Parallel Architecture Research Laboratory
Developer Guidelines

PVFS Development Team

Feb 12, 2001

Contents

1 TODO **3**

2 Introduction **3**

3 Editors **3**
3.1 Vi . . . 3
3.1.1 Vi variants . . . 3
3.1.2 Syntax highlighting . . . 4
3.2 Emacs . . . 4

4 CVS tutorial **5**
4.1 Overview . . . 5
4.2 Basic user commands . . . 5
4.3 Starting a new project . . . 7
4.4 Remote access . . . 7

5 Compiler flags **7**

6 Debugging tutorial **8**
6.1 Starting gdb . . . 8
6.2 Common gdb commands . . . 9

7 Makefile tutorial **10**

8 Electric Fence tutorial **10**
8.1 Using Electric Fence . . . 10
8.2 Electric Fence options . . . 10

9 C Programming **11**
9.1 Formatting . . . 11
9.1.1 GPL . . . 11
9.1.2 Other source code header information . . . 11
9.1.3 Commenting . . . 11
9.1.4 Brackets . . . 12
9.1.5 Indentation . . . 12

9.2 Hints for writing maintainable code . . . 13
9.2.1 General code layout . . . 13
9.2.2 Length of functions . . . 13
9.2.3 Preventing double inclusion . . . 13
9.2.4 Static declarations . . . 14
9.2.5 Initializing variables . . . 14
9.2.6 Allocating and deallocating complex structures . . . 15
9.2.7 Keeping up with work in progress . . . 15
9.2.8 Choosing good variable and function names . . . 16
9.3 Advanced topics . . . 16
9.3.1 Checking for interrupted system calls . . . 16
9.3.2 Constant arguments . . . 16
9.3.3 Obscure coding practices . . . 16
9.3.4 Locking data structures . . . 17
9.3.5 Select vs. poll . . . 17
9.3.6 String parsing . . . 17
9.3.7 Abstraction . . . 17
9.3.8 Function pointers . . . 18
9.3.9 Typedefs and opaque types . . . 18

10 Specific PVFS issues **19**
10.1 Copyright information . . . 19
10.2 Function commenting . . . 19
10.3 Function naming . . . 19
10.4 Error logging with Gossip . . . 20
10.5 Suggested error handling . . . 21
10.5.1 Traditional application error handling with errno . . . 21

1 TODO

- reorganize this to be PVFS-centric
- document the build system

2 Introduction

This document is intended to serve as an introduction and set of guidelines for programming style and development tools that are commonly used in the Parallel Architecture Research Lab at Clemson University. We do not claim that this is the best or only way to effectively structure code or use development tools. However, adhering to these guidelines should assist in the maintenance, debugging, and documentation of code that must be consistent over the lifetime of a large software engineering project. Most of this documentation was motivated by the PVFS project, but hopefully it is applicable to other projects as well.

The coding manual is intended for use by anyone who will be regularly contributing development effort to software projects within the Parallel Architecture Research Lab.

The first portion of this document provides a brief introduction to some of the most common development tools used in the lab. All of these tools are available on the lab workstations. They are also available on most other UNIX-like platforms.

The second portion of this document covers formatting and writing standard C code. Some of these guidelines are provided to maintain consistency in the appearance of the code, while others actually assist in writing "correct" code or making reasonable design decisions while coding. The third section provides the Java counterpart to these guidelines.

The final section contains information that is very specific to PVFS and may not be applicable to other projects.

3 Editors

In the PARL lab we advocate the use of either `vi` or `emacs`. These are the most common editors used for writing software and are therefore the most likely to be available on any given development platform. It is also useful to have everyone on a particular project use the same set of editors so that it is easier to maintain consistent formatting.

3.1 Vi

The general use of vi is beyond the scope of this document, but you can find out enough to get started by either asking a colleague or starting vi and typing ":help". There is also a tutorial available on many systems that can be started by typing "vimtutor" at the command line.

3.1.1 Vi variants

There are a few variations of the basic vi editor that support different features. The most popular is `vim`, or "Vi IMproved". vim adds several important features to vi, including multilevel undo, visual selection, and multiple buffers. It also is fully compatible with the original vi editor. Some machines utilize a version of vim that also contains optional context highlighting, while others provide a separate binary with this feature that is called `vimx`.

