Handout 3: Naming and Style Conventions

Creating software without naming and style conventions is equivalent to building homes without building codes. Without conventions, every programmer in an organization does their own thing. Problems arise whenever someone else has to look at the code. For example, suppose the same module is written by two different programmers. The code of one programmer takes 1 hour to understand and verify, while the same code by the other programmer takes 1 day. Using the first version instead of the second is an 800 percent increase in productivity!

The primary factor that affects the readability of code is the presence of naming conventions. If strict naming conventions are followed, simply looking at a symbol quickly tells the reader what the symbol is, where it is defined, and whether it is a variable, constant, macro, function, type, or some other declaration. Such conventions must be posted, just as a legend must appear on a design diagram, so that any reader of the code knows the conventions.

This section describes the naming and style conventions that are enforced in the *Software Engineering for Real-Time Systems (SERTS) Laboratory* (*www.ece.umd.edu/serts*) at the University of Maryland and used in the development of Echidna.

1. Naming Conventions

Naming conventions are extremely important. Software maintainability is directly related to the use of good naming conventions. By looking at any symbol name, you should immediately be able to distinguish between constants, variables, macros, and functions, as well as whether something is local, global, internal, or external. Consistently using the naming conventions in Table 1 will yield all of this information from simply the symbol name.

1.1 Pairing Function Names

You should always name functions such that each exported function has a converse, as shown in Table 2. By defining functions as pairs, there are two important benefits. It forces you, the designer, to ensure completeness. It also allows you to create the two portions simultaneously and use each part to test the other part.

Make sure that pairings are consistent. For example, if the conventions shown in Table 2 are used, make sure the converse of send is not read and that the converse of create is not finish. If you are creating the code for reading and writing at the same time, you can test both pieces of code by writing from one process and reading from the other. It is worthwhile to always use the same conventions for similar components, especially if they are in different modules.

1.2 Compounding Function Names

To allow for further decomposition, put names in “big object” to “small object” order for compounded function names instead of in the order that it would naturally be read. For example, if module *xyz* has a secondary structure *xyzFile_t*, then functions that operate on that structure should be named the following:

DRAFT

Table 1: Naming conventions for files, variables, types, and functions in C.

Symbol	***Description***	***Symbol***	***Description***
xyz.h	**File** containing header info for module *xyz*. Anything defined in this file MUST have an "xyz" or "XYZ" prefix and be exported by the module.	*xyz.c*	**File** containing code for module *xyz*.
xyz_t	**Primary data type** for module *xyz*. Define in *xyz.h*	*Abcde_t* *AbcdeFgh_t*	**Internally-defined data type**. Define at top of *xyz.c*. Note that data types and functions for internal use start with a capital letter.
xyzAbcde_t *xyzAbcdeFgh_t*	**Secondary data type** "Abcde" for module *xyz*. Define in *xyz.h.* All *enum*'s should be *typedef*'d with this format.	*_ABCDE* *_ABCDE_FGH*	**Internal constant**. Define at top of *xyz.c*. E.g., if ABCDE_FGH is used instead of _ABCDE_FGH, it implies module *abcde*.
XYZ_ABCDE *XYZ_ABCDE_FGH*	**Constant** for module *xyz*. For **internal or external** use. Define in *xyz.h*.	*_ABCDE()*	***#define*'d macro** for **internal** use only. Define in *xyz.c*.
XYZ_ABCDE()	***#define*'d macro** for module *xyz*. For **internal or external** use. Define in *xyz.h*.	*_abcde* *_abcdeFgh*	**Internal global variable**. Define as *static* at top of *xyz.c.*
xyz_abcde *xyz_abcdeFgh*	**Exported global variable** defined for module xyz. Declare as extern in xyz.h and define in xyz.c. Global variables should be AVOIDED!	*abcde*	**Local variable**. Define inside a function. Also define fields within a structure using this convention.
xyzAbcde() *xyzAbcdeFgh()*	**Exported function** "Abcde" defined in module *xyz*. Declare as *extern* in *xyz.h* and define in *xyz.c.*	*Abcde()* *AbcdeFgh()*	**Internal function**. Declare prototype as *static* at top of *xyz.c.* Define function at bottom of *xyz.c* after declaring all exported functions.

