Coding Guidelines
and Quick Start Tips
for
Software Development

Version 0.6 (in progress)

Includes: C, Python, and some Assembler and C++

File: "C:\Travel_Briefcase\EricSchool\Research\Coding Guidelines.doc"

Last modified by Eric L. Michelsen

The goal of coding guidelines is to improve the productivity of all software development:
Easier, faster, more reliable.

1. Source code is a language for people, not just computers.
2. Comment as you go. Don't wait for later.
3. Ask yourself: "How will the next person know that?"

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by [implication], not smart enough to debug it." **- Brian W. Kernighan**

"Whenever possible, ignore the coding standards currently in use by thousands of developers in your project's target language and environment."
- Roedy Green, *How To Write Unmaintainable Code*, www.strauss.za.com/sla/code_std.html

"There is much more to programming than simply writing the code."
- T. M. R. Ellis, *Fortran 90 Programming*, Addison-Wesley, 1994, p693.

"These guidelines will make everyone's life easier, even yours." **- Eric L. Michelsen**

Table of Contents

Acknowledgement

We thank Copper Mountain Networks for their support in the preparation of this document.

1. Why Coding Guidelines?

Why? Because

> Code is read much more often than it is written.

Coding guidelines are a tool for cost-effective engineering. (They are not a religion or an art form.) When evaluating coding guidelines, it is important to **focus on the *utility* of the guidelines, and let go of things that we "like" or are "used to."**

> 1. The goal of coding guidelines is to improve the productivity of all software development: Easier, more reliable, faster.

While many coding styles are efficient and maintainable, agreeing on *this* set of guidelines allows the entire team to work cohesively. Close adherence to these guidelines is essential to producing software that is consistent project-wide and maintainable by diverse team members.

Of course, no finite set of guidelines is applicable to *all* situations. There will be rare times when developers consciously and properly diverge from these guidelines. When such cases arise, include an explanatory comment (to facilitate the review process) .

> 2. Comment as you go. It only takes a few seconds. Don't wait for later.

The code will never be fresher in your mind than right now. The file is already open in the editor. This is the most effective time there will be to comment your code. It only takes a few seconds. And you know you won't get around to it later.

> 3. Source code is a language for people, not just computers.

Source code has two purposes: (1) To instruct a computer what to do, and (2) to instruct a human developer/reviewer what the code is doing. Source code is as much a language for *people* as it is for the computer. Coding guidelines are tugged in two opposing directions. Coding guidelines which are too strict limit the designer's flexibility to express design concepts in the source code. Coding guidelines that are too "loose" allow too much confusion in the code.

> 4. Ask yourself: "How will the next person know that?"

Things that were hard for you to figure out are hard for other people, too. Comment what you learned, what you know, what your code is doing, and how it is doing it.

1.1 Guidelines Summary

Here's a summary of the major guidelines; details and justifications are in the rest of the document.

General Guidelines

- It's not always the best idea to design something for all situations or all time, because it may be a waste of resources. However, it's usually pretty easy to test for the known limitation, such as
```
if(year >= 2009)  print error message, and terminate.
```
- All code *should* be compiled with an ANSI C Compiler to ensure compatibility (at least from the compiler's point of view) with the ANSI C standard. The INLINE macro is a safe extension.
- Be C++ compatible: do not use C++ reserved words. Do not typedef structure tag names, but do typedef structures.
- Always start new source files with the latest version of the file templates, template.c and template.h. Don't copy old source files, as things may have been improved since then.
- Old code may be grandfathered, but updating to the guidelines is encouraged.
- A quality requirement for code is no lint/compiler/linker warnings.
- Developers *should not* use Hungarian notation for identifier names.
- Do not use double underscore to start any identifier or macro; these are reserved for the compiler.
- When documenting designs, a small number of large documents is much easier to search through than a large number of small documents.

Code Organization

- Group functions so as to maximize the amount of 'local' data and minimize the amount of 'shared' and 'global' data. Therefore, **group functions because they reference the same data, rather than because they perform similar kinds of functions**.
- Programs with multiple files and multiple functions *should* be organized into modules. A **module** is a file or set of files that form a process or library function. All files in a module *should* be in a single directory, except header files shared with other modules, which are in a common directory.
- Header files that are used by files in only one directory *should* be put in that directory. Header files used by files in multiple directories *should* be stored in the lowest parent directory of all software modules that include them.
- Each file in a given module *should* be prefixed with a short prefix unique to the module.
- Include files that contain global declarations that are shared between modules (i.e. not within a module) *should* have their file names prefixed with "gi" (Global Include) to indicate that they are shared. This *should* alert the developer to be especially careful when making changes.

File Layout

- All functions and data that are used outside the module (i.e. global) *should* be prefixed with the same prefix that identifies the module.
- Files *should* define functions in top-down order, i.e., the highest level functions first, because top-down is usually the easiest way to learn a new module. This organization means local (forward) prototypes are needed, because low level functions are defined *after* they are referenced.
- Each source file *should* begin with a header comment block that indicates the file name, the module that the file belongs to, and the general purpose of the functions in the file. This comes for free by using the file templates (template.c and template.h) for new source files.

- Put comments on your #include lines to document the most important identifiers supplied by the header file. This helps avoid #include bloat:

```
#include <time.h>      // clock(), time_t
#include "apollo.h"    // card slots
```

- 'C' Library header files, or OS header files *should* be included with <> (angle brackets) to indicate they are library files. Project generated header files *should* be included using " " (quotes).
- Header files define the *interfaces* for a module. Header files *should* include the minimum amount of information (Mandatory typedefs, prototypes, constants, structure declarations, global data definitions) that is needed by external modules to use public functions. Private typedefs, prototypes, etc. *should* be in the *.c file, *not* the *.h file.
- Developers *should* avoid including header files within header files.

Functions

- All function calls MUST refer to previously declared functions (C99 and C++ mandate this).
- Each function *should* begin with a header comment block which conveys information about the purpose of the function, the expected inputs, and the outputs.
- The scope of all functions *should* be declared using the macros PUBLIC, PRIVATE, or SHELL.
- Functions that act on objects should have a name giving the object type first, then the operation, e.g., 'date_increment', rather than 'increment_date'.
- The function name *should* reflect how the caller uses its return value. E.g., a function whose job it is to determine whether a file name is valid, and return a boolean result, should be called something like 'filename_is_valid' rather than 'validate_filename'.
- Private function prototypes *should* be at the top of the *.c file, to serve as forward declarations.
- You *should* qualify *pointer* parameters to functions with the following modifiers to help create self documenting code: const, OUT, INOUT.
- Arithmetic macro function definitions *should* enclose all the arguments in parentheses, and themselves be enclosed in overall parentheses.

Typedefs

- Structures *should* be typedef'd to allow users to reference the typedef name directly. (C++ classes should not be typedef'd, since the classname is already effectively a typedef.)
- Use the project predefined typedefs (comtypes.h) in place of most of the standard C data types: `uint8, int8, uint16, int16, uint32, int32, uint64, int64, bool.` These are consistent with Python's numpy types (http://scipy.org/Tentative_NumPy_Tutorial).

Variables

- Each variable declaration *should* have a comment describing its purpose, if not self-explanatory.
- There *should* be no single character variable names. Single character variable names are hard to search for in code and not descriptive enough to be useful.

Constants

- Let the compiler do the work: when coding compile time constants which are calculated, write them as an arithmetic expression that conveys their meaning and derivation:

```
#define TDC_SLOT_BITS  (TDC_SLOT  <<  12)
```

- Use upper case to name compile-time constants (including 'const', #define, and enums):
- Run-time constants *should not* use upper case (as are compile-time constants) since they are more akin to variables than to compile-time constants.

Statement Formatting

- Each indent level is 4 spaces. Source code *may* include tabs, however, any ASCII <tab>s in source code *MUST* use 8-space tab stops to ensure interoperation with standard display utilities.
- Paired braces *should* always line up in the same column and with the controlling statement**.** This makes it obvious where a block of code begins and ends. [This is *not* K&R-like.]

```
while (!done)
{   if (idx == 5)
    {   ...la de da;
    }
}
```

- Use the newer "//" token to start comments that end at the end of the source line. This is now standard in both C99 and C++.
- Use conventionalized comments to flag sensitive code, especially `switch()` fall throughs.
- Expect everyone to know the 4 basic operator precedence groups:
  function( ), subscript [ ], `struct.member`, `struct_ptr->member`
  arithmetic: `*, /, +, -`
  boolean: `&&, ||`
  assignment: =, += , -=, etc.
  Use parentheses for the rest.
- You *should not* put assignments within other statements. This avoids unexpected behavior, especially in arguments passed to macros. This applies to some degree to the ++ and -- operators.
- For pointer manipulation, use `sizeof()` and `offsetof()` for readability and portability.

Simple Examples

```
// small 'if'
if(color == RED)  nred++;     // count each color's frequency

if(condition == DANGER)       // medium 'if'
    alert_operator();

// small blocks of code
while (!done)
{   if (idx == 5)
    {   sumx += curval();
        nsum++;
    }
}

// big blocks of code
while (!done)
{
    if (idx == 5)
    {
        ...long block of code
    } // if(idx ...)
} // while(!done)

switch(value)
{
case 1:
    ...code
    break;
 ...
}
```

1.2 Document Overview

1.2.1 Scope

This document describes the coding practices and procedures recommended for *all* software development, in groups of 1 or more. The guidelines are "lightweight" enough to be easy to follow, but complete enough to enable new developers to join a coding project quickly. This document contains guidelines for writing C, C++, and Python software, porting 3rd party software, and writing assembly language procedures.

1.2.2 Notation

```
Acceptable code fragments are printed in this style;
Deprecated code fragments are printed in this style.
```

1.2.3 Terminology

Modeled on IETF RFC2119, this document uses the terms *MUST*, *should*, *may*, *MUST NOT*, and *should not* to clearly specify the degree of compliance required:

may	this word means that an item is truly optional. One developer may choose to include the item because they feel it adds value in terms of readability or behavior to their software implementation.
MUST	this word means that the guideline is an absolute requirement for software implementation
MUST NOT	this phrase means that the guideline is an absolute prohibition of software implementation
should	this word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course
should not	this phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label

1.3 Issues

1.3.1 Open Issues

This section outlines open issues that need to be resolved.

1. lint
2. Do we need copyright notices in our software?
3. ?? Both C and C++ allow objects and functions to have static file linkage, also known as internal linkage. C++, however, deems this as deprecated practice (says who?), preferring the use of unnamed namespaces instead. (C++ objects and functions declared within unnamed namespaces have external linkage unless they are explicitly declared static. C++ deems the use of static specifiers on objects or function declarations within namespace scope as deprecated.) ?? Do PRIVATE and PUBLIC fix this?

1.4 Assumptions

Assumptions made in these guidelines:

1. These guidelines assume you are already familiar with your programming languages.

1.5 Definitions, Abbreviations, Acronyms

This section contains definitions, abbreviations, and acronyms that are common in this document.

The default format for terms in this section is bold with the first letter capitalized. Other formatting (all caps, italics, etc.) is included if the terms are always used with that formatting.

<Term>	<definition >
Actual arguments	the parameters passed to a function by the caller.
Automatic	describes variables of function-local scope that are not statically declared.
Definition	in C, the statements where a variable or function is fully described/initialized.
Declaration	in C, the statement that allows further code to reference a function or variable. [K&R, 2nd ed., p210].
Formal parameters	the parameter definitions in prototypes and function definitions.
Global	describes any data or functions that are accessible between modules and components in a software build. The externs and prototypes for these *should* be included in a header file that is hierarchically above all the modules that reference them.
Hungarian Notation	Do not use Hungarian notation. In Hungarian Notation, the data type of the variable or function return value is indicated by prefixing the name with characters that indicate the returned data type. For example, a function that returns an integer value might be named '**i**MyFunction', a function that returns a pointer to an integer might be '**pi**MyFunction', etc.
Local	describes any data or functions that are accessible to a single file. These *should* always declared with the PRIVATE macro.
may	this word means that an item is truly optional. One developer may choose to include the item because they feel it adds value in terms of readability or behavior to their software implementation.
MUST	this word means that the guideline is an absolute requirement for software implementation
MUST NOT	this phrase means that the guideline is an absolute prohibition of software implementation
PRIVATE	is a label that is applied to all data and function definitions that are NOT visible outside the scope of the file they are defined in. Not to be confused with the C++ keyword "private".
PUBLIC	is a label that is applied to all data and function definitions that are visible outside the scope of the file they are defined in. Not to be confused with the C++ keyword "public".
Shared	describes any data or functions that are accessible within multiple files of a single module. The externs and prototypes for these *should* be included in a header file that is in the module's directory.
SHELL	is a label that is applied to all data and function definitions that are NOT intended to be used by operational software, but they need to be visible at the Shell for diagnostic use. There *should* be no external references for SHELL functions or variables. If there are external references, then they *should* be defined as PUBLIC.
should	this word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course

<Term>	<definition >
should not	this phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label

1.6 References

[1] RFC2119 *Key words for use in RFCs to Indicate Requirement Levels*: Contains the IETF standard definitions for *MUST*, *MUST NOT*, *should*, *should not*, and *may*

[2] <http://david.tribble.com/text/cdiffs.htm>, comparison of C99 and C++.

