GALACTICOMM CODING STANDARDS

by Tim Stryker
12 June 1990

1. Why Standardize?

The benefits of all Galacticomm programmers using the same standards for code design and layout are substantial. We can read each others' code more easily, without constantly being struck by unfamiliar formatting. We can make mods to each others' code, without creating a mishmosh of different coding styles. In addition, if we choose the "right" standard, our mutual productivity can be enhanced. Details like whether or not you always put curly-braces around an if-clause, for example, have a significant impact on the time it takes to change a programming construct, both during initial development and later in the life-cycle of the program.

The objections to code format standardization generally fall into one of the following categories:

"It takes time to learn the standards, that could be spent coding."
"My way is better."
"It's not that big of a deal."
"Just use a pretty-printer program if you think it's so important!"

It doesn't take very long to learn at all, since most of the standards outlined here are ordinary common sense. In the cases where an arbitrary choice has been made, such as always putting spaces around a double-equals (x == y), never around a single-equals (x=y), it is not all that hard to remember and it quickly becomes second nature. The central thing is that there BE a standard, so that you don't have to stop and think, each line of code you write, how it should best appear. The time lost learning the standard is very quickly recouped because bugs are reduced, code is reused more, and more interchange with other programmers becomes possible.

Your way may be better, but if it's different from other people's, you will have a hard time reading their code and vice versa. Your eyes get used to seeing code laid out in a certain way after a while, and when you have to deal with code in which the placement of braces, use of upper and lower case, spacing conventions and so forth are different from yours, the comprehension of every line is a battle. Brain cells are in short enough supply... the less you have to devote to deciphering the formatting of someone else's code, the more you have available to understand what's really important, which is its content and function.

Formatting is definitely "not that big of a deal" when compared with the issues of code architecture, flow, encapsulation, etc. That's why it's silly to let it interfere with the understanding of these larger issues. Reading code in an unfamiliar format is like slogging through mud -- you have to consciously dope out which brace goes with which, and whether something is a comparison or an assignment, etc. Reading code that looks like what you are accustomed to, you barely notice that there is a format at all -- you see straight through it, to the important aspects of what the code actually does.

Pretty-printers are not the solution, either. Besides the fact that pretty-printers are time-consuming to configure and use, there is an inherent benefit in writing your actual code in a standardized way. Your thoughts become more organized, because the layout on the screen in front of you is organized and can be read without conscious effort. If another programmer is working with you, he may be able to catch bugs that you didn't, just by looking over your shoulder as you edit. And, calling in a "guru" to help with particularly knotty problems will not result in hostile mutterings in your direction.

2. C Source

The standards we adhere to are basically those of Kernighan and Ritchie, with a few small extensions.

2.01 Block Comment Header

Each C source file should begin with a block comment of the form:

```
/***************************************************************************
 *                                                                         *
 *   FILENAME.C                                                            *
 *                                                                         *
 *   Copyright (C) 19xx-19xx GALACTICOMM, Inc.    All Rights Reserved.     *
 *                                                                         *
 *   Brief description of the contents of the source file.                 *
 *                                                                         *
 *                                                 - Author MM/DD/YY       *
 *                                                                         *
 ***************************************************************************/
```

2.02 '#include' Statements

Following the block comment header should be all of the C header files to be included in the compilation, in order from the most standard to the most application-specific. Generally "stdio.h" should be the very first in the list, followed by files like "ctype.h", "dos.h", and "setjmp.h". After these would come Galacticomm's own global utility files such as "btvstf.h" or "dosface.h", followed by application-specific files such as "majorbbs.h" or "usracc.h". At the end of this list would appear specialized headers only shared by one or two source files, such as "flash.h" or "message.h".

Popping an '#include' into the middle of other declarations, or code, is frowned upon.

The use of angle brackets to enclose the included filename as opposed to double-quotes is a matter of personal taste. We use double-quotes (and give the compiler explicit path directives at compile-time), but this is not a critical issue.

2.03 Declarations and '#define' Statements

After the '#include' statements should come all of the public data declarations, external data declarations, structure declarations, global non-int function return value overrides, and '#define' statements in the file. Use whatever order seems clearest -- there is no need to put all of the structure declarations together, all of the '#define' statements together, etc. However, once the code starts, there should be no further appearance of any of these five classes of statements except in unusual circumstances. The purpose of this is so that when someone reading the code sees a reference to one of these things, he knows that if it's in the file at all (as opposed to a header file), he can find it toward the top: if he hasn't found it by the time he reaches the first actual function definition, it isn't there.

Within function definitions, all declarations of local and automatic variables, local structures, and local non-int function return value overrides should appear at the very top of the function, indented one tab stop. A blank line should separate these lines from the first line of actual code.

