

NWCHEM Programmer's Guide, Release 4.6

High Performance Computational Chemistry Group

August 11, 2004

DISCLAIMER

This material was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the United States Department of Energy, nor Battelle, nor any of their employees, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, SOFTWARE, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

LIMITED USE

This software (including any documentation) is being made available to you for your internal use only, solely for use in performance of work directly for the U.S. Federal Government or work under contracts with the U.S. Department of Energy or other U.S. Federal Government agencies. This software is a version which has not yet been evaluated and cleared for commercialization. Adherence to this notice may be necessary for the author, Battelle Memorial Institute, to successfully assert copyright in and commercialize this software. This software is not intended for duplication or distribution to third parties without the permission of the Manager of Software Products at Pacific Northwest National Laboratory, Richland, Washington, 99352.

ACKNOWLEDGMENT

This software and its documentation were produced with Government support under Contract Number DE-AC06-76RLO-1830 awarded by the United States Department of Energy. The Government retains a paid-up non-exclusive, irrevocable worldwide license to reproduce, prepare derivative works, perform publicly and display publicly by or for the Government, including the right to distribute to other Government contractors.

AUTHOR DISCLAIMER

This software contains proprietary information of the authors, Pacific Northwest National Laboratory (PNNL), and the US Department of Energy (USDOE). The information herein shall not be disclosed to others, and shall not be reproduced whole or in part, without written permission from PNNL or USDOE. The information contained in this document is provided "AS IS" without guarantee of accuracy. Use of this software is prohibited without **written** permission from PNNL or USDOE. The authors, PNNL, and USDOE make no representations or warranties whatsoever with respect to this software, including the implied warranty of merchant-ability or fitness for a particular purpose. The user assumes all risks, including consequential loss or damage, in respect to the use of the software. In addition, PNNL and the authors shall not be obligated to correct or maintain the program, or notify the user community of modifications or updates that will be made over the course of time.

Glossary of NWChem Terms

- API – Abstract Programming Interface: a common interface that can be used by many different modules to perform the same type of task. Functions are encapsulated so that information is passed in a standard way. The same information is transferred each time the interface is accessed, and the API does not need to know which module is actually calling it.
- CASSCF – (Complete Active Space Self Consistent Field): calculation type performed by the multi-configuration SCF module
- CCSD – (Coupled Cluster Single and Double excitations; theory for obtaining properties of molecular systems
- CI – (Configuration Interaction): module for general spin-adapted configuration-driven calculations for computing wavefunctions
- CVS – (Concurrent Versions System): software used by NWChem developers at EMSL to manage software releases and maintain configuration control in a multi-developer environment.
- DFT – (Density Functional Theory): module that uses the Gaussian basis set approach for computation of closed-shell and open-shell densities and Kohn-Sham orbitals in the local density, non-local density, local spin-density, non-local spin-density approximations
- DRA – (Disk Resident Arrays): an array oriented I/O library for out-of-core computations, extending the global arrays NUMA programming model to disk.
- driver – a particular type of module that controls some process, such as optimization or dynamics calculation (e.g., modules STEPPER and DRIVER; the module nwARGOS acts as a driver when performing QM/MM calculations)
- ECCE – (Extensible Computational Chemistry Environment): available software to support planning and management of chemical calculations; provides a common interface to multiple computational chemistry codes for selection of basis sets, a browsable calculation and chemistry database, and visualization of computational results
- Global Array Library – the functions comprising the Global Array tool, which allows the program to define memory that can be distributed across nodes or shared among nodes (on shared memory machines) in a parallel environment
- idempotent – a fundamental property of projection operators that means the square of the projection operator P_a is equal to the operator itself; i.e., $P_a^2 = P_a$.
- instantiation – creation of a unique instance of a particular object, in response to user input describing a specific molecule

- interface – generic term for a program feature that provides a well-defined way for the user to communicate information to the code, for different calculational modules in the code to communicate with each other, for the molecular modeling tools to communicate information to the different calculational modules
- MA – Memory Allocator: programming tool that allows allocation of memory that is local to the calling process only, and will not be shared by other processes in a parallel environment
- MCSCF – (Multi-Configuration Self Consistent Field): module for performing complete active space SCF (CASSCF) calculations with up to 20 active orbitals and hundreds of basis functions
- module – an essentially independent program within NWChem that performs some well-defined, high-level function (e.g., SCF, nwARGOS, MP2)
- MP2 – (Møller-Plesset (or Many Body)): module for computation of Møller-Plesset perturbation theory second-order correction to the Hartree-Fock energy calculation
- MPI – (Message Passing Interface): alternative to TCGMSG for message passing
- NUMA – Non-Uniform Memory Allocation: strategy for distributing data across multiple nodes for efficient and scalable performance in a parallel computing environment
- object – an encapsulated feature containing data that is organized in a specific pattern, instantiated based on user input, and can be accessed by any module in the code (NOTE: because NWChem is written mainly in Fortran-77, this encapsulation is highly artificial and can be maintained only by conscientious adherence to the prescribed programming conventions given in Chapter 9)
- operation – the calculation performed in a given task (e.g., single point energy evaluation, calculate derivative of energy with respect to nuclear coordinates, etc.)
- patch – a region of memory in a global array
- QM/MM – (Quantum-Mechanics and Molecular-Mechanics):
- runtime database – a persistent data storage mechanism that consists of a file created at runtime to allow different modules of the code to access the same information, and to communicate with each other in an orderly and repeatable manner in both sequential and parallel environments
- RHF – (Restricted Hartree-Fock): default closed-shell wavefunction type solved by SCF module
- RI-MP2 – (Resolution of the Identity for Møller-Plesset or (Many Body)): optional algorithm for resolution of the identity approximation to MP2
- ROHF – (Restricted Open-shell Hartree-Fock): option for type of wavefunction solved by SCF module
- SCF – (Self Consistent Field): calculation module for computing closed-shell restricted Hartree-Fock (RHF) wavefunctions, restricted high-spin open-shell Hartree-Fock (ROHF) wavefunctions, and spin-unrestricted Hartree-Fock (UHF) wavefunctions
- task – a specific job the code can be directed to do. Most commonly specifies some specific electronic structure calculation using a particular level of theory, but can also specify combined quantum-mechanics and molecular-mechanics calculations, or execution of UNIX commands in the Bourne shell.
- TCGMSG – (Theoretical Chemistry Group MeSsaGe): a toolkit for writing portable parallel programs using a message passing model; supported on a variety of common UNIX workstations, mini-super and super computers and heterogenous networks of such platforms, as well as on true parallel computers
- theory – a quantum mechanical method available in NWChem for calculation of molecular electronic structure properties, including energy, gradients, dynamics, and vibrational frequencies

- UHF – (Unrestricted Hartee-Fock): option for closed-shell spin unrestricted wavefunction solved by SCF module
- utilities – routines that perform well-defined functions which are useful but not directly related to chemical computation, (such as input processing, printing of output, timing statistics, etc) that can be accessed as needed by all modules in the code

About This Guide

NWChem is a computational chemistry package designed to run on high-performance parallel supercomputers. Code capabilities include the calculation of molecular electronic energies and analytic gradients using Hartree-Fock self-consistent field (SCF) theory, Gaussian density function theory (DFT), and second-order perturbation theory. For all methods, geometry optimization is available to determine energy minima and transition states. Classical molecular dynamics capabilities provide for the simulation of macromolecules and solutions, including the computation of free energies using a variety of force fields. Quantum mechanical molecular mechanics (QM/MM) can also be used to obtain single-point energy evaluation, minimization of the energy by variation of the molecular structure, and molecular dynamics. The level of theory used must, however, support analytic gradients.

NWChem is scalable, both in its ability to treat large problems efficiently, and in its utilization of available parallel computing resources. The code uses the parallel programming tools TCGMSG and the Global Array (GA) library developed at PNNL for the High Performance Computing and Communication (HPCC) grand-challenge software program and the Environmental Molecular Sciences Laboratory (EMSL) Project.

NWChem is written in Fortran-77 and the C programming language. It is currently capable of operating on a wide variety of hardware platforms, including Cray T3D, Intel Paragon, IBM SP2, Kendall Square Research KSR-2, SGI SMP systems, SUN and other workstations, and workstation networks. The code can be ported to nearly any sequential or parallel computer. (Installation instructions and a list of supported platforms are included in Section 8.2.)

Basic system requirements include

- Fortran-77 compiler
- C compiler
- GNU "make" (version 3.71 or higher)

Dependencies on other software include (but are not limited to) the following:

- TCGMSG or MPI for message passing in a parallel computing environment
- Memory Allocator Library
- Global Arrays Library eigensolver (for use as a wrapper for the eigensolver from PEIGS library)
- BLAS library
- LAPACK library
- MPI, SCALAPACK, PBBLAS, and BLACS libraries
- LaTeX (to obtain hardcopy and new release versions of User's Manual and Programmer's Manual)

A discussion of required environmental variables, makefiles, and libraries for all current installations is given in Section 8.4. Different implementations on the various platforms may use different vendor libraries and include (include.h and include.fh) files. Refer to Section 8.2.1 for details on the requirements of each platform. This section also discusses the requirements for TCGMSG for message passing, and the option of using MPI instead of TCGMSG.

Purpose

The purpose of this document is to provide a resource for NWChem developers that describes the prescribed practices for developing software utilizing the NWChem source tree. This document also delineates the scope, utilization, and features of the base software development tools (the “NWChem umbrella”) that are required to interface with other modules and develop new modules. The primary audience for this programming guide is the NWChem developer.

Scope

The Programming Guide provides in depth information on how to modify existing modules and create new modules within the structure of NWChem. For the developer new to the system, the Guide provides a conceptual level overview and introductory material. For the experienced developer, it provides reference material for the extensive list of functions, subroutines, makefiles, and libraries used in the program.

Prerequisites

Users of this guide should have or be working toward a PhD in theoretical or computational molecular chemistry, and have some familiarity with parallel computing and chemical modeling. It would also be helpful to know at least a small amount about the NWChem software development process. Users of this guide should have a working knowledge of UNIX operating systems, Fortran-77, and the C programming language. Some familiarity with object-oriented programming and design would also be useful.

How to Use This Guide

If you are a new NWChem developer, you should read Chapter 1 of this guide to gain an overview of the system. Read Chapters 2, 4, and 6 to acquire the basic knowledge needed to begin designing and programming. Read Chapter 7 for information on basic utility functions available for use by the different modules of the code.

Once comfortable with the development toolkit and modeling tools available, you can find specific information on each module in Chapter 3. Read Chapter 8 for guidance on installing the code on your system.

Read Chapter 9 for procedures for code development and modification. If you are an experienced developer and are intimately familiar with the structure of NWChem or specific modules within the program, you may want to start with Chapter 9, and use Chapters 3, 4, 6, and 7 as reference resources.

Read Chapters 10 and 11 to obtain guidance on requirements for testing and documentation of code modifications and new modules. Regardless of your level of expertise, you should become intimately familiar with Chapter 11 and follow the guidelines for documentation of any code that you write. This will help ensure that this manual is kept up to date and remains usable for NWChem developers yet unborn.

Organization

Chapter 1: NWChem Overview

This chapter provides a high-level overview of NWChem, including a detailed discussion of the code architecture.

Chapter 2: Generic Task Interface

This chapter discusses the details of the structure of NWChem and describes the flow of control through the five-tiered structure of the code.

Chapter 3: Molecular Calculation Modules

This chapter describes in detail the individual modules that perform the chemistry calculations, such as SCF gradient, DFT energy, QM/MM dynamics.

Chapter 4: The Molecular Modeling Toolkit

This chapter describes in detail the chemistry related tools that provide the basic functionality common to many of the chemistry algorithms in NWChem.

Chapter 6: The Software Development Toolkit

This chapter describes in detail the tools that provide the interface between the chemistry calculation modules and the hardware memory and disk. These tools consist of the runtime database for disk access, plus the memory allocator, global arrays, and ChemI/O tools for non-uniform memory access (NUMA) for parallel programming.

Chapter 7: Utilities

This chapter covers Utilities, the performance statistics (PSTAT) utility, and integral file I/O, with reference to details in App. B and C.

Chapter 8: Installing NWChem

This chapter describes procedures and requirements for installing NWChem on a new system. It also contains details on makefiles and libraries, and a description of the Concurrent Version System (CVS) used for configuration management.

Chapter 9: Developing New Modules and Enhancements

This chapter describes procedures for coding new modules and modifications to existing modules in NWChem. Coding style and design requirements are described in detail. The developer is strongly advised to study this chapter thoroughly before writing a single line of code.

Chapter 10: Testing the Program

This chapter describes the requirements for testing modifications and new modules for NWChem.

Chapter 11: Documenting New Modules and Code Enhancements

This chapter describes procedures for documenting modifications and new modules in NWChem. Sections cover documentation tools and procedures for notifying end users, trainers, and documenters of new or enhanced modules. Please read this chapter thoroughly and follow the procedures so that this document will co-evolve with the code.

Appendix A. The Integral Application Programmer's Interface

This appendix contains information on the structure of the IAP interface.

Appendix B. Performance Statistics Collection – PSTAT

This appendix documents the PSTAT library, developed to facilitate collecting and reporting performance statistics for parallel algorithms.

Appendix C. Integral File I/O

This appendix describes in detail the application- and i/o-level views of the data stream for NWChem.

Appendix D. Error Messages

This appendix lists the error messages NWChem spits out from time to time.

Related Product Information

The following related documents and information sources are available for those needing further information on NWChem and the various toolkits used by the code.

1. *NWChem User Document*

2. Homepage for William R. Wiley Environmental Molecular Sciences Laboratory (EMSL) – <http://www.emsl.pnl.gov:2080/>

- under hot link "Products", sublink "Software";
 - Web page for NWChem –
<http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html>
 - Web page for Extensible Computational Chemistry Environment (ECCE) –
<http://www.emsl.pnl.gov:2080/docs/ecce/>
 - Web page for Global Arrays (GA) toolkit –
<http://www.emsl.pnl.gov:2080/docs/global/>

- Web page for TCGMSG Message Passing Library –
`http://www.emsl.pnl.gov:2080/docs/tcgmsg/tcgmsg.html`
- Web page for Dynamic Memory Allocator (MA) Library –
`http://www.emsl.pnl.gov:2080/docs/parsoft/ma/MAapi.html`
- Web page for High Performance I/O for Computational Chemistry Applications (ChemIO) –
`http://www.emsl.pnl.gov:2080/docs/parsoft/chemio/chemio.html`

Notation Conventions

Unless otherwise noted, all programs are listed in standard Fortran-77 and C language notation.

UNIX commands, variable names, and other details of Fortran or C coding are given in teletype (i.e., Courier font) text.

Characters to be entered are shown in the appropriate upper or lower case letters.

Contents

1	NWChem Overview	25
1.1	Introduction	25
1.2	NWChem Architecture	27
1.2.1	Object Oriented Design	28
1.2.2	Non-Uniform Memory Access	29
1.2.3	The Five-Tiered Modular Architecture	29
2	Generic Task Interface	31
2.1	Flow of Control in NWChem	31
2.2	Task Execution in NWChem	33
2.2.1	Task Routines for NWChem Operations	33
3	Molecular Calculation Modules	37
3.1	Theories	37
3.1.1	Self-Consistent Field Module – SCF	38
3.1.2	Gaussian Density Functional Theory Module – DFT	38
3.1.3	Møllier-Plesset Module – MP2	39
3.1.4	Resolution of the Identity Approximate Integral Method	39
3.1.5	CCSD	40
3.1.6	MCSCF	40
3.1.7	CI	40
3.1.8	Molecular Mechanics (MM)	40
3.2	Operations	40
3.2.1	Energy	41
3.2.2	Gradient	41
3.2.3	Optimization	41

3.2.4	Frequencies	41
3.2.5	Properties	41
3.2.6	Dynamics	41
4	Molecular Modeling Toolkit	43
4.1	The Geometry Object	43
4.1.1	Creating, destroying, loading and storing geometries	44
4.1.2	Information About the Geometry	46
4.1.3	Information About Centers and Coordinates	47
4.1.4	Support for Periodic Systems	48
4.1.5	Printing and Miscellaneous Routines	50
4.2	The Basis Set Object	51
4.2.1	Creating, Destroying, Loading and Storing Basis Sets	51
4.2.2	Information About the Entire Basis	52
4.2.3	Mapping Between Centers, Shells/Contractions and Functions	54
4.2.4	Printing Basis Sets	55
4.2.5	Miscellaneous Other Functions for Basis Sets	55
4.2.6	Other — unique contraction information and adding centers	57
4.3	Linear Algebra Routines	57
4.3.1	Simple linear operations	57
4.3.2	Linear algebra and transformations	58
4.3.3	Iterative linear algebra operations	61
4.3.4	Miscellaneous	62
4.4	Symmetry	62
4.4.1	Symmetry Group and Operations Functions	62
4.4.2	Geometries and Gradients	64
4.4.3	Character Tables	65
4.4.4	Atomic/Molecular Orbitals	66
4.4.5	‘Skeleton’ integral lists	67
4.4.6	Printing Symmetry Information	69
4.4.7	Internal symmetry stuff that might be useful	70
4.4.8	Miscellaneous	70
5	Integral Application Programmer’s Interface	71
5.1	Overview	71

<i>CONTENTS</i>	17
5.2 Adding a new base integral code to the NWChem INT-API	72
6 Software Development Toolkit	73
6.1 Non-Uniform Memory Allocation (NUMA)	73
6.1.1 Message Passing	74
6.1.2 Memory Allocator (MA)	75
6.1.3 Global Arrays (GA)	79
6.1.4 ChemI/O	89
6.2 The Run Time Data Base (RTDB)	100
6.2.1 Functions to Control Access to the Runtime Database	100
7 Utilities	107
7.1 Input Parser	107
7.1.1 Free-format Fortran Input Routines – INP	107
7.1.2 Initialization	107
7.1.3 Basic Input Routines	108
7.1.4 Routines concerning fields within a line	110
7.1.5 String routines	111
7.1.6 Error handling routines	112
7.2 NWChem Output to Ecce	113
7.2.1 Contents of Output for Ecce	113
7.2.2 Format of Output	114
7.2.3 NWChem Ecce Output API	115
7.2.4 Standard exit status	118
7.2.5 Standard keywords	119
7.3 Utility routines	121
7.3.1 Printing utilities	122
7.3.2 Error Routines	123
7.3.3 Parallel Communication	123
7.3.4 Naming Files	123
7.3.5 Sequential Fortran Files	125
7.3.6 Parallel file operations	125
7.3.7 Data packing and unpacking	126
7.3.8 Checksums	126
7.3.9 Source version information	128

7.3.10	Times and dates	128
7.3.11	System operations and information	129
7.3.12	C to Fortran interface	130
7.3.13	Debugging aids	130
7.3.14	Miscellaneous BLAS-like operations	130
7.4	Print Control	131
7.4.1	Other Relevant Routines	133
8	Installing NWChem	135
8.1	How to Obtain NWChem	135
8.2	Supported Platforms	135
8.2.1	Porting Notes	136
8.3	Environmental Variables	136
8.4	Makefiles and Libraries	137
8.5	Managing NWChem	142
8.5.1	Introduction to CVS	142
8.5.2	The CVS Model	142
8.5.3	The CVS Program	143
8.5.4	Summary of CVS commands	145
8.5.5	Troubleshooting CVS	146
9	Developing New Modules and Enhancements	147
9.1	General Design Guidelines	147
9.2	Coding Style	148
9.2.1	Version information	148
9.2.2	Standard interface for top-level modules	149
9.2.3	No globally defined common blocks	149
9.2.4	Naming of routines and common blocks	149
9.2.5	Inclusion of common block definitions	149
9.2.6	Convention for naming <code>include</code> files	150
9.2.7	Syntax for including files using the C preprocessor	150
9.2.8	No implicitly typed variables	150
9.2.9	Use <code>double precision</code> rather than <code>real*8</code>	151
9.2.10	C macro definitions should be in upper case	151
9.2.11	Fortran source should be in lower or mixed case	151

<i>CONTENTS</i>	19
9.2.12 Naming of variables holding handles/pointers obtained from MA/GA	151
9.2.13 Fortran unit numbers	151
9.2.14 Use standard print control	151
9.2.15 Error handling	152
9.2.16 Comments	152
9.2.17 Message IDs	153
9.2.18 Bit operations — <code>bitops.fh</code>	153
9.2.19 Blockdata statements and linking	154
10 Testing the Program	155
11 Documenting New Modules and Code Enhancements	157
11.1 Content of the Documentation	157
11.1.1 Documentation of a Molecular Calculation Module	158
11.1.2 Documentation of Modeling or Development Tools	159
11.1.3 Content for In-Source Documentation of Routines	160
A Integral Application Programmer's Interface	163
A.1 INT-API: Initialization, Integral Accuracy and Termination	163
A.1.1 <code>int_init</code>	163
A.1.2 <code>intd_init</code>	163
A.1.3 <code>int_terminate</code>	164
A.1.4 <code>intd_terminate</code>	164
A.1.5 <code>intdd_terminate</code>	164
A.1.6 <code>int_acc_std</code>	164
A.1.7 <code>int_acc_high</code>	165
A.1.8 <code>int_acc_get</code>	165
A.1.9 <code>int_acc_set</code>	165
A.2 INT-API: Memory Management Routines	165
A.2.1 <code>int_mem</code>	166
A.2.2 <code>int_mem_1e</code>	166
A.2.3 <code>int_mem_2e4c</code>	166
A.2.4 <code>int_mem_h1</code>	166
A.2.5 <code>int_mem_2eec</code>	167
A.2.6 <code>int_mem_2e2c</code>	167

A.2.7	int_mem_3ov	167
A.2.8	int_mem_print	167
A.2.9	intb_mem_2e4c	167
A.3	INT-API: Integral Routines	168
A.3.1	int_1estv	168
A.3.2	int_1eov	169
A.3.3	int_1eke	169
A.3.4	int_1epe	170
A.3.5	int_1eh1	170
A.3.6	int_1eall	171
A.3.7	int_1cg	172
A.3.8	int_1e3ov	172
A.3.9	int_11e3ov	173
A.3.10	int_11eall	173
A.3.11	int_pgen1e	174
A.3.12	int_11eh1	174
A.3.13	int_11eke	175
A.3.14	int_11eov	176
A.3.15	int_11epe	176
A.3.16	int_11gen1e	177
A.3.17	int_2e2c	177
A.3.18	int_2e2c	178
A.3.19	int_12e3c	178
A.3.20	int_2e3c	179
A.3.21	int_2e4c	179
A.3.22	int_12e4c	180
A.3.23	intb_init4c	181
A.3.24	intb_2e4c	181
A.3.25	intb_nw_2e4c	183
A.4	INT-API: Property Integral Routines	184
A.4.1	int_mpole	184
A.4.2	int_projpole	184
A.5	INT-API: Miscellaneous Routines	185
A.5.1	exact_mem	185

A.5.2	emem_3ov	185
A.5.3	emem_1e	186
A.5.4	emem_1e_dk	186
A.5.5	emem_1e_rel	186
A.5.6	emem_2e4c	187
A.5.7	emem_2e3c	187
A.5.8	emem_2e2c	188
A.5.9	emem_2e4c_full	188
A.5.10	int_nbf_max	188
A.5.11	int_mem_zero	188
A.5.12	api_is_ecp_basis	189
A.5.13	emem_1e_pvp	189
A.5.14	exactd_mem	189
A.5.15	emem_d1e	190
A.5.16	emem_d1e_rel	190
A.5.17	emem_d2e4c	190
A.5.18	emem_d2e4c_full	190
A.5.19	int_canon	191
A.5.20	int_chk_init	191
A.5.21	int_chk_sh	191
A.5.22	int_nospherical_check	191
A.5.23	int_nogencont_check	192
A.5.24	int_nospshell_check	192
A.5.25	int_bothsp_gc_check	192
A.5.26	int_hf1sp	192
A.5.27	int_hf1sp_ecp	193
A.5.28	int_1psp	194
A.5.29	int_1dsp	195
A.5.30	int_1spsp	196
A.5.31	int_1spa	196
A.5.32	int_sp1b	197
A.5.33	int_nint	197
A.5.34	int_unint	197
A.5.35	int_nint_cart	198

A.5.36	int_unint_cart	198
B	Performance Statistics Collection — PSTAT	199
B.1	Model	199
B.2	API	199
B.2.1	Include files	199
B.2.2	pstat_init	199
B.2.3	pstat_terminate	200
B.2.4	pstat_allocate	200
B.2.5	pstat_free	200
B.2.6	pstat_on	200
B.2.7	pstat_off	200
B.2.8	pstat_acc	200
B.2.9	pstat_print_all and pstat_print	201
B.2.10	Usage Notes	201
B.3	Closing Comment	201
C	Integral File I/O – INT2E	203
C.1	Application- and I/O-Level Views of the Data Stream	203
C.2	Internal Data Structures (all are node-private)	203
C.2.1	Cache	204
C.3	Subprograms	204
C.3.1	sread, swrite (in util directory)	204
C.3.2	int2e_file_open (API)	204
C.3.3	int2e_file_close (API)	204
C.3.4	int2e_file_rewind (API)	204
C.3.5	int2e_file_read (API)	205
C.3.6	int2e_file_write (API)	205
C.3.7	int2e_file_write_big (internal)	205
C.3.8	int2e_buf_read, int2e_buf_write (mostly internal)	205
C.3.9	int2e_buf_clear (internal)	205
C.3.10	int2e_buf_cntr_pack, int2e_buf_cntr_unpack (internal)	205
C.3.11	int2e_buf_pack, int2e_buf_unpack (internal)	206
C.3.12	int2e_set_bf_range, int2e_get_bf_range (API)	206

CONTENTS

23

D NWChem Error Messages

207

Chapter 1

NWChem Overview

1.1 Introduction

NWChem is a computational chemistry package designed to run on high-performance parallel supercomputers. The code contains many methods for computing properties of molecular and periodic systems using standard quantum mechanical descriptions of the electronic wavefunction or density. In addition, NWChem can perform classical molecular dynamics and free energy simulations. These approaches can be combined to perform mixed quantum-mechanics and molecular-mechanics simulations.

The code functions by performing specific tasks requested by the user, executing the particular operations using the specified theory. The code currently supports the following theory options:

- Self Consistent Field (SCF) or Hartree Fock (RHF, UHF, high-spin ROHF).
- Multiconfiguration SCF (MCSCF)
- Gaussian Density Functional Theory (DFT) for molecules
- Density Functional Theory for periodic systems (GAPSS)
- MP2 using a semi-direct or fully direct algorithm
- MP2 using Resolution of the Identity (RI) approximation
- Coupled-cluster single and double (CCSD) excitations
- Selected configuration interaction (CI) with perturbation correction
- Classical molecular dynamics simulation (nwARGOS)

For these theories, numerical first and second derivatives are automatically computed if analytic derivatives are not available. Any of these theories can be used to perform the following operations:

- Single point energy
- Geometry optimization (minimization and transition state)
- Molecular dynamics on the fully *ab initio* potential energy surface

- Normal mode vibrational analysis.
- Generation of the electron density file for the *Insight* graphical program
- Evaluation of static, one-electron properties.

The following quantum mechanical methods are available to calculate energies and analytic first derivatives with respect to atomic coordinates. (Second derivatives are computed by finite difference of the first derivatives.)

- Self Consistent Field (SCF) or Hartree Fock (RHF, UHF, high-spin ROHF).
- Gaussian Density Functional Theory (DFT), using many local and non-local exchange-correlation potentials (RHF or UHF).
- MP2 semi-direct using frozen core and RHF and UHF reference.
- Complete active space SCF (CASSCF).

The following methods are available to compute energies only. (First and second derivatives are computed by finite difference of the energies.)

- MP3, MP4, CCSD, CCSD(T), CCSD+T(CCSD), with RHF reference.
- Selected-CI with second-order perturbation correction.
- MP2 fully-direct with RHF reference.
- Resolution of the identity integral approximation MP2 (RI-MP2), with RHF or UHF reference.

In addition, automatic interfaces are provided to perform calculations using the following external programs:

- The COLUMBUS multi-reference CI package
- The Natural Bond Orbital (NBO) package

Classical molecular dynamics simulations can be performed using the nWARGOS module. The operations supported currently include the following:

- Single configuration energy evaluation
- Energy minimization
- Molecular dynamics simulation
- Free energy simulation

NWChem also has the capability to combine classical and quantum descriptions in order to perform the following calculations:

- Mixed quantum-mechanics and molecular-mechanics (QM/MM) minimizations
- Molecular dynamics using any of the quantum mechanical wavefunctions.

The broad functionality in the code and the requirements of efficient programming for parallel processing demand modularity of design. This means that the architecture of NWChem must be very carefully structured, and any new modules developed to add functionality to the code must adhere very strictly to the prescribed design and programming practices. The following section describes in broad outline the architecture of NWChem, and serves as an introduction to the detailed discussion of the elements and modules of the code, which is found in the subsequent chapters of this manual.

Anyone wishing to develop new modules or enhancements for NWChem should study this chapter and the following three chapters very carefully before attempting to modify the code. A Glossary is also included at the front of this manual, to help clarify the usage of specific terms and specialized jargon used in reference to the structure, functionality, and operation of NWChem. Questions on the code should be addressed to the NWChem developers group, which can be reached via electronic mail at `nwchem-developers@emsl.pnl.gov`.

Developers can also subscribe to this electronic mailing list by sending a message to

```
majordomo@emsl.pnl.gov
```

The body of the message must contain the line

```
subscribe nwchem-developers
```

Code modifications should be undertaken only when one has achieved a more than superficial understanding of the inner workings of the code, and obtained the blessing of the NWChem Program Manager.

1.2 NWChem Architecture

NWChem has a five-tiered modular architecture. This structure is illustrated conceptually by the diagram in Figure 1, which shows the five tiers and their relationships to each other. The first tier is the *generic task interface*. This interface¹ serves as the mechanism that transfers control to the different modules in the second tier, which consists of the *Molecular Calculation Modules*. The molecular calculation modules are the high level programming modules that accomplish computational tasks, performing particular operations using the specified theories defined by the user input file. These independent modules of NWChem share data only through a disk-resident data base, which allows modules to share data or to share access to files containing data. The third tier consists of the *Molecular Modeling Tools*. These routines provide basic chemical functionality such as symmetry, basis sets, grids, geometry, and integrals. The fourth tier is the *Software Development Toolkit*, which is the basic foundation of the code. The fifth tier provides the *Utility Functions* needed by nearly all modules in the code. These include such functionality as input processing, output processing, and timing.

In addition to using a modular approach for the design, NWChem is built on the concepts of object oriented programming (OOP) and non-uniform memory access (NUMA). The OOP approach might seem incompatible with a code written primarily in Fortran77, since it does not have all of the necessary functionality for an object oriented language (OOL). However, many of the required features can be simulated by careful adherence to the guidelines for encapsulation and data hiding outlined in Section 9.2. The main advantage of an object-oriented approach is that it allows for orderly and logical access to data more-or-less independent of why or when a given module might require the information. In addition, it allows considerable flexibility in the manipulation and distribution of data on shared memory, distributed memory, and massively parallel hardware architectures, which is needed in a NUMA approach to parallel computations. However, this model does require that the program developer have a fairly comprehensive understanding of the overall structure of the code and the way in which the various parts fit together.

The following subsections describe this structure in broad outline, and refer to specific chapters and sections in the code where the various modules, tools, and "objects" are described in detail.

¹Note that this is an abstract programming interface, not a user interface. The user's 'interface' with the code is the input file.

1.2.1 Object Oriented Design

The basic principles of object-oriented software development are abstraction, hierarchy, encapsulation, and modularity. *Abstraction* is the separation of the problem to be solved from the process used to solve it, which facilitates the introduction of new methods as programming tools and hardware capabilities evolve. In complex systems, abstraction can be carried out on many levels, resulting in a hierarchy that allows connections between many different components and the development of further abstractions. *Encapsulation* is the creation of isolated data structures or other objects in such a way that they can be manipulated only in carefully controlled and well-defined ways, which helps to reduce the problems due to unexpected interactions between components that are supposed to be independent. *Modularity*, which is the use of relatively small program units having well-defined functionality, can also help reduce interaction problems. It can also aid overall code efficiency, if the modules are written to be easily reused.

In an object oriented language such as C++, this methodology can be a feature of the actual coding of the program. NWChem is written in a mixture of C and Fortran, however, and uses object oriented ideas at the design stage. This requires some self-discipline on the part of the developers, but the effort is well rewarded in improved implementation and easier code maintenance. In a programming language such as Fortran77, which is not object oriented by design, the concept of objects can be simulated by developing a well defined interface for the programmer to use that in essence hides all of the gory details of "creating", "manipulating", and "destroying" an object. The objects are treated as if they can be manipulated only through the interface. In reality, of course, Fortran 77 allows the programmer to use any of the "private" data and routines that are underneath the interface. For this reason, the rules for encapsulation and data hiding must be adhered to religiously, by following the guidelines outlined in Section 9.2.

One of the basic features of an object is that all of the data and functions related to the data are encapsulated and available only through a "public" programming interface. This encapsulation feature allows programmers to put related data together in one object to be accessed in a well defined manner. For example, the basis set object (described further in Section 4.2) contains the number of basis functions, the exponents, the coefficients and other data related to basis sets. It also has a very well defined interface that can be used to access and manipulate the data. Because the data description, the internal "private" functions, and the "public" interface together define the abstract concept of the object, specific examples of the objects need to be created (instantiated).

Instantiations (or unique copies) of objects are simulated by allowing the user and the programmer to use different handles for different objects of the same type. This feature gives the user the capability of defining different basis sets during computation simply by naming different basis set objects (see Section 4.2). For example, two different basis sets can be defined for a molecule in an input file, as follows;

```
geometry
  Ne 0.0 0.0 0.0
end
basis "dz set"
  Ne library cc-pvdz
end
basis "qz set"
  Ne library cc-pvqz
end
set "ao basis" "dz set"
task scf
set "ao basis" "qz set"
task scf
task mp2
```

The above example has two basis sets that have the same object abstraction, (exponents and coefficients, etc.), but are different instantiations of the object, "dz set" and "qz set", with different handles (i.e., names). The

handles can then be used to represent the currently "active" basis set for the computation, using the input command set "ao basis" "qz set".

Related to the object oriented design is the idea of an abstract programming interface (API). An API provides a common interface to many different methods that perform the same type of task. An API is different from an object in the sense that there is no instantiation process. Also, while the functions are encapsulated, there is really no data that is encapsulated. For example, memory objects, basis objects, and geometry objects are passed into the integral API and integrals are passed back out in the memory objects. The integral API decides which of three different integral packages will be used to compute the integrals.

