrays in order to provide results to the user.

Rather than having dedicated routines through which to access individual data arrays or having hard-coded names for these arrays, the data arrays are accessed by using string names. This allows the interface to be more flexible so that the MDAPI can be used with specialized MD engines. For the sake of interoperability between front ends and engines, there are particular names that should be used for data arrays providing basic MD functionality. The guidelines for these array names, along with their expected data types and units of measure, are presented in Chap. 5.

The full list of names of data arrays offered by the engine is available through the `MD_namelist()` routine.

```
const char **name;
int32 len;

name = MD_namelist(eng, &len);
```

This routine returns the name list as an array of strings, with its length returned to the variable `len`.

Although names provide the advantage of flexibility, they are not that fast or easy with which to work. Each data array has an identification (ID) number associated with it that will be used as its handle. The ID number is returned by calling `MD_idnum()`. The following example shows an attempt to obtain the ID of the position array.

```
int32 pos_id;

pos_id = MD_idnum(eng, "pos");
if (pos_id == MD_FAIL) {
  fprintf(stderr, "ERROR: there is no position array\n");
  exit(1);
}
```

Note that the data array names are case sensitive. There is also a function that maps the other direction, from data ID numbers to names. To continue the previous code example, the following will print *pos* to standard out.

```
printf("%s\n", MD_name(eng, pos_id));
```

Each engine data array also has a set of attributes associated with it. The attributes of a data array are its type, used length, maximum allocation, and front end access permissions. The attributes are stored in the special interface data structure `MD_Attrib` and returned by calling `MD_attrib()` with the array ID number. The following example demonstrates obtaining and accessing the position array attributes.

```
MD_Attrib attr;

attr = MD_attrib(eng, pos_id);
printf("position array has attributes:\n");
printf("  type = %d\n", attr.type);
printf("  used length (number of elements) = %d\n", attr.len);
printf("  maximum allocation (number of elements) = %d\n", attr.max);
printf("  access permissions = %x\n", attr.access);
```

The type number, similar to an ID number, associates the data type with an `int32` value. The position array should be of some predefined 3D-vector type, so its `type` attribute is equal to either `MD_DVEC` or `MD_FVEC`. See Sec. 2.2 for more information on the predefined data types defined in `mdtypes.h` (included automatically by including `mdfront.h`). The `len` and `max` attributes are given in number of elements (not bytes); it is always the case that $0 \leq$ `len` $\leq$ `max`.

The `access` attribute indicates permissions that the engine has granted to the front end. These determine how the front end may use the array, including resizing (which in turn modifies `len` and/or `max`), reading, writing, and availability in callbacks. The access permissions are stored as a bit field, most easily discerned

using the %x option to printf(); the individual access permission constants are defined in mdcommon.h and discussed in Sec. 6.1.2. For the position array, the access permissions should include at least reading, writing, and setting the length.

```
printf("access permissions for \"%s\":\n", MD_name(eng, pos_id));
printf("  read: %s\n", (attr.access & MD_READ ? "yes" : "no"));
printf("  write: %s\n", (attr.access & MD_WRITE ? "yes" : "no"));
printf("  set length: %s\n", (attr.access & MD_SETLEN ? "yes" : "no"));
printf("  set maximum: %s\n", (attr.access & MD_SETMAX ? "yes" : "no"));
```

The engine does not automatically know how long the data arrays need to be. The front end must set the length of most of the data arrays, which can be accomplished by calling the MD_setlen() routine with the ID number of the array and the new length. The length of the position array should be set to the number of atoms in the system.

```
MD_setlen(eng, pos_id, natoms);
```

Doing this changes the position array attribute len to the value natoms. If the position array did not already have enough elements allocated, then the memory allocation of the array is resized to accomodate (assuming MD_SETMAX permission on the array), which changes the position array attribute max. There are two other related interface calls: MD_setmax() changes the maximum memory allocation, and MD_resize() changes both the length and the maximum allocation.

The front end also needs to be able to write data values into an array and read data values from an array. The simplest routines for this purpose are MD_write() and MD_read(), each of which takes the array ID number and an array buffer along with its length provided by the front end. The MD_write() routine copies the contents of the front end array buffer into the engine data array, whereas MD_read() routine copies the contents of the engine data array into the front end array buffer. Continuing with the position array, after setting the correct length, the front end must provide the initial atom positions. Assuming that the pos array declared below contains the initial positions (and that the type attribute is in fact MD_DVEC), then the position array is initialized as follows.

```
int32 natoms = MD_attrib(eng, pos_id).len;
MD_Dvec *pos = (MD_Dvec *) malloc(natoms * sizeof(MD_Dvec));

/*** read initial positions from data file ***/

MD_write(eng, pos_id, pos, natoms);
```

After the simulation is finished, the front end might want to retrieve and save the final atom positions.

```
MD_read(eng, pos_id, pos, natoms);
MD_wait(eng);
```

Note that since MD_read() requires data communication from the engine, it has nonblocking semantics defined. However, MD_write() can store the data with the front end until the engine needs to know it, so does *not* have nonblocking semantics.

There are alternative ways for the front end to access engine data arrays, which will be discussed later in Sec. 3.3.

3.2.5 Running the simulation

Once the engine has been provided with its data, the front end can run the simulation using the `MD_run()` routine. This works by integrating the system of atoms for a specified number of steps, for which some of the computational aspects are presented in Chap. 2.

The MDAPI layer keeps a step number counter that is incremented after each time step. The value of the step number counter defaults to zero but can be initialized by calling `MD_firststep()`. Suppose the simulation is a continuation of a previous one-nanosecond simulation using a time step of $\Delta t = 1.25$ fs. Then it is desirable to use `MD_firststep()` to keep the step numbering consistent.

```
MD_firststep(eng, 800000);
```

After some amount of stepping, the step number counter can be checked using `MD_stepnum()`.

```
int32 stepnum;

stepnum = MD_stepnum(eng);
```

Note that the engine does not really care what value the step number counter has, since it is just a label.

The front end calls `MD_run()` to run the simulation for some specified number of steps. Building on previous code examples, suppose that the simulation should run for an additional nanosecond using time step $\Delta t = 1.25$ fs and the position coordinates should be saved after every 50 steps for analysis. This can be done using the following simple code loop.

```
int32 totalsteps = 800000;
int32 incrstep = 50;
int32 numsteps = 0;
int32 firststep;

/*** assume engine knows that dt=1.25 fs ***/

firststep = MD_stepnum(eng);
while (numsteps < totalsteps) {
  if (MD_run(eng, incrstep, 0) || MD_wait(eng)) {
    fprintf(stderr, "Error returned by MD_run()\n"
        "ERROR %d: %s\n", MD_errnum(eng), MD_errmsg(eng));
    exit(1);
  }
  if (MD_read(eng, pos_id, pos, natoms) || MD_wait(eng)) {
    fprintf(stderr, "Error returned by MD_read()\n"
        "ERROR %d: %s\n", MD_errnum(eng), MD_errmsg(eng));
    exit(1);
  }

  /*** save position array to file ***/

  numsteps += incrstep;
  if (MD_stepnum(eng) != firststep + numsteps) {
    fprintf(stderr, "Invalid step number counter\n");
```

```
    exit(1);
  }
}
```

Note that `MD_run()` has nonblocking semantics.

Although the code loop works correctly, it is not most efficient. The front end does no work while the engine runs; more importantly, the engine performs no computation while the front end saves the position data. Although this is not a big concern if the front end and engine are running as a single process on one CPU, it is a poor utilization of separate front end and engine processes running on different processors. Furthermore, for a remote front end and engine, every 50 steps of computation is interrupted by two non-overlapping communication exchanges, with one an array of position coordinates of length N. Finally, there is also some overhead double buffering the position array and restarting the `MD_run()` routine 16,000 times. All of these deficiencies will be addressed through the use of advanced features presented in the next section.

3.3 Advanced Features

The interface routines that provide high performance and improve flexibility are presented in this section.

3.3.1 More ways to access data arrays

There are several alternative ways to access data arrays besides `MD_read()` and `MD_write()`. A generalization of these routines is provided that copies to and from a contiguous sub-array of an engine data array. Like the former routines, the `MD_readsub()` and `MD_writesub()` also require `MD_READ` and `MD_WRITE` access permission, respectively, on the array. Both of these routines accept an additional argument that indicates the index of the first element of the sub-array, using 0-based indexing. For example, the following loop using `MD_readsub()` reads from the position array, introduced in the previous section, but here does so in 20-element chunks.

```
const int32 CHUNK_LEN = 20;
int32 natoms = MD_attrib(eng, pos_id).len;
int32 nchunks = natoms / CHUNK_LEN;
int32 nremainder = natoms % CHUNK_LEN;
int32 first = 0;
int32 k;
MD_Dvec *posbuf;

posbuf = (MD_Dvec *) malloc(natoms * sizeof(MD_Dvec));
for (k = 0;  k < nchunks;  k++) {
  MD_readsub(eng, pos_id, posbuf + first, CHUNK_LEN, first);
  MD_wait();
  first += CHUNK_LEN;
}
if (nremainder > 0) {
  MD_readsub(eng, pos_id, posbuf + first, nremainder, first);
  MD_wait();
}
```

Note that the destination pointer into the front end position buffer must point to the address receiving the data. The `MD_writesub()` works similarly, except that the pointer is the source from which to copy.

The MD_readsub() has nonblocking semantics, whereas MD_writesub() does not. Clearly, this example is sub-optimal for reading the entire array. The improvement in performance occurs when the front end needs only a sub-array from an engine data array.

Another way to access a data array is directly through a pointer to its buffer. The MD_direct() routine returns a void * pointer to the engine data buffer (or, in the case of communicating to a remote engine, to the local copy of the engine data buffer). The array must permit MD_DIRECT access in order to acquire this pointer. An example follows involving reading the position array.

```
int32 natoms = MD_attrib(eng, pos_id).len;
int32 isdirect = MD_attrib(eng, pos_id).access & MD_DIRECT;
MD_Dvec *posbuf;

if (isdirect) {
  posbuf = (MD_Dvec *) MD_direct(eng, pos_id);
  MD_wait();
}
else {
  posbuf = (MD_Dvec *) malloc(natoms * sizeof(MD_Dvec));
  MD_readsub(eng, pos_id, posbuf, natoms, 0);
  MD_wait();
}

/*** save positions to a file ***/

if (!isdirect) {
  free(posbuf);
}
```

Using direct access avoids double buffering the position array data, however, a robust front end must fall back to MD_read() if direct access is not permitted. The MD_direct() routine has nonblocking semantics since, for a remote engine, a local copy of the engine array must be made. In the event that the front end and engine share the same memory space, the call to MD_direct() simply returns the array buffer pointer, so it finishes immediately.

Care must be taken with MD_direct() array access. The array pointer is likely to be invalidated if any memory reallocation is performed on the array. This means that MD_direct() should not be called on an array until after any resizing with MD_setlen() and related commands has been performed. Also, the buffer from MD_direct() is not necessarily valid following an MD_run() call. If the engine is remote, the buffer will contain previous array data rather than current data. Also, if the array has MD_ESETMAX enabled, it is possible that the engine performed a memory reallocation on the array during the MD_run() call. So the general rule of thumb for direct array access is to always re-invoke MD_direct() on an array following an MD_run() call.

When using a remote engine, the communication for MD_direct() is by default one-sided, with the data sent from the engine to the front end. Direct access support for modifying an engine data array also requires the use of the companion routine MD_setmod().

```
if (isdirect) {
  posbuf = (MD_Dvec *) MD_direct(eng, pos_id);
  MD_wait();
}
else {
```

```
  /*** use MD_read() ***/
}
posbuf[0].x = 0.0;    /* modify position of first atom */
if (isdirect) {
  MD_setmod(eng, pos_id);  /* tell engine about modification */
}
```

Calling `MD_setmod()` indicates that the array has been modified by setting the `MD_MODIFY` "dirty bit" status flag on the array. This status flag signals the engine that the data in its array buffer has changed; for communication with a remote engine, it also signals the MDAPI layer that the array contents need to be sent to the engine on the next `MD_run()` call, thus having the array data communicated in both directions. Note that the `MD_MODIFY` status flag is set automatically by calls to `MD_write()` and `MD_writesub()` and also by calls to `MD_share()` and `MD_unshare()` discussed next.

Under certain conditions, the front end is permitted to supply the buffer space for an array with `MD_share()`. Doing so is especially good practice for the large data arrays that the front end must initialize from input files, reducing the memory usage for large systems by close to one-half.

```
MD_Dvec *posbuf;
int32 len = natoms;
int32 max = natoms;

posbuf = (MD_Dvec *) malloc(max * sizeof(MD_Dvec));

/*** initialize the atom positions ***/

if (MD_attrib(eng, pos_id).access & MD_SHARE) {
  MD_share(eng, pos_id, posbuf, len, max);
}
else {
  MD_setlen(eng, pos_id, len);
  MD_write(eng, pos_id, posbuf, len);
}

/*** run the simulation ***/

MD_done(eng);    /* finished with engine object */
free(posbuf);    /* must free allocated memory */
```

As the example shows, the data array must permit `MD_SHARE` access in order to call `MD_share()` successfully. Array buffer length (`len`) and maximum allocation (`max`) are both specified. After calling `MD_share()`, the engine has control over the contents of the buffer space, and the front end should not directly modify the contents without first calling `MD_direct()`. The data array attributes are modified by the new `len` and `max` values set for the buffer; also, `MD_SETMAX` and `MD_ESETMAX` are both disabled so that neither the front end nor the engine is permitted to resize the memory allocation for this buffer. As the example shows, the front end regains control over the buffer after `MD_done()` and is still responsible for freeing whatever memory it allocates. Note that `MD_SHARE` access is enabled only if the current array buffer is `NULL` (i.e. the data attributes have `len` = `max` = 0), and a call to `MD_setlen()` that extends the array buffer (`max` > 0) will *disable* `MD_SHARE` access.

After a successful call to `MD_share()`, the front end is permitted to regain control over the buffer by calling `MD_unshare()` (i.e. `MD_share()` also enables the `MD_UNSHARE` access flag).

```
MD_share(eng, pos_id, posbuf, len, max);

/*** stuff is done ***/

posbuf = (MD_Dvec *) MD_unshare(eng, pos_id);
```

The call to `MD_unshare()` returns the pointer to the array buffer provided to `MD_share()` and also resets the data array access attributes back to their previous state before the `MD_share()` call.

Observe that reading from engine data arrays requires communication from the engine, so these types of routines have nonblocking semantics defined. On the other hand, writing to engine data arrays can be performed by the front end to local buffer space, so these types of routines do *not* have nonblocking semantics. Communication of buffered writes back to the engine is generally delayed until the next `MD_run()` invocation. The data array modifications can be written back to the engine sooner by calling `MD_update()`.

```
MD_update(eng);
MD_wait(eng);
```

3.3.2 Establishing callbacks

Callback routines are entry points into the front end that are called by the engine (via the MDAPI layer) during a running simulation, executed by `MD_run()`. The main idea is for the front end to receive intermediate results from and possibly to also send data to the engine, all without halting its computation. Judicious use of callbacks can improve performance substantially. For a remote engine, this permits useful computation to be performed concurrently with file writing or realtime visualization of results. In the simplest case of a front end invoking a single-threaded engine within the same process, there is still the advantage of direct access by the front end to data arrays without double buffering and without the overhead required to stop and restart the `MD_run()` routine.

Three types of callbacks can be registered with the interface, distinguished by data communication and processing. Message callbacks receive a text string from the engine providing status of the simulation, invoked at the discretion of the engine to convey information to the front end. Standard callbacks and force callbacks are able to receive engine data arrays. Standard callbacks are processed at the completion of a time step and are intended to checkpoint the simulation, save or visualize trajectories, or monitor energy. The front end designates for each standard callback the frequency for which it should be invoked, given in number of steps. Force callbacks are processed during each force computation and are intended for the front end to provide external forces to the simulation. This enables support for interactive molecular dynamics or to simply extend the force field beyond the capabilities of a particular engine.

Each type of callback receives in its leading argument `void *info` intended for front end state information. A message callback must have the following prototype.

```
int32 msgcb(void *info, const char *msg, int32 stepnum);
```

The `msg` argument receives a string, and `stepnum` receives the step number counter. The following simple example shows registering a message callback through `MD_msgcallback()` that prints the status message to standard output.

```
int32 print_engmsg(void *info, const char *msg, int32 stepnum) {
  const char *engname = (const char *) info;
  return printf("%s: step %d: %s\n", engname, stepnum, msg) < 0;
```

```
}

/*** registering message callback ***/

const char *engname;

engname = MD_engine_name(eng);
if (engname == NULL) engname = "engine";
MD_msgcallback(eng, print_engmsg, engname);
```

The `info` argument receives a string providing the name of the engine. Note that `MD_engine_name()` returns the engine name string originally provided to `MD_init()`. The example callback routine `print_engmsg()` prints the message preceded by the engine name and the step number. The return value from a callback routine is important: success is indicated by zero and failure by nonzero. However, a nonzero return value will result in stopping the `MD_run()` (or `MD_test()`/`MD_wait()`) call with an error. In the example, an output error will terminate the simulation.

Multiple callbacks of each type can be registered and are invoked in sequence. It is even possible to register the same callback routine multiple times. For message callbacks, the callback routines will each be invoked with the same string message. This feature might be useful for registering one callback that prints to standard output and a second callback that logs messages to a file. Callbacks may later be un-registered. Continuing the previous example, the message callback is un-registered by calling `MD_msgcallback_undo()` as follows.

```
MD_msgcallback_undo(eng, print_engmsg);
```

This will remove each instance of the `print_engmsg()` callback, even multiple registrations of it. Using `NULL` in place of the function pointer will undo *all* callback registrations.

Standard and force callbacks can each receive multiple engine data arrays. The arrays are specified through the `MD_Cbdata` type. An array of `MD_Cbdata` is passed to the registration routine, in which each element of the array specifies some sub-array of an engine data array to be received. The front end must typically initialize the first four members of each `MD_Cbdata` element:

- `idnum` — the data array identification number
- `access` — the kind of access requested
- `nelems` — number of elements in sub-array
- `first` — index of first element to appear in sub-array

The `idnum` accepts the values returned by `MD_idnum()`. The `nelems` and `first` members are used in almost the same way as in calls to `MD_writesub()` and `MD_readsub()`. The only difference is that `nelems` can be set to -1, meaning "the rest of the elements in the array starting at the index given by `first`." This enables a callback to receive data from an array whose length might have been changed by the engine. The `access` member is set to some bitwise ORed combination of access rights, selecting between `MD_CBREAD`, `MD_CBWRITE`, and `MD_CBSHARE` for standard callbacks or `MD_FCBREAD`, `MD_FCBWRITE`, and `MD_FCBSHARE` for force callbacks. Read access provides to the callback meaningful data to be used, write access obtains data from the callback, and the combination of read and write access provides meaningful data to be modified. Share access allows the callback routine to provide its own data buffer to the engine for copying, allowing the front end to provide a variable length array to the engine; however, the array semantics are changed in such a way that it is not possible to combine share access with read and write access. Note that the referenced engine data array must already permit the type of access being requested.

