Coding Guidelines

This document basically addresses questions of syntax, style and layout.In general, all code written for the SOA project must comply with the guidelines and rules given in the remainder of this document, and non-compliance is considered a code defect. In practice, it is accepted that there will always be justifiable reasons to break any guideline or rule. In such case the developer should have valid reasons for any code that is non-compliant and must be prepared to justify the non-compliance at the code inspection.

All new code must follow the standards given in this document. Small modifications being made to code which has already passed a code inspection should be made in the same style as the original code. A small modification is one affecting fewer than 10% of the original lines of code.

When writing code, remember at all times that code is, in general, written once but read many times. If a decision has to be made between ease of writing or ease of later understanding, always select the route which gives the easiest understanding.

Many of the following rules relate to formatting of programs: indentation,spacing, layout, and so forth.

1. Use four-character indentations.

2. Indent with either tabs or spaces (tabs expand to eight spaces). Note: For vi users this means setting tabstop=8 and hardtabs=8. You can also assist yourself in item 1 by setting shiftwidth=4.

3. Spacing must follow normal English style. Specifically, there should be a single space before and after each word (keyword, variable, or constant), except when following an opening parenthesis or preceding aclosing parenthesis. Operators should have a space before and after except when readability is enhanced by doing otherwise (such as to illustrate precedence or for compactness). Braces should always have a space before and after, and closing braces must be on separate lines. Here is an example of an acceptable layout (with nonsense functionality):

while (!found && (--i >= 0)) {

if (ptr[i] == NULL) {

null\_ptr\_index = i;

found = TRUE;

} else {

last\_ptr = ptr[i];

}

}

4. Note from the previous example that all if statements must be at least two lines long. That is, do not use this style:

if (length > MAX\_LEN) return;

Instead, always use this:

if (length > MAX\_LEN)

return;

or this:

if (length > MAX\_LEN) {

return;

}

The following style is also to be avoided, because it gains nothing in clarity and wastes too many lines, limiting the amount of code that can be seen on a screen:

if (length > MAX\_LEN)

{

return;

}

5. Normal variable names must begin with a lower case character, but may be mixed case (e.g. allPages is a legal variable name). You will see a mixture of both the "underscore" style (such as stop\_message\_age\_timer) and the "mixed case" style (such as getnextCurCfgIpIfTable), as well as hybrids like gne\_BM\_frameRelease, so feel free to use whichever style you feel is easiest to understand and best fits the existing code.

6. Do not use single character variable names except for loop indexes.

7. Typedefs must start with lower case, can be mixed case or use underscores, and must end in \_t.

8. Names in all caps are reserved for macros and #define.

9. Do not use more than 80 characters on a line unless impossible. This includes comments.

10. Do not use C++ style comments (i.e., only use /\* \*/ not //).

11. Function definitions consist of optional static and return type on one line with the function name and argument names in ANSI standard prototype style on a second line.

-OR

Function definitions consist of optional static, return type, function name, and argument names and types in ANSI standard prototype style on a single line (with extra lines if the arguments would extend past column 80). In addition, please maintain the style of the existing code for existing modules.

12. There are many ways to indent the case statements following a switch, but the following form is the one implemented by the our syntax-aware editors and is therefore the only one to be used. The switch is indented as appropriate, and the case statements go in the same column as the switch. This results in the actual statements within each case being indented four from the switch, which is both correct and conservative of line width. Here is an example of the correct form:

if (ltr->command == CMD\_TABLEX) {

switch(ltr->modifier) {

case TABLEX\_MOD\_ADD:

add\_tablex\_entry(ltr);

break;

case TABLEX\_MOD\_DELETE:

delete\_tablex\_entry(ltr);

break;

default:

rc = RC\_ERROR;

break;

}

}

Also note the break following the default case. This is good programming practice, because it minimizes errors if new cases happen to be added after the default.

13. Header files should not include other header files, except for the convenience of external subsystem users (e.g., the single #include "st\_api.h" includes all headers needed by users of the STEM API).