[3] http://www.oreillynet.com/pub/a/network/2003/10/07/michael_barr.html, standard integers.

1.7 Revision History

Track the document history in this section. Format is:

<date>	<Writer's initials>: <Document version>. <Description of changes>
7/21/04	ELM: Version 0.1: Initial draft.
9/7/2004	ELM: V 0.2.
1/2009	ELM: many major updates accumulated over time.

2. 'C' Coding Guidelines

This chapter outlines 'C' syntax, style, and coding conventions.

2.1 General Guidelines

ANSI C All code *should* be compiled with an ANSI C Compiler to ensure compatibility (at least from the compiler's point of view) with the ANSI C standard.

Do not use double underscore to start any identifier or macro; these are reserved for the compiler.

Developers *should not* use Hungarian notation for identifier names, because they are hard to read. ["Hungarian Notation is the tactical nuclear weapon of source code obfuscation techniques" -Roedy Green, www.strauss.za.com/sla/code_std.html]

2.1.1 Templates

Templates help to maintain consistency among a set of developers, and make it easy to get started on the coding process. This consistency also allows for automatic text search utilities to find things, and extract them for documentation. We have created a set of templates for coding in 'C' with the following names:

'C' Code File: `template.c`

'C' Header File: `template.h`

Make File: `template.mak`

Whenever creating a new module/file, developers *should* start with the latest version of the templates, because they improve over time.

The current templates are typically stored in the project "include" directory, e.g. /home/apollo/include.

2.1.2 Grandfathering

Because guidelines tend to change over time, some older code may not completely comply with the current guidelines. You *may* leave older code as is, or you *may* change it. Bear in mind that there are risks to both options. Particularly for comments, updating to the current guidelines is encouraged.

2.1.3 No Warnings

A quality requirement for code is no lint/compiler/linker warnings. While some warnings may seem harmless, they often mask larger problems. Also, code that regularly produces warning messages can cause the developer to miss a *new* warning that *is* a problem. And when others inherit code with warnings, it's difficult for them to know if the warnings are problems, or "harmless."

2.1.3.1 Lint Warnings

All code *MUST* pass lint tests. The standard configuration parameters for LINT are defined in a set of lint configuration files that are referenced in our make files.

2.1.3.2 Compiler Warnings

To turn off inappropriate compiler warnings, you *may* create dummy statements with a comment to indicate the purpose. For example, this is useful for variables that are defined but not referenced.

```
:
PRIVATE uint8   *author_ptr = "Eric Michelsen";
:
main
{
   (void ) author_ptr;   // Avoid warning
   ...
}
```

2.1.3.3 Linker Warnings

Code *MUST* link without warnings.

When documenting designs, a small number of large documents is much easier to search through than a large number of small documents.

2.2 C++ and C99 Compatibility

Migration to C++ may be necessary at some time. Because some constructs that are valid in C are *invalid* in C++, or behave differently in C++, you *should* avoid the following incompatible cases, and code strictly in "clean C." "Clean C" compiles and runs properly in C++.

Developers *should not* use the following C++ reserved words (as always, case sensitive), **common ones are indicated in bold**:

```
asm             bool          catch          class          const_cast
delete          dynamic_cast  explicit       export
false           friend        inline         mutable
namespace       new           operator       private        protected      public
reinterpret_cast              static_cast
template        this          throw          true           try            typeid
typename        using         virtual        wchar_t

and             and_eq        bitand         bitor          compl
not             not_eq        or             or_eq          xor            xor_eq
```

Developers *should* avoid repeating structure or union tags in structure/union typedefs, because in C++, a structure/union tag is automatically a typedef. Therefore, including both the tag and the typedef causes a duplicate definition error in C++.

```
typedef struct { ... } xyz;                  // C++ compatible
instead of:  typedef struct xyz { ... } xyz; // Not C++ compatible
```

A consequence of C++ function name mangling is that identifiers in C++ are not allowed to contain two or more consecutive underscores (e.g., the name `foo__bar` is invalid). Such names are reserved for the implementation, ostensibly so that it may have a guaranteed means of mangling source function names into unique object symbolic names [2].

?? Both C and C++ allow objects and functions to have `static` *file linkage*, also known as **internal linkage**. C++, however, deems this as deprecated practice, preferring the use of *unnamed namespaces* instead. (C++ objects and functions declared within unnamed namespaces have *external linkage* unless they are explicitly declared `static`. C++ deems the use of `static` specifiers on objects or function declarations within namespace scope as deprecated.) ?? Do PRIVATE and PUBLIC fix this?

C99 provides a predefined identifier, `__func__`, which acts like a string literal containing the name of the enclosing function. While this feature is likely to be provided as an extension by many C++ compilers, it is unclear what its value would be, especially for member functions within nested template classes declared within nested namespaces.

C99 has a few reserved keywords that are not recognized by C++: `restrict`, `_Bool`, `_Complex`, `_Imaginary`, `_Pragma`

This will cause problems when C code containing these tokens is compiled as C++. For example:

```
extern int   set_name(char *restrict n);
```

[C99: §6.4.1, 6.7.2, 6.7.3, 6.7.3.1, 6.10.9, 7.3.1, 7.16, A.1.2]
[C++98: §2.11]

2.2.1 Enums As Arguments

There is no guarantee that a given enumeration type is implemented as the same underlying type in both C and C++, or even in different C implementations. This affects the calling interface between C and C++ functions. This may also cause incompatibilities for C code compiled as C++, if the C++ compiler chooses to implement an enumeration type as a different size that it would be in C, or if the program relies on the results of expressions such as `sizeof(RED)`:

```
// C++ code

enum Color {RED, WHITE, BLUE};

extern "C" void  foo(Color c);    // Parameter types might not match

void bar(Color c)
{
    foo(c);                       // Enum types might be different sizes
}
```

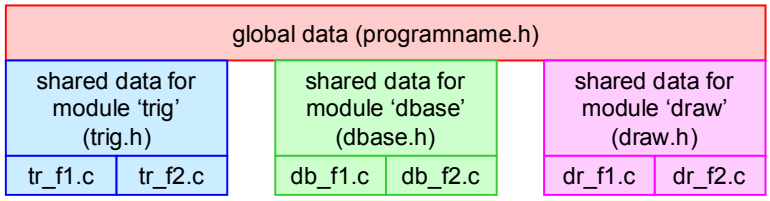
[C99: §6.4.4.3, 6.7.2.2]. [C++98: §4.5, 7.2]

2.3 Code Organization

Software *should* be organized into files in such a way as to increase cohesion and minimize coupling. One guideline for producing this effect is to group functions so as to maximize the amount of 'local' data and minimize the amount of 'shared' and 'global' data. Therefore:

> Group functions because they reference the same data, rather than because they perform similar kinds of functions.

From the definitions of "shared" and "PUBLIC" given earlier, the basic hierarchy is this:



Definitions of "shared" and "PUBLIC": each file can see the data in boxes above it.

For instance, if you need to decide how to organize functionality that performs the following major functions on 2 different kinds of data (represented abstractly here as 'aa' and 'bb'):

- Initialize data for module 'aa' and 'bb'
- Collect and manipulate data for 'aa' and 'bb'
- Provide database access functions to 'aa' and 'bb'

It would be best to group them like:

Module aa
Initialize 'aa'
Collect and manipulate 'aa'
DB Access Functions to 'aa'

Module bb
Initialize 'bb'
Collect and manipulate 'bb'
DB Access Functions to 'bb'

Such an approach will often result in a structure where a 'higher-level' module (say the parent of 'aa' and 'bb') will provide an initialization routine which will call the initialization routines of sub-modules. The following would be an example of this paradigm.

```
PUBLIC void top_initialize(void)
{       aa_init();
        bb_init();

        ... more initialization code goes here ...
}
```

Programs with multiple files and multiple functions *should* be organized into modules. A module is a file or set of files that form a process or library.

2.4 Directory Layout

A good goal for directory layout is that you can copy a single directory tree and get everything you need.

Large programs may need to be divided into modules. A **module** comprises one or more files. All files in a module *should* be in a single directory, except header files shared with other modules, which are included in a common directory. Sometimes, multiple small modules may be together in one directory.

The current APOLLO directory layout is this:

```
Makefile                    // the makefile for all programs in 'src/'
include\                    // All shared include files
src\                        // common code/data library used by many programs
    library-name\           // one of the libraries
        *.c                 // it's *.c files.  No *.h files here!

    ... other libraries similar

    housctl\                // houston control program
        *.h                 // *private* include files
        *.c                 // houston control *.c files

    ... other pograms similar
```

When a declaration is shared between two files, put the declaration in the header file as low in the directory hierarchy as possible. In the example above, declarations shared by mod1_main.c and mod1_util.c *should* go in mod1_private.h. Declarations global to all code would go in a header file in Common\INC\.

I would rather see the "include/" directory be inside "src/", so that *all* the source is under the single directory "src/". Similarly, the makefile should be in src/, not in the top level. -- ELM

2.5 File Layout

Need a picture??

Each file in a given module *should* be prefixed with a short prefix unique to the module.

Include files that contain global declarations that are shared between modules (i.e. not within a module) *should* have their file names prefixed with "gi" (Global Include) to indicate that they are shared. This *should* alert the developer to be especially careful when making changes. For example, the project include file might be named gi_apollo.h

All functions and data that are used outside the module (i.e. global) *should* be prefixed with the same prefix that identifies the module. This helps developers to quickly identify where the function and variables are defined, and provides more insight into their general purpose.

For example, the executive may be broken up into 3 files:

```
ex_main.c
ex_data.c
ex_util.c
```

Examples of PUBLIC functions found in the above files:

```
ex_set_current_time()
ex_get_current_time()
```

2.5.1 File Layout: *.c Files

*.c files will look similar to this:

```
/*      abc.c   description


Copyright 2009 blah.  All rights reserved.

This file contains ...??

Author:         Your name here
Design Doc:     Name of HLD/LLD that pertains to the functions in this file
Date Created:   ??
============================================================================*/

// ---------------- Open Issues             (do not remove) ------------------

// ---------------- System Includes         (do not remove) ------------------
// use <> (angle brackets) for library headers
#include <stddef.h>           // offsetof()

#include "comtypes.h"

// ---------------- Local Includes          (do not remove) ------------------
#include "abc.h"              // abc_needthis()

// ---------------- Constants               (do not remove) ------------------

// ---------------- Structures/Types        (do not remove) ------------------

// ---------------- Public Variables (use sparingly, do not remove) ----------

// ---------------- Private Variables       (do not remove) ------------------

// ---------------- Private Prototypes/Macros (do not remove) ----------------



/*----------------------------------------------------------------------------
Descriptive Name (optional)
Description of the purpose of the routine, and any algorithm information.

Static  IN:     List the externally defined static variables that are used by
                this function
Static  OUT:    List the externally defined static variables that are modified
                by this function.

Notes:  This section relates any special circumstances of this code.  It is
flush left for easy editing.
----------------------------------------------------------------------------*/
PUBLIC  type    ex_func(        // describe return values here ...
        type    arg1,           // decribe arg1 here
  const type    *arg2_ptr,      // describe arg2_ptr here ...
  INOUT type    *arg3_ptr,      /* describe arg3_ptr here, if this is a
                                 particularly complex input, use multiple lines*/
  OUT   type    *arg4_ptr)      // describe arg4_ptr here, terminate with paren
{
    ... YOUR CODE HERE ...
} // ex_func()
```

2.5.1.1 Function Placement

Files *should* define functions in top-down order, i.e., the highest level functions first, because top-down is usually the easiest way to learn a new module. This organization means local (forward) prototypes are needed, because low level functions are defined *after* they are referenced.

2.5.1.2 Source Module/File Comment Blocks

Each source file *should* begin with a header comment block that indicates the file name, the module that the file belongs to, and the general purpose of the functions in the file. Someone reading this header *should* be able to determine the purpose of the file, and how it fits into the overall software picture. The header *should* be brief and to the point. A template for the source file header is provided in template.c.

Each file header *should* include a copyright notice, the name of the original author of the file, a reference to the design document associated with the source in the file, and the date that the file was created. The other information included in the file header *should* be divided by type into sections with a standard set of section delimiters (Open Issues, System Includes, Local Includes, etc.). The section delimiters allow for consistent formatting of source files by many different programmers and make it easy to find specific information. They *should* be included even for empty sections so that the sections are in the same order in all files.