The callback routine will receive the same MD_Cbdata array as used when registering it, so the MD_Cbdata arrays allocated by the front end must persist during the simulation. Two additional members within each MD_Cbdata element are made available to the callback routine:

- data — points to the data
- attrib — the attributes of this engine data array

The attrib member is of type MD_Attrib, providing the attributes. In particular, attrib.len contains the length of the array, useful if nelems was unknown with value -1, and attrib.type contains the type number, useful to distinguish between single precision MD_Fvec and double precision MD_Dvec. The data pointer is of type void * so must be typecast to the correct data type. If the front end and engine are in the same memory address space, then data will simply be a pointer directly into the engine data array. In the case of a remote engine, data will point to a buffer large enough to contain the requested sub-array elements. Note that data should be indexed starting from 0 through nelems $-$ 1 (or through attrib.len $-$ first $-$ 1 if nelems $= -1$).

Access rights MD_CBSHARE and MD_FCBSHARE change the semantics. In this case, data = NULL and nelems = first = 0 is received in that MD_Cbdata element by the callback routine. It is up to the callback to provide a data buffer to the engine. The sub-array semantics still apply, with updates to nelems and first describing what portion of the engine data array is being provided and possibly resizing it. The front end retains ownership of the memory it sends via the data pointer, with copy semantics like MD_write(), which means that the same memory buffer can be reused on subsequent callbacks, and the front end is eventually responsible for freeing the memory allocation for this buffer. Although share access cannot be combined with read or write access, it is possible to reference the same array using two different MD_Cbdata elements. For instance, assuming that an engine data array permitted both read and share access to standard callbacks, the first MD_Cbdata element could set MD_CBREAD access to receive the previous values and the second MD_Cbdata element could set MD_CBSHARE access to provide a data buffer back to the engine with updated values. Note that data array resize changes are only permitted if the front end has already been granted access to do so, and will otherwise result in an error.

The first of two examples that follow establishes a standard callback that receives the position coordinates and saves them for later analysis. This is the more efficient version of the run-loop presented at the end of Sec. 3.2.

```
/* the callback */
int32 save_pos(void *info, MD_Cbdata *cbd, int32 cbdlen, int32 stepnum) {
  FILE *trajfile = (FILE *) info;
  MD_Dvec *pos = cbd[0].data;
  int32 natoms = cbd[0].attrib.len;

  /*** write pos array into trajfile ***/

  return 0;    /* if successful */
}

/* main code section */
int32 totalsteps = 800000;
int32 incrstep = 50;
int32 pos_id = MD_idnum(eng, "pos");
#define CBDLEN 1
MD_Cbdata cbd[CBDLEN];
FILE *trajfile;    /* points to trajectory file */
```

```
cbd[0].idnum = pos_id;
cbd[0].access = MD_CBREAD;
cbd[0].nelems = -1;
cbd[0].first = 0;

/*** open trajfile ***/

MD_callback(eng, save_pos, trajfile, cbd, CBDLEN, incrstep);
MD_run(eng, totalsteps, 0);
MD_wait(eng);

/*** close trajfile ***/
```

The `save_pos()` function shows the prototype of a standard callback, accepting the `info` argument as discussed previously, followed by the `MD_Cbdata` array, its length, and the current value of the step number counter. The call to `MD_callback()` to register `save_pos()` as a standard callback is sent as arguments the function pointer `save_pos`, the value of the "`info`" pointer, the `MD_Cbdata` array, its length, and the step counter increment. In this case, `save_pos()` will be invoked during the simulation every 50 time steps and receive the position array along with the trajectory file handle. Using the default first step number 0 to be assigned to the initial position coordinates, the call to `MD_run()` will result in saving the trajectories after $50, 100, 150, \ldots, 800,000$ steps. Changing the `MD_run()` call to

```
MD_run(eng, totalsteps, MD_CBFIRST);
```

will also save the initial trajectories at step 0. The `MD_CBFIRST` flag will make sure to process the standard callbacks at the initial step number value. Standard callbacks can each be registered using a different step increment. This means that registering a second callback with a step increment of 75 will result in the second routine being called after $75, 150, 225, \ldots, 799,950$ steps, with both called at step numbers $150, 300, \ldots, 799,950$. Standard callbacks can be un-registered by calling `MD_callback_undo()` as was done for message callbacks.

The second example establishes a force callback to provide the system with external forces. Suppose that this callback routine applies a spring-type force to a single atom to bring it to a fixed location. This routine would need the position of the atom and would compute an additional force on that atom.

```
/* create a new type using related predefined data types */
typedef Spring_tag {
  MD_Bond bond;    /* special spring bond parameters */
  MD_Dvec r;       /* anchored end of spring */
  double dt;       /* time step */
} Spring;

/* the force callback */
int32 apply_force(void *info, MD_Cbdata *cbd, int32 cbdlen, int32 stepnum,
                  double frac) {
  Spring *spr = (Spring *) info;
  MD_Dvec *pos = (MD_Dvec *) cbd[0].data;
  MD_Dvec *force = (MD_Dvec *) cbd[1].data;
  double time = (stepnum + frac) * spr->dt;  /* ellapsed simulation time */

  /*** compute interaction between spr[0].r and pos[0] ***/
```

```
  /*** store result in force[0] ***/

  return 0;
}

/* main code section */
int32 pos_id = MD_idnum(eng, "pos");
int32 extforce_id = MD_idnum(eng, "extforce");
int32 atomindex = 42;    /* index of atom connected to special spring */
int32 natoms = MD_attrib(eng, pos_id).len;
#define FCBDLEN 2
MD_Cbdata fcbd[FCBDLEN];
Spring spr;
MD_Dvec *extforce;

/* allocate (and zero) external force buffer */
extforce = (MD_Dvec *) calloc(natoms, sizeof(MD_Dvec));
MD_share(eng, extforce_id, extforce, natoms, natoms);

/* setup force callback MD_Cbdata array */
fcbd[0].idnum = pos_id;
fcbd[0].access = MD_FCBREAD;
fcbd[0].nelems = 1;
fcbd[0].first = atomindex;
fcbd[1].idnum = extforce_id;
fcbd[1].access = MD_FCBWRITE;
fcbd[1].nelems = 1;
fcbd[1].first = atomindex;

/*** initialize spr ***/

MD_fcallback(eng, apply_force, &spr, fcbd, FCBDLEN);
MD_run(eng, totalsteps, 0);
MD_wait(eng);
```

The use of force callbacks is almost the same as that of standard callbacks. The `MD_fcallback()` routine does not need take a step number increment because force callbacks are invoked every time the force is evaluated. Observe that the `MD_Cbdata` array sent to `apply_force()` provides no more data than is necessary, rather than sending the entire position and external force arrays. The one additional argument received by the force callback routine is `frac`, which is defined as the fractional amount to be added to the step number counter so that multiplying by the time step gives the simulation time of the position approximation. Assuming that the initial position configuration is assigned step number 0, then the value of `time` computed in `apply_force()` is the ellapsed simulation time, as indicated. This is necessary if the computed force is time-dependent. Note that the value of `frac` depends on the integration method used. For leapfrog/Verlet integration, `frac` = 1.0. An integration method that requires multiple force evaluations per time step might require a different value for `frac` for each force evaluation. The `MD_fcallback_undo()` routine can be used to un-register force callbacks.

3.3.3 Using engine-defined types

Engine data arrays may be of any predefined data type, as presented in Sec. 2.2. The engine is also able to extend the interface by defining new data types. The routines discussed here enable the front end to use an unknown new data type.

Data types have string names in the same way that engine data arrays do. The full list of data type names is available through the `MD_type_namelist()` routine.

```
const char **typename;
int32 len;

typename = MD_type_namelist(eng, &len);
```

This routine returns the type name list as an array of strings, with its length returned to the variable `len`. The type name of a predefined type is simply the string of its hard-coded type name, for example, predefined data type `double` has type name `"double"` and `MD_BondPrm` has type name `"MD_BondPrm"`.

Working with data types is made easier by also associating a type number with each name. Type numbers have already been encountered, for example, the `type` member of `MD_Attrib`. Given the type name, the type number is available using the `MD_type()` routine.

```
int32 bondprm_type;

bondprm_type = MD_type(eng, "MD_BondPrm");
```

Note that type names are case sensitive. There is also a function that maps the other direction from type numbers to type names. To continue the previous code example, the following will print *MD_BondPrm* to standard output.

```
printf("%s\n", MD_type_name(eng, bondprm_type));
```

Besides providing an easily used identification scheme, type numbers also encode the byte size of an instance of the type.

```
printf("%s has size %d bytes\n",
    MD_type_name(eng, bondprm_type), MD_SIZEOF(bondprm_type));
```

The `MD_SIZEOF()` macro maps the type number to the size in bytes, similar to the C `sizeof` operator. In many situations, such as when performing memory management, knowing the size allows data types to be parameterized.

A defined data type can be viewed, much like a C `struct`, as an agglomeration of members of previously defined data types. The fundamental types that are not based on smaller data types are known as *primary types*. All other data types are known as *derived types*. The `MD_Member` type provides information regarding a single member of a data type. `MD_Member` contains the following members.

- `type` — the type number of the member
- `len` — the number of elements of this member

- `name` — the string name of this member

The `len` value allows the data type member to be defined as a (static) one-dimensional array. The `name` string is the name of the member, not the type. Since the type has some collection of members, it is described by an array of `MD_Member`. As an example, the primary type `double` has a one-element `MD_Member` array.

```
{ { MD_DOUBLE, 1, "" } }
```

Note that any primary type has a one-element `MD_Member` array, with `name` given by `""` (the empty string) since it is a "smallest" type that conceptually has no members. For another example, the derived type `MD_BondPrm` has a three-element `MD_Member` array,

```
{ { MD_DOUBLE, 1, "k" }, { MD_DOUBLE, 1, "r0" }, { MD_NAME, 2, "type" } }
```

that describes the following type definition.

```
typedef struct MD_BondPrm_tag {
  double k;
  double r0;
  MD_Name type[2];
} MD_BondPrm;
```

Note that types, such as `MD_Name`, `MD_Fvec`, and `MD_Dvec`, are treated as primary types. Also note that interface types, such as `MD_Attrib` and `MD_Member`, are *not* data types.

The `MD_Member` array for a defined data type is returned by the routine `MD_type_memberlist()`.

```
const MD_Member *mlist;
int32 mlistlen;

mlist = MD_type_memberlist(eng, bondprm_type, &mlistlen);
```

This allows the front end to "comprehend" an unknown data type by discovering its components. Another routine `MD_type_member()` enables the front end to access an individual member of a data type by name, given an object of that type.

```
MD_BondPrm bondprm;
double *r0;
MD_Member m;

/*** assume that bondprm has been initialized ***/

r0 = (double *) MD_type_member(eng, bondprm_type, &bondprm, "r0", &m);
printf("equilibrium length is %g\n", *r0);
```

The return value of `MD_type_member()` is `void *` so must be typecast. The arguments to `MD_type_member()` are the type number, a pointer to the object, the name of the member, and a pointer to a variable of type `MD_Member`. In this example, the variable `m` will return from the call with the following value.

```
{ MD_DOUBLE, 1, "r0" }
```

Two data types that engines are encouraged to offer are `Param` and `Result`. The `Param` data type is a collection of simulation parameters, including the time step, the dielectic constant, and the cutoff. The `Result` data type is a collection of computed quantities from the simulation, such as kinetic and potential energy and temperature. Note that `Param` and `Result` both gather together small data types (mostly `double` or `float`). This is an especially good idea for `Result`, since these values will probably be received by a frequently invoked callback.

The following example shows a callback that outputs the step number and total energy after every five steps. It can easily be expanded to do greater things.

```
/* info type for energy callback */
typedef EnergyInfo_tag {
  FILE *f;
  int offset;
} EnergyInfo;

/* callback to save energy values */
int32 save_energy(void *info, MD_Cbdata *cbd, int32 cbdlen, int32 stepnum) {
  Energy *enfo = (Energy *) info;
  double *energy = (double *) ((char *) cbd[0].data + enfo->offset);

  fprintf(enfo->f, "%d %g\n", stepnum, *energy);
  return 0;
}

/* main code section */
MD_Cbdata cbd[1];
EnergyInfo enfo;
int32 result_type = MD_type(eng, "Result");

cbd[0].idnum = MD_idnum(eng, "result");
cbd[0].access = MD_CBREAD;
cbd[0].nelems = 1;
cbd[0].first = 0;

enfo.offset = (char *) MD_type_member(eng, result_type, NULL, "energy", NULL)
    - (char *) NULL;

/*** open energy file, store handle in enfo.f ***/

MD_callback(eng, save_energy, &enfo, cbd, 1, 5);
MD_run(eng, totalsteps, MD_CBFIRST);
MD_wait(eng);

/*** close energy file ***/
```

The expectation here is that the engine provides a single-element data array `result` of unknown data type `Result`, containing a member `energy` of type `double`. The sly part is the use of `MD_type_member()` to compute an offset in bytes from the front of the `result` object to the `energy` member. Knowing this offset is necessary since there is no guarantee that the callback receives an identical data pointer as, say, a call to `MD_direct()` in the main code section. The `MD_type_member()` call sets the object instance to `NULL` to compute the address of `energy` starting from 0, then subtracts `NULL` (with both typecast to `char *`) in order to obtain the `int` difference in bytes. Although `MD_type_member()` could have been called within

`save_energy()` to obtain the pointer offset from a legitimate instance of type `Result`, that approach is significantly slower than simply adding an offset. An alternative approach to saving these results is to write the entire `Result` object to a binary file that can later be parsed. Since different engines are likely to have differences in their type definitions of `Result`, care must be taken to preserve the member names, offsets, and object byte size.

Chapter 4

Guide to Using Engine Interface

The engine is primarily responsible for computing a molecular dynamics simulation. To accomplish this, the engine must enable a front end to setup and obtain results from the simulation by providing access to data arrays. The engine interface is available by including the `mdengine.h` header file into the engine source code and linking to the MDAPI library. The details for building and linking to the current MDAPI implementation are presented in Sec. 1.3.

This chapter provides guidelines for using the engine interface. The interface is object oriented, with calls operating on a front end object. The engine has three entry points from the MDAPI layer:

1. initialization routine (called through `MD_init()`),
2. run routine (called through `MD_run()`),
3. cleanup routine (called through `MD_done()`).

Each of these three engine parts has particular engine interface routines associated with it. The sections that follow discuss each part, providing code examples that demonstrate the recommended order of engine interface calls and their use. This chapter should be read with Chap. 6 used for reference, especially Secs. 6.1 and 6.3.

4.1 Initialization

The engine initialization routine called through `MD_init()` has the following prototype.

```
int32 engine_init(MD_Front *frnt, int32 flags);
```

The name *engine_init* is used here as a placeholder; the actual name of this function is either specified by the front end (passed as a function pointer to `MD_init()`) or determined by the MDAPI implementation support for dynamic loading of an engine module. The received `frnt` argument points to the front end object. The interface places no meaning on the `flags`, so its value is entirely engine-dependent.

The initialization mostly involves setting up the engine internal data, establishing the engine data arrays, and specifying the run routine. These tasks are done once for each engine *object*, so *engine_init* can be viewed as the *constructor* for the engine. The return value from *engine_init* should be either 0 for success

or MD_FAIL for failure. Any MDAPI calls made during initialization may be considered fatal, so MD_FAIL should immediately be returned. The cleanup routine will still be called through MD_done(), so the freeing of memory allocations can be delayed until then.

4.1.1 Setting up internal data

The engine requires maintaining internal data. This is best done by allocating a main engine data object on the heap, so as to make the engine thread-safe. For the remainder of the chapter, assume that there is an Engine type defined as follows.

```
typedef struct Engine_tag {
  /* internals to be announced */
} Engine;
```

Note that Engine might itself contain other data objects in order to separate the computational aspects of molecular dynamics. The first task of *engine_init* might be to allocate this engine object.

```
Engine *e;

e = (Engine *) calloc(1, sizeof(Engine));
```

It is probably best to use calloc() since this will automatically zero the contents of the Engine object.

The handle to the Engine object must somehow be retrieved later by the engine at its other entry routines. Storing the pointer for later retrieval is accomplished by calling MD_setup_engine() as follows.

```
if (MD_setup_engine(frnt, e)) return MD_FAIL;
```

The other engine entry routines will call MD_engine_data() to obtain the Engine object pointer. It is important to check the return value of MD_setup_engine() because it also checks that the engine is using the same version of the MDAPI as the front end. For this reason, MD_setup_engine() should be the first engine interface routine called.

The alternative to dynamic memory allocation is to keep global variables. Not only is this not thread-safe (unless some sort of locking is done), but it also makes it more difficult to support simultaneous simulations. One solution is to keep a variable indicating whether the engine is in use. The code might be something like the following.

```
/******* NOT ADVISED *******/

int is_active = 0;

int32 engine_init(MD_Front *frnt, int32 flags) {
  extern is_active;

  if (is_active) return MD_FAIL;    /* engine is already in use */
  else is_active = 1;

  /*** other initialization stuff ***/
```

```
  return 0;
}

/******* NOT ADVISED *******/
```

The `is_active` variable can later be cleared at the end of the engine cleanup routine. However, since the MDAPI interface permits multiple calls to `MD_init()` to be "in flight" at the same time on different interface objects, this approach might still fail unless some mutual exclusion mechanism is used to control access to `is_active`.

4.1.2 Establishing engine data arrays

The engine data arrays provide the communication mechanism for the front end to setup the simulation and obtain results. Particular engine data arrays should be provided to ensure basic interoperability with different front ends. See Chap. 5 for a list. An engine data array has assigned to it an unique string name that the front end uses to identify the array. The elements of an engine data array must be of some defined data type. There are a number of predefined data types, presented in Sec. 2.2, and the engine can define new data types, as discussed in Sec. 4.1.4.

