  - it can be called by node 0 (master) only,
  - it must be called collectively (collective),
  - it may be called by any node, in a noncollective manner (noncollective - note that this is the default)

- a description of the status of the subroutine as private/public to an API or object

Strongly Suggested:

- a description of action on detecting an error condition
- a terse description of input and output parameters the function gets from or gives to an API
- a description of any side effects to file, common blocks or the RTDB
- a description of any dependencies that the subroutine has, such as certain subroutines must be called before or after calling the current subroutine
- a list of available print levels (We are working on a script to pull this information "automagically" out of the code.)

Examples of nicely documented routines can be found in some directories of the NWChem source tree. (There are also many poor examples, so please follow the above template and do not rely on the form of existing code for guidance.) Some examples are reproduced here, to illustrate the current state of in-source documentation in the code. (There are no outstandingly excellent examples in the code, as yet. Think of it as your opportunity to shine.)

Example 1: in-source documentation of function `rtdb_parallel`

```
  logical function rtdb_parallel(mode)
  logical mode              [input]

c  This function sets the parallel access mode of all databases to {\tt mode}
c  and returns the previous setting. If {\tt mode} is true then accesses are
c  in parallel, otherwise they are sequential.
```

Comment: This function meets about half of the requirements of the desired level of documentation. It lacks a definition of the argument `mode` (although it could be argued that in this case the definition is obvious). It does not provide a definition of the "colectiveness" of the call to the function, nor a definition of private/public routine to the rtdb module.

Example 2: in-source documentation of function `task_energy`

```
      logical function task_energy(rtdb)
      integer rtdb
c
c     RTDB input parameters
c     ---------------------
c     task:theory (string) - name of (QM) level of theory to use
c
c     RTDB output parameters
c     ----------------------
c     task:status (logical)- T/F for success/failure
c     if (status) then
c     .  task:energy (real)   - total energy
c     .  task:dipole(real(3)) - total dipole moment if available
c     .  task:cputime (real)  - cpu time to execute the task
c     .  task:walltime (real) - wall time to execute the task
c
c     Also returns status through the function value
c
```

Comment: This is also a fairly typical example. It is a little terse for the non-telepathic perhaps, but contains most of the essential information on the `task` that executes the operation `energy`.

Example 3: in-source documentation of routine `sym_symmetrize`

```
  subroutine sym_symmetrize(geom, basis, odensity, g_a)
  integer geom       ! [input] Geometry handle
  integer basis      ! [input] Basis handle
  integer g_a        ! [input] Global array to be symmetrized
  logical odensity   ! [input] true=density, false=hamiltonian

c  Symmetrize a skeleton AO matrix (in global array with handle g_a)
c  in the given basis set.  This is nothing more than applying the
c  projection operator for the totally symmetric representation,
c
c     B = (1/2h) * sum(R) [RT * (A + AT) * R]
c
c  where R runs over all operators in the group (including identity),
c  and h is the order of the group.
c
c  Note that density matrices tranform according to slightly different
c  rules to Hamiltonian matrices if components of a shell (e.g.,
c  cartesian d's) are not orthonormal.  (see Dupuis and King, IJQC 11,
```

```
c  613-625, 1977).  Hence, specify \verb+odensity+ as \TRUE\ for
c  density-like matrices and \FALSE\ for all other totally symmetric
c  Hamiltonian-like operators.
c
```

Comment: This is about as good as it gets.

Appendix A

Integral Application Programmer's Interface

This appendix describes the interface to all routines for the NWChem integral API. This includes the actual subroutine documentation and design specifications.

A.1 INT-API: Initialization, Integral Accuracy and Termination

These routines set the scope for the integral computation that is about to be performed.

A.1.1 int_init

This is the main initialization routine for integrals. Default memory requirements, accuracy thresholds, and other initializations for all base integral codes are set here. This routine will read (from the rtdb) any integral settings changed by the user.

Syntax:

```
subroutine int_init(rtdb, nbas, bases)
```

```
integer rtdb        ! [input] run time data base handle
integer nbas        ! [input] number of basis sets to be used
integer bases(nbas) ! [input] basis set handles
```

A.1.2 intd_init

This is the main initialization routine for integral derivatives. Default memory requirements, accuracy thresholds, and other initializations for all base integral codes are set here. This routine will read (from the rtdb) any integral settings changed by the user.

Syntax:

```
subroutine intd_init(rtdb,nbas,bases)

integer rtdb        ! [input] run time data base handle
integer nbas        ! [input] number of basis sets to be used
integer bases(nbas) ! [input] basis set handles
```

A.1.3 int_terminate

This is the main termination routine for integrals. After this call the INT-API is ready for re-initialization.

Syntax:

```
subroutine int_terminate()
```

No formal arguments

A.1.4 intd_terminate

This is the main termination routine for integral derivatives. After this call the INT-API is ready for re-initialization.

Syntax:

```
subroutine intd_terminate()
```

No formal arguments

A.1.5 intdd_terminate

This is the main termination routine for integral second derivatives. After this call the INT-API is ready for re-initialization.

Syntax:

```
      subroutine intdd_terminate()
```

No formal arguments

A.1.6 int_acc_std

This routine sets the integral threshold for radial cutoffs in all integral codes used in the api via a parameter statement. Other routines have access via the apiP.fh common blocks and the set/get API.

Syntax:

```
      subroutine int_acc_std()
```

The default "standard" value for the integral accuracy is:

```
c       parameter(val_def = 1.0d-15)
```

A.1.7 int_acc_high

This routine sets the integral threshold to "high" accuracy for radial cutoffs in all integral codes used in the api via a parameter statement. Other routines have access via the apiP.fh common blocks and the set/get API.

Syntax:

```
      subroutine int_acc_high()
```

The default "high accuracy" value for the integral API is:

```
c       parameter (val_def_high = 1.0d-30)
```

This is needed for certain algorithms within NWChem, e.g., the Schwarz inequality screening.

A.1.8 int_acc_get

This routine returns the current integral threshold for radial cutoffs in all integral codes used in the api via a parameter statement.

Syntax:

```
      subroutine int_acc_get(retval)

      double precision retval ! [output] current threshold
```

A.1.9 int_acc_set

This routine sets the current integral threshold for radial cutoffs in all integral codes used in the api via a parameter statement.

Syntax:

```
subroutine int_acc_set(setval)

double precision setval ! [input] new threshold
```

A.2 INT-API: Memory Managment Routines

These routines compute, store, and return memory requirements for particular "classes" of integral computations. These routines are "overloaded" since the application uses the same query function whether integrals or integral derivatives are computed. For example, `int_mem_2e4c` is used to get the maximum buffer size and scratch array size for both integrals computed using `int_2e4c` and integral derivatives computed using `intd_2e4c`. The INT-API is also designed such that if you initialize INT-API for integral derivatives, the memory estimates are also valid for integrals as well.

A.2.1 int_mem

This routine returns the maximum buffer and scratch array sizes for both one electron and two electron (4 center) integrals.

Syntax:

```
subroutine int_mem(max1e, maxg, mscratch_1e, mscratch_2e)

integer max1e       ! [output] max 1e buffer size
integer maxg        ! [output] max 2e4c buffer size
integer mscratch_1e ! [output] max scr for 1e ints
integer mscratch_2e ! [output] max scr for 2e ints
```

A.2.2 int_mem_1e

This routine returns the maximum buffer and scratch array sizes for one electron (2 center) integrals.

Syntax:

```
subroutine int_mem_1e(max1e, mscratch_1e)

integer max1e       ! [output] max 1e buffer size
integer mscratch_1e ! [output] max scr for 1e ints
```

A.2.3 int_mem_2e4c

This routine returns the maximum buffer and scratch array sizes for two electron (4 center) integrals.

Syntax:

```
subroutine int_mem_2e4c(maxg, mscratch_2e)

integer maxg        ! [output] max 2e4c buffer size
integer mscratch_2e ! [output] max scr for 2e ints
```

A.2.4 int_mem_h1

This routine returns the maximum buffer and scratch array sizes for one electron hamiltonian integrals.

Syntax:

```
subroutine int_mem_h1(maxh1, mscratch_h1)

integer maxh1       ! [output] max buffer size for H1 ints
integer mscratch_h1 ! [output] max scr size for H1 ints
```

A.2.5 int_mem_2eec

This routine returns the maximum buffer and scratch array sizes for two electron (3 center) integrals.

Syntax:

```
subroutine int_mem_2e3c(maxg, mscratch_2e3c)

integer maxg           ! [output] max buf size for 2e3c ints
integer mscratch_2e3c  ! [output] max scr size for 2e3c ints
```

A.2.6 int_mem_2e2c

This routine returns the maximum buffer and scratch array sizes for two electron (2 center) integrals.

Syntax:

```
subroutine int_mem_2e2c(maxg, mscratch_2e2c)

integer maxg          ! [output] max buf size for 2e2c ints
integer mscratch_2e2c ! [output] max scr size for 2e2c ints
```

A.2.7 int_mem_3ov

This routine returns the maximum buffer and scratch array sizes for one electron (3 center) integrals.

Syntax:

```
subroutine int_mem_3ov(maxbuf, mscratch_3ov)

integer maxbuf       ! [output] max buf size for 3 center ov
integer mscratch_3ov ! [output] max scr size for 3 center ov
```

A.2.8 int_mem_print

This routine prints the maximum buffer and scratch array sizes for all known “classes” of integrals.

Syntax:

```
subroutine int_mem_print()
```

A.2.9 intb_mem_2e4c

This routine returns the maximum buffer and scratch array sizes for two electron (4 center) integrals from the blocking interface.

Syntax:

```
subroutine intb_mem_2e4c(maxg, mscratch_2e)

integer maxg        ! [output] max buf size for blocked 2e4c ints
integer mscratch_2e ! [output] max scr size for blocked 2e4c ints
```

A.3 INT-API: Integral Routines

These routines compute and return integrals based on shell quartets or groups of shell quartets.