Table 2: Naming conventions for pairing function names

xyzCreate	↔	xyzDestroy	xyzAlloc	↔	xyzFree	xyzRead	↔	xyzWrite
xyzInit	↔	xyzTerm	xyzOpen	↔	xyzClose	xyzSnd	↔	xyzRcv
xyzStart	↔	xyzFinish	xyzUp	↔	xyzDown	xyzStatus	↔	xyzControl
xyzOn	↔	xyzOff	xyzGo	↔	xyzStop	xyzNext	↔	xyzPrev

```
xyzFileCreate
xyzFileDestroy
xyzFileRead
xyzFileWrite
```

and not

```
xyzCreateFile
xyzDestroyFile
xyzReadFile
xyzWriteFile
```

Note that the last word for any function name should be the verb that represents the action performed by the function. The middle words are typically nouns to represent the object(s) on which the verbs act.

1.3 Matching names to modules

This convention makes it obvious that xyzFile is a sub-structure of the xyz module in the first part of the previous example. In the second part, it is not at all obvious. Furthermore, if module xyz grows and you decide to further decompose it, then it will be easy to move the entire xyzFile sub-structure and corresponding functions to a separate module (e.g., xyzfile). A global search-and-replace of xyzFile to xyzfile would result in all the necessary changes, and within a few minutes, the decomposition is complete. If this naming convention is not followed, it will take much longer to revise all of the names for use in the new module.

1.4 Abbreviating function names

While it is acceptable to have an abbreviated module name because the name serves as a prefix to everything, only use obvious abbreviations for function names. If an obvious abbreviation is not available, then use the full name. If an abbreviation is used, then use it everywhere in the project.

For example, it can be a convention to always use xyzInit as the initialization code for module xyz; and never to use xyzInitialize. As another example, use either snd and rcv, or send and receive, but don't intermix the two. Examples of other common abbreviations include intr for interrupt, fwd for forward, rev for reverse, sync for synchronization, stat for status, ctrl for control.

On the other hand, an abbreviation like trfm, supposedly short for transform, is not recommended, because the abbreviation is not obvious and thus readability is compromised. In such a case, it would be better to choose the function name without abbreviation, *xyzTransform()*. As another example of over-using abbreviations, consider the difference between *xyzFileCreate()* and *xyzFilCrt()*. The second one uses uncommon abbreviations, and it is difficult to follow when reviewing the code during the software maintenance phase. It is much better to use slightly longer names, and avoid confusion as to what the function does.

1.5 Global variables

Global variables are often frowned upon by software engineers, as they violate encapsulation criteria of object-based design and make it more difficult to maintain the software. While those reasons also apply to real-time software development, it is even more crucial to avoid the use of global variables in real-time systems.

In most RTOS's, processes are implemented as threads or lightweight processes. Processes share the same address space to minimize the overhead for executing system calls and switching contexts. The side-effect, however, is that a global variable is automatically shared among all processes. Two processes that use the

same module with a global variable defined in it will share the same value. Such conflicts will break the functionality. Thus, the issue goes beyond just software maintenance.

Many real-time programmers use global variables to their advantage, as a way of obtaining shared memory. In such a case, however, care must be taken, as any access to it must be guarded as a critical section to prevent problems due to race conditions. Unfortunately, most mechanisms to avoid race conditions (e.g., semaphores), are not real-time friendly, and they can create undesired blocking. The alternatives, such as the priority ceiling protocol, use significant overhead. You can find more information about critical sections and race conditions in any operating systems textbook.

2. STYLE CONVENTIONS

Using naming conventions is only the first step in creating reusable software. You can make your code even easier to read and maintain by following proper guidelines for indentation, spacing, and commenting.

2.1 Indentation

Indentation must always be 4 spaces. Nested and sub-statements must be indented properly.