Each engine data array is stored using a `MD_Engdata` object. These objects are allocated by the MDAPI layer, referenced by the engine using pointers. There are three different ways to create `MD_Engdata` objects depending on the desired memory management strategy for the buffer. The choices are:

- `MD_engdata()` — the MDAPI layer manages the data buffer memory,
- `MD_engdata_buffer()` — a fixed buffer is assigned that cannot be resized,
- `MD_engdata_manage()` — the memory management method is specified.

The `MD_engdata()` routine is best used for data arrays that depend on the size of the system, as determined by the input files read by the front end. The following code shows the creation of the position array.

```
Engine *e;    /* has member:  MD_Engdata *pos */

e->pos = MD_engdata(frnt, "pos", MD_DVEC, MD_READ | MD_WRITE | MD_RESIZE);
  /* might also add MD_CBREAD and MD_FCBREAD access to support callbacks */
```

The `MD_engdata()` returns a pointer to the newly created `MD_Engdata` object or `NULL` if an error occurs. The arguments provided are the name of the engine data array, its data type, and the access permissions. The name string, for all routines in this section, must persist until the cleanup routine, making the use of a string literal a good choice. The buffer starts out having zero length, and it is left to the front end to resize the buffer.

The `MD_engdata_buffer()` routine is best used for data arrays that have a fixed size. This includes "scalar" values such as simulation parameters. The time step is such an example.

```
Engine *e;    /* has members:  MD_Engdata *timestep, double dt */
MD_Attrib timestep_attr = { MD_DOUBLE, 1, 1, MD_READ | MD_WRITE };

e->timestep = MD_engdata_buffer(frnt, "timestep", timestep_attr, &(e->dt));
```

For this routine, it is necessary to provide the full set of attributes. The order of attributes is the type, the length, the maximum allocation, and the access permissions. The arguments provided are the name of the engine data array, the data array attributes, and a pointer to the buffer, which in this case is just the address of the variable `e->dt` that will hold the time step.

Another example involving `MD_engdata_buffer()` might be if it is desirable to either pre-allocate or maintain a static array for, say, the position array.

```
Engine *e;    /* has members:  MD_Engdata *pos, MD_Dvec posbuf[10000] */
MD_Attrib pos_attr = { MD_DVEC, 0, 10000, MD_READ | MD_WRITE | MD_SETLEN };

e->pos = MD_engdata_buffer(frnt, "pos", pos_attr, e->posbuf);
```

In this case, the engine allows a system no larger than 10,000 atoms to be simulated. Note that the length attribute is set to 0 since no positions have yet been initialized, whereas the maximum allocation of 10,000 is indicated. The `MD_SETLEN` flag allows the front end to set the length of the position array up to the 10,000 atom limit. Although this particular instance is contrived, this approach could be used to wrapper a Fortran-based engine.

A more practical example using a fixed buffer space is accepting a string engine parameter that has certain predetermined values. For example, NAMD accepts a string parameter in its simulation configuration file for the `"exclude"` keyword, whose value describes the nonbonded exclusion policy, one of `"none"`, `"1-2"`, `"1-3"`, `"1-4"`, or `"scaled1-4"`. The following code fragment shows a partial definition of the `Engine` data structure and the `engine_init()` source used to handle this case.

```
typedef struct Engine_tag {
  MD_Engdata *eng_exclude;

  char exclude[12];
} Engine;

int32 engine_init(MD_Front *frnt, int32 flags) {
  Engine *e;
  MD_Attrib attr = { MD_CHAR, 0, 12, MD_READ | MD_WRITE | MD_SETLEN };

  strcpy(e->exclude, "none");
  attr.len = 5;  /* strlen("none") + nil-terminator */
  e->eng_exclude = MD_engdata_buffer(frnt, "exclude", attr, e->exclude);

  return 0;
}
```

Note that the default value of `"none"` is provided to the `"exclude"` array, with the length set to count the string nil-terminator. (The front end should comply with this policy of writing strings to the engine that are nil-terminated.) Here the value 12 was chosen for the `char` buffer length for the sake of 4-byte (32-bit) word alignment.

The `MD_engdata_manage()` routine is used in the (rare) event that the engine needs to manage its own memory. The position array is again used for the following example.

```
Engine *e;    /* has member:  MD_Engdata *pos */
MD_Attrib pos_attr = { MD_DVEC, 0, 0, MD_READ | MD_WRITE | MD_RESIZE };
```

```
e->pos = MD_engdata_manage(frnt, "pos", pos_attr, NULL, realloc);
```

The arguments are the name of the engine data array, the data array attributes, the pointer to the buffer, and the memory management routine. Note that the length and maximum allocation attributes are both set to 0, since there is no buffer space initially allocated, and the buffer pointer is set to `NULL`. The memory management routine must have the same prototype and semantics as the C library `realloc()` function (and the example simply uses `realloc` for the function pointer).

```
void *realloc(void *ptr, size_t size);
```

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes, with the contents at `ptr` moved to the new memory block up to the new size. The return value is a pointer to the new block of memory or `NULL` if an error occurs. The function must behave like `malloc()` when `ptr` is `NULL` and like `free()` when `size` is 0, but keep in mind that the return value from `realloc(ptr, 0)` is not necessarily `NULL`.

Establishing an engine data array associates a string name with the array. Alternative names can also be provided to reference a particular array. The following example demonstrates setting the additional name `"position"` for the position array.

```
MD_engdata_alias(frnt, "position", e->pos);
```

Note that the engine data array must have already been established using one of the three previous routines. Providing an alternative name might be useful in the case of, say, maintaining compatibility with NAMD simulation parameters.

4.1.3 Specifying the run routine

Engine initialization also requires specifying the run routine. Assuming that the run routine is named `engine_run()`, the following example shows that this is accomplished by passing the address of the function.

```
MD_setup_run(frnt, engine_run);
```

Guidelines for the `engine_run()` routine along with its responsibilities are presented in Sec. 4.2.

4.1.4 Defining new data types

The engine has the option of defining new data types that can be used when establishing data arrays. A defined data type is a C `struct` whose members are of previously defined types. There is a constraint that 4-byte numeric types (`int32`, `float`, `MD_Fvec`) be aligned at 4-byte boundaries and 8-byte numeric types (`double`, `MD_Dvec`) be aligned at 8-byte boundaries. Also, the entire size of the type must be divisible by 8 if it contains some 8-byte numeric type or be divisble by 4 if it contains some 4-byte numeric type.

A defined data type is described by an array of elements of type `MD_Member`, in which each element defines a member of the data type. The `MD_Member` itself contains the following members.

- `type` — the type number of the member

- `len` — the number of elements of this member
- `name` — the string name of this member

The `name` string is the name of the member. The `len` value allows the data type member to be defined as a (static) one-dimensional array. The `type` denotes the type of the member, using its *type number*, which itself was the return value from a previous call to the `MD_new_type()` routine. Note that this definition is recursive: the base cases are the *primary types* (such as `int32`, `double`, `MD_Dvec`, `MD_Name`) that are not considered to be based on previous types; the *derived types* are based on previously defined types (such as `MD_Atom`, `MD_BondPrm`). See Sec. 2.2 for more information regarding the predefined data types.

The `MD_Member` array description of a primary type is always an one-element array in which the `name` string is the empty string (`""`). For example, the primary type `double` has the following `MD_Member` array.

```
{ { MD_DOUBLE, 1, "" } }
```

A derived type may be viewed as a C `struct` in which each named member is of a previously defined type. For example, the derived type `MD_BondPrm` has a three-element `MD_Member` array,

```
{ { MD_DOUBLE, 1, "k" }, { MD_DOUBLE, 1, "r0" }, { MD_NAME, 2, "type" } }
```

that describes the following type definition.

```
typedef struct MD_BondPrm_tag {
  double k;
  double r0;
  MD_Name type[2];
} MD_BondPrm;
```

New types are defined by calling the `MD_new_type()` routine. The easiest way to present its use is through an example. Suppose that the engine wishes to support polarizable force fields. This would be most readily done by creating a new type comprised of a `MD_Atom` member along with a `MD_Dvec` member that provides the polarizability constant for the atom.

```
#define NELEMS(a)  (sizeof(a) / sizeof(a[0]))

typedef struct PolarAtom_tag {
  MD_Atom atom;
  MD_Dvec d;
} PolarAtom;

static const MD_Member PolarAtomMembers[] = {
  { MD_ATOM, 1, "atom" }, { MD_DVEC, 1, "d" }
};

/*** within initialization routine ***/
eng->polar_atom_typenum = MD_new_type(frnt, "PolarAtom", PolarAtomMembers,
    NELEMS(PolarAtomMembers), sizeof(PolarAtom));
```

The arguments are the name of the type, the `MD_Member` array, the length of the `MD_Member` array, and the size of the type in bytes. The return value is the type number (`int32`) for the newly defined type, or

MD_FAIL if the call is unsuccessful. Both the name string and the MD_Member array must persist until the cleanup routine, so making the name string a constant literal and declaring the MD_Member array as static const is recommended. The NELEMS() macro is a popular C idiom that provides the length of a statically defined array. Note that maintenance of this code requires only that the PolarAtom type definition and the PolarAtomMembers array are kept consistent. Also, since the MD_Atom type contains double members, the definition of PolarAtom satisfies the byte alignment and size constraints.

There are two types (and data arrays) that the engine is recommended to define. The Param type contains simulation parameters, such as the time step, cutoff distance, and nonbonded exclusion policy. The Result type contains computed quantities, such as potential energy, kinetic energy, and temperature. These types would each be used to define single-element data arrays named param and result, respectively. See Chap. 5 for further guidelines.

The advantage to clustering engine simulation parameters into a Param type referenced through a single-element "param" engine data array is to provide the engine with an improved means of handling front end modifications to the simulation parameters. Notification of front end modification of engine data arrays is passed in the flags to the engine run routine if those engine data arrays have been established with MD_NOTIFY the access bit. As an example, the engine might contain the following code.

```
typedef struct Param_tag {
  double timestep;
  double cutoff;
  char exclude[16];
} Param;

static const MD_Member paramMemberList[] = {
  { MD_DOUBLE, 1, "timestep" },
  { MD_DOUBLE, 1, "cutoff" },
  { MD_CHAR, 16, "exclude" }
};

static const Param defaultParam = {
  /* timestep = */  1.0,
  /* cutoff =   */  0.0,
  /* exclude =  */  "none"
};

typedef struct Engine_tag {
  MD_Engdata *eng_param;

  Param param;
} Engine;

int32 engine_init(MD_Front *frnt, int32 flags) {
  Engine *e;
  MD_Attrib param_attr = { 0, 1, 1, MD_READ | MD_WRITE | MD_NOTIFY };

  param_attr.type = MD_new_type(frnt, "Param", paramMemberList,
      NELEMS(paramMemberList), sizeof(Param));
  e->param = defaultParam;
  e->eng_param = MD_engdata_buffer(frnt, "param", param_attr, &(e->param));

  return 0;
```

```
}
```

Now when the front end writes to "param", the engine will be notified through its run routine flags, which will be discussed in detail in Sec. 4.2.2. Any changes to cutoff or exclude would likely require significant updates to engine force evaluation data structures. Note that MD_READ access is also granted to permit the front end to see the default simulation parameter settings. Notice also that the Param type number is absorbed into the engine data array, available to the engine (if needed) as:

```
e->eng_param->attrib.type
```

so it does not need to be stored separately in Engine.

The advantage to clustering engine simulation results into a Result type referenced through a single-element "result" engine data array is to provide the front end with a single addressable quantity to obtain all of the simulation reductions. The engine should probably set the "result" access permissions to (MD_READ | MD_CBREAD) to allow the front end to receive the reductions to some callback function. Note that MD_WRITE should not be set since the front end will not be determining any of these data itself, and the MD_NOTIFY is useless since the front end cannot modify the data. The general rule of thumb is to include into Result any (primarily scalar) reductions that change after each simulation time step. Quantities that are determined once at the beginning of a simulation, such as the number of degrees of freedom of the system, are *not* good candidates for Result.

4.1.5 Defining new error conditions

The engine is also permitted to define new error conditions that can be raised during the simulation. The MD_new_error() routine is demonstrated in the following example.

```
e->too_hot = MD_new_error(frnt, "system temperature too hot", 0);
e->unstable = MD_new_error(frnt, "simulation is unstable", 1);
```

The return value is the new error number (int32), or MD_FAIL if the call is unsuccessful. The arguments to MD_new_error() are the message string, whose value must persist until the cleanup routine, and an int32 value, in which zero indicates that the error is recoverable and nonzero indicates that the error is fatal.

4.2 Running the Simulation

The engine run routine is called by the MDAPI layer through its MD_run() routine and must have the following prototype.

```
int32 engine_run(MD_Front *frnt, int32 numsteps, int32 flags);
```

The name *engine_run* is used here as a placeholder; the actual name of this function is specified by the engine to the interface during the engine initialization routine, where the engine passes a function pointer to the run routine in its call to MD_setup_run(), so the run routine does not need external linkage. The arguments are frnt which points to the front end object, numsteps which indicates the number of time steps to advance the system, and flags which provides information on engine data array modification by the front end. Note also that the 16 low-order bits of flags are reserved to the engine, allowing the front end to set engine-dependent flags.

This routine is the entry point into the engine for the actual simulation computation. The `numsteps` argument should be nonnegative, where the value 0 indicates that the forces and reductions should be brought up-to-date if not already. The engine is responsible for the maintenance of its state with respect to the front end provided data, so it should initially check for modifications of the engine data arrays or for the need to initialize (or reinitialize) its internal data structures. After this it performs the integration of the system for the indicated number of steps. There are some required MDAPI calls to coordinate the engine computation with the front end expectations, however, the interface has been designed with the intention of providing maximum flexibility with minimum interference to the computational work that must be performed. The following sections provide details about the interface offered to the run routine, followed by a larger code example that brings the details together into a more complete illustration of the entire routine.

4.2.1 Accessing the engine data

One of the first things necessary is to obtain the engine internal data from the interface. Using names from the previous examples, the pointer to `Engine` is obtained as follows.

```
Engine *e = (Engine *) MD_engine_data(frnt);
```

4.2.2 Dealing with modified data arrays

The engine, before integrating the system, will likely need to perform some additional initialization based on the data provided by the front end. This initialization might include setting up appropriate data structures to efficiently handle force evaluation as well as sanity checks on the data.

Since the run routine might be repeatedly called by the front end, it is important for the engine to determine whether or not any initialization must take place. A simple check on the `flags` argument to the run routine can be used. The `flags` argument will have the `MD_UPDATE` status bit set if any engine data arrays with `MD_NOTIFY` access enabled have been modified by the front end. Modification includes changes to the data or buffer length. Any modified engine data array will have the `MD_MODIFY` status bit set on its individual access flags, regardless of whether or not the `MD_NOTIFY` flag was enabled for that array.

So which engine data arrays should be established with the `MD_NOTIFY` flag? The answer is, any whose modification might affect integrating the system before some initialization is performed, which includes almost all engine data arrays. For example, it is probably sufficient to check for consistent sizes between the `"atom"`, `"pos"`, and `"vel"` arrays. An additional test on the `"atom"` array might be needed to ensure that all mass values are positive. The value of `"cutoff"` (which might be a member of the engine-defined `"Param"` type) will affect setup of the geometric hashing done for efficient evaluation of cutoff nonbonded forces. The mixed blessing and curse of keeping simulation parameters gathered in a `"Param"` type is that, although a single test against the `MD_MODIFY` status flag will reveal changes to the set, it will not be obvious which individual members have been changed. One guideline might be to maintain simulation parameters that affect the force evaluation (such as `"cutoff"` or `"exclude"`) within the `"Param"` type; the initialization of the force evaluation data structures tends to be costly. Simulation parameters that do not affect force evaluation (such as `"timestep"`) might be best kept outside of `"Param"`.

Whenever the engine has performed initialization (or reinitialization) regarding a modified engine data array that was established with the `MD_NOTIFY` flag, the engine should then call `MD_engdata_ackmod()` on the array to acknowledge that it dealt with its modification. This will clear the `MD_MODIFY` status flag on that array. Note that the only way for the engine to clear the `MD_UPDATE` global status flag is to acknowledge data array modifications for any data array established with the `MD_NOTIFY` flag that has its `MD_MODIFY` status set.

The following code demonstrates how the engine may deal with modified data arrays.

```
int32 engine_run(MD_Front *frnt, int32 numsteps, int32 flags) {
  Engine *e = MD_engine_data(frnt);
  Param *param = &(e->param);  /* point directly to param data */

  if (flags & MD_UPDATE) {
    /* must check flags of individual data arrays */
    if (e->pos->attrib.access & MD_MODIFY) {
      /* position array was modified - deal with it */
      /* access position array through (MD_Dvec *)(e->pos->buf) */
      MD_engdata_ackmod(frnt, e->pos);
    }
    if (e->eng_param->attrib.access & MD_MODIFY) {
      /* param was modified - deal with it */
      /* access param through "param" pointer, e.g. param->cutoff */
      MD_engdata_ackmod(frnt, e->eng_param);
    }
    /* etc. */
  }

  /* perform integration for "numsteps" */

  return 0;
}
```

Since the update procedure might be somewhat long and involved, depending on the complexity of the engine, it might be best performed in a separate routine.

4.2.3 Incrementing the step number counter

The interface layer keeps track of integration the step numbering. The front end is allowed to initialize the step number counter. The engine does not need to be concerned with this labeling of step numbers, rather its job is to integrate the system for `numsteps`, the number of steps specified to the run routine. However, the engine is still required to increment the step number counter after each integration step by performing:

```
MD_incrstep(frnt);
```

If the engine wishes to also know the value of the step number counter, this value is returned by calling:

```
int32 step = MD_stepnum(frnt);
```

4.2.4 Reporting and handling errors

In order for an engine to be robust, it must properly identify and handle error conditions. Almost all of the MDAPI engine routines discussed so far will return an error indicator if the call does not succeed. Generally speaking, the routines whose return value is a pointer will indicate failure with `NULL`, and the routines whose return value is an integer will indicate failure with `MD_FAIL` (-1). The return values from MDAPI routines should be monitored and handled correctly.

