The PSI3 Programmer's Manual

T. Daniel Crawford,[a] C. David Sherrill,[b] Edward F. Valeev,[b],
Justin T. Fermann,[c] and C. Brian Kellogg[c]

[a] *Department of Chemistry, Virginia Tech, Blacksburg, Virginia 24061*

[b] *Center for Computational Molecular Science and Technology, Georgia Institute of Technology, Atlanta, Georgia 30332-0400*

[c] *Center for Computational Quantum Chemistry, University of Georgia, Athens, Georgia 30602-2525*

PSI3 Version: 3.2.0 (stable)
Created on: December 31, 2003

Contents

1 Introduction

The PSI suite of *ab initio* quantum chemistry programs is the result of an ongoing attempt by a cadre of graduate students, postdoctoral associates, and professors to produce efficient and fast computer code. Some of the earliest contributions to what is now referred to as "PSI" include a direct configuration interaction (CI) program (Robert Lucchese, 1976, now at Texas A&M), the well-known graphical unitary group CI program (Bernie Brooks, 1977-78, now at N.I.H.), and the original integrals code (Russ Pitzer, 1978, now at Ohio State). From 1978-1987, the package was know as the `BERKELEY` suite, and after the Schaefer group moved to the Center for Computational Quantum Chemistry at the University of Georgia, the package was renamed `PSI`. Thanks primarily to the efforts of Curt Janssen (Sandia Labs, Livermore) and Ed Seidl (LLNL), the package was ported to UNIX systems, and substantially improved with new input formats and a C-based I/O system.

Beginning in 1999, an extensive effort was begun to develop `PSI3` — a `PSI` suite with a completely new face. All of the code is now in C and C++, including new integral/derivative integral, coupled cluster, and CI codes. In addition, new I/O libraries have been added, as well as an improved checkpoint file structure and greater automation of typical tasks such as geometry optimization and frequency analysis. The package has the capability to determine wavefunctions, energies, analytic gradients, and various molecular properties based on a variety of theories, including spin-restricted, spin-unrestricted, and restricted open-shell Hartree-Fock (RHF, UHF, and ROHF); configuration interaction (CI) (including a variety of multireference CI's and full CI); coupled-cluster (CC) including CC with variationaly optimized orbitals; second-order Møller-Plesset perturbation theory (MPPT) including explicitly correlated second-order Møller-Plesset energy (MP2-R12); and complete-active-space self-consistent field (CASSCF) theory.

The purpose of this manual is to provide a reasonably detailed overview of the source code and programming philosophy of `PSI3`, such that programmers interested in contributing to the code will have an easier task. Section 2 gives a succint explanation of the steps required to obtain the source code from the main repository at Virginia Tech. (Installation instructions are given separately in the installation manual or in $PSI3/INSTALL.) Section 3 discusses the essential elements of a C-language `PSI3` program, with emphasis on the input parsing and I/O functions. Section 4 provides documentation of a number of other important libraries, including the library of functions for reading from the checkpoint file, `libchkpt.a`, the Quantum Trio miscellaneous function library, `libqt.a`, the `libiwl.a` for reading and writing one- and two-electron integrals in the "integrals with labels" format. Section 5 offers advice on appropriate programming style for `PSI3` code, and section 6 describes the structure of the package's `Makefiles`. Section 6.3 gives a brief overview of the necessary steps to adding a new module to `PSI3`, section 7 gives some suggestions on debugging it, and section 8 explains conventions for documenting it. The appendices provide important reference material, including the currently accepted `PSI3` citation and format information for some of the most important text files used by `PSI3` modules.

2 The PSI3 Source Code

The concurrent versions system (CVS) (`www.cvshome.org`) provides a convenient means by which programmers may obtain the latest (or any previous) version of the PSI3 source from the main repository or a branch version, add new code to the source tree or modify existing PSI3 modules, and then make changes and additions available to other programmers by checking the modifications back into the main repository. CVS also provides a "safety net" in that any erroneous modifications to the code may be easily removed once they have been identified. This section describes how to use CVS to access and modify the PSI3 source code. (Note that compilation and installation instructions are given in a separate document.)

2.1 Secure Access to the PSI3 Repository

The main repository for the PSI3 Source code is currently maintained by the Crawford group at Virginia Tech. To check out the code, one must first obtain a CVS account by emailing `crawdad@vt.edu`. After you have a login-id and password, set the `CVSROOT` environmental variable for your shell to point to the main repository:

For csh/tcsh:

```
setenv CVSROOT :pserver:login@sirius.chem.vt.edu:/home/users/psi3/master
```

For sh/bash:

```
CVSROOT=:pserver:login@sirius.chem.vt.edu:/home/users/psi3/master
export CVSROOT
```

where `login` is your login name. Next, provide a password to the CVS server by typing `cvs login`. (Subsequent requests to the repository will not require you to re-login.)

2.2 PSI3 CVS Policies: Which Branch Should I Use?

The PSI3 repository is comprised of a main trunk and several release branches. The branch you should use depends on the sort of work you plan for the codes:

1. For any piece of code already in the most recent release, bug fixes (defined as anything that doesn't add functionality — including documentation updates) should be made *only* on the most recent stable release branch.

2. The main trunk is reserved for development of new functionality. This allows us to keep new, possibly unstable code away from public access until the code is ready.

3. Code that you do not want to put into next major release of PSI3 should be put onto a separate branch off the main trunk. You will be solely responsible for maintenance of the new branch, so you should read the CVS manual before attempting this.

Fig. 1 provides a schematic of the CVS revision-control structure and branch labeling. Two release branches are shown, the current stable branch, named `psi-3-2`, and a planned future release, to be named `psi-3-3`. The tags on the branches indicate release points, where bugs have been fixed and the code has been or will be exported for public distribution. As soon as a release branch is created, a tag will be generated so that updates may be made to that version of the code. We have adopted the convention that the first tag on a new release branch will be given the suffix, `-rc-1`, for "release candidate 1" (i.e., a numbered beta release). Subsequent tags for new released on the same branch will be updated with `-rc-2`, `-rc-3`, etc. Once the code is appropriately stable to be beyond beta status, release tags will be given the suffix, `-0`, `-1`, etc., indicating patch levels. The dotted lines in the figure indicate merge points: before each public release, changes made to the code on the stable release branch will be merged into the main trunk.

Figure 1: `PSI3` CVS branch structure with examples of branch- and release-tag labelling.

A frequently encountered problem is what to do about bug fixes that are necessary for uninterrupted code development of the code on the main trunk. As Rule 1 of the above policy states, all bug fixes of the code already in the recent stable release must go on the corresponding branch, not on the main trunk. The next step depends on the severity of the bug:

1. If the bug fix is critical and potentially affects every developer of the code on the main trunk, then `PSI3` administrators should be notified of the fix. If deemed necessary, appropriate steps to create a new patch release will be made. Once the next patch release is created then the bug fixes will be merged onto the main trunk. If the bug fix doesn't warrant an immediate new patch release, then you can incorporate the bug fix into your local copy of the main trunk code manually or using CVS merge features (see the CVS documentation for the `-j` option). This will allow you to continue development until next patch release is created and the bug fix is incorporated into the main trunk code in the repository. However you should *never* merge such changes into the main trunk yourself.

2. If the bug fix is not critical (e.g. a documentation update/fix), then you should wait until next patch release when it will be merged into the main trunk automatically.

The following are some of the most commonly used CVS commands for checking out and updating working copies of the `PSI3` source code are.

• To checkout a working copy of the head of the main trunk:

```
cvs co psi3
```

• To check out a working copy of the head of a specific release branch, e.g., the branch labelled `psi-3-2`:

```
cvs co -r psi-3-2 psi3
```

Note that this sets a "sticky" branch tag, i.e. subsequent `cvs update` commands will provide updates only on the chosen branch. Note also that after you have checked out a fresh working copy of the code you must run the `autoconf` command to generate a `configure` script for building the code. (See the installation manual for configuration, compilation, and testing instructions.)

The working copy of your code will be placed in the directory `psi3`, regardless of your choice of branch. In this manual, we will refer to this directory from now on as `$PSI3`. Subsequent CVS commands are usually run within this top-level directory.

• To update your current working copy to include the latest revisions:

```
cvs update -dP
```

Notes: (a) This will update only the revisions on your current branch; (b) the `-d` flag tells CVS to grab any new directories that have appeared in the repository on this branch; (c) the `-P` option tells CVS to remove (prune) any empty directories from your working copy.

• To convert your working copy to the head of a specific branch:

```
cvs update -r psi-3-2
```

• To convert your working copy to the head of the main trunk:

```
cvs update -A
```

Warning: The previous two `update` commands should be used WITH CAUTION. If you have made changes to your working copy but have not yet checked them into the repository, the update to another branch will often (always?) delete your changes!

• To find out what branch your working copy is on, run this in your top-level `PSI3` source directory:

```
cvs status -v configure.in
```

This will list all tags and branch tags available for the file (which should a correct list for the whole `PSI3` repository) and it will indicate whether your working copy is on a particular branch or not.

Some words of advice:

1. Be careful! If you aren't sure if you should run the CVS command you're about to try, perhaps you should first make a backup of of your working copy of the code.

2. Read the CVS manual. Seriously.

   `http://www.loria.fr/~molli/cvs/doc/cvs_toc.html`

3. If you're about to start some significant development or bug-fixes, first update your working copy to the latest version on your branch. In addition, if you do development over a long period of time (say weeks to months) on a specific module or modules, be sure to run a `cvs update` occasionally. In can be *very* frustrating to try to check in lots of changes, only to find out that the `PSI3` has changed dramatically since your last update.

2.3 Checking in altered `PSI3` binaries or libraries

If you have changes to Psi binaries or libraries which already exist, one of two series of steps is necessary to check these changes in to the main repository. The first series may be followed if all changes have been made only to files which already exist in the current version. The second series should be followed if new files must be added to the code in the repository.

- No new files need to be added to the repository. We will use `libciomr` as an example.
  1. `cd $PSI3/src/lib/libciomr`
  2. `cvs ci`
  3. Edit the comment file that CVS provides.
- New files must be added to the repository. Again, we use `libciomr` as an example. Suppose the new file is named `great_code.c` .
  1. `cd $PSI3/src/lib/libciomr`
  2. `cvs add great_code.c`
  3. `cvs ci`
  4. Edit the comment file that CVS provides.

The `cvs ci` command in both of these sequences will examine all of the code in the current `libciomr` directory against the current version of the code in the main repository. Any files which have been altered (and for which no conflicts with newer versions exist!) will be identified and checked in to the main repository (as well as the new file in the second situation). CVS will open a comment file for editing; you should enter a description of the changes you have made to the code here, as well as in the code itself. Exit the editing of the comment file, and CVS will check the new code into the main repository.

2.4 Adding entirely new code to the main `PSI3` repository

If the programmer is adding a new executable module or library to the `PSI3` repository, a number of important conventions should be followed:

1. Since such changes almost always involve additional functionality, new modules or libraries should be added only on the main CVS trunk. See section 2.2 for additional information.

2. The directory containing the new code should be given a name which matches the name of the installed code (e.g. if the code will be installed as `newcode`, the directory containing the code should be named `newcode`). New executable modules must be placed in `$PSI3/src/bin` and libraries in `$PSI3/src/lib` of the user's working copy.

3. The Makefile should be converted to an input file for the configure script (`Makefile.in` — see any of the current `PSI3` binaries for an example) and should follow the conventions set up in all of the current `PSI3` `Makefiles`. This includes use of `MakeVars` and `MakeRules`.

4. New binaries should be added to the list contained in `$PSI3/src/bin/Makefile.in` so that they will be compiled automatically when a full compilation of the `PSI3` distribution occurs. This step is included in the sequence below.

5. A documentation page should be included with the new code (see section 8 for more information). As a general rule, if the code is not ready to have a documentation page, it is not ready to be installed in `PSI3`.

6. The `configure.in` file must be altered so that users may check out copies of the new code and so that the `configure` script will know to create the Makefile for the new code. These steps are included in the sequence below.