An example of proper indenting and location of parentheses and braces:

```
void funcname(int a) {
    code goes here
    code goes here
    code goes here
} // end funcname
```

- No space between **funcname** and (
- Always use a space between) and {
- A comment can be used to show end of function

2.2 if() and if()-else statements

```
if (condition) {
    code goes here
}

if (condition) {
    code goes here
} else {
    else code goes here
}
```

- Always use a space between **if** and **(**, or it will look like a function name.
- Always use a space between **)** and **{**

For if-else statements, if you use parentheses on the **if** part, then use it on the **else** part too, and vice versa:

Don't write:

```
if (condition) {
    line1;
    line2;
} else
    line3
```

Do write:

DRAFT

```
if (condition) {
    line1;
    line2;
} else {
    line3
} // end if-else
```

Similarly:

Don't write:

```
if (condition)
    line1;
else {
    line2:
    line3
} // end if-else
```

Do write:

```
if (condition) {
    line1;
} else {
    line2:
    line3
} // end if-else
```

Reason: If you need to add any code later on, it is easy to make a mistake and not add the braces when you add a line of code.

The only time you don't need the braces is with short one-liners. However, having braces can be helpful even then, similar to the example in the **for()** loop below. For anything more than one-liners, always use braces.

For **if()-else if()-else**, always use braces:

```
if (condition1) {
    line1;
} else if (condition2) {
    line2:
    line3;
} else {
    line4;
} // end if-else
```

- A comment can be used at the end of the last **if()**

2.3 while() and do-while() loops

```
while (condition) {
    code goes here
    code goes here
    code goes here
} // end while
```

- Formatting is similar to **if**
- A comment can be used to show the end of the loop

```
do {
   code goes here
   code goes here
   code goes here
} while (condition);
```

- A semi-colon after **)** is necessary.
- Always use a space between **do** and **{**
- Always use a space between **}** and **while**
- Always use a space between **while** and **(**

2.4 for() loops

An example of proper indenting and location of parentheses, braces, and semicolons:

```
for (init;condition;increment) {
   code goes here
   code goes here
}
```

Always use braces when nesting loops, for all loops except possibly the innermost loop, even if they are not needed:

Don't write:

```
for (i=0;i<10;++i)
   for (j=0;j<10;++j)
      a[i] += j;
```

Do write:

```
for (i=0;i<10;++i) {
   for (j=0;j<10;++j)
      a[i] += j;
}
```

Reason: Suppose we realize, oops, we forgot to **init a[i]**. Modify code:

```
for (i=0;i<10;++i) {
   a[i]=0;
   for (j=0;j<10;++j)
      a[i] += j;
}
```

Now, we are forced to add the braces. It is safer to just always use braces.

2.5 switch() statements

An example of a **switch** with only one line per case statement:

```
switch (value) {
   case 0:     line0; break;
   case 1:     line1; break;
   case 2:     line3; break;
   default:    line3; break;
} // end switch
```

An example of a switch with small number of short lines per **case** statement:

```
switch (value) {
   case 0:
      line1;
      line2;
      line3;
      break;

   case 1:
      line4;
      line5;
      line6;
      break;

   default:
      line5;
      line6;
      break;
} // end switch
```

Following is an example of a **switch** with a large number of lines per case, especially when nesting within the **case** statements, or if lines are very long. This method minimizes the white space from indenting many times:

```
switch (value) {
   case 0: {
      line1;
      line2;
      line3;
   } break;

   case 1: {
      line4;
      line5;
      line6;
   } break;

   default: {
      line5;
      line6;
   } break;
} // end switch
```

2.6 Blank Lines

Use blank lines wisely to organize your program into blocks. Think of your code as you would a well-written document: one thought per paragraph, and each paragraph perhaps containing multiple sentences. In code, there is one thought per ‘block’, and each block may contain multiple lines of code. Examples of a block include a loop, a complete if-then statement, or a set of assignments that belong together.

For example:

DRAFT

```
int main(void) {
   int a,b,c;
   float y,z;
                        blank line
   while (condition) {
      statement1;
      statement2;
      statement3;
   }
                        blank line
      // comment belongs to block, so no blank line
      // between it and the code.
   for (...) {
      statement4;
      statement5;
   }
                        blank line
   if (condition) {
      code;
   } else {
      more code;
   }
                        blank line
   return 0;
}
```

Too many blank lines cause your program to span too many pages when you print it out or require extra scrolling when editing. Not enough blank lines makes it difficult to read the code.

2.7 Commenting

You can use either the **/* */** or the **//** method. The **//** method is usually much easier to type and is highly recommended.