As stated previously, errors that occur during `engine_init()` should be treated as fatal, so `MD_FAIL` may be immediately returned. Although some exceptional conditions might be recoverable by the engine during

engine_run(), many errors cannot be handled by the engine itself and instead should be reported back to the MDAPI layer. The MD_error() routine is used to report errors, where a particular error is identified by its number. The error number is stored by the MDAPI layer and made available to the front end for attempted error recovery, much like the system library maintains errno. There are several error numbers defined by the MDAPI itself, for instance MD_ERR_MEMALLOC to denote that memory allocation failed. The engine can also define its own error number values by calling MD_new_error() as discussed previously. The MD_error() routine accepts an error number and sets the error number maintained by the MDAPI layer to this value. Also, the return value from MD_error() is MD_FAIL, so its call can be used as the unsuccessful return from an engine routine. The following partial source example illustrates.

```
int32 engine_init(MD_Front *frnt, int32 flags) {
  MD_Attrib timestep_attr = { MD_DOUBLE, 1, 1, MD_READ | MD_WRITE };
  int32 accrwz = MD_READ | MD_WRITE | MD_RESIZE | MD_NOTIFY;

  Engine *e = (Engine *) calloc(1, sizeof(Engine));
  if (e == NULL) return MD_error(frnt, MD_ERR_MEMALLOC);
      /* system library call failed, set error number */

  if (MD_setup_engine(frnt, e)) return MD_FAIL;
      /* error number set by failed MD_setup_engine() routine */

  e->atom = MD_engdata(frnt, "atom", MD_ATOM, accrwz);
  if (e->atom == NULL) return MD_FAIL;
      /* error number set by failed MD_engdata() routine */

  e->timestep = MD_engdata_buffer(frnt, "timestep", timestep_attr, &(e->dt));
  if (e->timestep == NULL) return MD_FAIL;
      /* error number set by failed MD_engdata_buffer() routine */
  e->dt = 1.0;  /* set default timestep value */

  e->unstable = MD_new_error(frnt, "simulation is unstable", 1);
  if (e->unstable == MD_FAIL) return MD_FAIL;
      /* error number set by failed MD_new_error() routine */

  e->badparam = MD_new_error(frnt, "invalid parameter value", 0);
  if (e->badparam == MD_FAIL) return MD_FAIL;
      /* error number set by failed MD_new_error() routine */

  /* etc. */

  return 0;
}

int32 engine_run(MD_Front *frnt, int32 numsteps, int32 flags) {
  Engine *e = MD_engine_data(frnt);
  MD_Atom *atom = (MD_Atom *)(e->atom->buf);
  int32 natoms = e->atom->attrib.len;

  if (flags & MD_UPDATE) {
    if (natoms == 0) return MD_error(frnt, e->badparam);
    if (e->dt <= 0) return MD_error(frnt, e->badparam);
        /* tell front end that some param is invalid */
```

```
    /* etc. */

  }

  /* perform simulation */

  if (simulation_is_unstable) return MD_error(frnt, e->unstable);

  /* etc. */

  return 0;
}
```

If an error is returned, the front end is able to obtain the error number and message description string. The fact that `e->unstable` error above was defined with the `isfatal` argument set to 1 makes this error fatal; if this error occurs, the front end has no alternative but to destroy the engine object. The `MD_ERR_MEMALLOC` is another example of an unrecoverable error. The `e->badparam` error was defined with the `isfatal` argument set to 0, meaning that the front end can possibly recover from this error (after supplying valid engine data values, which might depend on some interaction with the user to supply appropriate data files or simulation parameter values).

4.2.5 Processing callbacks

Callbacks come in three flavors: standard callbacks, force callbacks, and message callbacks. Standard callbacks are the most common type of callback, allowing the front end to access certain data arrays at regular intervals in between time steps. Force callbacks are processed every time that the force is evaluated, intending for the front end to supply additional force values to the engine. Message callbacks permit the engine to send an arbitrarily long text string message to the front end, in order to provide status information.

The front end is allowed to register an arbitrary number of callbacks of each type. In order to ease the difficulty of processing the callbacks, the engine needs only to take the viewpoint of there being three types of callbacks. Whenever a type of callback is proceesed by the engine, all of the registered callbacks of that type due to be processed at that time are executed in sequence. There is nonblocking semantics defined for each type of callback processing, so the engine can potentially overlap callback execution with other useful computation. Furthermore, the execution of the three types of callbacks can be overlapped.

Message callbacks are the simplest variety, simply sending a text string from the engine to the front end at the convenience of the engine. This means that the engine does not need to execute message callbacks unless it really wants to provide information to the front end. Standard callbacks and force callbacks are more involved because they involve some sort of access of engine data arrays. The engine designates via access flags which arrays are eligible for a particular kind of callback as well as the type of access permitted on the array during the callback. This means that an engine will not need support for, say, force callbacks if none of the engine data arrays permit the front end with this kind of access. However, if some type of callback access has been permitted on a particular array, then the engine is obligated to execute that type of callback when expected.

There are three types of engine data array access permitted for both standard and force callbacks (i.e. six distinct access flags). Read access allows the array to be received by the front end, but not modified. Write access allows the array to be written to by the front end, but the current values in the array might not be available. Read and write access can be requested together by the front end, allowing the array values to be received and modified. Share access permits the front end to provide its own array buffer of values

that will be copied into the engine array buffer. Although the copying of data from the front end provided buffer occurs transparently to the engine, if share access is permitted with resize access, the front end is actually able to change the length (and maximum capacity) of the array. In the event that the length of the array changes, the `MD_MODIFY` status bit is set on the array which should then be acknowledged by the engine through `MD_engdata_ackmod()` after the callback processing has finished. The main purpose for share access is to allow the front end to send to the engine during the simulation an arbitrarily large array of data whose length cannot be determined a priori. The engine can enable any of these types of access on a given array, however, the front end may only access a particular array as one of either read, write, read–write, or share. Note that the front end may request an array be provided multiple times to a particular callback; for example, this would allow read access to the array through one `MD_Cbdata` element and share access to the same array through another `MD_Cbdata` element. A complete discussion of access flags is presented in Sec. 6.1.2, and the details of the `MD_Cbdata` data structure are provided in Sec. 6.1.1.

There are four routines associated with the processing of the separate callback types. The "ready" routine returns immediately telling if any callbacks of that type are ready to be processed. For message and force callbacks, these routines simply tell whether any callbacks of that given type have been registered, since they will then always be ready for processing. This means that `MD_ready_msgcallback()` and `MD_ready_fcallback()` need only be called once at the beginning of the run routine. The processing of a particular standard callback depends on the step number, so `MD_ready_callback()` must be called after the completion of each time step to see if any standard callbacks need processing. The actual processing of some type of callback is performed by calling the appropriate "exec" routine. The syntax of these calls is as follows.

```
MD_exec_callback(frnt);
MD_exec_fcallback(frnt, timestepfrac);
MD_exec_msgcallback(frnt, "engine message to front end");
```

No additional data is needed to process standard callbacks. Force callbacks need to know the `timestepfrac` which, when added to the current step number and the quantity scaled by the time step, will give the time value for the positions at that force evaluation. As an example, leapfrog (velocity–Verlet) integration requires only one force evaluation for each step in which the positions have already been advanced by one time step, meaning that `timestepfrac` = 1. Message callbacks need the text string as an additional argument. The "exec" routines each have nonblocking semantics, so each of the callback types has its own routines "test" to see if the processing has finished and "wait" to block for completion. The engine is permitted to overlap processing of different types of callbacks, but it must not initiate another callback processing of the same type before a previous callback processing of that type has completed. Also, the engine should not update a data array while it might be involved in a callback, nor should it make use of a data array while it might be updated by a callback.

The final concept to keep in mind is that standard callbacks might require processing before taking any time steps in order to, say, provide energy reductions at step 0. This would, of course, require an initial force evaluation at time 0 which might entail processing force callbacks. Assume that the `Engine` data structure contains at least the following `MD_Engdata` arrays that have been established using the indicated access permissions.

```
typedef struct Engine_tag {
  MD_Engdata *pos;       /* MD_READ | MD_WRITE | MD_CBREAD | MD_FCBREAD */
  MD_Engdata *vel;       /* MD_READ | MD_WRITE | MD_CBREAD */
  MD_Engdata *extforce;  /* MD_WRITE | MD_FCBWRITE */
  MD_Engdata *result;    /* MD_READ | MD_CBREAD */
  MD_Engdata *atom;      /* MD_WRITE */
} Engine;
```

The access permission flags listed for the `MD_Engdata` arrays would be typical in practice. The main data

of interest to standard callbacks would be the position and velocity for saving trajectory and restart files along with the reduction results for monitoring energy and temperature during the simulation. A single array might be devoted entirely to external forces supplied by the front end, the computation of which would depend on receiving the positions.

The following code example based on the `Engine` data outlines a simple run routine implementation in which there is no overlap of callback processing with computation and the computational work shown using comments. Nevertheless, the form of the run routine is evident, including dealing with data array modifications, incrementing the step number, and processing callbacks.

```
int32 engine_run(MD_Front *frnt, int32 numsteps, int32 flags) {
  Engine *e = MD_engine_data(frnt);
  int32 isfcb = MD_ready_fcallback(frnt);
  int32 natoms = e->atom->attrib.len;

  /* clear memory for external force array */
  memset(e->extforce->buf, 0, natoms * sizeof(MD_Dvec));

  if (flags & MD_UPDATE) {
    update(frnt);
    /* must also evaluate forces */
    /* force:      f = force(pos)              */
    if (isfcb && (MD_exec_fcallback(frnt, 0) || MD_wait_fcallback(frnt))) {
      return MD_FAIL;
    }
    /* add in external forces:  f += extforce  */
  }

  if (MD_ready_callback(frnt)) {
    /* update result reductions before callback */
    if (MD_exec_callback(frnt) || MD_wait_fcallback(frnt)) {
      return MD_FAIL;
    }
  }

  for (step = 0;  step < numsteps;  step++) {
    /* half-kick:  vel += dt/2 * force / mass  */
    /* drift:      pos += dt * vel             */
    /* force:      f = force(pos)              */
    if (isfcb && (MD_exec_fcallback(frnt, 0) || MD_wait_fcallback(frnt))) {
      return MD_FAIL;
    }
    /* add in external forces:  f += extforce  */
    /* half-kick:  vel += dt/2 * force / mass  */

    MD_incrstep(frnt);

    if (MD_ready_callback(frnt)) {
      /* update result reductions before callback */
      if (MD_exec_callback(frnt) || MD_wait_fcallback(frnt)) {
        return MD_FAIL;
      }
    }
```

```
    }
    /* update result reductions before returning */
    return 0;
  }
```

The processing of force callbacks can be overlapped with the force computation, which turns out to require the majority of computational effort. However, force callback processing cannot be performed until after the next positions are available and must complete with the contribution totaled before the final half-kick of the integration. For standard callbacks, if additional buffer space is used and if the position array with `MD_CBREAD` access is separated from a different position array with `MD_FCBREAD` access, then the overlap could be from the end of one step until right before the next step number incrementation.

4.2.6 Examining callback data requirements

The complexity of the engine might necessitate specific knowledge of the data requirements for callbacks. For example, a parallel engine would need to perform a gather operation to ensure update position and velocity arrays for a callback that needs them, but would not want to waste time unless these quantities are needed.

The `MD_callback_list()` function returns the `const` array of type `MD_Callback` that contains all of the registered standard callbacks. The members of interest in each `MD_Callback` structure element are `MD_Cbdata *cbarg` that gives the array of `MD_Cbdata` that is received by that callback, `cbarglen` that gives the length of the `cbarg` array, `stepincr` that indicates in number of steps how often that callback will need processing, and `nextstepnum` that tells the number of the next step when that callback will need processing. (Recall that the current step number is available through `MD_stepnum()`.)

Examination of the `MD_Cbdata` array for each callback will tells exactly its engine data array requirements. Each `MD_Cbdata` structure element has a member `engdata` that points to a particular engine data array, as well as members `access` which indicates the requested access (for a standard callback, some combination of `MD_CBREAD`, `MD_CBWRITE`, and `MD_CBSHARE`) and members `nelems` and `first` that will tell the extent of the sub-array needed. See Sec. 6.1.1 for further details regarding the `MD_Cbdata` data structure.

The `MD_fcallback_list()` function returns the `const` array of type `MD_Callback` that contains all of the registered force callbacks. This reveals exactly the same information as discussed for standard callbacks, except that the `stepincr` and `nextstepnum` fields are meaningless since all force callbacks need to be processed with every force evaluation.

4.2.7 Resizing data arrays

The engine is able to resize data arrays if they have been established with some combination of `MD_ESETLEN` and `MD_ESETMAX` (or `MD_ERESIZE`) access. One example might be to offer the front end a force array with `(MD_READ | MD_ERESIZE)` permission. The engine would then, during `update()`, resize the array to match the number of atoms in the system. The routines that perform the resizing are:

```
MD_engdata_setlen(frnt, e->force, newlen);
MD_engdata_setmax(frnt, e->force, newmax);
MD_engdata_resize(frnt, e->force, newlen, newmax);
```

They work almost exactly like their front end counterparts, except that the second argument takes a pointer to `MD_Engdata` rather than a data identification number. There are probably other examples in which it is easier to have the MDAPI layer manage array memory rather than the engine.

4.3 Cleanup

The engine cleanup routine called through `MD_done()` has the following prototype.

```
void engine_done(MD_Front *frnt);
```

The name *engine_done* is used here as a placeholder; the actual name of this function is either specified by the front end (passed as a function pointer to `MD_init()`) or determined by the MDAPI implementation support for dynamic loading of an engine module. The received `frnt` argument points to the front end object.

The main responsibility is for the engine to free memory that it has allocated. The only exception is with any `MD_Engdata` arrays established using `MD_engdata_manage()`, which might have been initially allocated by the engine but will be freed by the MDAPI. These tasks are done once at the end of each engine *object*, so *engine_done* can be viewed as the *destructor* for the engine.

The MDAPI layer `MD_done()` routine initially calls *engine_done* before freeing its own memory allocations. In situations involving engine-managed buffer space established with `MD_engdata_manage()`, it is possible that the array buffer space will need to be freed before, rather than after, the engine. The engine is able to force the MDAPI layer to free all `MD_Engdata` buffer allocations immediately by calling:

```
MD_free_data(frnt);
```

Chapter 5

Assumptions for Interoperability

In order to maintain a basic level of interoperability between any given front end and engine, expected units and standard names for particular data arrays must be defined. The following table, repeated in a slightly abbreviated form from Chap. 2, gives the standard names for data arrays along with type and intended purpose, and has been extended to include expected access permissions as well as units for position, velocity, and force. (Note that the units for force field parameter data types are already well-defined.) This table offers the minimum requirements for engines to be compatible with front ends. It is likely that these arrays will also permit `MD_RESIZE` and, for performance, enable `MD_CBREAD` on position and velocity arrays.

array name	*data type*	*access*	*purpose and properties*
`"atom"`	`MD_Atom`	`MD_WRITE`	parameters required for each atom, defines ordering of atoms in system
`"pos"`	`MD_Dvec` or `MD_Fvec`	`MD_READ | MD_WRITE`	position (in Å) for each atom, uses same ordering as `"atom"`
`"vel"`	`MD_Dvec` or `MD_Fvec`	`MD_READ | MD_WRITE`	velocity (in Å/fs) for each atom, uses same ordering as `"atom"`
`"force"`	`MD_Dvec` or `MD_Fvec`	`MD_READ`	force (in kcal/mol/Å) for each atom, uses same ordering as `"atom"`
`"bond"`	`MD_Bond`	`MD_WRITE`	defines covalent bonds in system
`"angle"`	`MD_Angle`	`MD_WRITE`	defines angle bonds in system
`"dihed"`	`MD_Tors`	`MD_WRITE`	defines dihedral torsion angles in system
`"impr"`	`MD_Tors`	`MD_WRITE`	defines improper torsion angles in system
`"excl"`	`MD_Excl`	`MD_WRITE`	pairs excluded from nonbonded interactions
`"atomprm"`	`MD_AtomPrm`	`MD_WRITE`	nonbonded force field parameters for each atom type
`"bondprm"`	`MD_BondPrm`	`MD_WRITE`	covalent bond force field parameters based on atom types involved in bond
`"angleprm"`	`MD_AnglePrm`	`MD_WRITE`	angle bond force field parameters based on atom types involved in angle
`"dihedprm"`	`MD_TorsPrm`	`MD_WRITE`	dihedral torsion angle parameters based on atom types involved in dihedral
`"imprprm"`	`MD_TorsPrm`	`MD_WRITE`	improper torsion angle parameters based on atom types involved in improper
`"nbfixprm"`	`MD_NbfixPrm`	`MD_WRITE`	pairwise corrections to nonbonded force field parameters

The engine may wish to define a single-element array `"param"` of engine-defined type `"Param"` containing simulation parameters. Simulation parameters should include at least the following.

array name	*data type*	*purpose and properties*
`"timestep"`	`double` or `float`	time step (in fs) for integration
`"cutoff"`	`double` or `float`	nonbonded interaction cutoff (in Å)
`"switchdist"`	`double` or `float`	van der Waals switching distance (in Å)

They should either be members of the `"Param"` type or fixed-size single element arrays. Generally, the front end should consider any engine data array a simulation parameter if it has a single-element fixed size, is of a primary type, and permits write access. The exception is if an engine data array of `char` has multiple elements or is resizeable; in this case it should be treated as a string (and any value set by the front end should be nil-terminated). It is also advantageous for a simulation parameter to permit read access to allow the front end to see default parameter values. Note that predefined types `MD_String` and `MD_Message` should be considered deprecated.

The engine may also wish to define a single-element array `"result"` of engine-defined type `"Result"` containing simulation reduction results. Reductions should include at least the following.

array name	*data type*	*purpose and properties*
`"energy"`	`double` or `float`	total energy (in kcal/mol) of system
`"temp"`	`double` or `float`	temperature (in °K) of system

They should either be members of the `"Result"` type or fixed-size single element arrays. Generally, the front end should consider any engine data array a simulation reduction result if it has a single-element fixed size, is of a primary type, and permits read without write access. For performance reasons, it is recommended that the engine does define `"result"` with `MD_CBREAD` access to allow the front end to monitor it through a callback. The `"Result"` members should only contain time-dependent reductions. For example, the system energy, temperature, and pressure are all time-dependent quantities. The number of degrees of freedom is not a time-dependent quantity, so should not be a `"Result"` member.

Chapter 6

Complete Reference

This chapter provides a complete reference for the MDAPI, including all user-level constants, type definitions, and function prototypes that comprise the interface specification. The semantics of each routine is presented along with the errors that might possibly occur. The predefined data types and related constants from `mdtypes.h` are not presented here, since the interface is kept independent from the data communicated between front end and engine; see Chap. 2 for details about the representation of the molecular system.