Assume the new code is an executable module and is named `great_code`. The directory containing the new code must contain only those files which are to be checked in to the repository! Then the following steps will check in a new piece of code to the main repository:

1. `cd $PSI3/src/bin`
2. `cvs add great_code`
3. Answer "y" when CVS asks if you wish to create the new directory in the repository.
4. `cd great_code`
5. `cvs add *`
6. `cvs ci`
7. Edit the comments file that CVS provides.
8. `cd $PSI3`
9. Edit `configure.in` and add `great_code` to the list.
10. `cvs ci`
11. Edit the comments file that CVS provides.
12. `autoconf`

13. `cd $PSI3/src/bin`
14. Edit `Makefile.in` and add `great_code` to the list.
15. `cvs ci`
16. Edit the comments file that CVS provides.

At this point, all of the code has been properly checked in. However, you must test to make sure that the code can be checked out by other programmers, and that it will compile correctly. The following steps will store your personal version of the code, check out the new code, and test-compile it:

1. `cd $PSI3/src/bin`
2. `mv great_code great_code.bak`
3. `cd $PSI3/..`
4. `cvs co $PSI3/src/bin/great_code`
5. `cd $objdir`
6. `$PSI3/configure --prefix=$prefix`
7. `cd src/bin/great_code`
8. `make install`

(Note that $prefix and $objdir to the installation and compilation directories defined in the `PSI3` installation instructions.) Your original version of the code remains under `great_code.bak`, but should be no longer necessary if the above steps work. Note that it is necessary to re-run `configure` explicitly, instead of just running `config.status`, because the latter contains no information about the new code.

2.5 Updating checked out code

If the code in the main repository has been altered, other users' working copies will of course not automatically be updated. In general, it is only necessary to execute the following steps in order to completely update your working copy of the code:

1. `cd $PSI3`
2. `cvs update -dP`

This will examine each entry in your working copy and compare it to the most recent version in the main repository. When the file in the main repository is more recent, your version of the code will be updated. If you have made changes to your version, but the version in the main repository has not changed, the altered code will be identified to you with an "M". If you have made changes to your version of the code, and one or more newer versions have been updated in the main repository, CVS will examine the two versions and attempt to merge them – this process usually has conflicts however, and is sometimes unsuccessful. You will be notified of any conflicts that arise (labelled with a "C") and you must resolve them manually.

The `-dP` flags to `cvs` above will also ensure that any directories added to or removed from the main repository are also added or deleted in your working copy. Keep in mind, however, that the appearance of new directories will require you to re-run the configure script to compile the new codes. For example,

1. `cd $PSI3`
2. `cvs update -dP`
3. `autoconf`
4. `cd $objdir`
5. `$PSI3/configure --prefix=$prefix`
6. `cd src/bin/great_code`
7. `make install`

This will check out any new code that has been added to the repository since the last time the `cvs update` command was issued, correctly update the `configure` script (assuming the programmer who added the code followed the steps from section 2.4 and edited the `configure.in` file), re-configure for the new code (note that you must run `configure` and not `config.status` here), and then compile the code.

2.6 Removing code from the repository

If alterations of libraries or binaries under Psi involves the deletion of source code files from the code, these must be explicitly removed through CVS.

The following steps will remove a source code file named `bad_code.F` from a binary module named `great_code`:

1. `cd $PSI3/src/bin/great_code`
2. `rm bad_code.F`
3. `cvs remove bad_code.F`

4. `cvs ci`

5. Edit the comments file that CVS provides.

2.7 Checking out older versions of the code

It is sometimes necessary to check out older versions of a piece of code. Assume we wish to check out an old version of `detci`. If this is the case, the following steps will do this:

1. `cd $PSI3/src/bin/detci`

2. `cvs update -D"2 months ago"`

This will check the main repository and provide you with the code as it stood exactly 2 months ago. CVS is quite impressive in this respect: It will accept all sorts of input to the -D option. You could even use `-D"a fortnight ago"` and CVS would get the correct version. (But it doesn't get `"two fortnights ago"` right.) You can always get the recent version back by simply using `-D"now"` or `-A`. Note that subsequent updates of the current code will use the same date you give with the `-D` option. (See the CVS documentation about so-called "sticky tags".)

2.8 Examining the revision history

It can be very useful to use cvs to see what recent changes have been made to the code. Anytime one checks in a new version of a file, CVS opens an editor and requests a comment describing what changes have been made. These comments go into a log file which may be easily accessed through CVS. To see what changes have been made recently to the file `detci.cc`, one would go into the `detci` source directory and type

```
cvs log detci.cc
```

Then some summary information (such as the tags applied to the code) is printed, and finally a list will be produced of all the versions of the code, along with the checkin comments, in reverse chronological order. For example,

```
sherrill@vergil(detci)% cvs log detci.cc

RCS file: /home/users/psi3/master/psi3/src/bin/detci/detci.cc,v
Working file: detci.cc
head: 1.26
branch:
locks: strict
access list:
symbolic names:
```

```
        psi-3-2-f77-delete: 1.22
        psi-3-2-branch: 1.20.0.2
        psi-3-2-release: 1.20
        crazy-russian-tag: 1.18
        gbye-file30-branch-tag: 1.16
        gbye-file30-merge: 1.16.2.2
        gbye-file30: 1.16.0.2
        release-3-1-branch: 1.10.0.2
        rel-3-0-2dir: 1.7.0.2
        PSI_3_0_0: 1.1.1.1
        CCQC_UGA: 1.1.1
keyword substitution: kv
total revisions: 30;    selected revisions: 30
description:
description:
----------------------------
revision 1.26
date: 2002/12/24 18:54:51;  author: sherrill;  state: Exp;  lines: +1 -0
Correct a mistake in input parsing so that the argument counter is
incremented *twice* for -c [num].
----------------------------
revision 1.25
date: 2002/12/23 22:39:10;  author: sherrill;  state: Exp;  lines: +2 -2
Convert -we to -e and -quiet to --quiet
...
```

Checking the log files is a very useful way to see what recent changes might be causing new problems with the code.

2.9 The structure of the PSI3 Source Tree

Your working copy of the PSI3 source code includes a number of important subdirectories:

- `$PSI3/lib` – Source files for OS-independent "library" data. This includes the main basis set data file (`pbasis.dat`) and the PSI3 program execution control file (`psi.dat`), among others. These files are installed in `$prefix/share`.
- `$PSI3/include` – Source files for OS-independent header files, including `physconst.h` (whose contents should be obvious from its name), `psifiles.h`, and `ccfiles.h`, among others. These files are installed in $prefix/include.
- `$PSI3/src/util` – Source code for the utility program `tocprint`. (Note that the `tmpl` module is no longer used and will eventually disappear.)
- `$PSI3/src/lib` – Source code for the libraries, including `libpsio`, `libipv1`, `libchkpt`, etc. The include files from the library source are used directly during the compilation of

PSI to avoid problems associated with incomplete installations. Some include files are architecture-dependent and go in an include subdirectory of the compilation (object) directory.

- `$PSI3/src/lib` – Source code for the executable modules.

After compilation and installation, the `$prefix` directory contains the executable codes and other necessary files. **NB:** The files in this area should never be directly modified; rather, the working copy should be modified and the `PSI3` `Makefile` hierarchy should handle installation of any changes. The structure of the installation area is:

- `$prefix/bin` – The main executable directory. This directory must be in your path in order for the driver program, `psi3`, to find the modules.
- `$prefix/lib` – The `PSI3` code libraries. (NB: The description of `PSI3` `Makefiles` later in this manual will explain how to use the libraries.)
- `$prefix/include` – Header files. These are not actually used during the compilation of PSI but are useful for inclusion by external programs because they are all in the same directory.
- `$prefix/share` – OS-independent data files, including basis set information. (Do not edit this file directly; any changes you make can be overwritten by subsequent `make` commands.)
- `$prefix/doc` – `PSI3` documentation, including installation, programmer, and user manuals.

3 Fundamental `PSI3` Functions

Each `PSI3` module (e.g. `cscf`) must perform two specific tasks, regardless of the individual module's specific purpose(s): (1) obtaining user input options and (2) writing to and reading from binary files (e.g. the checkpoint file). `PSI3` programs written in the C programming language make use of two libraries which provide all the tools necessary to carry out these functions:

- `libipv1.a` — the input parser
- `libpsio.a` — the I/O interface

In addition, the libraries `libciomr.a` and `libqt.a` provide important functions for memory allocation, mathematics, and code timing. In the next section, we will discuss the basic components of a `PSI3` C-language program, followed by detailed descriptions of the input parsing and I/O libraries.

```
#include <stdio.h>
#include <libipv1/ip_lib.h>
#include <libpsio/psio.h>
#include <libciomr/libciomr.h>

FILE *infile, *outfile;
char *psi_file_prefix;

int main(int argc, char *argv[])
{
  extern char *gprgid(void);

  psi_start(argc-1, argv+1, 0);
  ip_cwk_add(gprgid());
  psio_init();

  /* to start timing, tstart(outfile); */

  /* Insert code here */

  /* to end timing, tstop(outfile); */

  psio_done();
  psi_stop();
}

char *gprgid(void)
{
   char *prgid = ":CODE_NAME";
   return(prgid);
}
```

Figure 2: The essential elements of a PSI3 C-language program.

3.1 The Structure of a PSI3 C Program

To function as part of the PSI package, a program must incorporate certain required elements. This section will discuss the header files, global variables, and functions required to integrate a C program into PSI3. Figure 2 presents a minimal PSI3 program, whose elements are described below.

The required include files are libipv1/ip_lib.h, libciomr/libciomr.h, libpsio/psio.h, and of course stdio.h. The first of these is for the Input Parser Library, Version 1 (libipv1.a), which is described in section 3.2. The second file contains function prototypes for the C Math Routines and old-style I/O library, libciomr.a. The third file analogously provides clean

interface to functions of the new C I/O system described in section 3.3. The PSI libraries require that `infile`, `outfile`, and `psi_file_prefix` be global variables.

The integer function `main()` must be able to handle command-line arguments required by the `PSI3` libraries. In particular, all `PSI3` modules must be able to pass to the function `psi_start()` arguments for the user's input and output filenames, as well as a global file prefix to be used for naming standard binary and text data files. (NB: the default names for user input and output are `input.dat` and `output.dat`, respectively, though any name may be used.) The current standard for command-line arguments is for all module-specific arguments (*e.g.*, `--quiet`, used in `detci`) *before* the input, output, and prefix values. The `psi_start()` function expect to find *only* these last three arguments at most, so the programmer should pass as `argv[]` the pointer to the first non-module-specific argument. The above example is appropriate for a `PSI3` module that requires no command-line arguments apart from the input/output/prefix globals. See the `PSI3` modules `input` and `detci` for more sophisticated examples. The final argument to `psi_start()` is an integer whose value indicates whether the output file should be overwitten (1) or appended (0). Most `PSI3` modules should choose to append.

The `psi_start()` function initializes the user's input and output files and sets the global variables `infile`, `outfile`, and `psi_file_prefix`, based on (in order of priority) the above command-line arguments or the environmental variables `PSI_INPUT`, `PSI_OUTPUT`, and `PSI_PREFIX`. The value of the global file prefix can also be specified in the user's input file. The `psi_start()` function will also initialize the input parser and sets up a default keyword tree (described in detail in section 3.2). This step is required even if the program will not do any input parsing, because some of the functionality of the input parser is assumed by `libciomr.a` and `libpsio.a`. For instance, opening a binary file via `psio_open()` (see section 3.3) requires parsing the `files` section of the user's input so that a unit number (e.g. 52) can be translated into a filename.

The `psi_stop()` function shuts down the input parser and closes the user's input and output files.

Timing information (when the program starts and stops, and how much user, system, and wall-clock time it requires) can be printed to the output file by adding calls to `tstart()` and `tstop()` (from `libciomr.a`).

The sole purpose of the simple function `gprgid()` is to provide the input parser a means to determine the name of the current program. This allows the input parser to add the name of the program to the input parsing keyword tree. This function is used by `libpsio.a`, though the functionality it provides is rarely used.