A.3.1 int_1estv

This is an internal routine that most of the external 1 electron routines call. This is the actual workhorse routine. This routine computes the 1 electron integrals S, T, and V:

$$\begin{aligned}
S &= (\mu|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\nu(X_\nu, r_1) dr_1 \\
T &= (\mu| -\frac{1}{2}\nabla^2|\nu) \\
&= -\frac{1}{2}\int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\nabla^2(r_1) g_\nu(X_\nu, r_1) dr_1 \\
V &= (\mu|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|} g_\nu(X_\nu, r_1) dr_1
\end{aligned}$$

If an ECP is defined then the ECP integral contributions are summed directly into the V integrals.

If a relativistic basis is defined then the one-electron integrals for the case where both shells are relativistic are modified to

$$\begin{aligned}
S &= (\mu^L|\nu^L) - (\mu^S|\frac{\alpha^2}{4}\nabla^2|\nu^S) \\
T &= -\frac{1}{2}(\mu^L|\nabla^2|\nu^S) - \frac{1}{2}(\mu^S|\nabla^2|\nu^L) + \frac{1}{2}(\mu^S|\nabla^2|\nu^S) \\
V &= (\mu^L|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu^L) - \frac{\alpha^2}{4}(\mu^S|\nabla\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}\cdot\nabla|\nu^S)
\end{aligned}$$

Syntax:

```
 subroutine int_1estv(i_basis,ish,j_basis,jsh,lscr,scr,lstv,S,T,V,
&    doS,doT,doV)
```

```
integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
integer lstv              ! [input] length of integral buffer
double precision scr(lscr) ! [scratch] scratch array
double precision S(lstv)   ! [output] overlap integrals
double precision T(lstv)   ! [output] kinetic energy integrals
double precision V(lstv)   ! [output] potential energy integrals
logical doS                ! [input] flag for overlap integrals
logical doT                ! [input] flag for kinetic energy integrals
logical doV                ! [input] flag for potential energy integrals
```

A.3.2 int_1eov

This routine computes the 1 electron overlap integrals (S):

$$\begin{aligned} S &= (\mu|\nu) \\ &= \int_{-\infty}^{\infty} g_{\mu}(X_{\mu}, r_1) g_{\nu}(X_{\nu}, r_1) dr_1 \end{aligned}$$

Syntax:

```
subroutine int_1eov(i_basis,ish,j_basis,jsh,lscr,scr,lov,Ov)

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
integer lov                ! [input] length of Ov buffer
double precision Ov(lov)   ! [output] overlap integrals
```

A.3.3 int_1eke

This routine computes the 1 electron kinetic energy integrals, (T).:

$$T = (\mu|\frac{-1}{2}\nabla^2|\nu)$$

$$= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) \frac{-1}{2} \nabla^2(r_1) g_\nu(X_\nu, r_1) dr_1$$

Syntax:

```
subroutine int_1eke(i_basis,ish,j_basis,jsh,lscr,scr,lke,Ke)

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
integer lke                ! [input] length of Ke buffer
double precision Ke(lke)   ! [output] kinetic energy integrals
```

A.3.4 int_1epe

This routine computes the 1 electron potential integrals, (V):

$$\begin{aligned} V &= (\mu| \sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|} |\nu) \\ &= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) \sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|} g_\nu(X_\nu, r_1) dr_1 \end{aligned}$$

If an ECP is defined then the ECP integral contributions are summed directly into the V integrals.

Syntax:

```
subroutine int_1epe(i_basis,ish,j_basis,jsh,lscr,scr,lpe,Pe)

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
integer lpe                ! [input] length of Pe buffer
double precision Pe(lpe)   ! [output] kinetic energy integrals
```

A.3.5 int_1eh1

This routine computes the 1 electron hamiltonian, ($H1$).

$$
\begin{aligned}
H1 &= T + V \\
T &= (\mu|\frac{-1}{2}\nabla^2|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{-1}{2}\nabla^2(r_1)g_\nu(X_\nu, r_1)dr_1 \\
V &= (\mu|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}g_\nu(X_\nu, r_1)dr_1
\end{aligned}
$$

If an ECP is defined then the ECP integral contributions are summed directly into the $H1$ integrals.

If a relativistic basis is defined then the one-electron integrals for the case where both shells are relativistic are the modified integrals.

Syntax:

```
subroutine int_1eh1(i_basis,ish,j_basis,jsh,lscr,scr,lh1,H1)

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
integer lh1                ! [input] length of H1 buffer.
double precision H1(lh1)   ! [output] one electron
```

A.3.6 int_1eall

This routine computes the 1 electron integrals S, T, and V:

$$
\begin{aligned}
S &= (\mu|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)g_\nu(X_\nu, r_1)dr_1 \\
T &= (\mu| -\frac{1}{2}\nabla^2|\nu)
\end{aligned}
$$

$$
\begin{aligned}
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{-1}{2}\nabla^2(r_1)g_\nu(X_\nu, r_1)dr_1 \\
V &= (\mu|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}g_\nu(X_\nu, r_1)dr_1
\end{aligned}
$$

Syntax:

```
subroutine int_1eall(i_basis,ish,j_basis,jsh,lscr,scr,lstv,S,T,V)

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
integer lstv               ! [input] length of one electron buffers
double precision T(lstv)   ! [output] kinetic integral buffer
double precision V(lstv)   ! [output] potential integral buffer
double precision S(lstv)   ! [output] overlap integral buffer

integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
integer lstv               ! [input] length of integral buffer
double precision scr(lscr) ! [scratch] scratch array
double precision S(lstv)   ! [output] overlap integrals
double precision T(lstv)   ! [output] kinetic energy integrals
double precision V(lstv)   ! [output] potential energy integrals
logical doS                ! [input] flag for overlap integrals
logical doT                ! [input] flag for kinetic energy integrals
logical doV                ! [input] flag for potential energy integrals
```

A.3.7 int_1cg

This routine computes the integral of the one center gaussian.

$$(\mu) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)dr_1$$

Syntax:

```
      subroutine int_1cg(i_basis,ish,lscr,scr,l1g,G1e)

      integer i_basis ! [input] basis set handle
      integer ish     ! [input] lexical shell/contraction index
      integer lscr    ! [input] length of scratch array
      double precision scr(lscr)  ! [scratch] scratch space
      integer l1g     ! [input] length of integral buffer array
      double precision G1e(l1g)   ! [output] one center gaussian integrals
```

A.3.8 int_1e3ov

This routine computes the 3 center overlap integral:

$$(\mu\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\nu(X_\nu, r_1) g_\lambda(X_\lambda, r_1) dr_1$$

Syntax:

```
      subroutine int_1e3ov (i_basis, ish, j_basis, jsh, k_basis, ksh,
     &       lscr, scr, lov3, OV3)
c

      integer i_basis            ! [input] basis set handle for ish
      integer ish                ! [input] i shell/contraction
      integer j_basis            ! [input] basis set handle for jsh
      integer jsh                ! [input] j shell/contraction
      integer k_basis            ! [input] basis set handle for ksh
      integer ksh                ! [input] k shell/contraction
      integer lscr               ! [input] length of scratch v
      double precision scr(lscr) ! [scratch] scratch array
      integer lov3               ! [input] length of 3c overlap buffer
      double precision OV3(lov3) ! [output] 3c overlap integrals
```

A.3.9 int_l1e3ov

This routine computes the 3 center overlap integral with labels and it removes "zero" integrals:

$$(\mu\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\nu(X_\nu, r_1 g_\lambda(X_\lambda, r_1) dr_1$$

Syntax:

```
      subroutine int_l1e3ov(i_basis, ish, j_basis, jsh, k_basis, ksh,
     &       zerotol, lov3, OV3, ilab, jlab, klab, numov3, lscr, scr)

      integer i_basis            ! [input] basis set handle for ish
      integer ish                ! [input] i shell/contraction
      integer j_basis            ! [input] basis set handle for jsh
      integer jsh                ! [input] j shell/contraction
      integer k_basis            ! [input] basis set handle for ksh
      integer ksh                ! [input] k shell/contraction
      double precision zerotol   ! [input] "zero" tolerance threshold
      integer lov3               ! [input] length of 3c overlap array
      double precision OV3(lov3) ! [output] 3c overlap integrals
      integer ilab(lov3)         ! [output] i labels for 3c overlap ints
      integer jlab(lov3)         ! [output] j labels for 3c overlap ints
      integer klab(lov3)         ! [output] k labels for 3c overlap ints
      integer numov3             ! [output] number of integrals and labels
c . . . . . . . . . . . . . . . . .!          generated and returned
      integer lscr               ! [input] length of scratch array
      double precision scr(lscr) ! [input] scratch array
```

A.3.10 int_l1eall

This routine computes the 1 electron integrals S, T, and V with labels and it removes "zero" integrals:

$$
\begin{aligned}
S &= (\mu|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\nu(X_\nu, r_1) dr_1 \\
T &= (\mu|\frac{-1}{2}\nabla^2|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{-1}{2}\nabla^2(r_1) g_\nu(X_\nu, r_1) dr_1 \\
V &= (\mu|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|} g_\nu(X_\nu, r_1) dr_1
\end{aligned}
$$

If an ECP is defined then the ECP integral contributions are summed directly into the V integrals.

Syntax:

```
      subroutine int_l1eall(i_basis, ish, j_basis, jsh, zerotol,
     &       ilab, jlab, lstv, S, T, V, lscr, scr, numstv)

      integer i_basis ! [input] basis set handle for ish
      integer ish     ! [input] i shell/contraction
      integer j_basis ! [input] basis set handle for jsh
      integer jsh     ! [input] j shell/contraction
      integer lscr    ! [input] length of scratch array
      double precision zerotol ! [input] zero threshold for integrals
      double precision scr(lscr) ! [scratch] scratch array
      integer lstv               ! [input] length of one electron buffers
      integer ilab(lstv)         ! [output] i bas. fun. index array
      integer jlab(lstv)         ! [output] j bas. fun. index array
      double precision T(lstv)   ! [output] kinetic integral buffer
      double precision V(lstv)   ! [output] potential integral buffer
      double precision S(lstv)   ! [output] overlap integral buffer
      integer numstv             ! [output] number of returned integrals
```

A.3.11 int_pgen1e

This routine prints a generic one electron block of integrals. This requires the labels be generated and input to this routine.