The chapter is organized around the names of the header files. There should never be a need to explicitly include the `mdcommon.h` header file, since it is included by the other header files. Instead, codes using the MDAPI should include either `mdfront.h` to make use of the front end interface or `mdengine.h` to make use of the engine interface.

6.1 Definitions Common to Front End and Engine

The definitions common to the front end and engine are found in the `mdcommon.h` header file. This file is included by both `mdfront.h` and `mdengine.h`.

6.1.1 Objects

Type: `MD_Interface`

Summary: The interface object.

Definition:
```
typedef struct MD_Interface_tag {
    /* contents opaque to front end and engine */
} MD_Interface;
```

Description: This is the primary storage container for the MDAPI layer. The front end creates an instance of the object (either on the stack or allocated on the heap as an `MD_Engine` object, see Sec. 6.2 for details). The engine receives a pointer to the object as a parameter to its entry routines (as an `MD_Front` object, see Sec. 6.3 for details).

The `MD_Interface` object defines the state of the MDAPI layer as the interface between a particular front end and engine. A pointer to an instance of the object is the first argument to all of the MDAPI routines for both front end and engine. The internal members of `MD_Interface` are implementation dependent, so should never be accessed directly.

Type: MD_Attrib

Summary: The engine data attribute object.

Definition:
```
typedef struct MD_Attrib_tag {
    int32 type;
    int32 len;
    int32 max;
    int32 access;
} MD_Attrib;
```

Description: Every engine data array has an associated attribute object. The `type` member indicates the data type of the array elements. The `len` member gives the used storage length of the array, in number of elements. The `max` member gives the maximum allocation of the array, also in number of elements. The `access` member is a bit-field indicating the access permissions granted to the front end by the engine along with some internal status flags.

Type: MD_Cbdata

Summary: The callback data object specifies an engine data sub-array to be received by a callback routine.

Definition:
```
typedef struct MD_Cbdata_tag {
/* set by front end for initialization */
    int32 idnum;
    int32 access;
    int32 nelems;
    int32 first;

/* provided to engine */
    MD_Engdata *engdata;

/* provided to callback routine */
    void *data;
    MD_Attrib attrib;
} MD_Cbdata;
```

Description: The front end can establish callbacks to itself during a running simulation. There are three types of callbacks: standard, force, and message, all explained in Sec. 6.2. The standard and force callback routines receive an array of `MD_Cbdata` through which engine data sub-arrays are passed. Setting up a callback also requires an array of `MD_Cbdata` to specify the engine data sub-arrays needed by the callback routine.

Each `MD_Cbdata` object in the array specifies a particular engine data sub-array. The front end sets the fields `idnum`, `access`, `nelems`, and `first` for initialization. The `idnum` member is set to the engine data array identification number. The `access` member indicates the requested access, the choices being some bitwise ORing of access flags `MD_CBREAD`, `MD_CBWRITE`, and `MD_CBSHARE` for a standard callback, or `MD_FCBREAD`, `MD_FCBWRITE`, and `MD_FCBSHARE` for a force callback. Note that the semantics of `*SHARE` access is different so cannot be combined with either `*READ` or `*WRITE`. The `nelems` member indicates the number of elements desired from the sub-array, where -1 is used to indicate "all elements." The `first` member indicates the index of the first element from the engine data array to be included in the sub-array.

The `engdata` member is provided to the engine as a handle to the particular engine data array that is being accessed through the callback. (Note that the front end refers to the engine

data arrays using identification numbers, whereas the engine is provided with a pointer to an MD_Engdata object.) Use of this is important to the engine when determining which portions of arrays are needed for callbacks.

The last two members are provided to the callback routine. The attrib member contains the attributes for the engine data array. In the case of either or both of *READ or *WRITE sub-array access, the data member points to that sub-array of the indicated engine data array indexed starting with first and containing nelems number of elements, or extending through the last element of the array if nelems = −1. Having *READ access provides the callback with values to use, *WRITE access permits the callback to set values, and both *READ and *WRITE access provides the callback with sub-array values to use and modify. If the front end and engine share the same memory space, then data might point into the actual array. Otherwise, the sub-array space is buffered in the front end memory space.

The *SHARE access changes the semantics. In this case, the callback receives data = NULL and nelems = first = 0, and it is up to the callback to provide the sub-array buffer to the engine. The sub-array semantics still apply, with values to nelems and first describing what portion of the engine data array is being provided by the callback sub-array buffer and possibly resizing the data array, so these values need to be set carefully by the callback. The front end retains ownership of the memory it sends through the data pointer, with copy semantics like MD_writesub(), which means that the same memory buffer can be reused on subsequent callbacks, and the front end is responsible for freeing the memory. This mechanism is provided to allow the front end to send arbitrary length arrays to the engine during a simulation.

The *SHARE access cannot be combined with the other access types. For instance, the front end cannot set callback access to MD_CBREAD and MD_CBSHARE for the given MD_Cbdata element, even through the engine could have enabled both MD_CBREAD and MD_CBSHARE access on the same data array. It is possible for the front end to establish different access to the same data array using two different MD_Cbdata elements in the array of MD_Cbdata passed to the MD_callback() or MD_fcallback() routines. So, if the callback wished to see the current data array contents before updating its shared buffer, it could set one MD_Cbdata element to MD_CBREAD access and a second to MD_CBSHARE access (assuming that the engine had enabled both kinds of access on the data array).

Data array resize changes that might occur with *SHARE access are permitted only if the front end is granted access to do so. Otherwise, MD_ERR_CBSHARE will be set by the callback processing. If the data array is resized, the MD_MODIFY flag is set on the engine data array.

Examples: Use the declaration: MD_Cbdata c;

If the data array allows MD_CBREAD and MD_CBWRITE access, then it is permissible to set c.access = MD_CBREAD or c.access = MD_CBWRITE or c.access = MD_CBREAD | MD_CBWRITE.

Suppose that the engine data array has len = 10. Then setting c.first = 3 and c.nelems = 5 means that, within the callback, data[0..4] will correspond to engine array data[3..7].

Suppose that the callback needs to receive the entire engine data array, but the length is unknown (e.g. the engine is allowed to resize it). Then set nelems = −1 and first = 0. If all but the first element is desired, then set nelems = −1 and first = 1, etc.

Type: MD_Member

Summary: The member object describes a member of a derived data type.

Definition:
```
typedef struct MD_Member_tag {
    int32 type;
    int32 len;
```

```
    const char *name;
} MD_Member;
```

Description: Derived data types are viewed as structures of primary and previously defined derived types. An array of `MD_Member` describes the members of the derived type, where `type` indicates the data type number of the member, `len` indicates the number of elements of this member to allow for a statically defined array, and `name` is the nil-terminated string name of the member.

For primary types, `MD_Member` is given with `len` = 1 and `name` = `""` (the empty string). Note that `name` is *not* the name of the type, rather the name of the member having this type. Also, recall that `MD_Fvec` and `MD_Dvec` are considered primary types.

Examples: `int32` has a 1-element `MD_Member` array:

```
{{MD_INT32, 1, ""}}
```

`MD_Dvec` has a 1-element `MD_Member` array, since it is a primary type:

```
{{MD_DVEC, 1, ""}}
```

`MD_BondPrm` has a 3-element `MD_Member` array:

```
{{MD_DOUBLE, 1, "k"}, {MD_DOUBLE, 1, "r0"}, {MD_NAME, 2, "type"}}
```

6.1.2 Access flags

Engine data arrays each have an access flag attribute, the `access` member of `MD_Attrib`, stored as an `int32` bit-field to grant the front end particular access rights for that array. The access flags which may be set by the engine are listed by name in the following table along with a brief description.

access flag	description
MD_READ	read access
MD_WRITE	write access
MD_CBREAD	read during callback
MD_CBWRITE	write during callback
MD_CBSHARE	shared data from front end during callback
MD_FCBREAD	read during force callback
MD_FCBWRITE	write during force callback
MD_FCBSHARE	shared data from front end during force cb
MD_SETLEN	front end allowed to set length
MD_SETMAX	front end allowed to set maximum allocation
MD_RESIZE	MD_SETLEN | MD_SETMAX
MD_ESETLEN	engine allowed to set length
MD_ESETMAX	engine allowed to set maximum allocation
MD_ERESIZE	MD_ESETLEN | MD_ESETMAX
MD_DIRECT	direct access
MD_NOSHARE	do not permit front end shared buffer
MD_NOTIFY	notify engine of front end updates

The engine sets these access permissions when it establishes the engine data array, by calling `MD_engdata()` or one of the alternative routines. These particular access permissions remain fixed for the array.

The MDAPI layer may set additional access flags that act more like status flags since these particular access permissions might change.

access flag	description
MD_SHARE	front end allowed to share its buffer
MD_UNSHARE	front end allowed to take back its buffer
MD_MODIFY	front end has modified buffer ("dirty bit")

The MD_SHARE and MD_UNSHARE flags permit use of front end shared buffers through the MD_share() and MD_unshare() routines (defined in Sec. 6.2). The MD_MODIFY flag marks the array as having been modified by the front end, to assist the engine with the initialization of the simulation. This "dirty bit" is toggled off by the engine call to MD_engdata_ackmod() (defined in Sec. 6.3).

6.1.3 Global status flags

There are some internal flags denoting the status of the MD_Interface object. The two flags intended for external use are shown in the following table.

status flag	description
MD_CBFIRST	front end sets this to run flag to invoke callbacks before first step
MD_UPDATE	engine tests run flags to see if data array with MD_NOTIFY flag was modified

These flags are used in conjunction with running the simulation. The front end invokes MD_run() with a runflags argument (see Sec. 6.2). Logical ORing MD_CBFIRST into the run flags will tell the MDAPI layer that all callbacks are to be processed before the first time step (i.e. on the zeroth step). Otherwise, each callback is scheduled to be processed after stepincr steps have been taken (see the definitions of MD_callback() in Sec. 6.2 and MD_Callback in Sec. 6.3). The engine receives the runflags to its run routine entry point and can test against the MD_UPDATE flag being set to see if there is some data array having MD_NOTIFY access that has been modified by the front end (i.e. the MD_MODIFY status flag is set on the array). The engine should deal with these reported data array modifications and should then invoke MD_engdata_ackmod() to acknowledge the array modification, which removes the MD_MODIFY status (see Sec. 6.3 for details).

6.1.4 Error constants

MDAPI routines that return int32 typically denote an error condition with value MD_FAIL (-1). An error number, internal to the MD_Interface object, is set to report the specific error (similar to errno used by the standard C library). The possible error constants and corresponding diagnostic messages are listed in the following table.

error constant	diagnostic message
MD_ERR_NONE	"Success"
MD_ERR_VERSION	"Inconsistent version number"
MD_ERR_MEMALLOC	"Memory cannot be allocated"
MD_ERR_INIT	"Engine initialization failed"
MD_ERR_RUN	"Engine run simulation failed"
MD_ERR_CHECK	"Consistency check failed"
MD_ERR_NEWDATA	"Cannot create new engine data"
MD_ERR_NEWTYPE	"Cannot create new type"
MD_ERR_NAME	"Unrecognized name string"
MD_ERR_IDNUM	"Invalid data ID number"
MD_ERR_TYPENUM	"Invalid type number"
MD_ERR_RANGE	"Value is out of range"
MD_ERR_ACCESS	"Access is denied"
MD_ERR_CALLBACK	"Callback routine failed"
MD_ERR_CBSHARE	"Callback shared buffer failed"

6.2 Front End Interface Specification

The routines presented here provide the front end interface to the MDAPI. The definitions and prototypes are all found in the `mdfront.h` header file. The front end declares a variable of type `MD_Engine` (or allocates an instance on the heap), then uses a pointer to this `MD_Engine` object in all of the routines.

6.2.1 Objects

Type: `MD_Engine`

Summary: The engine object.

Definition: `typedef MD_Interface MD_Engine;`

Description: This is just the `MD_Interface` from Sec. 6.1 renamed to appear as an engine object. The front end needs to allocate space for this, either on the stack or the heap.

All of the front end API calls operate on the engine object, requiring a pointer to the object passed as the first argument, analogous to the C++ `this` pointer. The first call must be to the `MD_init()` constructor to initialize the engine. Use of the object is finished by calling the `MD_done()` destructor to cleanup the engine and deallocate memory. The MDAPI layer is itself reentrant, so the front end can manipulate multiple engine objects simultaneously, and these can all invoke the same engine as long as the engine itself is also reentrant.

6.2.2 Initialization and cleanup

Function: `MD_init()`

Summary: Initialize the MDAPI layer and the specified engine.

Prototype:
```
int32 MD_init(MD_Engine *e, const char *engname, int32 flags,
    int32 (*engine_init)(MD_Engine *, int32 flags),
    void (*engine_done)(MD_Engine *));
```

Arguments: `e` — points to a valid, uninitialized `MD_Engine` object

`engname` — name of engine, nil-terminated string

`flags` — implementation dependent, passed on to `engine_init()`

`engine_init` — constructor for engine

`engine_done` — destructor for engine

Description: For the case in which one or more engines are linked to the front end, the engine constructor and destructor routines need to be passed. Here, the `engname` string is remembered, but useful only for labeling a particular engine.

For the case in which an engine is dynamically loaded or remotely invoked, the `engine_init` and `engine_done` pointers should be `NULL`, and behavior is determined by the `engname` string (and perhaps also by the flags). Suggested string interpretation is

[[user@]hostname:]pathname

which covers dyanmically loadable engines on the local machine as well as remote engines.

No `flags` are (currently) defined by the API, but this value is passed as an argument to `engine_init()` to permit implementation dependent flags.

Call has nonblocking semantics (see below).

Return value: Returns `0` on success or `MD_FAIL` on error.

Errors: `MD_ERR_VERSION` — inconsistent version number

`MD_ERR_MEMALLOC` — memory cannot be allocated

`MD_ERR_NEWDATA` — (from `engine_init()`) cannot create new engine data

`MD_ERR_NEWTYPE` — (from `engine_init()`) cannot create new type

`MD_ERR_INIT` — default fatal error indicating that the engine initialization failed

Note that any error state returned here is fatal.

Function: `MD_done()`

Summary: Cleanup engine and MDAPI layer.

Prototype: `void MD_done(MD_Engine *);`

Description: Calls `engine_done()` routine (see `MD_init()`), frees engine data array buffers (where necessary), and frees memory allocations made within MDAPI layer.

6.2.3 Obtaining data array names and ID numbers

Function: `MD_idnum()`

Summary: Obtain the ID number for an engine data array.

Prototype: `int32 MD_idnum(MD_Engine *, const char *name);`

Arguments: `name` — data array name identifier, nil-terminated string

Description: Given the nil-terminated string name of a data array, return its identification number. Case sensitive string-matching is performed.

Return value: Returns the nonzero identification number corresponding to the recognized data array name, or `MD_FAIL` if no match is found.

Errors: `MD_ERR_NAME` — name does not match any engine data arrays

Function: `MD_name()`

Summary: Obtain the name for an engine data array.

Prototype: `const char *MD_name(MD_Engine *, int32 idnum);`

Arguments: `idnum` — data array identification number

Description: Given the identification number of a data array, return its (primary) name. This routine is the inverse of `MD_idnum()`.

Return value: Returns the nil-terminated string name of the data array or `NULL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

Function: `MD_namelist()`

Summary: Obtain the list of data array names.

Prototype: `const char **MD_namelist(MD_Engine *, int32 *listlen);`

Arguments: `listlen` — points to an integer variable

Description: The list of names of all engine data arrays is returned, with the length of the list returned through the `listlen` pointer.

Return value: Returns a pointer to an array of nil-terminated strings and modifies the value of the `int32` variable pointed to by `listlen`.

Errors: None (always succeeds).

6.2.4 Obtaining attributes of data arrays

Function: `MD_attrib()`

Summary: Obtain the attributes of an engine data array.

Prototype: `MD_Attrib MD_attrib(MD_Engine *, int32 idnum);`

Arguments: `idnum` — data array identification number

Description: The attributes of the indicated engine data array are returned.

Return value: Returns the `MD_Attrib` attribute structure for the indicated data array. If an error occurs, the return value is the `MD_Attrib` structure with all fields set to `MD_FAIL` (e.g. test against the `attrib.type` field).

Errors: `MD_ERR_IDNUM` — invalid data ID number

6.2.5 Resizing data arrays

Function: `MD_setlen()`

Summary: Set the length attribute for an engine data array.

Prototype: `int32 MD_setlen(MD_Engine *, int32 idnum, int32 newlen);`

Arguments: `idnum` — data array identification number
`newlen` — new length value

Description: This call sets the length for the specified engine data array, indicating the number of elements used in the array. The array must permit `MD_SETLEN` access. If `newlen` > `attrib.max` with `MD_SETMAX` access also permitted, then the buffer allocation is extended to `attrib.max` = `attrib.len` = `newlen`.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number
`MD_ERR_ACCESS` — access `MD_SETLEN` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newlen` < 0 or `newlen` > `attrib.max` without `MD_SETMAX` access permission

Function: `MD_setmax()`

Summary: Set the maximum allocation attribute for an engine data array.

Prototype: `int32 MD_setmax(MD_Engine *, int32 idnum, int32 newmax);`

Arguments: `idnum` — data array identification number
`newmax` — new maximum array allocation value

Description: This call sets the maximum allocation for the specified engine data array, indicating the maximum number of elements that can be stored in the memory buffer. The array must permit `MD_SETMAX` access. If `newmax` < `attrib.len` with `MD_SETLEN` access also permitted, then the buffer allocation is truncated to `attrib.len` = `attrib.max` = `newmax`.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number
`MD_ERR_ACCESS` — access `MD_SETMAX` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newmax` < 0 or `newmax` < `attrib.len` without `MD_SETLEN` access permission

Function: `MD_resize()`

Summary: Set both the length and maximum allocation attributes for an engine data array.