Comments should expose the thought process behind what a line or block of code is doing, rather than repeat what the code is doing. Saying exactly what the code is doing is often redundant. For example, here are extreme cases of bad comments that are seen very often:

```
count+=2;        // increment counter by 2.
return (sum);    // Return the sum
```

Better comments:

```
count+=2;        // increment by 2 because we only want even numbers
return(sum);     // sum could be negative if error in input data
```

If a comment would be redundant, and there is really no "thought" that needs to go behind it, then don't comment it. Useless comments are worse than no comments.

Comments are always indented with respect to the code, so that the left-most column is the code. This is accomplished in either of two ways:

- Always start at column 40 (or at column 32. Just make sure it starts at the same column throughout the code), so that your code looks like two columns.
- The left column is code, the right column is comments.

E.g. **Bad commenting**:

```
// comment goes here in Column 8, but code in column 12
abc = def;
// the problem is that the code gets lost.
```

E.g. Good commenting:

```
abc = def;                    // comment goes here in Column 40

   // long comment can go here, indented immediately before the lines
   // of code to which it pertains.
def = pqr+uvw;
```

When a function, loop, or if-else statement is more than 10 or so lines, or you have multiple levels of indentation, put a // **end xyz** comment after the closing brace. For example:

```
void funcname(int mmax, int nmax, int pmax) {
   int m,n,p;

   for (m=0;m<mmax;+m) {

      for (n=0;n<nmax;+n) {
         for (p=0;p<pmax;+p) {
            code goes here
         } // end for p
         more code goes here
         more code goes here

      } // end for n
      more code goes here

   } // end for m

   more code goes here

return;
} // end funcname
```

Define code in “paragraphs.” That is, each thought is single-spaced (like a small loop), and double-spaced between different “thoughts”. Put a comment to begin each paragraph that explains what the block of code is doing in high-level terms.

Example:

```
static void _sample_function (void) {
       // single space variable declarations
       // double space after them.
   int i,j;
   int a[N],b[N];

       // single space here
   for (i=0;i<N;++i)
       a[i]=i;

       // then double space before next thing, but single space
       // the "paragraph".
   if (b[i] < a[i]) {
          // you can put a comment here
       printf("b[i] is smaller");
   } else {
       printf("b[i] is bigger");  // or put a comment here
   }

   return;
}
```

Note the **if()-else** bracket and indenting. Another example:

```
      // comment for entire if-else structure
if (condition1) {
       // comment for condition 1
   code for condition 1
   code for condition 1
   code for condition 1
} else if (condition 2) {
       // comment for condition 2
   code for condition 2
   code for condition 2
   code for condition 2
} else {
       // comment for else part
   code for else part
   code for else part
   code for else part
}
```

2.8 Expressions and Conditions

Spaces in equations and conditions are optional. Use them to help you quickly see the order of operations. You can also add parentheses, even if they are not needed.

Examples:

Hard to read:

```
x = 4 * y + 3 * z / 2;
x=4*y+3*z/2;
```

Easier to read:

```
x = 4*y + 3*z/2;
```

DRAFT

Hard to read:

```
if ( x == 3 * z + 4&&y == 2 - z) {
```

Easier to read:

```
if ( (x == 3*z + 4)  &&  (y == 2-z) ) {
```

If you have really long conditions, put one condition per line, or format it in such a way that makes sense. E.g.:

```
if ( (long-condition-one)   &&
   (long-condition-two)   ||
   (long-condition-three) ) {
```

3. SUMMARY

The goal of strictly following naming and style conventions is to minimize the maintenance of the software. That is, sometime after the code is written, someone will need to look at the code, determine what it is doing, and make one or more changes to it. The difference between code that follows strict conventions versus code with no conventions could mean the difference between taking 1 hour versus 1 day to make the change; this is an 800% productivity factor!

It is difficult to prove that following these conventions lead to those kinds of savings during the maintenance phase. But after experiencing both methods, you will quickly learn that conventions are important. During this semester, you ***must*** following these conventions ***exactly***. After one semester of using them, you will be on your own; if you don't like them, don't use them. But more than likely, most of these conventions, with maybe a few of your own minor customizations, will be adopted.

The one guarantee I make: you will learn to appreciate them when you are required to read and modify someone else's code, and they did not follow any such conventions!