1.2.2 Non-Uniform Memory Access

One of NWChem's design goals is to scale to massively parallel hardware architectures in all aspects of the hardware: CPU, disk, and memory. With this goal in mind, distributing the data across all of the nodes becomes necessary. Therefore, in addition to the modular and object oriented architecture discussed above, NWChem is built on the principle of non-uniform memory access (NUMA). Just as a workstation has various levels of memory (registers, primary and secondary cache, memory and swap space) with varying sizes and access speed, distributing data across nodes "simply" adds another level of remote memory. The programmer must be aware of this extra level of memory access when designing the parallel algorithms in NWChem to get efficient, scalable code.

The MA tool allows the programmer to allocate memory that is local to the calling process. This is data that will generally not be directly shared with other processes, such as workspace for a particular local calculation or for replication of very small sets of data.

The GA tool supports the NUMA model by allowing nodes to share arrays between processes as if the memory is physically shared. It allows the programmer to use relatively simple routines to access and manipulate data in the shared arrays. However, the programmer must be aware that access to shared data will be slower than access to local data.

Just as GA allows the programmer to effectively use the NUMA model for memory, ChemIO is used to create files that are either local to the process or distributed among file systems. This allows the programmer to perform parallel I/O in the most efficient method for the particular algorithm or the particular hardware.

Together, MA, GA, and ChemIO provide the tools needed to accomplish a NUMA architecture. They also form a significant part of the Software Development Toolkit layer.

1.2.3 The Five-Tiered Modular Architecture

With the basic understanding of the object oriented approach and the NUMA approach, the programmer also needs to understand the basic modular architecture that is used in NWChem. This section provides a basic overview of each of the tiers describes how they fit together to make a cohesive and extensible program.

The Generic Task Interface

In old-fashioned structured Fortran programming, the Generic Task Interface would be referred to as the main program. As the "interface" between the user and the chemistry modules comprising NWChem, the generic task interface processes the input, sets up the parallel environment, and performs any initialization needed for the desired calculations. It then transfers control to the appropriate module, which performs the calculation. After a particular task is completed, control returns to the main program. If the input specifies more than one task, control is transferred to the appropriate module for the next task. This process continues until all specified tasks have been completed, or an error condition occurs. When all tasks complete successfully, the interface terminates program execution in an orderly manner. When

errors occur, the interface tries to terminate program execution gracefully, but the degree of success depends somewhat on the severity of the error. Chapter 2 presents a detailed discussion of the Generic Task Interface, and how it functions in NWChem.

The Molecular Calculation Modules

The second level of the five-tiered structure of NWChem consists of the high level molecular calculation modules. These are independent modules that perform the various functions of the code specified by the task directives. Examples include the self-consistent field (SCF) energy, the SCF analytic gradient, and the density functional theory (DFT) energy modules. The independent molecular calculation modules in NWChem can share data only through a run time database or through other well defined disk files. Each of the modules in this layer use toolkits and routines in the lower layers of the architecture to accomplish their tasks. Chapter 3 presents discussions of each of the calculational modules in NWChem, and the various operations that can be performed with these modules.

The Molecular Modeling Toolkit

The third level of the architecture of NWChem consists of the molecular modeling toolkit. Chapter 4 describes the elements of this toolkit in detail, including discussions of the geometry object (see Section 4.1), the basis set object (see Section 4.2), the linear algebra routines (see Section 4.3), symmetry (see Section 4.4, and the integral API (see Section ??). Each of these tools provides a basic functionality that is common to many of the algorithms in chemistry. The integral API provides a common interface to the three integral packages available in NWChem. The basis set object provides the programmer with information related to a specific basis set. The geometry object provides the basic geometry in different formats and provides for the definition of molecular as well as periodic systems. It also has information such as symmetry, atomic charges, and atomic mass. The linear algebra routines provide many general algorithms for basic vector-vector, vector-matrix, and matrix-matrix operations, and for solving eigenproblems.

The Software Development Toolkit

The Software Development Toolkit makes up the foundation level of the five-tiered structure of the code, and is the feature that makes it possible to develop an object oriented code that is constructed mainly in Fortran77. Chapter 6 presents a detailed discussion of this toolkit, which consists of four objects. These are the runtime database (RTDB) (see Section 6.2), the memory allocator (MA) (see Section 6.1.2), Global Arrays (GA) (see Section 6.1.3), and ChemIO (see Section 6.1.4). Each of these tools provides the interface between the chemistry specific part of the program and the hardware. They also support the NUMA parallel programming paradigm used by NWChem.

The RTDB is a persistent data storage mechanism used in NWChem to hold calculation specific information for the high level programming modules. Since NWChem does not destroy the RTDB at the end of a calculation unless specifically directed to do so by the user, a given RTDB can be used in several independent calculations. The MA tool allocates memory that will be used local to the processor. The GA tool allocates memory that is distributed across nodes (or shared in the case of shared memory machines) and is addressable by all of the nodes. ChemIO is a high performance I/O API designed to meet the requirements of large-scale computational chemistry problems. It allows the programmer to create I/O files that may be local or distributed.

The Utility Routines

This lowest level of the architecture contains many of the basic "odds-and-ends" functionality. These are utility routines that most of the above tiers need. Examples include the timing routines, the input parser, and the print routines.

Chapter 2

Generic Task Interface

The high-level flow of control within NWChem was broadly outlined in the discussion of NWChem architecture (see Section 1.2). This chapter covers the details of internal communication between modules and the control of program execution. This information is needed if NWChem is to be embedded in another application or when new modules are developed for the code.

2.1 Flow of Control in NWChem

The Generic Task Interface controls the execution of NWChem. The flow of control proceeds in the following steps:

1. Begin initialization of the parallel environment.
2. Identify and open the input file.
3. Complete the initialization of the parallel environment.
4. Scan the input file for a memory directive.
5. Process start-up directives.
6. Summarize start-up information and write it to the output file.
7. Open the runtime database with the appropriate mode.
8. Process the input sequentially (ignoring start-up directives), including the first `task` directive.
9. Execute the task.
10. Repeat steps 8 and 9 until reaching the end of the input file, or encountering a fatal error condition.

In Step 1, the parallel environment is initialized by calling the TCGMSG wrapper routine `pbeginf()`. This creates the parallel processes and provides basic message-passing. Before the global arrays can be initialized, however, user-specified memory parameters must be obtained from the input file. This requires execution of Step 2 to open the input file.

The input file is opened *only* by process zero. The name of the input file is determined by the routine `get_input_file_name()`. (The convention for the input file name is documented in the user manual. The default name is `nwchem.nw`.) The

input file is scanned by process zero for a memory directive using the routine `input_mem_size()`. Defaults are provided for all memory parameters not provided by the user, and results are broadcast to all nodes. At this point the local memory allocator (MA) and then the global array library (GA) are initialized. Completion of these steps fully initializes the parallel environment.

The next step is to process the startup directives that are contained in the input file. This is done to determine the type of calculation being undertaken (i.e., startup, restart, or continue), the name of the database, and the location of the permanent and scratch directories. Note that only process zero scans the input file, using routine `input_file_info()`. The information obtained by process zero when reading the input file are broadcast to all nodes, however, and this information is summarized to the output file. Process zero then opens the database with the appropriate mode (*empty* for startup, *old* for restart or continue).

At this point NWChem is fully functional and ready to process user input beyond the startup information. If the startup mode is 'continue', however, no more input is processed and the code attempts to continue the previously executing task from the information in the database. No new input information will be processed until that task is completed. Once the continued task is finished, however, or if the startup mode is for a new or restarted input file, the input file is read sequentially from the beginning (ignoring startup directives since they have already been processed).

As long as input is available from the input file, the input module (routine `input_parse()`) is invoked to read it, up to and including a task directive. Each input line is processed, and data is inserted into the data base for later retrieval. Note that within the input module, only process zero is executing code, reading input or putting data into the database. To enable this, the database is switched into sequential mode at the beginning of the input module, and back to parallel at the end.

Once a task directive is processed and entered into the database, control is returned to the main program so that the task can be carried out. The main program initiates the execution of the task by calling the routine `task()`. If the task fails, a fatal error is generated either by `task()` itself or a lower level routine. The task information remains in the database so that the task may be continued in another job. If the task finishes successfully, `task()` removes information about the completed task from the database, and the main program invokes the input module once again.

The input module continues to sequentially process the input. If it encounters another task directive, control is returned to the main program and the execution of the task is initiated, as described above. Upon successful completion, the main program again returns control to the input module and input processing continues. If the input module does not encounter a task directive before running out of input (physical or logical end of file) it returns false and the loop in the main program terminates.

Once all input has been processed and there are no more tasks to execute, the code attempts to clean up by closing the database, tidying up GA, and finally gracefully killing the parallel processes. Statistics concerning the database, MA and GA are printed to the output file, and execution terminates.

When a new module is introduced into NWChem, it must conform to this orderly control process. The new module must be appropriately invoked by the task routines. In addition, if it requires new input, the new module's input routine must be appropriately invoked by `input_parse()` (see Section 7.1 for details on the input parser). The new module's input routine must also be structured so that it allows only process zero to execute the code that reads user input.

Any new module developed for NWChem must also conform to the design goal that restart/continuation jobs with no repeated input behave exactly the same as if all input and tasks were specified in a single file. These attributes imply that *all* input data be processed and entered into the database or another persistent file. This means that in-core data structures should not be initialized within the input module. (Doing so will result in only process zero having the information and restarts will not work correctly.) In addition, input routines must not require having basis set or geometry information available, since these are not known until a task is actually invoked.

2.2 Task Execution in NWChem

As described above, NWChem executes all tasks by invoking the routine `task()`. The main program does not actually know what a particular task is — the necessary information is passed from the input module to the task library via the database. This makes the top level structure of NWChem very simple. The same simplicity is desirable in many applications. For instance, molecular geometry optimizers (or QM/MM programs) should work for all levels of theory and should not have to be modified if a new theory is introduced into the code. Similarly, routines that compute gradients and Hessians by finite difference need to be able to save and restore the state associated with each type of wavefunction.

NWChem contains a layer of routines that can perform the most common tasks/computations for all available wavefunctions. The following subsection lists the routines in this layer, with their arguments and calling conventions.

2.2.1 Task Routines for NWChem Operations

The highest level of the task routines is subroutine (`task()`), which is only invoked by the main NWChem program. The other task routines, however, can be invoked from almost any module. (Nested calls to the same subroutine should be avoided, however, since most NWChem routines are not reentrant.) The database argument passing conventions of modules in NWChem were developed in their present form mainly to support this layer.

`task`

```
subroutine task(rtdb)
integer rtdb           ! [input] data base handle
```

Called by ALL processes. After `task_input` has read the task directive and put stuff into the database this routine gets the data out and invokes the desired action.

If the operation is in the list of those supported by generic routines then the generic routine is called. Otherwise, a match is attempted for a specialized routine. If no operation is specified and no specialized routine located, then it is assumed that a generic energy calculation is required.

This needs extending to accommodate QM/MM and other mixed methods by having both MM and QM pieces specified (e.g., `task md dft`).

`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

```

Generic NWChem interface to compute the energy. Currently only the QM components are supported.

task_gradient

```

        logical function task_gradient(rtodb)

c   RTDB input parameters
c   -----
c   task:theory (string) - name of (QM) level of theory to use
c   task:numerical (logical) - optional - if true use numerical
c       differentiation. If absent or false use default selection.
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:gradient (real array) - derivative w.r.t. geometry cart. coords.
c   . task:dipole (real(3)) - total dipole 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

```

Generic NWChem interface to compute the energy and gradient. Currently only the QM components are supported.

Since this routine is directly invoked by application modules no input is processed in this routine. If the method does not have analytic derivatives the numerical derivative routine is automatically called.

task_freq

```

        logical function task_freq(rtodb)

c   RTDB input parameters
c   -----
c   task:theory
c
c   RTDB output parameters
c   -----
c   task:hessian file name (string) - name of file containing hessian
c   task:status (logical) - T/F on success/failure
c   task:cputime

```

```
c    task:walltime
```

Central difference calculation of the hessian using the generic energy/gradient interface. Uses a routine inside stepper to do the finite difference ... this needs to be cleaned up to be independent of stepper.

Also will be hooked up to analytic methods as they are available.

Since this routine is directly invoked by application modules no input is processed in this routine.

```
task_hessian
```

```

    logical function task_hessian(rtdb)

c    RTDB input parameters
c    -----
c    task:theory (string) - name of (QM) level of theory to use
c    task:numerical (logical) - optional - if true use numerical
c    differentiation. if
c    task:analytic (logical) - force analytic hessian
c
c    RTDB output parameters no for analytic hessian at the moment.
c    -----
c    task:hessian file name - that has a lower triangular
c    (double precision) array
c    derivative w.r.t. geometry cart. coords.
c    task:status (logical) - T/F for success/failure
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
```

Generic NWChem interface to compute the analytic hessian.

If the method does not have analytic derivatives automatically calls the numerical derivative routine.

```
task_optimize
```

```

    logical function task_optimize(rtdb)

c    RTDB input parameters
c    -----
c    task:theory (string) - must be set for task_gradient to work
c
c    RTDB output parameters
c    -----
c    task:energy (real) - final energy from optimization
c    task:gradient (real) - final gradient from optimization
c    task:status (real) - T/F on success/failure
c    task:cputime
c    task:walltime
c    geometry - final geometry from optimization
```

c

Optimize a geometry using stepper and the generic task energy/gradient interface. Eventually will need another layer below here to handle the selection of other optimizers.

Since this routine can be directly invoked by application modules no input is processed in this routine. c

task_num_grad

```
logical function task_num_grad(rtdb)
integer rtdb          ! [input]
```

Returns energy and gradient at current geometry.

Computes derivatives of `task_energy()` with respect to nuclear displacements using numerical finite difference.

Uses symmetry and projects out rotations/translations.

task_save_state **and** task_restore_state

```
logical function task_save_state(rtdb,suffix)
```

```
integer rtdb          ! [input]
character*(*) suffix ! [input]
```

c

c Input argument ... the suffix

c

c RTDB arguments ... the theory name

c

c Output ... function value T/F on success/failure

c

Each module saves any files/database entries necessary to restart the calculation at its current point by appending the given suffix to any names.

The exact (and perhaps only) application of this routine is in computation of derivatives by finite difference. The energy/gradient is computed at a reference geometry (or zero field) and then the wavefunction is saved by calling this routine. Subsequent calculations at displaced geometries (or non-zero fields) call `task_restore_state()` in order to use the wavefunction at the reference geometry as a starting guess for the calculation at the displaced geometry. Thus, there is no need to save basis or geometry (or field) information. E.g., in the SCF only the MO vector file is saved.

Chapter 3

Molecular Calculation Modules

The molecular calculation modules are the high level molecular calculation programs within NWChem for performing *ab initio* electronic structure calculations. A wide range of computational chemistry methods have been implemented in NWChem, representing the core functionality of a general purpose computational chemistry package. These modules are essentially independent programs that perform various functions (such as energy minimization, geometry optimization, normal mode vibrational analysis, and molecular dynamics) using the appropriate theory for a specified operation. This chapter describes each module in detail under the appropriate theory heading below. The various operations that can be performed with the different modules are also described.

3.1 Theories

NWChem contains modules to support ten different theory options for molecular calculations.

- Self-Consistent Field (SCF) or Hartree-Fock
- Density Functional Theory (DFT) for molecules
- Density Functional Theory for periodic systems (GAPPS)
- MP2 using a fully direct or semi-direct algorithm
- MP2 using the Resolution of the Identity (RI) approximation
- Coupled-cluster single and double excitations
- Multiconfiguration SCF
- Selected configuration interaction with perturbation correction
- Classical molecular dynamics simulation using nwARGOS

The following subsections describe the internal program structure of each of these modules.

3.1.1 Self-Consistent Field Module – SCF

The essential core functionality of NWChem is provided by the direct self-consistent field (SCF) module. SCF theory is based on the concept that in a system of N electrons each electron interacts with a mean potential created by the entire system, rather than explicitly with the other $(N-1)$ electrons. The self-consistent field (SCF) method is generally derived by assuming a specific form of the solution to the quantum mechanical equation as expressed in the electronic Schrödinger equation. This solution leads to a set of coupled integro-differential equations that can be solved numerically. Rather than actually solving these equations, however, the assumed solution is expanded in a finite set of primitive functions called the basis set, which is usually chosen to be the atomic orbitals. This yields a set of coupled homogeneous equations (the Hartree-Fock equations) that can be written in matrix form. The eigenvalues and eigenvectors of the matrix (which is the Fock matrix) describe the particle interactions.

The total energy of the molecular system is a function of the positions of the atoms and one-particle wavefunctions. A density matrix is defined over the occupied orbitals and can be used along with the one- and two-electron integrals of the atomic basis in an appropriate representation of the Fock matrix. In an SCF solution procedure, the molecular orbital coefficients are used to compute the density matrix, which in turn is used to construct the Fock matrix from the list of atomic orbital two-electron integrals. A new set of coefficients is obtained by solving the eigenvalue equation, and the cycle is repeated. Convergence of the wave function is satisfied when the molecular orbital coefficients in the matrix are self-consistent.

The implementation of the parallel direct SCF method in NWChem distributes the arrays describing the atoms and the corresponding basis functions across the aggregate memory of the system using the GA tools. The size of the system that can be modeled therefore scales with the size of the MPP and is not unduly constrained by the capacity of a single processor.

The construction of the Fock matrix, which is the computationally dominant step in the method, is readily parallelized since the integrals can be computed concurrently. A strip-mined approach is used, in which the integral contributions to small blocks of the Fock matrix are computed locally and accumulated asynchronously into the distributed matrix.

The conventional SCF solution scheme is based on repeated diagonalizations of the Fock matrix, but in parallel this operation can become a severe bottleneck in parallel implementations of the method. Quadratically convergent SCF is implemented in NWChem. In this approach, the equations are recast as a non-linear minimization. This bypasses the diagonalization step, replacing it with a quadratically convergent Newton-Raphson minimization. The scheme consists only of data parallel operations and matrix multiplications. This guarantees high efficiency on parallel machines. The method is also amenable to performance enhancements that can substantially reduce computation expense with no effect on the final accuracy, such as computing the orbital-Hessian vector products only approximately.

The scalability of this approach has been demonstrated on a wide variety of platforms. Solutions can be obtained for a closed-shell spin restricted (RHF) wavefunction, closed-shell spin unrestricted (UHF) wavefunction, or spin-restricted open shell (ROHF) wavefunction.

3.1.2 Gaussian Density Functional Theory Module – DFT

Density functional theory (DFT) provides an approach to solving the Kohn-Sham equation in which the total energy of the molecular system is a function of the positions of the atoms and one-particle densities. The approach in DFT is to assume a charge density and then obtain successively better approximations of the Hamiltonian. In traditional *ab initio* methods, by contrast, the approach is to assume an exact Hamiltonian and then obtain successively better approximations of the wavefunction. When the total energy is minimized with respect to the variational parameters, the resulting one-particle equations are exactly the same as the Hartree-Fock method except for the handling of the exchange terms and the way the electron exchange correlation is incorporated. The DFT method can yield results similar to those obtained with *ab initio* methods such as SCF, but at a substantially reduced computational effort.

NWChem contains a parallel implementation of the Hohenberg-Kohn-Sham formalism of density functional theory. The Gaussian basis DFT method breaks down the Hamiltonian into the same basic one-electron and two-electron components as traditional Hartree-Fock methods. In DFT, the two-electron component is further broken down into a Coulomb term and an exchange correlation term. The electron density and the exchange-correlation functional can also be expanded in terms of Gaussian basis sets.

DFT differs significantly from other methods in the treatment of the exchange-correlation term used in building the Fock matrix. The computationally intensive components of a DFT calculation include the fitting of the charge density, construction of the Coulomb potential, construction of the exchange-correlation potential, and the subsequent diagonalization of the resulting equations. The integrals required for the fitting of the charge density and the construction of the Coulomb contribution to the Fock matrix are independent and therefore can be computed in parallel. As with the SCF method, these independent integral contributions are computed locally using a strip-mined approach and accumulated asynchronously into the distributed matrix. Very little communication is required between nodes, other than a shared counter and global array accumulation step.

3.1.3 Møller-Plesset Module – MP2

Under construction.

3.1.4 Resolution of the Identity Approximate Integral Method

The amount of time spent computing the two-electron four-center integrals over gaussian basis functions is a significant component of many *ab initio* algorithms. Improvements in the computational efficiency of the base integral evaluation algorithms can have a significant effect on the overall speed of the calculation. The resolution of the identity (RI) method is an option available in NWChem for obtaining an approximation of the two-electron four-center integrals for Møller-Plesset theory (MP2). The method is also available as an extension to SCF calculations and DFT.

The basic approach of the RI method is to factor the four-center integral into two parts;

This identity is inserted into the two-electron integrals and (then it gets really complicated... Do we really want to go into this here?)

In the implementation of the RI method in NWChem the transformed three-center integrals are computed and then stored for repeated use. The integrals are stored in a global array using a distributed in-core method or a disk-based method. The in-core array may be distributed over the distributed memory of a parallel computer. The disk-based array is stored in a Disk Resident Array library. This approach can be used if there is not enough memory available to store the global array in-core, but it will result in slower access times.

RI-MP2

Under construction.

RISCF

The transformed integrals can be used in the calculation of the Coulomb and exchange contributions to the Fock matrix for any of the modules. In the case of restricted closed shell SCF calculations, the number of operations can be further reduced by inserting the definition of the density matrix and using the molecular orbital (MO) vectors instead. In the second-order SCF procedure as implemented in NWChem, the MO vectors are available during the energy and gradient calculations, but not during the line-search algorithm. In a DIIS-based Restricted Hartree-Fock (RHF) or SCF procedure, these savings in computation time could be used for every Fock build.

3.1.5 CCSD

Under construction.

3.1.6 MCSCF

Under construction.

3.1.7 CI

In the configuration interaction method, the many-electron wave function is expanded in Slater determinants or spin-adapted configuration-state functions (CSF) usually constructed from orthonormal orbitals.

The CI energy is the expectation value of the Hamiltonian operator. Variation of the expansion coefficients so as to minimize the energy leads to the matrix eigenvalue equation. These matrix elements are relatively simple in a determinant basis, but the use of spin symmetry typically makes the CSF expansions shorter by a factor of four. There are advantages to either approach.

Conventional CI methods explicitly construct the Hamiltonian matrix and apply an iterative eigenvalue method. Most algorithms for the solution of the eigenvector problem require the formation of matrix-vector products for a set of intermediate vectors. This feature is exploited in integral-driven direct-CI methods, which avoid explicit construction and storage of the potentially large Hamiltonian matrix. For large-scale wavefunction expansions, the computation of these matrix-vector products dominates the overall procedure.

Conventional and selected-CI methods are straightforwardly parallelized. The Hamiltonian matrix elements may be independently computed and stored on disk or in memory. A replicated data approach may be adopted for the matrix-vector products.

The full-CI wave function includes all possible CSFs of the appropriate S^2 and S_z (or determinates of S_z) spin quantum numbers. Full CI is the exact solution of the non-relativistic Schrödinger equation in the chosen one-particle basis, and the energy is invariant to orbital rotations. The length of the full-CI expansion grows very rapidly with the number of electrons and molecular orbitals, and consequently full-CI wave functions can be computed only for relatively small systems.

3.1.8 Molecular Mechanics (MM)

Under construction.

3.2 Operations

Operations are specific calculations performed in a task, using the level of theory specified by the user. The following list gives the selection of operations currently available in NWChem:

- Evaluate the single point energy.
- Evaluate the derivative of the energy with respect to nuclear coordinate gradient.
- Minimize the energy by varying the molecular structure.
- Conduct a search for a transition state (or saddle point).

- Calculate energies on a LST path defined by means of a z-matrix input.
- Compute second derivatives and print out an analysis of molecular vibrations.
- Compute molecular dynamics using nwARGOS.
- Perform multi-configuration thermodynamic integration using nwARGOS.

3.2.1 Energy

Under construction.

3.2.2 Gradient

Under construction.

3.2.3 Optimization

Under construction.

3.2.4 Frequencies

Under construction.

3.2.5 Properties

Under construction.

3.2.6 Dynamics

Molecular dynamics simulation in NWChem is based on a spacial decomposition of the molecular volume. This approach to parallelizing is based on a decomposition of the molecular simulation volume over the processing elements available for the calculation. The main advantage of this approach is that memory requirements are significantly reduced, compared to replicating all data on all nodes. In addition, the locality of short-range interactions significantly reduces the required communication between nodes to evaluate interatomic forces and energies.

There are two major disadvantages to this type of decomposition, however. Periodic redistribution of the atoms over the simulation volume is necessary, since the atoms are not constrained to remain within the region boundaries. The distribution of atoms in a system is usually not homogeneous, so in general the computational work will not be uniformly distributed over all nodes. Some nodes will be working hard while others are essentially idle. Periodic and dynamic balancing of the computation load is therefore required to reduce excessive synchronization times and increase parallel efficiency.

Communication is implemented using the Global Arrays toolkit, which allows the physically distributed memory to be treated as a single logical data object, using logical topology independent array addressing for simple data communication as well as for linear algebra operations. Remote memory access is one-sided and asynchronous when using Global Arrays. The data needed on one node can be retrieved by that node without actually communicating

directly with the node that owns the data. In the calculation of forces for a dynamics simulation, this allows a node to obtain the remote coordinates needed for the calculation of the forces, to accumulate the local forces rapidly, and to accumulate the the remote forces asynchronously. All of these steps are executed without synchronization or remote node involvement in initiating the data transfer. Point-to-point communication is required only when an atom moves from its current domain to a domain assigned to another node. This is implemented using a global synchronization to redistribute the atoms, and consists of the following five-step process:

1. Determine new node ownership of each local atom.
2. Copy the atomic data for each atom leaving a node domain into the local portion of a global array.
3. For each node that has atoms leaving its domain, send the pointers for the atomic data of the atoms changing domains to the global array space of the receiving node(s), in a one-sided communication.
4. Perform a global synchronization to ensure that all nodes that have atoms leaving their domain have done Step 2 and Step 3.
5. For each node that has atoms entering its domain, retrieve the atomic data from the global array (in a one-sided communication), using the pointers received in step 3.

Dynamic load balancing is used in NWChem to increase the efficiency of the spacial decomposition molecular dynamics algorithm by trying to keep all nodes more or less equally busy. Two methods are implemented in NWChem to accomplish this. In one option, load balancing is collective. The physical space assigned to the busiest node is decreased, reducing the size of its domain, and the domain size of all other nodes is increased. In the other option, the load balancing is local. The physical space assigned to the busiest node is decreased, but the domain size is increased only for the least busy immediately adjacent node. The collective method results in the most equitable allocation of work, but requires additional global communication. The local method requires minimal additional communication, but may not do much in the way of load balancing if all nodes near the busiest node are also working hard.

A molecular dynamics simulation in NWChem consists of the following major steps.

1. perform dynamic load balancing using the option selected by input
2. determine particle ownership –
 - perform asynchronous local one-sided communication to put atomic data and pointers into global arrays
 - perform global synchronization so that all coordinates will be updated
3. perform force evaluation, including the particle-mesh Ewald summation (pme)
4. perform synchronization required for dynamic load balancing
5. update coordinates, perform property evaluations, and record results

This sequence is repeated until the simulation is complete. Step 3, the force evaluation, is the most computationally intensive part of the calculation. In particle-mesh Ewald summation, the calculation of electrostatic forces and energies is separated into short range interactions and long range interactions. The short range interactions are calculated explicitly, and the long range interactions are approximated using a discrete convolution on an interpolating grid. Three-dimensional fast Fourier transforms are used to perform the convolution efficiently. Additional efficiency is achieved by performing the calculation of energies and forces in reciprocal space on a subset of the available nodes. All nodes must be involved in setting up the charge grid, but only a subset of the nodes have to perform the fast Fourier transforms and the computations in reciprocal space. Separating this work from the calculation of the pme atomic forces allows nodes that are not involved in the reciprocal work to continue immediately with calculation of the real space forces.

Chapter 4

Molecular Modeling Toolkit

The Molecular Modeling Toolkit provides the basic functionality common to many chemistry algorithms in NWChem. These tools include the geometry object, the basis set object, the integral API, and the linear algebra routines. These modules are not strictly "objects" in the sense usually used in an object oriented language, but they serve essentially the same purpose in that they encapsulate specific data and provide access to it through a well-defined abstract programming interface. The geometry object specifies the physical makeup of the chemical system by defining atomic centers, spatial location of the centers, and their nuclear charge. It can include an applied electric field, the symmetry, and other characteristics of the system. The basis set object handles the details of the Gaussian basis sets, and with the geometry object defines all information required by an *ab initio* calculation that is not specific to a particular electronic state. The integral API is a layer between the actual integral package and the application module. It allows the developer to essentially ignore the details of how the integrals are computed, which facilitates programming of new modules and allows incorporation of new integral techniques with minimal disruption of applications that use those integrals. The explicit separation of these objects greatly simplifies the development of the chemistry modules of NWChem and allows more flexible use of the code and easier maintenance. This chapter describes each of the tools in detail, so that developers can use them correctly when inserting new modules in the code or modifying existing modules.

4.1 The Geometry Object

The geometry object is used in NWChem to store and manipulate important information describing the molecular system to be modeled, not all of which is specifically connected with the geometry of the system. The geometry object serves four main purposes;

- provides a definition of the coordinate system and positioning in space (including lattice vectors for periodic systems)
- defines an association of names/tags with coordinates in space
- specifies the external potential (nuclear multipole moments, external fields, effective core potentials, ...) that define the Hamiltonian for all electronic structure methods
- stores most Hamiltonian related information (but not wavefunction related information).

The tag associated with a geometric center serves a number of purposes in NWChem. It provides a convenient and unambiguous way to refer to

- a specific chemical element (which provides default values for information such as nuclear charge, mass, number of electrons, ...)
- the name of an ‘atomic’ basis set
- a DFT grid

The tag can also serve as a test for symmetry equivalence, since lower symmetry can be forced by specifying different tags for otherwise symmetry equivalent centers.

The data contained in the geometry object (or information that can be derived from data in the object) include the following;

1. A description of the coordinates of all types of centers (e.g., atom, basis function)
2. Charges (or optionally, ECPs, ...) associated with those centers
3. Tags (names) of centers
4. Masses associated with centers
5. Variables for optimization (e.g., via constrained cartesians or Z-matrix variables)
6. Symmetry information
7. Any other simple scalar/vector attribute associated specifically with a center

Specific geometries are referenced through an integer handle. Multiple geometries can be defined such that any one of them may be accessible at any instant for a given problem. However, geometries can consume a large amount of memory, so it is usually advisable to keep the number of simultaneously ‘open’ geometries to a minimum.

Logical functions return true on success, false on failure. The following subsections describe in more detail the functions that return something other than the logical state.

4.1.1 Creating, destroying, loading and storing geometries

The following functions are used to create, destroy, load and store geometries.

`geom_create`

```
logical function geom_create(geom, name)
integer geom           [output]
character*(*) name    [input]
```

This is the only way to get a valid geometry handle. The user-supplied string for name is used only for identification in printout and subsequent executions of the `geom_create` function. If the geometry has already been opened, a handle to the existing copy is returned.

`geom_destroy`

```
logical function geom_destroy(geom)
integer geom          [input]
```

This function deletes the *in core* data structures associated with the geometry and makes the geometry invalid for further use. (Note that disk resident data is not deleted. The runtime database is preserved between calculations.)

`geom_check_handle`

```
logical function geom_check_handle(geom, msg)
integer geom          [input]
character*(*) msg     [input]
```

If the specified string `geom` is not a valid geometry handle this function prints out the string `msg` and returns `.false..`

`geom_rtdb_load`

```
logical function geom_rtdb_load(rtdb, geom, name)
integer rtdb          [input]
integer geom          [input]
character*(*) name    [input]
```

This function loads the named geometry from the data base. One level of translation is attempted upon the name. An entry with the name `name` is searched for in the database and if located the value of that entry is used as the name of the geometry, rather than `name` itself. The string specified for `geom` must be a valid handle created by `geom_create`. The same geometry in the database may be loaded into distinct in-memory geometry objects.

`geom_rtdb_store`

```
logical function geom_rtdb_store(rtdb, geom, name)
integer rtdb          [input]
integer geom          [input]
character*(*) name    [input]
```

This function stores the named geometry in the database. One level of translation is attempted upon the string supplied for `name`.

`geom_rtdb_delete`

```
logical function geom_rtdb_delete(rtdb, name)
integer rtdb          [input]
character*(*) name    [input]
```

This function deletes the named geometry from the data base. One level of translation is attempted. Nothing happens to in-core copies of any geometries.

4.1.2 Information About the Geometry

This section describes functions that are used to define information about specific geometries.

`geom.ncent`

```
logical function geom_ncent(geom, ncent)
integer geom           [input]
integer ncent         [output]
```

Returns in `ncent` the number of centers.

`geom.nuc.charge`

```
logical function geom_nuc_charge(geom, total_charge)
integer geom           [input]
double precision total_charge [output]
```

Returns the sum of the nuclear charges.

`geom.nuc.rep.energy`

```
logical function geom_nuc_rep_energy(geom, energy)
integer geom           [input]
double precision energy [output]
```

Returns the effective nuclear repulsion energy. (Refer also to functions `geom_include_bq bq()` and `geom_set_bq bq()`).

`geom.include.bq bq`

```
logical function geom_include_bq bq(geom)
integer geom           [input]
```

By default the nuclear repulsion energy returned by `geom_nuc_rep_energy` does not include the interactions between point-charges (i.e., centers which tag begins with `bq`). This is so that it is easy for QM-MM programs to generate effective Hamiltonians based on point charges and avoid double counting of contributions. This routine returns `.true.` or `.false.` if the BQ-BQ contributions are or are not being computed. The default (don't include BQ-BQ interactions) thus corresponds to a return value of `.false.`.

`geom.set.bq bq`

```
logical function geom_set_bq bq(geom, value)
integer geom           [input]
logical value         [input]
```

Sets the logical variable that determines if BQ-BQ interactions are included to `value`.

4.1.3 Information About Centers and Coordinates

This section describes functions that define information about the centers and coordinate system for the geometry object.

geom_cart_set

```
logical function geom_cart_set(geom, ncent, t, c, q)
integer geom           [input]
integer ncent         [input]
character*16 t(ncent) [input]
double precision c(3,ncent) [input]
double precision q(ncent) [input]
```

This function is a simple interface for setting tags (*t*), cartesian coordinates (*c*) and charges (*q*) for the geometry.

geom_cart_get

```
logical function geom_cart_get(geom, ncent, t, c, q)
integer geom           [input]
integer ncent         [output]
character*16 t(ncent) [output]
double precision c(3,ncent) [output]
double precision q(ncent) [output]
```

This function extracts information from the geometry. (It performs essentially the opposite action to that of the *set* functions described above.) The user must ensure that the arrays are of sufficient dimension to hold the output.

geom_cent_get