Each file *should* contain a revision history, written and controlled by the configuration management package. The revision history is stored at the end of the file, so that the programmer does not need to wade through the history each time the file is opened. Example:

2.5.1.3 #include

Put comments on your #include lines to document the most important identifiers supplied by the header file. This helps avoid #include bloat: when later a reference is removed, one can more readily remove the now-spurious #include for it.

When including header files in a source file, do not include a directory path for the header file. The rules for searching the include paths *should* be encoded in the make file. This allows the developer to move header files without impacting the source code.

'C' Library header files, or OS header files *should* be included with < > (angle brackets) to indicate they are library files. Project generated header files *should* be included using " " (quotes). Compilers use different search rules for include files enclosed in < > and " ". For include files enclosed in "", the compiler searches in the local directory first.

2.5.2 File Layout: *.h (Header) Files

Header files define the *interfaces* for a module.

Header files should include the minimum amount of information (prototypes, constants, structure declarations, global data definitions) that is needed by external modules to use public functions. Private typedefs and prototypes should be in the *.c file, not the *.h file.

Developers *should* avoid including header files within header files [Str p211: "... the single header file approach is unworkable..."]. Header files usually contain only prototypes, constants, and data structures. [Where global data is used, it *may* be defined using the EXPORTED macro defined in an appendix.]

The following is an example of a header file, abc.h:

```
/*      abc.h   description

Copyright 2009 blah.  All rights reserved.

All the declarations [needed to use][private to] the ?? module.

Author:         ??
Date Created:   ??
============================================================================*/

#ifndef INC_ABC_H
#define INC_ABC_H      // Make sure the include happens only once

// ---------------- Prerequisites -----------------------------------------
// e.g., Requires "comtypes.h"

// ---------------- Constants           (do not remove) ---------------------

// ---------------- Structures/Types    (do not remove) ---------------------

#ifdef __cplusplus
extern "C" {
#endif
// ---------------- Public Variables (use sparingly, do not remove) ----------

// ---------------- Prototypes/Macros   (do not remove) ---------------------
// C prototypes



#ifdef __cplusplus
}
// C++ prototypes
#endif // __cplusplus

#endif // INC_ABC_H
```

The #ifndef INC_ABC_H is needed in case this file is included in other header files. Developers *should* avoid including header files within header files. However, if you run into a situation that warrants it, the format above *should* be used, and you *should* put nested #includes inside the #ifndef INC_xxx_H.

2.6 Functions

2.6.1 Function Calls

All function calls MUST refer to previously declared functions. Implicit function declarations are not allowed (C99 and C++ mandate this).

2.6.2 Function Headers and Footers

Each function *should* begin with a header comment block which conveys information about the purpose of the function, the expected inputs, and the outputs. These headers *should* be direct and concise. The header *should* also include any unexpected behavior that may occur when the function is called. A developer *should* be able to understand how to use a function, and what to expect of it by reading the header.

The scope of all functions *should* be declared using the macros PUBLIC, PRIVATE, or SHELL. This forces the developer to explicitly identify functions that are meant to have a scope outside the current file.

Sections of the header that do not apply to a particular function *should* be retained in the file, and left blank. This allows them to be filled-in in the future, if needed. You *may* precede the function header with <FF> (control-L) to force the function to appear on a new page in a listing.

The function definition *should* follow the function header. Each function *should* have a comment on its closing brace that indicates the function name. (This is particularly useful when rummaging through medium/large modules.)

The preferred format places the parameter information (including return value) as comments in the actual definition:

```
//---------------------------------------------------------------------------
Descriptive Name (optional)
Description of the purpose of the routine, and any algorithm information.

Static IN:  List the externally defined static variables that are used by this
            function
Static OUT: List the externally defined static variables that are modified by
            this function.

Notes:  This section relates any special circumstances of this code.  It is
flush left for easy editing.
---------------------------------------------------------------------------*/
PUBLIC type ex_func(       // describe return values here ...

          type arg1,       // decribe arg1 here
    const type *arg2_ptr,  // describe arg2_ptr here ...
    INOUT type *arg3_ptr,  /* describe arg3_ptr here, if this is a
                              particularly complex input, use multiple lines */
    OUT   type *arg4_ptr)  // describe arg4_ptr here, terminate with paren
{
} // ex_func()
```

NOTE: For functions with no parameters, the definition *may* go on a single line (with 'void' in parentheses).

NOTE: For lists of Static IN/OUT that are long, you *may* put in descriptive text about the kinds of statics referenced instead of listing each one in detail. The main goal is readability and understanding. "Static" refers to both local statics, as well as global data.

2.6.3 Function Naming

Functions *should* be named with an eye toward the readability of the code which will call the function. For instance, the function name *should* reflect how the caller uses its return value. E.g., a function whose job it is to determine whether a file name is valid, and return a boolean result, should be called something like 'filename_is_valid' rather than 'validate_filename'.

This has two benefits: both the caller and implementer of the function can easily keep in mind the sense of the return value, so the logic of the code which calls the function will be clearer. For example:

```
if (fm_filname_is_valid(filname_ptr))        // this is easier to read/understand
{
    ...
instead of:  if ( fm_validate_filename(filename_ptr) )  // this is less clear
```

Functions that act on objects should have a name giving the object type first, then the operation, e.g., 'date_increment', rather than 'increment_date'.

Multi-word function names *may* use underscores or capital letters to segregate words. Either method is readable (for example, fm_filename_is_valid and fmFileNameIsValid are both acceptable). Developers *should* use the same style consistently within a single module.

2.6.4 Function Prototypes

Function prototypes have different formatting rules than do function definitions. You *may* use the same format for the prototypes, or you *may* choose a more compact representation. In either case you *should* use the parameter qualifiers 'const', OUT, and INOUT as described below in 2.6.4.3 Function Formal Parameters.

2.6.4.1 Public Function Prototypes

Public function prototypes appear only in header files. Every file that uses public functions *should* include the header file for those functions (i.e. use the prototypes). You *should* declare public functions using the keyword **"**`extern`**"**.

Files that define public functions *should* include the header file(s) containing their prototypes. This allows the compiler to identify inconsistencies between the prototype and the function definition. Example: in ex_private.h:

```
// ---------------- Prototypes/Macros       (do not remove) ---------------
extern  int32     ex_intfunc(void);
```

Note *Header files use the keyword "extern," NOT the macro PUBLIC, because this is more appropriate.*

2.6.4.2 Private Function Prototypes

Private function prototypes *should* be at the top of the *.c file, to serve as forward declarations for functions appearing later in the file. Declare private functions with the macro PRIVATE (defined as "static" in comtypes.h). Example in ex_main.c:

```
// ---------------- Private Prototypes/Macros (do not remove) -------------
PRIVATE int32     intfunc(void);
```

2.6.4.3 Function Formal Parameters

Formal parameters are the parameter definitions in prototypes and function definitions.

You *should* qualify *pointer* parameters to functions with the following modifiers to help create self documenting code: const, OUT, INOUT. These modifiers indicate whether a parameter is an input-only parameter, output-only parameter, or both, and are described below:

const is ANSI C, and, when placed before the type, specifies that the target of the pointer is an input to the function and is not modified by the routine. The compiler verifies that the function does not modify the target of the pointer. NOTE: for other uses of const refer to your favorite C reference book.

OUT specifies that the target of the pointer will be modified by the function.

INOUT specifies that the target of the pointer is an input to the function, and will also be modified by the function as an output.

Do ***not*** use "IN", which is now deprecated (and replaced by "const").

The macros OUT and INOUT are defined as nothing in comtypes.h.

By definition of the C language, non-pointer parameters are always in-only, and *should not* be tagged (tagging would just be noise).

```
extern int32 ex_proc(      int32 iCount,
                     const int32 *int_ptr,
                     OUT   char  *result_ptr);
```

Thus, by simply looking at the prototype, it is clear which parameters are modified in the function.

You *should* always use "void" in the parameter list when creating a prototype for function that takes no parameters. For example,

```
extern  int32   ex_func(void);
```

is a prototype for a function with no parameters.

```
int32 func();
```

defines a function with no prototype, and should not be used, because the compiler cannot check that no parameters are passed to it.

2.7 Typedefs

All typedefs *should* be defined with at least one lower case letter (we reserve all-upper-case symbols for constants.)

Structures *should* be typedef'd to allow users to reference the typedef name directly. Typedefs for structures in global include files *should* be named with the module prefix.

Use the project predefined typedefs in place of most of the standard C data types. This makes it easier to port code between processors that may have different definitions for the standard types. The only allowed standard type is 'char', and only for actual character data, not for short integers. For example,

```
PRIVATE int32 function (int16 msg_size, uint32 transfer_id)
{
}
instead of:  PRIVATE int function (int msg_size, unsigned int transfer_id)
             {
             }
```

The following types are defined in comtypes.h:

Name	Definition
uint8	8-bit unsigned integer
int8	8-bit signed integer
uint16	16-bit unsigned integer
int16	16-bit signed integer
uint32	32-bit unsigned integer
int32	32-bit signed integer
nbo16	16-bit integer in Network Byte Order (MSByte first)
nbo32	32-bit integers in Network Byte Order (MSByte first)
unbo16	unsigned 16-bit integer in Network Byte Order (MSByte first)
unbo32	unsigned 32-bit integers in Network Byte Order (MSByte first)
bool	32-bit signed integer (with 2 values 0=false or 1=true)

Note *The "nbo" types are defined as integers, but are used for documentation. You* MUST *explicitly code any byte swapping necessary to maintain "nbo" variables.*

?? The one exception to using project defined data types is in parameters to shell functions. Shell functions are callable from the vxWorks Shell, and are expected to use 'ints' as parameters.

2.8 Variables

In variable declarations, the variable names *should* line up in the same column when possible. This makes it easier for the developer to distinguish between the types, and the names (especially, when there are many variables defined).

A variable declaration *should* contain a comment describing its purpose if the name is not self-explanatory. The comments for multiple variables *should* be lined up in the same column for readability.

Function variable declarations *should* be indented one level from function name. This helps to clarify where functions begin.

```
// ---------------- Constants               (do not remove) ---------------
#define   PI  3
:
:
PUBLIC ut_status ci_circle_draw (  // Returns status of draw operation
    uint32 radius )             // Radius of circle, in inches
{
    const uint32    area=2*PI*radius;
    ut_status       status;           // Status returned
    uint32          x point;
    uint32          y_point;          // for calculating coordinates in the circle

    :
    :

} // ci_circle_draw
```

2.8.1 Variable Names

Variable names *should* be as self-explanatory as possible. Cryptic variable names make the code harder to understand to maintainers. On the other hand, code filled with 30-character variables for everything is very difficult to read. In general, the more commonly used a variable is, the quicker a reader becomes familiar with it, and the shorter the name can reasonably be.

There *should* be no single character variable names. Single character variable names are hard to search for in code and not descriptive enough to be useful.

Index variables *should* be named consistently with the item they are indexing (i.e. board_idx).

2.8.2 Variable Prefixes and Suffixes

The standard prefixes for functions are defined in the Section 2.6.3: Function Naming. The standard prefixes for files are defined in Section 2.5: File Layout

There are several standard variable name suffixes that can make reading unfamiliar code easier. In order for these suffixes to be useful, they *should* be used consistently throughout the software. Below is a list of our standard suffixes:

Suffix	Meaning
_ptr	variable is a pointer
_p2p	variable is a pointer to a pointer

Additionally, the following suffixes *may* also be used to help clarify the variable usage.

Suffix	Meaning
_str	variable is a character string
_arr	variable is an array
_fp	variable is a file pointer
_s	identifies a structured data type
_t	identifies a typedef'd type (such as an enum)

Multi-word variable names *may* use underscores or capital letters to segregate the words. Either method is readable (for example, board_index and boardIndex are both acceptable). Developers *should* use the same style consistently within a single module.

2.8.3 Global/Shared Definitions

Global variables are those which are defined in one module, but are accessible from other modules as well. Their use is strongly discouraged, because code is much more maintainable and less error prone if direct access to any variable is restricted to a well defined interface within a single module. In general, a PRIVATE variable *should* be defined within the module and a set of PUBLIC functions for manipulating that variable (i.e. an API) *should* be provided by that module, including an initialization function when necessary. PRIVATE variables are discussed in the section on "Non-Global Definitions" which follows.

Shared variables, while similar to global, are visible to multiple files but are only used within a module. The concept of **shared** is at times hierarchical and may be shared across multiple directories. Shared variables *should* always be defined at the **lowest layer possible** based on their intended use. While use of global variables is rarely justified, performance considerations or code complexity sometimes justify the use of a variable which is **shared** among files within a module. An example is configuration data used by functions in the packet forwarding speed path. The declaration of a shared variable *should* be placed in a header file, just like an extern function declaration. The intent is to provide a well defined interface to the variable, even though it is technically global. Most accesses to the variable *should* be read-only. The variable's value *should* be changed in as few places as possible.