NB: The library `libciomr.a` contains older I/O functions that have been superceded by functions in `libpsio.a`. However, you are encouraged to use the many non-I/O functions in `libciomr.a`.

3.2 The Input Parser

The input parsing library is built for the purpose of reading in the contents of an input file with the syntax of `input.dat` and storing the contents specific to certain keywords supplied. To perform such a task `libipv1.a` has three parts: (1) the parser; (2) the lexical scanner; (3) keyword storage and retrieval.

The format of `input.dat` follows certain rules which should probably referred to as the PSI input grammar. There is a description of most of those rules in `PSI3` User's Manual. A complete definition of the PSI input grammar is encoded in `parse.y` (see below). To read a grammar we need a parser – the first component of `libipv1.a`. Then the identified lexical elements of `input.dat` (keywords and keyword values) need to be scanned for presence of "forbidden" characters (e.g. a space may not be a part of a string unless the string is placed between parentheses). This task is performed by the lexical scanner — the second component of `libipv1.a`. Finally, scanned-in pairs of keyword-value(s) are stored in a hierarchical data structure (a tree). When a particular option is needed, the set of stored keywords and values is searched for the one queried and the value returned. In this way, options of varying type can be assigned, i.e. rather than having a line of integers, each corresponding to a program variable, mnemonic character string variables can be parsed and interpreted into program variables. It's also easier to implement default options, allowing a more spartan input deck. The set of input-parsing routines in `libipv1.a` is really not complicated to use, but the manner in which data is stored is somewhat painful to grasp at first.

The following is a list of the names of the individual source files in `libipv1` and a summary of their contents. After that is a list of the syntax of specific functions and their use. Last is a simple illustration of the use of this library, taken mostly from `cscf`.

3.2.1 Source Files

- Header files
  - `ip_error.h` Defines for error return values.
  - `ip_global.h` cpp macros to make Curt happy.
  - `ip_lib.h` #include's everything.
  - `ip_types.h` Various structures and unions specific to `libipv1`.
- Other Source
  - `parse.y` Yacc source encoding the PSI input grammar. Read by `yacc` (or `bison`) – a parser generator program.
  - `scan.l` Lex source describing lexical elements allowed in `input.dat`. Read by `lex` (or `flex`) – a lexer generator program.
  - `*.gbl, *.lcl` cpp macros to mimic variable argument lists.
- C source

- `ip_alloc.c` Allocates keyword tree elements.
- `ip_cwk.c` Routines to manipulate the current working keyword tree.
- `ip_data.c` Routines to handle reading of arrays and scaler keyword assignments in input.
- `ip_error.c` Error reporting functions.
- `ip_karray.c` Other things to deal with keyword arrays.
- `ip_print.c` Routines to print sections of the keyword tree.
- `ip_read.c` All the file manipulation routines. Reading of `input.dat` and building the keyword tree from which information is later plucked.

3.2.2 Syntax

`ip_cwk.c`

```
void ip_cwk_clear();
```

Clears current working keyword. Used when initializing input or switching from one section to another (:DEFAULT and :CSCF to :INTCO, for instance).

```
void ip_cwk_add(char *kwd);
```

Adds `kwd` to the list of current working keywords. Allows parsing of variables under that keyword out of the input file (files) which has (have) been read or will be read in the future using `ip_append`. The keyword `kwd` can only be removed from the list of current working keywords by purging the entire list using `ip_cwk_clear`.

`ip_data.c`

```
int ip_count(char *kwd, int *count, int n);
```

Counts the elements in the n'th element of the array `kwd`.

```
int ip_boolean(char *kwd, int *bool, int n);
```

Parses n'th element of `kwd` as boolean (true, 1, yes; false, 0, no) into 1 or 0 returned in `bool`.

```
int ip_exist(char *kwd, int n);
```

Returns 1 if n'th element of `kwd` exists. Unfortunately, n must be 0.

```
int ip_data(char *kwd, char *conv, void *value, int n [, int o1, ..., int on]);
```

Looks for keyword `kwd`, finds the value associated with it, converts it according to the format specification given in `conv`, and stores the result in `value`. Note that `value` is a `void *` so this routine can handle any data type, but it is the programmer's responsibility to ensure that the pointer passed to this routine is of the appropriate pointer type for the data. The value found by the input parser depends on the value of `n` and any optional additional arguments. `n` is the number of additional arguments. If `n` is 0, then there are no additional arguments, and the keyword has only one value associated with it. If the keyword has an array associated with it, then `n` is 1 and the one additional argument is which element of the

array to pick. If kwd specifies an array of arrays, then n is 2, the first additional argument is the number of the first array, and the second argument is the number of the element within that array, etc. Deep in here, the code calls a sscanf(read, conv, value);, so that's the real meaning of variables.

```
int ip_string(char *kwd, char **value, int n, [int o1, ..., int on]);
```
Parses the string associated with kwd stores it in value. The role of n and optional arguments is the same as that described above for ip_data().

```
int ip_value(char *kwd, ip_value_t **ip_val, int n);
```
Grabs the section of keyword tree at kwd and stores it in ip_val for the programmer's use - this is usually not used, since you need to understand the structure of ip_value_t.

```
int ip_int_array(char *kwd, int *arr, int n);
```
Reads n integers into array arr.

ip_read.c

```
void ip_set_uppercase(int uc);
```
Sets parsing to case sensitive if uc==0, I think.

```
void ip_initialize(FILE *in, FILE *out);
```
Calls yyparse(); followed by ip_cwk_clear(); followed by ip_internal_values();. This routine reads the entire input deck and stores it into the keyword tree for access later.

```
void ip_append(FILE *in, FILE *out);
```
Same thing as ip_initialize();, except this doesn't clear the cwk first. Used for parsing another input file, such as intco.dat.

```
void ip_done();
```
Frees up the keyword tree.

ip_read.c

```
void ip_print_tree(FILE *out, ip_keyword_tree_t *tree);
```
Prints out tree to out. If tree is set to NULL, then the current working keyword tree will be printed out. This function is useful for debugging problems with parsing.

3.2.3 Sample Use from cscf

These are two slightly simplified pieces of actual code.

From cscf.c:

```
#include <libipv1/ip_lib.h>

   ffile(&infile,"input.dat",2);      /* input and output files. */
   ffile(&outfile,"output.dat",1);    /* call them whatever you want. */
```

```
ip_set_uppercase(1);              /* case sensitivity selection */
ip_initialize(infile,outfile);    /* reads input.dat and stores it all */

ip_cwk_add(":DEFAULT");           /* adds default section */
ip_cwk_add(":SCF");               /* adds scf section */

ip_string("OUTPUT",&output,0);    /* bet you didn't know you could */
if(!strcmp(output,"TERMINAL")) {  /* have cscf write to stdout!    */
  outfile = stdout;
  }
else if(!strcmp(output,"WRITE")) {
  fclose(outfile);
  ffile(&outfile,"output.dat",0);
  }
```

From `scf_input.c`:

```
errcod = ip_string("LABEL",&alabel,0);
if(errcod == IPE_OK) fprintf(outfile,"  label       = %s\n",alabel);

reordr = 0;    /* easy to set default - if not specified, then */
               /* this line changes nothing */
errcod = ip_boolean("REORDER",&reordr,0);
if(reordr) {
   errcod = ip_count("MOORDER",&size,0);
   for(i=0; i < size ; i++) {
      errcod = ip_data("MOORDER","%d",&iorder[i],1,i);
      errchk(errcod,"MOORDER");
      }
   }
second_root = 0;
if (twocon) {
   errcod = ip_boolean("SECOND_ROOT",&second_root,0);
   }

if(iopen) {
   errcod = ip_count("SOCC",&size,0);
   if(errcod == IPE_OK && size != num_ir) {
      fprintf(outfile,"\n SOCC array is the wrong size\n");
      fprintf(outfile," is %d, should be %d\n",size,num_ir);
      exit(size);
      }
   if(errcod != IPE_OK) {
```

```
fprintf(outfile,"\n try adding some electrons buddy!\n");
fprintf(outfile," need SOCC\n");
ip_print_tree(outfile,NULL);
exit(1);
}
```

3.3 The Binary Input/Output System

3.3.1 The structure and philosophy of the library

Almost all `PSI3` modules must exchange data with raw binary (also called "direct-access") files. However, rather than using low-level C or Fortran functions such as `read()` or `write()`, `PSI3` uses a flexible, but fast I/O system that gives the programmer and user control over the organization and storage of data. Some of the features of the PSI I/O system, libpsio, include:

- A user-defined disk striping system in which a single binary file may be split across several physical or logical disks.
- A file-specific table of contents (TOC) which contains file-global starting and ending addresses for each data item.
- An entry-relative page/offset addressing scheme which avoids file-global file pointers which can limit file sizes.

The TOC structure of PSI binary files provdes several advantages over older I/O systems. For example, data items in the TOC are identified by keyword strings (e.g., `"Nuclear Repulsion Energy"`) and the *global* address of an entry is known only to the TOC itself, never to the programmer. Hence, if the programmer wishes to read or write an entire TOC entry, he/she is required to provide only the TOC keyword and the entry size (in bytes) to obtain the data. Furthermore, the TOC makes it possible to read only pieces of TOC entries (say a single buffer of a large list of two-electron integrals) by providing the appropriate TOC keyword, a size, and a starting address relative to the beginning of the TOC entry. In short, the TOC design hides all information about the global structure of the direct access file from the programmer and allows him/her to be concerned only with the structure of individual entries. The current TOC is written to the end of the file when it is closed.

Thus the direct-access file itself is viewed as a series of pages, each of which contains an identical number of bytes. The global address of the beginning of a given entry is stored on the TOC as a page/offset pair comprised of the starting page and byte-offset on that page where the data reside. The entry-relative page/offset addresses which the programmer must provide work in exactly the same manner, but the 0/0 position is taken to be the beginning of the TOC entry rather than the beginning of the file.

3.3.2 The user interface

All of the functions needed to carry out basic I/O are described in this subsection. Proper declarations of these routines are provided by the header file `psio.h`. Note that before any open/close functions may be called, the input parsing library, libipv1 must be initialized so that the necessary file striping information may be read from user input, but this is hidden from the programmer in lower-level functions. NB, `ULI` is used as an abbreviation for `unsigned long int` in the remainder of this manual.

`int psio_init(void)`: Before any files may be opened or the basic read/write functions of libpsio may be used, the global data needed by the library functions must be initialized using this function.

`int psio_done(void)`: When all interaction with the direct-access files is complete, this function is used to free the library's global memory.

`int psio_open(ULI unit, int status)`: Opens the direct access file identified by `unit`. The `status` flag is a boolean used to indicate if the file is new (0) or if it already exists and is being re-opened (1). If specified in the user input file, the file will be automatically opened as a multivolume (striped) file, and each page of data will be read from or written to each volume in succession.

`int psio_close(ULI unit, int keep)`: Closes a direct access file identified by unit. The keep flag is a boolean used to indicate if the file's volumes should be deleted (0) or retained (1) after being closed.

`int psio_read_entry(ULI unit, char *key, char *buffer, ULI size)`: Used to read an entire TOC entry identified by the string `key` from `unit` into the array `buffer`. The number of bytes to be read is given by `size`, but this value is only used to ensure that the read request does not exceed the end of the entry. If the entry does not exist, an error is printed to stderr and the program will exit.

`int psio_write_entry(ULI unit, char *key, char *buffer, ULI size)`: Used to write an entire TOC entry idenitified by the string `key` to `unit` into the array `buffer`. The number of bytes to be written is given by `size`. If the entry already exists and its data is being overwritten, the value of size is used to ensure that the write request does not exceed the end of the entry.

`int psio_read(ULI unit, char *key, char *buffer, ULI size, psio_address sadd, psio_address *eadd)`: Used to read a fragment of `size` bytes of a given TOC entry identified by `key` from `unit` into the array `buffer`. The starting address is given by the `sadd` and the ending address (that is, the entry-relative address of the next byte in the file) is returned in `*eadd`.