This may seem a little confusing, but it is actually something you don't have to worry about if you just setup your environment to use the version that you prefer on any given system. The easiest way to do this is to add a conditional statement to your login configuration that aliases the "vi" command to start whichever version is available. This example illustrates how to do this if you use the tcsh shell. Add the following lines to your ~/.cshrc file:

```
if ( -X vimx ) then
   alias vi vimx
else if ( -X vim ) then
   alias vi vim
endif
```

The next time you login to the system, it will alias the vi command to either vimx of vim if they are available. Otherwise the vi command will simply start the original vi editor.

3.1.2 Syntax highlighting

Syntax highlighting is an editor feature that uses various colors to notate different parts of the syntax (dependent upon which language you are writing in). Vim has rule sets for several languages ranging from c to latex. This can be extremely useful when trying to quickly read code. It is also helpful in catching a few minor coding errors. You can control these options for vi by way of a file in your home directory name ".vimrc". This file can also be used to control other settings in vi, as an alternative to the environment variable used in section **??**. The following is an example of some color settings for vi taken from a .vimrc file:

```
set background=dark
if has("syntax")
   syntax on
   hi! Comment ctermfg=darkgreen
   hi Type NONE
   hi Structure NONE
   hi! Operator NONE
   hi! Include ctermfg=darkblue
   hi! PreCondit ctermfg=darkcyan
   hi! cIncluded ctermfg=darkblue
   hi! Statement ctermfg=brown
   hi! Conditional ctermfg=brown
   hi! Todo ctermfg=yellow
   hi! Operator ctermfg=NONE
   hi! Constant ctermfg=NONE
   hi! cCppOut ctermfg=darkred
   hi! cSpecial ctermfg=darkmagenta
endif
```

3.2 Emacs

I have no idea.

4 CVS tutorial

CVS is a network aware version control system. Several of the larger projects in the PARL lab use CVS to manage source code. These are some of the basic capabilities that it provides:

- automatically tracks changes to code so that old versions are backed up
- allows you to document incremental changes and browse this documentation
- multiple users can make changes simultaneously
- keeps multiple copies synchronized between different users
- allows you to work remotely

You can find out more information about CVS at http://www.cvshome.org.

4.1 Overview

CVS stores revision history for each file in your project. Each time you "commit" a file to CVS, a snapshot of its state at that time is stored in CVS. This allows you to back up to old versions at any time, or browse old versions to see how the code evolved.

It is important to note that CVS stores version information independently for each file. There is no global verion number associated with your project at any time unless you manually assign it (see the "tag" feature, which is beyond the scope of this document). Version numbers for each file revision are assigned automatically by CVS; there is no way to control this numbering.

If you wish to assign a version number or logical name to the state of the entire project at one time (for example, release version 1.0), then you must do this manually.

4.2 Basic user commands

Before using CVS, you must set an environment variable that tells it where to look for the CVS repositories. If you are using tcsh, you can do something like this (and add it to your .cshrc file):

```
setenv CVSROOT /projects/cvsroot
```

and for bash:

```
set CVSROOT=/projects/cvsroot
```

Hmm. Should we maybe have a sample project in cvs that anyone can check out? Might be nice to be able to play with this stuff without hurting a real project

- **export:** If you wish to get a copy of a source tree with no intention of modifying it or manipulating it with CVS, you can just export it. This command creates a directory for the project and populates it with the source code. To export the most recent copy of a project:

  ```
  cvs export -D today <projectname>
  ```

  (where <projectname> is the name of the project). When you are done with the code you can simply delete the directory.

- **check out:** The check out command is used to obtain a copy of the source code that will be tracked by CVS. This will let you make changes and additions to the project. You must have appropriate permissions in order to perform this operation:

  ```
  cvs co <projectname>
  ```

  This will create a directory for the project that contains source code and CVS information. See the "commit" command for information on submitting modifications, and the "release" command to get rid of the directory when you are done.

  The checkout command can be run repeatedly without any harmful side effects. This is useful for updating your copy of a project to make sure that it matches the most recent modifications before adding any modifications of your own.

- **release:** When you are done with a project, you may release the directory. This will warn you if you have made any modifications to the source code that have not been committed to CVS. The options listed below will also cause your copy of the project directory to be deleted. Note that it is perfectly fine to leave code checked out for long periods of time if you are going to be working on it regularly.

  ```
  cvs release -d <projectname>
  ```

- **commit:** Once you have made any changes to the project, you must perform a commit operation to record the changes in CVS and make them available to other users. It is usually advisable to only commit code that compiles and works correctly, unless you are the only person working on the project. Otherwise, you may interfere with someone else's work. This command should be carried out within the project directory:

  ```
  cvs commit
  ```

  When you do this, CVS will open up a vi session that allows you to write a brief summary of the changes that you have made. Note that this summary will be associated with all of the files that you have changed since the last checkin, not just one. The commit will complete when you close the vi session.

