Suggested coding rules for SCT and Pixel ROD software

Draft 2.3

J.C.Hill

1 March 2000

These notes are intended to give guidance on coding rules to be followed when writing software for the SCT and Pixel RODs. They are not meant to be proscriptive – if there is a good reason to break a rule, then do so – I am not a great fan of rigid structures myself. However, you should understand the consequences when you do this. The rules have been modified from the "C++ Coding Standard Specification" at URL

http://spider.cern.ch/Processes/Programming/CodingStandard/c++standard.pdf.

In the immediate future, we will program in C. However, the final code will almost certainly be in C++, and ATLAS Online is committed to the above coding standards, so we should try to follow the rules as much as possible from day 1. I've split the following notes into the same sections as used in the above note, but the numbering scheme is my own.

The intention is to ensure portable code. An explicit exception to this is the code for the DSPs, where it is likely that some of the rules will be ignored in the interests of efficiency and the special requirements for the coding of these devices. There are occasional references to this in the rules.

Though not strictly a coding rule, I suggest that any code be kept with a CVS repository. As the systemtest software development has demonstrated, this provides a convenient machine-independent way of storing software, and can be used with (e.g.) Visual C++ without major problems.

A. Naming

A1. Don’t use cryptic names for functions or variables.

> It is much more readable for non-experts and newcomers if the names are pronounceable and meaningful. Also it is helpful in understanding the code.

A2. Function names should start with text that indicates the part of the system they act on.

> Functions that interact with the ROD should have names like “rodxxxx” for example (the exact naming scheme to be used is open for discussion).

A3. Don’t use identifiers that start with underscore.

> Operating systems have a tendency to use names starting with underscore as internal names, and they can also be reserved keywords in C.

A4. Use uppercase for parameters (e.g. in #define) and typedefs, and lowercase with capitals (e.g.myVariable) for variables and functions.

> This is close to my convention (but modified after input from interested parties), rather than from the C++ coding rules. It is certainly invaluable to be able to distinguish constants from variables in code. An alternative that achieves the same goal (but might make more readable code) is of course acceptable.

B. Coding

B1. Keep internal and external definitions in separate header files.

> The internal definitions should not be required by the users of the library (if they are, then the code design is faulty), so only the header file of external definitions should contain information needed by the user.

B2. Use “__STDC__” to handle function primitive definitions in a portable way.

> This shouldn’t really be necessary as most modern compilers are ANSI-compatible. However, if you encounter a K&R compiler, the function primitives cannot specify the arguments, and this case is best handled using the __STDC__ reserved keyword and a #ifdef or #if defined (rather than removing the arguments from the standard definitions). Quite a few ANSI-C compilers will accept function primitives without arguments, but they then tend to assume the arguments are of type **int**, which will cause severe problems if they aren’t.
> This rule doesn’t apply to the DSP coding, which is specific to one platform.

B3. Avoid unnecessary inclusion.

Don't include a header file twice – design the code to ensure that this doesn't happen. If it becomes unavoidable, you should add multiple-inclusion protection to the header files.

B4. Do not change a loop variable inside a **for** loop.

This can be extremely difficult to debug, it makes it hard to understand the flow of control, and is error-prone. Change the loop variable in the expression executed after each iteration.

B5. Always use braces with **for**, **while** etc.

Even if the block is a single statement, this makes the code easier to read and less prone to mismatched braces. I.e.

```
if ( condition) {
   statement;
}
```

is better than

```
if ( condition ) statement;
```

and infinitely better than

```
if ( condition )
   statement;
```

B6. Always have a **default** in a **switch.**

Even if the default can never be reached, it should be present to catch additions.

B7. An **else** is always recommended for an **if** statement.

Certainly an **else** should be provided if there are any **else if** blocks. For a simple **if**, it is probably overkill, but certainly recommended in the C++ coding specification.

B8. Do not use **goto**.

You should use **break** or **continue.** I would allow an occasional exception where the code has to become over-complicated to avoid a **goto** – others would suggest this is just bad coding.

B9. Keep functions simple.

If a function is overly complex, with lots of control flow primitives, it will be difficult to understand and maintain. It would probably be better to split it up into a number of simpler functions.

B10. Declarations which set variable values should be in well-defined places.

Spreading such declarations all over the code makes it hard to maintain – best to put them all (for example) at the start of a function.

B11. Declare non-pointers and pointers of the same size on different lines.

Mixing (for example) **int** and **int*** declarations on the same line can lead to errors and confusion - you may want to change all the **int**s to **long**s, but leave the pointers alone. Some would advocate each variable should be on a separate line.

B12. Use explicit rather than implicit type conversion.