2.8.4 Local Definitions

File local static variables shared by more than one function in a file *should* be declared at the top of the file with the PRIVATE macro (because the C-standard "static" keyword is a misnomer here). Function local static variables *should* be declared in that function with the 'static' reserved word, because "static" is appropriate here: it defines a storage class, not a scope modifier.

2.8.5 Bit Fields

You *MUST* declare bit fields explicitly as either **signed** or **unsigned**, because C does not define the default, and different compilers may make different choices. You *may* want to use bit fields for private data structures where memory efficiency is important, and code speed is less so. Note that bit field packing order is implementation-specific, and so bit fields *MUST NOT* be used for inter-processor communication.

Example:

```
struct         // Explanatory comment here ...
{   signed   field1    :3;
    unsigned field2    :3;
    unsigned field3    :10;
} big_array[100000];
```

Don't make assumptions about alignment or bit order of bit fields, because such things are undefined in ANSI C, and not portable.

2.9 Constants & Enums

Let the compiler do the work: when coding compile time constants which are calculated, write them as an arithmetic expression that conveys their meaning and derivation:

```
#define TDC_SLOT_BITS  (TDC_SLOT  <<  12)
```

You *may* use 'const' instead of #define for most compile time constants. "const" variables allocate storage, and have an address. The advantage to const is that it will appear in the symbol table, and can be examined with the debugger.

You *should* use upper case to name compile-time constants (including 'const', #define, and enums):

```
#define MAX_DS3S 4
const uint8 MAX NOISE MARGIN = 32;
typedef enum
{   ON = 1,
    OFF
} switch_states;
```

You can use constant structures to initialize default values in structure variables:

```
const msg_addr NULL_ADDR = { AL_NO_COMP, 0, 0, 0 };
```

You can then use NULL_ADDR in a source file:

```
PRIVATE void sc_discover (void)
{
    my_addr = NULL_ADDR;
...
}
```

One unfortunate limitation to 'const's is that compilers do not allow you to declare arrays whose bounds are specified by const if the array is static.

2.9.1 Run Time Constants

C also allows you to declare run-time constants. Run-time constants are objects that are initialized to some value (computed at run-time) but are not allowed to change their value after definition. Run-time constants *should not* use upper case (as are compile-time constants) since they are more akin to variables than to compile-time constants. You are encouraged to use run-time constants since they can improve readability and prevent errors. For example:

```
PRIVATE void foo(char *str_ptr)
{
    const uint32 length_plus_pad = strlen(str_ptr) + pad;
    ...
```

2.10 Statement Formatting

2.10.1 Indentation

Each indent level is 4 spaces. For example:

```
if ( ... )
{
    while ( ... )
    {
        if ( ... )
        {
        }
    }
}
```

2.10.2 Tabs

Source code *may* include tabs, however, text editors *should not* be reconfigured to set Tabs every 4 spaces. Editors *should* be configured with hard Tabs every 8 spaces. This is because general utility routines (search, browse, etc.) often do not have settable tab stops. Any ASCII <tab>s in source code *MUST* use 8-space tab stops to ensure interoperation with standard searching and browsing utilities (which define default tab stops to 8-spaces).

Developers *may* use spaces for all indents, or configure their editors to replace tabs with spaces.

2.10.3 Line Length

Lines *should not* exceed 80 characters. Many utilities (browsers, diff tools, etc.) wrap lines that are greater than 80 characters which makes code with long lines difficult follow.

2.10.4 Braces

Paired braces *should* always line up in the same column and the column is in alignment with the controlling statement**.** This makes it obvious where a block of code begins and ends. This is especially true for code that has several levels of nesting. [This is *not* K&R-like. Note that K&R is inconsistent: function braces line up, but other statement braces don't.]

```
while (!done)
{
    if (idx == 5)
    {
        ...la de da;
    }
}
```

Braces *should* always be included around multiple line statements, even if the other lines are just comments. This keeps the code structure clear to the casual observer.

```
while (!done)
{
    if (idx == 5)
    {
        // Set flag to break out of loop
        done = TRUE;
    }
}
```

Braces *should* be used around conceptually big statements, even if they are physically small. This informs the reader that the block is significant. For example:

```
// The following construct conveys a small concept in the conditional statement:

idx++;
if (idx >= max_idx)  idx = 0; // circular increment

// But this construct conveys a major event:
if (idx > max_idx)
{
    ex_restart_board ();      // unrecoverable error!
}
```

2.10.5 Comments

Comments are an important part of any software. It is important that comments be used to enhance the understanding of code, not reiterate what is already obvious.

- Comments *should* be used freely. Things that may seem obvious today may not be obvious to someone maintaining the code a year from now.
- Long comment blocks *should* be separated from code by at least a line of white space.
- Use the newer "//" token to start comments that end at the end of the line of text. This is now standard in both C99 and C++.
- Comments *should* be on a line by themselves unless they are VERY short.
- Comments *should* be indented at the same level that the code is indented. This helps to maintain the logical flow of the software.

```
// Check for error condition before getting too far
if ( error )
{
    // Let operator know that the command wasn't executed
    oper_notify_error ( NOT_EXECUTED ) ;
}
```

- Developers should put comments on the closing brace of long code blocks, and the closing brace of every function.

```
if (db_access_is_ok(msg))
{
    while ( idx > 0 )
    {
        :
        :
    } // while (idx > 0)
} // if (db_access_is_ok(msg))
```

- In the case where an 'else' branch of an 'if' statement is far from its 'if ' (more than 10 lines, say), it *should* carry a comment which explains the condition causing the else clause to execute. e.g.,

```
if (db_access_is_ok(msg))
{
    :
}
else // db access is not OK
{
    :
} // if db_access_is_ok
```

2.10.6 Conventionalized Comments

Conventionalized comments are comments of a fixed syntax. These comments are used to flag questions, concerns, and sensitive code that a developer *should* be aware of when modifying the code. These comments *should* be used to flag returns in the middle of routines, taking/giving semaphores, enabling/disabling preemption, etc.

On pairs of complementary statements (such as taking/giving a semaphore), you *should* use conventionalized comments to highlight their special relationship. This comment *should* appear on the line of code, if feasible. In any case, such related comments *should* be lined up visually 'to the right of code' to indicate their relationship.

The structured comments identified thus far are:

Comment	**Meaning**
`EMBEDDED RETURN`	return in the middle of function
`DYNMEM alloc`	allocate dynamic memory
`DYNMEM free`	free dynamic memory
`INTERRUPT disable`	disable interrupts
`INTERRUPT enable`	enable interrupts
`PREEMPTION disable`	disable task preemption
`PREEMPTION enable`	enable task preemption
`fall through`	switch statement fall through (with no 'break')

`GOTO`	goto in the middle of a function
`SEMAPHORE take`	take a mutual exclusion semaphore
`SEMAPHORE give`	give a mutual exclusion semaphore
$$ <initials; date>	Used to flag inefficient code that you know may need to be made more efficient in the future. Should include the developer's initials and a date stamp when it was entered.
?? <initials; date>	Questionable code. Should include the developer's initials and a date stamp when it was entered.
`TBR` <initials; date>	Code that needs to be reviewed before it is finished. Should include the developer's initials and a date stamp when it was entered.
`begin change, end change`	3rd-party software (documented in 3rd-party software chapter)

An example usage would be:

```
ut_mutex_take(...)                    // <SEMAPHORE> take
// ?? Does this code work in the case of interrupts?
if(...)
{
    // $$ <LMM;4/11/99> Can we simplify this code?
    ut_mutex_give(...)                // <SEMAPHORE> give
    return;                           // EMBEDDED RETURN
}
:
ut_mutex_give(...)                    // <SEMAPHORE> give
```

2.10.7 Operators

The many levels of precedence in C can be confusing, and few people know them by heart. However, too many parentheses obscure code. A reasonable compromise is to expect everyone to know the following 4 precedence groups:

`f( ), a[ ], s.b, s->b`	Function arguments, array subscripts, structure members, and pointer to structure members
`*, /, %`	Multiply, divide
`+, -`, and their ilk	add, subtract
`&&, ||`	logical AND, logical OR (not equal precedence)

Of course, everyone knows that assignment is lowest of all. Consider using parentheses to indicate other precedences, rather than relying on the pre-defined relationships.

You *should* use assignment operators (`+=, -=, *=, /=, %= &=, ^=, |=, <<=`, and `>>=`) to improve readability (bear in mind that 'x *= y +1;' is equivalent to 'x = x * (y + 1);' not 'x = x*y + 1;').

You *should* use the ternary operator (?:) when it aids readability, thus

```
return (foo > 3) ? UT_OK : UT_ERROR;
instead of:  if (foo > 3)
                 return UT_OK;
             else
                 return UT_ERROR;
```

because it is clearer that the point of the code is to return something, and the undecided point is the value to return.

You *should* always consider readability when applying any of these rules. For example:

```
// Breaking out this logic makes it clear
if (foo > 3)    y *= 100;
else            y *= 128;
instead of:  y *= (foo > 3) ? 100 : 128;  // this is less clear
```

2.10.8 Assignments within Other Statements

You *should not* put assignments within other statements. Doing so can produce unexpected behavior, especially in arguments passed to macros. This also applies to some degree to the ++ and -- operators. Making the increments explicit can save hours of looking through code for counters gone awry. Therefore, use embedded ++ and -- sparingly and cautiously.

```
// NOTE:  The 'min' macro often uses the parameters multiple times
new = min(old[idx], new);
idx++;
```

instead of:

```
new = min(old[idx++], new); // doesn't work!
```

2.10.9 White Space

To keep code clear, feel free to use white space between operators, braces, and parenthesis. Code that is too dense is often difficult to read.

```
// Leave white space between operators
error = op_set_privileges();

// Use additional white space as appropriate for readability
if ( cmd > max cmd )
{
    // Let the operator know that the command wasn't executed
    op_notify_error( NOT_EXECUTED ) ;
}
```

2.10.10 Switch Statements

In a switch statement, if code needs to fall through from one case to another, there *should* be a comment starting with "fall through" that describes the intentional fall through. This makes it clear the missing break is intentional. Lint is configured to flag a fall through without a comment as an error. NOTE: Multiple "cases" for the same code do not need fall through comments.

When "switching" on an enumeration value, you can have the GNU compiler verify that all possible values are covered by omitting the "default" case. With no default, and if some enumeration values are missing, the compiler produces a warning. Include a comment explaining your use of this feature:

```
typedef enum   {ABC, DEF, GHI} alpha_enum;
:
PRIVATE alpha_enum alpha_var;
:
switch (alpha_var)
{
    case ABC:                // these two 'cases' go together
    case DEF:                // no comment needed
        oper_notify_error (error_code);

        // fall through to next step after message is sent
    case GHI:
        io_close_contacts (); // activate error output
        break;

    // NOTE:  no default to insure all cases are covered
}
```

Switch statements on non-enumerated values *MUST* have a default case. If you don't expect the default case to ever be used, consider putting in an assert (or equivalent) statement.

```
#define ERR_OPER_WARNING  1
#define ERR_OPER_ERROR    2
#define ERR_FATAL_ERROR   3
:
PRIVATE int32 error_code;     // takes on a variety of error values
:
switch ( error_code )
{
    case ERR_OPER_WARNING:    // these two 'cases' go together
    case ERR_OPER_ERROR:      // no pragma or comment needed
        oper_notify_error (error_code);

        // FALL THROUGH to next step after message is sent
    case ERR_FATAL_ERROR:
        io_close_contacts (); // activate error output
        break;
    default:                  // no action on other cases
        break;
}
```

2.10.11 Checking Error Returns

Error return values are provided to allow a calling procedure to make an informed decision as to appropriate action to take when unexpected results occur. The primary goal of the error checking is to ensure the integrity of the system. It *may* not be necessary (or desirable) to check every status.

Code *may* ignore error return values if it cannot do anything meaningful with the error. Sometimes developers put a '(void)' statement in front of functions where the return value is intentionally ignored. Because this clutters the code, we prefer *not* to do this.