```
logical function geom_cent_get(geom, icent, t, c, q)
integer geom           [input]
integer icent         [input]
character*16 t        [output]
double precision c(3) [output]
double precision q     [output]
```

Returns tag/coordinates/charge about the center *icent*.

geom_cent_set

```
logical function geom_cent_set(geom, icent, t, c, q)
integer geom           [input]
integer icent         [input]
character*16 t        [input]
double precision c(3) [input]
double precision q     [input]
```

This function sets values for center `icent` inside the geometry. It is essentially the opposite of the function `geom_cent_get`.

`geom_cent_tag`

```
logical function geom_cent_tag(geom, icent, tag)
integer geom          [input]
integer icent         [input]
character*16 tag      [output]
```

Returns just the tag of the center `icent`.

`geom_check_cent`

```
logical function geom_check_cent(geom, msg, icent)
integer geom          [input]
character*(*) msg     [input]
integer icent         [input]
```

This function returns `.true.` if center `icent` is a valid center. Otherwise it returns `.false.` and prints out the message and other information.

4.1.4 Support for Periodic Systems

This section describes functions that are applicable only to periodic systems.

`geom_systype_get`

```
logical function geom_systype_get(geom, itype)
integer geom          [input]
integer itype         [input]
```

This function returns an integer flag corresponding to the system type in `itype`. Valid entries include the following

- 0 = Molecule
- 1 = Polymer
- 2 = Slab
- 3 = Crystal

`geom_latvec_get`

```
logical function geom_latvec_get(geom, vectors)
integer geom          [input]
double precision vectors(3) [output]
```

For periodic systems, this function returns the lattice constants.

geom_latang_get

```
logical function geom_latang_get(geom, angles)
integer geom           [input]
double precision angles(3) [output]
```

For periodic systems, this function returns the angles defining the lattice.

geom_recipvec_get

```
logical function geom_recipvec_get(geom,rvectors)
integer geom           [input]
double precision rvectors(3)[output]
```

For periodic systems, this function returns the constants of the reciprocal lattice.

geom_recipang_get

```
logical function geom_recipang_get(geom, rangles)
integer geom           [input]
double precision rangles(3) [output]
```

For periodic systems, this function returns the angles defining the reciprocal lattice (**units?**).

geom_volume_get

```
logical function geom_volume_get(geom,volume)
integer geom           [input]
double precision volume [output]
```

For periodic systems, this function returns the volume of the unit cell (**units?**).

geom_amatrix_get **and** geom_amatinv_get

```
logical function geom_amatrix_get(geom,amat)
integer geom           [input]
double precision amat(3,3) [output]
```

```
logical function geom_amatinv_get(geom,amatinv)
integer geom           [input]
double precision amatinv(3,3) [output]
```

For periodic systems, this function returns the ‘A-matrix’ or its inverse. This is the matrix that transforms fractional coordinates to a Cartesian system in atomic units (???). This matrix is the unit matrix for molecular systems.

4.1.5 Printing and Miscellaneous Routines

This section describes various useful functions that can be called upon to manipulate data in the geometry object.

`geom_print` **and** `geom_print_xyz`

```
logical function geom_print(geom)
integer geom          [input]

logical function geom_print_xyz(geom, unit)
integer geom          [input]
integer unit          [input]
```

This function prints out the geometry to standard output. The XYZ form prints the geometry out to the specified Fortran unit in the XYZ format of the molecular viewer *Xmol*.

`geom_set_user_units`

```
logical function geom_set_user_units(geom, units)
integer geom          [input]
character*(*) units  [input]
```

This function sets the coordinates that the user expects for input/output. It currently understands either ‘a.u.’ or ‘angstrom’. Note that geometries are always internally stored as cartesians in atomic units.

`geom_tag_to_element`

```
logical function geom_tag_to_element(tag, symbol, element, atn)
character*16 tag      [input]
character*(*) symbol  [output]
character*(*) element [output]
integer atn           [output]
```

This function attempts to interpret a tag as the name of a chemical element. If successful, it return the symbol, full name and atomic number of the element.

`geom_charge_center`

```
logical function geom_charge_center(geom)
integer geom          [input]
```

This function adjusts the cartesian coordinates so that the nuclear dipole moment is zero (i.e., defines the origin of the coordinate system at the center of charge.)

geom_num_core

```
logical function geom_num_core(rtdb, geom, module, ncore)
integer rtdb           [input]
integer geom           [input]
character*(*)         [input]
integer ncore          [output]
```

This function determines the number of core orbitals in a system based on the user defining the number of orbitals per atom. If there is no user input, the number of core orbitals in a system is determined by constituent atoms and the standard general chemistry concepts of core and valance.

geom_freeze

```
logical function geom_freeze(rtdb, geom, module, ncore)
integer rtdb           [input]
integer geom           [input]
character*(*) module  [input]
integer ncore          [output]
```

This function determines the number of frozen core orbitals in a system. If successful, it returns the number of frozen core orbitals, as well as a logical true value.

4.2 The Basis Set Object

The basis set object and corresponding API provides access to all information concerning a basis set from a unique handle. In this fashion, multiple distinct basis sets may be manipulated simultaneously on an equal footing. The internal data structures store only information for the unique tags in the geometry.

4.2.1 Creating, Destroying, Loading and Storing Basis Sets

Basis set handles must be created with `bas_create`. Other routines load and store basis sets from/to the database.

bas_create

```
logical function bas_create(basis, name)
integer basis          ! [output] returned handle
character*(*)name     ! [input] name of basis set.
```

This is the only source of a valid basis set handle. The input name is used for output/debug purposes and is not associated with anything in the database. An empty basis set is created (in memory only) and the handle is returned in `basis`.

bas_destroy

```
logical function bas_destroy(basis)
```

```
integer basis ![input] handle to basis set to be destroyed
```

Frees memory and destroys all information about an active in-memory basis and the associated mapping arrays.

```
bas_check_handle
```

```
logical function bas_check_handle(basis,msg)
integer basis      ! [input] handle
character*(*) msg  ! [input] error message
```

Returns `.true.` if `basis` is a valid basis set handle. Otherwise it returns `.false.` and prints the message and a list of known basis sets on `STDOUT`.

```
bas_rtdb_load
```

```
logical function bas_rtdb_load(rtdb, geom, basis, name)
integer rtdb      ! [input] rtdb handle
integer geom      ! [input] geometry handle with info loaded
integer basis     ! [input] basis handle
character*(*) name ! [input] name of basis in the rtdb
```

Routine loads a named basis set from the database (specified with the handle `rtdb`), and using the geometry information builds the mapping arrays to contractions or shells, basis functions, and centers. One level of translation is attempted upon the name — an entry with name `name` is searched for in the database and if located the value of that entry is used as the name of the basis, rather than `name` itself.

```
bas_rtdb_store
```

```
logical function bas_rtdb_store(rtdb, name, basis)
integer rtdb      ! [input] handle to database
character*(*) name ! [input] name to use when storing
integer basis     ! [input] handle to basis set
```

Stores the in-memory basis (referenced by the handle `basis`) into the specified database (referenced by the handle `rtdb`) using the specified name. One level of translation is attempted upon the name — an entry with name `name` is searched for in the database and if located the value of that entry is used as the name of the basis, rather than `name` itself. The in-memory basis set is unchanged.

4.2.2 Information About the Entire Basis

```
bas_high_angular
```

```
logical function bas_high_angular(basis,high_angular)
integer basis      ! [input] basis set handle
integer high_angular ! [output] high angular momentum of basis
```

Returns the highest angular-momentum present in the basis set.

bas_numbf

```
logical function bas_numbf(basis,nbf)
integer basis      ! [input] basis set handle
integer nbf        ! [output] number of basis functions
```

Returns the total number of functions in the basis set.

bas_name

```
logical function bas_name(basis,basis_name,trans_name)
integer      basis      ! [input] basis set handle
character*(*) basis_name ! [output] symbolic basis name
character*(*) trans_name ! [output] actual/translated basis name
```

Returns the name of the basis set. The “symbolic” name used by the program to load the basis is returned in name. If this name was used to refer to another basis (i.e., indirection was used) then the actual name of the basis is returned in trans (i.e., the translated name). Otherwise trans returns the same as name.

bas_numcont

```
logical function bas_numcont(basis,numcont)
integer basis      ! [input] basis set handle
integer numcont    ! [output] total number of contractions
```

Returns the total number of mapped general contractions (or shells) for the given basis set.

bas_nbf_cn_max

```
logical function bas_nbf_cn_max(basisin,nbf_max)
integer basisin      ! [input] basis set handle
integer nbf_max      ! [output] max(nbf in any contraction)
```

Returns the maximum number of basis functions in any general contraction.

bas_nbf_ce_max

```
logical function bas_nbf_ce_max(basisin,nbf_max)
integer basisin      ! [input] basis set handle
integer nbf_max      ! [output] max(nbf on any center)
```

Returns the maximum number of basis functions on any single center.

4.2.3 Mapping Between Centers, Shells/Contractions and Functions

bas_cn2ce

```
logical function bas_cn2ce(basis,cont,center)
integer basis      ! [input] basis set handle
integer cont       ! [input] mapped contraction index
integer center     ! [output] center index
```

Returns the center for a given mapped (as opposed to unique) contraction.

bas_cn2bfr

```
logical function bas_cn2bfr(basis,cont,ifirst,ilast)
integer basis      ! [input] basis set handle
integer cont       ! [input] mapped contraction index
integer ifirst     ! [output] first basis function
integer ilast      ! [output] last basis function
```

Returns the first basis function index of a mapped contraction in *ifirst* and the last basis function index in *ilast*.

bas_ce2bfr

```
logical function bas_ce2bfr(basis, icent, ibflo, ibfhi)
integer basis      ! [input] handle
integer icent      ! [input] no. of center
integer ibflo, ibfhi ! [output] range of functions on center
```

Returns the range of basis functions on a given center.

bas_ce2cnr

```
logical function bas_ce2cnr(basis,center,ifirst,ilast)
integer basis      ! [input] basis set handle
integer center     ! [input] center index
integer ifirst     ! [output] first mapped contraction
integer ilast      ! [output] last mapped contraction
```

Returns the range of mapped contractions on a given center.

bas_bf2ce

```
logical function bas_bf2ce(basis,testbf,center)
integer basis      ! [input] basis set handle
integer testbf     ! [input] basis function index
integer center     ! [output] center index
```

Returns the center on which a basis function resides.

bas_bf2cn

```
logical function bas_bf2cn(basis,testbf,cont)
integer basis    ! [input] basis set handle
integer testbf   ! [input] basis function index
integer cont     ! [output] mapped contraction index
```

Returns the mapped contraction index that contains the given basis function index.

4.2.4 Printing Basis Sets

bas_print

```
logical function bas_print(basis)
integer basis    ! [input] basis handle
```

Prints the information about the basis set on unique centers.

bas_print_all

```
logical function bas_print_all()
```

Debugging routine. Prints (using bas_print) information about all active basis sets.

gbs_map_print

```
logical function gbs_map_print(basis)
integer basis    ! [input] basis set handle
```

Prints detailed information about the mapping of the unique basis set information to the centers (using the geometry information). Mostly useful only for debugging.

4.2.5 Miscellaneous Other Functions for Basis Sets

The following subsections describe functions that can be used to obtain detailed contraction information, exponents, coefficients, and other information on a basis set.

bas_continfo

```
logical function bas_continfo(basis,icont,
&      type,nprimo,ngeno,sphcart)
integer basis          ! [input] basis handle
integer icont          ! [input] contraction index
integer type           ! [output] type (sp/s/p/d/..)
integer nprimo         ! [output] no. of primitives
integer ngeno          ! [output] no. of contractions
integer sphcart        ! [output] 0/1 for cartesian/spherical
```

Returns information about the specified general contraction or shell. Type is encoded so that the sequence *spd/sp/s/p/d/f...* map into *-2/-1/0/1/2/3/...*. The number of primitives is equivalent to the number of exponents. The number of contractions is the number of radial functions to which the primitives are contracted, or equivalently, the number of sets of coefficients.

`bas_get_exponent` **and** `bas_set_exponent`

```
logical function bas_get_exponent(basis,icont,exp)
integer basis          ! [input] basis set handle
integer icont          ! [input] mapped contraction index
double precision exp(*) ! [output] exponents

logical function bas_set_exponent(basis,icont,exp,nexp) integer
integer basis          ! [input] basis set handle
integer icont          ! [input] mapped contraction index
double precision exp ! [input] "new" exponents for contraction
integer nexp          ! [input] number of new exponents
```

Get/set the exponents associated with a contraction. When setting the exponents two points must be noted:

1. the number of new exponents must *exactly* match the number of old exponents, and
2. since internally exponents are only stored once for atoms of the same type, changes effect all atoms of the same type.

`bas_get_coeff` **and** `bas_set_coeff`

```
logical function bas_get_coeff(basis,icont,coeff)
integer basis          ! [input] basis set handle
integer icont          ! [input] mapped contraction index
double precision coeff(*) ! [output] mapped contraction coeffs.

logical function bas_set_coeff(basis,icont,coeff,ncoeff)
integer basis          ! [input] basis set handle
integer icont          ! [input] mapped contraction index
integer ncoeff         ! [input] number of coeffs.
double precision coeff(ncoeff) ! [input] "new" coeffs.
```

Get/set the contraction coefficients associated with a generally contracted function. The coefficients are stored as if the array was declared as `coeff(nprim,ngen)` where `nprim` is the number of primitive and `ngen` is the number of sets of coefficients. When setting the coefficients two points must be noted:

1. the number of new coefficients must *exactly* match the number of old coefficients (i.e., `ncoeff = nprim*ngen`, and
2. since internally coefficients are only stored once for atoms of the same type, changes effect all atoms of the same type.

4.2.6 Other — unique contraction information and adding centers

Routines exist to do all of this stuff, however, it is not anticipated that this functionality is necessary outside of existing input routines. Exceptions might include automatic creation of fitting basis sets or automatic optimization of an existing basis set. Rather than confuse most users by documenting this “private interface”, anyone seeking additional functionality should contact Rick or Robert — the interface you want is probably there.

4.3 Linear Algebra Routines

The linear algebra routines in NWChem provide standard functions for simple operations and transformations, and iterative operations. These routines all operate with global arrays, and have names prefaced with `ga_`. (Note that this does *not* mean they are part of the GA library, however.) Some of these routines reference other NWChem objects, such as geometries or basis sets, and all are collective. That is, all processes must invoke them at the same time, otherwise deadlock or a fatal error will result.

4.3.1 Simple linear operations

`ga_get_diag`

```
subroutine ga_get_diagonal(g_a, diags)
  integer g_a           ! [input] GA handle
  double precision diags(*) ! [output] diagonals
```

Extracts the diagonal elements of the square (real) global array in a ‘scalable’ fashion, broadcasting the result to everyone. The local array (`diags`) must be large enough to hold the result. The only communication (apart from synchronization to avoid a race condition) is a global sum of length the diagonal.

`ga_maxelt`

```
subroutine ga_maxelt(g_a, value)
  integer g_a           ! [input] GA handle
  double precision value ! [output] abs max value
```

Returns the absolute value of the element with largest absolute magnitude. The only communication is (apart from synchronization to avoid a race condition) is a global maximum of unit length.

`ga_ran_fill`

```
subroutine ga_ran_fill(g_a, ilo, ihi, jlo, jhi)
  integer g_a           ! [input] GA handle
  integer ilo, ihi, jlo, jhi ! [input] patch specification
```

Fills a patch of a global array (`a(ilo:ihi, jlo:jhi)`) with random numbers uniformly distributed between 0 and 1. The only communication is necessary synchronization.

ga_screen

```
subroutine ga_screen(g_a, value)
  integer g_a          ! [input] GA handle
  double precision value ! [input] Threshold
```

Set all elements whose absolute value is less than value to a hard zero. The only communication is necessary synchronization.

ga_mat2col **and** ga_col2mat

```
subroutine ga_mat2col( g_a, ailo, aihi, ajlo, ajhi,
&  g_b, bilo, bihi, bjlo, bjhi)
  integer g_a
  integer g_b
  integer ailo, aihi, ajlo, ajhi
  integer bilo, bihi, bjlo, bjhi

subroutine ga_col2mat( g_a, ailo, aihi, ajlo, ajhi,
&  g_b, bilo, bihi, bjlo, bjhi)
  integer g_a
  integer g_b
  integer ailo, aihi, ajlo, ajhi
  integer bilo, bihi, bjlo, bjhi
```

Obsolete routines to copy patches with reshaping. Use `ga_copy_patch` instead.

4.3.2 Linear algebra and transformations

ga_mix

```
subroutine ga_mix(g_a, n, nvec, b, ld)
  integer g_a          [input]
  integer n, nvec, ld  [input]
  double precision b(ld,nvec) [input]
```

This routine is set up to optimize the rotation of a (small) set of vectors among themselves. The matrix ($A(n, n_{vec})$) referenced by GA handle `g_a` must be distributed by columns so that an entire row is present on a processor — a fatal error results if this is not the case. The matrix `b` must be replicated. With these conditions no communication is necessary, other than that required for synchronizations to avoid race conditions. The routine performs the following operation

$$A_{ij} \leftarrow \sum_{l=1, n_{vec}} A_{il} B_{lj}, i = 1, n; j = 1, n_{vec}$$

which can be regarded as a multiplication of two matrices, one global and the other local, with the result overwriting the input global matrix.

It would be easy to make this routine use more general distributions but still leave the optimized code for column-wise distribution.

two_index_transf

```
subroutine two_index_transf( g_a, g_lhs, g_rhs, g_tmp, g_b )
integer g_a          ! [input] Handle to initial GA
integer g_lhs, g_rhs ! [input] Handles to transformation
integer g_tmp        ! [input] Handle to scratch GA
integer g_b          ! [input] Handle to output GA
```

Two-index square matrix transform — $B = U_{LHS}^T A U_{RHS}$. Done using calls to `ga_dgemm`. The scratch array must be a square array of the same dimension as all the other arrays. It would be easy (and very useful) to generalize this to handle non-square transformations.

ga_matpow

```
subroutine ga_matpow(g_v, pow, mineval)
integer g_v          ! [input/output] Handle to GA
double precision pow ! [input] Exponent
double precision mineval ! [input] Threshold for evals
```

The square matrix referenced by `g_v` is raised to the power `pow` by diagonalizing it, discarding (if `pow` is less than zero) eigenvectors whose eigenvalue is smaller than `mineval`, raising the diagonal matrix to the required power, and transforming back. The only allowed values for `pow` are 1, -1, $\frac{1}{2}$, and $-\frac{1}{2}$, though it would be easy to generalize the routine to handle any value.

The input GA is overwritten with the exponentiated result. It is *not* guaranteed that the same handle will be returned — if it is most efficient, the original GA may be destroyed and a new GA created to hold the result.

Uses a GA the size of `V` and a local array the size of the number of rows of `V`. The eigensolver requires additional memory.

Due to the use of a generalized eigensolver, an additional GA the size of `V` is also used.

mk_fit_xf

```
logical function mk_fit_xf(approx, split, basis, mineval, g_v)
character*(*) approx, split [input]
integer basis                [input]
integer g_v                  [output]
double precision mineval     [input]
```

Returns in `g_v` a newly allocated global array containing the appropriate fitting matrix for the specified resolution-of-the-identity (RI) approximation.

Arguments:

- `approx` — RI approximation used (SVS, S, or V)
- `split` — Whether or not to return the square root of the matrix so that it can be used to transform both sets of 3c ints. (Y or N).
- `basis` — Handle to fitting basis

- `mineval` — Minimum eigenvalue of V matrix to be retained in the inversion
- `g_v` — Returns new global array handle to the $V^{-1/2}$ matrix

Return value:

- `.true.` if successful, even if some eigenvalues fell below `mineval`.
- `.false.` if errors occurred in dynamic memory (MA or GA) operations, inquiries about the basis, or in obtaining the required integrals.

Note: the integral package must be initialized before calling this routine.

Memory use:

- Creates and returns a global array (`g_v`) the size of `bas_numbf(basis)2`.
- Additional temporary usage consists of the largest of:
 1. Integral requirements, reported by `int_mem_2e2c`.
 2. `bas_numbf(basis)2 + bas_numbf(basis)` and whatever additional space is required by `ga_diag_std`.
 3. `2 * bas_numbf(basis)2`.

ga.orthog

```
subroutine ga_orthog(g_vecs, g_over, ometric)
integer g_vecs ! [input] Vectors to be orthonormalized
integer g_over ! [input] Optional metric/overlap matrix
logical ometric ! [input] If .true. use metric matrix
```

The columns of the GA referenced by the handle `g_vecs` are assumed to be vectors that must be orthonormalized. If `ometric` is specified as `.false.` then the standard inner product is used. Otherwise the `g_over` is assumed to refer to the metric (or overlap). Internally, MA is used to allocate a copy of the matrix (and the metric) in a specific distribution. If insufficient memory is available or the matrix is singular a fatal error results.

ga.orthog.vec

```
subroutine ga_orthog_vec(n, nvec, g_m, g_x, j)
integer n ! vector length
integer nvec ! no. of vectors
integer g_m ! GA handle for matrix
integer g_x ! GA handle for vector
integer j ! Column for vector
```

Orthogonalize the vector `x(1:n, j)` to the vectors `g(1:n, 1:nvec)`. Note that `x` is *not* normalized. This routine is/was used by some of the iterative equation solvers.

4.3.3 Iterative linear algebra operations

ga_iter_diag

```

logical function ga_iter_diag(n, nroot, maxiter, maxsub, tol,
&   precondition, product, oprint, eval0, g_evec, eval, rnorm, iter)
integer n                ! Matrix dimension
integer nroot           ! No. of eigen vectors sought
integer maxiter         ! Max. no. of iterations
integer maxsub          ! Max. dimension of iterative subspace
double precision tol    ! Required norm of residual
external precondition   ! Preconditioner
external product        ! Matrix-vector product
logical oprint          ! Control printing to unit 6
double precision eval0  ! Estimate of lowest eval
integer g_evec          ! n by nroot GA for guess and final
double precision eval(nroot) ! Returns eigen values
double precision rnorm(nroot) ! Returns residual norms
integer iter            ! Returns no. of iterations used

```

Solve the eigenvalue equation $\mathbf{A}x = \lambda x$ with the vectors x in GA and a routine (product) to form a matrix vector product to a required precision. Return .true. if converged, .false. otherwise. The function rnorm returns the actual attained precision for each root.

The block-Davidson-like algorithm solves for the best solution for each eigenvector in the iterative subspace $(x_i, i = 1, k)$ with

$$\mathbf{A}y = \mathbf{S}y\lambda, \text{ where } A_{ij} = x_i^\dagger \mathbf{A}x_j, \text{ and } S_{ij} = x_i^\dagger x_j$$

Note: The matrix vector products $\mathbf{A}x_i$ are performed by the user- provided routine product () to a precision specified by this routine (currently products are performed one at a time, but it is easy to improve the routine to perform many in one call).

The best solution within the iterative subspace is then

$$x = \sum_i y_i x_i$$

New expansion vectors are added by multiplying the residual

$$r = (\mathbf{A} - s\mathbf{I})x, \text{ where } s \text{ is the shift,}$$

with some approximation (P) to the inverse of $\mathbf{A} - s\mathbf{I}$. This preconditioning is performed by the user-provided routine precondition (). If eval0 is a hard zero then the shift (s) is chosen as the current estimate for the eigenvalue that the next update strives to improve. Otherwise the shift is fixed as eval0 which is appropriate for convergence to a known energy spectrum from some poor initial guess.

The program cycles through the lowest nroot roots, updating each that does not yet satisfy the convergence criterion, which is

$$\text{rnorm}(\text{root}) = \|r\| < \text{tol}$$

On input the global array `g_evec(n, nroot)` should contain either an initial guess at the eigen vectors or zeroes. If any vector is zero then random numbers are used.

The use must provide these routines:

- subroutine `product(precision, g_x, g_ax)` — computes the product $\mathbf{A}x$ to the specified precision (absolute magnitude error in any element of the product) returning the result in the GA `g_ax`.
- subroutine `precond(g_r, shift)` — Apply an approximation (P) to the inverse of $\mathbf{A} - s\mathbf{I}$ to the vector in `g_r` overwriting `g_r` with the result.

If the initial guess is zero no redundant matrix product is formed.

Temporary global arrays of dimension `n*maxsub` and `n` are created.

`ga_iter_ksolve`

`ga_iter_orthog`

`ga_iter_project`

4.3.4 Miscellaneous

`ga_pcg_minimize`

`int_1e_ga`

`int_2c_ga`

4.4 Symmetry

The symmetry functionality is intended to work for both molecular and periodic systems, so bits and pieces will change over time as the capability to solve periodic systems is developed in the code. All of the the symmetry information is buried in the geometry object, so unless changes are required in the orbitals or basis, the geometry handle alone is sufficient to obtain all required information.

4.4.1 Symmetry Group and Operations Functions

The symmetry module functions that define the group name and operations are described in this section.

`sym_group_name`

```
subroutine sym_group_name(geom, name)
integer geom           ! [input]
character*(*) name    ! [output] returns the group name
```

sym_number_ops

```
integer function sym_number_ops(geom)
integer geom          ! [input]
```

This routine returns the number of operations in the group, *excluding* the identity. Thus, C_1 is represented as containing zero operators and C_{2v} has three operators.

sym_center_map

```
integer function sym_center_map(geom, cent, op)
integer geom          ! [input]
integer cent          ! [input] Geometrical center
integer op            ! [input] Operator
```

Returns the index of the center that the input center (*cent*) maps into under the action of the operator (numbered 1, ..., *sym_number_ops(geom)*).

sym_inv_op

```
subroutine sym_inv_op(geom, op, opinv)
integer geom          ! [input]
integer op            ! [input] Operator number
integer opinv        ! [output] Inverse operator
```

Returns in *opinv* the index of the operator that is the inverse to the operator *op*.

sym_apply_op

```
subroutine sym_apply_op(geom, op, r, rnew)
integer geom
integer op
double precision r(3)
double precision rnew(3)
```

This routine applies the operator *op* to the 3-vector *r* returning the result in *rnew*. *Note that this routine acts on coordinates natural to the system — Cartesian for molecules and fractional for periodic systems.*

sym_apply_cart_op

```
subroutine sym_apply_cart_op(geom, op, r, rnew)
integer geom
integer op
double precision r(3)
double precision rnew(3)
```

This routine applies the operator *op* to the Cartesian 3-vector *r* returning the result in *rnew*. *Note that this routine acts only on Cartesian coordinates.*

sym_get_cart_op

```
subroutine sym_get_cart_op(geom, op, matrix)
integer geom          ! [input]
integer op            ! [input] Operator
double precision matrix(3,4) ! [output] Returns cartesian operator
```

This routine returns the matrix representation of the operator that acts on Cartesian coordinates. The first three columns correspond to the point group operator and the final column is the translation.

$$OP * r(1:3) = r'(1:3) = matrix(1:3,1:3)*r(1:3) + matrix(1:3,4)$$

sym_ops_get

```
subroutine sym_ops_get(geom, numops, symops)
integer geom          ! [input]
integer numops        ! [input] Leading dim. of symops
double precision symops(numops*3,4) ! [input] Returns operators
```

Returns in `symops` the first `numops` operators. It's probably not necessary to use this routine.

sym_op_mult_table

```
subroutine sym_op_mult_table(geom, table, ld)
integer geom          ! [input]
integer ld
integer table(ld,*)
c
c !! THIS ROUTINE HAS NOT BEEN COMPILED OR TESTED !!
c
```

This routine returns in `table` the multiplication table for the operators excluding the identity — inside the table the identity is labelled as zero.

4.4.2 Geometries and Gradients

The two routines `sym_geom_project` and `sym_grad_symmetrize` actually do exactly the same thing internally, but the interface differs according to their natural usage.

sym_geom_project

```
subroutine sym_geom_project(geom, tol)
integer geom          ! [input]
double precision tol  ! [input]
```

This routine applies a projection operator to the geometry so that it possesses the symmetry of the group, to machine precision. An atom and the image of that atom under the operations of the group are considered to be identical if (and

only if) they are less than `tol` distant from each other. If two centers that should be symmetry equivalent differ by more than `tol`, then a fatal error results. This operation should be idempotent; that is, the square of the projection operator, P_a , is equal to the operator itself, P_a .

`sym_grad_symmetrize`

```
subroutine sym_grad_symmetrize(geom, grad)
integer geom          ! [input]
double precision grad(3,*) ! [input/output]
```

This routine also applies a projection operator to the gradient so that it possesses the symmetry of the group, to machine precision. This is appropriate for projecting out the totally symmetric component of a gradient constructed from a skeleton integral list. This operation should also be idempotent.

4.4.3 Character Tables

In order to make use of the character table you need to determine the class of each operator. Note that the identity is the only operator in the first class.

`sym_char_table`

```
logical function sym_char_table(zname, nop, nir, class_dim,
&    zir, zclass, chars)
character*8 zname          ! [input]
integer nop                ! [output] Returns no. ops (with identity)
integer nir                ! [output] Returns no. irreducible reps.
integer class_dim(*)      ! [output] Returns dim. of each class
character*8 zir(*)        ! [output] Returns name of each irrep
character*8 zclass(*)     ! [output] Returns name of each class
double precision chars(*) ! [output] Returns the character table
```

Given the name of the group, this routine returns the total number of operators (`nop`) including the identity, the number of irreducible representations (`nir`), the name of each irreducible representation (`zir(i)`, $i=1, \dots, nir$), the dimension and name of each class (`class_dim(i)`, `zclass(i)`, $i=1, \dots, nir$), and the character table. Returns `.true.` if group character table was available, `.false.` otherwise.

The character of class C in irreducible representation R is stored in `char(C,R)` if `char` is dimensioned as

```
double precision char(nir,nir)
```

The maximum number of irreducible representations in any point group is 20 and the maximum number of operators is 120. Thus, you can just paste these declarations into your code to call this routine

```
integer maxop, maxireps
parameter (maxop = 120, maxireps=20)
integer nop, nir, nop_table, class_dim(maxireps)
character*8 zir(maxireps), zclass(maxireps)
double precision chars(maxireps*maxireps)
```

```

if (.not. sym_char_table(zname, nop, nir, class_dim,
$      zir, zclass, chars)) call errquit(' ... ',0)

```

All is simple except for complex conjugate pairs of irreducible representations that are stored with one having the real pieces of the characters and the other the imaginary. This leads to the second having a zero character for the identity, however a valid projection operator can still be constructed (look in `sym_movecs_adapt()`).

`sym_op_classify`

```

subroutine sym_op_classify(geom, op_class_index)
integer geom          ! [input] Geometry handle
integer op_class_index(*) ! [output] Class number of each op

```

Return an array that has for each operator the number of the class to which it belongs. This index makes the connection between the operator and the character table. The operators are numbered, excluding the identity, from 1 to `sym_number_ops()`.

4.4.4 Atomic/Molecular Orbitals

`sym_bas_irreps`

```

subroutine sym_bas_irreps(basis, oprint, nbf_per_ir)
integer basis          ! [input] basis handle
logical oprint        ! [input] if true then print
integer nbf_per_ir(*) ! [output] no. of functions per irr. rep.

```

Returns in `nbf_per_ir` the number of functions per irreducible representation that are present in the specified basis set. The maximum number of irreducible representations in any point group is 20.

`sym_movecs_adapt`

```

subroutine sym_movecs_adapt(basis, thresh, g_vecs, irs, nmixed)
integer basis          ! [input]
double precision thresh ! [input]
integer g_vecs        ! [input]
integer irs(*)        ! [output]
integer nmixed        ! [output]

```

Symmetry adapts the molecular orbitals in the GA `ga_vecs`, returning in `irs(i)` the number of the irreducible representation of the *i*'th molecular orbital. In `nmixed` is returned the number of input molecular orbitals that were symmetry contaminated greater than `thresh`. An MO is deemed contaminated if it contains two or more irreps. with coefficients greater than `thresh`.

Note: If the input MOs are nearly linearly dependent then the output MOs may be exactly linearly dependent since if the component distinguishing two vectors is not the dominant symmetry component it will be projected out. If there is reason to suspect linear dependence, `ga_orthog()` should be called before calling this routine.

Note: If mixing was present it may be necessary to call `ga_orthog()` to reorthogonalize the output vectors.

sym_movecs_apply_op

```
subroutine sym_movecs_apply_op(basis, op, v, t)
integer basis          ! [input]
integer op             ! [input]
double precision v(*)  ! [input]
double precision t(*)  ! [output]
```

Apply the group operation `op` to the vector of basis function coefficients (i.e., a MO vector) in `v(*)`, returning the result in `t(*)`.

sym_bas_op

```
subroutine sym_bas_op(geom, op, r, maxf, ang_max)
integer geom          ! [input]
integer op            ! [input] Desired operator
integer maxf         ! [input] Leading dimension of r
integer ang_max      ! [input] Max. ang. momentum of shell
double precision r(1:maxf,1:maxf,0:ang_max) ! [output] The operator
```

Return the transformation matrices for basis functions up to the specified maximum angular momentum under the specified group operation.

Note that the identity operation is not included.

Note that only cartesian shells are supported, but sphericals will be integrated when available.

Let $X(I, L)$ be the I th function in a shell with angular momentum L . The application of a symmetry operator will map shell X into an equivalent shell on a possibly different center and will also mix up the components of the shell according to

$$R \text{ op } X(I, L) = \text{sum}(J) X(J, L) * R(J, I, L)$$

In dealing with Cartesian functions it is necessary to pay careful attention to the difference between using the inverse of an operator or its transpose (see Dupuis and King, IJQC 11, 613-625, 1977). To apply the inverse operator simply use both the center mapping and transformation matrices of the inverse operator. However, since the representation matrices are *not* unitary in the Cartesian basis then to generate the effect the transposed matrices of an operator you must

- map (atomic or basis function) centers according to the mapping provided for the inverse operation (see section 4.4.1)
- apply the transpose of coefficients (i.e., use $R(I, J, L)$ instead of $R(J, I, L)$ in the above transformation).

For examples of how this routine is used in practice look in `symmetry/sym_mo_adapt.F` or `symmetry/sym_sym.F`.

4.4.5 ‘Skeleton’ integral lists

Note that the constituency number (point group component only) for shells is exactly the same as that for the atoms on which they reside.

`sym_atom_pair`

```

logical function sym_atom_pair(geom, iat, jat, q2)
integer geom          ! [input] Geometry handle
integer iat, jat     ! [input] Atom indices
double precision q2  ! [output] Constituency number

```

Return `.true.` if `(iat, jat)` is the lexically highest pair of symmetry equivalent atoms. If `.true.` also return the constituency factor `q2` (which is the number of symmetry equivalent pairs).

This routine uses the exchange symmetry `iat <-> jat` but does not incorporate any factors into `q2` to account for this (i.e., `q2` includes point group symmetry only).

`sym_atom_quartet` **and** `sym_atom_gen_quartet`

```

logical function sym_atom_quartet(geom, iat, jat, kat, lat, q4)
integer geom          ! [input] Geometry handle
integer iat, jat, kat, lat ! [input] Atom indices
double precision q4  ! [output] Constituency number

```

Return `.true.` if `(iat, jat, kat, lat)` is the lexically highest quartet of symmetry equivalent atoms. If `.true.` also return the constituency factor `q4` (which is the number of symmetry equivalent quartets).

This routine uses the standard three index exchange symmetries `(iat<->jat) <-> (kat<->lat)` but does not incorporate any additional factors into `q4` (i.e., `q4` reflects only the point group symmetry). Look in the `ddscf/` directory for examples of its use.