Syntax:

```
      subroutine int_pgen1e(msg, i_basis, ish, j_basis, jsh,
     &       ilab, jlab, l1e, Gen1e, print_level)

      character*(*) msg           ! [input] informational message
      integer i_basis             ! [input] basis set handle for i shell
      integer j_basis             ! [input] basis set handle for j shell
      integer ish                 ! [input] i shell
      integer jsh                 ! [input] j shell
      integer l1e                 ! [input] number of integrals and labels
      integer ilab(l1e)           ! [input] i basis functions labels
      integer jlab(l1e)           ! [input] j basis functions labels
      double precision Gen1e(l1e) ! [input] integrals to be printed
      integer print_level         ! [input] print level to be used
c. . . . . . . . . . . . . . .! print_level = 0 print lables and integrals
c. . . . . . . . . . . . . . .!             = 1 also print shell info.
```

A.3.12 int_l1eh1

This routine computes the 1 electron hamiltonian, ($H1$) with labels and "zero" integrals removed.

$$
\begin{aligned}
H1 &= T + V \\
T &= (\mu|\frac{-1}{2}\nabla^2|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{-1}{2}\nabla^2(r_1)g_\nu(X_\nu, r_1)dr_1 \\
V &= (\mu|\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\sum_\alpha \frac{-Z_\alpha}{|r_1 - R_\alpha|}g_\nu(X_\nu, r_1)dr_1
\end{aligned}
$$

If an ECP is defined then the ECP integral contributions are summed directly into the $H1$ integrals.

Syntax:

```
 subroutine int_l1eh1(i_basis, ish, j_basis, jsh, zerotol,
&       ilab, jlab, lh1, H1, lscr, scr, numgen)

 integer i_basis ! [input] basis set handle for ish
 integer ish     ! [input] i shell/contraction
 integer j_basis ! [input] basis set handle for jsh
 integer jsh     ! [input] j shell/contraction
 integer lscr    ! [input] length of scratch array
 double precision zerotol   ! [input] zero threshold
 double precision scr(lscr) ! [scratch] scratch array
 integer lh1                ! [input] length of 1e buffers.
 integer numgen             ! [output] number of H1 integrals
 integer ilab(lh1)          ! [output] i bas fun labels array
 integer jlab(lh1)          ! [output] j bas fun labels array
 double precision H1(lh1)   ! [output] 1e H1 integrals
```

A.3.13 int_l1eke

This routine computes the 1 electron kinetic energy integrals, (T). with labels and "zero" integrals removed:

$$
\begin{aligned}
T &= (\mu|\frac{-1}{2}\nabla^2|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{-1}{2}\nabla^2(r_1)g_\nu(X_\nu, r_1)dr_1
\end{aligned}
$$

Syntax:

```
 subroutine int_l1eke(i_basis, ish, j_basis, jsh, zerotol,
&       ilab, jlab, lke, Ke, lscr, scr, numgen)

 integer i_basis ! [input] basis set handle for ish
 integer ish     ! [input] i shell/contraction
 integer j_basis ! [input] basis set handle for jsh
 integer jsh     ! [input] j shell/contraction
 integer lscr    ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] scratch array
 double precision zerotol   ! [input] zero threshold
 integer lke                ! [input] length of 1e buffers
 integer numgen             ! [output] number of Ke integrals
 integer ilab(lke)          ! [output] i bas fun labels array
 integer jlab(lke)          ! [output] j bas fun labels array
 double precision Ke(lke)   ! [output] kinetic energy integrals
```

A.3.14 int_l1eov

This routine computes the 1 electron overlap integrals (S) with labels and "zero" integrals removed:

$$
\begin{aligned}
S &= (\mu|\nu) \\
&= \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)g_\nu(X_\nu, r_1)dr_1
\end{aligned}
$$

Syntax:

```
 subroutine int_l1eov(i_basis, ish, j_basis, jsh, zerotol,
&       ilab, jlab, lov, Ov, lscr, scr, numgen)
```

```
integer i_basis ! [input] basis set handle for ish
integer ish     ! [input] i shell/contraction
integer j_basis ! [input] basis set handle for jsh
integer jsh     ! [input] j shell/contraction
integer lscr    ! [input] length of scratch array
double precision scr(lscr) ! [scratch] scratch array
double precision zerotol   ! [input] zero threshold
integer lov                ! [input] length of overlap buffer
integer numgen             ! [output] num of ints generated
integer ilab(lov)          ! [output] i bas fun labels array
integer jlab(lov)          ! [output] j bas fun labels array
double precision Ov(lov)   ! [output] overlap integral buffer
```

A.3.15 int_l1epe

This routine computes the 1 electron potential integrals, (V): If an ECP is defined then the ECP integral contributions are summed directly into the V integrals. Integrals are computed with labels and "zero" integrals removed.

$$
\begin{aligned}
V &= (\mu|\sum_{\alpha}\frac{-Z_{\alpha}}{|r_1-R_{\alpha}|}|\nu) \\
&= \int_{-\infty}^{\infty} g_{\mu}(X_{\mu},r_1)\sum_{\alpha}\frac{-Z_{\alpha}}{|r_1-R_{\alpha}|}g_{\nu}(X_{\nu},r_1)dr_1
\end{aligned}
$$

Syntax:

```
 subroutine int_l1epe(i_basis, ish, j_basis, jsh, zerotol,
&       ilab, jlab, lpe, Pe, lscr, scr, numgen)

 integer i_basis ! [input] basis set handle for ish
 integer ish     ! [input] i shell/contraction
 integer j_basis ! [input] basis set handle for jsh
 integer jsh     ! [input] j shell/contraction
 integer lscr    ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] scratch array
 double precision zerotol   ! [input] zero integral threshold
 integer lpe                ! [input] length of potential buffer
 integer numgen             ! [output] number of integrals generated
 integer ilab(lpe)          ! [output] i bas fun labels array
```

```
      integer jlab(lpe)          ! [output] j bas fun labels array
      double precision Pe(lpe)   ! [output] potential integrals
```

A.3.16 int_l1gen1e

This routine generates labels for general 2 index one electron integrals. This is mostly unused since the other integral type specific label routines are now used. This routine requires that the integral block be computed prior to the label call. Other routines now integrate label generation with computation.

Syntax:

```
      subroutine int_lgen1e(i_basis, ish, j_basis, jsh, zerotol,
     &       ilab, jlab, l1e, Gen1e, lscr, scr, numgen)

      integer i_basis ! [input] bra basis set handle
      integer ish     ! [input] bra shell lexical index
      integer j_basis ! [input] ket basis set handle
      integer jsh     ! [input] ket shell lexical index
      double precision zerotol ! [input] zero threshold
      integer l1e       ! [input] length of buffers for integrals
      integer ilab(l1e) ! [output] i bas func labels array
      integer jlab(l1e) ! [output] j bas func labels array
      double precision Gen1e(l1e) ! [input/output] 1e integrals
      integer lscr ! [input] length of scratch array
      double precision scr(lscr) ! [scratch] array
      integer numgen ! [output] number of integrals
c . . . . . . . . . . .!           saved and returned
```

A.3.17 int_2e2c

this routine computes the 2 center 2 electron integrals:

$$(\mu|\nu) = \int_{-\infty}^{\infty} g_{\mu}(X_{\mu}, r_1)\frac{1}{r_{12}}g_{\nu}(X_{\nu}, r_2)dr_1dr_2$$

Syntax:

```
      subroutine int_2e2c(brain, ish, ketin, jsh,
     &       lscr, scr, leri, eri)
```

```
integer brain ! [input] bra basis set handle
integer ish   ! [input] shell/contraction index
integer ketin ! [input] ket basis set handle
integer jsh   ! [input] shell/contraction index
integer lscr  ! [input] length of scratch array
double precision scr(lscr) ! [scratch] array
integer leri  ! [input] length of integral array
double precision eri(leri) ! [output] 2e2c integrals
```

A.3.18 int_2e2c

this routine computes the 2 center 2 electron integrals with labels and "zero" integrals removed:

$$(\mu|\nu) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{1}{r_{12}}g_\nu(X_\nu, r_2)dr_1dr_2$$

Syntax:

```
 subroutine int_l2e2c(brain, ish, ketin, jsh,
&       zerotol, leri, eri, nint, ilab, jlab,
&       lscr, scr)

 integer brain ! [input] bra basis set handle
 integer ish   ! [input] shell/contraction index
 integer ketin ! [input] ket basis set handle
 integer jsh   ! [input] shell/contraction index
 integer lscr  ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] array
 double precision zerotol   ! [input] zero threshold
 integer leri  ! [input] length of integral array
 integer nint  ! [output] num of ints computed
 integer ilab(leri) ! [output] i bas func label array
 integer jlab(leri) ! [output] j bas func label array
 double precision eri(leri) ! [output] 2e2c integrals
```

A.3.19 int_l2e3c

this routine computes the 3 center 2 electron integrals with labels and "zero" integrals removed:

$$(\mu|\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{1}{r_{12}}g_\nu(X_\nu, r_2 g_\lambda(X_\lambda, r_2)dr_1dr_2$$