- **add:** The add command is used to add new files to the project. You must create the new file first. Then run the following command from within the project directory that contains the new file:

  ```
  cvs add <newfilename>
  ```

  You must follow this up with a "commit" in order for other users to see the new file. This command may also be used to add new directories to a project. Add directories with caution, however. Unlike files, subdirectories are very difficult to remove from a CVS project.

- **update:** You will often find it necessary to update your copy of the project to reflect the most recent changes. Rather than use the update command, however, it is often better to simply run the checkout command again. It will detect if your project is already checked out and will update if needed.

- **status:** The status command will tell you the status of all of the files within a particular project directory. This is useful for determining if your copy is up to date or if it has been modified but not checked in:

  ```
  cvs status
  ```

  If you wish to filter the output from this command so that it only shows you files that are not up to date you may do the following (you can make this an alias or script if you wish to use this regularly):

  ```
  cvs status |& grep Status: | grep -v Up-to-date
  ```

There are many other CVS commands and features, but the ones listed above should be enough to get you started. You can find out more information in the CVS man pages or by reading documentation at http://www.cvshome.org.

4.3 Starting a new project

Lab users can also create new CVS directories to keep up with software projects. These can later be merged in as part of another project if desired. To create a new CVS entry, go into the source directory and delete all of the binary or object files (assuming that you only wish to track the source code). Then run this command:

```
cvs import -m "Imported sources" <projectname> PARL start
```

(for the sake of clarity, <projectname> should probably match the name of the directory that contains the source)

4.4 Remote access

You can also access the PARL CVS repository remotely (outside of the lab) if you have an internet connection and the CVS and ssh programs installed on your remote machine. It works exactly like using CVS locally. You can use the same commands listed above, except substitute the following for the the word "cvs" in the command lines:

```
cvs -d :ext:<userid>@cvs.parl.clemson.edu:/projects/cvsroot
```

(where userid is your lab user login id) You may wish to create an alias or script to do this for convenience. CVS will prompt you for a password in this situation.

5 Compiler flags

Gcc is the standard c compiler used for PARL software. The general use of gcc is beyond the scope of this document, but there are a few guidelines that should be followed when building software with gcc:

- Always use the -Wall command line flag to gcc. This turns on most of the warnings that gcc is capable of generating at compile time. The warnings tend to point out bad coding habits and ambiguous statements. Use the -Wall option from the very beginning of your project- it is often overwhelming to try to apply it after the code base has gotten large because of the sheer number of warnings that it will probably find at that point.
- Always use the -Wstrict-prototypes command line flag to gcc. This turns on additional warnings that enforce the use of proper prototypes for all functions. See section 9.2.1 for more information.
- Make use of the -g option to gcc in development code. This enables debugging symbols for use with gdb (section 6). When code is released to the public, it may be the case that this option will be removed in order to reduce binary size or increase optimization, but it is invaluable during the development cycle.

6 Debugging tutorial

Gdb is a debugger for c and c++ programs. I lets you control the execution of a program, see what line of source code is being executed at any given time, and inspect data structures while the program is running. There are several other debuggers or interfaces built on top of gdb, but gdb is still one of the most popular and most flexible.

6.1 Starting gdb

In order to use gdb, the code in question must have been compiled with the debugging option turned on. For the gcc compiler (see section 5), this just means using the -g option:

```
gcc -g test.c -o test
```

In order to start debugging, just launch gdb with with the name of the program you wish to debug as the only argument:

```
gdb test
```

If you start gdb by this mechanism, then gdb will present you with a prompt at which you can enter commands to gdb. At this point, the program you wish to debug has not been started and must be launched with the `run` command outlined below.

Alternatively, you can attach to a process that is already running if you know it's pid. In this case, the command line arguments to gdb are the program name followed by the pid of the running program.

```
> ./test &
> ps | grep test
23716 ttyp4    00:00:00 test
> gdb test 23716
```

In this scenario, gdb will still present you with a prompt for entering gdb commands. However, the program that you are debugging will be in the middle of execution but will have stopped running. In order to make it continue where it left off you must use the `continue` command outlined below.