```

logical function sym_atom_gen_quartet(geom, iat, jat, kat, lat, q4)

```

This routine differs from `sym_atom_quartet` only in that it uses just two index exchange symmetries `(iat<->jat)` and `(kat<->lat)`. Look in the `moints/` directory for examples of its use.

`sym_shell_pair`

```

logical function sym_shell_pair(basis, ishell, jshell, q2)
integer basis          ! Basis set handle [input]
integer ishell, jshell ! Shell indices [input]
double precision q2    ! Constituency number [output]

```

Return `—TRUE` if `(ishell, jshell)` is the lexically highest pair of symmetry equivalent shells. If `.true.` also return the constituency factor `q2` (which is equal to the number of symmetry equivalent pairs).

This routine uses the exchange symmetry `ishell <-> jshell` and *incorporates a factor of two into `q2` to account for this*. However, this factor of two may be removed at some point in order to make the shell based routines exactly consistent with the atom based code.

`sym_shell_quartet`

```

logical function sym_shell_quartet(basis,

```

```
&      ishell, jshell, kshell, lshell, q4)
integer basis          ! Basis set handle [input]
integer ishell, jshell ! Shell indices [input]
integer kshell, lshell ! Shell indices [input]
double precision q4    ! Constituency number [output]
```

Return `.true.` if `(ishell, jshell, kshell, lshell)` is the lexically highest quartet of symmetry equivalent shells. If `.true.` also return the constituency factor `q4` (which is the number of symmetry equivalent quartets).

This routine uses the standard three index exchange symmetries `(ishell<->jshell) <-> (kshell<->lshell)` but does not incorporate any additional factors into `q4` (i.e., `q4` reflects only the point group symmetry). Look in the `ddscf/` directory for examples of its use.

`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
```

Symmetrize a skeleton AO matrix (in global array with handle `g_a`) in the given basis set. This is nothing more than applying the projection operator for the totally symmetric representation.

$$B = (1/2h) * \text{sum}(R) [RT * (A + AT) * R]$$

where `R` runs over all operators in the group (including identity) and `h` is the order of the group.

Note that density matrices transform according to slightly different rules to Hamiltonian matrices if components of a shell (e.g., cartesian `d`'s) are not orthonormal. (see Dupuis and King, IJQC 11, 613-625, 1977). Hence, specify `odensity` as `.true.` for density-like matrices and `.false.` for all other totally symmetric Hamiltonian-like operators.

4.4.6 Printing Symmetry Information

`sym_print_all`

```
subroutine sym_print_all(geom, oinfo, ouniq, omap, oops, ochar)
integer geom          ! [input]
logical oinfo        ! [input] print information
logical ouniq        ! [input] print list of unique atoms
logical omap         ! [input] print mapping of atoms under ops
logical oops         ! [input] print operator matrices
logical ochar        ! [input] print character table
```

Print out all symmetry related information inside the geometry object

`oinfo` — prints the name and order of the group

`ouniq` — prints the list of symmetry unique atoms

omap — prints the transformation of atoms under group operations

oops — prints the matrix representation of operators including class information

ochar — prints the character table

sym_print_char_table

```
subroutine sym_print_char_table(geom)
integer geom          ! [input]
```

Print the character table for the group to Fortran unit 6.

sym_print_ops

```
subroutine sym_print_ops(geom)
integer geom          ! [input]
```

Called by `sym_print_all` to print the operators. You can call it too if you like.

4.4.7 Internal symmetry stuff that might be useful

sym_op.type

sym_op.class_name

4.4.8 Miscellaneous

cross_product

deter3

Chapter 5

Integral Application Programmer's Interface

The integral (INT) Application Programmer's Interface (API) is the interface to the base integral technology available in the NWChem application software. The INT-API interfaces currently three integral codes, the sp rotated axis code (from GAMESS-UK), the McMurchie-Davidson code (PNNL, Stave, Früchtl, and Kendall), and the Texas 93/95 Integral code (Wolinski and Pulay). The API is currently limited to the requisite functionality of NWChem. Further functionality will be added over time as requirements are determined, prioritized and implemented.

5.1 Overview

The integral code operates as a single threaded suite and all parallelization is achieved at the level of the routines that call the API or above. The API requires a collective initialization phase to determine operating parameters for the particular run based on both user input and the basis set specification. The API will select the appropriate base integral code for the requested integrals at the time of each request. Once all integral computations have completed for the module the termination routines should be called (in a collective fashion).

Coupled initialization and termination can be executed as many times as required. It is imperative that the basis set object, ECP object, and the geometry object are constant between initialization and termination, e.g., normalization must occur prior to initialization. If this data must be modified then a termination and re-initialization of the integral API is *required*.

The INT-API has the following kinds of routines:

- initialization, integral accuracy and termination,
- memory requirements,
- integral routines (both shell based and blocked),
- derivative integral routines,
- property integral routines,
- periodic integral routines,
- Internal API Routines

Details of the API specification are in appendix A.

5.2 Adding a new base integral code to the NWChem INT-API

This is a straightforward but non-trivial task. Requirements include a set of APIs for the base integral code to marry it to the NWChem style. The computation of integral batches (e.g., in shell quartets or groups of shell quartets, i.e., blocks) must be autonomous and use a scratch buffer passed at the time integral batch request for unified memory management. Any precomputation must be done in the initialization phase and stored for later use. The initialization routines must be based on using the NWChem basis set and geometry data. This may be translated and stored for later use in the base integral code format but it must not require significant amounts of memory. A memory estimate routine that tells the application code the amount of scratch memory and buffer memory that is required. This should be dynamic in nature and not be a fixed dimension. In other words, the memory utilization should scale with the size of the problem. Termination routines should completely cleanup all temporary memory storage that is done in the Memory Allocator.

Chapter 6

Software Development Toolkit

The Software Development Toolkit is the foundation of the functional architecture in NWChem. It consists of various useful elements for memory management and data manipulation that are needed to facilitate the development of parallel computational chemistry algorithms. The memory management elements implement the NUMA memory management module for efficient execution in parallel environments and provides the means for interfacing between the calculation modules of the code and the system hardware. Efficient data manipulation is accomplished using the runtime data base, which stores the information needed to run particular calculations and allows different modules to have access to the same information. This chapter describes the various elements of the Software Development Toolkit in detail.

6.1 Non-Uniform Memory Allocation (NUMA)

All computers have several levels of memory, with parallel computers generally having more than computers with only a single processor. Typical memory levels in a parallel computer include the processor registers, local cache memory, local main memory, and remote memory. If the computer also supports virtual memory, local and remote disk memory are added to this hierarchy. These levels vary in size, speed, and method of access, and in NWChem the differences among them are lumped under the general concept Non-Uniform Memory Access (NUMA). This approach allows the developer to think of all memory anywhere in the system as accessible to any processor as needed. It is then possible to focus independently on the questions of memory access methods and memory access costs. Memory access methods are determined by the programming model and available tools and the desired coding style for an application. Memory access costs are determined by the program structure and the performance characteristics of the computer system. The design of a code's major algorithms, therefore, is critical to the creation of an efficient parallel program.

In order to scale to massively parallel computer architectures in all aspects of the hardware (i.e., CPU, disk, and memory), NWChem uses Non-Uniform Memory Access to distribute the data across all nodes. Memory access is achieved through explicit message passing using the TCGMSG interface. The Memory Allocator (MA) tool is used to allocate memory that is local to the calling process. The Global Arrays (GA) tool is used to share arrays between processors as if the memory were physically shared. The complex I/O patterns required to accomplish efficient memory management are handled with the abstract programming interface ChemIO.

The following subsections discuss the TCGMSG message passing tool, the Memory Allocator library, the Global Arrays library, and ChemIO, and describe how they are used in NWChem.

6.1.1 Message Passing

TCGMSG¹ is a toolkit for writing portable parallel programs using a message passing model. It is relatively simple, having limited functionality that includes point-to-point communication, global operations, and a simple load-balancing facility, and was designed with chemical applications in mind. This simplicity contributes to the robustness of TCGMSG and its exemplary portability, and also to its high performance for a wide range of problem sizes.

The model used by TCGMSG operates as if it is sending a block until the message is explicitly received, and the messages from a particular process can be received only in the order sent. Processes should be thought of as being connected with ordered synchronous channels, even though messages are actually sent without any synchronization between sender and receiver, so far as buffering permits. The amount of buffering is greatly dependent on the mechanism used by the particular platform, so it is best not to count on this feature. Detailed information that includes documentation of the programming interface is available on-line as part of the EMSL webpage, at

<http://www.emsl.pnl.gov:2080/docs/parsoft/tcgmsg/>

A more general tool for message passing is MPI, which includes concepts such as process groups, communication contexts, and virtual topologies. Process groups can be used to specify that only certain processes are involved in a particular task, or to allow separate groups of processes to work on different tasks. Communication context provides an additional criterion for message selection, enhancing internal communication flexibility without incurring conflicts with other modules. MPI has been implemented in NWChem as an alternative to TCGMSG, and the code can be compiled with this option specified. However, it is not an undertaking for the faint of heart and it is highly advisable to contact nwchem-support@emsl.pnl.gov before trying this option.

The TCGMSG-MPI library is distributed with the Global Arrays package. This library is an implementation of the TCGMSG message passing interface on top of MPI and system-specific resources. Using this library, it is possible to use both MPI and TCGMSG interfaces in the same application. TCGMSG offers a much smaller set of operations than MPI, but these include some unique capabilities, such as

- `nxtval` - a shared memory counter with atomic updates, often used in dynamic load balancing operations
- `plcopy` - function to copy content of a sequential file to all processes
- `mitoh`, `mdtoh`, etc. - portable Fortran equivalents of the C `sideol` operator

The `nxtval` operation is implemented in TCGMSG-MPI in different ways, depending on the platform.

- SGI Origin-X - shared memory and mutexes or semaphores
- IBM SP
 - under MPL - interrupt receive
 - under LAPI communication library – atomic read-modify-write
 - under thread-safe MPI – atomic read-modify-write
- Intel NX - interrupt receive, with signal-based implementation of the MPI library
- Cray T3D/E – SHMEM library
- Fujitsu VX/VPP – MPLib

¹R.J. Harrison, Int. J. Quantum Chem., Vol. 40, Issue 337, 1991. The toolkit is in the public domain and is distributed with the Global Arrays package.

- server implementation using dedicated MPI process

Detailed information that includes documentation of the programming interface is available on-line as part of the EMSL webpage, at

<http://www.emsl.pnl.gov:2080/docs/parsoft/tcgmsg-mpi/>

6.1.2 Memory Allocator (MA)

The Memory Allocator (MA) is used to allocate data that will generally not be directly shared with other processes, such as workspace for a particular local calculation or for replication of very small sets of data. The MA tool is a library of routines that comprises a dynamic memory allocator for use by C, FORTRAN, or mixed-language applications. It provides both heap and stack memory management disciplines, debugging and verification support (for detecting memory leaks, for example), usage statistics, and quantitative memory availability information.

Applications written in FORTRAN require this sort of library because the language does not support dynamic memory allocation. Applications written in C can benefit from using MA instead of the ordinary `malloc()` and `free()` routines because of the extra features MA provides, which include both heap and stack memory management disciplines, debugging and verification support, usage statistics, and quantitative memory availability information. MA is designed to be portable across a large variety of platforms.

Detailed information on specific routines is available in the MA man pages. This can be accessed by means of the command, `man ma`. (Note: this will work only if the local environmental variable `MANPATH` includes the path `$(NWCHEM_TOP)/src/man/ma/man`. See Section 8.3 for information on system and environmental requirements for running NWChem.) The following subsections present a summary list of the MA routines, and a brief discussion of the implementation of this feature.

MA Data Types

All MA memory must be explicitly assigned a specific type by defining each data item in units of integer, logical, double precision, or character words. The type of data is specified in arguments using predefined Fortran parameters (or macros in C). These parameters are available in the include files `mafdecls.fh` in Fortran and in `macdecls.h` in C. The parameters are typed as follows:

`MT_INT` — integer

`MT_DBL` — double precision

`MT_LOG` — logical

`MT_CHAR` — character*1

Implementation

To access required MA definitions, C applications should include `macdecls.h` and FORTRAN applications should include `mafdecls.fh`. These are public header files for a dynamic memory allocator, and are included in the `.../src/ma` subdirectory of the NWChem directory tree. The files contain the type declarations and parameter declarations for the datatype constants, and define needed functions and variable types.

The memory allocator uses the following memory layout definitions:

- segment = heap_region stack_region
- region = block block block ...
- block = AD gap1 guard1 client_space guard2 gap2

A segment of memory is obtained from the OS upon initialization. The low end of the segment is managed as a heap. The heap region grows from low addresses to high addresses. The high end of the segment is managed as a stack. The stack region grows from high addresses to low addresses.

Each region consists of a series of contiguous blocks, one per allocation request, and possibly some unused space. Blocks in the heap region are either in use by the client (allocated and not yet deallocated) or not in use by the client (allocated and already deallocated). A block on the rightmost end of the heap region becomes part of the unused space upon deallocation. Blocks in the stack region are always in use by the client, because when a stack block is deallocated, it becomes part of the unused space.

A block consists of the client space, i.e., the range of memory available for use by the application. Guard words adjacent to each end of the client space to help detect improper memory access by the client. Bookkeeping information is stored(?) in an "allocation descriptor" AD Two gaps, each zero or more bytes long, are defined to satisfy alignment constraints (specifically, to ensure that AD and client_space are aligned properly).

List of MA routines

All MA routines are shown below, grouped by category and listed alphabetically within each category. The FORTRAN interface is given here. Information on the the C interface are available in the man pages. (The man pages also contain more detailed information on the arguments for these routines.)

Initialization:

- MA_init(datatype, nominal_stack, nominal_heap)
 - integer datatype
 - integer nominal_stack
 - integer nominal_heap
- MA_sizeof(datatype1, neleml, datatype2)
 - integer datatype1
 - integer neleml
 - integer datatype2
- MA_sizeof_overhead(datatype)
 - integer datatype
- MA_initialized()

Allocation:

- MA_alloc_get(datatype, neleml, name, memhandle, index)
 - integer datatype
 - integer neleml

- character*(*) name
- integer memhandle
- integer index
- MA_allocate_heap(datatype, nelem, name, memhandle)
 - integer datatype
 - integer nelem
 - character*(*) name
 - integer memhandle
- MA_get_index(memhandle, index)
 - integer memhandle
 - integer index
- MA_get_pointer() — C only
- MA_inquire_avail(datatype)
 - integer datatype
- MA_inquire_heap(datatype)
 - integer datatype
- MA_inquire_stack(datatype)
 - integer datatype
- MA_push_get(datatype, nelem, name, memhandle, index)
 - integer datatype
 - integer nelem
 - character*(*) name
 - integer memhandle
 - integer index
- MA_push_stack(datatype, nelem, name, memhandle)
 - integer datatype
 - integer nelem
 - character*(*) name
 - integer memhandle

Deallocation:

- MA_chop_stack(memhandle)
 - integer memhandle
- MA_free_heap(memhandle)

- integer memhandle
- `MA_pop_stack(memhandle)`
 - integer memhandle

Debugging:

- `MA_set_auto_verify(value)`
 - logical value
 - integer ivalue
- `MA_set_error_print(value)`
 - logical value
 - integer ivalue
- `MA_set_hard_fail(value)`
 - logical value
 - integer ivalue
- `MA_summarize_allocated_blocks`
- `MA_verify_allocator_stuff()`

Iteration Over Allocated Blocks:

- `MA_get_next_memhandle(ithandle, memhandle)`
 - integer ithandle
 - integer memhandle
- `MA_init_memhandle_iterator(ithandle)`
 - integer ithandle

Statistics:

- `MA_print_stats(oprintroutines)`
 - logical printroutines

MA Errors

Errors considered fatal by MA result in program termination. Errors considered nonfatal by MA cause the MA routine to return an error value to the caller. For most boolean functions, false is returned upon failure and true is returned upon success. (The boolean functions for which the return value means something other than success or failure are `MA_set_auto_verify()`, `MA_set_error_print()`, and `MA_set_hard_fail()`.) Integer functions return zero upon failure; depending on the function, zero may or may not be distinguishable as an exceptional value.

An application can force MA to treat all errors as fatal via `MA_set_hard_fail()`.

If a fatal error occurs, an error message is printed on the standard error (stderr). By default, error messages are also printed for nonfatal errors. An application can force MA to print or not print error messages for nonfatal errors via `MA_set_error_print()`.

6.1.3 Global Arrays (GA)

Globally addressable arrays have been developed to simplify writing portable scientific software for both shared and distributed memory computers. Programming convenience, code extensibility and maintainability are gained by adopting the shared memory programming model. The Global Array (GA) toolkit provides an efficient and portable "shared memory" programming interface for distributed memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed matrices without need for explicit cooperation by other processes. The trade-off with this approach is that access to shared data will be slower than access to local data, and the programmer must be aware of this in designing modules.

From the user perspective, a global array can be used as if it was stored in the shared memory. Details of the data distribution, addressing and communication are encapsulated in the global array objects. However, the information on the actual data distribution can be obtained and taken advantage of whenever data locality is important.

The Global Arrays tool has been designed to complement the message-passing programming model. The developer can use both shared memory and message passing paradigms in the same program, to take advantage of existing message-passing software libraries such as TCGMSG. This tool is also compatible with the Message Passing Interface (MPI). The Global Arrays toolkit has been in the public domain since 1994 and is actively supported. Additional documentation and information on performance and applications is available on the web site <http://www.emsl.pnl.gov:2080/docs/global/>.

Currently support is limited to 2-D double precision or integer arrays with block distribution, at most one block per array per processor.

Interaction Between GA and MA

Available global (GA) and local (MA) memory can interact within NWChem in only two ways,

1. GA is allocated within MA, and GA is limited only by the available space in MA.
2. GA is not allocated within MA, and GA is limited at initialization (within NWChem input this is controlled by the MEMORY directive)

If GA is allocated within MA, then the available GA space is limited to the currently available MA space. This also means that the total allocatable memory for GA *and* MA must be no more than the available MA space. If GA is not allocated within MA, then local and global arrays occupy essentially independent space. The allocatable memory for GA is limited only by the available space for GA, and similarly, the allocatable memory for MA is limited only by the available local memory.

When allocating space for GA, some care must be exercised in the treatment of the information returned by the routine `ga_memory_avail()`, whether or not the allocation is done in MA. The routine `ga_memory_avail()` returns the amount of memory (in bytes) available for use by GA in the calling process. This returned value must be converted to double precision words when using double precision. If a uniformly distributed GA is desired, it is also necessary to find the minimum of this value across all nodes. This value will in general be a rather large number. When running on a platform with many nodes and having a large memory, the aggregate GA memory, even in double precision words, could be a large enough value to overflow a 32-bit integer. Therefore, for calculations that require knowing the size of total memory, it is advisable to first store the size of memory on each node in a double precision number and then sum these values across all the nodes.

The following pseudo-code illustrates this process for an application.

```
#include "global.fh"
#include "mafdecls.fh"
```

```

integer avail_ma, avail_ga

avail_ma = ma_inquire_avail(mt_dbl)
avail_ga = ga_memory_avail()/ma_sizeof(mt_dbl,1,mt_byte)

if (ga_uses_ma()) then
c
c available GA space is limited to currently available MA space,
c and GA and MA share the same space
c
    allocatable_ga + allocable_ma <= avail_ma = avail_ga

else
c
c GA and MA are independent
c
    allocatable_ga <= avail_ga
    allocatable_ma <= avail_ma

endif

c
c find the minimum value of available GA space over all nodes
c
    call ga_igop(msgtype,avail_ga,1,'min')
c
c determine the total available GA space
c
double precision davail_ga
davail_ga = ga_memory_avail()/ma_sizeof(mt_dbl,1,mt_byte)
call ga_dgop(msgtype,davail_ga,1, '+')

```

List of GA Routines

The following routines are invoked for operations that are globally collective. That is, they must be simultaneously invoked by all processes as if in SIMD mode.

- `ga_initialize()` — initialize global array internal structures
- `ga_initialize_ltd(mem_limit)` — initialize global arrays and set memory usage limits (note: if `mem_limit` is less than zero specifies unlimited memory usage.)
 - integer `mem_limit` — [input] GA total memory (specifying less than 0 means "unlimited memory")
- `ga_create(type,dim1,dim2,array_name,chunk1,chunk2,g_a)` — create an array
 - integer `type` — [input] MA type
 - integer `dim1, dim2` — [input] array dimensions (`dim1,dim2`) as in FORTRAN
 - character `array_name` — [input] unique character string identifying the array

- integer `chunk1`, `chunk2` — [input] minimum size that dimensions should be chunked up into; setting `chunk1=dim1` gives distribution by rows setting `chunk2=dim2` gives distribution by columns Actual chunk sizes are modified so that they are at least the min size and each process has either zero or one chunk. (Specifying both as less than or equal to 1 yields an even distribution)
- integer `g_a` [output] integer handle for future references
- `ga_create_irreg(type, dim1, dim2, array_name, map1, nblock1, map2, nblock2, g_a)` — create an array with irregular distribution
 - integer `type` — [input] MA type
 - integer `dim1`, `dim2` — [input] array dimensions (`dim1,dim2`) as in FORTRAN
 - character `array_name`— [input] unique character string identifying the array
 - integer `map1` — [input] number ilo in each block
 - integer `nblock1` — [input] number of blocks `dim1` is divided into
 - integer `map2` — [input] number jlo in each block
 - integer `nblock2` — [input] number of blocks `dim2` is divided into
 - integer `g_a` — [output] integer handle for future references
- `ga_duplicate(g_a, g_b, array_name)` — create an array with same properties as reference array
 - character `array_name` — [input] unique character string identifying the array
 - integer `g_a` — [output] integer handle for reference array
 - integer `g_b` — [output] integer handle for new array
- `ga_destroy_(g_a)` — destroy an array
 - integer `g_a` — [input] integer handle of array to be destroyed
- `ga_terminate_()` — destroys all existing global arrays and de-allocates shared memory
- `ga_sync_()` — synchronizes all processes (a barrier)
- `ga_zero_(g_a)` — zero an array
 - integer `g_a` — [input] integer handle of array to be zeroed
- `ga_ddot_(g_a, g_b)` — dot product of two arrays (double precision only)
 - integer `g_a` — [input] integer handle of first array in dot product
 - integer `g_b` — [input] integer handle of second array in dot product
- `ga_dscal` — scale the elements in an array by a constant (double precision data only)
- `ga_dadd` — scale and add two arrays to put result in a third (may overwrite one of the other two, doubles only)
- `ga_copy(g_a, g_b)` — copy one array into another
 - integer `g_a` — [input] integer handle of array to be copied
 - integer `g_b` — [input] integer handle of array `g_a` is copied into
- `ga_dgemm(transa, transb, m, n, k, alpha, g_a, g_b, beta, g_c)` — BLAS-like matrix multiply
 - character*1 `transa`, `transb`

- integer `m`, `n`, `k`
 - double precision `alpha`, `beta`
 - integer `g_a`, `g_b`, `g_c`
- `ga_ddot_patch(g_a, t_a, ailo, aihi, ajlo, ajhi, g_b, t_b, bilo, bihi, bjlo, bjhi)`
— dot product of two arrays (double precision only; patch version) (Note: patches of different shapes and distributions are allowed, but not recommended, and both patches must have the same number of elements)
 - integer `g_a` — [input] integer identifier of first array containing patch for dot product
 - integer `t_a` — [input] transpose of first array
 - integer `ailo`, `aihi` — [input] high and low indices for *i* dimension of patch of array for dot product
 - integer `ajlo`, `ajhi` — [input] high and low indices for *j* dimension of patch of array for dot product
 - integer `g_b` — [output] integer identifier of second array containing patch for dot product
 - integer `t_b` — [input] transpose of second array
 - integer `bilo`, `bihi` — [input] high and low indices for *i* dimension of patch of array for dot product
 - integer `bjlo`, `bjhi` — [input] high and low indices for *j* dimension of patch of array for dot product
 - `ga_dscal_patch` — scale the elements in an array by a constant (patch version)
 - `ga_dadd_patch` — scale and add two arrays to put result in a third (patch version)
 - `ga_ifill_patch` — fill a patch of array with value (integer version)
 - `ga_dfill_patch` — fill a patch of array with value (double version)
 - `ga_matmul_patch(transa, transb, alpha, beta, g_a, ailo, aihi, ajlo, ajhi, g_b, bilo, bihi, bjlo, bjhi, g_c, cilo, cihi, cjlo, cjhi)` — matrix multiply (patch version)
 - character `transa` — [input] transpose of first array for matrix multiply
 - character `transb` — [input] transpose of second array for matrix multiply
 - double precision `alpha` — ??
 - double precision `beta` — ??
 - integer `g_a` — [input] integer identifier of first array for matrix multiply
 - integer `ailo`, `aihi` — [input] high and low indices for *i* dimension of patch of first array for matrix multiply
 - integer `ajlo`, `ajhi` — [input] high and low indices for *j* dimension of patch of first array for matrix multiply
 - integer `g_b` — [input] integer identifier of second array for matrix multiply
 - integer `bilo`, `bihi` — [input] high and low indices for *i* dimension of patch of second array for matrix multiply
 - integer `bjlo`, `bjhi` — [input] high and low indices for *j* dimension of patch of second array for matrix multiply
 - integer `g_c` — [input] integer identifier of resultant array for matrix multiply
 - integer `cilo`, `cihi` — [input] high and low indices for *i* dimension of patch of resultant array for matrix multiply
 - integer `cjlo`, `cjhi` — [input] high and low indices for *j* dimension of patch of resultant array for matrix multiply

- `ga_diag(g_a, g_s, g_v, eval)` — real symmetric generalized eigensolver (sequential version `ga_diag_seq` also exists)
 - integer `g_a` — matrix to diagonalize
 - integer `g_s` — metric
 - integer `g_v` — global matrix to return evecs
 - double precision `eval(*)` — local array to return evals
- `ga_diag_reuse(reuse, g_a, g_s, g_v, eval)` — a version of `ga_diag` for repeated use
 - integer `reuse` — allows reuse of factorized `g_s`: flag is 0 first time, greater than 0 for subsequent calls, less than 0 deletes factorized `g_s`
 - integer `g_a` — matrix to diagonalize
 - integer `g_s` — metric
 - integer `g_v` — global matrix to return evecs
 - double precision `eval(*)` — local array to return evals
- `ga_diag_std(g_a, g_v, eval)` — standard real symmetric eigensolver (sequential version also exists)
 - integer `g_a` — [input] matrix to diagonalize
 - integer `g_v` — [output] global matrix to return evecs
 - double precision `eval(*)` — [output] local array to return evals
- `ga_symmetrize(g_a)` — symmetrizes matrix A into $0.5(A+A')$ (NOTE: $\text{diag}(A)$ remains unchanged.)
 - integer `g_a` — [input] matrix to symmetrize
- `ga_transpose(g_a)` — transpose a matrix
 - integer `g_a` — [input] matrix to transpose
- `ga_lu_solve(trans, g_a, g_b)` — solves system of linear equations based on LU factorization (sequential version `ga_lu_solve_seq` also exists)
 - character*1 `trans` — [input] transpose or not
 - integer `g_a` — [input] matrix to diagonalize (coefficient matrix A)
 - integer `g_b` — [output] rhs matrix B , overwritten on exit by the solution vector, X of $AX = B$
- `ga_print_patch(g_a, ilo, ihi, jlo, jhi, pretty)` — print a patch of an array to the screen
 - integer `g_a` — [input] integer identifier of array to be printed
 - integer `ilo, ihi` — [input] high and low indices for i dimension of patch of array to be printed
 - integer `jlo, jhi` — [input] high and low indices for j dimension of patch of array to be printed
 - integer `pretty` — [input] flag for format of output to screen;
 - * `pretty = 0`, spew output out with no formatting
 - * `pretty = 1`, format output so that it is readable
- `ga_print(g_a)` — print an entire array to the screen
 - integer `g_a` — [input] integer identifier of array to be printed

- `ga_copy_patch(trans, g_a, ailo, aihi, ajlo, ajhi, g_b, bilo, bihi, bjlo, bjhi)` — copy data from a patch of one global array into another array, (Note: patch can change shape, but total number of elements must be the same between the two arrays)
 - character*1 `trans` — [input] transpose or not
 - integer `g_a` — [input] integer identifier of array to be copied
 - integer `ailo, aihi` — [input] high and low indices for i dimension of patch of array to be copied
 - integer `ajlo, ajhi` — [input] high and low indices for j dimension of patch of array to be copied
 - integer `g_b` — [output] integer identifier of array data is to be copied into
 - integer `bilo, bihi` — [input] high and low indices for i dimension of patch of array being copied into
 - integer `bjlo, bjhi` — [input] high and low indices for j dimension of patch of array being copied into
- `ga_compare_distr(g_a, g_b)` — compare distributions of two global arrays
 - integer `g_a` — [input] integer identifier of first array
 - integer `g_b` — [output] integer identifier of second array

Operations that may be invoked by any process in true MIMD style:

- `ga_get(g_a, ilo, ihi, jlo, jhi, buf, Id)` — read from a patch of an array
 - integer `g_a` — [input] integer handle of array
 - integer `ilo, ihi` — [input] high and low indices for i dimension of region
 - integer `jlo, jhi` — [input] high and low indices for j dimension of region
 - integer `buf` — [output] ???
 - integer `Id` — [output] ???
- `ga_put(g_a, ilo, ihi, jlo, jhi, buf, Id)` — write from a patch of an array
 - integer `g_a` — [input] integer handle of array
 - integer `ilo, ihi` — [input] high and low indices for i dimension of region
 - integer `jlo, jhi` — [input] high and low indices for j dimension of region
 - integer `buf` — [output] ???
 - integer `Id` — [output] ???
- `ga_acc(g_a, ilo, ihi, jlo, jhi, buf, Id, alpha)` — accumulate into a patch of an array (double precision only)
 - integer `g_a` — [input] integer handle of array
 - integer `ilo, ihi` — [input] high and low indices for i dimension of region
 - integer `jlo, jhi` — [input] high and low indices for j dimension of region
 - integer `buf` — [output] ???
 - integer `Id` — [output] ???
 - integer `alpha` — ????
- `ga_scatter(g_a, v, i, j, nv)` — scatter elements of `v` into an array
 - integer `g_a` — [input] integer handle of array that elements of `v` are to be scattered into

- ??? v — [input] array from which elements are to be scattered
- integer i, j — [input] array element indices (i,j) as in FORTRAN
- integer nv — ???
- ga_gather_g_a, v, i, j, nv — gather elements from an array v into array g_a
 - integer g_a — [input] integer handle of array that elements of v are to be gathered into
 - ??? v — [input] array from which elements are to be gathered
 - integer i, j — [input] array element indices (i,j) as in FORTRAN
 - integer nv — ???
- ga_read_inc(g_a, i, j, inc) — atomically read and increment the value of a single array element (integers only)
 - integer g_a — [input] integer handle of array
 - integer i, j — [input] array element indices (i,j) as in FORTRAN
 - integer inc — [input] amount to increment array element value
- ga_locate(g_a, i, j, owner) — determine which process ‘holds’ an array element (i,j)
 - integer g_a — [input] integer handle of array
 - integer i, j — [input] array element indices (i,j) as in FORTRAN
 - integer owner — [output] index number of processor holding the element
- ga_locate_region(g_a, ilo, ihi, jlo, jhi, map, np) — determine which process ‘holds’ an array section
 - integer g_a — [input] integer handle of array
 - integer ilo, ihi — [input] high and low indices for i dimension of region
 - integer jlo, jhi — [input] high and low indices for j dimension of region
 - ????? map — [output] ???
 - integer np — [output] index number of processor holding the region
- ga_error(string, icode) — print error message and terminate the program
 - character string — [input] ???
 - integer icode — [input] integer flag for error code
- ga_summarize(verbose) — print information about all allocated arrays (note: assumes no more than 100 arrays are allocated and are numbered -1000, -999, etc.)
 - integer verbose — [input] if non-zero, print distribution information

Operations that may be invoked by any process in true MIMD style and are intended to support writing of new functions:

- ga_distribution(g_a, me, ilo, ihi, jlo, jhi) — find coordinates of the array patch that is ‘held’ by a processor
 - integer g_a — [input] integer handle of array
 - integer me — [input] index number of processor holding the patch

- integer `ilo`, `ihi` — [output] high and low indices for i dimension of region
- integer `jlo`, `jhi` — [output] high and low indices for j dimension of region
- `ga_access(g_a, ilo, ihi, jlo, jhi, index, Id)` — provides access to a patch of a global array
 - integer `g_a` — [input] integer handle of array to be accessed
 - integer `ilo`, `ihi` — [output] high and low indices for i dimension of region
 - integer `jlo`, `jhi` — [output] high and low indices for j dimension of region
 - integer `index` — ????
 - integer `Id` — ????
- `ga_release(g_a, ilo, ihi, jlo, jhi)` — relinquish access to internal data
 - integer `g_a` — [input] integer handle of array to be released
 - integer `ilo`, `ihi` — [output] high and low indices for i dimension of region
 - integer `jlo`, `jhi` — [output] high and low indices for j dimension of region
- `ga_release_update(g_a, ilo, ihi, jlo, jhi)` — relinquish access after data were updated
 - integer `g_a` — [input] integer handle of array to be updated and released
 - integer `ilo`, `ihi` — [output] high and low indices for i dimension of region
 - integer `jlo`, `jhi` — [output] high and low indices for j dimension of region
- `ga_check_handle(g_a, fstring)` — verify that a GA handle is valid
 - integer `g_a` — [input] integer handle of array
 - character* `fstring` — [input] name of routine originating the check

Operations to support portability between implementations:

- `ga_nodeid_()` — find requesting compute process message id
- `ga_nnodes_()` — find number of compute processes
- `ga_dgop(type, x, n, op)` — equivalent to TCGMSG `dgop`, for use in data-server mode where only compute processes participate
 - integer `type` — [input] integer handle of array
 - integer `n` — [input]
 - double precision `x` — [input]
 - character `op` — [input]
- `ga_igop(type, x, n, op)` — equivalent to TCGMSG `igop`, for use in data-server mode where only compute processes participate; performs the operation specified by the input variable `op` (supported operations include addition, multiplication, maximum, minimum, and maximum or minimum of the absolute value), and returns the value in `x`.
 - integer `type` — [input] integer handle of array
 - integer `n` — [input]
 - double precision `x` — [input/output]

- character `op` — [input]
- `ga_brdcst(type, buf, len, originator)` — equivalent to TCGMSG `brdcst`, for use in data server mode with predefined communicators
 - integer `type` — [input] integer handle of array
 - `buf` — [input]
 - integer `len` — [input]
 - integer `originator` — [input] number of originating processor

Other utility operations:

- `ga_inquire_(g_a, atype, adim1, adim2)` — find the type and dimensions of the array
 - integer `g_a` — [input] integer identifier of array
 - integer `atype` — [output] MA type
 - integer `adim1, adim2` — [output] array dimensions (`adim1, adim2`) as in FORTRAN
- `ga_inquire_name_(g_a, array_name)` — find the name of the array
 - integer `g_a` — [input] integer identifier of array
 - character* `array_name` — [output] string containing name of the array
- `ga_inquire_memory_()` — find the amount of memory in active arrays
- `ga_memory_avail_()` — find the amount of memory (in bytes) left for GA
- `ga_summarize(verbose)` — prints summary info about allocated arrays
 - integer `verbose` — [input] if non-zero, print distribution information
- `ga_uses_ma_()` — finds if memory in arrays comes from MA (memory allocator)
- `ga_memory_limited_()` — finds if limits were set for memory usage in arrays

Note that consistency is only guaranteed for

1. Multiple read operations (as the data does not change)
2. Multiple accumulate operations (as addition is commutative)
3. Multiple disjoint put operations (as there is only one writer for each element)

The application has to worry about everything else (usually by appropriate insertion of `ga_sync` calls).