Syntax:

```
 subroutine int_l2e3c(brain, ish, ketin, jsh, ksh,
&       zerotol, canket, leri, eri, nint, ilab, jlab, klab,
&       lscr, scr)

 integer brain ! [input] bra basis set handle
 integer ish   ! [input] shell/contraction index
 integer ketin ! [input] ket basis set handle
 integer jsh   ! [input] shell/contraction index
 integer ksh   ! [input] shell/contraction index
 integer lscr  ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] array
 double precision zerotol   ! [input] zero threshold
 integer leri  ! [input] length of integral array
 integer  nint ! [output] number of integrals computed
 integer ilab(leri) ! [output] i bas fun labels array
 integer jlab(leri) ! [output] j bas fun labels array
 integer klab(leri) ! [output] k bas fun labels array
 double precision eri(leri) ! [output] 2e3c integrals
 logical canket ! [input] canonicalize ket bas. fun. label pairs
```

A.3.20 int_2e3c

this routine computes the 3 center 2 electron integrals:

$$(\mu|\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1)\frac{1}{r_{12}}g_\nu(X_\nu, r_2 g_\lambda(X_\lambda, r_2)dr_1 dr_2$$

Syntax:

```
 subroutine int_2e3c(brain, ish, ketin, jsh, ksh,
&       lscr, scr, leri, eri)

 integer brain ! [input] bra basis set handle
 integer ish   ! [input] shell/contraction index
 integer ketin ! [input] ket basis set handle
 integer jsh   ! [input] shell/contraction index
 integer ksh   ! [input] shell/contraction index
 integer lscr  ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] array
 integer leri  ! [input] length of integral array
 double precision eri(leri) ! [output] 2e3c integrals
```

A.3.21 int_2e4c

this routine computes the 4 center (traditional) 2 electron integrals:

$$(\mu\rho|\nu\lambda) = \int_{-\infty}^{\infty} g_{\mu}(X_{\mu}, r_1) g_{\rho}(X_{\rho}, r_1) \frac{1}{r_{12}} g_{\nu}(X_{\nu}, r_2 g_{\lambda}(X_{\lambda}, r_2) dr_1 dr_2$$

Syntax:

```
 subroutine int_2e4c(brain, ish, jsh, ketin, ksh, lsh,
&       lscr, scr, leri, eri)

 integer brain ! [input] bra basis set handle
 integer ish   ! [input] shell/contraction index
 integer jsh   ! [input] shell/contraction index
 integer ketin ! [input] ket basis set handle
 integer ksh   ! [input] shell/contraction index
 integer lsh   ! [input] shell/contraction index
 integer lscr  ! [input] length of scratch array
 double precision scr(lscr) ! [scratch] array
 integer leri  ! [input] length of integral array
 double precision eri(leri) ! [output] 2e4c integrals
```

A.3.22 int_l2e4c

this routine computes the 4 center (traditional) 2 electron integrals with labels and "zero" integrals removed:

$$(\mu\rho|\nu\lambda) = \int_{-\infty}^{\infty} g_{\mu}(X_{\mu}, r_1) g_{\rho}(X_{\rho}, r_1) \frac{1}{r_{12}} g_{\nu}(X_{\nu}, r_2 g_{\lambda}(X_{\lambda}, r_2) dr_1 dr_2$$

Syntax:

```
 subroutine int_l2e4c(brain, ish, jsh, ketin, ksh, lsh,
&       zerotol, canonicalize, leri, eri, nint, ilab, jlab, klab,
&       llab, lscr, scr)

 integer brain ! [input] bra basis set handle
 integer ish   ! [input] shell/contraction index
```

```
integer jsh   ! [input] shell/contraction index
integer ketin ! [input] ket basis set handle
integer ksh   ! [input] shell/contraction index
integer lsh   ! [input] shell/contraction index
double precision zerotol ! [input] zero threshold
integer lscr  ! [input] length of scratch array
double precision scr(lscr) ! [scratch] array
integer leri  ! [input] length of integral array
integer nint  ! [output] number of integrals computed
integer ilab(leri) ! [output] i bas fun label array
integer jlab(leri) ! [output] j bas fun label array
integer klab(leri) ! [output] k bas fun label array
integer llab(leri) ! [output] l bas fun label array
double precision eri(leri) ! [output] 2e4c integrals
logical canonicalize ! [input] canonicalize labels
```

A.3.23 intb_init4c

This logical function sets up the blocking integral API based on the input of a group of shell quartets (a block).

Return Values:

.true.	blocking API initialization okay
.false.	blocking API detected a problem

Side Effects: The shell quartet information may be reordered since this routine pulls out blocks that consist of only *s*, *p*, and *sp* functions. Thes blocks are computed using the *sp* rotated axis code since it is faster than even the Texas integral code.

The following code excerpt describes the proper use of the blocking API routines.
Pseudo Code:

```
* begin atom/shell loops

      call collect_group_of_shells()
      okay = intb_init4c($\cdots$) ! with group of shells info
      if (.not.okay) stop ' error setting up blocking interface '
00001 continue
      more = intb_2e4c($\cdots$) ! with group of shells info
      call use_integrals_in_some_way()
      if (more) goto 00001

* end atom/shell loops
```

Syntax:

```
      logical function intb_init4c(brain, icl, jcl, ketin, kcl, lcl,
     &    num_q, q4, use_q4, lscr, scr, l_erilab, block_eff)

      integer brain      ! [input] basis set handle for bra basis
      integer ketin      ! [input] basis set handle for ket basis
      integer num_q      ! [input] number of quartets
      integer icl(num_q) ! [input] i-contraction labels for quartets
      integer jcl(num_q) ! [input] j-contraction labels for quartets
      integer kcl(num_q) ! [input] k-contraction labels for quartets
      integer lcl(num_q) ! [input] l-contraction labels for quartets
      double precision q4(num_q) ! [input] scaling factors
      logical use_q4            ! [input] true if scaling
      integer l_erilab   ! [input] size of eri and label arrays that
*................................ will be used in intb_2e4c.F
      integer lscr       ! [input] length of scratch array
      double precision scr(lscr)  ! [input] scratch array
      double precision block_eff  ! [output] blocking efficiency
```

A.3.24 intb_2e4c

This logical function returns the integrals and labels based on the input of a group of shell quartets (a block). This function cannot be called without a call to intb_init4c using the same block information. "zero" integrals are removed.

$$(\mu\rho|\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\rho(X_\rho, r_1) \frac{1}{r_{12}} g_\nu(X_\nu, r_2 g_\lambda(X_\lambda, r_2) dr_1 dr_2$$

Return Values:

.true.	more integrals from this block another call to intb_2e4c required
.false.	all integrals from this block computed

The following code excerpt describes the proper use of the blocking API routines.

Pseudo Code:

```
* begin atom/shell loops

      call collect_group_of_shells()
      okay = intb_init4c($\cdots$) ! with group of shells info
      if (.not.okay) stop ' error setting up blocking interface '
00001 continue
      more = intb_2e4c($\cdots$) ! with group of shells info
      call use_integrals_in_some_way()
      if (more) goto 00001

* end atom/shell loops
```

Syntax:

```
      logical function intb_2e4c(brain, icl, jcl, ketin, kcl, lcl,
     $     num_q, q4, use_q4, zerotol, canonicalize,
     $     ilab, jlab, klab, llab, eri,
     $     l_erilab, nints, lscr, scr)

      integer brain             ! [input]  basis handle for bra
      integer ketin             ! [input]  basis handle for ket
      integer num_q             ! [input]  number of quartets input
      integer icl(num_q)        ! [input]  i-contraction quartet labels
      integer jcl(num_q)        ! [input]  j-contraction quartet labels
      integer kcl(num_q)        ! [input]  k-contraction quartet labels
      integer lcl(num_q)        ! [input]  l-contraction quartet labels
      logical use_q4            ! [input]  logical for use of q4
      double precision q4(num_q) ! [input] symmetry prefactors for ints
*
      integer l_erilab          ! [input]  length of eri and label arrays
      integer ilab(l_erilab)    ! [output] integral labels for ``i''
      integer jlab(l_erilab)    ! [output] integral labels for ``j''
      integer klab(l_erilab)    ! [output] integral labels for ``k''
      integer llab(l_erilab)    ! [output] integral labels for ``l''
      integer nints              ! [output] number of integrals returned
      double precision eri(l_erilab) ! [output] integral values
      integer lscr              ! [input] length of scratch array
      double precision scr(lscr) ! [input] scratch array
      double precision zerotol  ! [input] zero integral threshold
      logical canonicalize      ! [input] Canonicalize integral labels?
```

A.3.25 intb_nw_2e4c

This logical function returns the integrals and labels based on the input of a group of shell quartets (a block). This interfaces to the NWChem McMurchie-Davidson code. This routine should *never* be called directly by an application module!! "zero" integrals are removed.

$$(\mu\rho|\nu\lambda) = \int_{-\infty}^{\infty} g_\mu(X_\mu, r_1) g_\rho(X_\rho, r_1) \frac{1}{r_{12}} g_\nu(X_\nu, r_2 g_\lambda(X_\lambda, r_2) dr_1 dr_2$$

Return Values:

.true.	more integrals from this block another call to intb_2e4c required
.false.	all integrals from this block computed

Syntax:

```
  logical function intb_nw_2e4c(brain, icl, jcl, ketin, kcl, lcl,
 $     num_q, q4, use_q4, zerotol, canonicalize,
 $     ilab, jlab, klab, llab, eri,
 $     l_erilab, nints, lscr, scr)

  integer brain             ! [input]  basis set handle for bra
  integer ketin             ! [input]  basis set handle for ket
  integer num_q             ! [input]  number of quartets input
  integer icl(num_q)        ! [input]  i-contraction quartet labels
  integer jcl(num_q)        ! [input]  j-contraction quartet labels
  integer kcl(num_q)        ! [input]  k-contraction quartet labels
  integer lcl(num_q)        ! [input]  l-contraction quartet labels
  logical use_q4            ! [input]  logical for use of q4
  double precision q4(num_q) ! [input] symmetry prefactors for ints
*
  integer l_erilab          ! [input]  length of eri and label arrays
  integer ilab(l_erilab)    ! [output] integral labels for ``i''
  integer jlab(l_erilab)    ! [output] integral labels for ``j''
  integer klab(l_erilab)    ! [output] integral labels for ``k''
  integer llab(l_erilab)    ! [output] integral labels for ``l''
  integer nints              ! [output] number of integrals returned
  double precision eri(l_erilab) ! [output] integral values
  integer lscr              ! [input] length of scratch array
  double precision scr(lscr) ! [input] scratch array
  double precision zerotol  ! [input] zero integral threshold
  logical canonicalize      ! [input] Canonicalize integral labels?
```

A.4 INT-API: Property Integral Routines

These routines compute and return property integrals based on shell pairs.