`int psio_write(ULI unit, char *key, char *buffer, ULI size, psio_address sadd, psio_address *eadd)`: Used to write a fragment of `size` bytes of a given TOC entry identified by `key` to `unit` into the array `buffer`. The starting address is given by the `sadd` and the ending address (that is, the entry-relative address of the next byte in the file) is returned in `*eadd`.

The page/offset address pairs required by the preceeding read and write functions are supplied via variables of the data type `psio_address`, defined by:

```
typedef struct {
  ULI page;
  ULI offset;
} psio_address;
```

The `PSIO_ZERO` defined in a macro provides a convenient input for the 0/0 page/offset.

3.3.3 Manipulating the table of contents

In addition, to the basic open/close/read/write functions described above, the programmer also has a limited ability to directly manipulate or examine the data in the TOC itself.

`int psio_tocprint(ULI unit, FILE *outfile)`: Prints the TOC of `unit` in a readable form to `outfile`, including entry keywords and global starting/ending addresses. (`tocprint` is also the name of a `PSI3` utility module which prints a file's TOC to stdout.)

`int psio_toclen(ULI unit, FILE *outfile)`: Returns the number of entries in the TOC of `unit`.

`int psio_tocdel(ULI unit, char *key)`: Deletes the TOC entry corresponding to `key`. NB that this function only deletes the entry's reference from the TOC itself and does not remove the corresponding data from the file. Hence, it is possible to introduce data "holes" into the file.

`int psio_tocclean(ULI unit, char *key)`: Deletes the TOC entry corresponding to `key` and all subsequent entries. As with `psio_tocdel()`, this function only deletes the entry references from the TOC itself and does not remove the corresponding data from the file. This function is still under construction.

3.3.4 Using `libpsio.a`

The following code illustrates the basic use of the library, as well as when/how the `psio_init()` and `psio_done()` functions should be called in relation to initialization of `libipv1`. (See section 3.1 later in the manual for a description of the basic elements of C-language `PSI3` program.)

```
#include <stdio.h>
#include <libipv1/ip_lib.h>
#include <libpsio/psio.h>
#include <libciomr/libciomr.h>

FILE *infile, *outfile;

int main()
```

```
{
  int i, M, N;
  double enuc, *some_data;
  psio_address next;  /* Special page/offset structure */

  ffile(&infile,"input.dat",2);
  ffile(&outfile,"output.dat",1);
  ip_set_uppercase(1);
  ip_initialize(infile,outfile);
  ip_cwk_add(":DEFAULT");
  ip_cwk_add(progid);

  /* Initialize the I/O system */
  psio_init();

  /* Open the file and write an energy */
  psio_open(31, PSIO_OPEN_NEW);
  enuc = 12.3456789;
  psio_write_entry(31, "Nuclear Repulsion Energy", (char *) &enuc,
                   sizeof(double));
  psio_close(31,1);

  /* Read M rows of an MxN matrix from a file */
  some_data = init_matrix(M,N);

  psio_open(91, PSIO_OPEN_OLD);
  next = PSIO_ZERO;/* Note use of the special macro */
  for(i=0; i < M; i++)
      psio_read(91, "Some Coefficients", (char *) (some_data + i*N),
                N*sizeof(double), next, &next);
  psio_close(91,0);

  /* Close the I/O system */
  psio_done();

  ip_done();
}

char *gprgid()
{
   char *prgid = "CODE_NAME";
   return(prgid);
}
```

The interface to the PSI3 I/O system has been designed to mimic that of the old `wreadw()` and `wwritw()` routines of `libciomr` (see the next section of this manual). The table of contents system introduces a few complications that users of the library should be aware of:

- As pointed out earlier, deletion of TOC entries is allowed using `psio_tocdel()` and `psio_tocclean()`. However, since only the TOC reference is removed from the file and the corresponding data is not, a data hole will be left in the file if the deleted entry was not the last one in the TOC. A utility function designed to "defrag" a PSI file may become necessary if such holes ever present a problem.
- One may append data to an existing TOC entry by simply writing beyond the entry's current boundary; the ending address data in the TOC will be updated automatically. However, no safety measures have been implemented to prevent one from overwriting data in a subsequent entry thereby corrupting the TOC. This feature/bug remains because (1) it is possible that such error checking functions may slow the I/O codes significantly; (2) it may be occasionally desirable to overwrite exiting data, regardless of its effect on the TOC. Eventually a utility function which checks the validity of the TOC may be needed if this becomes a problem, particularly for debugging purposes.

4 Other PSI3 C Libraries

There are several other PSI C libraries besides the previously-mentioned `libipv1.a`, `libpsio.a`, and `libciomr.a`:

`libchkpt.a` This library provides many routines for reading from and writing to the "checkpoint" file, `file30`. There is generally a different function associated with each quantity in `file30` (such as the SCF energy, nuclear repulsion energy, geometry, basis set information, etc). This library uses the `libpsio.a` library to do its input and output. It replaces an older library `libfile30.a` which served the same purpose but which used the old I/O from `libciomr.a`.

`libiwl.a` The new format for storing two-electron integrals is IWL, or "integrals with labels." The library `libiwl.a` provides functions for reading and writing files in the IWL format. The code was written with the goal that it could be easily modified to allow for more than 256 basis functions. Its current limit is 32768 basis functions.

`libqt.a` This is the "Quantum Trio" library, which contains a number of very experimental functions or functions which don't otherwise fit anywhere else.

In this section we will consider these libraries in greater detail.

4.1 The Checkpoint File Library

4.1.1 Library Philosophy

The `libchkpt.a` library is a collection of functions used to access the `PSI3` checkpoint file (`file32`) – the file which contains all most frequently used information about the computation such as molecular geometry, basis set, HF determinant, etc. Previously, the checkpoint file was a fixed-format file which is accessed using the old `PSI3` I/O system. However, this changed in the spring of 2002 to use the new `libpsio.a` I/O system to access the checkpoint file, and it is now free format. That is, any programmer can add content to the file at will. The old checkpoint file interface has been updated to access the new underlying I/O system. It is *mandatory* that the checkpoint file is accessed via the `libchkpt.a` functions *only*.

4.1.2 Basic Use Instructions

Following the philosophy that a programmer who wants to read, say, the number of atoms and the irrep labels from the checkpoint file should not have to use fifty lines of code to do so, `libchkpt.a` was written. Following a call to a single command, `chkpt_init()`, the programmer can extract many useful bits of info from the checkpoint file relatively painlessly. `libchkpt.a` is dependent upon `libipv1.a` and `libpsio.a` and thus requires that the input parser and I/O system each be initialized so that the proper file name labels may be referenced. An example of a minimal program that sets up the input parser, initilizes a special structure within the `libchkpt.a` library, and reads the SCF HF energy, eigenvector and eigenvalues is given below. In order to illustrate the writing capability of the library routines, a dummy correlated energy is written to the checkpoint file and then read back again within the code.

```
#include <stdio.h>
#include <libipv1/ip_lib.h>
#include <libciomr/libciomr.h>
#include <libpsio/psio.h>
#include <libchkpt/chkpt.h>

FILE *infile, *outfile;

void main(void)
{

  int nmo;
  double escf, etot;
  double *evals;
  double **scf;

  /*-------------------------------------
```

```
  initialize the input parser, read in
  the files information from the
  default section
 ------------------------------------*/
ffile(&infile,"input.dat",2);
ffile(&outfile,"output.dat",1);
tstart(outfile);
ip_set_uppercase(1);
ip_initialize(infile,outfile);
ip_cwk_clear();
ip_cwk_add(":DEFAULT");
psio_init();

/*-----------------------------------
  now initialize the checkpoint structure
  and begin reading info
 ------------------------------------*/
chkpt_init(PSIO_OPEN_OLD);

escf = chkpt_rd_escf();
evals = chkpt_rd_evals();
scf = chkpt_rd_scf();
nmo = chkpt_rd_nmo();

chkpt_wt_etot(-1000.0);

etot = chkpt_rd_etot();

chkpt_close();

/*-----------------------------------------
  print out info to see what has been read in
 -----------------------------------------*/
fprintf(outfile,"\n\n\tEscf  = %20.10lf\n",escf);
fprintf(outfile,"\tEtot = %20.10lf\n",etot);
fprintf(outfile,"SCF EIGENVECTOR\n");

eivout(scf,evals,nmo,nmo,outfile);

psio_done();
tstop(outfile);
ip_done();
}

/*-------------------------------------------------
```

```
    dont forget to add the obligatory gprgid section
   ------------------------------------------------*/
char *gprgid()
{
   char *prgid = ":TEST";

   return(prgid);
}
```

4.1.3 Initialization

`int chkpt_init()`
Initializes the `checkpoint` struct to allow other `chkpt_*` functions to perform their duties.

Arguments: the `libpsio` status marker PSIO_OPEN_OLD; also requires that the input parser be initialized so that it can open the checkpoint file.
Returns: zero. Perhaps this will change some day.

`int chkpt_close()`
Closes the checkpoint file, frees memory, etc.

Arguments: none, but `chkpt_init` must already have been called for this to work.
Returns: zero. Perhaps this, too, will change one day.

4.1.4 Functions for reading information from the checkpoint file

This section gives an overview of many of the most widely used functions from `libchkpt.a`. For more details and descriptions of newer functions that are not yet described here, see the source code itself.

Functions that return `char*`

`char *chkpt_rd_corr_lab()`
Reads in a label from the checkpoint file which describes the wavefunction used to get the correlated energy which is stored in the checkpoint file (see `chkpt_rd_ecorr()`).

Arguments: takes no arguments.
Returns: a string, like "CISD", or "MCSCF" or some other wavefunction designation.

`char *chkpt_rd_label()`
Reads the main the checkpoint file label.

Arguments: takes no arguments.
Returns: calculation label.

`char *chkpt_rd_sym_label()`
Reads the label for the point group.

Arguments: takes no arguments.
Returns: point group label.

Functions that return `char**`

`char **chkpt_rd_irr_labs()`
Read in the symmetry labels for all irreps in the point group in which the molecule is considered.

Arguments: takes no arguments.
Returns: an array of labels (strings) which denote the irreps for the point group in which the molecule is considered, _regardless_ of whether there exist any symmetry orbitals which transform as that irrep.

`char **chkpt_rd_hfsym_labs()`
Read in the symmetry labels _only_ for those irreps which have basis functions.

Arguments: takes no arguments.
Returns: an array of labels (strings) which denote the irreps which have basis functions (in Cotton ordering). For DZ or STO-3G water, for example, in C_{2v} symmetry, this would be an array of three labels: "A1", "B1", and "B2".

Functions that return `int`

`int chkpt_rd_iopen()`
Reads in the dimensionality (up to a sign) of ALPHA and BETA vectors of two-electron coupling coefficients for open shells (see `chkpt_rd_ccvecs()`). Note : `iopen` = MM * (MM + 1), where MM is the total number of irreps containing singly occupied orbitals.

Arguments: takes no arguments.
Returns: the +/- dimensionality of ALPHA and BETA vectors of coupling coefficients for open shells.

`int chkpt_rd_max_am()`
Reads in the maximum orbital quantum number of AOs in the basis.

Arguments: takes no arguments.
Returns: the maximum orbital quantum number of AOs in the basis.

`int chkpt_rd_mxcoef()`
Reads the value of the constant `mxcoef`.

Arguments: takes no arguments.
Returns: the sum of the squares of the number of symmetry orbitals for each irrep. This gives the number of elements in the non-zero symmetry blocks of the SCF eigenvector. For STO-3G water `mxcoef`= $(4*4)+(0*0)+(1*1)+(2*2)=21$.

`int chkpt_rd_nao()`
Reads in the total number of atomic orbitals (read: Cartesian Gaussian functions).

Arguments: takes no arguments.
Returns: total number of atomic orbitals.

`int chkpt_rd_natom()`
Reads in the total number of atoms.

Arguments: takes no arguments.
Returns: total number of atoms.

`int chkpt_rd_ncalcs()`

Reads in the total number of calculations in the checkpoint file (was always 1 in old `libfile30.a`, probably still is for now).

Arguments: takes no arguments.
Returns: total number of calculations in the checkpoint file.