2.04 All Caps for '#define' Constants

Generally the only use of upper case in a C file (besides in comments and quoted character strings) should be for constants defined via the '#define' statement. If the symbol defined in a '#define' statement takes an argument list, though, the symbol may be in lower or upper case. The central rules are: always use upper case for CONSTANTS defined via '#define'; use lower case for all normal non-preprocessor-related code text.

2.05 Data Declaration Comments

Each variable declaration, and each element of each data structure declaration, should have a short (25 to 40 byte) comment on it.

2.06 Order Functions 'Top Down'

The order of arranging functions in a source file should be generally from the highest-level functions to the lowest. This is not an ironclad rule, except that the function 'main()' should always appear first if it is present. Similarly, function groups in which some function acts as a kind of master gateway to a group of subsidiary functions should generally have the master function first, followed by its direct subsidiaries, followed by subsidiaries of those subsidiaries, etc.

2.07 Code Comments

The code should be clear and understandable by the very nature of its design and architecture. Also, by the appropriate choice of variable and function names it should be as self-documenting as possible. Therefore there should be very few actual comments in the code itself. Ideally, each function should have a one-liner comment, and functions with special considerations or especially complicated argument lists can have explanatory text around them. However, if you think that your code would be unclear to a reasonably diligent reader, the solution is not to comment the unclear aspects. The solution is to redesign that portion of the code.

2.08 Short Names

Variable and function names should be no more than 8 characters, preferably no more than 6. This keeps the code from getting bulky and requiring a lot of continuation lines. Shorter names also take less time to read, and less time to type. They can be just as easy to understand and remember as long names, if chosen well. The name 'set_attribute()' is no more intrinsically clear than 'setattr()', and it takes much longer to type (over and over!), and to read afterward.

The use of the underscore character, in particular, should be avoided whenever possible.

2.09 Indentation and Curly-Braces

Use the K&R standard of tab stops every five columns. More than this leads to code walking off the right edge of the screen; less makes it hard to tell, visually, which code sections line up with one another vertically.

Indent the body of every if, else, do, for, and while block one tab stop. In a switch construct, keep each 'case' statement at the same indent level as the 'switch' statement itself, but indent the code under each case one tab stop.

Each and every if, else, do, for, while, and switch should have a left curly brace positioned at the end of the opening line of the construct (or its continuation), and a right curly brace on a line by itself at the end of the dependent code. This rule most definitely applies even if there is only one line of dependent code in the construct. Also, dependent code should never appear on the same line with the left curly brace. The right curly brace at the end of the dependent code should be at the same tab stop as its parent keyword.

In a series of if-else if-else if-...-else clauses, the initial 'if', its corresponding right curly brace, each 'else', and each of their corresponding right curly braces should all be at the same tab stop.

The only time that a left curly brace should appear on a line by itself is in column 1 at the start of a new function definition, right after the function parameter declarations if any.

Never put more than one statement on a single line: this leads to time-wasting tradeoff analysis, and further time-wasting reformatting in the event that the analysis is wrong, or becomes invalid in the light of later developments.

2.10 Spaces

Besides in comments and quoted character strings, there should be no spaces anywhere in your code except:

> as preprocessor statement delimiters
> for indenting
> after declarators such as 'char', 'int', 'struct', 'static', etc.
> on either side of comparison operators (==, !=, >=, <=, <, and >)
> on either side of binary logical operators (&& and ||)
> after if, else, do, for, while, switch and corresponding close-parens
> on either side of each ';' in 'for' statements
> on either side of '?' and ':' in conditionals

In particular, spaces should NOT appear around commas, or around any arithmetic, boolean, or assignment operators. The following are examples of operators that should NOT have spaces on either side of them:

```
+    -    *    /    <<   >>   &    |    ^    ~    !    ++   --   ->
+=   -=   *=   /=   <<=  >>=  &=   |=   ^=   =
```

A short piece of code with examples of where to use and not to use spaces:

```
        #define P2TSIZ 5            /* power of 2 table size */
        int powof2[P2TSIZ];         /* power of 2 table      */

        compow2()                   /* compute powers of 2   */
        {
                int i;

                for (i=0 ; i < P2TSIZ ; i++) {
                        powof2[i]=(i == 0 ? 1 : powof2[i-1]<<1);
                }
        }
```

Note: it is especially important to maintain the distinction between single-equals (assignment) and double-equals (equality comparison). The advantage here is that, once you are used to this standard, your eyes will react to the appearance of a single-equals with spaces around it, or double-equals without them, as an alarm situation requiring investigation. This greatly cuts down on the class of bugs in which an intended comparison is inadvertantly coded as an assignment.