In general, type conversion should be avoided. Where they are necessary, they must be done explicitly – the code will be more readable and also easier to debug. An exception to this can be the generic pointer (i.e. void *ptr) – but I prefer to explicitly type pointers anyway.

B13. Always pass by value unless the function modifies the argument.

Structures and arrays are general exceptions to this rule.

B14. Ensure a **malloc** or **calloc** is matched by a **free.**

The use of **malloc** tends to be deprecated precisely because too many programmers forget to **free** the memory, hence causing leaks. Also make sure the returned address from the **malloc** is also used in the **free** – operating system behaviour when they don't match is unpredictable (some release the memory if the address is somewhere in the range of allocated memory, some ignore the request, some crash the program).

B15. Test for all errors from functions.

After a function return, all errors should be checked-for and handled. An exception to this might be if the error is known to be benign and the function handles it properly – but this would normally be rare.

B16. Put the error code in the function return.

This is personal convention – it is equally valid to have the error returned as a function argument. An error code = 0 should indicate no errors, and (again by convention) I suggest <0 for error returns. >0 has tended to be unused in my code, but I envisage these codes indicating successful function execution with some proviso or other.

B17. Keep the error codes used consistent where possible.

For example, -1 might indicate the hardware being accessed hasn't been defined in all functions addressing that hardware.

B18. Don't do pointer arithmetic that relies on the pointer length.

E.g. don't do

```
short *ip1, *ip2;
ip1 = (short *)BASE_ADDRESS;
ip2 = ip1 + 1;
```

which is confusing and prone to misunderstanding (it also assumes the length of a **short***, which may vary from machine to machine, though I don't address that). Using

```
ip2 = (short *) (BASE_ADDRESS + 2);
```

is clearer for the reader.

B19. Don't reuse code by cutting and pasting to several locations.

Better to put such a piece of code in a function and call it when needed – you don't then have to remember to change the code in several places. It also makes it easier to reuse the code.

B20. Don't write clever, cryptic code unless there are overriding reasons.

There may be a need to write tight code in locations where performance is critical, but in general it is more important for the code to be comprehensible to the casual reader.

B21. Write in ANSI C.

Do not use extensions to ANSI C that may be offered by some compilers, nor take advantage of defaults that may not be universal.

B22. Put non-portable code together as much as possible.

Typically this will be the machine-dependent access to the hardware – this should be kept in a minimal number of functions, with the use of **#ifdef** to isolate the code.

B23. Include header files in a standard way.

Use

#include <stdio.h>

for standard C header files, and

#include “myheader.h”

for application header files.

B24. Don’t rely on how memory allocation might be done.

Don’t assume for example that **int** is 32 bits, or that structures will be aligned on a particular address boundary. Use **sizeof** wherever necessary.

B25. Be extremely careful with casting pointers.

Casting a **short*** to a **int*** could produce illegal addressing if the address is not on an **int** boundary.

B26. Beware of machine precision and rounding errors when using **float**s.

To compare two **float**s, use something like

if (fabs(value2 – value2) < TOLERANCE) ...

where TOLERANCE is chosen to be appropriate to the machine accuracy and the required accuracy of the equality.

B27. Do not assume order of argument evaluation.

For example, func(i++,x(i)) is not well-defined.

B28. Use the standard library rather than system calls.

This makes the code easier to port.

B29. Code must protect against exceptions (eg. underflow, overflow, divide by zero).

Do not rely on the operating system to do the recovery – check in the code. Divides by zero and overflows often indicate logic problems anyway.

C. Style

C1. Avoid very long statements.

> These can be difficult to read – split a long statement into 2 or 3 shorter ones if at all possible.

C2. Function declarations should have dummy arguments with meaningful names.

> This makes it easier to read, and easier to spot errors.

C3. Comments – use /* */.

> Some compilers accept the C++ // notation, but others don't, and changing all comment lines to the appropriate format is tedious and error-prone. The /* / form has a number of inadequacies, most notably the lack of nesting, but for portability it should be used. As a corollary, it is preferable to remove blocks of code using **#if 0** and **#endif**, as this avoids the nesting problems of /* */.

C4. Be generous with comments (in English).

> Plenty of comments are better than a few, but they should also be descriptive – a cryptic phrase is usually not helpful in understanding the code.

C5. Describe each function in a comment section.

> My convention is to put a comment section at the start of each function – alternatively put the descriptions at the start of the file. The details remain to be worked out.

C6. Add the name of the preprocessor directive as a comment to **#else** and **#endif** directives.

> It makes it easier to understand the code if there is a comment to indicate which directive these statements relate to.