For example we prefer

```
ut_free(msg_ptr);      // ignore return value
instead of: (void)ut_free(msg_ptr);   // ignore return value
```

2.10.12 Return Statements

Attempt to structure your code to minimize the indenting impact of status checking. The idea here is that the indenting of the code *should* reflect the 'normal' processing. Rather than having extensive indenting which reflects error checking, consider returning when an error is detected (after performing appropriate 'cleanup'). When you do so, you *should* place "// EMBEDDED RETURN " on such lines:

```
PRIVATE void foo(uint32 size)
{
    ut_status status;

    if (size > MAX_POOL_SIZE) return;          // EMBEDDED RETURN

    status = ut_alloc(&ptr, size, LOCAL_POOL) // <DYNMEM> alloc
    if (status != UT_OK) return;               // EMBEDDED RETURN
    .
    .
    .
    status = ut_send_msg(q_id, ptr, format, SIZE,…);
    if (status != UT_OK)
    {
        ut_free(ptr);                          // <DYNMEM> free
        return;                                // EMBEDDED RETURN
    }
    :
    :
    ut_free(ptr);                              // <DYNMEM> free
}
```

2.10.13 goto

Avoid goto's for normal control flow, but there are at least 3 cases where a goto statement *may* be used effectively: error handling, nested loop exiting, and switch case combining. You *should* comment a goto to alert the reader of possible unexpected control flow.

Error handling: The first reasonable use of goto is for handling errors without adding confusing "if" nesting:

```
PRIVATE void foo(void)
{
    if(!(ptr = malloc(size) )          // <DYNMEM> alloc
        return;                        // no action possible   EMBEDDED RETURN

     :
    if( ... error ...)  goto cleanup_1;      // GOTO
     :
    if(!(ptr2 = malloc(size) )         // <DYNMEM> alloc
        goto cleanup_1;                // GOTO
     :
    if( ... error ...)  goto cleanup_2;
     :

cleanup_2:
    free(ptr2);                        // <DYNMEM> free
cleanup_1:
    free(ptr);                         // <DYNMEM> free
} // foo()
```

Note that this is emulating "by hand" what C++ provides automatically in calling class object destructors before exiting a procedure.

Nested loop exiting: The continue statement allows a jump to the end of a loop, but sometimes you want to jump to the end of an outer, enclosing loop. Goto can be used for that:

```
    for(trial = 11;  trial < 121;  ++trial)
    {   for(divx = 0;  divx < divlim;  ++divx)
        {   if(trial % prime[divx] == 0)
                goto next_trial;             // GOTO. not prime
        }
        printf("%d is prime\n", trial);
next_trial:
    }
```

Switch case combining: If two or more cases have identical "finish-up" processing, it is sometimes effective to use a simple goto to share the common code. The goto *should* jump forward, so that there is no confusing backward goto.

```
    switch(process_type)
    { case ODDBALL:
        ...
        break;

      case COMMON_1:
        ... code specific to COMMON_1
        goto common_finish;                  // GOTO

      case COMMON_2:
        ... code specific to COMMON_2
common_finish:
        ... code common to both cases
        break;
    }
```

2.10.14 #if Pre-Processor Directive

Use "#if 0" to "comment out" large sequences of code and pseudo-code, rather than "// ... */", and include a comment on the #if line explaining why the code is retained in the source. #if avoids problems with comment nesting in the if'd-out code, and is itself nestable.

Beware of too many compile-time switches, i.e. #ifdef. Too many #ifdefs can make code hard to follow, especially if they are nested.

Beware of comparing macro constants in #if statements, because all undefined macros in C-pre-processor statements are replaced by "0" before expression evaluation. For example:

```
#if CPU == POWER_PC
```

will evaluate to true if neither CPU nor POWER_PC are defined, because `0 == 0`. If you forget the header file, the above line produces no error, but includes POWER_PC code, which is probably wrong. As an alternative, try:

```
#if defined(CPU) && CPU == POWER_PC
```

2.10.15 #error Pre-Processor Directive

Use "#error <description of error>" in an "#if ….. [#else …..] #endif" sequence of statements to report an invalid combination of definitions detected at compile time. For example, use

```
#if MAX_BC_INTQ_SIZE < 6600
#error: MAX BC INTQ SIZE is too small
#endif
```

2.10.16 Testing for Null Pointers

To test for NULL pointers, you *may* use either "`if (!ptr)`" or "`if (ptr == NULL)`". Note that the boolean interpretation of a pointer can be thought of as "pointer is valid."

If a function accepts (void *) pointers, don't cast actual arguments to it with (void *), (void **), etc. It's just noise. For example:

```
PRIVATE struct complex *complex_ptr;

PRIVATE func(void *ptr);

func(complex_ptr);            // no cast
instead of:  func((void*)complex_ptr);       // unnecessary cast
```

2.10.17 Use sizeof() and offsetof()

For pointer manipulation, code that uses sizeof() and offsetof() is more readable, and more portable, than hard-coding constants.

Also use sizeof for buffer manipulation. This includes copying, allocating, etc. For example:

```
char          temp_buffer [MAX_LINE_LENGTH];

memset (temp_buffer, 0, sizeof(temp_buffer));
instead of:  memset(temp_buffer, 0, MAX_LINE_LENGTH);
```

For referencing structures from a void pointer, cast the pointer to your structure type, and dereference that normally:

```
((struct_type *)ptr)->field
instead of:  *(int32 *)(ptr+4)
```

For unusual cases where you need the actual offset of a structure field, use the `offsetof()` macro, from `stddef.h`:

```
    char       *new_ptr, *old_ptr;
    new_ptr = old_ptr + offsetof(struct_type, field);
```

If you need to step a pointer over a given data type, you *should* do it symbolically:

```
    char       *xyz_ptr;
    xyz_ptr += sizeof(int32);
instead of:  xyz_ptr += 4;
```

2.11 Macro Functions and Inline Functions

This section covers macro *functions*. Macro *constants* are covered in Section '2.9 Constants & Enums'.

Macro functions *should* be declared using lower case names (that comply with our function naming conventions). This makes it easier to rewrite a macro as a function (or vice versa), with minimal changes to the application code that uses the macro.

Arithmetic macro function definitions *should* enclose all the arguments in parentheses, and themselves be enclosed in overall parentheses:

```
#define plus_two(x)    ((x)+2)
idx = plus_two(idx << 1)*3
```

If either set of parentheses are missing in 'plus_two(x)', the above code behaves unexpectedly.

2.11.1 Multi-statement Macros

There are some problems when using multi-line macro functions. Note that inline functions solve all these problems, and more. Because macros are simple text substitutions, when a macro includes multiple C-language statements, you *MUST* be careful to avoid the famous "if(x) macro;" problem. For example:

```
#define init(x, y)     x = 1; y = 1

if(...)  init(a, b);
```

probably does not do what is expected. The intent is for 'a' and 'b' to both either be initialized, or both be not initialized. However, the above code expands into the following (indented for clarity):

```
if(...)
        a = 1;
b = 1;
```

'b' is *always* initialized. A simple fix for assignment statements is to use the comma operator to reduce the macro to a single C-language statement:

```
#define init(x, y)     (x = 1, y = 1)
```

A similar problem occurs if the macro itself contains an 'if' statement:

```
#define reset(x)        \
        if(reset_allowed)  x = 1

if(time_to_reset)
        reset(x);
else
        printf("not time to reset");
```

The above code does not work as it appears, because it expands to this (indented for clarity):

```
if(time_to_reset)
        if(reset_allowed)
                x = 1;
        else
                printf("not time to reset");  // wrong
```

The above problem can be fixed by re-coding the macro thus:

```
#define reset(x)        \
        if(reset_allowed)  x = 1;      \
        else (void) 0          // '(void) 0' avoids null-statement warning
```

A general approach for solving more complex multi-line macros is to enclose the entire macro in an if(1) {} else wrapper. Again note that an inline function handles this much more cleanly:

```
#define multimac                \
        if(1)                   \
        {       line 1; \
                line 2; \
                ...;            \
        }                       \
        else (void) 0
```

2.11.2 "inline" Functions

As an extension to ANSI C, some compilers allow the use of an 'inline' modifier for functions (this is standard in C++). This requests (but does not require) the compiler to treat the function much like a macro, expanding the function "inline" for each call to it. The function thus incurs no function-call overhead (like a macro), but retains all the benefits of parameter type checking and conversion (which a macro forgoes).

We define a macro 'INLINE' that *should* be used for defining inline functions.

Since most compilers today are both C and C++, and since "inline" is a standard C++ keyword, most C compilers accept "inline." If we ever needed to port code which uses "INLINE" to a compiler which didn't support it, we could easily #define INLINE with an empty definition, and the code would work without further modification.

2.12 Network and Inter-Processor Communication

2.12.1 Packing

Most compilers generate code that is efficient for the target processor. To generate such code, compilers may add "padding" bytes to structures to align integers to the alignment of the processor.

We require our compilers to support packing of structures with no padding, when needed. To achieve packing, we add the PACKED_ATTR macro to the definition of each field in the structures. Example:

```
typedef struct
{
    msg_al_addr   dest    PACKED_ATTR;
    msg_al_addr   src     PACKED_ATTR;
    unbo16        pid     PACKED_ATTR;
} msg_al_hdr;
```

PACKED_ATTR is defined in a project-standard header. For the GNU compiler the following definition of PACKED_ATTR is used.

```
//-----------------------------------------------------------------
  Define a macro for setting the packed attribute on structures
  that are shared across interfaces.
  -----------------------------------------------------------------*/
#define PACKED_ATTR  __attribute__ ((packed))
```

NOTE: This definition may need to be changed to support new compilers/processors:

NOTE: Not all compilers support this construct. For example, some compilers must specify alignment at the beginning of the header file using a pragma.

To insure that data is accessed consistently across processors, developers *MUST* pack such data structures using the appropriate method for the compiler they are using.

2.12.2 Byte Order Independence

For data that is transported between processors, data *should* be sent in Network Byte Order. Don't assume a byte order (little- vs. big-endian) in the source code.

When defining integer fields in structures that contain data stored in Network Byte Order (NBO), use the following defined typedefs:

```
    nbo16    16 bit integer stored in NBO
    unbo16   16 bite unsigned integer stored in NBO
    nbo32    32 bit integer stored in NBO
    unbo32   32 bit unsigned integer stored in NBO
```

These typedefs are used to alert the developer that this data *MUST* be translated back into host byte order before it is evaluated.

If the structure that the data is stored in is known to meet the alignment requirements of the host processor (see the alignment section), use the standard *sockets* macros/functions for byte order independence. The 4 sockets macros for byte order independence are these:

```
unbo32 htonl(uint32 hostlong);     convert host to network long
unbo16 htons(uint16 hostshort);    convert host to network short
uint32 ntohl(unbo32 netlong);      convert network to host long
uint16 ntohs(unbo16 netshort);     convert network to host short
```

They are documented in standard Unix texts on network processing.

Note that these macros are defined for unsigned, but work on signed integers as well..

2.12.3 Byte Alignment

The main issue with alignment is that a 16 (or 32) bit processor may only support loading integers if they are stored on a 16 (or 32) bit boundary. Some processors support (in hardware) loading integers that are unaligned, but access to unaligned data may be slower.

The x86 supports unaligned integer references. The compiler for the x86 supports packed data structures on an 8-bit boundary.

The 68302 does not support unaligned integer references. The compiler for the 68302 will generate correct code for packed data structures, but it assumes that the top of the data structure is aligned to a 16-bit boundary.

Therefore, data structures that are exchanged between different processors (such as message buffers) *MUST* follow the following rules:

1. Data structures *MUST* be defined using the PACKED_ATTR attribute to guarantee their size on both processors.
2. Each programmer *MUST* guarantee that data buffers referenced by common code (such as message buffers) start on a memory location that is aligned to the processors needs (either 16-bit or 32-bit). For example, for messages coming from off board, this might mean making sure the local buffer that the message data is copied into starts on an aligned memory address. NOTE: malloc( ) guarantees byte alignment for the processor.

2.12.4 No Inter-Processor Bit Fields

You *MUST NOT* use bit fields for data structures that are transported between processors. Instead of C-language bit fields, use integers and masks to define bit fields on inter-processor interfaces, because of the ambiguities in the ANSI C standard for bit fields. (In ANSI C, the order of the bits is undefined, the alignment boundary following bit fields is undefined, and whether bit fields are signed or unsigned by default is undefined.)

2.13 Diagnostic Code

2.13.1 ASSERT

NOTE WELL: ASSERT doesn't follow our naming convention. This is an oversight, not to be emulated.

When writing code, people often make assumptions about values of variables: things like NULL pointers, variables in range, etc. During debug, it is often helpful to include code to validate these assumptions, and notify you of any discrepancies. For production code, however, you may not want to incur the memory and speed penalty of extensive assumption checking. ANSI Standard C provides an "assert" macro, and an associated "`assert.h`" to support these needs. The "assert" macro is *conditionally* defined according to the needs of the software being developed: during debug, it is defined to validate assumptions; for production code, it is defined as a null macro. Therefore, you can include or exclude a wide range of debug code with a single conditional definition.