Prototype: `int32 MD_resize(MD_Engine *, int32 idnum, int32 newlen, int32 newmax);`

Arguments: `idnum` — data array identification number
`newlen` — new length value

`newmax` — new maximum array allocation value

Description: This call sets both the length and maximum allocation for the specified engine data array. The maximum allocation indicates the number of elements that can be stored in the memory buffer. The length indicates the number of elements used in the array. It is necessary that $0 \leq$ `newlen` $\leq$ `newmax`. The array must permit `MD_RESIZE` access (which is the same as both `MD_SETLEN` and `MD_SETMAX` access). If `newlen` $>$ `attrib.max` then the buffer allocation is extended to accomodate. Similarly, if `newmax` $<$ `attrib.len` then the buffer allocation is truncated.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number
`MD_ERR_ACCESS` — access `MD_RESIZE` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newlen` < 0 or `newmax` < 0 or `newmax` $<$ `newlen`

6.2.6 Synchronizing nonblocking routines

Some of the front end API calls that follow (as well as `MD_init()` previously) have nonblocking calling semantics. An implementation of MDAPI that supports nonblocking calls enables use of MDAPI for advanced front ends (e.g. provides a GUI) which cannot afford to lose control for an indefinite length of time. In such an implementation, a nonblocking call returns control back to the front end immediately but has not necessarily completed the results of its operation. The completion of the nonblocking call must be either tested true with `MD_test()` or waited for with `MD_wait()`, synchronizing the front end and engine, before any other API call is invoked.

Note that for the single-threaded MDAPI implementation, all routines run to completion and the synchronization routines are stubbed: `MD_test()` always returns 1 (true) and `MD_wait()` always returns 0 (success).

Function: `MD_test()`

Summary: Tests whether a nonblocking routine has completed.

Prototype: `int32 MD_test(MD_Engine *);`

Description: This call returns immediately, indicating whether or not the most recent nonblocking front end API routine has completed.

Return value: Returns true (positive value) if the nonblocking call has successfully completed or 0 if the call has not yet completed. Returns `MD_FAIL` if the call has completed but an error occurred.

Errors: Depends on the nonblocking routine.

Function: `MD_wait()`

Summary: Wait for a nonblocking routine to complete.

Prototype: `int32 MD_wait(MD_Engine *);`

Description: This call blocks to wait for the most recent nonblocking front end API routine to complete, synchronizing the front end with the engine and MDAPI layer.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: Depends on the nonblocking routine.

6.2.7 Reading from engine data arrays

Function: `MD_read()`

Summary: Read from an entire engine data array.

Prototype: `int32 MD_read(MD_Engine *, int32 idnum, void *buf, int32 len);`

Arguments: `idnum` — data array identification number

`buf` — buffer to be filled from data array

`len` — length of array in elements

Description: Read from the beginning of an engine data array `len` elements and copy into `buf`. Array must permit `MD_READ` access. This call has nonblocking semantics. Front end needs to have allocated sufficient space for `buf` and retains control of this memory. `buf` should not be accessed until after call completes.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_READ` is not permitted

`MD_ERR_RANGE` — `len` < 0 or `len` $>$ `attrib.len`

Function: `MD_readsub()`

Summary: Read from part of an engine data array.

Prototype:
```
int32 MD_readsub(MD_Engine *, int32 idnum, void *buf, int32 nelems,
    int32 first);
```

Arguments: `idnum` — data array identification number

`buf` — buffer to be filled from data array

`nelems` — number of elements to read

`first` — first element to index from

Description: Read from a sub-array of an engine data array and copy into `buf`. Array must permit `MD_READ` access. The sub-array is defined by `first` as the first index and `nelems` as the number of elements (or length). This call has nonblocking semantics. Front end needs to have allocated sufficient space for `buf` and retains control of this memory. `buf` should not be accessed until after the call completes.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_READ` is not permitted

`MD_ERR_RANGE` — `first` < 0 or `nelems` < 0 or `first` $+$ `nelems` $>$ `attrib.len`

`MD_read(e,id,buf,n)` is equivalent to `MD_readsub(e,id,buf,n,0)`.

6.2.8 Writing to engine data arrays

Function: MD_write()

Summary: Write to an entire engine data array.

Prototype: `int32 MD_write(MD_Engine *, int32 idnum, const void *buf, int32 len);`

Arguments: `idnum` — data array identification number

`buf` — buffer used for writing to data array

`len` — length of array in elements

Description: Write to the beginning of an engine data array `len` elements the contents of `buf`. Array must permit `MD_WRITE` access. Front end retains control of the `buf` memory. If communicating with a remote engine, the front end writes to local buffer space instead of immediately modifying the engine data array, so the call does *not* have nonblocking semantics.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_WRITE` is not permitted

`MD_ERR_RANGE` — `len` < 0 or `len` $>$ `attrib.len`

Function: MD_writesub()

Summary: Write to part of an engine data array.

Prototype:
```
int32 MD_writesub(MD_Engine *, int32 idnum, const void *buf, int32 nelems,
    int32 first);
```

Arguments: `idnum` — data array identification number

`buf` — buffer used for writing to data array

`nelems` — number of elements to write

`first` — first element to index from

Description: Write to a sub-array of an engine data array, copied from `buf`. Array must permit `MD_WRITE` access. The sub-array is defined by `first` as the first index and `nelems` as the number of elements (or length). Front end retains control of the `buf` memory. If communicating with a remote engine, the front end writes to local buffer space instead of immediately modifying the engine data array, so the call does *not* have nonblocking semantics.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_WRITE` is not permitted

`MD_ERR_RANGE` — `first` < 0 or `nelems` < 0 or `first` $+$ `nelems` $>$ `attrib.len`

6.2.9 Providing data buffer space

Function: MD_share()

Summary: The front end shares its data buffer with the engine.

Prototype: `int32 MD_share(MD_Engine *, int32 idnum, void *buf, int32 len, int32 max);`

Arguments: `idnum` — data array identification number
`buf` — allocated buffer
`len` — number of elements initialized in buffer
`max` — maximum number of elements available in buffer

Description: The engine data array must permit `MD_SHARE` access. Note that `MD_SHARE` access is provided by the MDAPI layer, not by the engine.

The front end provides the data buffer to be used for this data array. Control of the data buffer is yielded by the front end until either `MD_unshare()` is called on this `idnum` or until `MD_done()`. However, the front end is still responsible for managing this memory (i.e. freeing it) after control is regained.

When the data buffer is established through `MD_share()`, `MD_SETMAX` is no longer permitted (i.e. the memory allocation cannot be resized). Until control of memory is regained, the front end should access it through API calls rather than directly.

Note that an engine data array may have `MD_SHARE` access only if there is no buffer space yet allocated to it (i.e. `attrib.max` = 0). Such an array will have `MD_SHARE` access disabled if the buffer is resized by, say, `MD_setmax()`.

This call does not block, so it *does not* have nonblocking semantics.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number
`MD_ERR_ACCESS` — access `MD_SHARE` is not permitted
`MD_ERR_RANGE` — `len` < 0 or `max` < `len`

Function: MD_unshare()

Summary: Front end regains control of shared data buffer previously offered through `MD_share()`.

Prototype: `void *MD_unshare(MD_Engine *, int32 idnum);`

Arguments: `idnum` — data array identification number

Description: `MD_UNSHARE` access is enabled on a data buffer (by the MDAPI layer) by a successful call to `MD_share()`.

After successful completion of `MD_unshare()`, the engine data buffer will no longer have any memory allocation (i.e. `attrib.max` = 0) and `MD_SHARE` access will again be enabled, returning the engine data buffer to its prior state before `MD_share()` was invoked on it.

This call does not block, so it *does not* have nonblocking semantics.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number
`MD_ERR_ACCESS` — access `MD_UNSHARE` is not permitted

6.2.10 Accessing data array buffers directly

Function: `MD_direct()`

Summary: Obtain direct access to an engine data array buffer.

Prototype: `void *MD_direct(MD_Engine *, int32 idnum);`

Arguments: `idnum` — data array identification number

Description: Buffer must permit `MD_DIRECT` access.

This routine makes the data array buffer directly accessible to the front end. Hoever, it must be used with caution. For instance, if `MD_setmax()` or `MD_resize()` resizes the memory buffer of a data array, this will most likely invalidate any pointer returned for that array prior to the resizing.

This call has nonblocking semantics. For a remote engine MDAPI implementation, this routine might have to allocate local buffer for the array and populate it.

Return value: Returns either a pointer to the data array memory buffer or `NULL` if array has `attrib.max` = 0 or if an error occurs.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_DIRECT` is not permitted

`MD_ERR_MEMALLOC` — local buffer space cannot be allocated

Function: `MD_setmod()`

Summary: Indicate that a directly accessed engine data array buffer has been modified.

Prototype: `int32 MD_setmod(MD_Engine *, int32 idnum);`

Arguments: `idnum` — data array identification number

Description: Buffer must permit `MD_DIRECT` access.

This routine marks the "dirty bit" on the access flag, indicating that the contents of the data array has been modified. So this should be called if the front end directly modifies the memory buffer obtained from `MD_direct()`.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_IDNUM` — invalid data ID number

`MD_ERR_ACCESS` — access `MD_DIRECT` is not permitted

6.2.11 Updating data array modifications to the engine

Function: `MD_update()`

Summary: Provide the engine with data array buffer modifications.

Prototype: `int32 MD_update(MD_Engine *);`

Description: For a front end connected to a remote engine, writing to data array buffers will (most likely) be cached local to the front end. This routine sends all updated data array buffers (i.e. those marked with the "dirty bit") to the engine.

Note that `MD_run()` must perform the functionality of `MD_update()` before running the simulation. Judicious use of `MD_update()` might reduce the overhead required upon calling `MD_run()`.

This call has nonblocking semantics.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: None for single-threaded MDAPI implementation (no functionality).

6.2.12 Running the simulation

Function: `MD_firststep()`

Summary: Initialize the step number counter before running a simulation.

Prototype: `int32 MD_firststep(MD_Engine *, int32 firststep);`

Arguments: `firststep` — first step number

Description: This routine sets the step number counter to value `firststep`. It also has the side effect of resetting callback processing: all callbacks start their step increments together with respect to this initial step number value.

The step numbering defaults to start at value 0 without calling this routine.

Return value: Returns 0.

Errors: None (always succeeds).

Function: `MD_stepnum()`

Summary: Obtain the step number counter.

Prototype: `int32 MD_stepnum(MD_Engine *);`

Return value: Returns the step number counter.

Errors: None (always succeeds).

Function: `MD_run()`

Summary: Run a simulation, integrating the system for the specified number of steps.

Prototype: `int32 MD_run(MD_Engine *, int32 numsteps, int32 runflags);`

Arguments: `numsteps` — nonnegative integer

`runflags` — runtime flags

Description: Calling with `numsteps` = 0 simply ensures that the engine is fully initialized, which might include evaluation of the forces for the current position configuration. In this case, callbacks are also processed if they have not already been for this step (see the callback notes that follow).

The `runflags` may include `MD_CBFIRST` bitwise ORed with any relevant engine-dependent flags.

This call has nonblocking semantics. Communication with a running simulation is performed via pre-established callbacks.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_CALLBACK` — callback routine returned nonzero

`MD_ERR_CBSHARE` — callback shared buffer failed (access does not permit resizing of target engine data array buffer)

`MD_ERR_RANGE` — `numsteps` < 0 or bug in engine resizing data arrays

`MD_ERR_MEMALLOC` — memory cannot be allocated (resizing data arrays by engine or from callback shared buffer)

`MD_ERR_RUN` — default error state if engine run routine returns nonzero without setting the error state

`MD_ERR_ACCESS` — indicates bug in engine (incorrect attempt to resize data arrays or acknowledge data array modification)

`MD_ERR_CHECK` — consistency check failed, indicates bug in engine (either step counter is not at projected value or callbacks were not processed at some expected step number)

The engine might also set its own error state values.

6.2.13 Establishing callbacks

The callback routines are called during the engine execution of an `MD_run()` call. Establising callbacks provides the means for the front end to receive data from and send data to the engine without interrupting the simulation.

Function: `MD_callback()`

Summary: Register a standard callback.

Prototype:
```
int32 MD_callback(MD_Engine *,
    int32(*cb)(void *info, MD_Cbdata *data, int32 len, int32 stepnum),
    void *info, MD_Cbdata *data, int32 datalen, int32 stepincr);
```

Arguments: `cb` — points to a standard callback function

The callback function will receive the following arguments:

`info` — pointer to front end specific information

`data` — array of `MD_Cbdata` to receive engine data array contents (the same data array as below but containing current data array contents)

`len` — length of the `MD_Cbdata` array

`stepnum` — the current step number

All of these callback function argments, except `stepnum`*, are from the following arguments.*

`info` — (to be passed to the callback function)

`data` — array of `MD_Cbdata` to setup call to `cb`

`datalen` — length of the `MD_Cbdata` array

`stepincr` — how often to call back the `cb` function, counting from step number value set using `MD_firststep()`

Description: This routine establishes a callback to function `cb` providing engine data array information through the `MD_Cbdata` array `data` of length `datalen` along with front end specific information pointed to through `info`. See the description of `MD_Cbdata` from Sec. 6.1 for an explanation on referencing engine data sub-arrays and front end access rights. A standard callback will be called every `stepincr` steps counting from the step number value set by using `MD_firststep()`. The standard callbacks are called right after the completion of a time integration step, so their purpose is to log data such as atom trajectories, monitor energies or temperature, etc. A function `cb` can be registered to receive multiple callbacks and even to receive different data sets through a different `MD_Cbdata` array on each invocation.

A callback function relinquishes any access rights to the `MD_Cbdata` array data after returning control. This means, for instance, performing asynchronous I/O to write trajectory files requires copying the trajectory data received through the `MD_Cbdata` array into a separate buffer before returning control back to the MDAPI layer.

The `MD_Cbdata` array must persist until this callback is un-registered (see `MD_callback_undo()` description) or until cleanup through `MD_done()`.

The return value of the callback function has significance. The function should return 0 to indicate success and nonzero (preferably `MD_FAIL`) for an error. Upon failure of a callback, the engine is expected to stop the simulation and return control of its run routine back to the MDAPI layer, which in turn returns control back to the front end with an error message from the `MD_run()` call. So this provides the front end with a mechanism to terminate the engine.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

`MD_ERR_IDNUM` — invalid data ID number for some `MD_Cbdata` element

`MD_ERR_ACCESS` — access `MD_CBREAD` and/or `MD_CBWRITE` or `MD_CBSHARE` not permitted, or attempt to combine `MD_CBSHARE` with other access

`MD_ERR_RANGE` — `stepincr` ≤ 0 or, for some `MD_Cbdata` element, `first` < 0 or `nelems` < -1 or `first` + `nelems` $>$ `attrib.len`

Function: `MD_callback_undo()`

Summary: Un-register a standard callback.

Prototype:
```
int32 MD_callback_undo(MD_Engine *,
    int32 (*cb)(void *info, MD_Cbdata *data, int32 len, int32 stepnum));
```

Arguments: `cb` — points to a standard callback function

Description: Any previously registered callbacks to routine `cb` are removed from the standard callback queue. This call will succeed (but without doint anything) if there are no pointers to `cb` existing in the queue. Calling with `cb` = `NULL` will remove *all* callbacks from the queue.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

Function: MD_fcallback()

Summary: Register a force callback.

Prototype:
```
int32 MD_fcallback(MD_Engine *,
    int32 (*fcb)(void *, MD_Cbdata *, int32 len, int32 stepnum, double frac),
    void *info, MD_Cbdata *data, int32 datalen);
```

Most of what is described for the `MD_callback()` standard callback registration applies to force callbacks. The differences are noted below.

Arguments: `fcb` — points to a force callback function

Additional argument received by callback:

`frac` — time step fraction, meaning that the force is being evaluated for the atom position approximation at time $t = \Delta t(\texttt{stepnum} + \texttt{frac})$. For instance, with leapfrog (velocity-Verlet) integration, `frac` = 1.0, since the force is evaluated for the new positions, but since the step has not yet completed, `stepnum` retains the value for the current step. Depending on the integrator used, `frac` could actually be greater than 1 or less than 0.

Note that the `stepincr` argument is no longer needed since force callbacks are processed on every force evaluation.

Description: Force callbacks are the intended mechanism for the front end to supply external forces to the simulation without having to know details about the particular integrator being used to propagate the system. The main difference between force callbacks and standard callbacks is when the callbacks are processed. Standard callbacks are processed at the beginning of a new step being reached. Force callbacks are processed during each force evaluation, which, depending on the integration method, often occurs in the middle of a step.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

`MD_ERR_IDNUM` — invalid data ID number for some `MD_Cbdata` element

`MD_ERR_ACCESS` — access `MD_FCBREAD` and/or `MD_FCBWRITE` or `MD_FCBSHARE` not permitted, or attempt to combine `MD_FCBSHARE` with other access

`MD_ERR_RANGE` — `stepincr` ≤ 0 or, for some `MD_Cbdata` element, `first` < 0 or `nelems` < -1 or `first` + `nelems` $>$ `attrib.len`

Function: MD_fcallback_undo()

Summary: Un-register a force callback.

Prototype:
```
int32 MD_fcallback_undo(MD_Engine *,
    int32 (*fcb)(void *, MD_Cbdata *, int32 len, int32 stepnum, double frac));
```

Arguments: `fcb` — points to a force callback function

Description: Any previously registered callbacks to routine `fcb` are removed from the force callback queue. This call will succeed (but without doing anything) if there are no pointers to `fcb` in the queue. Calling with `fcb` = `NULL` will remove *all* force callbacks from queue.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

Function: MD_msgcallback()

Summary: Register a message callback function.

Prototype:
```
int32 MD_msgcallback(MD_Engine *,
    int32 (*msgcb)(void *info, const char *msg, int32 stepnum),
    void *info);
```

Arguments: `msgcb` — points to the message callback function

The message callback function receives the following arguments:

`info` — pointer to front end specific information

`msg` — the text message, a nil-terminated string

`stepnum` — the current step number

`info` — (to be passed to the message calback function)

Description: This routine establishes a function to receive message callbacks. This is simply a mechanism for the engine to provide the front end with a status message (as a nil-terminated string) whenever it wants to document an event. The return value from a message callback function has the same significance as that from a force or standard callback. However, unlike the other callback types, message callback functions are not guaranteed to be called by the engine.

Although for consistency with the other callback types it is possible to register multiple message callbacks, this feature does not seem to serve much purpose in practice since they will all receive exactly the same message.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

Function: MD_msgcallback_undo()

Summary: Un-register a message callback function.

Prototype:
```
int32 MD_msgcallback_undo(MD_Engine *,
    int32 (*msgcb)(void *info, const char *msg, int32 stepnum));
```

Arguments: `msgcb` — points to a message callback function

Description: Any previously registered callbacks to routine `msgcb` are removed from the message callback queue. This call will succeed (but without doing anything) if there are no pointers to `msgcb` in the queue. Calling with `msgcb` = `NULL` will remove *all* callbacks from the message callback queue.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

6.2.14 Using engine-defined types

Function: MD_type()

Summary: Obtain the type number from its name.

