**Coding Style and Doxygen Documentation**

|  |
| --- |
| 1. [Introduction](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#introduction) 2. [Comments for Files](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#files) 3. [Comments for Functions and Data Structures](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#functions) 4. [Coding Style / Coding Standards](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#style) 5. [Description of Commands](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) 6. [Putting it All Together](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#example)   **Introduction**  Throught this course you will be working on big software projects and an important part of any large project is documentation. Documentation is especially important in group projects, like many of the projects in this class. Imagine how useful a browsable index will be when your partner is asleep and you're trying to figure out his/her code. In order to ensure that your source code has adequate documentation, we will be requiring that your code be fully documented using doxygen, a documentation system for C similar to JavaDoc. This document serves as a brief overview of doxygen and the features you will use on a regular basis. For a more detailed description of doxygen and all of its features visit the [doxygen homepage](http://www.stack.nl/%7Edimitri/doxygen/). You can get detailed documentation and information about compiling and installing doxygen on a home system.  There are two types of comments we want in your code.   1. The first kind is comments at the beginning of each file which describes the file and lists things like author and known bugs. 2. The second kind of comments is those that describe your functions and data structures. We want you to put comments on all your functions and data structures. You do not have to comment individual variable instances but are welcome to if you'd like.   The rest of this document talks about the doxygen commands that you need for each of the two kinds. First, we'll describe what we expect to see and then talk about the specific commands that you need to use including simple examples. Lastly there will be a larger example showing all of commands together.  **Comments for Files**  Each file needs to begin with the [**@file**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command stating the name of the file. This should be followed by a brief description of the file using the [**@brief**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command. If necessary, you can follow this with a more detailed description. Next you should put your name and ID, along with your partners name and ID, using the [**@author**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) tag. This needs to be followed with a bugs section with a list of known bugs using the [**@bug**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command. If there are no known bugs, explicitly state that using the [**@bug**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command.  **Comments for Functions and Data Structures**  Before each function, data structure, and macro you should put a comment block giving at least a brief description using the [**@brief**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command. A brief description will suffice for your data structures but for you macros and functions you will need to use a few more commands. After your description, you should use the [**@param**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command to describe all of the parameters to your function. These descriptions should be followed by a description of the return value using the [**@return**](http://www.cs.cmu.edu/%7E410/doc/doxygen.html#commands) command. Note: When we say "each" function, that is not a strong statement. You can leave out simple helper functions, like a max() macro, so you don't waste time.  You can choose to comment your functions either in the header files where they are declared, in the source files where they are implemented or both. This is a matter of taste. If you put it in the header file, like in the example, then you should be sure to remember to update the comments with the latest details of the implemenation. If you choose to put the comments in both places note that if there is a difference between the two sets of comments, the block at the declaration will superceed the one at the implementation. The long and short of this is that you should **not** put the comments in both. For assembly files, you can put the comments in the header file where they are declared.  