2.11 80-Column Lines

Source code lines should never contain characters beyond the 80th column. This keeps all code visible when scrolling through with a standard text editor.

If you have statements that require more than 80 columns, split them across two or more lines. There are no fixed rules determining where the splits should occur -- just try to make the code as readable as possible. This means lining up similar sub-clauses where possible, for example:

```
if (sameas(usaptr->userid,esgptr->msg.to)
 || sameas(usaptr->userid,esgptr->msg.from)) {
```

Where no similar sub-clauses exist, try to break lines at logical boundaries, such as after commas, before '&&' and '||' operators, etc. When breaking on '&&' or '||', always put the operator at the start of the continuation line, not at the end of the line being continued. This helps the reader more quickly identify the continuation line as such, and not as a new statement in its own right. Also, use a non-tab-stop column for the start of continuation lines, as this helps further cue the reader to the fact that the line is not a normal standalone statement.

Avoid the use of the 'backslash' method of creating continuation lines, since this tends to mar the otherwise pristine indentation structure of your code (except on global-declaration lines which are not indented anyhow).

2.12 Blank Lines

Use a single blank line to separate:

> the block comment header from the '#include' statements
> the '#include' statements from the global declarations section
> each new function definition from what precedes it
> the declarations at the top of a function from its code

You can use blank lines, sparingly, in other places too, but be careful of the syndrome that some programmers fall into in which they try to enclose every minor grouping of variables or code between blank lines -- it is important that the reader of your code be able to fit as much of it as possible on the CRT at the same time. Double-spacing your code lines, in effect, makes your CRT act like it is only 12 lines high rather than 24. This makes it more difficult to relate sections of code to each other that may be, say, 20 lines apart.

2.13 Start Actual Function Name Definitions in Column 1

This little bit of esoterica has several benefits. For text editors such as KEDIT that have a primitive for string searches constrained to start in column 1, it becomes much easier to find the definition of a function in a file without having to wade through a potentially large number of references to it. Secondly, it leaves more space for the one-liner function comment on the function prototype line itself. Also, it helps to visually separate the function name from its return value type, making it quicker to tell exactly what the return value type is. Here's an example of the contrast we are talking about:

```
struct qmitem *getpitm(who,name)    /* little room for comment! */
```

vs.

```
struct qmitem *
getpitm(who,name)    /* plenty of room for a nice long comment! */
```

2.14 ANSI Function Prototypes

There is room for debate here, and the world does seem to be moving toward requiring ANSI prototypes for all functions before referencing them. Galacticomm standard at the moment is not to use ANSI prototypes. The main reason for this is coding overhead: to prototype every silly little function as you are coding along is more time consuming than not to prototype them. The only benefit of prototyping is that the class of bugs in which the caller passes different argument types to a function from those that it is expecting are caught at compile time (and maybe link-time, if the type-checking implementation of the development environment is thorough enough -- not all are). Our position at the moment is that if the programmer is any good he does not need to be protected from himself in this way, which means that the overhead of fooling around with prototypes is a pure waste of time.

2.15 Miscellaneous Idiosyncracies

Here are a few hairsplitters that yield a generally more graceful look and feel to your code:

> When dealing with an 'int' variable that you are using as a counter or index, and making a test based on whether it is or is not zero, use an explicit 'blah == 0' or 'blah != 0' test, don't just say 'if (!blah) {' or 'if (blah) {' respectively. This helps the code reader grok that 'blah' takes on values other than 0 and 1.

> When incrementing or decrementing a variable on a line by itself, use the statement 'blah+=1;' or 'blah-=1;' rather than just 'blah++;' or 'blah--;'. This keeps the code more general (suppose it turns out that you need to increment it 2 instead of just 1?), and it avoids the incongruity of creating a net expression value and then just throwing it away. (By convention, this incongruity often exists in the special case of the third clause of a 'for' statement, and we do follow this practice too -- who said a standard has to be perfectly consistent!)

> Always use parentheses around the return value (if any) in a 'return' statement. Somehow it looks more well-defined that way. This is one of those purely arbitrary situations where the only justification for having a standard is that people who get used to seeing it one way will be distracted by its appearance the other way, so to minimize debate about nonessential issues we just mandate that return values should always have parentheses around them and we can all relax.

> Explicitly declare functions returning int as doing so, at least in the definition line -- this alerts the code reader to the fact that the function does in fact return a value, and the type of that value.

> Coding techniques to avoid: typedefs, the 'continue' statement, lots of macros, particularly when heavily nested or interdependent, and complicated data entities such as casts of dereferenced pointers to arrays of pointers to functions returning pointers to doubly-indexed arrays of data structures. Nuff said.