New(?) Stuff

Subroutines that appear in the files of directory `.../src/global/src`, but are not in the `(ga.tex)` document;

- `ga_get_local(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc)` — local read of a 2-dimensional patch of data into a global array

- `ga_get_remote(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc)` — read an array patch from a remote processor
- `ga_put_local(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc)` — local write of a 2-dimensional patch of data into a global array
- `ga_put_remote(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc)` — write an array patch from a remote processor
- `ga_acc_local(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc, alpha)` — local accumulate of a 2-dimensional patch of data into a global array
- `ga_acc_remote(g_a, ilo, ihi, jlo, jhi, buf, offset, Id, proc, alpha)` — accumulate an array patch from a remote processor
- `ga_scatter_local(g_a, v, i, j, nv, proc)` — local scatter of `v` into a global array
- `ga_scatter_remote(g_a, v, i, j, nv, proc)` — scatter of `v` into an array patch from a remote processor
- `ga_gather_local(g_a, v, i, j, nv, proc)` — local gather of `v` into a global array
- `ga_gather_remote(g_a, v, i, j, nv, proc)` — gather of `v` into an array patch from a remote processor
- `ga_dgop_clust(type, x, n, op, group)` — equivalent to TCGMSG `dgop`, for use in data-server mode where only compute processes participate
- `ga_igop_clust(type, x, n, op, group)` — equivalent to TCGMSG `igop`, for use in data-server mode where only compute processes participate
- `ga_brdcst_clust(type, buf, len, originator, group)` — internal GA routine that is used in data server mode with predefined communicators
- `ga_debug_suspend()` — ??? option to suspend debugging for a particular process
- `ga_copy_patch_dp(t_a, g_a, ailo, aihi, ajlo, ajhi, g_b, bilo, bihi, bjlo, bjhi)` — copy a patch by column order (Fortran convention)
- `ga_print_stats_()` — print GA statistics for each process
- `ga_zeroUL(uplo, g_A)` — set to zero the L/U triangle part of an NxN double precision global array `A`
- `ga_symUL(uplo, g_A)` — make a symmetric square matrix from a double precision global array `A` in L/U triangle format
- `ga_llt_s(uplo, g_A, g_B, hsA)` — solves a system of linear equations $[A]X = [B]$,
- `ga_cholesky(uplo, g_a)` — computes the Cholesky factorization of an NxN double precision symmetric positive definite matrix to obtain the L/U factor on the lower/upper triangular part of the matrix
- `ga_llt_f(uplo, g_A, hsA)` — computes the Cholesky factorization of an NxN double precision symmetric positive definite global array `A`
- `ga_llt_i(uplo, g_A, hsA)` — computes the inverse of a global array that is the lower triangle `L` or the upper triangular Cholesky factor `U` of an NxN double precision symmetric positive definite global array (`LL'` or `U'U`)

- `ga_llt_solve(g_A, g_B)` — solves a system of linear equations $[A]X = [B]$ using the cholesky factorization of an NxN double precision symmetric positive definite global array A
- `ga_spd_invert(g_A)` — computes the inverse of a double precision array using the cholesky factorization of an NxN double precision symmetric positive definite global array A
- `ga_solve(g_A, g_B)` — solves a system of linear equations $[A]X = [B]$, trying first to use the Cholesky factorization routine; if not successful, calls the LU factorization routine `ga_llt_solve`, and solves the system with forward/backward substitution
- `ga_ma_base_address(type, address)` — auxiliary routine to provide MA base addresses of the data (calls C routines `ga_ma_get_ptr()`)
- `ga_ma_sizeof(type)` — auxiliary routine to provide MA sizes of the arrays (calls C routines `ga_ma_diff()`)

Use of TCGMSG global operation routines

In some cases (notably workstation clusters) the global array tools use a “data-server” process on each node in addition to the compute processes. Data-server processes don’t follow the same flow of execution of compute processes, so TCGMSG global operations (`brdcst`, `igop`, and `dgop`) will hang when invoked. The global array toolkit provides “wrapper” functions (`ga_brdcst`, `ga_igop`, and `ga_dgop`) which properly exclude data server processes from the global communication and must be used instead of the corresponding TCGMSG functions.

Interaction between GA and message-passing

The limited buffering available on the IBM SP-1/2 means that GA and message-passing operations cannot interleave as readily as they do on other machines. Basically, in transitioning from GA to message passing or vice versa the application must call `ga_sync()`.

6.1.4 ChemI/O

ChemIO is a high-performanc parallel I/O abstract programming interface for computational chemistry applications². The development of out-of-core methods for computational chemistry requires efficient and portable implementation of often complex I/O patterns. The ChemIO interface addresses this problem by providing high performance implementations on multiple platforms that hides some of the complexity of the underlying I/O patterns from the programmer through the use of high-level libraries. The interface is tailored to the requirements of large-scale computational chemistry problems and supports three distinct I/O models. These are

1. *Disk Resident Arrays (DRA)* — for explicit transfer between global memory and secondary storage, allowing the programmer to manage the movement of array data structures between local memory, remote memory, and disk storage. This component supports collective I/O operations, in which multiple processors cooperate in a read or write operation and thereby enable certain useful optimizations.
2. *Exclusive Access Files (EAF)* — for independent I/O to and from scratch files maintained on a per-processor basis. It is used for out-of-core computations in calculational modules that cannot easily be organized to perform collective I/O operations.

²The ChemIO project is a joint effort of Argonne National Laboratory and Pacific Northwest National Laboratory, in affiliation with a DOE Grand Challenge project developing Massively Parallel Methods for Computational Chemistry, with the multi-agency Scalable I/O Project, and with the EMSL.

3. *Shared Files (SF)* — for creation of a scratch file that can be shared by all processors. Each processor can perform noncollective read or write operations to an arbitrary location in the file.

These models are implemented in three user-level libraries in ChemIO; Disk Resident Arrays, Exclusive Access Files, and Shared Files. These libraries are layered on a device library, the Elementary I/O library (ELIO), which provides a portable interface to different file systems. The DRA, EAF, and SF modules are fully independent. Each one can be modified or even removed without affecting the others. ELIO itself is not exposed to applications.

Elementary I/O Library (ELIO)

The ELIO library implements a set of elementary I/O primitives including blocking and non-blocking versions of read and write operations, as well as wait and probe operations to control status of non-blocking read/writes. It also implements file operations such as open, close, delete, truncate, end-of-file detection, and an inquiry function for the file/filesystem that returns the amount of available space and the filesystem type. Most of these operations are commonly seen in various flavors of the UNIX filesystem. ELIO provides an abstract portable interface to such functionality.

(Insert gory details here.)

Disk Resident Arrays

The computational chemistry parallel algorithms in NWChem have been implemented in terms of the Global Arrays shared memory programming model. The GA library (see Section 6.1.3) uses a shared memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. The GA library allows each process in a MIMD parallel program to access asynchronously logical blocks of physically distributed matrices without the need for explicit cooperation from other processes.

The GA model exposes to the programmer the non-uniform memory access (NUMA) characteristics of modern high-performance computer systems. The disk resident array (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage. It introduces the concept of a disk resident array — a disk-based representation of an array — and provides functions for transferring blocks of data between global arrays and disk arrays. It allows the programmer to access data located on disk via a simple interface expressed in terms of arrays rather than files.

At the present time, (*NOTE: The source of this statement is a document created 5/10/95*) all operations are declared to be collective. This simplifies implementation on machines where only some processors are connected to I/O devices.

Except where stated otherwise, all operations are synchronous (blocking) which means that control is returned to the calling process only after the requested operation completes.

All operations return an error code with value 0 if successful, greater than zero if not successful.

A program that uses Disk Resident Arrays should look like the following example:

```

    program foo
#include "mafdecls.h"
#include "global.fh"
#include "dra.fh"
c
    call pbeginf()                ! initialize TCGMSG
    if(.not. ma_init(...)) ERROR ! initialize MA

```

```

        call ga_initialize()           ! initialize Global Arrays
        if(dra_init(...).ne.0) ERROR ! initialize Disk Arrays

c      do work

        if(dra_terminate().ne.0)ERROR ! destroy DRA internal data structures
        call ga_terminate             ! terminate Global Arrays
        call pend()                   ! terminate TCGMSG
        end

```

List of DRA operations:

- `status = dra_init(max_arrays, max_array_size, total_disk_space, max_memory)` — initializes disk resident array I/O subsystem; `max_array_size`, `total_disk_space` and `max_memory` are given in bytes; `max_memory` specifies how much local memory per processor the application is willing to provide to the DRA I/O subsystem for buffering. The value of "-1" for any of input arguments means: "don't care", "don't know", or "use defaults"
 - integer `max_arrays` — [input]
 - double precision `max_array_size` — [input]
 - double precision `total_disk_space` — [input]
 - double precision `max_memory` — [input]
- `status = dra_terminate()` — closes all open disk resident arrays and shuts down DRA I/O subsystem.
- `status = dra_create(type, dim1, dim2, name, filename, mode, rdim1, rdim2, d_a)` — creates new disk resident array with specified dimensions and `type`. (Note: Only one DRA object can be stored in DRA meta-file identified by `filename`. DRA objects persist on the disk after calling `dra_close()`. `dra_delete()` should be used instead of `dra_close()` to delete disk array and associated meta-file on the disk. Disk array is implicitly initialized to "0".
 - integer `type` — [input] MA type identifier
 - integer `dim1` — [input]
 - integer `dim2` — [input]
 - character*(*) `name` — [input]
 - character*(*) `filename` — [input] name of an abstract meta-file that will store the data on the disk. The
 - integer `mode` — [input] specifies access permissions as read, write, or read-and-write
 - integer `rdim1, rdim2` — [input] specifies dimensions of a "typical" request; value of "-1" for either `rdim1` or `rdim2` means "unspecified"
 - integer `d_a` — [output] DRA handle
- `status = dra_open(filename, mode, d_a)` — Open and assign DRA handle to disk resident array stored in DRA meta-file `filename`. Disk arrays that are created with `dra_create` and saved by calling `dra_close` can be later opened and accessed by the same or different application.
 - character*(*) `filename` — [input] name of an abstract meta-file that will store the data on the disk. The
 - integer `mode` — [input] specifies access permissions as read, write, or read-and-write
 - integer `d_a` — [output] DRA handle

- `status = dra_write(g_a, d_a, request)` — writes asynchronously specified global array to specified disk resident array; dimensions and type of `g_a` and `d_a` must match. If dimensions don't match, `dra_write_section` should be used instead. The operation is by definition asynchronous (but could be implemented as synchronous i.e., it would return only when I/O is done.)
 - integer `g_a` — [input] GA handle
 - integer `d_a` — [input] DRA handle
 - integer `request` — [output] request id

- `status = dra_write_section(transp, g_a, gilo, gihi, gjlo, gjhi, d_a, dilo, dihi, djlo, djhi, request)` — writes asynchronously specified global array section to specified disk resident array section: `OP(g_a[gilo:gihi, gjlo:gjhi]) --> d_a[dilo:dihi, djlo:djhi]`, where `OP` is the transpose operator (`.true./false.`). Returns error if the two section's types or sizes mismatch. See `dra_write specs` for discussion of `request`.
 - logical `transp` — [input] transpose operator
 - integer `g_a` — [input] GA handle
 - integer `d_a` — [input] DRA handle
 - integer `gilo` — [input]
 - integer `gihi` — [input]
 - integer `gjlo` — [input]
 - integer `gjhi` — [input]
 - integer `dilo` — [input]
 - integer `dihi` — [input]
 - integer `djlo` — [input]
 - integer `djhi` — [input]
 - integer `request` — [output] request id

- `status = dra_read(g_a, d_a, request)` — reads asynchronously specified global array from specified disk resident array; Dimensions and type of `g_a` and `d_a` must match; if dimensions don't match, `dra_read_section` could be used instead. See `dra_write specs` for discussion of `request`.
 - logical `transp` — [input] transpose operator
 - integer `g_a` — [input] GA handle
 - integer `d_a` — [input] DRA handle
 - integer `request` — [output] request id

- `status = dra_read_section(transp, g_a, gilo, gihi, gjlo, gjhi, d_a, dilo, dihi, djlo, djhi, request)` — reads asynchronously specified global array section from specified disk resident array section: `OP(d_a[dilo:dihi, djlo:djhi]) --> g_a[gilo:gihi, gjlo:gjhi]` where `OP` is the transpose operator (`.true./false.`). See `dra_write specs` for discussion of `request`.
 - logical `transp` — [input] transpose operator
 - integer `g_a` — [input] GA handle
 - integer `d_a` — [input] DRA handle
 - integer `gilo` — [input]

- integer gihi — [input]
 - integer gjlo — [input]
 - integer gjhi — [input]
 - integer dilo — [input]
 - integer dihi — [input]
 - integer djlo — [input]
 - integer djhi — [input]
 - integer request — [output] request id
- `status = dra_probe(request, compl_status)` — tests for completion of `dra_write/read` or `dra_write/read.section` operation which sets the value passed in `request` argument; completion status is 0 if the operation has been completed, non-zero if not done yet
 - integer request — [input] request id
 - integer compl_status — [output] completion status
 - `status = dra_wait(request)` — blocks operations until completion of `dra_write/read` or `dra_write/read.section` operation which set the value passed in `request` argument.
 - integer request — [input] request id
 - `status = dra_inquire(d_a, type, dim1, dim2, name, filename)` — returns dimensions, type, name of disk resident array, and filename of DRA meta-file associated with `d_a` handle.
 - integer d_a — [input] DRA handle
 - integer type — [output]
 - integer dim1 — [output]
 - integer dim2 — [output]
 - character*(*) name — [output]
 - character*(*) filename — [output]
 - `status = dra_delete(d_a)` — deletes a disk resident array associated with `d_a` handle. Invalidates handle. The corresponding DRA meta-file is destroyed.
 - integer d_a — [input] DRA handle
 - `status = dra_close(d_a)` — closes DRA meta-file associated with `d_a` handle and deallocates data structures corresponding to this disk array. Invalidates `d_a` handle. The array on the disk is persistent.
 - integer d_a — [input] DRA handle
 - subroutine `dra_flick()` — returns control to DRA for a VERY short time to improve progress of pending asynchronous operations.

Exclusive Access Files (EAF)

The EAF module supports a particularly simple I/O abstraction in which each processor in a program is able to create files that it alone has access to. The EAF interface is similar to the standard C UNIX I/O interface and is implemented as a thin wrapper on the ELIO module. It provides Fortran and C applications with capabilities that include

- `eaf_write` and `eaf_read` — blocking write and read operations
- `eaf_await` and `eaf_aread` — non-blocking (asynchronous) write and read operations
- `eaf_wait` and `eaf_probe` — operations that can be used to control or determine completion status of outstanding nonblocking I/O requests
- `eaf_stats` — operation that takes a full path to a file or directory and returns the amount of disk space available and the filesystem type (e.g., PFS, PIOFS, standard UNIX, etc.)
- `eaf_length` and `eaf_truncate` — operations that can allow the programmer to determine the length of a file, and truncate a file to a specified length.
- `eaf_eof` — operation that determines whether the eof of the file has been reached
- `eaf_open`, `eaf_close`, and `eaf_delete` — functions that interface to UNIX `open`, `close`, and `unlink` operations

The syntax of EAF is similar to the standard Unix C file operations, although there are some differences, as a result of introducing new semantics or extended features available through EAF.

The primary functionality of EAF is illustrated here by tracing execution of example program segments.

Example 1: basic open-write-read-close sequence.

```
#include "chemio.h"
#include "eaf.fh"

integer fh  ! File Handle
integer sz  ! Return value of size written
integer stat ! Return status
integer buf(100) ! Data to write

fh = EAF_OpenPersist('/tmp/test.out', ELIO_RW) <- We probably want
      CHEMIO_RW here

sz = EAF_Write(fh, 0, buf, 100*EAF_SZ_INT)      <- What's the NWChem
                                                    macro for int size?

if(sz .ne. 100*EAF_SZ_INT)
  $      write(0,*) 'Error writing, wrote ', sz, ' bytes'

sz = EAF_Read(fh, 0, buf, 100*EAF_SZ_INT)
if(sz .ne. 100*EAF_SZ_INT)
  $      write(0,*) 'Error reading, read ', sz, ' bytes'

stat = EAF_Close(fh)
end
```

The include file 'chemio.h' defines the permission macros `ELIO_R`, `ELIO_W`, and `ELIO_RW` for read, write, and read-write permissions, respectively. The header file 'eaf.fh' is a Fortran program segment externally defining the EAF routines and must appear *before* any executable code using EAF.

`EAF_OpenPersist` opens a persistent file, as opposed to a scratch file (`EAF_OpenScratch`) which is deleted when it is closed. This file is named '/tmp/test.out' and has read-write permissions. The returned value is the file handle for this file and should not be directly manipulated by the user.

`EAF_Write` writes to the file opened with file handle, `fh`, at absolute offset 0. It is legal to write a scalar or array, for instance in the above example both 'buf' and 'buf(1)' have the same meaning. The last argument is the number of bytes to be written. It is important to multiply the number of array elements by the element size. The following macros are provided in 'eaf.fh':

- `EAF_SZ_BYTE`
- `EAF_SZ_CHARACTER`
- `EAF_SZ_INTEGER`
- `EAF_SZ_LOGICAL`
- `EAF_SZ_REAL`
- `EAF_SZ_COMPLEX`
- `EAF_SZ_DOUBLE_COMPLEX`
- `EAF_SZ_DOUBLE_PRECISION`

The return value is the number of bytes written. If this number does not match the requested number of bytes to be written, an error has occurred.

Example 2: read/write operations

`EAF_Read` is syntactically and semantically identical to `EAF_Write`, except the buffer is read, not written.

```
#include "chemio.h"
#include "eaf.fh"

integer fh ! File Handle
integer id1, id2 ! asynchronous ID handles
integer stat ! Return status
integer pend ! Pending status
integer iter ! Iterations counter
integer buf(100), x ! Data

iter = 0

fh = EAF_OpenScratch('/piofs/mogill/test.out', ELIO_RW)

stat = EAF_AWrite(fh, 0, buf, 100*EAF_SZ_INT, id1)
if(stat .ne. 0) write(0,*) 'Error doing 1st asynch write. stat=', stat

stat = EAF_AWrite(fh, 100*EAF_SZ_INT, x, 1*EAF_SZ_INT, id2)
if(stat .ne. 0) write(0,*) 'Error doing 2nd asynch write. stat=', stat
```

```

100 stat = EAF_Probe(id1, pend)
iter = iter + 1
write(0,*) 'Waiting', iter
if(iter .lt. 100 .and. pend .eq. ELIO_PENDING) goto 100
EAF_Wait(id1)

stat = EAF_ARead(fh, 0, buf, 100*EAF_SZ_INT, id1)
if(stat .ne. 0) write(0,*) 'Error doing 1st asynch read. stat=', stat

EAF_Wait(id2)
stat = EAF_AWrite(fh, 100*EAF_SZ_INT, x, 1*EAF_SZ_INT, id2)
if(stat .ne. 0) write(0,*) 'Error doing 2nd asynch write. stat=', stat
EAF_Wait(id2)
EAF_Wait(id1)

stat = EAF_Close(fh)
end

```

This example demonstrates use of asynchronous reading and writing. The entire buffer 'buf' is written to offset 0, the beginning of the file. The file is simultaneously written to from the scalar x in the position following the buffer. The positions in the file are determined by absolute offset argument as with the synchronous write.

The first write, id1, is repeatedly probed for completion for 100 tries or until completion, whichever comes first. The two possible pending statuses are ELIO_DONE and ELIO_PENDING.

When a completed asynchronous operation is detected with EAF_Wait or EAF_Probe, the id is invalidated with ELIO_DONE. The following EAF_Wait(id1) blocks until id1 completes. Using EAF_Probe or EAF_Wait with an invalidated ID has no effect.

Once id1 is freed, it is reused in the first asynchronous read statement. The following EAF_Wait blocks for completion and invalidation of id2, which is then used to asynchronously read the scalar X.

The EAF_Close deletes the file because it was opened as a scratch file.

List of EAF Functions

- integer EAF_OpenPersist(fname, type) — opens a persistent file; returns file handle, or -1 upon error
 - character fname — Character string of a globally unique filename (path may be fully qualified)
 - integer type — Read write permissions. Legal values are ELIO_W, ELIO_R, and ELIO_RW
- integer EAF_OpenScratch(fname, type) — open a scratch file that is automatically deleted upon close; returns file handle, or -1 upon error
 - character fname — Character string of a globally unique filename (path may be fully qualified)
 - integer type — Read write permissions. Legal values are ELIO_W, ELIO_R, and ELIO_RW
- integer EAF_Write(fh, offset, buf, bytes) — synchronously write to the file specified by the file handle; returns number of bytes written, or -1 on error
 - integer fh — File Handle
 - integer offset — Absolute offset, in bytes, at which to start writing

- any `buf` — Scalar or array of data
- integer `bytes` — Size of buffer, in bytes
- integer `EAF_AWrite(fh, offset, buf, bytes, req_id)` — asynchronously writes to the file specified by the file handle, and returns a handle to the asynchronous operation; if there are more than `MAX_AIO_REQ` asynchronous requests (reading or writing) pending, the operation is handled in a synchronous fashion and returns a "DONE" handle. Returns 0 if successful, -1 if an error occurs. (On architectures where asynchronous I/O operations are not supported, all requests are handled synchronously, returning a "DONE" handle.)
 - integer `fh` — [input] file descriptor
 - integer `offset` — [input] absolute offset, in bytes, to start writing at
 - any `buf` — [input] scalar or array of data
 - integer `bytes` — [input] size of buffer, in bytes
 - integer `req_id` — [output] handle of asynchronous operation
- integer `EAF_Read(fh, offset, buf, bytes)` — synchronously reads from the file specified by the file handle; returns number of bytes read, or -1 if an error occurs
 - integer `fh` — [input] file descriptor
 - integer `offset` — [input] absolute offset, in bytes, to start writing at
 - any `buf` — [input] scalar or array of data
 - integer `bytes` — [input] size of buffer, in bytes
- integer `EAF_ARead(fh, offset, buf, bytes, req_id)` — asynchronously reads from the file specified by the file handle, and returns a handle to the asynchronous operation. If there are more than `MAX_AIO_REQ` asynchronous requests (reading or writing) pending, the operation is handled in a synchronous fashion and returns a "DONE" handle. On architectures where asynchronous I/O operations are not supported, all requests are handled synchronously, returning a "DONE" handle. Returns 0 if successful; -1 if an error occurs.
 - integer `fh` — [input] file descriptor
 - integer `offset` — [input] absolute offset, in bytes, to start writing at
 - any `buf` — [input] scalar or array of data
 - integer `bytes` — [input] size of buffer, in bytes
 - integer `req_id` — [output] handle of asynchronous operation
- integer `EAF_Probe(id, status)` — determines if an asynchronous request is completed or pending; returns `ELIO_OK` if successful, or `ELIO_FAIL` if not successful; 'status' returns `ELIO_PENDING` if the asynchronous operation is not complete, or `ELIO_DONE` if finished. When the asynchronous request is complete, the 'id' is invalidated with `ELIO_DONE`.
 - integer `id` — [input] handle of asynchronous request
 - integer `status` — [output] pending or completed status argument
- integer `EAF_Wait(id)` — waits for the completion of the asynchronous request, `id`; returns `ELIO_OK` if successful, or `ELIO_FAIL` if not successful; 'id' is invalidated with `ELIO_DONE`
 - integer `id` — [input] handle of asynchronous request
- integer `EAF_Close(fh)` — closes a file; returns `ELIO_OK` if successful; aborts if not successful
 - integer `fh` — [input] file handle

Shared Files (SF)

The Shared File module supports the abstraction of a single contiguous secondary storage address space (a "file") that every processor has access to. Processes create and destroy SF objects in a collective fashion, but all other file I/O operations are non-collective. A shared file can be thought of as a one-dimensional array of bytes located in shared memory, except that the library interface is required to actually access the data.

The library is capable of determining the striping factor and all other internal optimizations for the "file". The programmer has the option, however, of giving the library a few helpful hints, to reduce the number of decisions the interface must take care of. These hints are supplied when the shared file is created, and can be any or all of the following:

1. Specify a hard limit (not to be exceeded) for the file size.
2. Specify a soft limit for the file size; that is, an estimate of the expected shared file size, which can be exceeded at run time, if necessary.
3. Specify the size of a "typical" request.

Non-collective I/O operations in SF include read, write, and wait operations. Read and write operations transfer the specified number of bytes between local memory and disk at a specified offset. The library does not perform any explicit control of consistency in concurrent accesses to overlapping sections of the shared files. For example, SF semantics allow a write operation to return before the data transfer is complete. This requires special care in programs that perform write operations in critical sections, since unlocking access to a critical section before write completes is unsafe. To allow mutual exclusion control in access to shared files, the `sf_wait` function is provided. It can be used to enforce completion of the data transfer so that the data can be safely accessed by another process after access to the critical section is released by the writing process. The function `sf_waitall` can be used to force the program to wait for completion of multiple SF operations specified through an argument array of request identifiers.

The actual size of a shared file might grow as processes perform write operations beyond the current end-of-file boundary. Data in shared files are implicitly initialized to zero, which means that read operations at locations that have not been written to return zero values. However, reading beyond the current end-of-file boundary is an error.

Shared files can be used to build other I/O abstractions. In many cases, this process requires adding an additional consistency control layer. A single file pointer view, for example, can be implemented by adding an automatically modifiable pointer variable located in shared memory by using the GA toolkit, or some other means.

The shared files model consists of the following elements:

- Shared files are non-persistent (temporary)
- Shared files resemble one-dimensional arrays in main memory
- Each process can independently read/write to any location in the file
- The file size has a hard limit specified when it is created
- User can also specify (or use "don't know" flag) the estimated approximate file size – might be exceeded at run-time (a hint)
- `sf_flush` flushes the buffers so that previously written data goes to the disk before the routine returns.
- All routines return an error code: "0" means success.
- `sf_create` and `sf_destroy` are collective

- file, request sizes, and offset (all in bytes) are DOUBLE PRECISION arguments, all the other arguments are INTEGERS
- read/writes are asynchronous

List of SF Functions:

```
integer sf_create(fname, size_hard_limit, size_soft_limit, req_size, handle)
    fname           -- meta-file name
    size_hard_limit -- max file size in bytes not to be exceeded (a hint)
    size_soft_limit -- estimated file size (a hint)
    req_size        -- size of a typical request (a hint)
    handle          -- returned handle to the created file
```

Creates shared file using name and path specified in `fname` as a template. Function `req_size` specifies size of a typical request (-1 = "don't know").

```
integer sf_write(handle, offset, bytes, buffer, request_id)
    handle           -- file handle returned from sf_create [in]
    offset           -- location in file (from the beginning)
                    -- where data should be written to [in]
    buffer           -- local array to put the data [in]
    bytes            -- number of bytes to read [in]
    request_id       -- id identifying asynchronous operation [out]
```

asynchronous write operation

```
integer sf_read(handle, offset, bytes, buffer, request_id)
    handle           -- file handle returned from sf_create [in]
    offset           -- location in file (from the beginning)
                    -- where data should be read from [in]
    buffer           -- local array to put the data [in]
    bytes            -- number of bytes to read [in]
    request_id       -- id identifying asynchronous operation [out]
```

asynchronous read operation

```
integer sf_wait(request_id)
    request_id       -- id identifying asynchronous operation [in/out]
```

blocks calling process until I/O operation associated with `id` completed, invalidates `request_id`

```
integer sf_waitall(list, num)
    list(num)        -- array of ids for asynchronous operations [in/o]
    num              -- number of entries in list [in]
```

blocks calling process until all "num" I/O operations associated with ids specified in `list` completed, invalidates ids on the list

```
integer sf_destroy(handle)
    handle           -- file handle returned from sf_create [in]
```

6.2 The Run Time Data Base (RTDB)

The run time data base is the parameter and information repository for the independent modules (e.g., SCF, RIMP2) comprising NWChem. This approach is similar in spirit to the GAMESS dumpfile or the Gaussian checkpoint file. The only way modules can share data is via the database or via files, the names of which are stored in the database (and may have default values). Information is stored directly in the database as typed arrays, each of which is described by

1. a name, which is a simple string of ASCII characters (e.g., "reference energies"),
2. the type of the data (real, integer, logical, or character),
3. the number of data items, and
4. the actual data (an array of items of the specified type).

A database is simply a file and is opened by name. Usually there is just one database per calculation, though multiple databases may be open at any instant.

By default, access to all open databases occur in parallel, meaning that

- all processes must participate in any read/write of any database and any such operation has an implied synchronization
- writes to the database write the data associated with process zero but the correct status of the operation is returned to all processes
- reads from the database read the data named by process zero and broadcast the data to all processes, checking dimensions and types of provided arrays

Alternatively, database operations can occur sequentially. This means that only process zero can read/write the database, and this happens with no communication or synchronization with other processes. Any read/write operations by any process other than process zero is an error.

Usually, all processes will want the same data at the same time from the database, and all processes will want to know of the success or failure of operations. This is readily done in the default parallel mode. An exception to this is during the reading of input. Usually, only process zero will read the input and needs to store the data directly into the database without involving the other processes. This is done using sequential mode.

The following subsections contain a detailed listing of the C and Fortran API. Programs using RTDB routines must include the appropriate header file; `rtdb.fh` for Fortran, or `rtdb.h` for C. These files define the return types for all `rtdb` functions. In addition, `rtdb.fh` specifies the following parameters

- `rtdb_max_key` — an integer parameter that defines the maximum length of a character string key
- `rtdb_max_file` — an integer parameter that defines the maximum length of a file name

The Fortran routines return logical values; `.true.` on success, `.false.` on failure. The C routines return integers; 1 on success, or 0 on failure. All `rtdb_*` functions are also mirrored by routines `rtdb_par_*` in which process 0 performs the operation and all other processes are broadcast the result of a read and discard writes.

6.2.1 Functions to Control Access to the Runtime Database

The functions that control opening, closing, writing to and reading information from the runtime database are described in this section.

rtdb_parallel

C routine:

```
int rtdb_parallel(const int mode)
```

Fortran routine:

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

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

rtdb_open

C routine:

```
int rtdb_open(const char *filename, const char *mode, int *handle)
```

Fortran routine:

```
logical function rtdb_open(filename, mode, handle)
character *(*) filename  [input]
character *(*) mode      [input]
integer handle           [output]
```

This function opens a database. It requires the following arguments:

- `Filename` — path to file associated with the data base
- `mode` — specify initial condition of data base
 - `new` — Open only if it does not exist already
 - `old` — Open only if it does exist already
 - `unknown` — Create a new database or open the existing database `Filename` (preserving contents)
 - `empty` — Create a new database or open the existing database `Filename` (deleting contents)
 - `scratch` — Create a new database or open the existing database `Filename` (deleting contents) that will be automatically deleted upon closing. Note that items cached in memory are not written to disk when this mode is specified.
- `handle` — returns an integer handle which must be used in all future references to the database

`rtdb_close`

C routine:

```
int rtdb_close(const int handle, const char *mode)
```

Fortran routine:

```
logical function rtdb_close(handle, mode)
integer handle           [input]
character*(*) mode      [input]
```

This function closes a database. It requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `mode` — specifies the fate of the information in the database after closing;
 - `keep` — Preserve the data base file to enable restart
 - `delete` — Delete the data base file, freeing all resources

When closing a database file that has been opened with the `rtdb_open` argument `mode` specified as `scratch`, the value for `mode` for the function `rtdb_close` is automatically set to `delete`. Database files needed for restart must not be opened as `scratch` files.