14. It is good programming form to parenthesize for the human reader, not for the compiler. That is, if adding some redundant parentheses will make it less likely for a human to misunderstand, then add them, even if the rules of C don’t strictly require them. This is particularly important when mixing shift, relational, and bitwise logical operators. For example, consider the following expression:

previous >> align + 1 == next >> align & mask

It is not at all obvious what this does, or whether it is correct.Instead, clarify it by parenthesizing, such as:

(((previous >> align) + 1) == ((next >> align) & mask))

And by the way, these two expressions are quite different.

15. "Magic numbers" should never be used anywhere in code or strings. ("Magic numbers" are numbers other than the obvious ones like 0 and 1, such as 256 for maximum input string length or 16 for an sprintf() buffer size.) Instead, #define an appropriate symbol and use that symbol. For example, don’t use this:

printf("Enter new Spanning Tree Group index [1-8]: ");

Use this instead:

printf("Enter new Spanning Tree Group index [1-%d]: ", CFG\_MAX\_NUM\_STP\_GROUPS);

Unfortunately, there are situations in which use of a #define symbol is difficult or even impossible. For example, the strings displayed in a CLI menu are constants, and therefore cannot incorporate #define symbols. Files other than C source can also present problems. In such situations, it is acceptable to use "magic numbers," but only if the relevant #define symbol is also included as a comment. For example,

char usage[]="Usage: cmd with up to 15 params" /\* 15 is CLI\_MAX\_PARAMS \*/

This makes the implicit use of the #define symbol visible to grep, making it much less likely to be missed when making a change.

16. Enums are to be used only when clearly superior to #define and other techniques. In particular, enum must not be used for values that should remain constant in traces and dumps, such as error codes and message command codes.

17. Builds are to be done with standard Makefiles and without compiler warnings. Clean code is healthy code.

18. To minimize mistakes, variables should have the minimum scope necessary to do their job. If a function is used only within a single module, it should be declared static. If a variable is used only within a single function, it should be declared inside the function, to be put on the stack. If a variable is used only within a single compound statement (such as following an if or a within a for loop), it should be declared inside the compound statement. If a variable that is used only within a single function needs to be static, it should still be declared inside the function; such local static variables have permanency, but are visible only within the function, minimizing errors. This points out the dual nature of static in C, by the way: static indicates both permanence and also some degree of privacy. Minimizing the scope of variables minimizes the chance for misuse, always a good thing.

19. All non-static functions must have prototype declarations, and these prototypes must go in a header file to be included by all callers of the function. Putting a prototype immediately in front of its function is useless; don’t do it. (Prototypes are needed for static functions only if the functions must be defined after they are first called, and in this particular situation it is acceptable to put the prototype at the top of the C file. A better technique is to define static functions before they are referenced, however.)

20. Prototypes must contain argument names along with their types, for added documentation. That is, use

char \*strcpy(char \*dst, char \*src);

rather than

char \*strcpy(char \*, char \*);

21. Because they defeat the compiler’s ability to do error checking, casts are to be used only when absolutely necessary. There is no error checking on void \* assignments, however, so casting a void \* does nothing but add documentation and therefore is encouraged.

22. In keeping with modern programming principles, use of the goto statement is strongly discouraged. There are situations where goto is appropriate, however, so it is acceptable. Labels for goto must be on a line by themselves and are to be "outdented" four spaces from the current position. For example (and note the following does not indicate an appropriate use of goto, merely proper indentation):

switch (transaction\_type) {

case ADD:

add\_it();

goto finish\_up;

case REMOVE:

remove\_it();

finish\_up:

finish\_transaction();

break;

}

23. Conditional compilation must use #if, #else, #elif, and #endif only; #ifdef and #ifndef are to be avoided. The reason for this is that #if and #elif can handle arbitrarily complex logical expressions while #ifdef and #ifndef annot.

24. There is one exception to the previous rule: Recursion protection in header files must use #ifndef rather than #if !defined. For example, the header file sample.h must have the following structure:

#ifndef \_SAMPLE\_H\_

#define \_SAMPLE\_H\_

. . .