A.4.1 int_mpole

This routine returns multipole integrals up to the level $lmax$

The general form is $< shell|pole|shell >$. Integrals are returned in shell blocks of $< ish|L|jsh >$ L=0 to $lmax$ one block for each L value.

For example ish = p and $L = 1$ and jsh = p you would get (3*1*3)+(3*3*3)=36 integrals. The order would be:

$$
\begin{array}{cccccccc}
< x|x > & < x|y > & < x|z > & < y|x > & < y|y > & \ldots & < z|z > & \text{(first nine)} \\
< x|x|x > & < x|x|y > & < x|x|z > & < x|y|x > & < x|y|y > & \ldots & < x|z|z > & \text{(second nine)} \\
< y|x|x > & < y|x|y > & < y|x|z > & < y|y|x > & < y|y|y > & \ldots & < y|z|z > & \text{(third nine)} \\
< z|x|x > & < z|x|y > & < z|x|z > & < z|y|x > & < z|y|y > & \ldots & < z|z|z > & \text{(fourth nine)}
\end{array}
$$

The integral for each L value computed is:

$$(\mu\hat{L}\lambda) = \int_{-\infty}^{\infty} g_{\mu}(X, r_1)\hat{L}g_{\lambda}(X, r_1)dr_1$$

See the int_order code inside .../NWints/int for specific order of a set of shells and dipole order.

Syntax:

```
      subroutine int_mpole(i_basis, ish, j_basis, jsh, lmax, centerl,
     &    lscr, scr, lmpint, MP)

      integer i_basis             ! [input] basis set handle for ish
      integer ish                 ! [input] i shell/contraction
      integer j_basis             ! [input] basis set handle for jsh
      integer jsh                 ! [input] j shell/contraction
      integer lmax                ! [input] maximum lvalue for
c . . . . . . . . . . . . . . . . !   multipole integrals in this batch
      double precision centerl(3) ! [input] coordinates of multipole
      integer lscr                ! [input] length of scratch array
      double precision scr(lscr)  ! [input] scratch array
      integer lmpint              ! [input] length of multipole ints
      double precision MP(lmpint) ! [output] multipole integrals
```

A.4.2 int_projpole

This routine computes projected multipole integrals up to level lmax (0→lmax):
The general form is <pole|shell>

Integrals are returned in <pole|shell> blocks one block for each L value 0→lmax.

For example, a multipole, L = 1 and a d shell would yield (1+3)*6 = 24 integrals.

The order would be:

$<0\|xx>$	$<0\|xy>$	$<0\|xz>$	$<0\|yy>$	$\ldots$	$<0\|zz>$	first six
$<x\|xx>$	$<x\|xy>$	$<x\|xz>$	$<x\|yy>$	$\ldots$	$<x\|zz>$	second six
$<y\|xx>$	$<y\|xy>$	$<y\|xz>$	$<y\|yy>$	$\ldots$	$<y\|zz>$	third six
$<z\|xx>$	$<z\|xy>$	$<z\|xz>$	$<z\|yy>$	$\ldots$	$<z\|zz>$	fourth six

Syntax:

```
      subroutine int_projpole(i_basis, ish, centerl, lmax,
     &    lscr, scr, lmpint, MP)

      integer i_basis            ! [input] basis set handle for ish
      integer ish                ! [input] i shell/contraction
      integer lmax               ! [input] maximum lvalue for
c . . . . . . . . . . . . . . . . . !     multipole ints in this batch
      double precision centerl(3) ! [input] coordinates of multipole
      integer lscr               ! [input] length of scratch array
      double precision scr(lscr)  ! [input] scratch array
      integer lmpint             ! [input/output] length of
c . . . . . . . . . . . . . . . . . !      multipole integrals array
      double precision MP(lmpint) ! [output] multipole integrals
```

A.5 INT-API: Miscellaneous Routines

These routines do a variety of functions mostly internal to the INT-API but may require use by those integrating a new base integral code into NWChem. These should be used with care and rarely in any application module.

A.5.1 exact_mem

This routine computes the memory required by the McMurchie-Davidson integral code developed at PNNL. This calls specific routines listed below for each integral type. The data is stored in a common block (apiP.fh) for fast retrieval. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine exact_mem(rtdb,bases,nbas)

integer rtdb        ! [input] the RTDB handle
integer nbas        ! [input] number of basis sets
integer bases(nbas) ! [input] basis set handles
```

Debugging Note: using a set directive to set the variable "int:full_mem" to true will force the more expensive $O(N^4)$ algorithm to be used to compute the memory requirements for the 2-electron integrals.

A.5.2 emem_3ov

This routine computes the memory for the 3-center overlap integrals based on the basis sets used. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_3ov(ibasin,jbasin,kbasin,lsz_buf,memsize)

integer ibasin  ! [input] basis set handle for ``i'' contractions
integer jbasin  ! [input] basis set handle for ``j'' contractions
integer kbasin  ! [input] basis set handle for ``k'' contractions
integer lsz_buf ! [output] maximum size of integral buffer
integer memsize ! [output] memory needed for scratch array
```

A.5.3 emem_1e

This routine computes the memory for any type of 1e integrals based on the basis sets used. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_1e(brain, ketin, max1e, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.4 emem_1e_dk

This routine computes the memory for the requested type of Douglas-Kroll integrals for the given fitting basis sets. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_1e_dk(brain, ketin, max1e, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.5 emem_1e_rel

This routine computes the memory for any type of relativistic 1e integrals based on the basis sets used. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_1e_rel(brain, ketin, max1e, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.6 emem_2e4c

This routine computes the memory required by the McMurchie-Davidson algorithm for 4 center two electron integrals based on the basis sets used.

The exact algorithm is an N^4 and an N^2 approximate algorithm is used here. Exact memory is computed for integrals over the following classes:

- Coulumb $(ii|jj)$
- Exchange $(ij|ij)$
- Triplet $(ii|ij)$

- Triplet $(ij|jj)$

An additional 10% is added to the maximum exact memory computation for each of these classes. Additional classes that have been periodically problematic are:

- $(ij|jk)$
- $(ij|kk)$

This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_2e4c(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.7 emem_2e3c

This routine computes the memory required for the the two electron three center integrals using the McMurchie-Davidson algorithm. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_2e3c(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.8 emem_2e2c

This routine computes the memory required for the the two electron two center integrals using the McMurchie-Davidson algorithm. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_2e2c(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.9 emem_2e4c_full

This routine computes the memory required by the McMurchie-Davidson algorithm for 4 center two electron integrals based on the basis sets used.

The exact algorithm is an N^4 and costly. This routine is used primarily as a debugging tool This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_2e4c_full(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.10 int_nbf_max

This routine computes the maximum cartesian nbf for a given basis set. Used in many memory computing routines to deterine maximum buffer sizes needed for integral computations. This also includes any general contraction information. This routine should never be called directly by a NWChem application module. *Syntax:*

```
subroutine int_nbf_max(basisin,nbf_max)

integer basisin  ! [input] basis set handle
integer nbf_max  ! [output] maximum number of basis functions
```

A.5.11 int_mem_zero

This routine zeros the memory pointers in the apiP.fh common that pertain to the memory utilization of the integral suite. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine int_mem_zero()
```

There are no formal arguments to this routine

A.5.12 api_is_ecp_basis

This routine identifies basis set handles to INT-API that store ECP information. This routine should never be called directly by a NWChem application module.

Syntax:

```
logical function api_is_ecp_basis(basisin)

integer basisin ! [input] basis set handle
```

Return value is true if "basisin" represents an ECP

A.5.13 emem_1e_pvp

This routine computes the memory for the p.Vp type 1e integrals based on the basis sets used. These integrals are in essence double derivative potential energy integrals of which only the dot product (diagonal) integrals are included. This routine should never be called directly by a NWChem application module.

Syntax:

```
subroutine emem_1e_pvp(brain, ketin, max1e, memsize, rel_typ)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer rel_typ ! [input] type of integrals to be computed
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.14 exactd_mem

This routine computes the memory required by the McMurchie-Davidson integral derivative code developed at PNNL. This calls specific routines listed below for each integral derivative type. The data is stored in a common block (apiP.fh) for fast retrieval from the int_mem routines.

Syntax:

```
subroutine exactd_mem(rtdb,bases,nbas)

integer rtdb        ! [input] the RTDB handle
integer nbas        ! [input] number of basis sets
integer bases(nbas) ! [input} array of basis set handles
```

Debugging Note: using a set directive to set the variable "int:full_mem" to true will force the more expensive $O(N^4)$ algorithm to be used to compute the memory requirements for the 2-electron integral derivatives.