6.2 Common gdb commands

This is a short list of the most common commands that you may wish to use with gdb. You can find out more specific information by using the `help` command or by looking at http://sources.redhat.com/gdb/#docum

- **help**: Typing just help at the prompt will give you a list of command classes that you can find out more information about. You can also get documentation for a particular command if you know its name.
- **break**: Break is used for specifying where you would like to pause execution of a program. You can either specify a function name or a line number with break. The next time you `run` or `continue` your program, it will stop at this breakpoint and wait for another gdb command. Once it stops you can either step through execution one line at a time, look at variables, or continue again. This is useful for debugging a particular area of interest in your program without having to step through the entire program. You can also specify multiple break points if you would like the execution to stop at more than one location.
- **run**: This command begins execution of your program. If you have not specified any breakpoints, the program will run until it exits normally or an error occurs. If you do specify a breakpoint, it will also stop when it hits that breakpoint.
- **continue**: Continue causes execution to resume if you are currently at a breakpoint or have been stepping through execution one line at a time. Continue will let program execution progress until it hits either an error, normal exit, or critical error.
- **step**: Step lets you execute one line of source code at a time. It will tell you what line it is on, as well as what line of code is about to be executed. This allows you to examine the effects of particular statements in your code.
- **next**: Next is very similar to step, except that it treats subroutine calls as a single instruction rather than stepping into all subroutines. This is very helpful for skipping over functions that you do not wish to inspect the internals of (such as `printf` for example). It is helpful to be able to skip over complex functions that are either known to work or were not compiled with debugging symbols.
- **print**: The print command is the primary mechanism for inspecting data structures and variables. Print can be used to show the contents of a single variable by using the variable name as the argument. It will also attempt to print out a comma separated list of the elements of a structure if you try to print a structure. Print is aware of a c like syntax for specifying struct elements and pointers so that you can specify elements and variables using struct.element, struct->element and *pointer notation. Parenthesis are often required in situations where they are not in c notation, however.
- **where**: This command shows your current location in the execution stack. It prints the line number of the current instruction, as well as the heirarchy of subroutine line numbers that were passed through to get to this point.
- **list**: List is used to list the source code surrounding a particular instruction. By default, you it lists the 10 lines surrounding the current instruction. You can also use it to list source code surrounding a particular line number or function definition.

7 Makefile tutorial

We may be able to get this from the MPICH coding docs or from Walt's web page?

8 Electric Fence tutorial

Electric Fence is a tool used for debugging buffer overruns and underruns that can occur when manipulating dynamically allocated memory. It is implemented as a static library that can be linked into your code without modifying the source in any way. It basically works by replacing the malloc system call with a modified malloc that surrounds any new memory regions with protected areas. If a process attempts to write into such a protected area, it will cause a segmentation violation. Without Electric Fence, buffer overruns can occur without being immediately obvious, which makes debugging difficult.

Take note that Electric Fence does not help at all with problems that occur with staticly allocated memory. It also does not indicate memory leaks. Other tools should be used for debugging those types of problems.

8.1 Using Electric Fence

To use Electric Fence, you just need to link in the efence library during the last stage of linking (or compilation, if you do not have a separate link step):

```
gcc -g -Wall -Wstrict-prototypes -lefence test.c
```

When you run your program, it should print a message to the screen indicating that Electric Fence is in use. If your program segfaults, it will not show you where it occurred, but you can then debug the program with gdb to determine this information (section 6).

8.2 Electric Fence options

There are several helpful Electric Fence options that can be controlled by way of environment variables. The following list summarizes the most useful ones. They can be turned on in tcsh by typing "setenv VARIABLE value" or in bash by typing "export VARIABLE=value", where VARIABLE is the option you wish to control and value is the value that you wish to set it to.

- EF_ALIGNMENT: This controls the allignment of dynamically allocated memory. By default, this alignment is equal to your machines's word size. This means that small overruns might go unnoticed because extra memory has been allocated for certain buffers. To make sure that this does not happen, set alignment to 1. This ensures that even the small overruns will be caught.
- EF_PROTECT_BELOW: When this option is set to 1, it tells Electric Fence to check for buffer underruns in addtion to buffer overruns.
- EF_PROTECT_FREE: When this option is set to 1, Electric Fence will check to be sure that memory is not being accessed after it has been released with the `free()` system call.

Turning on all of these options is helpful in debugging dynamic memory problems. Note that using Electric Fence (especially with the stricter options) will cause the memory utilization of your application to increase dramatically.