These guidelines customize the standard "assert" macro with the ASSERT macro (upper case). When your code is compiled in DEBUG mode, and an assumption fails at run-time, ASSERT prints an error message and notifies you (how/where the error message is displayed will vary depending on the target device). Typically, the error message contains the file name and line number where the error occurred (this information is provided by the C preprocessor). For example:

```
PUBLIC void ut_kill(INOUT char *ptr)
{   ASSERT(ptr != NULL, "'ptr' is NULL!");
    ut_free(ptr);
}
```

The ASSERT macro looks something like this:

```
//---------------------------------------------------------------------
   When compiled in DEBUG mode, the ASSERT macro evaluates the
   conditional. If the conditional passes, nothing happens.  If the
   conditional fails, ASSERT prints an error message with the file
   name, line number, and custom error string, then continues with the
   program.

   When NOT in DEBUG mode, the ASSERT macro is a comment and does
   nothing.

   NOTE:  The trailing ELSE is required to make the macro appear as a
   simple C statement to the invoking code.
---------------------------------------------------------------------*/
#ifdef DEBUG
    #define ASSERT ( condition, msg )                                  \
        if ( !(condition) )                                            \
        {                                                              \
            fprintf ( stderr, "ASSERT (" #condition ") FAILED "        \
                              "[File " __FILE__ ", Line %d]:\n\t%s\n", \
                              __LINE__, (msg) );                       \
        }                                                              \
        else (void) 0        // required to avoid compiler warning
#else  // DEBUG
    #define ASSERT ( c, m )   // empty
#endif // DEBUG
```

2.13.2 Debug Code

Some debug code or assumption checking code is more involved than ASSERT can handle. For these cases, you can use #ifdef DEBUG (just like the ASSERT macro definition does) to selectively include arbitrary debug code. Such debug code can be easily identified using text search utilities. Also, put some description in the comment about what the debug code is trying to accomplish.

Flag temporary or questionable code with "??", so it's easy to find with a text search.

2.14 Tips & Gotchas

This section contains some tips for writing C code that are not strictly guidelines, but may be useful to developers. Some of these are items that may cause you (or someone else) to spin their wheels for a time.

2.14.1 scanf() Problems

Adapted from: http://c-faq.com/stdio/scanfprobs.html:

Q: Why does everyone say not to use `scanf()`? What should I use instead?

A: `scanf()` has a number of problems. Use `fgets()` & `sscanf()`, and sometimes `fscanf()`, if you check its return value.

`scanf()`'s `%s` format has the same problem that `gets()` has: it's hard to guarantee that the receiving buffer won't overflow. More generally, `scanf()` is designed for relatively structured, formatted input (its name is in fact derived from ``scan formatted''). It can tell you only approximately where it failed, and not at all how or why. You have very little opportunity to do any error recovery.

Yet a well-designed interface will allow for the possibility of reading just about anything – not just letters or punctuation when digits were expected, but also more or fewer characters than were expected, or no characters at all (i.e. just the RETURN key), or premature EOF, or anything. It's nearly impossible to deal gracefully with all of these potential problems when using `scanf()`; it's far easier to read entire lines (with `fgets()` or the like), then interpret them, with say, `sscanf()`. If you do use any `scanf` variant, be sure to check the return value to make sure that the expected number of items were found. Also, if you use `%s`, be sure to guard against buffer overflow.

Note that criticisms of `scanf` are not necessarily indictments of `fscanf` and `sscanf`. `scanf` reads from `stdin`, which is usually an interactive keyboard and is therefore the least constrained, leading to the most problems. It's perfectly appropriate to parse strings with `sscanf` (as long as the return value is checked), because it's so easy to regain control, restart the scan, discard the input if it didn't match, etc. When a data file has a known format, on the other hand, it may be appropriate to read it with `fscanf`.

2.14.2 Huge Object Files

The compiler generates uncompressed initialization for the entirety of an initialized data structure, even if you only initialize a tiny fraction of it. For example:

```
PRIVATE int32 data[100000] = {1};
```

generates 400 kbytes of object code (100,000 * sizeof(int32)), because the compiler initializes the array with "1" in its first element, and zero in all its subsequent elements. The initialization to zero is required by the C-language standard. Such an initialization is probably better coded as:

```
PRIVATE int32 data[100000];

// somewhere in initialization
data[0] = 1;          // initialize ....
```

2.14.3 Null Procedure Bodies

The compiler will quietly generate null procedure bodies for procedures whose definition is terminated by a semi-colon. This seems like a compiler bug, to me [ELM].

```
PUBLIC void foo(void)
{
   ... lots of useful code that you would really like
}; // OOPS
```

2.14.4 'Make' can compile wrong file

A spurious C file in your compile directory (e.g., ..\testdir) will be compiled in preference to the correct file in its rightful directory. Close inspection of the compile invocation will show that the file is not preceded by a path name.

2.14.5 Comparing Macro Constants

Beware of comparing macro constants in #if statements, because all undefined macros in C-pre-processor statements are replaced by "0" before expression evaluation. For example:

```
#if CPU == POWER_PC
```

will evaluate to true if neither CPU nor POWER_PC are defined, because 0 == 0. If you forget the header file, the above line produces no error, but includes POWER_PC code, which is probably wrong.

2.14.6 Misleading vsprintf output

With Tornado1.0.1, if you pass vsprintf (and probably its brethren printf and sprintf) a pointer to a string which contains non-printable characters, you will get "pointer is NULL" appended to the formatted result (even though the pointer is not null).

2.14.7 Use 'const' for strings instead of #define

[I question the validity of this. -ELM 5/2010] In *.c files, consider 'const' to declare strings instead of #define. The reason is that when you use 'const' to declare a string, the variable may be used in multiple places without replicating storage for the entire string (as opposed to a macro where the entire string may be replicated whenever it is referenced). Some compilers will combine common strings, but explicitly declaring const strings insures it.

3. C++ Coding

C++ provides better tools to achieve code quality, maintainability, and portability. You can take advantage of most C++ features without a "paradigm shift" or change in the good design practices you already know.

3.1 C++ Coding Guidelines

- Just use the new C++ language capabilities to help you write better code. Designing complex class hierarchies is usually unnecessary, and can be confusing.
- Consider using inline functions and templates instead of macros. Both are more flexible, and provide greater error detection.
- Write casts as function calls to the target data type; it's much more clear:

```
x = int(y) + int(z)
instead of:    x = (int)y + (int)z
```

- Classes should not be typedef'd, since the classname is already effectively a typedef.
- Use "class" only when you really mean it. Use "typename" in templates, unless it must be a class type:

```
template <typename T> func(T arg) ...
template <typename T> class xyz
{ ... }
instead of:    template <class T> ...
```

- Use "explicit" on single-argument constructors to avoid implicit type conversion by the compiler. If you really want implicit type conversion, comment the constructor as such:

```
class xx
{   explicit xx(int x);
    xx(double y);             // implicit converting constructor
}
```

- Use "mutable" very sparingly. "mutable" is intended to allow a class method to modify a const class object in an invisible way, i.e., a way that the class user won't see. Its use is very rare. In most cases, just admit openly that the class object is changing, by omitting "const" from the object's declaration and the class method declaration. Don't use mutable to defeat the usual protection mechanisms of "const".
- "Mutable" can be thought of as saying that a class data member is "never const", even if the class object is. This sometimes allows compiler optimizations that would be disallowed without "mutable".

3.2 Object Oriented Programming

This is a much used and abused term, with no definitive definition. The goal of Object Oriented Programming (OOP) is to allow reusable code that is clean and maintainable. The best definition I've seen of OOP is that uses a language and approach with these properties:

- **User defined data types**, called **classes**, which allow (1) a single object (data entity) to have multiple data items, and (2) provide user-defined **methods** (functions and operators) for manipulating objects of that class. The data items are sometimes called **properties**. Together, data items and methods are **class members**.
- **Information hiding:** a class can define a public interface which hides the implementation details from the code which uses the class.
- **Overloading:** the same named function or operator can be invoked on multiple data types, including both built-in and user-defined types. The language chooses which of the same-named functions to invoke based on the data types of its arguments.

- **Inheritance:** new data types can be created by extending existing data types. The new **derived class** inherits all the data and methods of the existing **base class**, but can add data, and override any methods it chooses with its own, more specialized versions.
- **Polymorphism:** this is more than overloading. Polymorphism allows derived-class objects to be handled by (often older) code which only knows about the base class (i.e., which does not know of the existence of the derived class.) Even though such code knows nothing of the derived class, the data object itself insures calling proper specialized methods for itself.

In C++, polymorphism is implemented with virtual functions.

OOP does not have to be a new "paradigm." It is usually more effective to make it an improvement on the good software engineering practices you already use.

3.3 Type Casting

From http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=134, summary:

"C-style cast is neither safe nor explicit enough, as I have shown. It disables the compiler's type-safety checks, its syntactic form doesn't express the intended conversion clearly, and it cannot perform a dynamic cast. For all these reasons, you should avoid using it in new code.

Instead, use

- static_cast for safe and rather portable casts,
- const_cast to remove or add only the const/volatile qualifiers of an object,
- reinterpret_cast for low-level, unsafe and nonportable casts.
- dynamic_cast for conversions that must access the dynamic type of an object and RTTI [Run-Time Type Information] capability-queries."

4. Python Tips and Coding Guidelines

This chapter gives a brief introduction to Python, and why you might want to consider using it. It includes a simple, working program that demonstrates the 3 most common things to do with Python: command line arguments, reading files, and making plots. You can start with this program and modify it however you like.

4.1 Why Python?

Python is handy for smaller scripts, because:

- you don't need to declare variables
- variables are dynamically typed
- there's an interactive interpreter

Python has handy features:

- some handy string capabilities (but not as good as some claim)
- arbitrarily large integers
- sort-of classes (but no automatic parent-class constructor invocation)

As your program grows in complexity, I think Python becomes clumsier. However, some people think the opposite: that Python is good for large, multi-user developments.

4.2 Getting Started With Python: Quick Tips

4.2.1 Help on Installable Packages

You can get help even if you haven't loaded the package. For example,

```
help("pylab.legend")
```

gives you help, even if you haven't loaded pylab with something like

```
from pylab import *    or      import pylab
```

4.2.2 Strings, Lists, Tuples, and Sequences

These are commonly confused, and have weird and arbitrary properties and methods. Because they are so important, we summarize quickly here the commonly used features. (From docs.python.org/tutorial/datastructures.html)

A **string** is a sequence of characters. **Lists** and **tuples** are sequences of arbitrary objects. All are indexed with subscripts or slices, starting with 0 (up to length-1): e.g., "`s[0]`", "`s[len-1]`", "`s[start,end+1]`".

All are concatenated with the "+" operator, e.g. "`a+b`"

Lists are created with square brackets, e.g. "[1,2,3]". Lists can be changed:

`list.append(x)` Add an item to the end of the list; equivalent to `a[len(a):] = [x]`

`list.extend(L)` Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.insert(i, x)` Insert an item at a given position. The first argument is the index of the element before which to insert, so `list.insert(0, x)` inserts at the front of the list, and `list.insert(len(a), x)` is equivalent to `list.append(x)`.

list.pop`([i])` Remove the item at the given position in the list, and return it. If no index is specified, `list.pop()` removes and returns the last item in the list.

`len(list)` returns # objects in list

Tuples are created with parentheses, e.g. "(1,2,3)". Tuples and strings are immutable, which means they cannot be changed. Instead, you can assign a new tuple to a tuple variable (or string to string variable). There's usually no reason to use a tuple instead of a list, since lists are more flexible. However, www.python.org/doc/faq/general/ notes: "Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys."

A **sequence** is a generic name for a string, list, tuple (and a few other data types). Generally, one should avoid the word "sequence" and use either "string", "list", or "tuple" to be clear.

Dictionaries are created with curly braces, "{ }", not described here.

4.2.3 Common String Methods

From http://docs.python.org/library/stdtypes.html#string-methods:

`str.find(sub[, start[, end]])` Return the lowest index in the string where substring sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

`str.replace(old, new[, count])` Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

`str.split([sep[, maxsplit]])` Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified, then there is no limit on the number of splits (all possible splits are made).

If *sep* is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns `[]`. For example, `' 1 2 3 '.split()` returns `['1', '2', '3']`, and `' 1 2 3 '.split(None, 1)` returns `['1', '2 3 ']`.