A.5.15 emem_d1e

This routine determines the maximum buffer and scratch size for the one electron derivative integrals. This routine should not be called by application code. *Syntax:*

```
subroutine emem_d1e(brain, ketin, max1e, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.16 emem_d1e_rel

This routine determines the maximum buffer and scratch size for the one electron derivative relativistic integrals. This routine should not be called by application code. *Syntax:*

```
subroutine emem_d1e_rel(brain, ketin, max1e, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer max1e   ! [output] max size of 1e integrals buffer
integer memsize ! [output] max size of scratch space for 1e integral evaluation
```

A.5.17 emem_d2e4c

This routine determines the maximum buffer and scratch size for the given basis sets to compute 2 electron derivative integrals. The logic used is similar to that of the integral routine. This routine should not be called by application code. *Syntax:*

```
subroutine emem_d2e4c(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.18 emem_d2e4c_full

This routine determines the maximum buffer and scratch size for the given basis sets to compute 2 electron derivative integrals. The logic used is similar to that of the integral routine. This routine should not be called by application code. this routine computes the complete memory requirements using an $O(N^4)$ algorithm *Syntax:*

```
subroutine emem_d2e4c_full(brain, ketin, maxg, memsize)

integer brain   ! [input] bra basis set handle
integer ketin   ! [input] ket basis set handle
integer maxg    ! [output] max size of 2e integrals buffer
integer memsize ! [output] max size of scratch space for 2e integral evaluation
```

A.5.19 int_canon

This routine canonicalizes integral lables such that: $i \geq j$, $k \geq l$, and $ij \geq kl$

Syntax:

```
subroutine int_canon(i, j, k, l, ii, jj, kk, ll)

integer i,j,k,l      ! [input] labels
integer ii,jj,kk,ll  ! [output] canonical labels
```

A.5.20 int_chk_init

This function checks to see that the integral API is initialized. Returns .true. if initialized and .false. if not.

Syntax:

```
logical function int_chk_init(msg)

character*(*) msg ! [input] usually indentfy calling routine
```

A.5.21 int_chk_sh

This function checks to see that the given shell is valid Returns .true. if so else returns .false. if not. This subroutine call can be replaced by a statement function sequence:

```
#include "basP.fh"
#include "geobasmapP.fh"
      logical inline_chk_sh
      inline_chk_sh(bra,ish) =
     1     ((ish.gt.0) .and. (ish.le.ncont_tot_gb(bra)))
```

Where bra is the lexical basis index (not the handle). or you could use the following with the handle.

```
      inline_chk_sh(bra,ish) =
     1     ((ish.gt.0) .and.
     2(ish.le.ncont_tot_gb(bra+Basis_Handle_Offset)))
```

Syntax:

```
      logical function int_chk_sh(basisin,shell)

      integer basisin ! [input] basis set handle
      integer shell ! [input] lexical shell index
```

A.5.22 int_nospherical_check

This routine stubs out routines that are not ready for spherical basis functions by forcing an error condition.

Syntax:

```
      subroutine int_nospherical_check(basisin,tine)

      integer basisin     ! [input] basis set handle
      character*(*) tine  ! [input] routine identifier
```

A.5.23 int_nogencont_check

This routine stubs out routines that are not ready for general contraction basis functions by forcing an error condition.

Syntax:

```
subroutine int_nogencont_check(basisin,tine)

integer basisin     ! [input] basis set handle
character*(*) tine  ! [input] routine identifier
```

A.5.24 int_nospshell_check

This routine stubs out routines that are not ready for sp shells type basis functions by forcing an error condition.

Syntax:

```
subroutine int_nospshell_check(basisin,tine)

integer basisin     ! [input] basis set handle
character*(*) tine  ! [input] routine identifier
```

A.5.25 int_bothsp_gc_check

This routine checks to see if the basis sets used have both sp shells/type basis functions and general contractions. The 2e integral codes cannot handle this.

Syntax:

```
subroutine int_bothsp_gc_check(basesin,nbas,tine)

integer nbas           ! [input] number of basis sets
integer basesin(nbas)  ! [input] basis set handles
character*(*) tine     ! [input] routine identifier
```

A.5.26 int_hf1sp

This is a layer routine that calls the McMurchie-Davidson one electron routine. This layer routine handles all permutations to compute sp integrals. This routine should never be called by an application module.

Syntax:

```
      subroutine int_hf1sp(
     &       xyzi,expi,coefi, i_nprim, i_ngen, Li, ictri,
     &       xyzj,expj,coefj, j_nprim, j_ngen, Lj, ictrj,
     &       xyz,zan,exinv,nat,S,T,V,lstv,doS,doT,doV,canAB,
     &       dryrun,scr,lscr,msg)
```

For an integral $< i|Operator|j >$

```
integer i_nprim  ! [input] num. prims on function i
integer i_ngen   ! [input] num general conts on func. i
integer Li       ! [input] angular momentum of func. i
integer ictri    ! [input] lexical atom index for function i
integer j_nprim  ! [input] num. prims on function j
integer j_ngen   ! [input] num general conts on func. j
integer Lj       ! [input] angular momentum of func. j
integer ictrj    ! [input] lexical atom index for function j
integer nat      ! [input] number of atoms
integer lscr     ! [input] size of scratch array
integer lstv     ! [input] size of any integral buffer
double precision xyzi(3)  ! [input] position of center i
double precision expi(i_nprim) ! [input] exponents on i
double precision coefi(i_nprim,i_ngen) ! [input] i coeffs
double precision xyzj(3)  ! [input] position of center j
double precision expj(j_nprim)  ! [input] exponents on j
double precision coefj(j_nprim,j_ngen)  ! [input] j coeffs
double precision xyz(3,nat)  ! [input] all atom positions
double precision zan(nat)  ! [input] charges on all atoms
double precision exinv(nat)   ! [input] inverse nuclear exponents
double precision scr(lscr)  ! [scratch] scratch buffers
double precision S(lstv)  ! [output] overlap integrals
double precision T(lstv)  ! [output] kinetic energy integrals
double precision V(lstv)  ! [output] potential integrals
logical doS     ! [input] compute overlap (True/False)
logical doT     ! [input] compute kinetic (True/False)
logical doV     ! [input] compute potential (True/False)
logical canAB   ! [input] compute only canonical ints (false only)
logical dryrun  ! [input] true means only compute required memory
character*(*) msg ![input] calling func. identification message
```

A.5.27 int_hf1sp_ecp

This is a layer routine that calls the McMurchie-Davidson one electron routine. This layer routine handles all options for computing ecp integrals. This routine should never be called by an application module.

Syntax:

```
subroutine int_hf1sp_ecp(
```

```
     &       xyzi,expi,coefi, i_nprim, i_ngen, Li, ictri,
     &       xyzj,expj,coefj, j_nprim, j_ngen, Lj, ictrj,
     &       xyz,zan,exinv,nat,S,T,V,lstv,doS,doT,doV,canAB,
     &       dryrun,scr,lscr,msg)
```

For an integral $< i|Operator|j >$

```
      integer i_nprim  ! [input] num. prims on function i
      integer i_ngen   ! [input] num general conts on func. i
      integer Li       ! [input] angular momentum of func. i
      integer ictri    ! [input] lexical atom index for function i
      integer j_nprim  ! [input] num. prims on function j
      integer j_ngen   ! [input] num general conts on func. j
      integer Lj       ! [input] angular momentum of func. j
      integer ictrj    ! [input] lexical atom index for function j
      integer nat      ! [input] number of atoms
      integer lscr     ! [input] size of scratch array
      integer lstv     ! [input] size of any integral buffer
      double precision xyzi(3)  ! [input] position of center i
      double precision expi(i_nprim) ! [input] exponents on i
      double precision coefi(i_nprim,i_ngen) ! [input] i coeffs
      double precision xyzj(3)  ! [input] position of center j
      double precision expj(j_nprim)  ! [input] exponents on j
      double precision coefj(j_nprim,j_ngen)  ! [input] j coeffs
      double precision xyz(3,nat)  ! [input] all atom positions
      double precision zan(nat)  ! [input] charges on all atoms
      double precision exinv(nat)   ! [input] inverse nuclear exponents
      double precision scr(lscr)  ! [scratch] scratch buffers
      double precision S(lstv)  ! [output] overlap integrals
      double precision T(lstv)  ! [output] kinetic energy integrals
      double precision V(lstv)  ! [output] potential integrals
      logical doS     ! [input] compute overlap (True/False)
      logical doT     ! [input] compute kinetic (True/False)
      logical doV     ! [input] compute potential (True/False)
      logical canAB   ! [input] compute only canonical ints (false only)
      logical dryrun  ! [input] true means only compute required memory
      character*(*) msg ![input] calling func. identification message
```

A.5.28 int_1psp

This routine transforms integrals from the way they were computed $(p|s)$, $(p|p)$ to $(p|sp)$.

The transformation is done in place as follows:

	computed order	transformed order	
1	(x|s)	(x|s)	
2	(y|s)	(x|x)	$2 \rightarrow 5$
3	(z|s)	(x|y)	$3 \rightarrow 9$
4	(x|x)	(x|z)	$4 \rightarrow 2$
5	(x|y)	(y|s)	$5 \rightarrow 3$
6	(x|z)	(y|x)	$6 \rightarrow 4$
7	(y|x)	(y|y)	$7 \rightarrow 6$
8	(y|y)	(y|z)	$8 \rightarrow 7$
9	(y|z)	(z|s)	$9 \rightarrow 8$
10	(z|x)	(z|x)	
11	(z|y)	(z|y)	
12	(z|z)	(z|z)	

Syntax:

```
      subroutine int_1psp(block,num_blocks)
```

```
      integer num_blocks  ! [input] num. blocks to transform
      double precision block(12,num_blocks) ! [input/output]
c. . . . . . . . . . . . . . . . . . . . . . .! integral block
```

A.5.29 int_1dsp

This routine transforms integrals from the way they were computed $(d|s)$, $(d|p)$ to $(d|sp)$.