Here is what we expect to see in the non-brief section of the function:   * Anything a user needs to know to decide whether this is the right function for them to use for a given job. * Usage preconditions: must be called with interrupts disabled, etc. * Any use of an unusual algorithm (in which case, cite it -- academic format or page title and URL) * Why the code was written in a non-obvious structure. * Warnings about how to extend the function without breaking it.   Not all of that will be needed for every function. Just make sure it is enough to make it clear what the function does and how it does it. Don't worry to much about this documentation stuff. If we feel that it is not up to what we want, we'll let you know with your grade report and probably won't take points off right now, but may in future projects. Make sure that the documentation is there and that it makes sense.  **Coding Style / Coding Standards**  At some point in your career you may well be required, or at least expected to adhere to an explicit coding standard document. If you are ever present at the inception of a large project, you will have the opportunity to watch the inevitable flame war over tab stops and brace placement.  However, you should use a consistent and defensible style. It would probably be a good experience for you to read a few coding style documents to see how they differ and how you feel about the differences. When you begin work with your partner on Project 2 you will have an opportunity to discuss this issue. We won't require the two of you to use exactly the same style, but it would probably make sense for style to not vary wildly within a single file. For open source projects (i.e., potentially many authors over many years), one popular approach is for the "primary author" of each file to set the style (within reason) and other developers to try to emulate that style (again, within reason, but it really is good form for any change you suggest to blend in with the code that's already there).  Here are some coding style documents you may find useful. This purports to be a (the?) [Linux kernel coding style](http://lxr.linux.no/linux/Documentation/CodingStyle) document. To the eyes of an old BSD hand, this looks a little cavalier, which is unsurprising. The BSD [Kernel Normal Form (KNF)](http://cvsweb.netbsd.org/bsdweb.cgi/src/share/misc/style?rev=HEAD&content-type=text/x-cvsweb-markup) is also worth a look. Many moons ago C programmers inside AT&T put together the Indian Hill C Style guide, which has been updated to yield a generically titled [Recommended C Style and Coding Standards](http://www.psgd.org/paul/docs/cstyle/cstyle.htm) guide. Of course, they contradict each other and probably every other coding style document on the planet, but you shouldn't let that bother you, and you should be able to detect a common core.  You may also wish to consult [How *NOT* to go about a programming assignment](http://www.di.uniovi.es/%7Ecernuda/noprog_ENG.html) by Agustín Cernuda del Río.  What really matters is:  Modularity  The scheduler functions are in a file full of scheduler functions and nothing else; each function is coded up once; etc.  Documentation  This can be read at multiple levels. For example, people can read a short file to find out if your program is right for them; can read about a module or package to see if they want to steal it for their own programs; can read about a function to decide whether they need to read the code, etc.  Maintainability  Including basic things such as **essentially never** putting a hardwired constant such as 1128 into a file, and **never** using it twice, and **never ever** putting 1127 into your code because it's 1128 - 1. But you knew all that from 15-213, right? Since you asked, the right way to handle 1128 would be  (GREY << COLOR\_SHIFT) | (ELEPHANT << ANIMAL\_SHIFT) | 16 /\* tons \*/  Dead Code  There are two very different kinds of "dead code" which students tend to turn in. One kind is 'debug code' which was briefly useful but never will be again, stuff like       // printf("!!!! joe = %d\n", 4 \* i / 0);  or         int threadstatus = THR\_RUNNING; // THR\_RUNNING|128  or      // if (threadid ==17) MAGIC\_BREAK;   Realistically, even if something like that helped you track down a bug once, you wouldn't re-activate it to find the next one. Since this code will never be run again, leaving it around can't do anything except distract and slow down your audience. Don't turn it in.  In the other direction, occasional assert()s are fine. In fact, sensibly done, they actually help document the code. See the section on asserts.  Overall, if debug code is genuinely likely to be useful to readers or maintainers, that argues in favor of keeping it, but the vast majority of what gets stuck in temporarily is just noise, and should be deleted.  