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

4.2.4 A Simple Text Filter Example

A simple example of command line parameters and files:

```
""" Handmade simple file filter.  Manually edit for simple functions.

Reads the given file, editing each line, and writing to stdout
Strips all duplicate lines

"""

import sys              # argv[]

# Remove lines that start with numbers, and whose 1st 4 characters match the prev
line
f=open(sys.argv[1], "rU")   # r=read, U=universal newline

prev = ""
for line in f:
    if line == prev:
        pass                # could be filled in with something
    else:
        print line,         # the comma prevents extra \n at end

    prev = line

f.close()                   # we're done with 'f'
```

4.2.5 A Simple Example: Command-line Parameters, Files, Arrays, and Plotting

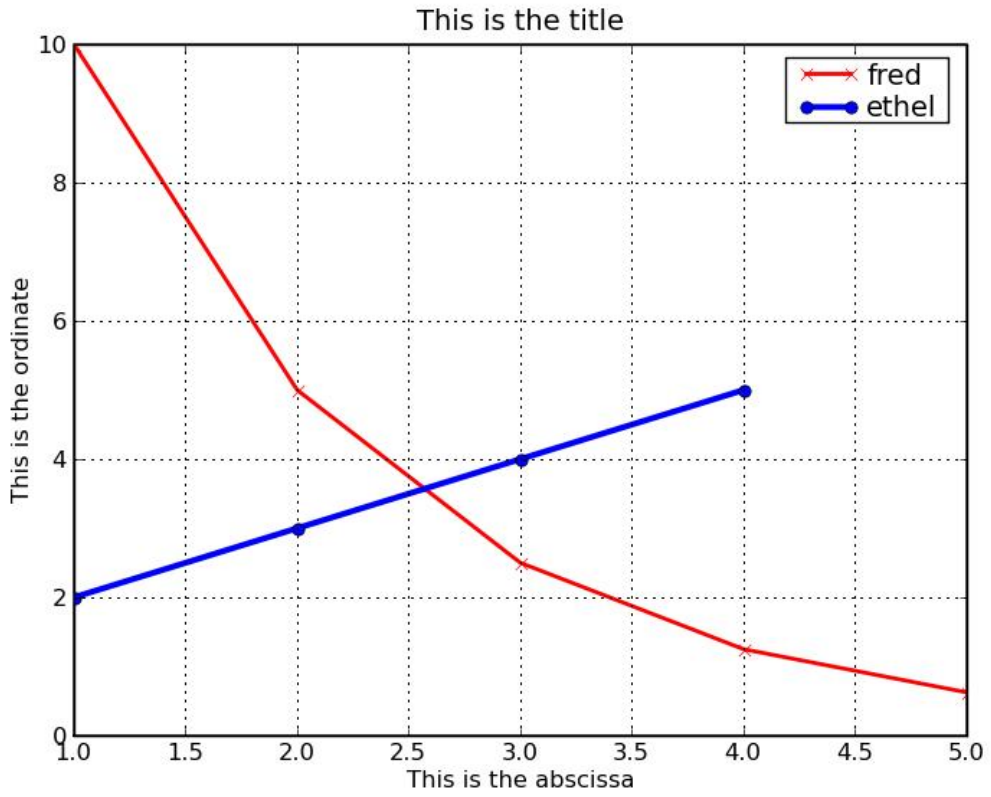
You can edit this code to suit your needs:

```
"""plot a line of Yn vs. Xn from a simple data file.
A simple example of command-line parameters, files, arrays, and plotting

plot <filename>

file:  Xn Yn
        :  :
"""

from pylab import *     # required for matplotlib (aka pylab)
import sys              # argv[]
import scipy.stats      # mean, samplestd
import time             # strftime, localtime

print "plotg:", time.strftime("%a %m/%d/%y %H:%M:%S", time.localtime())
labels=[]                       # the data column labels
y=array([])                     # the 2D array of numbers read in
name=''                         # filename to save plot in
nlab = "record number"          # default label
ylab=''
f=open(sys.argv[1], "rU")       # readonly, Universal line termination

for line in f:
    if line[:1]=="#":  continue         # ignore comments

    # It's a list of numbers
    words = line
    words = words.replace(","," ").split()  # remove commas, separate numbers

    new = [float(s) for s in words]     # the new row of data
    if len(y) == 0:                     # it's our first row
        y = new
    else: y = vstack( (y, new) )        # stack the new row below the previous

# Now all the data are read into 2D array y[,].
f.close()

# Now plot 'em
xlabel("This is the abscissa")
ylabel("This is the ordinate")
title("This is the title")
plot(y[:,0], y[:,1], linewidth=2, color="r", marker="x", label="fred")
x=[1,2,3,4]
z=[2,3,4,5]
plot(x, z, linewidth=3, color="b", marker="o", label="ethel")

grid(True)
legend()
savefig("plot.png")             # create PNG of plot
show()                          # display plot to user
```

With a data file named ‘values.dat’ consisting of:

```
1 10
2 5
3 2.5
4 1.25
5 0.625
```

Then executing **”plot.py values.dat”** produces ‘plot.png’ which looks like this:



See also: http://matplotlib.sourceforge.net/

http://matplotlib.sourceforge.net/users/screenshots.html Excellent code examples, easily adapted.

http://www.scipy.org/Cookbook/Matplotlib

4.2.6 Memory Leaks

Russell Owen's suggestions for freeing up unused memory:

```
From: Russell Owen [mailto:rowen@uw.edu]
Sent: Tuesday, April 27, 2010 10:27
Subject: Re: Memory leak still in ATUI

Some things to check for:

A long time ago I had memory leaks in TUI caused by keeping pointers to
callback functions around when I no longer needed them. As I recall this was
keeping memory for images around, so it was a bad leak.

Lessons I learned:

- If you are making use of callbacks, make sure you always free them (by
  setting the instance variable pointing to the callback function to None)
  when you no longer need to make such callbacks.
- Using weak references from the weakref library (a standard part of Python)
  was invaluable in diagnosing the problem.
- The snack sound library has known issues. The current RO package uses
  pygame instead to play sounds (but depending on whether you have been
  keeping up with TUI and RO, there may be enough changes to make this a
  difficult transition).
```

```
I will add: I doubt the problem is the garbage collector not being run. It is
far more likely a case that you are causing memory leaks (directly or
indirectly) by explicitly keeping things around.

However, there is a gc library that interacts with the garbage collector. I
don't recall it being very easy to use, but you may be able to use it to
locate the memory that is not being freed. You can also use it to explicitly
trigger a garbage collection, but my guess is that will not help.

Out of curiosity: have you kept up with Python? 2.6.x is current. 2.5.x is
quite reasonable. I would not use anything older unless you are desperate.
Older is slower and may have relevant bugs (though again, my bet is on a
coding error that is keeping memory around you don't actually want -- either
in your own code or as you say, in Hippodraw).
```

4.3 Style Guide

Guido van Rossum, the author of Python, has created the following style guide for python: http://www.python.org/dev/peps/pep-0008/

This document was adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide. [Thanks to James Battat of the APOLLO project for this link].

These Python guides are consistent with the philosophy of all coding guidelines:

- Code is read many more times than it is written.
- Consistency with this style guide is important.
- Consistency within a project is more important.
- Know when to be inconsistent.

[Eric M: I especially think the recommendation against mixing tabs and spaces is wrong. For one thing, the main goal of the guidelines is readability, and since tabs and spaces are both white, they both equally provide readability. The proper recommendation is that if you use tabs, they must stop every 8 columns. This insures the code is readable when printed with standard 8-column tab interpretation.

Every editor I've seen easily handles mixed tabs and spaces. The easiest way to achieve the recommended 4-column indent is to use a tab for 2-level (= 8-column) indentation. It helps readability greatly to have your comments on the ends of lines line up, by using a tab to put them there. It is then much easier to keep the comments aligned, as they are insensitive to small changes in the code line before them. Finally, good editors can easily convert between tabs and spaces, anyway.]

4.3.1 Guido van Rossum's Style Guide

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python[1]. [A PEP is a Python Enhancement Proposal; see http://www.python.org/dev/peps/.]

This document was adapted from Guido's original Python Style Guide essay[2], with some additions from Barry's style guide[5]. Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 [6] says,

> "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

(1) When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.

(2) To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

4.3.2 Code lay-out

Indentation

Use 4 spaces per indentation level.

For really old code that you don't want to mess up, you can continue to use 8-space tabs.

Tabs or Spaces?

Never mix tabs and spaces. ??

[Eric M: I especially think this recommendation against mixing tabs and spaces is wrong. For one thing, the main goal of the guidelines is readability, and since tabs and spaces are both white, they have nothing to do with readability. The proper recommendation is that if you use tabs, they must stop every 8 columns. This insures the code is readable when edited and printed with standard 8-column tab interpretation.

Every editor I've seen has no problem with mixed tabs and spaces. The easiest way to achieve the recommended 4-column indent is to use a tab for 2-level (= 8-column) indentation. It also help greatly to have your comments on the ends of lines line up, by using a tab to put them there. It is then much easier to keep the comments aligned, as they are insensitive to small changes in the code line before them. Good editors can easily convert between tabs and spaces, anyway, so it shouldn't bother anyone.]

The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. When invoking the Python command line interpreter with the -t option, it issues warnings about code that ~~illegally~~ [ELM: unconventionally] mixes tabs and spaces. When using -tt these warnings become errors. These options are ~~highly recommended~~ [ELM: discouraged]!

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices looks ugly. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better. Make sure to indent the continued line appropriately. Some examples:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so")
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. ^L) form feed character as whitespace;

Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file.

4.3.3 Encodings (PEP 263)

Code in the core Python distribution should always use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1).

Files using ASCII should not have a coding cookie. Latin-1 should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using \x escapes is the preferred way to include non-ASCII data in string literals.

4.3.4 Imports

- Imports should usually be on separate lines, e.g.:

```
Yes: import os
     import sys
No:  import sys, os
```

It's okay to say this though:

```
from subprocess import Popen, PIPE
```

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

1. standard library imports
2. related third party imports
3. local application/library specific imports

You should put a blank line between each group of imports.

Put any relevant __all__ specification after the imports.

- Relative imports for intra-package imports are highly discouraged.

Always use the absolute package path for all imports. Even now that PEP 328 [7] is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.

- When importing a class from a class-containing module, it's usually okay to spell this

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import myclass
import foo.bar.yourclass
```

and use "myclass.MyClass" and "foo.bar.yourclass.YourClass"

4.3.5 Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

- More than one space around an assignment (or other) operator to align it with another.

```
Yes:
    x = 1
    y = 2
    long_variable = 3
No:
    x             = 1
    y             = 2
    long_variable = 3
```

Other Recommendations

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

- Use spaces around arithmetic operators:

```
Yes:
    i = i + 1
    submitted += 1
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)
No:
    i=i+1
    submitted +=1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)
```

- Don’t use spaces around the ‘=’ sign when used to indicate a keyword argument or a default parameter value.

```
Yes:
    def complex(real, imag=0.0):
        return magic(r=real, i=imag)
No:
    def complex(real, imag = 0.0):
        return magic(r = real, i = imag)
```

- Compound statements (multiple statements on the same line) are generally discouraged.

```
Yes:
    if foo == 'blah':
        do_blah_thing()
    do_one()
    do_two()
    do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

Use the 0 < x < 5 syntax to check for a number within bounds; it's much easier to read than

```
if 0 < x and x < 5 ...
```

4.3.6 Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly. [I (Eric M) strongly disagree with this. Use inline comments profusely, but only when they communicate real information. Don't state the obvious.]

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

4.3.7 Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257 [3].

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the "def" line.
- PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""
```

- For one liner docstrings, it's okay to keep the closing """ on the same line.

4.3.8 Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision: 53621 $"
# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

4.3.9 Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent—nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES

- CapitalizedWords (or CapWords, or CamelCase—so named because of the bumpy look of its letters[4]). This is also sometimes known as StudlyCaps.

Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st_mode, st_size, st_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- _single_leading_underscore: weak "internal use" indicator. E.g. "from M import *" does not import objects whose name starts with an underscore.
- single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g.