The transformation is done in place as follows:

	computed order	transformed order	
1	(xx|s)	(xx|s)	
2	(xy|s)	(xx|x)	$2 \rightarrow 5$
3	(xz|s)	(xx|y)	$3 \rightarrow 9$
4	(yy|s)	(xx|z)	$4 \rightarrow 13$
5	(yz|s)	(xy|s)	$5 \rightarrow 17$
6	(zz|s)	(xy|x)	$6 \rightarrow 21$
7	(xx|x)	(xy|y)	$7 \rightarrow 2$
8	(xx|y)	(xy|z)	$8 \rightarrow 3$
9	(xx|z)	(xz|s)	$9 \rightarrow 4$
10	(xy|x)	(xz|x)	$10 \rightarrow 6$
11	(xy|y)	(xz|y)	$11 \rightarrow 7$
12	(xy|z)	(xz|z)	$12 \rightarrow 8$
13	(xz|x)	(yy|s)	$13 \rightarrow 10$
14	(xz|y)	(yy|x)	$14 \rightarrow 11$
15	(xz|z)	(yy|y)	$15 \rightarrow 12$
16	(yy|x)	(yy|z)	$16 \rightarrow 14$
17	(yy|y)	(yz|s)	$17 \rightarrow 15$
18	(yy|z)	(yz|x)	$18 \rightarrow 16$
19	(yz|x)	(yz|y)	$19 \rightarrow 18$
20	(yz|y)	(yz|z)	$20 \rightarrow 19$
21	(yz|z)	(zz|s)	$21 \rightarrow 20$
22	(zz|x)	(zz|x)	
23	(zz|y)	(zz|y)	
24	(zz|z)	(zz|z)	

Syntax:

```
      subroutine int_1dsp(block,num_blocks)
```

```
      integer num_blocks  ! [input] num. blocks to transform
      double precision block(24,num_blocks) ! [input/output]
c . . . . . . . . . . . . . . . . . . . . . ! integral block
```

A.5.30 int_1spsp

This routine transforms integrals from the way they were computed $(s|s)$, $(s|p)$, $(p|s)$, $(p|p)$ to $(sp|sp)$. The transformation is done in place as follows:

	computed order	transformed order	
1	(s|s)	(s|s)	
2	(s|x)	(s|x)	
3	(s|y)	(s|y)	
4	(s|z)	(s|z)	
5	(x|s)	(x|s)	
6	(y|s)	(x|x)	$6 \rightarrow 9$
7	(z|s)	(x|y)	$7 \rightarrow 13$
8	(x|x)	(x|z)	$8 \rightarrow 6$
9	(x|y)	(y|s)	$9 \rightarrow 7$
10	(x|z)	(y|x)	$10 \rightarrow 8$
11	(y|x)	(y|y)	$11 \rightarrow 10$
12	(y|y)	(y|z)	$12 \rightarrow 11$
13	(y|z)	(z|s)	$13 \rightarrow 12$
14	(z|x)	(z|x)	
15	(z|y)	(z|y)	
16	(z|z)	(z|z)	

Syntax:

```
      subroutine int_1spsp(block,num_blocks)

      integer num_blocks        ! [input] num. blocks to transform
      double precision block(16,num_blocks) ! [input/output]
c . . . . . . . . . . . . . . . . . . . . . . . ! integral block
```

A.5.31 int_1spa

This routine transforms integrals from the way they were computed $(s|X)$, $(p|X)$, to $(sp|X)$. The transformation is **NOT** done in place: *Syntax:*

```
      subroutine int_sp1a(sp_block,s_block,p_block,sizeb,num_blocks)

      integer sizeb  ! [input] size of non sp block
      integer num_blocks ! [input] num of blocks to transform
*
* . . . . . . . . . . . . . . . . .! [output] (sp|X) transformed integral block
      double precision sp_block(sizeb,1:4,num_blocks)
```

```
*
* . . . . . . . . . . . . . . . . . . . . . .! [input] computed (s|X) block
      double precision s_block(sizeb,num_blocks)
*
* . . . . . . . . . . . . . . . . . . . . . .! [input] computed (p|X) block
      double precision p_block(sizeb,2:4,num_blocks)
```

A.5.32 int_sp1b

This routine transforms integrals from the way they were computed $(X|s)$, $(X|p)$, to $(X|sp)$. The transformation is **NOT** done in place: *Syntax:*

```
      subroutine int_sp1b(sp_block,s_block,p_block,sizea,num_blocks)

      integer sizea ! [input] size of non sp block
      integer num_blocks ! [input] num of blocks to transform
*
* . . . . . . . . . . . . . . .! [output] (X|sp) transformed integral block
      double precision sp_block(1:4,sizea,num_blocks)
*
* . . . . . . . . . . . . . . . . . . . . . .! [input] computed (X|s) block
      double precision s_block(sizea,num_blocks)
*
* . . . . . . . . . . . . . . . . . . . . . .! [input] computed (X|p) block
      double precision p_block(2:4,sizea,num_blocks)
```

A.5.33 int_nint

This routine computes the number of integrals for a given shell/contraction grouping; if an input shell is zero then the routine ignores this shell. This routine will work for both cartesian and spherical basis sets. This routine should never be called by an NWChem application module.

Syntax:

```
      integer function int_nint(ibasin,icnt,jbasin,jcnt,
     &       kbasin,kcnt,lbasin,lcnt)

      integer ibasin   ! [input] basis set handle for icnt
      integer icnt     ! [input] contraction index (e.g., ish)
      integer jbasin   ! [input] basis set handle for jcnt
```

```
integer jcnt     ! [input] contraction index (e.g., jsh)
integer kbasin   ! [input] basis set handle for kcnt
integer kcnt     ! [input] contraction index (e.g., ksh)
integer lbasin   ! [input] basis set handle for lcnt
integer lcnt     ! [input] contraction index (e.g., lsh)
```

A.5.34 int_unint

This routine computes the number of integrals for a given shell/contraction grouping; if an input shell is zero then the routine ignores this shell. The input shell must be a unique shell in the sense of the basis set API. This routine will work for both cartesian and spherical basis sets. This routine should never be called by an NWChem application module.

Syntax:

```
      integer function int_unint(ibasin,icnt,jbasin,jcnt,
     &       kbasin,kcnt,lbasin,lcnt)

      integer ibasin   ! [input] basis set handle for icnt
      integer icnt     ! [input] unique contraction index (e.g., ish)
      integer jbasin   ! [input] basis set handle for jcnt
      integer jcnt     ! [input] unique contraction index (e.g., jsh)
      integer kbasin   ! [input] basis set handle for kcnt
      integer kcnt     ! [input] unique contraction index (e.g., ksh)
      integer lbasin   ! [input] basis set handle for lcnt
      integer lcnt     ! [input] unique contraction index (e.g., lsh)
```

A.5.35 int_nint_cart

This routine computes the number of integrals for a given shell/contraction grouping; if an input shell is zero then the routine ignores this shell. This routine will work for both cartesian and spherical basis sets, but *returns the cartesian size* (this is how the integrals are computed!). This routine should never be called by an NWChem application module.

Syntax:

```
      integer function int_nint_cart(ibasin,icnt,jbasin,jcnt,
     &       kbasin,kcnt,lbasin,lcnt)

      integer ibasin   ! [input] basis set handle for icnt
      integer icnt     ! [input] contraction index (e.g., ish)
```

```
integer jbasin   ! [input] basis set handle for jcnt
integer jcnt     ! [input] contraction index (e.g., jsh)
integer kbasin   ! [input] basis set handle for kcnt
integer kcnt     ! [input] contraction index (e.g., ksh)
integer lbasin   ! [input] basis set handle for lcnt
integer lcnt     ! [input] contraction index (e.g., lsh)
```

A.5.36 int_unint_cart

This routine computes the number of integrals for a given shell/contraction grouping; if an input shell is zero then the routine ignores this shell. The input shell must be a unique shell in the sense of the basis set API. This routine will work for both cartesian and spherical basis sets, but *returns the cartesian size* (this is how the integrals are computed!). This routine should never be called by an NWChem application module.

Syntax:

```
 integer function int_unint_cart(ibasin,icnt,jbasin,jcnt,
&       kbasin,kcnt,lbasin,lcnt)

 integer ibasin   ! [input] basis set handle for icnt
 integer icnt     ! [input] unique contraction index (e.g., ish)
 integer jbasin   ! [input] basis set handle for jcnt
 integer jcnt     ! [input] unique contraction index (e.g., jsh)
 integer kbasin   ! [input] basis set handle for kcnt
 integer kcnt     ! [input] unique contraction index (e.g., ksh)
 integer lbasin   ! [input] basis set handle for lcnt
 integer lcnt     ! [input] unique contraction index (e.g., lsh)
```

Appendix B

Performance Statistics Collection — PSTAT

The pstat library is intended to facilitate collecting and reporting performance statistics for parallel programs. The design is based to some extent on the ptimer and pmon facilities provided by Kendall Square Research, and also by getstat in the COLUMBUS program system.

B.1 Model

Applications can allocate "timers" associated with events in the program. "Timers" are actually generalized data structures which can record elapsed CPU and wall clock time, accumulate information (i.e. the number of integrals produced) and other (possibly system-dependent) data. (In the present implementation only times and accumulators are available.) Timers are represented within the program by opaque handles.

B.2 API

B.2.1 Include files

All routines using the pstat library should include `pstat.fh`, which includes predefined constants for the various statistics that can be collected.

B.2.2 pstat_init

```
Status = PStat_Init( Max_Timers, NAcc, Names )
Logical Status
Integer Max_Timers, NAcc [IN]
Character*(*) Names(NAcc) [IN]
```

Initialize package, reserving space for Max_Timers different timers. Also defines NAcc user-defined accumulation registers labeled by the given Names.

B.2.3 pstat_terminate

```
Status = PStat_Terminate()
Logical Status
```

Free up all temporary space used by pstat package

B.2.4 pstat_allocate

```
Status = PStat_Allocate( Name, Functions, NAcc, Accumulators, Handle )
Logical Status [OUT]
Character*(*) Name [IN]
Integer Functions [IN], NAcc [IN], Accumulators(NAcc) [IN], Handle [OUT]
```

Create a timer with the given descriptive name which records the statistics described by the Functions argument. This timer will also allow accumulation into the NAcc accumulation registers listed in the Accumulators array.

B.2.5 pstat_free

```
Status = PStat_Free( Handle )
Logical Status [OUT]
Integer Handle [IN]
```

Frees up a timer so it can be re-pstat_allocated later. Does not free the storage associated with the timer.