`int chkpt_rd_nirreps()`

Reads in the total number of irreducible representations in the point group in which the molecule is being considered.

Arguments: takes no arguments.
Returns: total number of irreducible representations.

`int chkpt_rd_nmo()`

Reads in the total number of molecular orbitals (may be different from the number of basis functions).

Arguments: takes no arguments.
Returns: total number of molecular orbitals.

`int chkpt_rd_nprim()`

Reads in the total number of primitive Gaussian functions (only primitives of _symmetry independent_ atoms are counted!).

Arguments: takes no arguments.
Returns: total number of primitive Gaussian functions.

`int chkpt_rd_nshell()`

Reads in the total number of shells. For example, DZP basis set for carbon atom (contraction scheme $[9s5p1d/4s2p1d]$) has a total of 15 basis functions, 15 primitives, and 7 shells. Shells of _all_ atoms are counted (not only of the symmetry independent; compare `chkpt_rd_nprim`).

Arguments: takes no arguments.
Returns: total number of shells.

`int chkpt_rd_nso()`

Reads in the total number of symmetry-adapted basis functions (read: Cartesian or Spherical Harmonic Gaussians).

Arguments: takes no arguments.
Returns: total number of SOs.

`int chkpt_rd_nsymhf()`

Reads in the total number of irreps in the point group in which the molecule is being considered which have non-zero number of basis functions. For STO-3G or DZ water, for example, this is three, even though `nirreps` is 4 (compare `int chkpt_rd_nirreps()`).

Arguments: takes no arguments.
Returns: total number of irreducible representations with a non-zero number of basis functions.

`int chkpt_rd_num_unique_atom()`

Reads in the number of symmetry unique atoms.

Arguments: takes no arguments.
Returns: number of symmetry unique atoms.

`int chkpt_rd_num_unique_shell()`

Reads in the number of symmetry unique shells.

Arguments: takes no arguments.
Returns: number of symmetry unique shells.

`int chkpt_rd_phase_check()`

Reads the phase flag, which is 1 if the orbital phases have been checked and is 0 otherwise (phase checking just helps ensure the arbitrary phases of the orbitals are consistent from one geometry to the next, which helps various guessing or extrapolation schemes).

Arguments: takes no arguments.
Returns: flag.

`int chkpt_rd_ref()`

Reads the reference type from the flag in the checkpoint file. 0 = RHF, 1 = UHF, 2 = ROHF, 3 = TCSCF.

Arguments: takes no arguments.
Returns: flag indicating the reference.

`int chkpt_rd_rottype()`

Reads the rigid rotor type the molecule represents. 0 = asymmetric, 1 = symmetric, 2 = spherical, 3 = linear, 6 = atom.

Arguments: takes no arguments.
Returns: rigid rotor type.

Functions that return `int*`

`int *chkpt_rd_am2canon_shell_order()`

Reads in the the mapping array from the angmom-ordered to the canonical (in the order of appearance) list of shells.

Arguments: takes no arguments.
Returns: an array `nshell` long that maps shells from the angmom-ordered to the canonical (in the order of appearance) order.

`chkpt_rd_atom_position()`

Reads in symmetry positions of atoms. Allowed values are as follows:

- 1 - atom in a general position
- 2 - atom on the c2z axis
- 4 - atom on the c2y axis
- 8 - atom on the c2x axis
- 16 - atom in the inversion center
- 32 - atom in the sigma_xy plane

- 64 - atom in the sigma_xz plane
- 128 - atom in the sigma_yz plane

This data is sufficient to define stabilizers of the nuclei.

Arguments: takes no arguments.
Returns: an array of symmetry positions of atoms.

`int *chkpt_rd_clsdpi()`

Reads in an array which has an element for each irrep of the point group of the molecule (n.b. not just the ones with a non-zero number of basis functions). Each element contains the number of doubly occupied MOs for that irrep.

Arguments: takes no arguments.
Returns: the number of doubly occupied MOs per irrep.

`int *chkpt_rd_openpi()`

Reads in an array which has an element for each irrep of the point group of the molecule (n.b. not just the ones with a non-zero number of basis functions). Each element contains the number of singly occupied MOs for that irrep.

Arguments: takes no arguments.
Returns: the number of singly occupied MOs per irrep.

`int *chkpt_rd_orbspi()`

Reads in the number of MOs in each irrep.

Arguments: takes no arguments.
Returns: the number of MOs in each irrep.

`int *chkpt_rd_shells_per_am()`

Reads in the number of shells in each angmom block.

Arguments: takes no arguments.
Returns: the number of shells in each angmom block.

`chkpt_rd_sloc()`

Read in an array of pointers to the first AO from each shell.

Arguments: takes no arguments.
Returns: Read in an array `nshell` long of pointers to the first AO from each shell.

`chkpt_rd_sloc_new()`

Read in an array of pointers to the first basis function (not AO as `chkpt_rd_sloc` does) from each shell.

Arguments: takes no arguments.
Returns: an array `nshell` long of pointers to the first basis function from each shell.

`int *chkpt_rd_snuc()`

Reads in an array of pointers to the nuclei on which shells are centered.

Arguments: takes no arguments.
Returns: an array `nshell` long of pointers to the nuclei on which shells are centered.

`int *chkpt_rd_snumg()`

Reads in array of the numbers of the primitive Gaussians in the shells.

Arguments: takes no arguments.
Returns: an array `nshell` long of the numbers of the primitive Gaussians in shells.

`int *chkpt_rd_sprim()`

Reads in pointers to the first primitive from each shell.

Arguments: takes no arguments.
Returns: an array `nshell` long of pointers to the first primitive from each shells.

`chkpt_rd_sopi()`

Read in the number of symmetry-adapted basis functions in each symmetry block.

Arguments: takes no arguments.
Returns: an array nirreps long of the numbers of symmetry orbitals in symmetry blocks.

`int *chkpt_rd_stype()`

Reads in angular momentum numbers of the shells.

Arguments: takes no arguments.
Returns: Returns an array `nshell` long of the angular momentum numbers of the shells.

`int *chkpt_rd_symoper()`

Read in the mapping array between "canonical" ordering of the symmetry operations of the point group and the one defined in `symmetry.h`.

Arguments: takes no arguments.
Returns: a mapping array `nirrep` long

`int *chkpt_rd_ua2a()`

Read in the mapping array from the symmetry-unique atom list to the full atom list.

Arguments: takes no arguments.
Returns: a mapping array `num_unique_atom` long

`int *chkpt_rd_us2s()`

Read in the mapping array from the symmetry-unique shell list to the full shell list.

Arguments: takes no arguments.
Returns: a mapping array `num_unique_shell` long

Functions that return `int**`

`int **chkpt_rd_ict()`

Reads the transformation properties of the nuclei under the operations allowed for the particular symmetry point group in which the molecule is considered.

Arguments: takes no arguments.
Returns: a matrix of integers. Each row corresponds to a particular symmetry operation, while each column corresponds to a particular atom. The value of `ict[2][1]`, then, should be interpreted in the following manner: application of the third symmetry operation of the relavant point group, the second atom is placed in the location originally occupied by the atom number `ict[2][1]`.

`int **chkpt_rd_shell_transm()`

Reads in the transformation matrix for the shells. Each row of the matrix is the orbit of the shell under symmetry operations of the point group.

Arguments: takes no arguments.
Returns: a matrix of `nshell*nirreps` integers.

Functions that return `double`

`double chkpt_rd_ecorr()`
Reads in the correlation energy stored in the checkpoint file. To get some information (a label) on the type of correlated wavefunction used to get this energy, see `chkpt_rd_corr_lab()`.

Arguments: takes no arguments.
Returns: the correlation energy.

`double chkpt_rd_enuc()`
Reads in the nuclear repulsion energy

Arguments: takes no arguments.
Returns: the nuclear repulsion energy.

`double chkpt_rd_eref()`
Reads in the reference energy (may be different from HF energy).

Arguments: takes no arguments.
Returns: the reference energy.

`double chkpt_rd_escf()`
Reads in the SCF HF energy.

Arguments: takes no arguments.
Returns: the SCF HF energy.

`double chkpt_rd_etot()`
The total energy, be it HF, CISD, CCSD, or whatever! This is the preferred function to use for geometry optimization via energies, printing energies in analysis, etc., since this value is valid whatever the calculation type.

Arguments: takes no arguments.
Returns: The total energy.

Functions that return `double*`

`double *chkpt_rd_evals()`
`double *chkpt_rd_alpha_evals()`
`double *chkpt_rd_beta_evals()`
Reads in the (spin-restricted HF, α UHF, and β UHF) eigenvalues: the orbital energies.

Arguments: take no arguments.
Returns: an array of _all_ of the SCF eigenvalues, ordered by irrep, and by increasing energy within each irrep. (i.e. for STO-3G water, the four a_1 eigenvalues all come first, and those four are ordered from lowest energy to highest energy, followed by the single b_1 eigenvalue, etc. — Pitzer ordering)

`double *chkpt_rd_exps()`
Reads in the exponents of the primitive Gaussian functions.

Arguments: takes no arguments.
Returns: an array of doubles.

`double *chkpt_rd_zvals()`

Reads in nuclear charges.

Arguments: takes no arguments.
Returns: an array natom long of nuclear charges (as doubles).

Functions that return `double**`

`double **chkpt_rd_blk_scf(int irrep)`
`double **chkpt_rd_alpha_blk_scf(int irrep)`
`double **chkpt_rd_beta_blk_scf(int irrep)`

Reads in a symmetry block of the (RHF, α UHF, β UHF) eigenvector.

Arguments: `int irrep`, designates the desired symmetry block
Returns: a square matrix has `orbspi[irrep]` rows. The eigenvectors are stored with the column index denoting MOs and the row index denoting SOs: this means that `scf_vector[i][j]` is the contribution of the ith SO to the jth MO.

`double **chkpt_rd_ccvecs()`

Reads in a matrix rows of which are ALPHA (ccvecs[0]) and BETA (ccvecs[1]) matrices of coupling coefficients for open shells stored in lower triangular form. Coupling coefficients are defined NOT as in C.C.J.Roothaan Rev. Mod. Phys. **32**, 179 (1960) as it is stated in the manual pages for CSCF, but according to Pitzer (no reference yet) and are **different** from those in Yamaguchi, Osamura, Goddard, and Schaefer's book "Analytic Derivative Methods in Ab Initio Molecular Electronic Structure Theory".
The relationship between the Pitzer's and Yamaguchi's conventions is as follows : ALPHA = 1-2*a , BETA = 1+4*b , where a and b are alpha's and beta's for open shells defined on pp. 69-70 of Dr. Yamaguchi's book.

Arguments: takes no arguments.
Returns: double **ccvecs, a matrix 2 by `abs(iopen)` rows of which are coupling coefficient matrices for open-shells in packed form. For definition of `iopen` see chkpt_rd_iopen().

`chkpt_rd_contr_full()`

Reads in the normalized contraction coefficients.

Arguments: takes no arguments.
Returns: a matrix `MAXANGMOM` (a constant defined in ???) by the total number of primitives `nprim`; each primitive Gaussian contributes to only one shell (and one basis function, of course), so most of these values are zero.

`double **chkpt_rd_geom()`

Reads in the cartesian geometry.

Arguments: takes no arguments.
Returns: The cartesian geometry is returned as a matrix of doubles. The row index is the atomic index, and the column is the cartesian direction index (x=0, y=1, z=2). Therefore, `geom[2][0]` would be the x-coordinate of the third atom.

`chkpt_rd_lagr()`
`chkpt_rd_alpha_lagr()`
`chkpt_rd_beta_lagr()`
Reads in an (RHF, α UHF, β UHF) Lagrangian matrix in MO basis.

Arguments: takes no arguments.
Returns: a matrix `nmo` by `nmo`.

`double **chkpt_rd_scf()`
`double **chkpt_rd_alpha_scf()`
`double **chkpt_rd_beta_scf()`
Reads in the (RHF, α UHF, β UHF) eigenvector.

