CSE 465M
Digital Systems Laboratory
Fall 2006

C Coding Guidelines

The following coding guidelines will be required of all C code in CSE 465M. This will be enforced in two ways: 1) anything turned in not conforming to these guidelines will be penalized, and 2) anything you ask the instructors to help debug will first be checked for conformance, with any help being initially limited to moving into conformance. These are not recommendations; they are requirements.

Think of your code as a composition, which is judged on style as much as (or more than) simply on correctness. If the instructors cannot understand it, that is just as “incorrect” as if it gives the wrong answer. Rest assured, it is in your best interest to keep the instructors happy.

Also, your code must compile without errors or warnings (with -Wall)!!! Fix the stuff the compiler is telling you about first, and then look for the logic errors that remain after that.

1. Symbol Naming

Use underscore notation for multi-word symbols, e.g., sample_buffer or frame_delimiter. When choosing a name, err on the side of being over-descriptive. Never use names like “flag”; tell the reader what is being flagged. The following conventions must be adhered to for symbol names:

externally visible functions:
Begin name with lower case prefix identifying the module followed by the name of the function itself, such as “xx_name” for function “name” of module “xx”.

static functions:
Follow the same convention as for externally visible functions, but the name need not have the prefix.

variables:
Always use lower case when naming variables, such as in “last_frame_flag” rather than “Last_Frame_Flag”.

The choice of a variable name should be mnemonic, that is, designed to indicate to the casual observer the intent of its use. One character variable names should be avoided except for temporary “throw-away” variables. Common names for temporary variables include i, j, k, m, and n for integers; c, d, and e for characters.

Pointer variables should always start with the letter “p”; non-
pointer variables should almost never start with the letter “p”. An exception can be made if there is no room for ambiguity, such as a name like “prev_contents”.

#defines:
Name all macros with all uppercase names, such as “FRAME_SIZE”.

An explicit exception is allowed for macros that encapsulate a function call combined with the necessary error checking on the return value. For example,

#define Read_Header(f,h) {if (read_header(f,h) < 0) exit(1);}

For these macros, use underscore separated names that are capitalized and correspond precisely to the lower case true function names. The capitalization should not include the module prefix for externally visible functions.

typedefs:
Type definitions come in two forms.

1) a set of type definitions to make clear the size of variables take the form “int8”, “uint8”, “int16”, “uint32”, etc. See the examples in the file “dl_types.h”

2) type definitions of compound data structures should have their names start with “t_”, e.g., “t_header”.

2. Code Documentation

All C files must contain a header that contains the following header information: filename, brief description of the function of the module, brief description of the inputs and outputs, author(s), and creation date.

After the header comment, there should be a single line

```c
static const char *rcsid = "$Header$";
```

containing the CVS version information as a static. This can be subsequently extracted from the object file if there is doubt as to the version of the source code which was compiled.

Each function should be preceded by a description of the logical operation and (most important) its purpose in the design.

Don’t bother with comments that literally describe the code, the code does that just fine. Include comments that tell the part of the story that isn’t clear from the code itself.
3. Source Code Guidelines

C source files have the following ordering of internal components:

- header comments
- CVS version information
- #include statements
- #defines
- local data type definitions
- local (static) function prototypes
- local (static) data
- externally visible functions
- local (static) functions

Any exceptions to the above should be for good reason (e.g., improving some specific aspect of the readability or understandability of the code).

Follow the guidelines below for indentation:

tabs
Spaces are to be preferred over tab characters, which often don’t port across editors.

levels
Use a small number of spaces (2 to 3) as indentation to the next level.

braces
The open brace “{” goes at the end of the line before the start of the code block.
The close brace goes on its own line, indented to match the beginning of the line containing the corresponding open brace (an exception is “else” which can go on the same line as the closing brace for the corresponding “if”, so the closing brace for the “if” doesn’t appear on its own line).

code inside braces
Indent one level for each level of curly braces “{}”.

code inside consequents
Use curly braces and indent one level for consequents in conditional statements (braces may be omitted if there is only one statement in a consequent, but some editors expect the braces in order to do automatic indentation properly).

continued lines
When a statement continues across two or more lines, indent the second and remaining lines an equal amount past the start of the first line of the statement. Pay attention to the readability of the line break as well.