`rtdb_put`

C routine:

```
int rtdb_put(const int handle, const char *name, const int ma_type,
             const int nelelem, const void *array)
```

Fortran routine:

```
logical function rtdb_put(handle, name, ma_type, nelelem, array)
integer handle           [input]
character *(*) name      [input]
integer ma_type          [input]
integer nelelem          [input]
<ma_type>                [input]
nelelem                  [input]
array                    [input]
```

This function inserts an entry into the database, replacing the previous entry. It requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `name` — entry name of data array to be put into the database (null-terminated character string)
- `ma_type` — MA type of the entry
- `nelelem` — number of elements of the given type
- `array` — array of length `nelelem` containing data to be inserted

rtdb_get

C routine:

```
int rtdb_get(const int handle, const char *name, const int ma_type,
            const int nelelem, void *array)
```

Fortran routine:

```
logical function rtdb_get(handle, name, ma_type, nelelem, array)
integer handle           [input]
character *(*) name     [input]
integer ma_type         [input]
integer nelelem         [input]
<ma_type>              [output]
nelelem                [output]
array                  [output]
```

This function gets an entry from the data base. It requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `name` — entry name of data array to get from the database (null-terminated character string)
- `ma_type` — MA type of the entry (which must match entry type in the database)
- `nelelem` — size of array in units of `ma_type`
- `array` — buffer of length `nelelem` defined by calling routine to store the returned data

rtdb_cput **and** rtdb_cget

```
logical function rtdb_cput(handle, name, nelelem, buf)
integer handle           [input]
character *(*) name     [input]
character *(*) buf      [input]
```

```
logical function rtdb_cget(handle, name, nelelem, buf)
integer handle           [input]
character *(*) name     [input]
character *(*) buf      [output]
```

These functions are Fortran routines to provide put/get functionality for character variables. The functions have identical argument lists, the only difference between them is that for `rtdb_cput`, the specified character data is put into the database, and for `rtdb_cget` the data is copied from the database. The arguments are as follows;

- `handle` — unique handle created when the database was first opened
- `name` — entry name of data array to get from the database (null-terminated character string)
- `buf` — character variable to be put into the database (for `rtdb_cput`, or character buffer in calling routine to store returned character data (for `rtdb_cget`).

rtdb_ma_get

C routine:

```
int rtdb_ma_get(const int handle, const char *name, int *ma_type,
               int *nelem, int *ma_handle)
```

Fortran routine:

```
logical function rtdb_ma_get(handle, name, ma_type, nelem, ma_handle)
integer handle           [input]
character *(*) name     [input]
integer ma_type         [output]
integer nelem           [output]
integer ma_handle       [output]
```

This function returns the MA type, number of elements of that type, and the MA handle of the entry specified. (The MA handle is to memory automatically allocated to hold the data read from the database.) the function requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `name` — entry name of information to get from the database (null-terminated character string)
- `ma_type` — returns MA type of the entry in the database
- `nelem` — returns number of elements of type `ma_type` in data
- `ma_handle` — returns MA handle to data

rtdb_get_info

C routine:

```
int rtdb_get_info(const int handle, const char *name, int *ma_type,
                 int *nelem, char date[26])
```

Fortran routine:

```
logical function rtdb_get_info(handle, name, ma_type, nelem, date)
integer handle           [input]
character *(*) name     [input]
integer ma_type         [output]
integer nelem           [output]
character*26 date       [output]
```

This function queries the database to obtain the number of elements in the specified entry, it's MA type, and the date of its insertion into the rtdb. It requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `name` — entry name of data for which information is to be obtained (null-terminated character string in C, standard FORTRAN character constant or variable in FORTRAN)
- `ma_type` — returns MA type of the entry
- `nelem` — returns number of elements of the given type
- `date` — returns date of insertion (null-terminated character string or FORTRAN character variable)

`rtdb_first` **and** `rtdb_next`

C routines:

```
int rtdb_first(const int handle, const int namelen, char *name)
```

```
int rtdb_next(const int handle, const int namelen, char *name)
```

Fortran routines:

```
logical function rtdb_first(handle, name)
```

```
integer handle           [input]
```

```
character *(*) name     [output]
```

```
logical function rtdb_next(handle, name)
```

```
integer handle           [input]
```

```
character *(*) name     [output]
```

These routines enable iteration through the items in the database in an effectively random order. The function `rtdb_first` returns the name of the first user-inserted entry in the database. The function `rtdb_next` returns the name of the user-inserted entry put into the data base after the entry identified on the previous call to `rtdb_next` (or the call to `rtdb_first`, on the first call to `rtdb_next`).

The arguments required for the C routines are as follows:

- `handle` — unique handle created when the database was first opened
- `namelen` — size of buffer in calling routine required to store name
- `name` — buffer to hold returned name of next (or first) entry in the database

The Fortran routines require the same arguments for `handle` and `name`, but it is not necessary to define the length of the buffer required.

An example of the use of these functions in C is to count and print the name of all entries in the database. The coding for this can be implemented as follows;

```
char name[256];
```

```
int n, status, rtdb;
```

```
for (status=rtdb_first(rtdb, sizeof(name), name), n=0;
```

```
    status;
```

```
    status=rtdb_next(rtdb, sizeof(name), name), n++)
```

```
    printf("entry %d has name '%s'\n", n, name);
```

rtdb_delete

C routine:

```
int rtdb_delete(const int handle, const char *name)
```

Fortran routine:

```
logical function rtdb_delete(handle, name)
integer handle           [input]
character *(*) name     [input]
```

This function deletes an entry from the database.

- `handle` — unique handle created when the database was first opened
- `name` — entry name of data to delete from the database (null-terminated character string)

This function does not return any arguments. The value the function itself returns as indicates success or failure of the delete operation. The function returns as

- 1 if key was present and successfully deleted
- 0 if key was not present, or if an error occurred

rtdb_print

C routine:

```
int rtdb_print(const int handle, const int print_values)
```

Fortran routine:

```
logical function rtdb_print(handle, print_values)
integer handle           [input]
logical print_values     [input]
```

This function prints the contents of the data base to `STDOUT`. It requires the following arguments:

- `handle` — unique handle created when the database was first opened
- `print_values` — (boolean flag) if true, values as well as keys are printed out.

Chapter 7

Utilities

This Chapter describes the special features that are not specifically part of the Software Development Toolkit or the Molecular Modeling Toolkit, but are nevertheless integral to the functioning of NWChem. These special features include the input parser for processing the input and various utility routines that can be used by any module in the code, as needed.

The following sections describe each of these features in detail.

7.1 Input Parser

The input parser processes the user's input file and translates the information into a form meaningful to the main program and the driver routines for specific tasks. The parser translates input following the rules for free-format input specified in the NWChem Users Manual. The following subsections present detailed descriptions of the functions used by the input parser, and the conventional form of the processed input.

7.1.1 Free-format Fortran Input Routines – INP

All input routines must be declared in the header file `inp.fh`.

7.1.2 Initialization

```
inp_init
```

```
subroutine inp_init(ir, iw)
integer ir, iw      [input]
```

This function initializes free format input routines to take input from Fortran unit `ir` and send output to Fortran unit `iw`. The input file is processed from the current location.

Function `inp_init()` should be invoked each time the input file is repositioned using other than `inp_*` routines (e.g., `rewind`).

7.1.3 Basic Input Routines

The basic input routines read the format-free input provided by the user, and translate it by the syntax rules defined in the functions.

`inp_read`

```
logical function inp_read()
```

This routine reads a line from the input and splits it into white space (blank or tab) separated fields. White space may be incorporated into a field by enclosing it in quotes (for example, "new name"). The case of input is preserved. Blank lines are ignored and text beginning with a pound or hash symbol (#) is treated as a comment. A backslash(\) at the end of a line (followed only by white space) can be used to concatenate physical input lines into one logical input line. A semicolon (;) may be used to split a single physical input line into multiple logical input lines. The special command characters hash (#), semicolon (;) and quotation mark (") will be treated simply as characters only if prefaced by a backslash. (NOTE: This must be done even when the character appears within a character string enclosed in quotes.)

The number of fields read is initially set to 0, there being a total of `inp_n_field()` fields in the line.

If a non-blank line is successfully parsed then `.true.` is returned; otherwise an internal error message is set and `.false.` is returned. Possible errors include such actions as detection of unexpected EOF (which can be checked for with function `inp_eof()`), or failure to parse the line (e.g., a character string without a terminating quote).

End of file (EOF) is usually indicated by reaching the actual end of the physical input file. Alternatively, the user can specify the end of file location at any point by inserting a physical input line that begins with an asterisk (*), or a period, or the letters EOF (which may be in upper or lower case), and is followed only by trailing white space.

The maximum input line width is 1024 characters.

`inp_i`

```
logical function inp_i(i)
integer i      [output]
```

This function attempts to read the next field as an integer. Upon success, it returns `.true.` and advances to the next field. Otherwise it returns `.false.`, saves an internal error message and does not change the current field. The input argument (`i`) is not changed unless an integer is successfully read (so that any default value already present in variable `i` is not corrupted).

`inp_f`

```
logical function inp_f(d)
double precision d      [output]
```

This function attempts to read the next field as a floating point number. Upon success it returns `.true.` and advances to the next field. Otherwise it returns `.false.`, saves an internal error message and does not change the current field. The input argument (`d`) is not changed unless a double precision number is successfully read (so that any default value already present in variable `d` is not corrupted).

`inp_a`

```
logical function inp_a(a)
character *(*) a          [output]
```

This function attempts to read the next field as a character string. Upon success it returns `.true.` and advances to the next field. Otherwise it returns `.false.`, saves an internal error message and does not change the current field.

`inp_a_trunc`

```
logical function inp_a_trunc(a)
character *(*) a          [output]
```

This function attempts to read the next field as a character string, quietly discarding any data that does not fit in the user provided buffer. Upon success, it returns `.true.` and advances to the next field. Otherwise it returns `.false.`, saves an internal error message and does not change the current field.

`inp_line`

```
logical function inp_line(z)
character*(*) z          [output]
```

This function returns in `z` as much of the entire input line as it will hold and quietly discards any overflow. Upon success returns `.true.`, otherwise saves an internal error message and returns `.false.`.

`inp_cline`

```
subroutine inp_cline(z, len, success)
character*(*) z          [output]
integer len              [input]
logical success         [input]
```

This is a C-callable equivalent of `inp_line`, which puts $(len - 1)$ characters of the input line into the character string `z`. Trailing spaces are eliminated and the string is terminated with a 0 character, as is standard for C.

`inp_irange`

```
logical function inp_irange(first, last, stride)
integer first, last, stride [output]
```

This function attempts to read the next field as a Fortran90-style triplet specifying a range with optional stride. Upon success returns `.true.` and advances to the next field. Otherwise, returns `.false.`, saves an internal error message, and does not change the current field. The input arguments are not changed unless an integer range is successfully read.

The syntax is `<first>[:<last>[:<stride>]]`, where all terms are integers. The default `<stride>` is 1. A simple integer is, in essence, a degenerate triplet, and will be read correctly by this routine. The result will be as if the input had been `"<first>:<first>:1"`.

`inp_illist`

```

logical function inp_illist(maxlist, list, n)
integer maxlist      [input]
integer list(maxlist) [output]
integer n            [output]

```

This routine reads the remainder of the line as a list of integers and puts the results in `list`. Ranges of integers may be input compactly using the notation of `inp_irange()`. The number of elements set from the input is returned in `n`.

The function `inp_illist` returns `.true.` if the input is a valid integer list, and `.false.` otherwise, also setting an appropriate error message. If $n > \text{maxlist}$, it indicates that there is too much data on the line to fit in `list`.

`inp_search`

```

logical function inp_search(ocase, z, nz)
logical ocase      [input]
integer nz         [input]
character*(*) z(nz) [input]

```

This function positions the input file at the next logical input line which has a first input field that matches the leading non-blank characters of one of the elements of `z`. If `ocase` is `.true.` then matches are case sensitive.

The function returns `.true.` if such a line is found, and resets the current input field to 0 (i.e., as if `inp_read()` had just been called).

The function returns `.false.` if no such line is found. The file will be either at EOF or at a line which was not successfully parsed. EOF may be detected by `inp_eof()`.

7.1.4 Routines concerning fields within a line

`inp_nfield`

```

integer function inp_nfield()

```

This function returns the number of fields in the current input line (1, ...). A value of 0 implies either that EOF or some other error was detected or `inp_read()` has not yet been called.

`inp_curfield`

```

integer function inp_curfield()

```

This function returns the number of fields in the input line that have been processed so far (0, ...). For example, if `inp_curfield()` returns 2, then the next field read by `inp_f()` will be field 3.

`inp_setfield`

```

subroutine inp_setfield(value)

```

```
integer value          [input]
```

This function sets the current field (as returned by `inp_cur_field`) to be `value`, where $0 \leq \text{value} \leq \text{inp.n_field}()$. An out of range value results in an error and termination of execution.

```
inp_prev_field
```

```
subroutine inp_prev_field()
```

This is a convenience routine that makes it possible to read the field that was last read on the current input line. It is simply implemented as

```
call inp_set_field(max(0,inp_cur_field()-1))
```

If the current field is at the beginning of the line, however, this is a null operation.

7.1.5 String routines

These routines don't actually read input but are helpful in interpreting input or formatting output.

```
inp_strlen
```

```
integer function inp_strlen(z)
character*(*) z          [input]
```

This routine returns the index of the last non-blank character in `z`. It returns zero for a fully blank string.

```
inp_lcase
```

```
subroutine inp_lcase(z)
character*(*) z          [input/output]
```

This routine converts the character string `z` to all lower case.

```
inp_ucase
```

```
subroutine inp_ucase(z)
character*(*) z          [input/output]
```

This routine converts the character string `z` to all upper case.

```
inp_compare
```

```
logical function inp_compare(ocase, a, b)
logical ocase          [input]
character*(*) a, b     [input]
```

This routine returns `.true.` if all the characters in `a` match the first `len(a)` characters of `b`. If `ocase` is `.true.` then comparisons are case sensitive, otherwise comparisons ignore case.

`inp_match`

```
logical function inp_match(nrec, ocase, test, array, ind)
integer nrec                [input]
logical ocase              [input]
character*(*) test        [input]
character*(*) array(nrec) [input]
integer ind                [output]
```

This routine attempts to find a unique match of `test(1:L)` against elements of `array(*)`, where `L` is the length of the character string `test`, ignoring trailing blanks. If `ocase` is `.true.` then comparisons are case sensitive, otherwise comparisons ignore case.

If a unique match is found, the routine assigns the index of the element to `ind` and returns `.true.`

If the match is ambiguous, `ind` is set to 0, and the function returns `.false.`

If no match is found, `ind` is set to -1, and the function returns `.false.`

`inp_strtok`

```
logical function inp_strtok(z, sep, istart, iend)
character*(*) z          ! [input] string to parse
character*(*) sep       ! [input] token separators
integer istart, iend    ! [output] start/end of next token
```

This routine returns the number of the start and end character of the next token in the character string. Tokens are separated by one of the characters in `sep`. Note that all characters in `sep` are used, including any trailing blanks.

Before the first call to this routine, `istart` must be initialized to zero, and both `istart` and `iend` must remain *unchanged* for subsequent calls. Repeated calls return the next token and `.true.` It returns `.false.` if there are no more tokens. The separators may be changed between calls. No internal state is maintained (which is why `istart` and `iend` must not be modified between calls) so multiple strings may be parsed simultaneously.

For example, to split the character string `list` into tokens separated by `' : '` and print each token out, you might execute

```
istart = 0
10 if (inp_strtok(list, ':', istart, iend)) then
    write(6,*) list(istart:iend)
    goto 10
endif
```

7.1.6 Error handling routines

`inp_errout`

```
subroutine inp_errout()
```

If there is an internal error message, this routine prints out its value, the current line number and its contents. If appropriate, it also indicates the problematic position in the current input line.

`inp_outrec`

```
subroutine inp_outrec()
```

Prints out the current input line.

`inp_clear_err`

```
subroutine inp_clear_err()
```

This routine clears error conditions and messages that may no longer be relevant. For instance, if values are read from a line until no more are available, the error message “at end of line looking for ...” will be internally recorded. A call to this routine will clear this state.

`inp_eof`

```
logical function inp_eof()
```

This routine returns `.true.` if EOF has been detected, `.false.` otherwise.

7.2 NWChem Ouput to Ecce

This section describes the output file that can be generated by NWChem that can be processed in the Extensible Computational Chemistry Environment (Ecce). Ecce is an integrated molecular modeling tool for analysis and simulation of complex chemical problems. Information on Ecce itself can be found on the web at

<http://www.emsl.pnl.gov:2080/docs/ecce>

This site gives access to on-line help, release notes for each version of Ecce, information on publications and on-going research, and a FAQs page.

7.2.1 Contents of Output for Ecce

Any data object of potential interest to the user and of reasonable size should be output to this file. Larger objects (e.g., the MOs or density grids) in other files will be stored in XDR format; those filenames should be included here.

Results of interest include:

1. Exit status for each module
2. Messages (Character output, e.g., the basis name)
3. various energies

4. geometry
5. multipole moment components and magnitudes
6. energy gradient
7. density grids
8. performance, profiling, and memory usage data
9. convergence parameters and iteration number for iterative processes

The current version of NWChem includes output for

1. SCF, DFT, MP2 CCSD energies and gradients, optimization with STEPPER and DRIVER.
2. Correct module stack, including the task (e.g., `gradient | mp2 | scf`)
3. Module entry and exit (with status)
4. Geometry as input by the user
5. Geometry when updated by STEPPER or DRIVER
6. Name of basis set(s) used by the application (e.g., if `ao basis` was set to be `6-31g*`)
7. Convergence of SCF/DFT (energy and orbital-gradient norm)
8. Final converged total energies
9. Gradients w.r.t. geometry (analytic and numerical) and norm thereof
10. MO coefficients and energies
11. Error messages reported via `errquit`

7.2.2 Format of Output

To accommodate:

1. simple scalar data of various types
2. vector, matrix and tensor data of various types
3. begin/end of groups of data objects (for example, a matrix of normal mode elements, along with a vector of frequencies and a vector of symmetry labels).
4. enter/exit messages, with status, of each module

the following general format was proposed,

```
<module stack>%begin%<keyword>%<dim1> ... <dimn>%<type>
<data>
<module stack>%end%<keyword>%<dim1> ... <dimn>%<type>
```

where:

<module stack> is a white space separated list of module names representing the (logical) call tree,
 <dim1> ... <dimn> contains the size of each dimension for arrays or tensors (unity for a scalar),
 <keyword> identifies the quantity that is being output (See below for more detail on the actual keywords.)
 <type> is the data type ("char", "int", "long", "float", "double", "complex float", and "logical"),
 <data> contains the values (one character string per line, multiple values per line for other data types limited to circa 80 characters per line).

in addition:

1. Floating point exponential notation should use “e” or “E”, not “d” or “D”.
2. Module names will everywhere be printed with internal white space replaced with underscores.
3. Lines should not exceed 1023 characters
4. Logical s

Information such as units is handled by the Ecce code registration.

7.2.3 NWChem Ecce Output API

This API is written in C with FORTRAN wrappers and most of the source is located in the \$NWCHEM_TOP/src/util/ecce_print.c file.

routine ecce_print_module_entry(module)

```
subroutine ecce_print_module_entry(module)
character*(*) module

void ecce_print_module_entry(const char *module)
```

If printing is enabled, prints

```
<module stack>%begin%enter%1%character
<module>
<module stack>%end%enter%1%character
```

and then pushes <module> onto the module name stack.

Module names will everywhere be printed with internal white space replaced with underscores.

routine ecce_print_module_exit(module, status)

```
subroutine ecce_print_module_exit(module, status)
character*(*) module
character*(*) status

void ecce_print_module_exit(const char *module, const char *status)
```

If printing is enabled, prints

```
<module stack>%begin%exit%2%character
<module> <status>
<module stack>%end%exit%2%character
```

and (with the valid assumption that no nwchem module is reentrant) pops the module name stack until either <module> is popped off it, or the stack is empty. If <module> is not found on the stack print an informative message to stderr (no more than one such message per calculation) and continue.

Status will be lowercased and printed enclosed in double quotes with any quotes inside the string quietly replaced with single quotes.

”ok” implies success

”anything else” implies some sort of failure, though perhaps a recoverable one.

routine ecce_print1(keyword, ma_type, data, dim1)

```
subroutine ecce_print1(keyword, ma_type, data, dim1)
character*(*) keyword
integer ma_type
<ma_type> data(*)
integer dim1

void ecce_print1(const char *keyword, int ma_type,
                 const void *data, int dim1)
```

Print a 1-dimensional array of the specified type.

Real data will be printed with the printf format `%.14e`

Boolean data will be printed as `t` and `f`

routine ecce_print2(keyword, ma_type, data, dim1, dim2)

```
subroutine ecce_print1(keyword, ma_type, data, lda1, dim1, dim2)
character*(*) keyword
integer ma_type
<ma_type> data(lda1,dim2)
integer lda1
integer dim1, dim2

void ecce_print2(const char *keyword, int ma_type,
                 int dim1, int lda1, int dim2, const void *data)
```

Print a 2-dimensional array of the specified type.

routine ecce_print1_char(keyword, data, dim1)

```

subroutine ecce_print1_char(keyword, data, dim1)
character*(*) keyword
character*(*) data(*)
integer dim1

```

Print a 1-dimensional array of FORTRAN character strings (printing of arrays of character strings from C is not currently provided for).

Character string data will be printed one string per line with no quoting of special characters.

routine ecce_print2_dbl_tol**routine ecce_print_control(status, old)**

```

subroutine ecce_print_control(status, old)
integer status ! [input]
integer old    ! [output]

void ecce_print_control(int status, int *old)

```

Sets the boolean (0=FALSE, 1=TRUE) controlling printing to `status` and returns the previous setting.

routine ecce_print_file_open(filename)

```

subroutine ecce_print_file_open(filename)
character *(*) filename

void ecce_print_file_open(const char *filename)

```

Open with create/truncate the named file to accept Ecce output. If there is an error print a message to `stderr`, disable all other ecce routines, and continue.

Inside NWChem only process 0 would open the file and all other processes would therefore quietly ignore all `ecce_*` calls.

routine ecce_print_file_close()

```

subroutine ecce_print_file_close()

void ecce_print_file_close(void)

```

Close the `ecce_output` file, if any and disable Ecce printing. If there are any errors print a message to `stderr` and continue.

routine `ecce_print_echo_input`

routine `ecce_print_echo_string`

routine `is_ecce_print_on`

Several routines are available within NWChem that perform more general tasks, such as printing out the basis set, geometry and the orbital vectors. These routines are delineated below.

subroutine `movecs_ecce_print_on`

Found in `$NWCHEM.TOP/src/ddscf/movecs_ecce.F`

subroutine `movecs_ecce_print_off`

Found in `$NWCHEM.TOP/src/ddscf/movecs_ecce.F`

subroutine `movecs_ecce`

Found in `$NWCHEM.TOP/src/ddscf/movecs_ecce.F`

bas_ `ecce_print_basis`

Found in `$NWCHEM.TOP/src/basis/bas_input.F`

geom_ `print_rtdb_ecce`

Found in `$NWCHEM.TOP/src/geom/geom_print/ecce.F`

geom_ `print_ecce`

Found in `$NWCHEM.TOP/src/geom/geom_print/ecce.F`

7.2.4 Standard exit status

Each module needs to provide a short text description of its exit status. Some of these can be standardized, most probably cannot.

Standard exit status values include

"ok" — success.

"warning" — success, but user should check the output for more information.

"error" — a fatal user or program error.

"failed to converge" — sometimes this is OK.

7.2.5 Standard keywords

The keyword identifies to Ecce and the user of Ecce the quantity that is being output (e.g., an energy or dipole moment). The module stack is also output, so Ecce is already aware of the overall context (e.g., that this is an SCF energy computed in the course of computing the MP2 energy). To increase the ease with which data can be input into Ecce and also accessed within Ecce it is important that standard values of keywords be used.

The NWChem electronic structure modules should output all quantities in atomic units and leave it to Ecce to handle any necessary conversions. Exceptions to this convention can be made, but a distinct keyword should be used (perhaps including the actual units used, e.g., "vibrational frequencies (cm-1)").

Here is the master list of keywords. All Ecce print keywords should be registered here. (Note: not all keywords given by NWChem are used by Ecce. The ones listed below are those that NWChem uses.)

Converged/final energies

`total energy` — the total energy at level of the module stack.

`mp2 energy` — the total MP2 energy.

`ccsd total energy` — the total ccsd energy.

`total ccsd energy` — the total ccsd energy.

`total ccsd(t) energy` — the total ccsd(t) energy.

`total ccsdt(ccsd) energy+` — the total ccsd+t(ccsd) energy.

`two-electron energy` — the two-electron energy (Coulomb plus exchange).

`coulomb energy` — the (inter-electron) Coulomb energy

`exchange energy` — the exchange energy.

`correlation energy` — the correlation energy (relative to the pertinent SCF or MCSCF reference function), generally refers to MP2 or DFT.

`ccsd correlation energy` — the CCSD correlation energy (relative to the pertinent SCF or MCSCF reference function).

`nuclear repulsion energy` — the nuclear repulsion energy at the current geometry.

`zero point energy` — zero point energy from a frequency calculation.

Converged/final properties

`geometry` — the current geometry of the molecule.

`cartesian coordinates` — the current cartesian coordinates of the molecule.

`gradients` — the current gradient of the molecule.

`total gradient` — the current gradient of the molecule.

`total dipole` — a 3-vector of (x,y,z) dipole moments.

`open shell dipole` — a 3-vector of (x,y,z) dipole moments.

alpha electronic dipole — a 3-vector of (x,y,z) dipole moments.

beta electronic dipole — a 3-vector of (x,y,z) dipole moments.

nuclear dipole — a 3-vector of (x,y,z) dipole moments.

total quadrupole —

open shell quadrupole —

alpha electronic quadrupole —

beta electronic quadrupole —

nuclear quadrupole —

total mulliken atomic charges — an N_{atom} -vector of the charge assigned by the Mulliken analysis to each atom.

total mulliken shell charges — an N_{shell} -vector of the charge assigned by the Mulliken analysis to each shell of basis functions.

orbital symmetries — symmetry information for each molecular orbital.

orbital energies — orbital energies for each molecular orbital.

frequencies — frequencies.

projected frequencies — projected frequencies.

normal modes — normal modes associated with each frequency.

projected normal modes — projected normal modes associated with each frequency.

intensities — frequency intensities.

intensities — frequency intensities.

intensities (debye/ang)² — frequency intensities in (debye/ang)².

intensities (KM/mol) — frequency intensities in (KM/mol).

projected intensities — frequency projected intensities.

projected intensities (debye/ang)² — frequency projected intensities in (debye/ang)².

projected intensities (KM/mol) — frequency projected intensities in (KM/mol).

Convergence information

iteration — the iteration number for geometry steps, starting at 1.

iteration counter — the iteration counter when optimizing the wavefunction.

iterative total energy difference — the change in the total energy since the previous iteration.

gradient norm — an (estimate) of the norm of the gradient vector or something proportional to it for minimization/transition state algorithms (e.g., SCF orbital-gradient, DFT norm of occupied-virtual Fock-matrix, geometry optimization cartesian nuclear gradient).

`gradient max` — an (estimate) of the maximum absolute value element in the gradient vector.

`residual norm` — norm of the error vector in linear/non-linear equation solution

`residual max` — the absolute maximum value of the residual vector

`scaled residual norm` — the norm of the residual scaled by the norm of the RHS vector in the iterative solution of non-linear equations (roughly the negative logarithm of the number of significant figures)

Timing and performance information

The long term plan is to get all timing information into Ecce so that it will be possible to track performance and even form parallel speedup curves. By using the NWChem standard libraries and timing mechanisms most statistics will be automatically gathered.

- The standard module entry/exit protocol routines (to be written) will record entry and exit cpu/wall times, MA and GA usage statistics, and virtual memory activity etc.
- The CHEMIO library will eventually automatically record I/O activity and provide a routine to track activity on all nodes.
- The PSTAT module will very soon record all performance information it tracks.
- MA

PSTAT should be used for nearly all performance statistics — if it does not do what you want then it can be extended.

Additional keywords:

`all tasks cpu time` —

`single task cpu time` —

`cpu time` —

`all tasks wall time` —

`single task wall time` —

`wall time` —

7.3 Utility routines

The NWChem `util` directory is a dumping ground for all sorts of useful things, some of which have been described elsewhere. Here are the rest.

7.3.1 Printing utilities

`util_print_centered`

```
subroutine util_print_centered(unit, string, center, ounder)
integer unit           [input]
character*(*) string  [input]
integer center        [input]
logical  ounder       [input]
```

Write the string to specified Fortran unit centered about the given column. If `ounder` is `.true.` then the string is underlined.

`banner`

```
subroutine banner(unit, string, char, top, bot, sides)
integer unit           [input]
character*(*) string  [input]
character*(1) char     [input]
logical top, bot, sides [input]
```

Write the string to specified Fortran unit flush against the left margin, optionally enclosing the top/bottom/sides with a box constructed from the given character. At some point this routine should be renamed `util_banner`.

`output`

```
subroutine output (z, rowlow, rowhi, collow, colhi,
                  rowdim, coldim, nctl)
double precision z(rowdim, coldim)
integer rowlow, rowhi, collow, colhi, rowdim, coldim, nctl
```

`Output` is a classic routine that prints non-zero rows of a double precision matrix in formatted form with numbered rows and columns. The arcane input is as follows;

- `z` — matrix to be printed
- `rowlow` — row number at which output is to begin
- `rowhi` — row number at which output is to end
- `collow` — column number at which output is to begin
- `colhi` — column number at which output is to end
- `rowdim` — number of rows in the matrix
- `coldim` — number of columns in the matrix
- `nctl` — carriage control flag; 1 for single space 2 for double space 3 for triple space — only 1 looks any good.

Two examples might help. To print `z(3:6,8:12)`

```
double precision z(n,m)
call output(z, 3, 6, 8, 12, n, m, 1)
```

To print `x(3:12)`

```
double precision x(n)
call output(x, 3, 12, 1, 1, n, 1, 1)
```

7.3.2 Error Routines

`errquit`

```
subroutine errquit(string, status)
character*(*) string
integer status
```

All fatal errors should result in a call to this routine, which prints out the string and status value to both standard error and standard output and attempts to kill all parallel processes and to tidy any allocated system resources (e.g., system V shared memory). The integer status may be any non-zero integer that has some meaning to the programmer.

7.3.3 Parallel Communication

`util_char_ga_brdcst`

```
subroutine util_char_ga_brdcst(type, string, originator)
integer type [input]
character*(*) string [input/output]
integer originator [input]
```

The standard broadcast routine `ga_brdcst` does not work portably with Fortran character strings for which this routine should be used instead. The string is broadcast from process `originator` to all other processes. `Type` is the standard message type or tag. All processes should execute the same call and there is an implied weak synchronization (i.e., no process can complete this statement until at least process `originator` has reached it).

`fcsnd` **and** `fcrcv`

```
subroutine fcsnd(type, string, node, sync)
subroutine fcrcv(type, string, slen, nodeselect, nodefrom, sync)
```

Similarly, the basic point-to-point message-passing routines do not work portably with Fortran character strings. Here are routines that work only with character strings. Refer to the standard TCGMSG documentation for details on the other arguments.

7.3.4 Naming Files

The length of a file name can be large and also system dependent. The parameter `NW_MAX_PATH_LEN` is defined in `util.fh` to enable a portable definition. Use it as follows

```
#include "util.fh"
      character*(nw_max_path_len) filename
```

For easy management by a use of NWChem, and so that multiple jobs can run without interaction in the same directory tree, all files should by default have a common prefix (specified on the `START`, `RESTART`, or `CONTINUE` directive). In addition, files need to be routed to the correct directory (scratch or permanent) and parallel files need the process number appended to the name. Routines (should) exist to do all of these things individually, but one master routine does it all for you — wow!

`util_file_name`

```
subroutine util_file_name(stub, oscratch, oparallel, name)
character*(*) stub      ! [input] stub name for file
logical oscratch       ! [input] T=scratch, F=permanent
logical oparallel      ! [input] T=append .<nodeid>
character*(*) name     ! [output] full filename
```

This routine prepends the common file prefix (specified on the `START`, `RESTART`, or `CONTINUE` directive) and directory (scratch or permanent) and appends the process number for parallel files. For example

```
call util_file_name('movecs', .false., .false., name)
```

might result in name being set to `/msrc/home/d3g681/c60.movecs` (i.e., having the name of the permanent directory and the file prefix prepended onto the stub).

Another example,

```
call util_file_name('khalf', .true., .true., name)
```

might yield `/scratch/h2o.khalf.99` (i.e., having the name of the scratch directory and the file prefix prepended and the process number appended).

`util_file_prefix`

```
subroutine util_file_prefix(name, fullname)
character*(*) name      [input]
character*(*) fullname  [output]
```

This routine is superceded for most purposes by `util_file_name()`.

By default all filenames should be prefixed with the `file_prefix` which is the argument presented to the `START`, `RESTART`, or `CONTINUE` directive in the input. This is most simply accomplished by calling this routine which returns in `fullname` the value of `file_prefix` followed (with no intervening characters) by the contents of `name`.

`util_pname`

```
subroutine util_pname(name, pname)
character*(*) name      [input]
character*(*) pname     [output]
```

This routine is superseded for most purposes by `util_file_name()`.

Construct a unique parallel name by appending the process number after the stub name. E.g., `fred.0001`, `fred.0002`, ... The number of leading zeroes are adjusted so that there are none in front of the highest numbered processor. This is useful for generating names for files, but is probably superseded by the exclusive access files in CHEMIO (see Section 6.1.4.)

7.3.5 Sequential Fortran Files

`util_flush`

```
subroutine util_flush(unit)
integer unit      [input]
```

If possible, flush the Fortran output buffers associated with the specified unit. Note that this is generally required in order for output to be visible during the course of a calculation and thus should be called after most write operations to standard output. Also, on some machines it is a fatal error to flush a unit on which no output has been performed thus care must be taken to ensure that writes and flushes are paired — e.g., it is wrong to have all processes flush unit six when only process zero has written output.

`sread` **and** `swrite`

```
subroutine sread(unit, a, n)
integer unit      [input]
double precision a(n) [output]
integer n        [input]

subroutine swrite(unit, a, n)
integer unit      [input]
double precision a(n) [input]
integer n        [input]
```

Read/write an array of double precision words from the given Fortran unit (variable record length, binary file). These routines are valuable to avoid inefficient implied DO loops and also to circumvent system limitations on record lengths (e.g., some Cray systems). The I/O operations are internally chopped into 0.25 Mbyte chunks.

7.3.6 Parallel file operations

`begin_seq_output`, `write_seq`, **and** `end_seq_output`

```
subroutine begin_seq_output()

subroutine write_seq(unit, text)
integer unit      [input]
character*(*) text [input]

subroutine end_seq_output()
```

These routines support sequential (i.e., ordered) formatted output from all parallel processes. A call to `begin_seq_output` indicates the start of a section of sequentialized output. This can be followed by any number of calls to `write_seq`, which *must* be followed by calling `end_seq_output`. All output will be sent to node 0 and written there in order of increasing node number.

All nodes must participate in all calls of a sequential output section.

Because we have to declare a fixed length string, it is possible for some transmissions to be truncated. In practice, however, we choose something rather longer than typical line lengths and it should not be a serious problem.

Observe that the specified unit is the Fortran unit on node zero, not that of the invoking node!