9 C Programming

9.1 Formatting

This section will provide guidelines so that multiple users on a given project can write code with consistent appearance. This makes the code easier to maintain and audit in a group environment.

9.1.1 GPL

The GPL, or General Public License, is a software license created by the GNU organization (http://www.gnu.org). You can find out more information about it at their web site. A brief summary is that the GPL insists that the source code be distributed with any software released under the GPL. Furthermore, anyone who modifies and redistributes GPL code must release their modifications under the GPL as well. This is convenient for the research community because it encourages the sharing of ideas and also legally protects your developments.

Any PARL project code that you release to the community should include an electronic copy of the GPL. The easiest way to do this is to create a file in the project's top level directory called "COPYING" which contains the full text of the GPL version 2 as obtained from http://www.gnu.org. Then in *every* source code file in the project, include the following text at the very top of the file:

```
/*
 * (C) 2001 Clemson University.
 *
 * See COPYING in top-level directory.
 */
```

Other organizations that have contributed to the project may be listed in the copyright line as well (See section 10.1 for information on how do this in PVFS code). If you wish to credit particular developers or provide contact information, please do so in the README file located in the top level directory.

9.1.2 Other source code header information

In addition to the copyright comments, it is usually helpful to provide a brief description of what is contained in the each source file. This should just be a few summary lines below the copyright information.

9.1.3 Commenting

Commenting your code effectively is very important! Please comment important sections of your code clearly and concisely as you write it. The habit of commenting after completing the code often leads to poor comments.

Do not use c++ style comment delimiters (//) in c code. Some c compilers do not accept this as a comment delimiter, and it is not a part of the c language specification.

For single line comments (or brief comments trailing a line of code), just use the /* and */ delimiters. If the comment is longer than one line, use this format:

```
/* This code does lots of cool things.  It is also written perfectly and
 * will never break.  It is fast, robust, extensible, and resistant to
 * rust and corrosion.
 */
```

This makes it easy to tell where the comment begins and ends.

Comments that describe the operation of a particular function should be listed just above the function definition, not the prototype. The comment should give the function name, what it does, what any potential side effects are, and the range of return values. This is one example:

```
/* MC_finalize()
 *
 * This function shuts down the method control subsystem.  It is
 * responsible for tearing down internal data structures, shutting down
 * individual method devices, and gracefully removing any unfinished
 * operations.
 *
 * returns 0 on success, -errno on failure
 */
int MC_finalize(void)
{
   ...
}
```

If you are working on the PVFS project, then you should adhere to the function comments described in section 10.2.

9.1.4 Brackets

Brackets are of course used to delineate blocks of code contained within loops, conditional statements, or functions. For clarity, *any* statement executed within a conditional or loop should be enclosed in brackets, even if it is just one line. For example:

```
if(something true)
{
   do something;
}
```

and *not*

```
if(something true)
   do something;
```

Also note that each bracket gets it's own line in the source code.

9.1.5 Indentation

Indentation is also very important to writing clear code. The easiest rule to remember is that any new set of brackets should add a level of indentation for the code contained within it. This holds for functions, loops, and conditionals. The following is an example:

```
int foofunction(int x)
{
```

```
   int y = 0;

   if(x <= 0)
   {
      do some stuff;
   }
   else
   {
      for(y=0; y<x; y++)
      {
         do lots of stuff;
      }
   }

   return(0);
}
```

9.2 Hints for writing maintainable code

9.2.1 General code layout

These are a few general guidelines for how to organize your code:

- Group similar functions together into the same .c file.
- If a function will only be called from within the .c file where it is defined, then include the prototype for the function in the same .c file near the top. (see section 9.2.4 for information on static declarations)
- If a function will be called from outside of the .c file in which it is declared, then put the prototype in a header file separate from the .c file. This header should be included in any other .c file where the function will be called.
- Put comments describing the behavior of the function just before its definition, not with the prototype (see section 9.1.3 for more detailed information about commenting functions).
- Header files should *only* contain prototypes and structures that are needed by external pieces of code. It helps to encapsulate things by not providing extraneous information in the header files.

9.2.2 Length of functions

Try not to make extremely long functions. A good rule of thumb is to limit your functions to 100 lines or less. If a function is longer than this, then it should probably be broken apart into smaller subfunctions. Exceptions to this rule are rare.