Arguments: takes no arguments.
Returns: a square matrix of dimensions `nmo` by `nmo` (see: `chkpt_rd_nmo()`). The symmetry blocks of the SCF vector appear on the diagonal of this matrix.

`chkpt_rd_schwartz()`
Reads in the table of maxima of Schwartz integrals (ij—ij) for each shell doublet.

Arguments: takes no arguments.
Returns: `NULL` if no table is present in the checkpoint file, a matrix `nshell` by `nshell` otherwise.

`chkpt_rd_usotao_new()`
Reads in an AO to SO transformation matrix.

Arguments: takes no arguments.
Returns: a `nso` by `nao` matrix of doubles.

`chkpt_rd_usotbf()`
Reads in a basis function to SO transformation matrix.

Arguments: takes no arguments.
Returns: a `nso` by `nso` matrix of doubles.

Functions that return `struct *z_entry`

The z-matrix is read from the checkpoint file as an array of `z_entry` structs which are declared in `chkpt.h`. This structure contains the reference atom, an optimization flag, the coordinate value, and any label used for each internal coordinate. When not applicable (such as the first few lines of a z-matrix) `atom` variables are given values of -1, `opt` variables are given values of -1, `val` variables are given values of -999.9, and `label` strings are left empty.

`chkpt_rd_zmat()`
Reads in the z-matrix

Arguments: takes no arguments.
Returns: `struct *z_entry` natom long.

4.2 The Integrals-With-Labels Library

The library `libiwl.a` contains functions for reading and writing to files with the "Integrals With Labels" (IWL) format created by David Sherrill in 1994, modeled after the format of the old integrals file from `PSI2`. Most functions deal with four-index quantitites, but there are also a few which deal with two-index quantities such as one-electron integrals. The IWL format specifies that the 4-index quantities are stored on disk in several buffers; each buffer has a header segment which gives some useful info. Currently, the header is arranged as follows: one integer word is used as a flag, telling whether the current buffer is the last buffer in the file. The next integer gives the number of integrals (and their associated labels) in the current buffer. After this header information, each buffer contains two data segments: one for labels, and one for the values of the associated integrals. The datasize for the labels is defined using typedefs, so it is easy to change (currently, it is a short int); likewise for the integral values (currently of type double). The length of these data segments is NBUF * 4 * sizeof(Label) and NBUF * sizeof(Value), respectively. The current use of short ints for Label is really somewhat excessive, making the files somewhat larger than strictly necessary. However, this avoids confusing bit-packing schemes, and instantly allows us to have up to something like 65,536 basis functions addressable.

The functions previously documented in this manual have been removed because that documentation is now out of date. Documentation of the library is now created directly from the source code using the `doxygen` program and is available at `http://www.psicode.org/libs/doxygen/html`.

4.3 The "Quantum Trio" Library

The `libqt.a` library is a miscellaneous collection of useful math and other routines. The documentation previously found in this manual of `libqt.a` functions has been removed and is now obsolete. The current documentation of this library is generated automatically from the `doxygen` program and is available on the website at `http://www.psicode.org/libs/doxygen/html`.

5 Programming Style

In the context of programming, *style* can refer to many things. Foremost, it refers to the format of the source code: how to use indentation, when to add comments, how to name variables, etc. It can also refer to many other issues, such code organization, modularity, and efficiency. Of course, stylistic concerns are often matters of individual taste, but often validity and portability of the code will ultimately depend on stylistic decisions made in the process of code development. Hence some stylistic choices are viewed as universally bad (e.g. not prototyping every function just because "the code compiles and runs fine as is", etc.). Admittedly, it is easy to not have any style, but it takes years to learn what makes a good one. A good programming style can reduce debugging and maintenance times dramatically. For a large package such as `PSI3`, it is very important to adopt a style which makes the code easy to understand and modify by others. This section will give a few brief pointers on what

we consider to be a good style in programming.

5.1 On the Process of Writing Software

At first, we feel appropriate to touch upon the issue of programming style as referred to the approach to writing software. Often, "programming" is used to mean "the process of writing software". In general one has to distinguish "writing software" from "programming" meaning "implementation", because the latter is only a part of the former and does not include documentation, etc. In general, "writing software" should consist of five parts:

1. Get a clear and detailed understanding of what the code has to do (idea);
2. Identify key concepts and layout code and data organization (design);
3. Write source code (implementation);
4. Test the program and eliminate errors and/or design flaws (testing);
5. Write documentation (documentation).

Thus, writing software is significantly more complex than just coding. Each stage of writing software is as important as others and should not be considered a waste of time. The code written without a detailed understanding of what it has to do may not work properly. Poorly designed code may not be flexible enough to accomodate some new feature and will be rewritten. Poorly implemented code may be too slow to be useful. A paper full of incorrect values produced by your code may get you fired and will destroy your reputation. A documentation-free code will most likely be useless for others.

Of course, for very simple programs design and implementation may be combined and documentation may consist of one line. However, for more complex programs it is recommended that the five stages are followed. This means that you should spend only about 20-40% of your time writing source code! Our experience shows that following this scheme results in the most efficient approach to programming in the long run.

To learn more on each stage of the software writing process, you may want to refer to Stroustrup's "C++ Programming Language" book (3rd Ed.) as the most common reference source not dedicated solely to one narrow subject. Besides being an excellent description of C++, it is also an introduction to writing software as well. Particular attention is paid to the issue of *program design*.

5.2 Design Issues

Although C lacks the most powerful features of C++ as far as concepts and data organization is concerned, Stroustrup says: "Remember that much programming can be simply and clearly done using only primitive, data structures, plain functions, and a few library classes." This means that one can write many useful and *well-written* programs in C. Here are a few pointers that will assist you in structuring your C program:

- Identify groups of variables having common function (e.g. basis set, etc.) and organize them into structures. Use several levels of hierarchy if necessary (e.g. a basis set is a collection of basis functions each of which may be described by a structure). This is called "hierarchical ordering".

- Think as generally as possible. What you may not need today will be asked for tomorrow. Design data structures that are flexible and modular, i.e. one can be easily modified without affecting the others (e.g. you do not want the structure describing basis sets to know anything about the type of basis functions it contains so that plane waves can be used as easily as Gaussians).

- Write "constructors" for the structures, i.e. functions which will initialize data in the structures (e.g. read basis set information). Make as many "constructors" as necessary (e.g. basis set info can be read from the checkpoint file or from `pbasis.dat`). If it is difficult or impossible to write a "constructor" for some data structure is a sign that your data hierachy is poorly designed and there are mutual dependencies. Spend more time designing the system. If it doesn't help, then use source code comments heavily to describe the relationships not reflected in the code itself.

- Use global variables sparringly. Placing a variable into global scope leaves it unprotected against "unauthorized" use or modification (we are not talking about security here; it is a good idea to protect data from the programmer, because if you do not want some data `A` to be modified by function `B`, do not make `A` available to `B`) and may also have impact on program's performance. Sometimes it is a good idea to use global data to reduce the cost of passing that data to a function. However, the same effect may be achieved by organizing that data into a local structure and passing the structure instead.

- Learn how to use `static` variables local to a source file, it is a very powerful tool to protect data in a C program.

- Organize the source code such as to emphasize further the structure of the program (see section 5.3).

More material on data organization may be found in the Stroustrup's book.

5.3 Organization of Source Code

It is almost universally agreed that breaking the program up into several files is good style. An 11,592 line Fortran program, for example, is very inconvenient to work with, for several reasons: first, it can be difficult to locate a particular function[1] or statement; second, every recompilation during debugging involves compiling the *entire* file. Having several small files generally makes it easier to find a particular piece of code, and only source files which have been modified need to be recompiled, greatly enhancing the efficiency of the programmer

[1] Following the convention of C, the words function and subroutine will be used interchangeably.

during the debugging process. For smaller programs, it is recommended that the programmer have one file for each subroutine, giving each file the name of the subroutine (abbreviated filenames may be specified if the function names are too long). For larger programs, it may be helpful to group similar functions together into a single file.

In C programs, we also consider it a good idea to place all the `#include` statements in a file such as `includes.h`, which is subsequently included in each relevant C source file. This is helpful because if a new header file needs to be added, it can simply be added to `includes.h`. Furthermore, if a source file suddenly needs to have access to a global variable or function prototype which is already present in one of the header files, then no changes need to be made; the header file is *already* included. A downside to this approach is that each header file is included in every source file which includes `includes.h`, regardless of whether a particular header file is actually needed by that source file; this could potentially lead to longer compile times, but it isn't likely to make a discernable difference, at least in C.[2]

Along similar lines, it is helpful to *define* all global variables in one location (in the main program file, or else within `globals.c`), and they should be *declared* within another standard location (perhaps `globals.h`, or `common.h`).[3] Similarly, if functions are used in several different source code files, the programmer may wish to place all function prototype declarations in a single header file, with the same name as the program or library, or perhaps called `protos.h`.

5.4 Formatting the Code

By formatting, we mean how many spaces to indent, when to indent, how to match up braces, when to use capital vs. lower case letters, and so forth. This is perhaps a more subjective matter than those previously discussed. However, it is certainly true that some formatting styles are easier to read than others. For already existing code, we recommend that you conform to the formatting convention already present in the code. The author of the code is likely to get upset when he sees that you're incorporated code fragments with a formatting style which differs from his! On the other hand, in certain rare cases, it might be more beneficial to incorporate a different style: in the conversion of `intder95` from old-style to new-style input, we used lower-case lettering instead of the all-caps style of the original program. This was very useful in helping us locate which changes we had made.

It is very common that statements within loops are indented. Loops within loops are indented yet again, and so on. This practice is near-universal and very helpful. Computational chemistry programs often require many nested loops. The consequence of this is that lines can be quite long, due to all those spaces before each line in the innermost loops. If the lines become longer than 80 characters, they are hard to read within a single window; please try to keep your lines to 80 characters or less. This means that you should use about 2-4 spaces per indentation level.

[2] C++, which includes much of the actual code in header files, is a different matter.

[3] See page 33 of Kernighan and Ritchie, 2nd Ed., for an explanation of *definition vs. declaration.*

Table 1: Some Variable Naming Conventions in PSI3

Quantity	Variable(s)
Number of atoms	na, natom, num_atoms
Number of atoms * 3	natom3, num_atoms3
Nuclear repulsion energy	enuc, repnuc
SCF energy	escf
Number of atomic orbitals	nbfao, num_ao
Number of symmetry orbitals	nbfso, num_so
Size of lower triangle of AO's, SO's	nbatri, nbstri; ntri
Input file pointer	infile
Output file pointer	outfile
Offset array	ioff
Number of irreps	num_ir, nirreps
Open-shell flag	iopen
Number of orbitals per irrep	orbs_per_irrep, orbspi, mopi
Number of closed-shells per irrep	docc, clsd_per_irrep, clsdpi
Number of open-shells per irrep	socc, open_per_irrep, openpi
Orbital symmetry array	orbsym

The matching of braces, and so forth, is more variable, and we recommend you follow the convention of *The C Programming Language*, by Kernighan and Ritchie, or perhaps the style found in other PSI3modules.

5.5 Naming of Variables

All non-trivial data must be given descriptive names, although extremely long names are discouraged. For example, compound variable names like `num_atoms` or `atom_orbit_degen` should be preferred to `nat` or `atord`, so that non-specialists could understand the code. It is also a good idea to put a descriptive comment where a non-trivial variable is declared. However, simple loop indices should generally be named `i,j,k` or `p,q,r`.

PSI3 programs have certain conventions in place for names of most common variables, as shown in the Table 1.

5.6 Printing Conventions

At the moment, there isn't really a standard method for a PSI program to determine how much information to print to `output.dat`. Some older PSI3 modules read a flag usually

Table 2: Proposed Conventions for Printing Level

0	Almost no printing; to be used by driver programs with -quiet option
1	Usual printing (default)
2	Verbose printing
3	Some debugging information
4	Substantial debugging information
5	Print almost all intermediates unless arrays too large
6	Print everything

called `IPRINT` which is a decimal representation of a binary number. Each bit is a printing option (yes or no) for the different intermediates particular to the program.