7.3.7 Data packing and unpacking

```

subroutine util_pack_16(nunpacked, packed, unpacked)
integer nunpacked                [input]
integer packed(*)                [output]
integer unpacked(nunpacked)     [input]

subroutine util_unpack_16(nunpacked, packed, unpacked)
integer nunpacked                [input]
integer packed(*)                [input]
integer unpacked(nunpacked)     [output]

subroutine util_pack_8(nunpacked, packed, unpacked)
integer nunpacked                [input]
integer packed(*)                [output]
integer unpacked(nunpacked)     [input]

subroutine util_unpack_8(nunpacked, packed, unpacked)
integer nunpacked                [input]
integer packed(*)                [input]
integer unpacked(nunpacked)     [output]

```

These routines pack/unpack unsigned eight bit (0–255) or sixteen bit (0–65535) integers to/from standard Fortran integers. The number of unpacked numbers *must* be a multiple of the number of values that fit into a single Fortran integer. On 32 bit machines this is four eight-bit values and two sixteen-bit values. On 64 bit machines these numbers are eight and four respectively. The number of values that can be packed per integer can be computed in a machine independent fashion using MA

```

npacked_per_int = ma_sizeof(mt_int, 1, mt_byte) /
                  n_bytes_per_value

```

under the assumption that the word length is an exact multiple of the value length.

7.3.8 Checksums

Checksums are useful for rapid comparison and validation of data, such as digital signatures for verification of important messages, or, more relevant to us, to determine if input and disk resident restart data are still consistent. The checksum routines provided here are wrappers around the RSA implementation of the RSA Data Security, Inc. MD5

Message-Digest Algorithm. It is the reference implementation for internet RFC 1321, The MD5 Message-Digest Algorithm, and as such has been extensively tested and there are no restrictions placed upon its distribution or export. License is granted by RSA to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. Consider this done. The unmodified network posting is included in md5.txt for reference.

MD5 is probably the strongest checksum algorithm most people will need for common use. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^{64} operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^{128} operations.

The checksums are returned (through the NWChem interface) as character strings containing a 32 character hexadecimal representation of the 128 bit binary checksum. This form loses no information, may be readily compared with single statements of standard C/F77, is easily printed, and does not suffer from byte ordering problems. The checksum depends on both the value and order of data, and thus differing numerical representations, floating-point rounding behaviour, and byte ordering, make the checksum of all but simple text data usually machine dependent unless great care is taken when moving data between machines. The Fortran test program merely tests the Fortran interface. For a more definitive test of MD5 make `mddriver` and execute it with the `-x` option, comparing output with that in `md5.txt`.

Checksum C and Fortran interface

C routines should include `checksum.h` for prototypes. There is no Fortran header file since there are no functions.

The checksum of a contiguous block of data may be generated with

```
call checksum_simple(len, data, sum)
```

— to get more sophisticated see below and have a look at `fctest.F`.

```
C:   void checksum_init(void);
F77: subroutine checksum_init()
```

Initialize the internal checksum. `checksum_update()` may then be called repeatedly. The result does NOT depend on the number of calls to `checksum_update()` - e.g., the checksum of an array element-by-element is the same as the checksum of all elements (in the same order) at once.

```
C:   void checksum_update(int len, const void *buf)
F77: subroutine checksum_update(len, buf)
      integer len                ! [input] length in bytes
      <anything but character> buf(*) ! [input] data to sum
```

Update the internal checksum with `len` bytes of data from the location pointed to by `buf`. Fortran may use the MA routines for portable conversion of lengths into bytes.

```
F77: subroutine checksum_char_update(buf)
      character*(*) buf          ! [input] data to sum
```

Same as `checksum_update()` but only for Fortran character strings (trailing blanks are included).

```
C:   void checksum_final(char sum[33])
F77: subroutine checksum_final(sum)
      character*32 sum           ! [output] checksum
```

Finish generating the checksum and return the checksum value as a C (null terminated) or Fortran character string.

```
C:   void checksum_simple(int len, const void *buf, char sum[33]);
F77: subroutine checksum_simple(len, buf, sum)
      integer len                ! [input] length in bytes
      <anything but character> buf(*) ! [input] data to sum
      character*32 sum           ! [output] checksum
```

Convenience routine when checksumming a single piece of data. Same as:

```
      call checksum_init()
      call checksum_update(len, buf)
      call checksum_final(sum)
```

```
F77: subroutine checksum_char_simple(buf, sum)
      character*(*) buf          ! [input] data to sum
      character*32 sum           ! [output] checksum
```

Same as `checksum_simple()` but only for Fortran character strings (trailing blanks are included).

7.3.9 Source version information

```
util_version
```

```
      subroutine util_version
```

By default this routine does nothing since it is expensive to construct. If you execute the command `make version` in the `util` directory then all configured source files will be processed to generate a copy `util_version` which when called will printout the name and version information of all source files, organized by module. Of course, you'll also have to relink.

7.3.10 Times and dates

```
util_cpusec
```

```
      double precision function util_cpusec()
```

This function returns the `cputime` in seconds from the start of the process. On some systems this number will be the same as the wall time. The resolution and call overhead will also vary. This routine should provide the most accurate `cputime`. On nearly all systems the clocks are not synchronized between processes.

util.wallsec

```
double precision function util_wallsec()
```

Routine to return wall clock seconds since the start of execution. On nearly all systems the clocks are not synchronized between processes. Resolution will also vary.

util.date

```
subroutine util_date(date)
character*(*) date          [output]
```

Routine to return to Fortran the current date in the same format as the standard C routine `ctime()`. Note that there are 26 characters in this format and a fatal error will result if the argument `date` is too small.

7.3.11 System operations and information

util.hostname

```
subroutine util_hostname(name)
character*(*) name          [output]
```

Returns in `name` the hostname of the machine. A fatal error results if `name` is too small to hold the result — 256 characters should suffice.

util.file.unlink

```
subroutine util_file_unlink(filename)
character*(*) filename      [input]
```

The calling process executes the `unlink()` system call to delete the file. If the file does not exist then it quietly returns. If the file exists and the `unlink` fails then it aborts calling `ga_error()`.

util.file.copy

```
subroutine util_file_copy(input, output)
character*(*) input         [input]
character*(*) output        [input]
```

The calling process copies the named input file to the named output file. All errors are fatal.

util.system

```
integer function util_system(command)
character*(*) command       [input]
```

The calling processes execute the UNIX system call `system()` with `command` as an argument. This executes `command` inside the Bourne shell. Returned is the completion code of the command (typically 0 on success). If this functionality is not supported on a given machine then a non-zero value (1) is returned.

7.3.12 C to Fortran interface

`string_to_fortchar` **and** `fortchar_to_string`

```
#ifdef CRAY
#include "fortran.h"
int string_to_fortchar(_fcd f, int flen, char *buf);
int fortchar_to_string(_fcd f, int flen, char *buf,
                      const int buflen);
#else
int string_to_fortchar(char *f, int flen, char *buf);
int fortchar_to_string(const char *f, int flen, char *buf,
                      const int buflen);
#endif
```

These C callable routines automate the tricky conversion of C null-terminated character strings to Fortran character strings (`string_to_fortchar`) and vice versa (`fortchar_to_string`). The Cray interface is complicated by their use of character descriptors. We describe the non-Cray interface.

- `f` — a pointer to the Fortran character string
- `flen` — the length of the Fortran string (i.e., number of storage locations in bytes)
- `buf` — pointer to the C character string
- `buflen` — the size of `buf` (i.e., number of bytes in the buffer).

In converting to C format, strings are stripped of trailing blanks and terminated with a null-character. In converting to Fortran format, the null character is removed and the Fortran string padded on the right with blanks.

7.3.13 Debugging aids

`ieetrapp`

7.3.14 Miscellaneous BLAS-like operations

`dabsmax.F` — to be removed

`dabssum.F` — to be removed

`rsg.f` — Eispack diagonalization routine — should use Lapack equivalent instead

Initializing arrays — `dfill` **and** `ifill`

```
subroutine dfill(n, s, x, ix)
integer n          ! [input] No. of elements to initialize
```

```

double precision s      ! [input]  Value to set each element to
double precision x(*)  ! [output] Array to initialize
integer ix             ! [input]  Stride between elements

subroutine ifill(n, i, m, im)
integer n              ! [input]  No. of elements to initialize
integer i              ! [input]  Value to set each element to
integer m(*)           ! [output] Array to initialize
integer im             ! [input]  Stride between elements

```

Initialize n elements of the array $x(*)$ to the value s . The stride between elements is specified by ix which should be specified as one for contiguous data. Routine `ifill()` should be used for integer data.

7.4 Print Control

All modules should use the same print control mechanism, to provide both uniformity and flexibility. The routines in `util_print` do this. Using these routines ensures that

1. All modules understand the print levels
 - none (i.e., no output except for catastrophic errors, such as inconsistent data, or failure to converge)
 - low
 - medium = default
 - high
 - debug
2. Printing of specific quantities can be directly enabled or disabled from the input using already existing input routines
3. Modules operate independently and printing is controllable via context

The following example shows how it works. Inside the SCF input, include the directive

```

print low basis "final eigenvectors"
noprnt title

```

This sets the overall SCF print level to low, forces printing of the final eigenvectors and basis, and disables printing of the title.

The implementation is very simple. Each module defines (using provided input routines) one or two entries in the database which enable/disable printing

- `<module>:print` — list of names to enable print
- `<module>:noprnt` — list of names to disable print

The special values (none, low, ...) are recognized in the list of print keywords and are used to adjust the print level. The parsing of this list is encapsulated in the routine `util_print_rtdb_load()`. To support multiply nested modules, a stack of print options is maintained.

The coding needed in a module using print control is then simply:

- In the input routine, upon detecting a line with either the `print` or `noprint` directive, insert the call to `util_print_input`;

```
call util_print_input(rtdb, 'module_name')
```

- Set the default print level for a new module at the beginning of a module

```
call util_print_push
call util_print_rtdb_load(rtdb, 'module_name')
```

The routine `util_print_push()` sets the default print level for a new module. The routine `util_print_rtdb_load` then reads in any input parameters.

To control printing within a module, the following commands must be specified.

```
#include "util.fh"

if (util_print("name", level)) then
  write out data associated with "name"
endif
```

Level is one of the prespecified print levels (`print_none`, `print_low`, ...; see `util/printlevels.fh` for actual values).

At the end of a module, the output file can be invoked by the call,

```
call util_print_pop
```

An example of this usage is as follows;

```
#include "util.fh"

call util_print_push
call util_print_rtdb_load('scf')
if (util_print('information', print_low)) then
  write(6,*) ...
endif
...
call util_print_pop
```

If an application wants more direct control over printing there are routines to explicitly control the print level and to enable/disable printing of named items.

The required integers have been declared in `util.fh` and `util_print` has been declared an external logical valued function. The required integers are

- `print_none` (warning: paradoxical as it may seem, this argument for `util_print` will force printing even if none is asked for!)
- `print_low`
- `print_medium`

- `print_high`
- `print_debug`
- `print_never`
- `print_default = print_medium`

and to declare `util_print` as an external logical valued function.

7.4.1 Other Relevant Routines

`util_print`

```
logical function util_print(name, level)
character*(*) name    [input]
integer level         [input]
```

The value `.true.` is returned if `level` is less than or equal to the current print level or the printing of `name` was explicitly enabled and the printing of `name` has not been explicitly disabled. Otherwise `.false.` is returned. The current print level is set by either `util_print_rtdb_load` or `util_print_set_level`.

`util_print_input`

```
subroutine util_print_input(rtdb, prefix)
integer rtdb          [input]
character*(*) prefix [input]
```

The input routine of a module should call this routine upon detecting either the `print` or `noprint` directives. It should pass the name of the module in the character string `prefix`. This is prepended to actual entries made in the database.

`util_print_push`

```
subroutine util_print_push
```

Call this routine on entry to a module to push the print stack down. A call to this routine is usually immediately followed by a call to `util_print_rtdb_load`.

`util_print_pop`

```
subroutine util_print_pop
```

Call this routine immediately before exit from a module to pop the print stack to the previous context.

`util_print_rtdb_load`

```
subroutine util_print_rtdb_load(rtdb, prefix)
  integer rtdb          [input]
  character*(*) prefix [input]
```

This routine loads the print information from the database for a module with name provided in `prefix`. The value of `prefix` must match that provided in the corresponding call to `util_print_input`.

This routine is usually called at the start of a module immediately following a call to `util_print_push`.

`util_print_set_level`

```
subroutine util_print_set_level(level)
  integer level          [input]
```

Set the print level to `level`. This routine is rarely called from applications.

Chapter 8

Installing NWChem

This chapter contains guidance on how to obtain a copy of NWChem and install it on your system. The best source for installation instructions is the INSTALL file in the NWChem source distribution, so those instructions will not be repeated here. If you have problems with the installation, you can request help from NWChem support via e-mail at nwchem-support@emsl.pnl.gov.

The following subsections discuss some of the important considerations when installing NWChem, and provide information on environmental variables, libraries, and makefiles needed to run the code.

8.1 How to Obtain NWChem

The NWChem source code tree current release is version 4.5. To obtain NWChem a User's Agreement must be properly filled out and sent to us. The User's Agreement may be found on the NWChem webpages at

<http://www.emsl.pnl.gov:2080/docs/nwchem>

by clicking on the link "Download" and following the instructions as they appear. If you already have an older version of NWChem, new download information may be obtained at the location on the web. If you have any problems using the WWW pages or forms, or getting access to the code, send e-mail to nwchem-support@emsl.pnl.gov.

8.2 Supported Platforms

NWChem is readily portable to essentially any sequential or parallel computer. The source code currently contains options for versions that will run on the following platforms.

NWCHEM_TARGET	Platform	Checked	OS/Version	Precision
SOLARIS	Sun	***	Solaris 2.X	double
IBM	IBM RS/6000	***	AIX 4.X	double
DECOSF	DEC AXP	***	Tru64 4.0-5.0	double
SGI_N32	SGI 64 bit os	***	IRIX 6.5	double

	using 32 ints			
SGITFP	SGI 64 bit os	***	IRIX 6.5	double
CRAY-T3D	Cray T3D		UNICOS	single
CRAY-T3E	Cray T3E	***	UNICOS	single
LAPI	IBM SP	***	AIX/LAPI	double
LINUX	Intel x86	***	RedHat 5.2-6.2	double
	PowerPC	**	RedHat 6.0	double
LINUX64	Alpha	**	RedHat 6.2	double
HPUX	HP	**	HPUX 11.0	double
WIN32	Intel x86	*	Windows98/NT	double

 *Note: LAPI is now the primary way to use NWChem on an IBM SP system.
 If you don't have it get it from IBM.

The environment variable `NWCHEM_TARGET` must be set to the symbolic name that matches your target platform. For example, if you are installing the code on an IBM SP, the command is

```
% setenv NWCHEM_TARGET LAPI
```

Refer to Section 8.3 for additional discussion of environmental variables required by NWChem.

8.2.1 Porting Notes

While it is true that NWChem will run on *almost* any computer, there are always a few jokers in the deck. Here are some that have been found, and were considered sufficiently amusing to be documented.

- from the Intel Paragon OSF/1 R1.2.1 (discovered 16 July 1994 by DE Bernholdt); PGI's compilation system is braindamaged in some fascinating ways:
 1. `cpp860` by default defines `__PARAGON__` and other things, as stated in the man page, but when invoked by `if77`, these things are *not* defined.
 2. `ld`'s `-L` prepends directories to the search path instead of appending, as is done in almost every other unix compiler package
- from the HP-UX 9000/735, also some others (reported 08 Feb 1996 by Jarek Nieplocha):
 1. Avoid the "free" HP C compiler - use `gcc` instead: HP `cc` does not generate any symbols or code for several routines in one of the GA files. To make the user's life more entertaining, there are no warning or error messages either - compiler creates a junk object file quietly and pretends that everything went well. (Karl Anderson says: "(HP) `cc` is worth every penny you paid for it.")
 2. `fort77` instead of `f77` should be used to link fortran programs, since `f77` doesn't support the `-L` flag. Fortran code should be compiled with the `+ppu` flag that adds underscores to the subroutine names.

8.3 Environmental Variables

There are mandatory environmental variables, as well as optional ones, that need to be set for the compilation of NWChem to work correctly. The mandatory one are listed first:

NWCHEM_TOP	the top directory of the NWChem tree, e.g.
<code>setenv NWCHEM_TOP /u/adrian/nwchem</code>	
NWCHEM_TARGET	the symbolic name that matches your target platform, e.g.
<code>setenv NWCHEM_TARGET LAPI</code>	
NWCHEM_MODULES	the modules you want included in the binary that you build, e.g.
<code>setenv NWCHEM_MODULES "all gapss"</code>	

The following environment variables which tell NWChem more about your system are optional. If they are not set, NWChem will try to pick reasonable defaults:

8.4 Makefiles and Libraries

The working assumption in documenting the NWChem code is that the developers will be relatively sophisticated hackers, used to deciphering C and Fortran source code and makefile scripts. The steps for building an executable version of the code are as automated as is practical, with upward of a hundred individual makefiles in the source code directory tree. The developer is advised to look carefully at the top-level makefile `...src/config/makefile.h` to get a feel for what this package can be built for and for some instructions about variables that must be set before invoking `make`.

The makefiles will work only with GNU `make`, and it must be the one that appears first in your path, otherwise dependent "makes" will not invoke the right one.

The code in the CVS repository is set up to run on machines for which working floating point precision is DOUBLE PRECISION. If your platform wants REAL, you need to do a `make dbl_to_sngl` before anything else. (Remember to do `make sngl_to_dbl` before checking anything back into the repository. Doing this before updating from the repository helps avoid problems too. Since code comes out of the repository set up for DOUBLE PRECISION, you will probably have to convert after your update anyway.)

Which routines are subject to precision conversion is governed by a `USES_BLAS` macro in each makefile. Only files listed in this macro will be processed for precision changes. If you write new routines that use `BLAS` or `LAPACK`, be sure to add them to the `USES_BLAS` macro too.

The current philosophy is that by default objects are built without optimization (and normally with debugging on, to facilitate development). Only those routines which are performance hot spots are optimized. This helps avoid stupid compiler bugs which, by empirical observation, are more likely to show up on things other than hard-working number crunching. Each makefile contains an `OBJ_OPTIMIZE` macro which should name those routines that *should* be built with optimization. All the rest should be listed in the `OBJ` macro. The extra arguments used when compiling optimized code are the platform-dependent `[FC]OPTIMIZE` macros in `config/makefile.h`. If you want to turn off all optimization, change these and rebuild.

At present, the package is almost entirely self-contained. The only thing left out is George Fann's PEIGS (Parallel EIGenSolver, pronounced "pigs") library. For most platforms, the makefiles already include pointers to the canonical locations for the PEIGS library for that platform. If your link complains about this library, contact one of the NWChem developers for help, or send e-mail to `nwchem-developers@ems1.pnl.gov`.

To build the package, go to the source directory (`.../src`) in your local copy of the NWChem source directory

<p>NWCHEM_TARGET_CPU</p> <pre>setenv NWCHEM_TARGET_CPU P2SC</pre>	<p>more information about a particular architecture</p>
<p>SCRATCH_DEF_DIR</p> <pre>setenv SCRATCH_DEF_DIR "\'/scratch\'"</pre>	<p>default scratch directory for temporary files, e.g.</p>
<p>PERMANENT_DEF_DIR</p> <pre>setenv PERMANENT_DEF_DIR "\'/home/user\'"</pre>	<p>default permanent directory for files to keep, e.g.</p>
<p>NWCHEM_BASIS_LIBRARY_PATH</p> <pre>setenv NWCHEM_BASIS_LIBRARY_PATH "/bin/libraries/"</pre>	<p>location of the basis set libraries (the builder is responsible to make sure that the library gets to the place), e.g.</p>
<p>LARGE_FILES</p> <pre>setenv LARGE_FILES TRUE</pre>	<p>needed to circumvent the 2 GB limit on IBM (note that your system administrator must also enable large files in the file system), e.g.</p>
<p>JOBTIME_PATH</p> <pre>setenv JOBTIME_PATH /u/nwchem/bin</pre>	<p>directory where jobtime and jobtime.pl will be placed by the builder on IBM SP, e.g.</p>
<p>LIB_DEFINES</p> <pre>setenv LIB_DEFINES -DDFLT_TOT_MEM=16777216</pre> <p>This sets the dynamic memory available for NWChem to run, where the units are in doubles. Check out the Section for MEMORY SCRIPT below.</p>	<p>additional defines for the C preprocessor (for both Fortran and C), e.g.</p>
<p>TCGRSH</p> <pre>setenv TCGRSH /usr/local/bin/ssh</pre>	<p>alternate path for rsh, it is intended to allow usage of ssh in TCGMSG (default communication protocol for workstation builds).</p>
<p>IMPORTANT: ssh should not ask for a password. In order to do that:</p> <ol style="list-style-type: none"> 1) On the master node, run "ssh-keygen" 2) For each slave node, slave_node, <pre>scp ~/.ssh/identity.pub \ username@slave_node:~/.ssh/authorized_keys</pre> 	

tree and type 'make'. (NOTE: If you are messing around only in specific subdirectories, you can do 'make' inside them, and then 'make nwchem_link' in directory `src`. This will save you from traversing *every* directory in the package – but your version of these directories had better be up to date!) Object libraries are constructed in the `lib` directory. The executable ends up in `src` (strange but true).

Before you actually do the build, however, you must set up your own environment properly. In your environment, or on the `make` command line, you must specify two variables

- `NWCHEM_TARGET` — the name of the machine to build for.
- `NWCHEM_TOP` — the full path to the top level NWChem directory.

Look in the `README` file in the top-level NWChem directory for information about supported target platforms. For instance, you might insert the following in your `.cshrc` file on a SUN with SunOS 4.1.3

```
setenv NWCHEM_TARGET SUN
if (! $?NWCHEM_TOP) setenv NWCHEM_TOP $HOME/nwchem
```

(The test to see if `NWCHEM_TOP` is already defined permits you to build in an alternative directory without having to edit your `.cshrc` file). Given just this information the structure of NWChem makes it easy to write a makefile to build a library for a module (e.g. `libddscf.a` for the SCF module) or to add routines into a library shared between multiple modules (`libutil.a` which includes at least the `util`, `geometry`, `basis`, `global`, and `ma trees`).

A minimal makefile looks like this

```
LIBRARY = libminimal.a
OBJ = a.o b.o c.o

include ../config/makefile.h
include ../config/makelib.h
```

The above specifies that the object files are to be generated by compiling available source (C or Fortran, without optimization) and put into the library `libminimal.a` (in the NWChem library directory). Nothing else is necessary. If the library source is not located in a subdirectory of the NWChem `src` directory then the path to the included files must be modified accordingly.

A slightly more complex makefile looks like this

```
LIBRARY = libsimple.a
OBJ = a.o b.o c.o
OBJ_OPTIMIZE = d.o
USES_BLAS = c.o
HEADERS = simple.fh
LIB_TARGETS = test

include ../config/makefile.h
include ../config/makelib.h

test: test.o $(LIBRARY_PATH)
    $(LINK.f) -o $@ $^

a.o b.o c.o test.o: simple.fh private.fh
```

This makefile builds the library `libsimple.a` from four object files of which only one (`d.o`) is optimized. The source associated with `c.o` uses FORTRAN BLAS and will be automatically converted on machines where 64 bit reals are single precision (e.g., requiring `sgemm()` rather than `dgemm()`). The header file `simple.fh` is exported automatically into the NWChem include directory (`src/include`) where it may be included by other modules which reference these routines. Associated with the module is the executable `test` (not made by default) which will be cleaned up automatically with `make clean`. The final line specifies a dependency of certain object files on various header files.

At an absolute minimum, a makefile for a module must do the following:

1. include `../config/makefile.h` — this will define `TARGET` (among other things) from which any machine dependent actions are driven. Because the first rule in this file builds the library, there should be *no* targets before this. (NOTE: if you do not need to use `TARGET` then it is best to include this file at the same point that `makelib.h` is included.)
2. define `LIBRARY` as the name of the library to be made.

The makefile can also do the following, at the developer's option:

1. define `OBJ` as the list of object files to be made without optimization
2. define `OBJ_OPTIMIZE` as the list of object files to be made with optimization. (HINT: It is good practice to keep this list short, to minimize exposure to possible compiler errors.)
3. define `HEADERS` as the list of header/include files to be copied into the common include directory. (WARNING: Do NOT copy `include` files into the `include` directory by hand. This is done automatically, and any separately copied files will be lost.)
4. define `LIB_TARGETS` as any additional files made in this subdirectory that may need cleaning up
5. define `LIB_DEFINES` as any additional defines for the C preprocessor (for both Fortran and C)
6. define `LIB_INCLUDES` as any additional include directories
7. define `SUBDIRS` as any subdirectories to build (NOTE: If you do this, makefiles in subdirectories will need to modify the paths to the include files.)
8. define `USES_BLAS` to be the list of FORTRAN files that need BLAS names converting between single and double (e.g., `ddot` to `sdot`)
9. define any additional targets (e.g., test programs)

Additional things you will need to know how to do:

1. To modify the optimization being used, specify on the command line `C/FDEBUG` or `C/FOPTIMIZE` to override the flags for the `OBJ` and `OBJ_OPTIMIZE` files respectively. E.g.,

```
make FDEBUG="-g -O1"
make FOPTIMIZE="-O3 -Superfast -bugs" FDEBUG="-O1"
```

2. The library is put directly into the NWChem library directory and the full path to the library (if needed by your makefile) is automatically put into the variable `LIBRARY_PATH`.
3. The object files are put directly into the libraries and are not kept elsewhere. This has several implications

- You can (apart from TCGMSG and GA which are being fixed) build executables and libraries for multiple platforms in the same source tree.
- To force recompilation of all source in a given directory `make clean` works by deleting the object files from the library, and deletes the library itself only if it is empty. You have to actually either delete the corresponding library or touch the source files.
- To override the compilation options for a specific file (e.g., because of compiler errors on a specific platform) you must specify the dependency on the object file in the library. Here are two examples. The first one (`dosymops.f`) does not need preprocessing, whereas the second one (`sym_mo_ap_op.F`) does and this must be done explicitly within the rule for this file. This preprocessing is normally done automatically.

```

ifeq ($(TARGET),CRAY-T3D)
    FNOOPT = -dp -Ccray-t3d -wf"-o noscalar,jump,noieeedivide"
$(LIBRARY_PATH)(dosymops.o): dosymops.f
    $(FC) -c $(FNOOPT) $^
$(LIBRARY_PATH)(sym_mo_ap_op.o): sym_mo_ap_op.F
    $(FCONVERT)
    $(FC) -c $(FNOOPT) sym_mo_ap_op.f
    @/bin/rm -f sym_mo_ap_op.f
endif

```

4. The target `clean` will recursively descend subdirectories and delete object files from both the directory and associated library, core files and files defined in `LIB_TARGETS`.
5. The target `realclean` will, in addition to the actions of `clean`, also delete the library and any emacs backup files.
6. The target `cleanF` will recursively descend subdirectories and search for and delete `.f` files for which a corresponding `.F` file exists. This is useful on machines for which the conversion from `.F` to `.f` is done explicitly rather than by the compiler.
7. The target `depend` will recursively descend subdirectories and append onto the end of makefiles dependencies of `.F` and `.c` files on header files that have been included using the notation `#include "filename"`. File includes using angle brackets are assumed to be system files and dependencies are not generated. If the include file is in the local directory, the dependency is generated upon that. Otherwise, a dependency is generated upon a file in the NWChem include directory. Do not insert anything below the line

```
# DO NOT EDIT BENEATH THIS LINE ... GENERATED AUTOMATICALLY
```

since it will be lost the next time that `make depend` is run.

Typing `make` in the top-level NWChem directory will traverse the entire directory tree twice. Once to ensure the include files are up-to-date and then again for the libraries. This can take a while. Therefore, when working on development of a particular a module alone, it is usually much faster to

1. execute `make` in the subdirectory, and
2. execute `make link` in the top NWChem directory.

Note that this approach cannot be used if recompilation of another module is also required, since the special target `link` just relinks the code and does not traverse the directory tree. *After doing a `cvs update` you should:*

1. do a `make depend` if you have not recently, in any directory you have been working in, and

2. do a full make from the top level to ensure that all libraries incorporate any changed common blocks or declarations.

In addition, the top-level makefile has the target `test` which builds the executable `nwchem_test` in the `src/` directory (rather than the usual `$(BINDIR)/nwchem`), and the target `prof` which builds `nwchem_prof` (in `src/`) for performance profiling by linking with the `-p` option.

8.5 Managing NWChem

Concurrent Version System (CVS) is used for configuration management of NWChem at PNNL. Off-site users are not *required* to use this system when doing development work on the code, but it would probably make any collaborative work with EMSL/PNNL developers go much more smoothly. As a matter of simple prudence, is advisable to use some sort of configuration management system for any installation of NWChem, even if the users do not expect to be doing significant development work. The code is far too complex to ever be released on its own recognizance, and users will want to retain the ability to correct errors or make modifications to the code in a controlled and traceable manner.

CVS was chosen as the configuration management system for NWChem mainly because is designed to allow many different developers to work independently on a large code, while greatly mitigating the agony of merging independently developed sets of code changes. For developers working on unrelated modules of the code, the effects of changes made elsewhere in the code can in some cases be completely ignored. The identification of overlapping changes is greatly facilitated, allowing efficient and speedy resolution of conflicts.

This section provides a brief introduction and overview of the CVS system. Developers needing more detailed information on specific CVS commands and capabilities are referred to the on-line documentation included in the CVS code package (i.e., the man pages; consult your system administrator if your system does not have them installed.)

8.5.1 Introduction to CVS

CVS is a configuration control package designed to facilitate multiple developers working on the same software package. It is implemented as a layer on top of RCS and provides a number of useful features which RCS alone does not. The two most important of these features are

- The CVS check-in/check-out mechanism does not require exclusive locks on sources during the development process, and provides for merging of orthogonal changes to the same source file. (Overlapping changes are identified during the merger process, and must be resolved by human intervention.)
- Most CVS commands work recursively on the entire contents of a directory tree, unless specific command line switches are set to limit operation to the local directory.

The following subsections provides a brief description of how NWChem is managed in CVS. It also includes a very concise outline of how CVS works, and a summary of the most useful CVS commands.

8.5.2 The CVS Model

CVS divorces the directory tree in which development takes place from the directory tree in which the master copy of the sources are kept. The latter directory tree is referred to as the *repository*, and it has exactly the same structure as the working directory tree. Where the working tree would have source files, the repository has the RCS files for the sources (e.g., `source.f,v`).

Users working on a program check it out of the repository into their own directories. The individual working copies are by default created giving the user read and write permission on all of the files and can be used directly. When a developer has completed and tested a set of changes, the revised source file(s) can be checked into the repository. The other developers are unaffected by the change to the repository until they update their local copy of the source or check out a new copy. Anyone checking out a new working copy will always get the latest version present in the repository.

Users can poll the repository for changes at any time, and update *their own working copies* with the changes that have been entered in the interval between their last checkout or update and the current version. The repository is entirely unaffected by the update command. The user's private working copy is the only thing that is changed. If any changes were merged into the repository between the user's last check-out or update and the current one, dealing with any inconsistencies or overlaps with changes in the local working copies is the user's problem.

When a user checks a revised source file back into the repository, CVS automatically checks for all differences between the copy being checked in and the current version of the file in the repository. If changes in the new file being checked in overlap or conflict with changes that have been merged into the repository since the last check-out or update of the copy being merged, CVS will not automatically merge the new copy into the repository. If the changes do not overlap or conflict, however, CVS will merge the new source over the existing source in the repository.

In most cases, changes made independently by different developers will not conflict and CVS can handle the merger automatically. When they do conflict, the developer must fix the problem(s) and ensure that the new changes mesh properly with changes others have put into the repository. CVS allows users to work independently on the same source files without unduly interfering with each other, but it is still necessary for developers working on functionally related changes to communicate with each other, even if their source code changes do not conflict.

8.5.3 The CVS Program

CVS is implemented as a single program invoked by its program name `cvs`. A number of options can be specified on the command line following the program name. The command line can also include subcommands, which come after any options that may be specified. The syntax of the command line is as follows;

```
cvs [cvs_options] subcommand [subcommand_options] [arguments]
```

The man pages list the applicable options for the `cvs` command itself and for each subcommand.

CVS must be told of the location of the repository. This can be done with the `cvs` option `-d` (e.g., `-d /msrc/proj/mss`) or by setting the environment variable `CVSROOT`. Although the CVS man pages implicitly assume that a single repository will be used for all projects under CVS control, this is not strictly necessary. Different repositories can be defined by the simple expedient of changing the definition of `CVSROOT`.

CVS is designed to deal with source files organized into *modules*. A module is basically a collection of source files that form some sort of sensible unit and probably should be worked on as a group. The module can simply be the name of a directory within the repository (e.g. `nwchem` or `nwchem/src`), or it is defined as a collection of selected bits and pieces of the directories within the repository. For example, it might eventually be desirable allow users to check out NWChem without getting certain parts of the package, such as Argos sources or the distributed data package. Specific modules could be defined to give these results.

The procedure for checking out a working copy of the code stored in CVS repository is very simple. From the directory where the working copy is to be checked out to, a given module can be checked out using the following command;

```
cvs co module_name
```

To check out NWChem, the command is simply,

```
cv$ co nwchem
```

The working version of a module in a local directory can be compared with the source in the repository using the command

```
cv$ diff
```

This command accepts the same arguments as `rcsdiff`, and will compare particular files itemized on the command line or the entire directory tree recursively. (The command `cv$ log` is the equivalent of the RCS `rlog` command and operates similarly to `cv$ diff`.)

Changes made to the repository after a particular working copy has been checked out can be merged into the files on the working directory using the command

```
cv$ update
```

This command is recursive throughout the checked-out directory tree. It flags modified files in the working directory with an "M". Files that have changed in the repository since the last update are marked with a "U". New files in the working directory that do not occur in the repository are marked with a "?". There are a number of other codes for other circumstances, which are detailed in the man pages. A particularly useful command is the option to check on what has changed since the last update of the working directory, but without merging any of the changes from the repository. This can be done using the command

```
cv$ -n update
```

To remove a file from a repository controlled by CVS it first must be removed from the directory with the Unix `rm` command. The command `cv$ rm` is then used to notify CVS. When this (nonexistent) file is checked in at the next update, it will be moved to a special place in the repository where it can be recovered if old versions which require it are checked out, but where it will not appear in future working copies.

To add a new file to a repository controlled by CVS, the command is `cv$ add`. Like `cv$ rm`, the actual addition takes place at the next check-in. As with the first RCS check-in of a file, `cv$ add` will prompt for a description of the file (not a log message – that happens at check-in). New directories must also be added with `cv$ add`, but no description is requested.

The command to check-in changed files is `cv$ ci`. As with `cv$ diff`, CVS will accept particular file names or search recursively through the directory tree looking for files that have been modified. CVS prompts the user for a log message for the files being checked in. If the specific filenames are listed on the command at check-in, only a single log message that applies to all of them is required. If CVS must search and compare to find the files that are being checked in with changes, it prompts for a log message for all of the modified files in a given directory. The `EDITOR` environmental variable is used to decide which editor to bring up to enter the log message.

CVS automatically tracks which version(s) of the source a newly checked-in working copy is based on. This allows it to determine whether the changes would be checked in on a branch or the main trunk, etc.