9.2.3 Preventing double inclusion

If you are using a header file in several locations, it is easy to create a situation in which the same header file is indirectly included twice in a single compilation. This causes compilation errors because of function, variable, or type redefinition. In order to ensure that this does not happen,

you should always wrap your header files in preprocessor macros that prevent the code from being read more than once by the compiler. This may be done by creating a special define that can be detected the second time the code is included. The name of this define should stand out so that it does not conflict with other variables or definitions in your code. It is usually safe to pick the filename of header, convert it to all uppercase, and replace punctuation with underscores. Here is an example for a header file called bmi.h:

```
/*
 * (C) 2001 Clemson University and The University of Chicago
 *
 * See COPYING in top-level directory.
 */

/* This file contains the primary application interface to the BMI
 * library.
 */

#ifndef __BMI_H    /* these macros tell the compiler to skip the */
#define __BMI_H    /* following code if it hits it a second time */

/* now do whatever you would normally do in your header: */

#include<bmi_types.h>

struct foo{
   int x;
   int y;
};

int foo_function(double a, double b);

/* don't forget to end your header with this statement */
#endif /* __BMI_H */
```

9.2.4 Static declarations

Any function or variable that is declared global in a particular .c file but not referened in any other .c file should be declared static. This helps to keep the symbol name space from becoming cluttered. It also insures that local functions are not accidentally called somewhere that they were not intended to be called.

9.2.5 Initializing variables

Initialize all variables when they are declared in your software. Even if it is a trivial scalar variable, go ahead and initialize it. Integers and floats can typically be initialized to -1 or 0, while pointers can be initialized to NULL. This simple habit can help uncover many problems that occur when

the validity of a value is not checked before it is used. There is no guarantee what the value of a variable will be when it is created. Picking a known initial value to start out with can prevent garbage data from being interpreted as valid information.

A similar argument applies to memory regions that are dynamically allocated. Any dynamically allocated structure or variable should at least be zeroed out before being used in the code. This can be done with the `memset()` function:

```
foopointer = (struct foostruct)malloc(sizeof(struct foostruct));
if(foopointer == NULL)
{
   /* alloc failed */
   return(some error value);
}
memset(foopointer, 0, sizeof(struct foostruct));
```

If there are sentinal values other than 0 for elements contained in your struct, they should be set as well.

9.2.6 Allocating and deallocating complex structures

If there is a particular structure that you are frequently dynamically allocating or deallocating, it usually pays off to go ahead and create functions to handle those operations. This is especially helpful if there are further dynamically allocated structures within the original structure. Encapsulating all of this memory management in a pair of functions aids in debugging and makes your code more readable overall. A good naming convention is:

```
/* returns a pointer to new structure on success, null on failure */
struct foo* alloc_foo(void);

and

/* no return value */
void dealloc_foo(struct foo*);
```

9.2.7 Keeping up with work in progress

There are often questionable issues, or even issues that you don't have time to deal with at the moment, that come up when writing large pieces of code. It is generally helpful to document these questions or "todo" items in a known location so that they are not forgotten. There are two recommended ways of handling this. Keep larger or more imporant items listed in a file called "TODO" in the top level directory of your project. This file can be added to CVS so that other developers can see a quick list of known bugs or issues that need resolution. As items on this list are corrected, you may wish to log them in another file at the top level called "Changelog". Smaller issues, that are perhaps only important from a stylistic point of view, can be commented in the code and marked with the text string "TODO" within the comment. This is highlighted with a special color with vi syntax highlighting, and can easily be found with the `grep` tool later.

9.2.8 Choosing good variable and function names

Try to pick descriptive names for variables and functions, rather than saving keystrokes by picking obtuse abbreviations. This makes it easier for people who look at your code afterwards to understand what is going on. If your function or variable name is comprised of more than one word, then separate the word with underscores. If a collection of functions are related, or collectively form a common interface, the prepend an identifier to each function so that it is obvious that they belong together:

```
int test_control_open();
int test_control_close();
int test_control_read();
```

Function and variable nameing issues specific to PVFS can be found in section 10.3.

9.3 Advanced topics

9.3.1 Checking for interrupted system calls

If a system call fails, always check the return value to see if it was set to EINTR. If this happens, it means that the system call was interrupted by a signal and probably did not actually fail; it just needs to be restarted. This is a fairly common situation when doing `reads`, `writes`, or `polls`. You can restart operations either by wrapping them in a while loop that causes it to try again if EINTR occurs, or you can use a goto and a label to jump back to the the system call you wish to repeat.