B.2.6 pstat_on

```
PStat_On( Handle )
Integer Handle [IN]
```

Start statistics gathering for the timer Handle. Routine aborts with an error if timer is not in the "off" state at invocation. Aborts with an error if Handle is not assigned.

B.2.7 pstat_off

```
PStat_Off( Handle )
Integer Handle [IN]
```

End statistics gathering for the timer Handle. Routine aborts with an error if timer is not in the "on" state at invocation. Aborts with an error if Handle is not assigned.

B.2.8 pstat_acc

```
PStat_Acc( Handle, N, Data)
Integer Handle, N [IN]
Double precision Data(N) [IN]
```

Accumulate Data into the registers defined when Handle was allocated. N must match the number of accumulation registers specified in the declaration of the timer, and the elements of Data will be added to the registers as specified in the Accumulators array used then the timer was allocated.

B.2.9 pstat_print_all and pstat_print

```
PStat_Print_All
PStat_Print( Functions, NAcc, Accumulators )
Integer Functions, NAcc, Accumulators(NAcc) [IN]
```

Write a summary of statistics to stdout. PStat_Print_All reports all data which has been collected. PStat_Print reports those data specified in Functions and Accumulators. The report includes the number of calls to each timer and the data specified by Functions. For all data, including the number of calls, the min, max, and average across all processes is reported.

B.2.10 Usage Notes

In normal usage, an application module would allocate the appropriate timers, normally in a subroutine, and store the handles in a common block which is included by all routines in the module which use PStat. This is separate from the `pstat.fh` include file. And of course another subroutine at the end of the module would normally be used to free the timers.

The core routines, PStat_{On,Off,Acc}, do not return error codes in order to simplify putting them into & removing them from code easily. They abort with an error if the timer handle is invalid, or if they are called out of sequence (PStat_On and PStat_Off must be paired).

Different machines have different capabilities w.r.t. performance statistics collection. Those functions which are not available on a given implementation will be silently ignored.

In order to minimize the overhead of checking which statistics to collect in each PStat_{On,Off} call, the functions should represent related groups of statistics rather than a single item. Three predefined groups will always be available: PStat_NoStats, which is a NOP (for example when a timer will use only user-defined accumulators), PStat_AllStats, which expands to all available functions, and PStat_QStat, which is a minimal (quick) set (CPU time and wall clock time) intended for low-overhead usage. Multiple functions can be requested by adding their values together with the exception of PStat_QStat (to keep overhead low, PStat_QStat is checked first, and if true, no other functions are checked.

B.3 Closing Comment

The current version of pstat was created as a throwaway prototype, but it hasn't been thrown away quite yet. Things can certainly be improved, and hopefully they will be in due course. One of the most important design flaws is the lack of context-dependence in the timers. As an excuse, I can only offer that we *still* don't have a grip on how to handle context in general, so it is not surprising that pstat doesn't have it.

Appendix C

Integral File I/O – INT2E

C.1 Application- and I/O-Level Views of the Data Stream

The data stream coming into the package from the application consists of (floating point) integral values and (integer) labels (four labels per value) interspersed with calls which specify the ranges of the four labels for subsequent integrals.

On disk (or in the package's cache) the data appears in compressed form, in chunks of 64 KB holding up to 8192 value/label sets, some of which may contain structural information rather than integral data.

C.2 Internal Data Structures (all are node-private)

```
buffer (common block /cint2efile/)
        integer                 n_per_rec = 8192
        double precision        values(n_per_rec)
        integer                 labels(4, n_per_rec)
        integer                 n_in_rec
```

VALUES are actual floating point integral values and must be bounded in absolute value by MAXVALUE to allow for the fixed-point compression scheme.

LABELS are basis function labels relative to RANGE (set by int2e_set_bf_range). Must be representable in 8 bits to allow for compression.

Some elements of the buffer are devoted to special purposes, in which case the labels are used to store ¿ 8 bit integer values and the corresponding VALUES are set to zero (used in sanity checking). Special purposes are (1) a counter of the number of values in the current range (see int2e_buf_cntr_{pack,unpack}), and (2) specifying a new basis function range (see int2e_set_bf_range). The first element of the buffer is always a counter.

Related values:

next_value Points to next buffer element to be read/inserted

cntr_ptr Points to buffer element holding the current integral counter

nleft_in_range Running count of number of valid integrals remaining in the range. Initialized by int2e_buf_cntr_unpack and updated as the user obtains integrals with int2e_file_read.

```
compressed buffer (common block /cint2ebuf/)
        integer                 n\_per\_rec = 8192
        double precision        buf(n\_per\_rec)
        integer                 n\_in\_buf, pad
```

Note: BUF is equivalanced to the integer array IBUF. The IBUF representation is used during compression, while the BUF representation is used for storage. PAD insures that the common block has an even length in doubles regardless of the relative size of integers and doubles.

The first half of BUF contains the 32-bit integer fixed-point representation of the VALUES array. If the machine has 64-bit integers, the fixed-point data are packed two per integer. The final half of BUF contains the LABELS array compressed to 8 bits per element. (Note that the same bitstream results regardless of whether the platform uses one or two integers per double. In the case of one integer per double, the 32-bit fixed-point integral values are packed two to a word.)

C.2.1 Cache

Each node allocates local memory to act as a cache for its file. The size of the cache is determined by user input (via the RTDB). Operation is simple: the cache is filled with the initial records of integral data, the remainder go to disk. Data is never moved between cache and disk.

C.3 Subprograms

C.3.1 `sread`, `swrite` (in util directory)

Read (write) an array of doubles on a Fortran sequential access file. If more than 32767 elements (hardwired in the routines) are to be read (written), it broken into multiple records of at most 32767 elements each.

C.3.2 `int2e_file_open` (API)

Initializes integral file management variables (including filename). Determined numerical precision required to store (floating point) integral values and produces a scaling factor for the fixed-point compression scheme (values are represented as 32 bit integers relative to this scale factor). Allocates local memory for cache. Does not actually open the file (the Paragon is notorious for dying if you try to open too many files simultaneously, so actual opening is deferred until the first need to write, which is less likely to be synchronous).

C.3.3 `int2e_file_close` (API)

Closes integral files, frees cache (local memory).

C.3.4 `int2e_file_rewind` (API)

Rewinds the integral files, clears buffer.

C.3.5 `int2e_file_read` (API)

Fills user-provided arrays with integrals and four labels. Operates by repeated calls to int2e_buf_read followed by unpacking of the data into the user-supplied arrays. Unpacking involves adjusting the labels to reflect the range as set by int2e_set_bf_range. Data is read until MAXINTS (user-specified) values have been read or the end of the current range (see int2e_set_bf_range) is reached. Returning .FALSE. is a signal to call int2e_get_bf_range before the next call to this routine.

C.3.6 `int2e_file_write` (API)

Copies data into internal buffers (currently 8192 elements, defined in cint2efile.fh), writing to disk as the buffer fills. As each integral is being copied into the internal buffers, it is compared against a value which is the limit of what can be represented in the fixed-point compression scheme with the necessary precision. If the integral value exceeds this value (in absolute value) int2e_file_write_big is called deposit it into the buffer.

C.3.7 `int2e_file_write_big` (internal)

Splits up an integral too large to be represented accurately in the fixed point compression scheme into multiple smaller integrals (same labels, of course).

C.3.8 `int2e_buf_read, int2e_buf_write` (mostly internal)

There is one application-level call to int2e_buf_write in the SCF to insure that the final buffer is written to disk.

int2e_buf_read obtains a record of data from the cache (for records ¡ max_cache_rec) or from disk (via sread). The data is unpacked by int2e_buf_unpack, and the number of integrals in the current range is extracted. To write the buffer, the procedure is exactly the reverse.

C.3.9 `int2e_buf_clear` (internal)

Resets buffer pointers to "zero", effectively emptying the buffer and reserving the first entry in the buffer as a counter of the number of data values in the record (or until int2e_set_bf_range is called).

C.3.10 `int2e_buf_cntr_pack, int2e_buf_cntr_unpack` (internal)

Prepares the number of integrals counter for the data compression associated with storage. The counter occupies the cntr_ptr element in the buffer. During normal operation, the counter is maintained as an integer in labels(1, cntr_ptr), with no data in labels(2:4, cntr_ptr) or values(cntr_ptr). The counter can therefore represent up to 2^{24}. Since the data compression algorithm stores the label values as 8 bits each, the counter is "packed" (unpacked) by splitting it into three bytes and stored in labels(1:3, cntr_ptr). Zeros are stored in labels(4, cntr_ptr) and values(cntr_ptr) and used by int2e_buf_unpack as as part of a sanity check.

The first element of each record is a counter, and additional counters are generated by calls to int2e_set_bf_range.

C.3.11 `int2e_buf_pack, int2e_buf_unpack` (internal)

Compresses (decompresses) the integral buffer. Integral values are scaled to produce a 32 bit integer representation (fixed-point compression). Integral labels are packed into 32 bits as well. On machines with 64 bit integers, the compressed integrals and labels are combined into a single datum.

C.3.12 `int2e_set_bf_range, int2e_get_bf_range` (API)

Tells (extracts) the integral file module the ranges of the four integral labels to follow. The specified range is effective until it2e_set_bf_range is called again to change it.

The lowest 16 bits of the eight limit values are stored in four elements of the buffer as follows to survive the subsequent 8 bit packing:

```
high 8 bits: ilo, jlo, klo, llo --> labels(1:4, next_value    )
high 8 bits: ihi, jhi, khi, lhi --> labels(1:4, next_value + 1)
low  8 bits: ilo, jlo, klo, llo --> labels(1:4, next_value + 3)
low  8 bits: ihi, jhi, khi, lhi --> labels(1:4, next_value + 4)
```

Calling int2e_set_bf_range also terminates the current counter (see int2e_buf_cntr_{pack,unpack}) and starts a new one for the new basis function range at next_value+5.

Appendix D

NWChem Error Messages

This appendix lists the NWChem error messages, and where possible, provides some explanation of what the code is trying to tell you.