A practice which is probably preferable is to have a different print flag (boolean) for each of the major intermediates used by a program, and to have an overall print option (decimal) whose value determines the printing verbosity for the quantities without a specific printing option. The overall print option should be specified by a keyword `PRINT_LVL`, and its action should be as in Table 2.

5.7 Commenting Source Code

It is absolutely mandatory that each source file contains a reasonable number of comments. When a significant variable, data type, or function is declared, it must be accompanied with some descriptive information written in English. Every function prototype or body of it has to be preceeded by a short description of its purpose, algorithm (desirable; if it is too complex, provide a reference), what arguments it takes and what it returns.

Having said this, we will argue against excessive commenting: don't add a comment every time you do `i++`! It will actually make your code harder to read. Be sensible.

As of spring 2002, we have adopted the `doxygen` program to automatically generate source code documentation. This program scans the source code and looks for special codes which tell it to add the given comment block to the documentation list. The program is very fancy and can generate documentation in man, html, latex, and rtf formats. The file `psi3.dox` is the `doxygen` configuration file. The source code should be commented in the following way to work with `doxygen`.

The first file of each library defines a "module" via a special comment line:

```
/*! \defgroup PSIO libpsio: The PSI I/O Library */
```

Note the exclamation mark above — it is required by `doxygen`. The line above defines the `PSIO` key and associates it with the title "The PSI I/O Library." Each file belonging to this group will have a special comment of the following form:

```
/*!
** \file close.c
** \ingroup (PSIO)
*/
```

This tells `doxygen` that file `close.c` should be documented, it should be added to the list of documented files, and it belongs to the `PSIO` group.

All functions should be commented as in the following:

```
/*!
** PSIO_CLOSE(): Closes a multivolume PSI direct access file.
**
**  \param unit = The PSI unit number used to identify the file to all read
**                and write functions.
**  \param keep = Boolean to indicate if the file should be deleted (0) or
**                retained (1).
**
** Returns: always returns 0
**
** \ingroup (PSIO)
*/

int psio_close(ULI unit, int keep)
...
```

This will add the function `psio_close` to the list, associate it with the `PSIO` module, and define the various arguments.

6 Makefiles in `PSI3`

The `make` utility is designed to help maintain the many components of a large program, such as PSI. This section will describe the construction and usefulness of Makefiles in PSI, both in developmental code and in production-level modules. We will be concerned only with the GNU Project's `make` facility, and not older, less flexible versions. (For a complete explanation of GNU's `make`, see `info make` or go to `www.gnu.org`).

6.1 `Makefile` Structure

The primary purpose of the `make` program is to assist compilation and recompilation of a multi-file program, such that only those portions of the program are recompiled that require it. For example, if a header file is changed, then each source file which `#includes` that file must be recompiled. `make` provides an easy mechanism by which such *dependencies* (also called *prerequisites*) may be tracked.

`Makefiles` consist of *rules* which describe how to carry out commands. For example, a rule might explain how to compile a single source file, or how to link all the object files into the executable, or perhaps how to clean up all the object files. A rule has the following form

```
target: dependencies
        command
        command
        ...
```

The *target* is the name of the rule, e.g. the name of the program or file to be compiled. The first rule given in the `Makefile` is the default. The *dependencies* are the names of files (often names of other targets, as well) on which the construction of the target depends. A particular target does not necessarily have to have dependencies. The *commands* are the actual commands to be executed once all the dependencies are complete. Note that a `<TAB>` must be used to indent commands under the target name; if you use spaces or don't indent you'll get a (not entirely clear) error message. `Makefiles` may also contain variable definitions to make the file perhaps simpler.

6.2 PSI `Makefiles`

The `Makefiles` contained in the PSI package are complicated, in part due to the size of the package and the need for code portability. `PSI3 Makefiles` are generated automatically from simple input, called `Makefile.in`, by the `configure` script in the top-level $PSI directory. This script is designed to examine system-specific characterisctics, such as library locations, special compiler options, the existence of certain header files or functions, or Fortran-C cross-linkage conventions, among others. With the information it obtains, it constructs the large number of `Makefiles` needed for compilation of PSI's libraries, utilities, and modules.

As an example, consider the `Makefile.in` file associated with `cscf`:

```
srcdir = @srcdir@
VPATH = @srcdir@

include ../MakeVars

PSILIBS = -lPSI_file30 -lPSI_chkpt -lPSI_iwl -lPSI_psio -lPSI_ciomr -lPSI_ipv1

TRUESRC = \
cscf.c cleanup.c dft_inputs.c diis.c dmat.c \
dmat_2.c ecalc.c errchk.c findit.c \
formg2.c formgc.c formgo.c form_vec.c gprgid.c init_scf.c \
packit_c.c packit_o.c rdone.c rdtwo.c rotate_vector.c scf_input.c \
scf_iter.c scf_iter_2.c schmit.c sdot.c shalf.c check_rot.c phases.c\
guess.c sortev.c occ_fun.c init_uhf.c cmatsplit.c dmatuhf.c \
findit_uhf.c uhf_iter.c schmit_uhf.c diis2_uhf.c formg_direct.c \
```

```
orb_mix.c

BINOBJ = $(TRUESRC:%.c=%.o)
ALLOC =

include ../MakeRules

ifneq ($(DODEPEND),no)
$(BINOBJ:%.o=%.d): $(DEPENDINCLUDE)
include $(BINOBJ:%.o=%.d)
endif

install_man:: cscf.1
        $(MKDIRS) $(mandir)/man1
       $(INSTALL_INCLUDE) $^ $(mandir)/man1
```

The `@string@` directives tell the `configure` script where to insert certain variables is has determined from the system. This `Makefile` input also includes two external `Makefiles`, `MakeVars` and `MakeRules`, both of which are in the parent directory. These files contain (not surprisingly) numerous necessary variables (e.g. the local C compiler name) and rules (e.g. how to generate the module itself) for compilation and installation of `cscf`. Similar files exist for the PSI libraries as well. We recommend that programmer's spend some time studying the PSI `Makefile` structure.

6.3 Preparing to Develop New `PSI3` Code

Given the complexity of the `PSI3` package, the prospect of adding new modules or libraries may seem daunting at first. Let's assume you want to begin writing a new module named `great_code` for `PSI3`. The following series of steps will generate the proper directories and `Makefiles` to get started. For convenience, the top-level directory of the programmer's `PSI3` source tree will be referred to as $PSI3 and the top-level directory of the compilation area as $prefix:

1. Generate the new directory in the source tree:
   `mkdir $PSI3/src/bin/great_code`
2. `cd $PSI3/src/bin/great_code`
3. Copy an existing `Makefile.in` from another module:
   `cp ../cscf/Makefile.in .`
4. Edit the `Makefile.in` so that it lists only the source files for `great_code` and includes in `PSILIBS` only those libraries needed to link the executable.
5. Return to the top of the source tree: `cd $PSI3`

6. Add the name of great_code's Makefile to configure.in (near the bottom of the file) and run autoconf to generate a new configure script.

7. Go to the top of the compilation tree: cd $prefix.

8. Re-run the configure script to generate the Makefile for great_code. Make sure you use the same options to configure that you used before or other Makefile's may not function properly. The command you used before can be found in $prefix/config.status. (See also the PSI3 installation manual for more details on the options to configure.)

Now you are ready to work on the code. Changes to source files (including the Makefile should be made to the files in $PSI3/src/bin/great_code and all compilations should be run in $prefix/src/bin/great_code.

7 Code Debugging

Debugging PSI3 code using an interactive debugger, such as gdb or dbx can be difficult at times because of the complicated organization of this large program package. This section discusses some strategies and technical details of using such debuggers with the PSI3 code.

7.1 Code Re-compilation

Any section of PSI3 code that needs to be debugged must first be re-compiled with the "-g" flag turned on. This flag is set in the MakeVars file in the directory above each module or library's source code directory. For example, to turn on debugging in the cscf program, one would first clean the existing object code out of the $prefix/src/bin/cscf directory using make clean. Then edit $prefix/src/bin/MakeVars, one directory above the cscf source code: set CDBG = -g and, optionally COPT = to turn off optimization flags. (For modules using C++, the analogous variables are CXXDBG and CXXOPT. Then re-compile the module. If debugging information is needed for a library routine as well, then follow this same procedure for the library in question. Technically, only the routines of interest need to be re-compiled, though it is frequently more convenient to simply re-compile the entire library or module.

7.2 Multiple Source Code Directories

The most difficult problem of debugging PSI3 code is that object code and source code generally reside in separate directories to allow storage of objects for several achitectures simultaneously. In addition, library codes are kept separate from binary (module) codes. If the code is compliled with gcc/g++, then this separation of source and object code is of no consequence because the compiler builds the full path to the source file directly into the object code. However, for non-gcc compilations, one must know how to tell the debugger where to find the sources.

Most interactive debuggers allow the programmer to specify multiple source code search directories using simple command-line options. For example, if one were debugging the `cscf` program and needed access to the `libciomr.a` library source code in addition to that of `cscf`, one could use `gdb`'s "dir" command to search several source code directories:

```
dir \$PSI/src/lib/libciomr
```

Additionally, such commands can be placed in the user's `$HOME/.gdbinit` file. In `dbx`, the "use" command specifies multiple source directories.

8 Documentation

Documentation is often the only link between code's author and code's users. The usefulness of the code will depend heavily on the quality of its documentation. One great failing of most of the PSI code is that it contains little to no documentation. We strongly advocate documentation of three types:

1. A short description of the code's function and keywords *must* be written for each new module and library added to the `PSI3` package. There is no convention yet what should be the preferred medium for such a description, but the following are common:

   - A UNIX `man` page — all old and some newer `PSI3` codes use `man` pages as the medium of choice. To access the `PSI3` `man` pages, you will need to add the `man` directory to your `MANPATH`. For example, if you run csh or tcsh, and assuming `PSI3` has been installed in `/usr/local/psi3-bin`, the following can be added to your `.cshrc` or `.tcshrc` file:

     ```
     setenv MANPATH /usr/local/psi3-bin/doc/man:/usr/share/man
     ```

     The usual man path should be added after the `PSI3` part and will be different for different systems. Different directories are separated by colons.
   - HTML-based documentation — this is a much more flexible medium than `man` pages and is accessible by anyone anywhere in the world. Although HTML has its own drawbacks (e.g. the separation of the form and the function is not always enforced, and it does not allow tags to be customized), it is pretty safe to assume that HTML will remain the dominant means of distributing information. Hence we encourage `PSI3` contributors to write documentation in HTML format. Documentation for `cints` and `libpsio.a` can be used for examples.
   - Direct inclusion in the `PSI3` manuals — binaries (modules) should be included in the user's manual and libraries in the programmer's manual.

2. Second, as mentioned before, the source code should be directly documented by comment lines in the code.

3. A *complete manual* should be written for all finished programs, describing all input options, explaining how the program works (theory and technical details), and providing solutions to common problems encountered with the program. Sometimes, the latter documentation is included in the man page: for a good example, see the man page for `intder95`. Alternatively, a separate document can be created; for another example, see the documentation of `fcmgen`.

9 Creating New Test Cases

The `PSI3` test suite is designed to maximize code reuse and provide testing in $prefix before the `PSI3` executables have been installed. The configure script in $PSI3 will take all the necessary files in $PSI3/tests with the .in stub: Makefile.in, MakeRules.in, MakeVars.in, and runtest.pl.in, replace variables with system specific parameters, and copy/create the testing files and directories in $prefix/tests. The tests should be run in the object directory before installation.

If you have just added a new module for performing, say multireference coupled cluster, and you would like to add a test case to the current test suite, here is what you should do.

1. Copy one of the existing test case directories to an appropriately named directory for the new test case.

2. Create an appropriate input file for running the new module. Then, if your program produced the correct data, rename the output files to *.ref. Follow the convention of the existing test cases.

3. If the test case is small, add the directory name to the list in $PSI3/tests/Makefile.in. If the test is particularly tricky, see the psi_start or rhf-stab test cases as an example.