9.3.2 Constant arguments

If you are passing in pointers as arguments to a function, but *do not* wish for the value contained in the pointer to be modified, then it is sometimes helpful to make the argument declaration a constant. This makes the compiler present a warning or an error if the value is accidentally modified within your function. This technique is especially useful when one of the arguments to your function is a string. In this case, you will probably be passing in a char* argument for convenience. However, passing in a string in this manner allows the function to modify the argument, which may not be desirable. Using a const char* argument can prevent this. Example:

```
int string_key(const char *key, const char *id_string)
{
   /* within this function it is now impossible to accidentally modify
    * the character strings pointed to by key or id_string
    */
   return(0);
}
```

9.3.3 Obscure coding practices

By all means, try to avoid the use of obscure coding tricks when writing software as part of a group. This especially true when there is there is an equally valid but much clearer method of accomplishing your goal. Obscure coding practices include but are not limited to:

- the : ? conditional operator

- unecessary goto statements
- nested switches
- implicit type conversion
- placing too much emphasis on makeing code small

9.3.4 Locking data structures

If you are programming in a multithreaded or reentrant environment, it is very important to use locking mechanisms effectively. Any global variable should be locked before it is accessed in this type of environment. The pthread library contains almost any sort of portable primitive you may need for a single application. It is also helpful to wrap these calls behind an interface that allows you to turn locking on or off at compile time. The ability to disable locking can be useful during development or when running code on a system that does not require locking. Look in the pvfs-locks CVS module for an example.

9.3.5 Select vs. poll

Try to avoid using the select system call and use poll in its place. Poll scales more efficiently. It is also the most direct function call for accomplishing the desired task on modern Linux kernels because select is implemented on top of the kernel's poll function.

9.3.6 String parsing

Be careful with regards to which functions you use when doing simple string parsing. Some of the functions provided in `string.h` are dangerous to use, either because they do not return error values, or because they alter their arguments. Most of these issues are documented in the man pages. One common example occurs when an integer value must be read out of a string. In this case, it is better to use sscanf than atoi:

```
char number_string[] = "300";
int my_number = -1;
ret = -1;

/* if you use sscanf, you can check the return value */
ret = sscanf(number_string, "%d", &my_number);
if(ret < 1)
{
   return an error;
}

/* as opposed to atoi, which will not tell you if it fails */
my_number = atoi(number_string);
```

9.3.7 Abstraction

When you are designing new interfaces, think carefully about how to create an abstraction for what you want the interface to do. The important idea here is to not be tied down to a particular

implementation below your interface because you made the interface too restrictive. For example, suppose that you wish to create an interface for storing and retrieving a large number of independent objects. One way to implement this may be to use a hashing function. However, most people consider it to be much quicker to get a simple linked list working. If you abstract the interface correctly, you can implement the functionality with a linked list for now just to get your program working and then come back later and plug in a hash table implementation. This is only possible with a good abstraction, however. If your first interface has functions such as "add_to_head" or "create_new_list" that pass around pointers to lists, then it will of course be difficult to change this interface to use a hash table. It would be better to use functions such as "store_item" or "create_new_container" and use opaque types to keep up with your data structure.

9.3.8 Function pointers

Function pointers can be useful when creating modular code. They allow you to pick which function will be used to perform a given task at run time rather than compile time. This is not really any harder than manipulating pointers to variables:

```
/* this is the first way to send a message */
int send_message_one(void* data, int size);

/* this is the second way to send a message */
int send_message_two(void* data, int size);

/* this is a pointer to the prefered method */
int (*send_message_generic)(void*, int) = NULL;

...

if(something is true)
{
   send_message_generic = send_message_one;
}
else
{
   send_message_generic = send_message_two;
}

...

/* We don't care which method the user chose.  We know that it can be
 * accessed through this function pointer without us modifying our code.
 */
send_message_generic(my_data, sizeof(my_data));
```

9.3.9 Typedefs and opaque types

Choosing appropriate types for objects passed around in your code can be very important in some situations. There are a couple of different issues here:

- **Platform dependence:** Different architectures use a different number of bytes for some variable types. This means that it can sometimes be very helpful to explicitly choose the size of some variables to aid portability. This is especially true if the data is going to passed over a network, although there are more issues (such as big-endian vs. little-endian) to worry about in those situations. It is often a good idea to use typedefs to create new type names that have a known, fixed size:

  ```
  typedef int32_t pvfs_flag_t;
  ```

  This guarantees that when a pvfs_flag_t variable is declared, it will be a 32 bit integer, regardless of the host architecture.