Prototype:
```
int32 MD_type(MD_Engine *, const char *name);
```

Arguments: `name` — type name, nil-terminated string

Description: Given the nil-terminated string name of a data type, return its type number. Case sesitive string matching is performed. Note that different engines might define types of the same name, but this does not mean that they will be assigned the same type identification number.

Return value: Returns the type identification number (positive `int32`) corresponding to the recognized name or `MD_FAIL` if no match is found.

Errors: `MD_ERR_NAME` — name does not match any defined data types

Function: `MD_type_name()`

Summary: Obtain the type name from its number.

Prototype: `const char *MD_type_name(MD_Engine *, int32 type);`

Arguments: `type` — type identification number

Description: Given the type identification number, return the corresponding type name. This routine is the inverse of `MD_type()`.

Return value: Returns the nil-terminated string name or `NULL` on error.

Errors: `MD_ERR_TYPENUM` — invalid type number

Function: `MD_type_namelist()`

Summary: Obtain the list of type names.

Prototype: `const char **MD_type_type_namelist(MD_Engine *, int32 listlen);`

Arguments: `listlen` — points to an integer variable

Description: The list of names of all defined data types is returned, with the length of the list returned through the `listlen` pointer.

Return value: Returns a pointer to an array of nil-terminated strings and modifies the value of the `int32` variable pointed to by `listlen`.

Errors: None (always succeeds).

Function: `MD_type_memberlist()`

Summary: Obtain the member list for the defined type.

Prototype: `const MD_Member *MD_type_memberlist(MD_Engine *, int32 type, int32 *listlen);`

Arguments: `type` — type identification number
`listlen` — points to an integer variable

Description: Given the type number, return the array of `MD_Member` for that type, with the array length returned through `listlen`. The `MD_Member` array provides information on the members of the defined type (i.e. think C `struct` members). See the description of `MD_Member` in Sec. 6.1.

Return value: Returns a pointer to an array of `MD_Member` and modifies the value of the `int32` varialbe pointed to by `listlen`. `NULL` is returned if an error occurs.

Errors: `MD_ERR_TYPENUM` — invalid type number

Function: `MD_type_member()`

Summary: Obtain the address of a type member.

Prototype:
```
void *MD_type_member(MD_Engine *, int32 type, const void *obj,
    const char *member_name, MD_Member *pmember);
```

Arguments:
`type` — type identification number

`obj` — points to an object instance of the indicated type

`member_name` — nil-terminated string name of member from this type

`pmember` — either points to a variable of `MD_Member` or is `NULL`

Description: Given the member named `member_name` along with the object instance `obj` of the indicated type, a pointer to that member is returned. Case sensitive string matching is performed on the member name. The `pmember` field, if non-`NULL`, receives the `MD_Member` structure for the named member.

The first time this routine is called for a particular type, the MDAPI layer builds a search table of the member names and their address offsets from the start of the type. Subsequent calls on that type are fast, entailing a quick lookup in the search table.

Return value: Returns a pointer to the member in `obj` or `NULL` on error.

Errors:
`MD_ERR_MEMALLOC` — memory cannot be allocated

`MD_ERR_TYPENUM` — invalid type number

`MD_ERR_NAME` — name does not match any members of specified type

`MD_ERR_NEWTYPE` — indicates a bug in the engine (occurs if more than one member has the same name, really indicating an error from engine initialization that was not discovered until this routine)

6.2.15 Handling errors

Function: `MD_errnum()`

Summary: Obtain the error number.

Prototype:
```
int32 MD_errnum(MD_Engine *);
```

Return value: Returns the (nonzero) error number if an error condition has been set or 0 if there is no error.

Function: `MD_errmsg()`

Summary: Obtain the error message.

Prototype:
```
const char *MD_errmsg(MD_Engine *);
```

Return value: Returns the error message description (nil-terminated string) for the current error condition.

Function: MD_reset()

Summary: Reset the error state.

Prototype: `int32 MD_reset(MD_Engine *);`

Description: Attempts to reset the current error state to `MD_ERR_NONE`. The call will succeed if the current error state is recoverable, in which case subsequent calls to `MD_errnum()` will return 0. The call will fail if the current error state is considered fatal, in which case the front end should terminate use of this engine object by calling `MD_done()`.

Return value: Returns 0 on success, resetting the error state, or `MD_FAIL` for an unrecoverable error.

Function: MD_engine_name()

Summary: Obtain the name of the engine.

Prototype: `const char *MD_engine_name(MD_Engine *);`

Return value: Returns the engine name, as specified in the `MD_init()` call. Intended for diagnostic purposes.

6.3 Engine Interface Specification

The routines presented here provide the engine interface to the MDAPI. The definitions and prototypes are all found in the `mdengine.h` header file. The engine receives a pointer to `MD_Front` as its handle to the front end to be used in all of the routines. It is recommended that the engine allocate its state on the heap so that its code is reentrant. The engine should have its state accessible through a single pointer stored within `MD_Front`, originally initialized by calling `MD_setup_engine()`, then accessed later by calling `MD_engine_data()`. The engine establishes data arrays to be accessed by the front end, each of which is an instance of `MD_Engdata` that is contained as part of the engine state.

The engine has three entry points from the MDAPI layer:

1. the initialization routine, called through `MD_init()`;
2. the run routine, called through `MD_run()`;
3. the cleanup routine, called through `MD_done()`.

Each of these three code sections has available to it a particular set of engine API routines; the function definitions that follow are gathered to reflect these categories.

6.3.1 Objects

Type: MD_Front

Summary: The front end object.

Definition: `typedef MD_Interface MD_Front;`

Description: This is just the `MD_Interface` from Sec. 6.1 renamed to appear as the front end object. The engine receives into its initialization, cleanup, and run routines, a pointer to `MD_Front` as the handle to the front end. All of the engine API calls operate on the front end object, requiring a pointer to the object passed as the first argument, analogous to the C++ `this` pointer.

Type: MD_Engdata

Summary: The engine data array object.

Definition:
```
typedef struct MD_Engdata_tag {
    void *buf;
    MD_Attrib attrib;
    /* the rest is opaque to engine */
} MD_Engdata;
```

Description: Each engine data array, meaning an array that is accessible to the front end, is contained within an `MD_Engdata` object. The three routines that establish an engine data array, either `MD_engdata()` or one of the alternatives, return a pointer to `MD_Engdata`. The data buffer is available througn `buf`, and `attrib` provides the length of the array (through `attrib.len`).

Generally, the engine should not change the value of the `buf` pointer. If the engine needs to resize the data array, the array should be established using some combination of the `MD_ESETLEN` and `MD_ESETMAX` access flags, permitting `MD_engdata_setlen()`, `MD_engdata_setmax()`, and/or `MD_engdata_resize()` calls by the engine to control the array length and allocation.

Type: MD_Callback

Summary: The callback object.

Definition:
```
typedef struct MD_Callback_tag {
    MD_Cbdata *cbarg;
    int32 cbarglen;
    int32 stepincr;
    int32 nextstepnum;
    /* the rest is opaque to engine */
} MD_Callback;
```

Description: The `MD_Callback` object stores data needed for invoking a callback. One `MD_Callback` object is allocated for each standard and force callback registered by the front end. The MDAPI layer keeps one array of `MD_Callback` for standard callbacks and another array of `MD_Callback` for force callbacks. The engine can obtain either array in order to discover exactly what data is needed by the front end and when. The `cbarg` member provides the array of `MD_Cbdata`, describing each sub-array of each engine data array accessed by the callback routine (see Sec. 6.1 for details), with the length of the `MD_Cbdata` array given by member `cbarglen`. For a standard callback, the `stepincr` member indicates how often the callback is to be processed, and the `nextstepnum` member provides the particular step number at which the callback will be invoked.

Calls during engine initialization

The engine initialization routine, with prototype

```
int32 engine_init(MD_Front *f);
```

is called once from the front end API `MD_init()` call, where `f` points to the front end object. The name *engine_init* used here is intended as a descriptive placeholder for whatever name is to be used by the engine. The front end either passes this function pointer to `MD_init()` or else an advanced implementation of MDAPI (featuring dynamic loading of engines) specifies a way of gaining the initialization routine name from the `engine_name` string.

The main goal of engine initialization is to setup all data array communication that will take place between front end and engine. Initialization must allocate the engine data structures, define any new types and error conditions, establish the engine data array `MD_Engdata` objects, and setup the engine run routine. The following definitions present the engine API routines intended to be used during initialization. Any API calls here that fail are fatal, so the engine should return `MD_FAIL` immediately. The cleanup routine will still be called, so the freeing of memory allocations can be delayed until then. Note that it is also possible to call certain other routines, such as the data array resizing routines.

6.3.2 Setup engine handle

Function: `MD_setup_engine()`

Summary: Setup a handle for the engine.

Prototype: `int32 MD_setup_engine(MD_Front *, void *engine);`

Arguments: `engine` — points to main engine data structure

Description: This routine must be called before any other engine API routines, because it checks the consistency of the MDAPI version number. The pointer value passed as `engine` will later be returned by `MD_engine_data()` as the handle to the main engine data structure.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_VERSION` — inconsistent version number

6.3.3 Establising data arrays

Function: `MD_engdata()`

Summary: Create an engine data array with memory managed by MDAPI layer.

Prototype:
```
MD_Engdata *MD_engdata(MD_Front *, const char *name, int32 type,
    int32 access);
```

Arguments: `name` — nil-terminated string identifier for this data array

`type` — type number of data elements

`access` — bit field indicating front end access rights to data array

Description: This routine is the typical way for the engine to setup a data array. The MDAPI performs memory management for this array, initially with `attrib.len` = `attrib.max` = 0. The `name` string must persist until engine cleanup. The `type` number should either be one of the predefined data type number constants from Sec. 6.1 or some value returned by a successful call to `MD_new_type()` for an engine-defined data type. The `access` argument should be some bitwise ORing of the engine access flag constants from Sec. 6.1. Depending on these access flag values, the MDAPI may also enable the `MD_SHARE` flag for the data array. Specifically, the MDAPI enables `MD_SHARE` if `MD_WRITE` and `MD_RESIZE` are enabled and if `MD_NOSHARE` and `MD_ERESIZE` are disabled. Note that this is the only constructor for `MD_Engdata` that will permit `MD_SHARE` access.

Return value: Return value is pointer to the `MD_Engdata` object held by the MDAPI layer or `NULL` on error.

Errors: `MD_ERR_NEWDATA` — cannot create new engine data (due to incorrect attributes or name conflict with another engine data array)

`MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: This routine applies to most engine data arrays. Length is unknown a priori, instead determined by front end input files. In this case, the front end is responsible for setting the length of data arrays like atom, bond, angle, etc., based on size of the molecular system, with the memory allocation all managed by the MDAPI layer.

Function: `MD_engdata_buffer()`

Summary: Create an engine data array with a preallocated buffer supplied by the engine.

Prototype:
```
MD_Engdata *MD_engdata_buffer(MD_Front *, const char *name, MD_Attrib attrib,
    void *buf);
```

Arguments: `name` — nil-terminated string identifier for this data array

`attrib` — attribute for data array

`buf` — points to the memory buffer

Description: This routine lets the engine establish a data array with a static or preallocated buffer. The `name` string and `buffer` space must persist until engine cleanup. The `attrib` argument of type `MD_Attrib` (described in Sec. 6.1) not only provides the type number and access flags, but also should indicate the maximum allocation of the buffer and usable array length. The `buffer` must be large enough to accomodate `attrib.max`. The `attrib.max` value will remain fixed, so that the `MD_SETMAX` and `MD_ESETMAX` flags are disabled.

Make sure that the static buffers accessed through `MD_Engdata` do not overlap or reference each other. Although it would work whenever the front end and engine share the same memory address space, it can produce unpredictable results whenever the front end and engine are communicating remotely.

Return value: Return value is pointer to the `MD_Engdata` object held by the MDAPI layer or `NULL` on error.

Errors: `MD_ERR_NEWDATA` — cannot create new engine data (due to incorrect attributes or name conflict with another engine data array)

`MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: The engine can permit access to a scalar quantity (1-element array) having fixed `attrib.len` = `attrib.max` = 1.

Function: MD_engdata_manage()

Summary: Create an engine data array with buffer space managed by engine.

Prototype:
```
MD_Engdata *MD_engdata_manage(MD_Front *, const char *name, MD_Attrib attrib,
    void *buf, void *(*realloc)(void *ptr, size_t size));
```

Arguments:
`name` — nil-terminated string identifier for this data array

`attrib` — attribute for data array

`buf` — points to initial memory buffer

`realloc` — routine to be called by MDAPI to manage resizing for this buffer

Description: This routine permits the engine to control the method of memory management of the buffer allocation. The `name` string should persist until engine cleanup. The `buf` should either point to buffer space preallocated using the provided `realloc` routine or should be `NULL`. The `attrib` argument of type `MD_Attrib` (described in Sec. 6.1) not only provides the type number and access flags, but also should indicate the maximum allocation of `buf` and the inital usable array length. The `realloc` routine must have semantics identical with the `realloc()` function from the C standard library. This means that a call to `realloc` on `ptr` with `size` = 0 frees the memory allocation. The engine cleanup routine might want to call `MD_free_data()` to force data array buffer memory to be freed before the main engine data structure, in the event that the supplied `realloc` routine depends on the engine state.

Return value: Return value is pointer to the `MD_Engdata` object held by the MDAPI layer or `NULL` on error.

Errors:
`MD_ERR_NEWDATA` — cannot create new engine data (due to incorrect attributes or name conflict with another engine data array)

`MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: One way of computing the modified Hamiltonian involves the engine keeping an array of previously computed position and velocity buffers needed for centered differencing. Doing this allows the engine to switch the `MD_Engdata buf` pointer to point to any of the buffers. When memory is freed, the MDAPI calls the same memory management routine to free the "current" buffer as does the engine to free the other buffers.

6.3.4 Aliasing data arrays

Function: MD_engdata_alias()

Summary: Setup an alternative name for an engine data array.

Prototype:
```
int32 MD_engdata_alias(MD_Front *, const char *name, MD_Engdata *e);
```

Arguments:
`name` — nil-terminated string identifier to alias the indicated engine data array

`e` — pointer to previously established `MD_Engdata` object

Description: The engine can set an alternative name (alias) for an engine data array. This new name can be used through the front end API routines the same as the original name. The `name` string should persist until engine cleanup.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors:
`MD_ERR_NEWDATA` — cannot create alias (due to name conflict, string name has already been used for some other engine data array)

`MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: NAMD has its own names for simulation parameters. If these conflict with the MDAPI naming conventions, use both names to avoid breaking compatibility between old NAMD "config" files and a redesigned NAMD engine.

6.3.5 Defining new types

Function: `MD_new_type()`

Summary: Engine defines a new data type.

Prototype:
```
int32 MD_new_type(MD_Front *, const char *name, const MD_Member *member,
    int32 memberlen, int32 nbytes);
```

Arguments: `name` — nil-terminated string identifier for the type

`member` — array of `MD_Member` comprising the members of the type

`memberlen` — length of the array, number of elements

`nbytes` — size of the new type in bytes

Description: The MDAPI layer internally calls this routine on all of its predefined data types (see Chap. 2) before invoking the engine initialization routine. The `member` array describes each member of the new type (i.e. C `struct`), with `MD_Member` described in Sec. 6.1. The `name` string and the `member` array and its elements must persist until engine cleanup. For the value of `nbytes`, just use the C `sizeof` operator on the actual type.

A new type can be constructed using member elements of any previously defined type. The only caveat is that members must be positioned so that 4-byte numeric quantities (i.e. `int32`, `float`, `MD_Fvec`) start on 4-byte boundaries and 8-byte numeric quantities (i.e. `double`, `MD_Dvec`) start on 8-byte boundaries, counting from 0.

Return value: Return value is the new (nonzero) type number on success or `MD_FAIL` on error.

Errors: `MD_ERR_NEWTYPE` — cannot create new type (due to invalid type numbers in member list, misaligned numeric types, or disagreement between `nbytes` and internally computed size)

`MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: The following example illustrates the operation of the function for a predefined data type.

```
typedef struct MD_BondPrm_tag {
  double k;
  double r0;
  MD_Name type[2];
} MD_BondPrm;

const MD_Member bondprm_member[] =
  { { MD_DOUBLE, 1, "k" }, { MD_DOUBLE, 1, "r0" }, { MD_NAME, 2, "type" } };

bondprm_type = MD_new_type(frnt, bondprm_member,
  sizeof(bondprm_member)/sizeof(bondprm_member[0]), sizeof(MD_BondPrm));
```

After doing this, `MD_BondPrm` can now be used as a member of newly defined types. Note that the type number constants defined in `mdtypes.h` are each equal to the return values from the respective calls to `MD_new_type()`.

Two recommended types for the engine to create are `Param` and `Result`. The members of `Param` include the engine simulation parameters, and the members of `Result` include the scalar quantities computed by the engine during the simulation. See Chap. 3 for more details.

6.3.6 Defining new errors

Function: `MD_new_error()`

Summary: Engine defines a new error condition.

Prototype: `int32 MD_new_error(MD_Front *, const char *errmsg, int32 isfatal);`

Arguments: `errmsg` — nil-terminated string of error description

`isfatal` — zero indicates not fatal, nonzero indicates fatal

Description: This routine allows the engine to define its own errors to be reported from the run routine. Fatal errors are usually (but not necessarily) used for system-related errors that appear to be unrecoverable. Error values are used in calls to `MD_error()`. The `errmsg` string must persist until engine cleanup.

Return value: Returns new (nonzero) error number on success or `MD_FAIL` on error.

Errors: `MD_ERR_MEMALLOC` — memory cannot be allocated

Examples: A couple of error condition definitions might be as follows.

```
too_hot = MD_new_error(frnt, "system temperature too hot", 0);
unstable = MD_new_error(frnt, "simulation is unstable", 1);
  ...
return MD_error(frnt, unstable);
```

6.3.7 Setup the run routine

Function: `MD_setup_run()`

Summary: Sets up the engine run routine.

Prototype:
```
int32 MD_setup_run(MD_Front *,
    int32 (*run)(MD_Front *f, int32 numsteps, int32 flags));
```

Arguments: `run` — points to the run routine to be called by `MD_run()`

The run routine will receive the following arguments:

`f` — pointer to the `MD_Front` front end object

`numsteps` — number of time steps to integrate system

`flags` — status flags

Description: The engine supplied run routine is the entry point into the compute-instensive portion of the MD code that is responsible for evaluating the force field and propagating the molecular system for the indicated number of steps.