. . .

#endif /\* \_SAMPLE\_H\_ \*/

The reason for this requirement is that make depend will not follow nested #include without it. This exception also applies to modules that essentially disappear if a particular configuration symbol is not defined. For example, the module slb.c has #ifdef SLB at the very start, in front of the #includes. this needs to be #ifdef instead of #if in order to get make depend to process the headers. Put another way, #if should never be used to remove a include. Use #ifdef or #ifndef for that. And note that there are a lot of cases in the current source tree where this is not done correctly. You don’t have to fix all of those, but don’t add to the problem by adding new code that uses #if to conditionally remove a #include.

25. Conditional compilations should be based on the defined attribute rather than value tests. For example, #if defined(OPTION\_A) is preferred to #if OPTION\_A. There are times when value tests may be appropriate, however, as in:

#if (ERROR\_LEVEL == 2)

26. Every occurrence of #else and #endif must have comments that are exactly the same as the conditional expression on the corresponding #if, to make it easier to use editor searching tools to determine the scope of conditionals. For example:

#if defined(OPTION\_A)

...

#else /\* defined(OPTION\_A) \*/

...

#endif /\* defined(OPTION\_A) \*/

27. The #elif construct may also be used. Again the #endif must have comments that are exactly the same as the conditional expression on the corresponding #if. This does mean that the conditional expression on the #elif itself doesn’t appear as a comment, but it is felt that this is preferable to trying to include everything in the #endif comment. The following example is correct:

#if defined(OPTION\_A)

...

#elif defined(OPTION\_B)

...

#endif /\* defined(OPTION\_A) \*/

28. Do not put other comments on the conditionals themselves because an automatic conditional-removing program will remove the conditionals and their comments both. If comments are desired, they should be normal C comments, either in front of or behind the conditional as appropriate.

29. Sometimes it is useful to be able to temporarily remove a block of code. This must be done with #if defined for an obviously-undefined "symbol," not by commenting the code out. That is, use the following:

#if defined(REMOVE\_UNTIL\_RIP\_V2\_AVAILABLE)

if (x[i] == 0) { /\* if element is not yet defined, \*/

update\_index = i; /\* set flag for immediate update \*/

}

#endif /\* defined(REMOVE\_UNTIL\_RIP\_V2\_AVAILABLE) \*/

Do not use the following:

/\*

if (x[i] == 0) { /\* if element is not yet defined, \*/

update\_index = i; /\* set flag for immediate update \*/

}

\*/

There are several reasons for this. First, syntax-aware editors can hardly be expected to do the right thing with commented-out code. Second, information about why the code has been removed can be conveyed by the "symbol" used in the #if. And most importantly, there are problems with nested comments: in fact, the second example above would not remove the code as desired! So rather than encourage people to not use comments, temporary code removal must use #if.

30. There are (at least) four ways of writing "zero," and the appropriate form should always be used: 0 for numbers, NULL for pointers, FALSE for booleans, and ’\0’ for chars (such as when testing for a null string). Likewise, 1 should be used for numbers and TRUE for Booleans.

31. Speaking of Booleans, the following rules apply to their use.

A. Boolean variables should be assigned the values TRUE (1) and FALSE (0) only. So it is not acceptable to do things like the following:

bool nonword = (addr & 3); /\* TRUE if not word-aligned \*/

B. On the other hand, ANSI C does guarantee the results of logical operations will be 0 (FALSE) or 1 (TRUE). So it is acceptable to do things like the following:

bool active = (state == STATE\_ACTIVE);

This includes the operators ==, !=, <, >, <=, >=, &&, ||, and ! . It does not include &, |, ^, or ~ .Harking back to rule A above, the following would be acceptable:

bool nonword = ((addr & 3) != 0); /\* TRUE if not word-aligned \*/

C. The only acceptable use of Boolean variables is decision-making, such as the following:

if (nonword)

printf("whassup?");

printf("%s", nonword ? "whassup?" : "");