To delete an entire working directory, the simplest approach is to use the command `cv$ release -d nwchem` in the directory above it. This command checks the files in the working directory, looking for changes that have not yet been checked back into the repository. This is to ensure that changes are not accidentally abandoned. If no inconsistencies are found, CVS deletes the entire directory tree. (NOTE: leaving off the `-d` just does the check without deleting anything.)

The above commands provide a convenient starting point for learning how to use CVS. However, users wishing to obtain a more thorough understanding of the capabilities of the system should read through the CVS man pages to get

a better feel for everything that can be done. (Hint: If you are unsure of what a command will do, try it first with a `-n` option on `cvs` itself. This is like “make -n”, which reports what it would do if invoked without the `-n`. But it does not actually do anything. Honest.)

8.5.4 Summary of CVS commands

The following is provided as a quick reference guide to CVS. A more detailed short-form reference is available in `nwchem/doc/cvshelp.man`. Detailed documentation can be obtained using the command `man cvs`.

`setenv CVSROOT /msrc/proj/mss` — in `csh` this defines the path to the CVS repository. Put this in your `.cshrc` or `.mycshrc`.

`cvs co nwchem` — checks out the entire source for NWChem into the directory `nwchem`. The repository is unaffected.

`cvs -n update` — compares the contents of the current directory and all subdirectories against the repository and flags files according to their status:

- ? — the file is not maintained by CVS.
- M — your checked-out version differs from the original (i.e., you edited it).
- U — your checked-out version is out-of-date and needs updating.
- C — potential conflict. You have changed this file and the source in the repository has also changed.
- File not listed — your source is the same as that in the repository.

Neither the repository nor your source are changed.

`cvs update` — updates the contents of the current directory and all subdirectories with the latest versions of the source, again flagging files according to their status. *You are responsible for correcting files that CVS flags as containing conflicts between edits you and others have made.* However, CVS handles all other merging. New files will also be added to your source, but to get new directories you must append the `-d` flag. Your source is changed; the repository is unaffected.

`cvs diff filename` — generates differences between the file and the version of the file you checked out (i.e., it indicates edits you made). If you want to compare against the most recent version in the repository use `cvs diff -r head filename`. Neither the repository nor your source are changed.

`cvs add filename` — adds a new file to the repository. The new file is not actually added until you execute `cvs commit`. Changes CVS internal information in your source tree but does not affect the repository.

`cvs rm filename` — to delete a file from the repository delete it from your source with the standard UNIX `rm` command then tell CVS to delete it with this command. The file is not actually removed until you execute `cvs commit`. Changes CVS internal information in your source tree but does not affect the repository.

`cvs commit` — this is the only command that affects the repository. Before committing changes and updating the repository with changes in a list of files or the current directory tree you must

- ensure that all of your sources are up-to-date with respect to the repository by using `cvs update`,
- resolve all conflicts resulting from the update, and
- ensure that the updated code functions correctly.

Commit will verify that all source is up-to-date before proceeding. Then it will prompt (using an editor) for log messages describing the changes made. Be as detailed as possible.

8.5.5 Troubleshooting CVS

Under no circumstances edit, move, delete or otherwise mess with files in the NWChem repository. Contact NWChem support at `nwchem-support@emsl.pnl.gov` to report problems.

CVS version information is "sticky". That is, CVS usually remembers the specific version checked out to a working directory. This can be confusing, since the output of such commands as `cvsexec update`, etc., will not always refer to the latest (or head) version. Changes can magically disappear. This may be desirable. Or it may not be. The option `-A` forces the system to look at the latest version when doing the update. The form of the command is

```
cvsexec update -A
```

If CVS is interrupted, or there is an AFS to NFS translator problem, it may occasionally leave locked files in the CVS repository, causing subsequent commands to wait forever, printing messages indicating it is waiting for someone to relinquish access to a specific directory. Fixing this requires deleting files from the repository. Contact `nwchem-support@emsl.pnl.gov` for help.

It is unclear if this next problem still exists within EMSL but it may arise elsewhere. Because of a problem with the AFS version of the `ci` command, which is used by CVS, `/usr/local/lib/rcs/diff` must be available on the system. The easiest way to do this is to create the `/usr/local/lib/rcs` directory and put in it a symbolic link to the GNU diff program, `/msrc/bin/diff`.

Chapter 9

Developing New Modules and Enhancements

When developing new modules or enhancements to the code, the developer must pay careful attention to design and coding style. This chapter offers guidance on general design requirements and defines coding style rules. In addition, the specific considerations for inserting a new module into the code are presented in detail.

9.1 General Design Guidelines

The complexity of NWChem and the large number of developers working with the code makes it highly advisable to consider very carefully the effect of even minor changes in the code. This is particularly the case when considering changes that may impact the performance of the code in a parallel computing environment. The first, last, and only rule is *Think before you code!* Then think again. Nothing will be as simple as you thought at first.

However, the code is not likely to develop new capability on its own, so you must do something, sooner or later. The following list of design guidelines should be followed when adding code to NWChem.

1. Design your code before you start adding code.
2. Set up a performance model that will effectively estimate the CPU, communication and IO costs.
3. Use the model to guide the development of the code. Remember that several algorithms may need to be developed, but the programmer should try to develop an algorithm that will scale in CPU, memory and IO.
4. Use the interfaces and APIs that are defined. **DO NOT** use any of the lower level routines that are used by the APIs. If you deem it necessary to use a lower level routine, first talk to one of the primary NWChem developers.
5. When possible and appropriate, think about creating objects instead of just data structures.
6. If an object is not appropriate, think about creating an API that isolates details from other programmers.
7. Create well defined modules. When possible create and/or use “generic” routines to emphasize reuse of code.
8. Don't be afraid to ask questions. It is better to ask and move in a sensible direction than to not ask and have to redo some or all of the programming.

Remember, fortune favors the bold, but you will live longer and happier if you consult regularly with other NWChem developers and the NWChem Program Manager.

9.2 Coding Style

In a project this large, it is necessary to impose some standards on the coding style employed by developers. The primary goal of these standards is not to constrain developers, but to enhance both the quality of the final product and its functionality.

Code quality is somewhat subjective, but clearly embraces the ideas of

- correctness
- maintainability
- efficiency
- readability
- re-usability
- modularity
- ease of integration with other packages
- speed of development
- density of bugs
- ease of debugging
- detection of errors at run time
- exposure of available functionality
- ease-of-use of the API

Compromise is clearly necessary. We are interested in high-performance, so some key kernels may sacrifice readability (but perhaps not modularity) for efficiency, but most code (i.e., 99.9%) is not an inner loop in need of such optimization, as long as the overall structure is correct.

The single most important thing you can do to achieve quality code has little to do with programming style. It is the *design* — putting in the necessary thought and effort before even a single line of code is written.

The following subsections present the recommended "Do's and Don'ts" for programming modules and modifications in NWChem. The recommendations are organized by a 'top-down' logic, to reflect the most efficient order in thinking about the various considerations the developer must keep in mind when designing a new piece of code.

9.2.1 Version information

Each source file should include a comment line that contains the CVS revision and date information. This is accomplished by including a comment line containing the string `Id`. CVS substitutes the correct version information each time the file is checked out or updated. These lines are processed from the source and can be output at runtime to aid in bug-tracking.

9.2.2 Standard interface for top-level modules

In order to allow for automatic configuration of various modules in a compilation of NWChem (to control the size of the executable in memory-critical situations), all top-level modules must have a standard interface. Currently it looks like this;

```
logical function MODULE(rtdb)
```

The argument `rtdb` is the handle for the run-time database. The function should return `.true.` or `.false.` on success or failure respectively.

The only sources of information for a module are the database, or files with names that can be inferred from data in the database or from defaults. Furthermore the naming of database entries is standardized such that:

- The string with which database entries are prefixed must be lowercase and match the module name used in the input. E.g., input for the SCF module appears in the `scf ; . . . ; end` block and the prefix used in the database is `scf`. This is so that the user can delete all state information using the `UNSET` directive.
- Common quantities (such as energy, gradient, ...) should be stored using that name. E.g., `scf:energy`.

9.2.3 No globally defined common blocks

Use of global variables (e.g., common blocks) is generally a bad idea. Such variables break modularity, form hidden dependencies and make code hard to reuse and maintain. *Do not use common blocks to pass data between routines.*

However, common blocks are very useful in supporting a modular programming style which encourages code reuse and improves maintainability. To this end common blocks can be used to hide data behind a subroutine interface so that access to the common is limited to a few tightly integrated routines. The benefits of using common blocks (smaller argument lists, static data allocation, contiguous memory layout) can thus, with care, be realized without any problems. Examples of this include the basis, geometry, RTDB, integral, symmetry, global array, message passing, SCF, optimizer, input, and MP2 libraries.

9.2.4 Naming of routines and common blocks

To avoid name clashes and for easy identification, prefix all subroutine, function and common block names with the name of the module they are associated with. For instance,

- `rtdb...` — run-time database
- `ma...` — memory allocator
- `ga...` — global array
- `scf...` — SCF
- `stpr...` — Stepper (geometry optimization)

9.2.5 Inclusion of common block definitions

All common block definitions, including typing of variables in the common, are to be made once only in a single file (a `.fh` file), that is included in other source using the C preprocessor. The `include` file should document the

meaning of all variables. This helps ensure that variables in a common block are consistently named and dependencies of routines on common blocks are easily generated and maintained.

9.2.6 Convention for naming `include` files

All `include` files should be named using the following conventions,

- Use `.fh` for files that can be included only by Fortran routines
- Use `.h` for files that can be included by C routines only, or for files that are included by both C and Fortran routines

9.2.7 Syntax for including files using the C preprocessor

A very important distinction hinges on the seemingly trivial difference between the two different `include` forms,

- `#include "filename"`
- `#include <filename>`

According to Kernighan and Ritchie:

”If the *filename* is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in `<` and `>`, searching follows an implementation-defined rule to find the file.”

For this reason, and by common convention, only system-defined `include` files are included using angle brackets. Those `include` files that are defined within an application are included using quotes. The automatic generation of dependencies of source files upon `include` files within NWChem *relies* upon this convention.

9.2.8 No implicitly typed variables

The command `implicit none` should appear at the top of every routine in the NWChem code. No other implicit statements are permitted and all variables must be explicitly declared. *This rule should be religiously observed in new code.* It

- lets the compiler help you find typos and other errors
- makes the code more readable and more maintainable
- provides a natural point to document arguments and local variables
- makes silly variable names like `iii`, `iil` both obvious and even more embarrassing when others catch you doing it

When integrating existing code, this rule may seem to be more work than it is worth, but several bugs in existing code have been found in this fashion.

9.2.9 Use `double precision` rather than `real*8`

`REAL*8` is not standard Fortran. `DOUBLE PRECISION` is the standard, it is usually what you want, it is more portable, and standardization of declarations enables us to perform necessary code transformations more readily.

9.2.10 C macro definitions should be in upper case

NWChem uses the ANSI C preprocessor to handle machine dependencies and other conditional compilation requirements. By forcing all C macros to be upper case the code is made more readable and we also avoid potential accidental munging of Fortran source. This practice is consistent with conventional use of the preprocessor in C programs.

9.2.11 Fortran source should be in lower or mixed case

This convention is complementary to the above C macro convention. If there are no fully upper-case Fortran tokens then there can be no accidental conflict with the C preprocessor.

9.2.12 Naming of variables holding handles/pointers obtained from MA/GA

So that these critical variables are immediately recognizable, the following conventions are recommended.

- handles obtained from MA should be prefaced with `l_`
- pointers (into `dbl_mb()`, etc.) obtained from MA should be prefaced with `k_`
- handles obtained from GA should be prefaced with `g_`

Alternatively, you can insert comment lines describing the variables at the point of declaration, if you do not want to follow these conventions.

9.2.13 Fortran unit numbers

All references to Fortran I/O units should be done with parameters or variables instead of hardwired constants. For the “standard I/O” units, corresponding to the C `stdin`, `stdout`, and `stderr`, you should include the file `stdio.fh` and use the variables `luin`, `luout`, and `luerr` instead of 5, 6, and 0.

The code uses very few other files, and there is no organized list of parameter names for non-standard I/O units. Users are free to use parameter names that make sense to them, so long as they adhere to the convention. Using parameters rather than hardwired integer constants helps insure that I/O unit designations can be changed easily if needed, and may facilitate moving to a more general convention in a future version of the code.

9.2.14 Use standard print control

All modules should understand the `PRINT` directive and accept at least the following keywords for this

- `none` — no output whatsoever except for error messages
- `low` — minimal output; e.g., title, critical parameters and a final energy

- `medium = default` — usual output
- `high` — extra verbose output
- `debug` — anything useful for diagnosing problems

Ideally all applications should control most printing via the print control routines (see Section 7.4). A uniform look and feel is important.

9.2.15 Error handling

All fatal errors should result in a call to `errquit()` (see Section 7.3.2), which prints out the string and status value to both standard error and standard output and attempts to kill all parallel processes and to tidy any allocated system resources (e.g., system V shared memory).

9.2.16 Comments

The use of comment lines is strongly recommended in all coding. Commented code is easier to read, and often is easier to debug, maintain, and modify. Liberal use of comments is particularly important in NWChem, since it is used by a large and diverse group of people, it is constantly being modified as capabilities are added and refined, and it has only a limited amount of detailed documentation.

Requirements for in-source documentation are given in detail in Chapter 11 but the general recommendation for comment lines in the code is the more the merrier. At a minimum the source code should be able to provide the following information,

- terse comments at the top of each subroutine to describe (accurately!) its function,
- documentation of dependencies/effects on state that are not passed directly through its argument list (e.g., files, the database, common blocks)
- descriptions of all arguments, including the flow of information (i.e., label arguments as input or output, or input-output)
- documentation of local variables with functions that are not apparent from their names, or which have an algorithmic role that is opaque or obscure

In some circumstances, comments at the top of a routine can be quite lengthy since this is a *very* good place to store details of the algorithm. Automatic generation of documentation from code comments is being designed, but this will produce useful documentation *only if developers write clear and concise commentary in the code as they work*.

The following partial listings show examples of minimalist in-source documentation using comment lines. It would not be difficult to say more. The rule of thumb should be "from those who have much, more will be expected". The more important a routine is to a particular algorithm, the more it does in the way of carrying out the solution, the more detailed and voluminous should be its comment lines.

Example of comments in a simple routine:

```
logical function bas_numbf(basis,nbf)
implicit none
integer basis      ! [input] basis set handle
integer nbf        ! [output] number of basis functions
```

```
*
* nbf returns the total number of functions.
* Returns true on success, false if the handle is invalid
*
```

Example of comments in a less simple routine:

```
      subroutine sym_symmetrize(geom, basis, odensity, g_a)
C$Id: codingsty.tex,v 1.4 1998/12/15 16:22:36 d35162 Exp $
      implicit none
      integer geom, basis ! [input] Handles
      integer g_a        ! [input] Handle to input/output GA
      logical odensity   ! [input] True if matrix is a density
c
c   Symmetrize a skeleton matrix (in a global array) in the
c   given basis set.
c
c   A<- (1/2h) * sum(R) [RT * (A + AT) * R]
c
c   where h = the order of the group and R = operators of the
c   group (including the identity)
c
c   Note that density matrices transform according to slightly
c   different rules to Hamiltonian matrices if components
c   of a shell (e.g., Cartesian d's) are not orthonormal.
c   (see Dupuis and King, IJQC 11, 613-625, 1977)
```

9.2.17 Message IDs

The use of tags/IDs/types on messages is strongly suggested. If all messages with the program have distinct types and the message-passing software forces the types of messages to match between sender and receiver, then there is a way to prove that messages are being sent and received correctly. If they are not, a runtime error will be detected. This is especially important to NWChem since the code makes use of many third party linear algebra libraries that do a lot of message passing.

Modules which do a significant amount of messaging should reserve a section of the message ID space for their own use (e.g., GA or PEIGS). Most modules, however, do only a small amount of messaging. For these, the `include` file `msgids.fh` should be used to reserve individual message IDs. This file defines Fortran parameters for message IDs used in most NWChem *Hardwired message IDs should not be used in any NWChem routine.*

9.2.18 Bit operations — `bitops.fh`

The following bitwise operations (see Table 9.1 for definitions) are the recommended standards for use in NWChem.

- `ior(i, j)` — inclusive OR
- `ieor(i, j)` — exclusive OR
- `iand(i, j)` — AND

- `not(i)` — NOT or one's complement
- `rshift(i,nbits)` — right shift with zero fill
- `lshift(i,nbits)` — left shift with zero fill

ior	ieor	iand	not	lshift	rshift
110	110	110	10	10111011	10111011
100	100	100		2 bits	2 bits
110	010	100	01	11101100	00101110

Table 9.1: Effect of Bit Operations

These operations are readily generated using in-line functions from most other definitions. The shift examples in Table 3.1 use an eight bit word written with the most significant bit on the left. All operations operate on full integer words (32 or 64 bit as necessary) and produce integer results. The declarations and any necessary statement functions are in `bitops.fh`. The presence of data statements makes it impossible to have a single include file make declarations and define statement functions. To circumvent this the declarations are in `bitops_decls.fh` and the statement functions are in `bitops_funcs.fh`.

9.2.19 Blockdata statements and linking

At least one machine (the CRAY-T3D) discards all symbols that are not explicitly referenced, even if other symbols from the same `.o` file are used. Thus, `BLOCK DATA` subprograms are not linked in. One fix to this is to declare each `BLOCK DATA` subprogram as an undefined external on the link command, but this makes the link command depend on the list of modules being built. An alternative mechanism that works on the T3D is to reference each `BLOCK DATA` subprogram in an `EXTERNAL` statement within a `SUBROUTINE` or `FUNCTION` that is guaranteed to be linked if any reference is to be made to the `COMMON` block being initialized.

This is being redesigned.

Chapter 10

Testing the Program

The Quality Assurance (QA) tests are designed to test most of the functionality of NWChem. As such, it is useful to run at least some of the tests when first installing NWChem at a site. It is imperative to run these tests when porting to a new platform.

The directions given below for running the tests are for systems without a batch system. If you have a batch system, check out the contrib directory to see if there is an appropriate batch submission script. You will then need to run each of the tests separately and check the results (the `nwparse.pl` script can be used for the quantum (QM) and pspw tests for this purpose).

Here are some steps and notes on running the QA tests:

1. Set the environment variable `NWCHEM_EXECUTABLE` to the executable you want to use, e.g.

```
% setenv NWCHEM_EXECUTABLE \  
$NWCHEM_TOP/bin/${NWCHEM_TARGET}_${NWCHEM_TARGET_CPU}/nwchem
```

2. If you compiled without MPI (this is the default way to build NWChem), you will need to:

- (a) Set the environment variable `PARALLEL_PATH` to the location of the parallel program, e.g.

```
% setenv PARALLEL_PATH \  
$NWCHEM_TOP/bin/${NWCHEM_TARGET}_${NWCHEM_TARGET_CPU}/parallel
```

- (b) Run the QM tests sequentially using the `doqmtests` script. Note that you may want to comment out the largest tests at the bottom of the `doqmtests` file on slower machines or machines without much memory.

```
% doqmtests >& doqmtests.log &
```

- (c) Check the `doqmtests.log` file for potential problems. While running, the test scripts place files in the `$NWCHEM_TOP/QA/testoutputs` directory. You may wish to clean out this directory after checking that everything is working. If a job did not work, the output can be found in the `$NWCHEM_TOP/QA/testoutputs` directory. If the problem seems significant and/or you are unsure whether NWChem performed the calculation correctly, please send a message to `nwchem-support@emsl.pnl.gov` with details about your computer, the environment variables that were set when you compiled NWChem, and the output of the calculation that you are concerned about.

- (d) Run the QM tests in parallel by editing the `doqmtests` script so that `"procs #"` is placed after the `runtests.unix` commands (substituting in the number of processors that you want to use for #). E.g.

```
runtests.unix procs 2 h2o_dk u_sodft cosmo_h2o ch5n_nbo h2s_finite
```

- (e) Again check the log for potential problems.
- (f) Run most of the molecular dynamics (MD) tests using the `runtest.md` script. Note that this script assumes that you have a `/tmp` directory and that you want to use 2 processes. Both of these may be changed.

```
% runtest.md >& runtest.md.log &
```

- (g) Check the log (`runtest.md.log`) for potential problems.

3. If you compiled with MPI, you will need to

- (a) Set the environment variable `MPIRUN_PATH` to the location of `mpirun` if it is not in your path, e.g.

```
% setenv MPIRUN_PATH /usr/local/bin/mpirun
```

- (b) If the `mpirun` processor definition option is not `-np`, you will need to set the environment variable `MPIRUN_NPOPT` to the appropriate flag, e.g.

```
% setenv MPIRUN_NPOPT -n
```

- (c) Run the `doqmtests` and `runtest.md` scripts as described above, but first edit those files to substitute `"runtests.mpi.unix"` for `"runtests.unix"` and `"runtest.unix"`
- (d) Check the log for potential problems.

Chapter 11

Documenting New Modules and Code Enhancements

"Yes, of course we must document the code." –rjh

In keeping with the top-down approach outlined in Chapter 9 for developing new modules or enhancements to the code, the general approach to documentation should also reflect forethought and planning. The purpose of documentation is not only to communicate clearly and efficiently to new developers the existing structure of the code, but also to define the desired structure and organization of new code. Activities that require documentation fall into three broad categories;

1. development of a new capability (such as a new molecular calculation module) within any of the architecture levels,
2. development of a new subroutine or function,
3. modification or enhancement of an existing molecular calculation module, API, object or subroutine

Basically, whenever functionality is added or modified in the program, it must be documented. The basic philosophy of documentation in NWChem is to have as much of the documentation as possible in the source code itself. This is where one most likely would be looking when most in need of guidance. This approach also holds forth the shining hope that one day we will develop a system that allows the in-source documentation to be automatically extracted for inclusion in updated versions of this manual. However, for the high level modules in the code, the level of detail required for the documentation to be useful will generally result in too much verbiage to be readily included in the source code as comment lines. So some additional documentation may always be necessary.

There are two separate issues that must be discussed for documentation. The first is the content of the documentation. The second is the way the documentation must be formatted (i.e. a style guide). Both of these will be discussed in the following sections.

11.1 Content of the Documentation

The level of documentation is, by necessity, different for molecular calculation modules, modeling or development tools, and subroutines or functions. The molecular calculation modules represent a high level of functionality and can be considered as requiring the highest level of documentation. These modules in general require documentation

of the underlying theory and overall solution method, as well as details of the implementation of the algorithm. The modeling or development tools are used by various modules and may also use other modules. These tools also require a relatively high level of documentation. The documentation must describe the use and function of the tool and give detailed information on the abstract programming interfaces. Individual subroutines or functions that are not in themselves main modules or tools in general require only descriptive documentation. This usually consists of, at minimum, their input and output, and some description of their purpose. The following subsections delineate the information that should be documented at each level.

11.1.1 Documentation of a Molecular Calculation Module

The documentation of a new molecular calculation module for NWChem will generally require creating a stand-alone Latex document. This document should reside in the directory containing the source code for the module (or in a subdirectory named `doc` within that directory). It is the responsibility of the developer to write the documentation as an integral part of the development process, and then to keep it current as changes or modifications are made in the module. The developer can, as an alternative to writing a separate LaTeX document, put the documentation directly in the main subroutine of the module. Whichever approach is used, however, the documentation should conform to the following template.

Chapter 3 contains documentation of existing molecular calculation modules in the code, and can be referred to for guidance on style and appropriate level of detail when developing the documentation for a new module. In its current form (as of 11/10/98), this documentation is relatively sparse and incomplete, and should not be looked upon as an ideal to be emulated. Developers are encouraged to write in their own unique style, so long as the necessary information is communicated in a clear and concise manner. (*"A foolish consistency is the hobgoblin of small minds."*)

Module Documentation Template

- Introduction
 - Give a brief, but *non-recursive* description of the module, noting whatever might be unique about it, or what makes it worth the trouble of adding it to NWChem.
 - Note source of the underlying work, listing collaborators, if any, with full bibliography (if available); note any significant geneological information, if relevant.
- Overview
 - Describe the theory used by the module, and the operation(s) it performs.
 - Describe how the module interacts with the rest of NWChem.
- Solution Procedure
 - Describe the numerical solution used by the module, including how it interfaces with the NUMA model for memory management used by NWChem.
 - If the new module calls other modules in the code, describe in detail how this interface occurs (e.g., is there a specific calling order that the module relies on?).
- Performance Evaluation
 - Describe the testing of this module, and it's performance as evaluated by the criteria defined in Chapter 10.
 - Present results of applications showing the capability of the module, and where possible, comparing to results of other modules. (This may be for validation as well as evaluation.)

When modifying or enhancing an existing module, the documentation should also be updated to match the new form of the module. If by chance the module does not yet have adequate documentation, this is an opportunity for you to gain merit (in this world *and* the next) by providing the missing information, in addition to supplying the documentation for the new coding.

11.1.2 Documentation of Modeling or Development Tools

This section presents a content template for documentation of new modeling or development tools. The template can be used for in-source documentation, or as the outline for a separate LaTeX document. In-source documentation generally makes the most sense for modeling tools or development tools. The developer is also free to write a stand-alone LaTeX document, as these will not be spurned. However, the in-source documentation is the preferred method of documentation for modules at this level, since this is perhaps the only way to stack the odds in favor of continual updating of the documentation as the code is changed. Any documentation separate from the source code should reside in the same directory as the source code, to keep from losing it in the forest of the NWChem directory tree.

The template is based on the structure of the in-source documentation developed by Ricky Kendall for the Integral API. The format is general enough, however, to be applicable to almost any feature that might be added to the Molecular Modeling Toolkit or the Software Development Toolkit. The template consists of four main parts; an introduction, an overview, special instructions regarding modifications or enhancements to the feature, and a detailed description of all of its subroutines and functions.

Documentation of a code or module can be thought of as a dialogue between the developer and future developers or users of the code, in which the original developer must guess the questions the other person will ask. Fortunately, this is not all that difficult. If the documentation is written to answer these questions, then it is quite likely that the next person to pick up the code will readily understand what it is supposed to do, how it works, and may even be able to figure out how to fix it when it is broken. The template described below, therefore, is presented in terms of the questions each section of the documentation should be written to answer.

Modeling Tools Documentation Template

- Introduction
 - What is this thing? (List its name and a brief—but *non-recursive* – description of what it does.)
 - Where did it come from? (List source references, if any, with full bibliography (if available); note any significant geneological information, if relevant.)
- Overview
 - What does it do? (Give a detailed, nuts and bolts description of what the code does, and how it does it. Describe how it interacts with the rest of NWChem. If there are any special requirements or limitations on the use of the feature(s) of this coding, this is the place to mention them. If there is an order in which certain subroutines should be called (i.e. initialize, modify and delete), this should be listed here also.
- Modifications
 - Can this code be changed? (Describe any special considerations for modifying the code, especially if there are hidden repercussions of choices made at this level in the code. Note any compatibility problems with other modules in the code.)
- Annotated List of All Subroutines and Functions

- Instead of the list, there may instead be a pointer to a more complete description of the subroutines and functions in another document or an appendix. This list may also be automatically generated from the in-source documentation of routines (listed below).
- How many subroutines/functions are there in this element? (Note the number; if it is large, try to organize them into some sort of logical groupings, for ease of reference and to clarify the structure of the code. If there is no obvious structure, present them in alphabetical order.)
- What are the subroutines/functions in this element? (For each subroutine or function, include the information specified in Section 11.1.3.)

11.1.3 Content for In-Source Documentation of Routines

This is the base level of documentation, and is the one level that is almost guaranteed to actually be read by a new developer. Therefore, it is very important that the documentation at this level be as clear and complete as possible. At the very minimum, the in-source documentation should consist of lines containing the following information:

Required:

- a verbatim reproduction of the function or subroutine statement
- a list of all arguments, identifying for each argument
 - its data type
 - its status as input, output, or input/output data. If the argument is a handle to an object, the status of the handle as well as the object should be noted by `handle_status(object_status)`. For example, if the handle and the data in an object will be created in a subroutine, use the notation `output (output)`.
 - a concise (but informative) definition
- a terse description of what the routine does
- a description of the return value(s) of the function itself (if any)
- a description of the calling protocol for the subroutine; that is, whether
 - 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 "collectiveness" 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 transform 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(maxle, maxg, mscratch_1e, mscratch_2e)

integer maxle      ! [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(maxle, mscratch_1e)

integer maxle      ! [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_2e3c

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:

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

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

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

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):

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

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):

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

Syntax:

```

subroutine int_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):

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

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, ($H1$).

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

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_leh1(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:

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

Syntax:

```

subroutine int_1eall(i_basis,ish,j_basis,jsh,lscr,scr,lstsv,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 lstsv   ! [input] length of one electron buffers
double precision T(lstsv) ! [output] kinetic integral buffer
double precision V(lstsv) ! [output] potential integral buffer
double precision S(lstsv) ! [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 lstsv   ! [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, llg, Gle)

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 llg     ! [input] length of integral buffer array
double precision Gle(llg) ! [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_1le3ov

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_1le3ov(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_1leall

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_lleall(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_pgenle

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

Syntax:

```

subroutine int_pgenle(msg, i_basis, ish, j_basis, jsh,
&      ilab, jlab, lle, Genle, 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 lle       ! [input] number of integrals and labels
integer ilab(lle) ! [input] i basis functions labels
integer jlab(lle) ! [input] j basis functions labels
double precision Genle(lle) ! [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.

$$H1 = T + V$$

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

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

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

Syntax:

```

subroutine int_llehl(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 le 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] le H1 integrals

```

A.3.13 int_lleke

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_lleke(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 le 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_lleov

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

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

Syntax:

```

subroutine int_lleov(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_llepe

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.

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

Syntax:

```

subroutine int_llepe(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, lle, Genle, 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 lle     ! [input] length of buffers for integrals
integer ilab(lle) ! [output] i bas func labels array
integer jlab(lle) ! [output] j bas func labels array
double precision Genle(lle) ! [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:

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

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:

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

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:

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

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).

<i>Return Values:</i>	.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. These 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.

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

<i>Return Values:</i>	.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 $\langle shell|pole|shell \rangle$. Integrals are returned in shell blocks of $\langle ish|L|jsh \rangle$ $L=0$ 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:

$\langle x x \rangle$	$\langle x y \rangle$	$\langle x z \rangle$	$\langle y x \rangle$	$\langle y y \rangle$...	$\langle z z \rangle$	(first nine)
$\langle x x x \rangle$	$\langle x x y \rangle$	$\langle x x z \rangle$	$\langle x y x \rangle$	$\langle x y y \rangle$...	$\langle x z z \rangle$	(second nine)
$\langle y x x \rangle$	$\langle y x y \rangle$	$\langle y x z \rangle$	$\langle y y x \rangle$	$\langle y y y \rangle$...	$\langle y z z \rangle$	(third nine)
$\langle z x x \rangle$	$\langle z x y \rangle$	$\langle z x z \rangle$	$\langle z y x \rangle$	$\langle z y y \rangle$...	$\langle z z z \rangle$	(fourth nine)

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 $lmax$ ($0 \rightarrow lmax$):

The general form is $\langle pole|shell \rangle$

Integrals are returned in $\langle pole|shell \rangle$ blocks one block for each L value $0 \rightarrow lmax$.

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 > ... < 0|zz >   first six
< x|xx > < x|xy > < x|xz > < x|yy > ... < x|zz >   second six
< y|xx > < y|xy > < y|xz > < y|yy > ... < y|zz >   third six
< z|xx > < z|xy > < z|xz > < z|yy > ... < 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, maxle, memsize)

integer brain ! [input] bra basis set handle
integer ketin ! [input] ket basis set handle
integer maxle ! [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, maxle, memsize)

integer brain ! [input] bra basis set handle
integer ketin ! [input] ket basis set handle
integer maxle ! [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, maxle, memsize)

integer brain ! [input] bra basis set handle
integer ketin ! [input] ket basis set handle
integer maxle ! [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 determine 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(bra, ket, maxle, memsize, rel_typ)

integer bra ! [input] bra basis set handle
integer ket ! [input] ket basis set handle
integer rel_typ ! [input] type of integrals to be computed
integer maxle ! [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, maxle, memsize)

integer brain    ! [input] bra basis set handle
integer ketin    ! [input] ket basis set handle
integer maxle    ! [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, maxle, memsize)

integer brain    ! [input] bra basis set handle
integer ketin    ! [input] ket basis set handle
integer maxle    ! [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_hflsp(
&      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 $\langle i|Operator|j \rangle$

```

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_hflsp_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_hflsp_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 $\langle i|Operator|j \rangle$

```

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

	computed	transformed	
	order	order	
	1 (x s)	(x s)	
	2 (y s)	(x x)	2 → 5
	3 (z s)	(x y)	3 → 9
	4 (x x)	(x z)	4 → 2
in place as follows:	5 (x y)	(y s)	5 → 3
	6 (x z)	(y x)	6 → 4
	7 (y x)	(y y)	7 → 6
	8 (y y)	(y z)	8 → 7
	9 (y z)	(z s)	9 → 8
	10 (z x)	(z x)	
	11 (z y)	(z y)	
	12 (z z)	(z z)	

Syntax:

```

subroutine int_lpsp(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

	computed	transformed	
	order	order	
	1 (xx s)	(xx s)	
	2 (xy s)	(xx x)	2 → 5
	3 (xz s)	(xx y)	3 → 9
	4 (yy s)	(xx z)	4 → 13
	5 (yz s)	(xy s)	5 → 17
	6 (zz s)	(xy x)	6 → 21
	7 (xx x)	(xy y)	7 → 2
	8 (xx y)	(xy z)	8 → 3
	9 (xx z)	(xz s)	9 → 4
	10 (xy x)	(xz x)	10 → 6
in place as follows:	11 (xy y)	(xz y)	11 → 7
	12 (xy z)	(xz z)	12 → 8
	13 (xz x)	(yy s)	13 → 10
	14 (xz y)	(yy x)	14 → 11
	15 (xz z)	(yy y)	15 → 12
	16 (yy x)	(yy z)	16 → 14
	17 (yy y)	(yz s)	17 → 15
	18 (yy z)	(yz x)	18 → 16
	19 (yz x)	(yz y)	19 → 18
	20 (yz y)	(yz z)	20 → 19
	21 (yz z)	(zz s)	21 → 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 transfor-

	computed	transformed	
	order	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 → 9
mation is done in place as follows:	7 (z s)	(x y)	7 → 13
	8 (x x)	(x z)	8 → 6
	9 (x y)	(y s)	9 → 7
	10 (x z)	(y x)	10 → 8
	11 (y x)	(y y)	11 → 10
	12 (y y)	(y z)	12 → 11
	13 (y z)	(z s)	13 → 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_spla(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 equivalenced 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 \leq `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 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 `int2e_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.