The `flags` argument is a bit field status indicator. The `MD_UPDATE` flag will be set if any array with `MD_NOTIFY` access has been modified by the front end. The engine should take appropriate action on any modified arrays, then call `MD_engdata_ackmod()` for each such array. The low order bits (i.e. `0x0000FFFF`) of `flags` are reserved to the engine to define its own status flags.

The run routine is responsible for initiating the processing of callbacks. Message callbacks are solely for the benefit of the engine as a way to communicate text messages back to the front end. Force callbacks (if any) must be processed whenever the force is evaluated. Standard callbacks (if any) are processed after the completion of each step and also once before taking any steps. Any data array buffers that were granted either standard or force callback access might be involved in callback processing, so the engine should not modify these buffers while callback processing is active.

If the force is stale when the run routine is invoked, the run routine should first evaluate the force (which might involve a force callback).

After the completion of each step, before processing standard callbacks, the run routine should call `MD_incrstep()` to update the internal step counter that the MDAPI layer keeps. The value of this counter is avaliable through `MD_stepnum()`.

Return value: Returns 0.

Errors: None (always succeeds).

Calls during engine run routine

The engine run routine is called one or more times through the front end API `MD_run()` call. The run routine performs the integration and necessary force evaluations for the system. Its prototype is

```
int32 engine_run(MD_Front *f, int32 numsteps, int32 flags);
```

as discussed in the `MD_setup_run()` routine. The name *engine_run* is intended as a placeholder for whatever name the engine chooses for this routine.

The main API calls here concern the processing of callbacks. Although the specific callbacks to the front end are managed by the MDAPI layer, the engine is required to initiate them. There are three varieties of callbacks: standard callbacks, force callbacks, and message callbacks. The standard callbacks need to be processed (when so indicated) at the end of an integration step. The force callbacks (for the introduction of external forces) need to be processed during any evaluation of the force to obtain the total forces. The message callback is intended for sending arbitrary text strings that report the engine status to the front end. These three callback types can all be processed in parallel, with nonblocking calling semantics. The only restrictions are that you cannot initiate another callback of the same variety before the first callback of that variety has completed.

If the forces are stale (or have not yet been computed) when the run routine is first called, then immediately after finishing any remaining simulation initialization, the engine is expected to perform an initial evaluation of the forces, then check for a standard callback on the zeroth step. Also, upon completion of an integration step, the engine is expected to increment its step counter. (This is the mechanism by which the MDAPI layer synchronizes the processing of callbacks.)

The following definitions present the engine API routines intended while running the simulation. Note that the engine should not establish new data arrays, define new types, or define new error conditions during the run routine.

6.3.8 Obtaining engine handle

Function: `MD_engine_data()`

Summary: Obtain the pointer to the main engine data structure.

Prototype: `void *MD_engine_data(MD_Front *);`

Return value: Returns the pointer to the main engine data structure initially set by `MD_setup_engine()`.

Errors: None (always succeeds).

6.3.9 Controlling step number counter

Function: `MD_incrstep()`

Summary: Increment the internal step counter kept by the MDAPI layer.

Prototype: `int32 MD_incrstep(MD_Front *);`

Description: This routine should be called once after the completion of each step, before processing standard callbacks.

Return value: Returns 0.

Errors: None (always succeeds).

Function: `MD_stepnum()`

Summary: Obtain the step number counter.

Prototype: `int32 MD_stepnum(MD_Engine *);`

Return value: Returns the step number counter.

Errors: None (always succeeds).

6.3.10 Resizing data arrays

Function: `MD_engdata_setlen()`

Summary: Set the length attribute for an engine data array.

Prototype: `int32 MD_engdata_setlen(MD_Front *, MD_Engdata *e, int32 newlen);`

Arguments: `e` — pointer to previously established `MD_Engdata` object
`newlen` — new length value

Description: This call sets the length for the specified engine data array, indicating the number of elements used in the array. The array must permit `MD_ESETLEN` access. If `newlen` > `attrib.max` with `MD_ESETMAX` access also permitted, then the buffer allocation is extended to `attrib.max` = `attrib.len` = `newlen`.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_ACCESS` — access `MD_ESETLEN` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newlen` < 0 or `newlen` > `attrib.max` without `MD_ESETMAX` access permission

Function: `MD_engdata_setmax()`

Summary: Set the maximum allocation attribute for an engine data array.

Prototype: `int32 MD_engdata_setmax(MD_Front *, MD_Engdata *e, int32 newmax);`

Arguments: `e` — pointer to previously established `MD_Engdata` object
`newmax` — new maximum array allocation value

Description: This call sets the maximum allocation for the specified engine data array, indicating the maximum number of elements that can be stored in the memory buffer. The array must permit `MD_ESETMAX` access. If `newmax` < `attrib.len` with `MD_ESETLEN` access also permitted, then the buffer allocation is truncated to `attrib.len` = `attrib.max` = `newmax`.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_ACCESS` — access `MD_ESETMAX` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newmax` < 0 or `newmax` < `attrib.len` without `MD_ESETLEN` access permission

Function: `MD_engdata_resize()`

Summary: Set both the length and maximum allocation attributes for an engine data array.

Prototype: `int32 MD_engdata_resize(MD_Front *, MD_Engdata *e, int32 newlen, int32 newmax);`

Arguments: `e` — pointer to previously established `MD_Engdata` object
`newlen` — new length value
`newmax` — new maximum array allocation value

Description: This call sets both the length and maximum allocation for the specified engine data array. The maximum allocation indicates the number of elements that can be stored in the memory buffer. The length indicates the number of elements used in the array. It is necessary that $0 \leq$ `newlen` $\leq$ `newmax`. The array must permit `MD_ERESIZE` access (which is the same as both `MD_ESETLEN` and `MD_ESETMAX` access). If `newlen` > `attrib.max` then the buffer allocation is extended to accomodate. Similarly, if `newmax` < `attrib.len` then the buffer allocation is truncated.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_ACCESS` — access `MD_ERESIZE` is not permitted
`MD_ERR_MEMALLOC` — memory cannot be allocated
`MD_ERR_RANGE` — `newlen` < 0 or `newmax` < 0 or `newmax` < `newlen`

6.3.11 Acknowledging data array modification

Function: `MD_engdata_ackmod()`

Summary: Acknowledge the front end modification of an engine data array.

Prototype: `int32 MD_engdata_ackmod(MD_Front *, MD_Engdata *e);`

Arguments: `e` — pointer to previously established `MD_Engdata` object

Description: This call clears the `MD_MODIFY` access flag, acknowledging that the front end modified the data array. The engine should acknowledge all such `MD_Engdata` objects for which it had set `MD_NOTIFY` access.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: `MD_ERR_ACCESS` — status flag `MD_MODIFY` was not set for indicated `MD_Engdata` object

6.3.12 Processing callbacks

Callback processing has nonblocking calling semantics and allows processing of all three types of callbacks simultaneously. The only requirement is that a new set of callbacks of a given type cannot be proceesed until the first set of callbacks of that type has completed. Data arrays being communicated through a callback should not be altered or accessed by the engine until after the processing of callbacks is complete.

Function: `MD_ready_callback()`

Summary: Check if any standard callbacks are ready to be processed.

Prototype: `int32 MD_ready_callback(MD_Front *);`

Description: This routine enables the engine to perform any necessary internal manipulation of data before standard callbacks are processed. Since standard callbacks are setup to be processed after some number of steps, it is possible for this routine to sometimes return true and other times return false.

Return value: Returns true (nonzero) if a standard callback is ready to be processed for the current step number count. Otherwise returns false (zero).

Errors: None (always succeeds).

Function: `MD_exec_callback()`

Summary: Process the standard callbacks.

Prototype: `int32 MD_exec_callback(MD_Front *);`

Description: Standard callback processing should be performed after each increment of the step counter (call to `MD_incrstep()`), where the front end has specified the step number increment for each callback that has been registered. The `MD_ready_callback()` can first be used to determine if there are standard callbacks to be processed on this step number. This routine has nonblocking semantics requiring one or more calls to `MD_test_callback()` or a call to `MD_wait_callback()` to synchronize. Data arrays being communicated during callback processing should not be altered or accessed by the engine until the processing of callbacks is complete.

Return value: Returns 0 on success or `MD_FAIL` on error. If the call fails, the engine should immediately return `MD_FAIL` back to the MDAPI layer.

Errors:
- `MD_ERR_CALLBACK` — callback routine returned nonzero
- `MD_ERR_CBSHARE` — callback shared buffer failed (access does not permit resizing of target engine data array buffer)
- `MD_ERR_CHECK` — consistency check failed, indicates bug in engine (calback not processed at expected step number)
- `MD_ERR_MEMALLOC` — memory cannot be allocated (resizing data arrays from callback shared buffer)

Function: `MD_test_callback()`

Summary: Tests whether standard callback processing has finished.

Prototype:
```
int32 MD_test_callback(MD_Front *);
```

Description: This call returns immediately, indicating whether or not the most recent `MD_exec_callback()` call to process callbacks has completed.

Return value: Returns true (positive value) if the `MD_exec_callback()` has successfully completed or false (zero) if the call has not yet completed. Returns `MD_FAIL` if the call has completed but an error occurred.

Errors: Same as for `MD_exec_callback()`.

Function: `MD_wait_callback()`

Summary: Wait for standard callback processing to finish.

Prototype:
```
int32 MD_wait_callback(MD_Front *);
```

Description: This call blocks to wait for the most recent `MD_exec_callback()` call has completed, synchronizing the engine after the MDAPI callback processing.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: Same as for `MD_exec_callback()`.

Function: `MD_ready_fcallback()`

Summary: Check if any force callbacks are ready to be processed.

Prototype:
```
int32 MD_ready_callback(MD_Front *);
```

Description: This routine enables the engine to perform any necessary internal manipulation of data before force callbacks are processed. Since force callbacks must be processed for each force evaluation, this routine simply indicates whether any force callbacks have been registered by the front end. This means that this routine need only be called once for any invocation of the engine run routine.

Return value: Returns true (nonzero) if any force callbacks are registered. Otherwise returns false (zero).

Errors: None (always succeeds).

Function: MD_exec_fcallback()

Summary: Process the force callbacks.

Prototype: `int32 MD_exec_callback(MD_Front *, double timestepfrac);`

Arguments: `timestepfrac` — adjustment indicating the simulation time corresponding to the position approximation

Description: Force callback processing should be performed for each force evaluation. If the integration method requires multiple force evaluations per step, then force callback processing will also need to be performed multiple times per step. The `MD_ready_fcallback()` routine can first be used to determine if there are any force callbacks registered. This routine has nonblocking semantics requiring one or more calls to `MD_test_fcallback()` or a call to `MD_wait_fcallback()` to synchronize. Data arrays being communicated during callback processing should not be altered or accessed by the engine until the processing of callbacks is complete.

The `timestepfrac` argument indicates the fractional adjustment to the simulation time corresponding to the position approximation. Letting s be the step number counter and Δt be the time step, then the position approximation for this force evaluation is for time $t = s\Delta t +$ `timestepfrac`. Note that the value(s) of `timestepfrac` depend(s) on the integrator being used by the engine. For instance, the leapfrog/Verlet integrator would always have `timestepfrac` = 1.0 since the force is evaluated using the updated positions, but the step number counter has not been incremented. For integration methods that require multiple force evaluations per step, it is possible for `timestepfrac` to be less than 0 or greater than 1.

Return value: Returns 0 on success or `MD_FAIL` on error. If the call fails, the engine should immediately return `MD_FAIL` back to the MDAPI layer.

Errors: `MD_ERR_CALLBACK` — callback routine returned nonzero

`MD_ERR_CBSHARE` — callback shared buffer failed (access does not permit resizing of target engine data array buffer)

`MD_ERR_MEMALLOC` — memory cannot be allocated (resizing data arrays from callback shared buffer)

Function: MD_test_fcallback()

Summary: Tests whether force callback processing has finished.

Prototype: `int32 MD_test_fcallback(MD_Front *);`

Description: This call returns immediately, indicating whether or not the most recent `MD_exec_fcallback()` call to process callbacks has completed.

Return value: Returns true (positive value) if the `MD_exec_fcallback()` has successfully completed or false (zero) if the call has not yet completed. Returns `MD_FAIL` if the call has completed but an error occurred.

Errors: Same as for `MD_exec_fcallback()`.

Function: MD_wait_fcallback()

Summary: Wait for force callback processing to finish.

Prototype: `int32 MD_wait_fcallback(MD_Front *);`

Description: This call blocks to wait for the most recent `MD_exec_fcallback()` call has completed, synchronizing the engine after the MDAPI callback processing.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: Same as for `MD_exec_fcallback()`.

Function: `MD_ready_msgcallback()`

Summary: Check if any message callbacks are ready to be processed.

Prototype: `int32 MD_ready_msgcallback(MD_Front *);`

Description: This routine indicates if any message callbacks are registered, since it is left to the discretion of the engine whether message callbacks will ever be processed. This means that this routine need only be called once for any invocation of the engine run routine.

Return value: Returns true (nonzero) if any message callbacks are registered. Otherwise returns false (zero).

Errors: None (always succeeds).

Function: `MD_exec_msgcallback()`

Summary: Process the message callbacks.

Prototype: `int32 MD_exec_msgcallback(MD_Front *, const char *msg);`

Arguments: `msg` — nil-terminated string message

Description: A message callback permits the engine to send a status message, in the form of a text string, to the front end, perhaps to be printed to `stdout` or saved in a log file. `MD_ready_msgcallback()` routine can first be used to determine if there are message callbacks registered. This routine has nonblocking semantics requiring one or more calls to `MD_test_msgcallback()` or a call `MD_wait_msgcallback()` to synchronize. The `msg` string should persist until completion of the message callback processing.

Return value: Returns 0 on success or `MD_FAIL` on error. If the call fails, the engine should immediately return `MD_FAIL` back to the MDAPI layer.

Errors: `MD_ERR_CALLBACK` — callback routine returned nonzero

Function: `MD_test_msgcallback()`

Summary: Tests whether message callback processing has finished.

Prototype: `int32 MD_test_msgcallback(MD_Front *);`

Description: This call returns immediately, indicating if the most recent `MD_exec_msgcallback()` call to process callbacks has completed.

Return value: Returns true (positive value) if the `MD_exec_msgcallback()` has successfully completed or false (zero) if the call has not yet completed. Returns `MD_FAIL` if the call has completed but an error occurred.

Errors: Same as for `MD_exec_msgcallback()`.

Function: `MD_wait_msgcallback()`

Summary: Wait for message callback processing to finish.

Prototype: `int32 MD_wait_msgcallback(MD_Front *);`

Description: This call blocks to wait for the most recent `MD_exec_msgcallback()` call has completed, synchronizing the engine after the MDAPI callback processing.

Return value: Returns 0 on success or `MD_FAIL` on error.

Errors: Same as for `MD_exec_msgcallback()`.

6.3.13 Obtaining callback data requirements

Function: `MD_callback_list()`

Summary: Obtain the list of standard callbacks.

Prototype: `cosnt MD_Callback **MD_callback_list(MD_Front *, int32 *listlen);`

Arguments: `listlen` — points to an integer variable to receive list length

Description: This call makes available the list of standard callbacks registered by the front end. Inspection allows the engine to determine the data array requirements for callback processing.

Return value: Returns a pointer to the array of `MD_Callback` or `NULL` if no standard callbacks are registered.

Errors: None (always succeeds).

Function: `MD_fcallback_list()`

Summary: Obtain the list of force callbacks.

Prototype: `cosnt MD_Callback **MD_fcallback_list(MD_Front *, int32 *listlen);`

Arguments: `listlen` — points to an integer variable to receive list length

Description: This call makes available the list of force callbacks registered by the front end. Inspection allows the engine to determine the data array requirements for callback processing.

Return value: Returns a pointer to the array of `MD_Callback` or `NULL` if no force callbacks are registered.

Errors: None (always succeeds).

6.3.14 Reporting errors

Function: `MD_error()`

Summary: Set an error condition.

Prototype: `int32 MD_error(MD_Front *, int32 errnum);`

Arguments: `errnum` — error number

Description: This call reports to the MDAPI layer an error condition. The `errnum` argument should be either a predefined error number constant (listed in Sec. 6.1) or an error number value returned from `MD_new_error()`.

Note that in the case of cascading failures, `MD_error()` should be called only once before control is returned back to MDAPI layer. The MDAPI layer also uses this routine for its error reporting.

Return value: Returns `MD_FAIL`, so can be used as the return from a failed engine function.

Function: `MD_errnum()`

Summary: Obtain the error number.

Prototype: `int32 MD_errnum(MD_Engine *);`

Return value: Returns the (nonzero) error number if an error condition has been set or 0 if there is no error.

Function: `MD_errmsg()`

Summary: Obtain the error message.

Prototype: `const char *MD_errmsg(MD_Engine *);`

Return value: Returns the error message description (nil-terminated string) for the current error condition.

Calls during engine cleanup

The engine cleanup routine is called once through the front end API `MD_done()` call. The prototype of the cleanup routine is

`void` *engine_done*`(MD_Front *);`

The engine will need to first call `MD_engine_data()` to obtain a handle to its main data structure.

The primary responsibility for cleanup is for the engine to free memory that it allocated for itself. The only exception is with `MD_Engdata` objects created by `MD_engdata_manage()`, which might have been initially allocated by the engine but will be freed by MDAPI, through a call to the provided `realloc` routine. Typical behavior is for MDAPI layer to first call the engine cleanup routine, then to free all of its data allocations. However, the engine can override this behavior through the `MD_free_data()` routine defined next. Most likely the last thing that the engine cleanup routine does is to free its main data structure, essentially wiping its state.

6.3.15 Freeing data array allocations

Function: `MD_free_data()`

Summary: Free engine data array buffer allocations.

Prototype: `void MD_free_data(MD_Front *);`

Description: Calling this from the engine cleanup routine will force the MDAPI layer to immediately free the `MD_Engdata` array buffer memory allocations. Otherwise, the `MD_Engdata` buffers are freed after the engine cleanup routine has completed.

Examples: Suppose the memory management for the `realloc` routine passed to `MD_engdata_manage()` depends on the engine state. Any such data buffers would need to be freed before the engine cleanup routine wipes the engine state.