4. All the testing functionality is located in the perl library `runtest.pl.in`. If you are testing for a quantity that is not searched for currently, then add a function to the library following the format of the functions already available. If you have added functionality to the `PSI3` driver, make sure to update the appropriate functions in `runtest.pl.in`.

5. Add the location of the Makefile for the new test case to the configure script in $PSI3.

Please contact one of the authors of `PSI3` before making any major changes or if you have a problem adding a new test case. Remember, if all else fails, read the source code.

10 Special Considerations

The following is a list of special items that should be kept in mind while developing PSI code.

***Malloc() calls on IBM*:** The current IBM compilers (Visual Age C/C++ 5) do not properly prototype `malloc()` unless one includes `stdlib.h`. Please make sure that you `#include <stdlib.h>` anytime you call `malloc()` in a file. If you forget, it will still work in gcc but not on an IBM.

A PSI3 Reference

T. Daniel Crawford, C. David Sherrill, Edward F. Valeev, Justin T. Fermann, Rollin A. King, Matthew L. Leininger, Shawn T. Brown, Curtis L. Janssen, Edward T. Seidl, Joseph P. Kenny, and Wesley D. Allen, PSI 3.2, 2003.

B Text Files in PSI3

Psi uses several text files to store certain types of information. Storing information in text files makes it much easier for users to inspect and manipulate that information, provided that the user understands the format of that file. In the following file format descriptions, I will use the notation x_i, y_i, and z_i to denote the x, y and z coordinates of nucleus i, respectively, η_i will denote the i^{th} internal coordinate, and E will denote the sum of the electronic energy and nuclear repulsion energy.

`geom.dat`

A vectorized format which is appropriate for the routines in libipv1 or iomr is employed in `geom.dat` Generally, the first line of `geom.dat` is

`%%`

Though this does not affect the parsing routines in libipv1, or any of the common programs which read `geom.dat` (i.e. `rgeom` or `ugeom`), some `PSI2` modules (`bmat`, etc.) expected this line and would muddle up `geom.dat` if it is not present. `geom.dat` will frequently have several entries, with the topmost being the most recent addition by bmat.

format: n = number of atoms.

$$\begin{array}{llll} \texttt{geometry} = \texttt{(} & & & \\ \texttt{(} & x_1 & y_1 & z_1 \quad \texttt{)} \\ \texttt{(} & x_2 & y_2 & z_2 \quad \texttt{)} \\ & \vdots & \vdots & \vdots \\ \texttt{(} & x_n & y_n & z_n \quad \texttt{)} \\ \texttt{)} & & & \end{array} \tag{1}$$

Other geometries of the same format may follow.

`fconst.dat`

This file contains the force constant matrix produced by `optking` or `intder95`. Because the force constant matrix is symmetric, only the lower diagonal is stored here. The force constant matrix may be represented in either cartesian or internal coordinates, depending upon what flags were used when `intder95` was run to produce `fconst.dat`. `optking` is

the program which uses `fconst.dat` most frequently, and it assumes that the force constant matrix will be in terms of the internal coordinates as defined in `input.dat` or `intco.dat`. For this reason, it is best to have `intder95` produce a `fconst.dat` in internal coordinates. The order of internal coordinates is determined by the order set up in `input.dat` or `intco.dat`. The totally symmetric coordinates always come first, followed by all asymmetric coordinates.

In the following format, f_{η_i} is the force constant for internal coordinate η_i and f_{η_i,η_j} is the force constant for the mixed displacement of internal coordinates i and j.

format: n = total number of internal coordinates in `intco.dat` or `input.dat`.

$$\begin{array}{lllll} f_{\eta_1} & & & & \\ f_{\eta_2,\eta_1} & f_{\eta_2} & & & \\ f_{\eta_3,\eta_1} & f_{\eta_3,\eta_2} & f_{\eta_3} & & \\ \vdots & \vdots & \vdots & & \\ f_{\eta_n,\eta_1} & f_{\eta_n,\eta_2} & f_{\eta_n,\eta_3} & \cdots & f_{\eta_n} \end{array} \tag{2}$$

If the force constant matrix is stored in cartesian coordinates, however, the format, using a similar notation, with n now equal to the total number of atoms, is as follows:

$$\begin{array}{llllll} f_{x_1} & & & & & \\ f_{y_1,x_1} & f_{y_1} & & & & \\ f_{z_1,x_1} & f_{z_1,y_1} & f_{z_1} & & & \\ f_{x_2,x_1} & f_{x_2,y_1} & f_{x_2,z_1} & f_{x_2} & & \\ \vdots & \vdots & \vdots & \vdots & & \\ f_{z_n,x_1} & f_{z_n,y_1} & f_{z_n,z_1} & f_{z_n,x_2} & \cdots & f_{z_n} \end{array} \tag{3}$$

`file11.dat`

The number of atoms (n), total energy as predicted by the final wavefunction, cartesian geometry, cartesian gradients, atomic charges (Z_i) and a label are all contained in `file11`. The exact nature of the label depends upon the type of wavefunction for which the gradient was calculated. The first part of the label is determined by the label keyword in input.dat. If an SCF gradient is run, then the calculation type (*calctype*), and derivative type (*dertype*) will also appear. If a correlated gradient has been run, *calctype* [CI, CCSD, or CCSD(T)] and derivative type (FIRST) appear. `file11` will frequently have several entries, with the last entry being the latest addition by `cints --deriv1`.

format:

$$
\begin{array}{llll}
\textit{label} & \textit{calctype} & \textit{dertype} & \\
n \quad E & & & \\
Z_1 & x_1 & y_1 & z_1 \\
Z_2 & x_2 & y_2 & z_2 \\
\vdots & \vdots & \vdots & \vdots \\
Z_n & x_n & y_n & z_n \\
 & \frac{\delta E}{\delta x_1} & \frac{\delta E}{\delta y_1} & \frac{\delta E}{\delta z_1} \\
 & \frac{\delta E}{\delta x_2} & \frac{\delta E}{\delta y_2} & \frac{\delta E}{\delta z_2} \\
 & \vdots & \vdots & \vdots \\
 & \frac{\delta E}{\delta x_n} & \frac{\delta E}{\delta y_n} & \frac{\delta E}{\delta z_n}
\end{array}
\tag{4}
$$

`file12.dat`

Internal coordinate values and gradients, the number of atoms (n), and the total energy (E) may be found in `file12`. `file12` is produced by `intder95`, which can convert cartesian gradients into internal gradients. Generally, `file12` will have several entries, with each entry corresponding to an entry in the `file11` of interest.

format:

$$
\begin{array}{ll}
n & E \\
\eta_1 & \frac{\delta E}{\delta \eta_1} \\
\eta_2 & \frac{\delta E}{\delta \eta_2} \\
\vdots & \vdots \\
\eta_n & \frac{\delta E}{\delta \eta_n}
\end{array}
\tag{5}
$$

`file12a.dat`

In order to calculate second derivatives from gradients taken at geometries finitely displaced from a particular geometry, `intdif` requires a `file12a`. This file contains essentially the same information as `file12`, but each entry also has information concerning which internal coordinate (*numintco*) was displaced in the gradient calculation and by how much (*disp*) it was displaced.

format:

$$
\begin{array}{lll}
\textit{numintco} & \textit{disp} & E \\
\eta_1 & & \frac{\delta E}{\delta \eta_1} \\
\eta_2 & & \frac{\delta E}{\delta \eta_2} \\
\vdots & & \vdots \\
\eta_n & & \frac{\delta E}{\delta \eta_n}
\end{array}
\tag{6}
$$

`file15.dat`

The cartesian Hessian matrix is found in `file15`. The first line of this file gives the number of atoms (n) and, in case you are curious, six times the number of atoms (*sixtimesn*).

format:

$$
\begin{array}{lll}
n \quad sixtimesn & & \\
\frac{\delta^2 E}{\delta^2 x_1} & \frac{\delta^2 E}{\delta x_1 \delta y_1} & \frac{\delta^2 E}{\delta x_1 \delta z_1} \\
\frac{\delta^2 E}{\delta x_1 \delta x_2} & \frac{\delta^2 E}{\delta x_1 \delta y_2} & \frac{\delta^2 E}{\delta x_1 \delta z_2} \\
\vdots & \vdots & \vdots \\
\frac{\delta^2 E}{\delta z_1 \delta x_n} & \frac{\delta^2 E}{\delta z_1 \delta y_n} & \frac{\delta^2 E}{\delta z_1 \delta z_n} \\
\frac{\delta^2 E}{\delta x_2 \delta x_1} & \frac{\delta^2 E}{\delta x_2 \delta y_1} & \frac{\delta^2 E}{\delta x_2 \delta z_1} \\
\vdots & \vdots & \vdots \\
\frac{\delta^2 E}{\delta z_n \delta x_n} & \frac{\delta^2 E}{\delta z_n \delta y_n} & \frac{\delta^2 E}{\delta^2 z_n}
\end{array}
\tag{7}
$$

`file16.dat`

The second derivatives of the total energy with respect to the internal coordinates are found in `file16`. As in `file15`, the number of atoms (n) and six times that number (*sixtimesn*) are given.

format:

$$
\begin{array}{lll}
n \quad sixtimesn & & \\
\frac{\delta^2 E}{\delta^2 \eta_1} & \frac{\delta^2 E}{\delta \eta_1 \delta \eta_2} & \frac{\delta^2 E}{\delta \eta_1 \delta \eta_3} \\
\vdots & \vdots & \vdots \\
\frac{\delta^2 E}{\delta \eta_1 \delta \eta_{n-2}} & \frac{\delta^2 E}{\delta \eta_1 \delta \eta_{n-1}} & \frac{\delta^2 E}{\delta \eta_1 \delta \eta_n} \\
\vdots & \vdots & \vdots \\
\frac{\delta^2 E}{\delta \eta_n \delta \eta_{n-2}} & \frac{\delta^2 E}{\delta \eta_n \delta \eta_{n-1}} & \frac{\delta^2 E}{\delta^2 \eta_n}
\end{array}
\tag{8}
$$

`file17.dat`

First derivatives of the cartesian dipole moments (μ_x, μ_y, μ_z) with respect to the cartesian nuclear coordinates may be found in `file17`. The first line and subsequent format are similar to that of `file15`.

format:

$$
\begin{array}{lll}
n \quad threetimesn & & \\
\frac{\delta \mu_x}{\delta x_1} & \frac{\delta \mu_x}{\delta y_1} & \frac{\delta \mu_x}{\delta z_1} \\
\frac{\delta \mu_x}{\delta x_2} & \frac{\delta \mu_x}{\delta y_2} & \frac{\delta \mu_x}{\delta z_2} \\
\vdots & \vdots & \vdots \\
\frac{\delta \mu_x}{\delta x_n} & \frac{\delta \mu_x}{\delta y_n} & \frac{\delta \mu_x}{\delta z_n} \\
\frac{\delta \mu_y}{\delta x_1} & \frac{\delta \mu_y}{\delta y_1} & \frac{\delta \mu_y}{\delta z_1} \\
\vdots & \vdots & \vdots \\
\frac{\delta \mu_z}{\delta x_n} & \frac{\delta \mu_z}{\delta y_n} & \frac{\delta \mu_z}{\delta z_n}
\end{array}
\tag{9}
$$

`file18.dat`

First derivatives of the cartesian dipole moments (μ_x, μ_y, μ_z) with respect to the internal nuclear coordinates may be found in `file18`.

format:

$$
\begin{array}{lccc}
n & threetimesn & & \\
\frac{\delta\mu_x}{\delta\eta_1} & \frac{\delta\mu_x}{\delta\eta_2} & \frac{\delta\mu_x}{\delta\eta_3} & \\
\frac{\delta\mu_x}{\delta\eta_4} & \frac{\delta\mu_x}{\delta\eta_5} & \frac{\delta\mu_x}{\delta\eta_6} & \\
\vdots & \vdots & \vdots & \\
\frac{\delta\mu_z}{\delta\eta_{n-2}} & \frac{\delta\mu_z}{\delta\eta_{n-1}} & \frac{\delta\mu_z}{\delta\eta_n} &
\end{array}
\tag{10}
$$