```
Tkinter.Toplevel(master, class_='ClassName')
```

- __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below).
- __double_leading_and_trailing_underscore__: "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. Never invent such names; only use them as documented.

Prescriptive: Naming Conventions

Names to Avoid

Never use the characters `l' (lowercase letter el), `O' (uppercase letter oh), or `I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use `l', use `L' instead.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short—this won't be a problem on Unix, but it may be a problem when the code is transported to older Mac or Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. _socket).

Class Names

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

Global Variable Names

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via "from M import *" should use the __all__ mechanism to prevent exporting globals, or use the the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability. mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

Function and method arguments

Always use 'self' for the first argument to instance methods.

Always use 'cls' for the first argument to class methods.

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus "print_" is better than "prnt". (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a. (An insistent user could still gain access by calling Foo._Foo__a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of __names (see below).

Designing for inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores.

- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and __getattr__(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

4.3.10 Programming Recommendations

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Pyrex, Psyco, and such).

For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form a+=b or a=a+b. Those statements run more slowly in Jython. In performance sensitive parts of the library, the ".join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations. [Eric M: use what is easiest to read. 99% of the time, performance is irrelevant. Only worry about performance if it actually matters.]

- Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators. [Why??]

Also, beware of writing "if x" when you really mean "if x is not None" -- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

- Use class-based exceptions.

String exceptions in new code are forbidden, because this language feature is being removed in Python 2.6.

Modules or packages should define their own domain-specific base exception class, which should be subclassed from the built-in Exception class. Always include a class docstring. E.g.:

```
class MessageError(Exception):
    """Base class for errors in the email package."""
```

Class naming conventions apply here, although you should add the suffix "Error" to your exception classes, if the exception is an error. Non-error exceptions need no special suffix.

- When raising an exception, use "raise ValueError('message')" instead of the older form "raise ValueError, 'message'".

The paren-using form is preferred because when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses. The older form will be removed in Python 3000.

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare 'except:' clause.

For example, use:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

A bare 'except:' clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use 'except StandardError:'.

A good rule of thumb is to limit use of bare 'except' clauses to two cases:

1) If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.

2) If the code needs to do some cleanup work, but then lets the exception propagate upwards with 'raise'. 'try...finally' is a better way to handle this case.

- Use string methods instead of the string module.

String methods are always much faster and share the same API with unicode strings. Override this rule if backward compatibility with Pythons older than 2.0 is required.

- Use ".startswith() and ".endswith() instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone. For example:

```
Yes: if foo.startswith('bar'):
No:  if foo[:3] == 'bar':
```

The exception is if your code must work with Python 1.5.2 (but let's hope not!).

- Object type comparisons should always use isinstance() instead of comparing types directly.

```
Yes: if isinstance(obj, int):
No:  if type(obj) is type(1):
```

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2.3, str and unicode have a common base class, basestring, so you can do:

```
if isinstance(obj, basestring):
```

In Python 2.2, the types module has the StringTypes type defined for that purpose, e.g.:

```
from types import StringTypes
if isinstance(obj, StringTypes):
```

In Python 2.0 and 2.1, you should do:

```
from types import StringType, UnicodeType
if isinstance(obj, StringType) or \
   isinstance(obj, UnicodeType) :
```

- For sequences, (strings, lists, tuples), use the fact that empty sequences are false.

```
      Yes: if not seq:
           if seq:
      No: if len(seq)
          if not len(seq)
```

- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.
- Don't compare boolean values to True or False using ==

```
        Yes:   if greeting:
        No:    if greeting == True:
        Worse: if greeting is True:
```

4.3.11 References

[1] PEP 7, Style Guide for C Code, van Rossum

[2] http://www.python.org/doc/essays/styleguide.html

[3] PEP 257, Docstring Conventions, Goodger, van Rossum

[4] http://www.wikipedia.com/wiki/CamelCase

[5] Barry's GNU Mailman style guide

http://barry.warsaw.us/software/STYLEGUIDE.txt

[6] PEP 20, The Zen of Python

[7] PEP 328, Imports: Multi-Line and Absolute/Relative

4.4 Optimization and Profiling

There is a python module called "profile" which can determine time spent in various parts of a program:

```
profile and cProfile
  http://docs.python.org/lib/module-profile.html
```

James Battat

5. Assembly Coding Guidelines

Assembly coding requires tracking CPU resources (especially registers) more closely than is possible (or necessary) in C. Therefore, each procedure *MUST* include a "Modifies" section or a "Preserves" section (but not both) in its header to clearly state what CPU resources are used and changed by the procedure. The caller can safely assume that any CPU resources not "modified" by a procedure are preserved. Conversely, callers *MUST* assume that any CPU resource not "preserved" is modified and undefined after the call.

5.1 Assembly File Headers

The C-compiler defines not only the calling sequence, but also what CPU resources procedures must preserve, and what may be modified. All source files with C-callable routines MUST document in the file header the compiler for which they were written:

```
Compiler:       gcc 2.5.4b
```

5.2 Assembly-Callable Routines

Each CPU type must define what resources are covered by the Modifies/Preserves header comments. For example, most processor registers must be considered "resources," and are covered by the header. However, since arithmetic condition codes often change on every instruction, they are usually excluded, and assumed not preserved. The key is that each CPU type must have a well-known list of what resources are covered by procedure headers.

Callers *MUST NOT* use side effects of the routine. If a routine produces an intentional output, it *MUST* be listed in the "Out:" section of the routine header. E.g.,

```
;-----------------------------------------------------------------
Clear an area of memory.

In:     DI      pointer to start of memory to be cleared
        CX      count of bytes to clear

Out:    memory cleared

Modifies:       AX, DI, CX
;-----------------------------------------------------------------
clear_mem:
        :
```

If it happens, in this example, that the current implementation of this procedure leaves CX = 0 on exit, callers *MUST NOT* rely on this, because it is *not* an intentional output of the procedure, and may be changed at any time during code maintenance. However, this procedure header declares that it does not modify BX, DX, etc., therefore callers may rely on this. As another example:

```
;-----------------------------------------------------------------
Fill an area of memory with a byte value.

In:     DI      pointer to start of memory to be cleared
        CX      count of bytes to clear
        BX      byte value to fill into memory

Out:    memory filled

Preserves: BX
;-----------------------------------------------------------------
fill_mem:
        :
```

Callers are assured that BX is preserved across calls to this function, and may rely on it. Callers *MUST* assume that all other registers are undefined after the call.

5.3 C-Callable Routines

If an assembly-language routine is intended to be called from C, it MUST include a "prototype-like" header:

```
int ut_scan16( uint16 target, uint16 *list_ptr, uint16 list_size );

In:     esp -> return address
        esp+4-> target
        esp+8-> list_ptr
        esp+12-> list_size

Out:    eax: -1 Bad parameter.
        eax: 0 - list_size-1 Index of target in list.
        eax: list_size Target not found.

The list size is limited to 64K entries (words).
```

6. Integrating 3rd Party Software

This chapter discusses issues related to integrating 3rd party software into projects. The goal is to minimize, localize, and document changes to 3rd party software so as to ease upgrading to new versions of 3rd party code.

- The interface to 3rd party software *should* be clean and well documented. The documentation *should* be included in the source as well as the design documentation. At a minimum, the files and routines that are changed *should* be listed as part of the Low Level Design documentation.
- 3rd party software *should* be maintained in directories separate from project original code. Additions and changes which logically belong in the source directories of 3rd-party code *should* go there.
- The original, unmodified 3rd-party source code MUST be preserved and accessible for future reference.
- The original 3rd-party source, and any changes to it, will be librariedin the same manner as new software. This means that the original 3rd party software must be checked into version-control prior to any changes.
- Changes to 3rd party software *should* be written in the style of the original code (where possible). Any routines that are modified *should* have their headers updated. Modified (or new headers) must contain the same type of information defined in the coding-guideline headers. You should add headers to modified routines which had no header, so that the change is documented. Modified (or new headers) *must* at a minimum contain the same information defined in the coding-guideline headers.
- All changes *should* be flagged with the following comment string, so that anyone can find all the changes with a simple text search, e.g. “apollo change”. Multi-line changes *should* begin with “start apollo change” and end with “end apollo change”. For example:

```
i = 0;  // apollo change: index starts at 0

// start apollo change
// The following block was modified to ...
... new code goes here ...
// end apollo change
```

7. Appendix: EXPORTED

This method may be overkill for smaller sized projects, like Apollo. We include it for completeness.

The EXPORTED macro provides a way for developers to create a single declaration/definition in a header file that can be used for both the definition, and the declaration of a variable. The EXPORTED macro works in conjunction with a module defined macro such as ALLOC_ABC. Since the header file in which the shared variable is declared is included in all the files that reference it, the variable *should* be defined using the **ALLOC_xx** and **EXPORTED** macros in order to avoid allocating storage more than once.

The following is an example of a header file:

```
//===========================================================================
Module: abc                  Copyright 2004 UCSD.  All rights reserved.

These are all the declarations [needed to use][private to] the abc module.

Author:        Eric L. Michelsen
Date Created:  2/16/04
===========================================================================*/
#ifndef INC_ABC_H
#define INC_ABC_H  // Make sure the include happens only once

// ---------------- Requires                (do not remove) ---------------
// This is a list of dependencies on other header files
   gi_apollo.h
*/

// ---------------- Constants               (do not remove) ---------------

// ---------------- Structures/Types        (do not remove) ---------------

// ---------------- Variables               (do not remove) ---------------

// This macro should be used when declaring global variables.
   ALLOC_ABC should be defined in the C file where the variables are allocated.
*/
// First remove definitions if they already exist
#undef I
#undef EXPORTED
// Then set the definitions the way they should be
#ifdef ALLOC_ABC
   #define I(x)       x
   #define EXPORTED   // empty
#else
   #define I(x)       // empty
   #define EXPORTED   extern
#endif

// This is what global variable definitions look like using the above macro:
EXPORTED UINT32     global_x    I( = VALUE_X );
EXPORTED BOOLEAN    global_y    I( = TRUE );

// ---------------- Prototypes/Macros       (do not remove) ---------------

#endif // INC_ABC_H
```

If scalar variable initialization is necessary, it *should* be performed using the **I(x)** macro. A template for use of the ALLOC_xx, EXPORTED, and I(x) macros is provided in template.h.

For instance, to allocate storage for the variable led_ticks in the source file pf_filter.c and declare the variable in pf_private.h, the ALLOC_PFPR macro is defined in pf_filter.c before including pf_private.h:

```
// ---------------- Local Includes    (do not remove) ----------------
#define ALLOC_PFPR
#include "pf_private.h"
```

In pf_private.h, the EXPORTED and I(x) macros are defined, followed by the definition of led_ticks:

```
// Remove old definitions of I and EXPORTED
#undef I
#undef EXPORTED
// Set the definitions the way they should be
#ifdef ALLOC_PFPR
   #define I(x)       x
   #define EXPORTED   // empty
#else
   #define I(x)       // empty
   #define EXPORTED   extern
#endif

EXPORTED int32         led_ticks I(= 0); // timeout for LED
```

When the code is preprocessed, since pf_filter.c defines the ALLOC_PFPR macro, the EXPORTED macro resolves to nothing and the I(x) macro resolves to "x". The definition of led_ticks in pf_filter.c looks like:

```
int32   led_ticks=0;
```

But in other files which include pf_private.h but do not define ALLOC_PFPR, the EXPORTED macro resolves to "extern" and the I(x) macro resolves to nothing. The declaration of led_ticks looks like:

```
extern int32 led_ticks;
```

Unfortunately, the I macro only works for scalar variables, it can't be used for composite variables. For composite variables, it is recommended that you emulate the I macro, as shown in the following examples.

```
// To define these variables, define ALLOC_VER in a C file

// First remove definitions if they already exist
#undef EXPORTED
#undef I
// Then set the definitions the way they should be
#ifdef ALLOC_VER
   #define I(x)       x
   #define EXPORTED   // empty
#else
   #define I(x)       // empty
   #define EXPORTED   extern
#endif

EXPORTED const ver hw  cm hw ver
#ifdef ALLOC_VER
= { HW_MAJ, HW_MIN, HW_CHR }
#endif
;
```

You can use constant structures to initialize default values in structure variables. For example, if the following appears in a header file:

```
EXPORTED const msg_addr NULL_ADDR
#ifdef ALLOC MSG
= { AL_NO_COMP, 0, 0, 0 }
#endif // ALLOC_MSG
;
```

You can then use NULL_ADDR in a source file:

```
PRIVATE void sc_discover (void)
{
    my_addr = NULL_ADDR;
...
}
```

8. Stuff Needing Fixing

Don't pay much attention to this chapter; it needs fixing.

8.1.1 Directory Layout

Header files that are used by files in only one directory *should* be put in that directory. Header files used by files in multiple directories *should* be stored in the lowest parent directory of all software modules that include them.

Below is an example of a possible directory layout that adheres to these guidelines:

```
src\                            // common code/data library used by many programs
    BUILD\                      // make files for common source code
    include\
        gi_program1.h           // globals from program 1 code
        gi_program2.h           // globals from program 2 code
        gi_lib1.h               // globals from lib 1 code
    LIB1\
        lib1_private.h          // shared information for this module
        lib1_main.c
        lib1_util.c

Program1\
    BUILD\                       // program specific make files
    include\
        gi_mod1.h               // declarations shared between modules
    MOD1\
        mod1_private.h          // private data for this module
        mod1_main.c             // module specific code
        mod1_util.c
    MOD2\
        mod2_private.h          // private data for this module
        mod2_main.c             // module specific code
        mod2_util.c

Program2\
    BUILD\                      // program specific make files
    include\
        gi_mod1.h               // declarations shared between modules
    MOD1\
        mod1_private.h          // private data for this module
        mod1_main.c             // module specific code
        mod1_util.c
    MOD2\
        mod2_private.h          // private data for this module
        mod2_main.c             // module specific code
        mod2_util.c
```