Asserts  Asserts are statements that check that the assumptions that were made in your program are not violated. For example, let's say you have a variable \*nodeptr. If, by design, nodeptr can never point to NULL, then a statement like  ASSERT (nodeptr); or, in the Linux idiom,  if (!nodeptr)     BUG();  is a way of verifying that this assumption is not violated. This is useful in two ways. Firstly, it documents your assumption. Secondly, in future, if anyone modifying your code violates this assumption, (s)he is informed of this early, rather than having to discover this through a debugging session. Hardcore performance evangelists will argue that such code will almost never be executed, and that the extra conditional statement just serves to affect the branch prediction accuracy. People with such concerns should look at the [likely() and unlikely()](http://lxr.linux.no/source/include/linux/compiler.h#L60) preprocessor macros in the Linux kernel and the associated gcc builtin function [\_\_builtin\_expect()](http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html). However, people who *don't* have measurable performance problems should probably avoid littering their code with cryptic incantations without actual benefit.  **Descriptions of Commands**  Go here: [http://www.stack.nl/~dimitri/doxygen/commands.html](http://www.stack.nl/%7Edimitri/doxygen/commands.html) for descriptions of the commands mentioned.  **Putting it All Together**  Here is a short example showing all the elements together. This is an old version of the Project 1 starter code, presented in order to demonstrate doxygen (i.e., don't cut and paste this code into a current project!). As a note, README.dox is a text file with a single block of C-style comments in it.  **README.dox:**  /\*\*  @mainpage 15-410 Project 1  @author Harry Q. Bovik (hqbovik)  Here you should tell us about how your game works. How to play,  any special rules you have, etc. Also, explain any non-trivial  design decisions you make, like how your ball gets its position  updated, how you designed your buffer for readchar(), etc. You should  also comment on the stability of your code. Any big bugs should be listed  here. Basically, anything that you think we need to know in general about  your project should go here.  Any additional comments you want to make can go here. Did you like the  project? Was it too hard, too easy? My TA smells bad. Well, you get  the idea.  As a last little note, please make sure that all the documentation you  turn in is true and accurate. Otherwise, we will deduct points.  \*/  **console.h:**  /\*\* @file console.h  \* @brief Function prototypes for the console driver.  \*  \* This contains the prototypes for the console  \* driver and eventually any macros, constants,  \* or global variables you will need.  \*  \* @author Harry Q. Bovik (hqbovik)  \* @author Fred Hacker (fhacker)  \* @bug No known bugs.  \*/  #ifndef \_MY\_CONSOLE\_H  #define \_MY\_CONSOLE\_H  #include <video\_defines.h>  /\*\* @brief Prints character ch at the current location  \* of the cursor.  \*  \* If the character is a newline ('\n'), the cursor should  \* be moved to the next line (scrolling if necessary). If  \* the character is a carriage return ('\r'), the cursor  \* should be immediately reset to the beginning of the current  \* line, causing any future output to overwrite any existing  \* output on the line. If backsapce ('\b') is encountered,  \* the previous character should be erased (write a space  \* over it and move the cursor back one column). It is up  \* to you how you want to handle a backspace occurring at the  \* beginning of a line.  \*  \* @param ch the character to print  \* @return The input character  \*/  int putbyte( char ch );  /\*\* @brief Prints the string s, starting at the current  \* location of the cursor.  \*  \* If the string is longer than the current line, the  \* string should fill up the current line and then  \* continue on the next line. If the string exceeds  \* available space on the entire console, the screen  \* should scroll up one line, and then the string should  \* continue on the new line. If '\n', '\r', and '\b' are  \* encountered within the string, they should be handled  \* as per putbyte. If len is not a positive integer or s  \* is null, the function has no effect.  \*  \* @param s The string to be printed.  \* @param len The length of the string s.  \* @return Void.  \*/  void putbytes(const char\* s, int len);  /\*\* @brief Changes the foreground and background color  \* of future characters printed on the console.  \*  \* If the color code is invalid, the function has no effect.  \*  \* @param color The new color code.  \* @return Void.  \*/  void set\_term\_color(int color);  /\*\* @brief Writes the current foreground and background  \* color of characters printed on the console  \* into the argument color.  \* @param color The address to which the current color  \* information will be written.  \* @return Void.  \*/  void get\_term\_color(int\* color);  /\*\* @brief Sets the position of the cursor to the  \* position (row, col).  \*  \* Subsequent calls to putbytes should cause the console  \* output to begin at the new position. If the cursor is  \* currently hidden, a call to set\_cursor() must not show  \* the cursor.  \*  \* @param row The new row for the cursor.  \* @param col The new column for the cursor.  \* @return Void.  \*/  void set\_cursor(int row, int col);  /\*\* @brief Writes the current position of the cursor  \* into the arguments row and col.  \* @param row The address to which the current cursor  \* row will be written.  \* @param col The address to which the current cursor  \* column will be written.  \* @return Void.  \*/  void get\_cursor(int\* row, int\* col);  /\*\* @brief Hides the cursor.  \*  \* Subsequent calls to putbytes must not cause the  \* cursor to show again.  \*  \* @return Void.  \*/  void hide\_cursor();  /\*\* @brief Shows the cursor.  \*  \* If the cursor is already shown, the function has no effect.  \*  \* @return Void.  \*/  void show\_cursor();  /\*\* @brief Clears the entire console.  \* @return Void.  \*/  void clear\_console();  /\*\* @brief Prints character ch with the specified color  \* at position (row, col).  \*  \* If any argument is invalid, the function has no effect.  \*  \* @param row The row in which to display the character.  \* @param col The column in which to display the character.  \* @param ch The character to display.  \* @param color The color to use to display the character.  \* @return Void.  \*/  void draw\_char(int row, int col, int ch, int color);  /\*\* @brief Returns the character displayed at position (row, col).  \* @param row Row of the character.  \* @param col Column of the character.  \* @return The character at (row, col).  \*/  char get\_char(int row, int col);  #endif /\* \_MY\_CONSOLE\_H \*/  **console.c:**  /\*\* @file console.c  \* @brief A console driver.  \*  \* These empty function definitions are provided  \* so that stdio will build without complaining.  \* You will need to fill these functions in. This  \* is the implementation of the console driver.  \* Important details about its implementation  \* should go in these comments.  \*  \* @author Harry Q. Bovik (hqbovik)  \* @author Fred Hacker (fhacker)  \* @bug No know bugs.  \*/  #include <console.h>  int  putbyte( char ch )  {  return ch;  }  void  putbytes( const char \*s, int len )  {  }  void  set\_term\_color( int color )  {  }  void  get\_term\_color( int \*color )  {  }  void  set\_cursor( int row, int col )  {  }  void  get\_cursor( int \*row, int \*col )  {  }  void  hide\_cursor()  {  }  void  show\_cursor()  {  }  void  clear\_console()  {  }  void  draw\_char( int row, int col, int ch, int color )  {  }  **kernel.c:**  /\*\* @file kernel.c  \* @brief An initial kernel.c  \*  \* This file contains the kernel's  \* main() function.  \*  \* You should add your own comments to  \* replace this one.  \*  \* This is where you will eventually setup  \* your game board and have it run.  \*  \* @author Harry Q. Bovik (hqbovik)  \* @author Fred Hacker (fhacker)  \* @bug No known bugs.  \*/  /\* -- Includes -- \*/  /\* libc includes. \*/  #include <stdio.h> /\* for lprintf\_kern() \*/  /\* multiboot header file \*/  #include <multiboot.h> /\* for boot\_info \*/  /\* memory includes. \*/  #include <lmm.public.h> /\* for lmm\_remove\_free() \*/  /\* x86 specific includes \*/  #include <x86/seg.h> /\* for install\_user\_segs() \*/  #include <x86/pic.h> /\* for pic\_init() \*/  #include <x86/base\_irq.h> /\* for base\_irq\_master/slave \*/  /\*  \* state for kernel memory allocation.  \*/  extern lmm\_t malloc\_lmm;  /\*  \* Info about system gathered by the boot loader  \*/  extern struct multiboot\_info boot\_info;  /\*\* @brief Kernel entrypoint.  \*  \* This is the entrypoint for your kernel.  \* You will use this to test and debug your  \* drivers and it will eventually hold the  \* code for your game. Right now, it is  \* A tight while loop.  \*  \* @return Should not return  \*/  int main()  {  /\*  \* Tell the kernel memory allocator which memory it can't use.  \* It already knows not to touch kernel image.  \*/  lmm\_remove\_free( &malloc\_lmm, (void\*)USER\_MEM\_START, USER\_MEM\_SIZE );  lmm\_remove\_free( &malloc\_lmm, (void\*)0, 0x100000 );  /\*  \* Install interrupt handlers here.  \*/  /\*  \* initialize the PIC so that IRQs and  \* exception handlers don't overlap in the IDT.  \*/  pic\_init( BASE\_IRQ\_MASTER\_BASE, BASE\_IRQ\_SLAVE\_BASE );  lprintf\_kern( "Hello from a brand new kernel!" );  while(1);  return 0;  }  Click [here](http://www.cs.cmu.edu/%7E410/doc/doxygen/index.html) for the corresponding html document that is created by doxygen. |