Some misc. tips for the kernel programmer.

This document contains information not found elsewhere. Some good practices and other useful tricks are reported too.

Kernel

Check for erroneous memory allocations on each attempt to allocate memory. All errors should be reported back to the application program. In the case of errors, it is up to the application programmer to deal with it. Further, try to minimize damage by cleaning up possibly earlier memory allocation before returning.

Stack grows from higher to lower addresses.

Turn off interrupts on memory handling (malloc/calloc/free is not reentrant). Check for other non-reentrant functions (such as printf) and be careful with those.

The data structure called TCB is internal.

One good programming practice is to split hardware dependant stuff from rest of code

Be careful with the use of **printf()**. If you want to use **printf()** be sure to turn off interrupts before the call (printf uses memory allocation routines in the background, and is a huge function memory vise).

Using LED's for debugging is generally a good idea.

Application program

Arguments sent as pointers (to e.g. send_wait()) should be declared **static** in functions (or global variables).

Always check return values on each function i.e. DEADLINE_REACHED and possibly memory allocation failures.

Be careful with the use of **printf()**. If you want to use **printf()** be sure to turn off interrupts before the call.

Testing

Testing of the kernel can be divided in two parts, kernel internal testing, and kernel external testing.

In the **kernel internal testing** the modules constituting the kernel is tested function by function. This testing is performed early in the development process.

Start the testing with all interrupts turned off. When the interrupts are turned off the debugging complexity is increased drastically.

Module testing is a good thing. Test each of your modules individually before you test them together.

Kernel external testing should test the kernel so each kernel function is behaving according to the specification. The tests are written as application programs.

Make test suits. I.e. lump many tests together in a "big test". Never delete a good test program. Writing a good test suit is at least as hard as writing the kernel it self.

Stress testing can be made to find memory holes, and "hard to find" bugs.

Be sure to test all error codes, as well as passing data in the message passing routines.

Development

Turn off optimization and software pipelining in project.

A macros such as this one can help sometimes:

```
#define OS_ERROR(a) if(!(a)) {
           printf("OS-ERROR in file: (%s), on line: %d\n",
           __FILE__, __LINE__); while(1); }
```

Lint is a nice program that assist in the hunt for bugs. Try the free lclint (UNIX/WIN32) at:
http://lclint.cs.virginia.edu/

A nice free code documentation system, Doxygen, can be found here (UNIX/WIN32):
http://www.stack.nl/~dimitri/doxygen/

Another one, Autoduck, can be found here (WIN32):
http://www.helpmaster.com/zip/autoduck.zip

Good coding styles in C is reported here:
http://www.cs.umd.edu/users/cml/cstyle/indhill-cstyle.html
(Check “The Ten Commandments for C Programmers” at the bottom of that page)