In particular, while we know that Booleans will only have the value 0 or 1, the following is not acceptable for setting bit 16 to 1 if the Boolean variable active" is TRUE:

bit\_mask |= (active << 16);

This should be done using something like one of the following:

if (active)

bit\_mask |= (1 << 16);

bit\_mask |= (active ? (1 << 16) : 0);

bit\_mask |= ((active ? 1 : 0) << 16);

D. It is not necessary to test Booleans for equality to TRUE or FALSE, and this can lead to errors if someone violates rule A. Therefore the following should not be used:

if ((active == FALSE) || (nonword == TRUE))

error\_abort();

Use this instead:

if (!active || nonword)

error\_abort();

E. Because of rule D, Boolean names should always reflect their TRUE setting. For example, don’t use the name sp\_state , use either sp\_up or sp\_down , as appropriate.

32. Bit fields should not be used to pack single-bit flags or minimize field size except in very unusual circumstances (such as arrays of thousands of elements). Bit field accesses require additional machine instructions for masking and shifting, and these almost always outweigh any space savings due to bit-packing. For example, do not squeeze a 12-bit number and four Boolean flags into a single 16-bit integer. Instead, use a 16-bit integer for the 12-bit number and four separate bool variables (unsigned chars) for the flags. This "wastes" four bytes of storage, but that’s far less than the additional instructions required to combine and separate the bit fields. The only normally-acceptable use of bit fields is to make structures agree with some external specification, such as a BPDU or an IP header. Unfortunately, this must also be done with caution, because C says nothing about how bit fields are to be stored; be sure to check carefully when adding or changing such bit fields.

**Naming Conventions**

This chapter provides additional naming conventions to those of Section 2.

**Name Length**

**1.** Do not use names containing more than 20 characters.

Note that other naming conventions in this document have priority over the length of the name. The goal is to provide a name that complies with all the rules and recommendations that is a short as possible.

**Types**

**2.** Use type and scope attribute suffixes as given in the following table.

pointer \_p enum or constant \_c global \_g void \_v

**Identifier Spelling**

**3.** Spell out all names in full as if they were to be used in a document.

**4.** Only abbreviate to proper English spelling and common acronyms.

**Function Names**

**5.** Use verbs in function names rather than nouns.

**6.** Treat acronyms in function names as normal words.

**Variable Names**

**7.** Use nouns in variable names rather than verbs.

**Parameter Names**

**8.** Parameter names used in function prototypes shall match exactly the names used in the implementation.

**File Contents**

The following section specifies a standard file layout for .h and .c files.

**4.1 Include (.h) File Layout**

Note that two techniques may be employed for the naming of .h files: . all inline code may be contained in one .h file which itself includes the main .h file . both inline code and other definitions may exist in the same .h file

**1.** If a .h file contains more than 100 lines then separate out the inline code into a separate .h file.

**2.** Name include files which contain no code with names of the form <name>.h.

**3.** Name include files which contain inline functions with names of the form <name>IN.h. Include the file <name>.h from within the <fname>NI.h file. **4.** Do not place any non-inline code in any include file.

**5.** Even if the code is trivial, do not include inline code with the function declaration.

**6.** Place items within an include file in the following order:

. #ifndef \_<filename>\_H\_

. #define \_<filename>\_H\_

. File block comment which includes copyright notice and file history

. #include statements for system .h files

. #include statements for local .h files

. Forward declarations

. constants, enums

. structure definitions

. function prototypes

. inline functions

. #endif /\* \_<filename>\_H\_ \*/

**C File Layout**

**7.** Place items within a C file in the following manner:

. File block comment which includes copyright notice and file history

. #include statements for system .h files

. #include statements for local .h files

. static constants

. structure definitions

. static function prototypes

. function code, including function block comment immediately preceding the

function definition which includes description of the function, its inputs, and its outputs

**8.** Order functions in a sensible groupings, typically the main flow of execution first withmiscellaneous utility functions at the end.