- **Opaque types:** Sometimes you wish to have an interface operate in terms of a specific type. If you are not certain of what type should be used for this purpose in the long term, you can hide it behind a typedef'd opaque type. That way, if you change the type later, you may not have to change every reference to it in the code. You just have to change the initial typedef statement. This can be done for structs or scalar types.

  Guess I need an example here...

10 Specific PVFS issues

10.1 Copyright information

Copyright information at the top of source code in PVFS should include the University of Chicago. (The University of Chicago is affiliated with Argonne National Lab, where several key PVFS developers are located).

```
/*
 * (C) 2001 Clemson University and The University of Chicago.
 *
 * See COPYING in top-level directory.
 */
```

10.2 Function commenting

Figure out what standard should be to match up with autodocument tools. This is on hold until we settle on such a tool, so use standard put forth in section 9.1.3 for now.

10.3 Function naming

Interface functions and global variables in PVFS should use a standard naming convention for clarity. Here are a few guidelines:

- The letters "PVFS" should only be prepended to functions and global variables that exist as part of an application level interface. Some examples would be `PVFS_open` and `PVFS_read`. For clarity, do not use this naming scheme for interfaces internal to PVFS.

- Well defined internal PVFS interfaces should use the prefix "PINT" (this is short for "PVFS interface"). This should then be followed by an identifier for the interface, and then a description of what the particular function does. Some examples are `PINT_flow_alloc`, `PINT_flow_free`, and `PINT_flow_post`.

- There are exceptions to the above rule. For example, well defined interfaces that exist within a very distinct module of PVFS may use a different prefix. Examples include the method functions within the BMI layer of PVFS communications have names such as `METH_tcp_send` and `METH_tcp_recv`.

- Any variables that are globally visible should follow the rules listed above as well. This naming convention is for both functions and global variables.

10.4 Error logging with Gossip

Gossip is a simple library for logging both errors and debugging messages. It allows you to send logging messages to either stderr, syslog, or a text file.

Gossip uses a `debug mask` to determine which messages get logged. You may specify a mask level with each debugging call. These messages can then be toggled on or off depending on what the global mask value is. This allows you to turn debugging on or off just for specific parts of your software at run time. The global mask may be made up of several individual mask values logially or'd together in order to enable logging for multiple parts of your software simultaneously.

Gossip also allows you to send error messages. These are similar to debugging messages, except that they get logged regardless of the mask value and whether debugging is turned on or off. These error messages should only be used in situations in which a critical error should be recorded.

The following is a list of functions provided in the Gossip library:

- **gossip_enable_stderr()**: Directs logging messages to stderr.

- **gossip_enable_file(filename, mode)**: Directs logging messages to a specified file. The arguments are the same as those taken by the `fopen()` function.

- **gossip_enable_syslog(priority)**: Directs logging to syslog. The priority argument is the same as that given to the `syslog()` function.

- **gossip_set_debug_mask(debug_on, mask)**: Turns debugging messages on or off and specifies the mask value to use if turned on.

- **gossip_disable()**: Gracefully shuts down the Gossip logging facilities.

- **gossip_debug(mask, format, ...)**: Logs a debugging message. Uses the same format syntax as the `printf()` function call. It will only print if debugging is turned on and the mask value matches the global mask specified with gossip_set_debug_mask().

- **gossip_ldebug(mask, format, ...)**: Same as above, except that it prepends each message with the file name and line number of the source code that invoked it.

- **gossip_err(format, ...)**: Logs error messages. These will print regardless of the mask and whether debugging is turned on or off.

- **gossip_lerr(format, ...)**: Same as above, except that it prepends each message with the file name and line number of the source code that invoked it.

Examples of how to use Gossip can be found in the `gossip/examples` directory of the Gossip source code. This code can be found in the `pvfs2/src/common/gossip` directory within the PVFS 2 source tree.

10.5 Suggested error handling

10.5.1 Traditional application error handling with errno

Most unix system calls set a global variable called `errno` when an error condition occurs. Since this is a global variable, it is overwritten everytime a system call is made. This means that it must be checked immediately following the failure of the system call in question. The errno values correspond to to various error conditions, wuch as "bad file descriptor" or "permission denied." One can print out a textual description of these error values using the `perror()` or `strerror()` functions. More information about the use of `errno` can be found in the man pages for `errno`, `perror`, and and `strerror`.

The use of errno in this manner is fine for small applications, but becomes more tedious when building larger software projects. The problem is that you must store the error value somewhere when passing the error back through multiple abstraction layers. This tends to cause confusion in large projects.