Q&A

Q: The Autodoc for the Intuition function `ActivateWindow()` *says:*

```
* RESULT
*   V35 and before: None.
*   V36 and later: returns zero if
*   no problem queuing up
*   the request for deferred action
```

Is this true?

A: No, it's actually none, even under V36, V37, etc.

Q: If I use the trackdisk.device to write on a section of a write-protected floppy, when I read that section of the floppy, the data I wrote appears to be there. What is going on?

A: When writing to a write-protected floppy using the *trackdisk.device*, the *trackdisk.device* does not write on the disk, but it does write to the disk buffer in memory, which is what you are reading. This is a bug.

In order to make sure that the state of the disk is as you expect after a failed write, you should do a `CMD_CLEAR` to make it flush the buffer.

This is not normally a problem with the file system, since it checks write-protect on every insertion, and doesn't attempt writes to write-protected disks.

Q: What's wrong with calling the Exec function `AllocMem()` *using the* `MEMF_REVERSE` *flag?*

A: Under normal conditions, the `MEMF_REVERSE` flag makes `AllocMem()` search Exec's free memory list in reverse order. If the `MEMF_REVERSE` allocation fails due to low memory, the OS will either clear low memory or get stuck in an infinite loop (or, when *Enforcer* is running, it will cause a number of Enforcer hits!)

Workaround:

If you really want to do this and don't want to have to do the `MEMF_REVERSE` yourself, you can do the following workaround. It is not very fast but if your allocations are rare, it will not be too bad.

```
Forbid();
if (mem=AllocMem(size,
    <normal flags, no MEMF_REVERSE> ))
{
     flags=TypeOfMem(mem);
     FreeMem(mem,size);
     mem=AllocMem(size,
                  MEMF_REVERSE|flags);
}
Permit();

if (mem)
{
     /* Got the memory... */
}
else  /* Failed! */
```

Warning: This will only work if there is only one memory list with the attributes given (which is usually the case with `MEMF_CHIP`). If there are more than one memory lists, `AllocMem()` may work in the second list while the reverse will fail in the first (and crash).

Warning: Tools such as *Memoration* can cause errors in the second `AllocMem()` from the workaround above.

This bug exists in all versions of Exec to date.

Q: The Autodoc for the DOS function `InternalLoadSeg()` *states that* `ReadFunc()` *takes it arguments in registers d1/a0/d0. Is that true?*

A: No, it actually takes them in registers d1/d2/d3.

Q: Does the input.device ever try to lock the blitter?

A: Sure, all the time. All input handlers run on the *input.device* task, and the grandest input handler of all is called ‘‘Intuition’’. When an application calls Intuition, part or most of the function executes on the application’s task, but part may execute on the *input.device* task. All user-initiated actions (e.g., dragging a window) always happen on the *input.device* task. This means the *input.device* does rendering, layer operation, copper-list and ViewPort operations, etc.

Q: I program in assembler. I hear that many software compatibility problems are traced to assembler application code containing a hidden misuse of a register. How can I check for this?

A: While programming in assembler, it is not uncommon for programmers to forget to refresh a scratch register (d1/a0/a1) after a system call, or even look at the wrong register for the result of the system call. These registers contain leftover values from the internal code of the system function, which may *happen to be* the original value which was in the register before the call, or *may happen to be* a copy of the result (d0). If this is the case, the assembler application’s register misuse bug may have no symptoms or only sporadic symptoms under one version of the OS. However, the slightest change to the system function’s internal code can drastically change the leftover values in the scratch registers. In some cases, *one* instance of register misuse can render a major application *unusable* under a new version of the OS.

Here is a simple example of such a hidden coding error:

```
* GfxBase already in A6.  Both SetDrMd and
* SetAPen expect a rastport pointer in A1
    MOVEA.L rastport, a1   * Put rport in A1
    MOVE.L  #JAM1, d0      * JAM1
    JSR _LVOSetDrMd(a6)    * set draw mode
    MOVE.L  #3, d0         * pen 3
* Here’s the problem: the programmer assumes
* A1 still contains the rastport pointer.
* Since A1 is a scratch register, SetDrMd
* may have overwritten A1 with garbage, so
* SetAPen will get a bogus RastPort pointer.
    JSR _LVOSetAPen(a6)    * set pen
```

If the rastport pointer passed in A1 *happens to be* left over in A1 after the call to `SetDrMd()`, the call to `SetAPen()` will succeed. If not, the call to `SetAPen()` will trash memory, and possibly crash the system.

If you program is assembler, you *must* test your code with *Scratch* (by Bill Hawes) to test for misuse of registers after system calls. *Scratch* and the script that installs it (*scratchall.script*) are on the Software Toolkit II disk of the *2.0 Native Developer Update*. It may also be found with the debugging tools on the Denver/Milan Devcon disks. *Scratch* will *invalidate* the scratch registers upon the exit from each system library call. If a program is failing to refresh a scratch register or looking at a scratch register improperly, you may get Enforcer hits (if you are running *Enforcer and Scratch*), and/or Mungwall hits, and/or obvious misbehavior or crashing of your code.

Use the *scratchall.script* to install *Scratch* before starting *Mungwall*. When running running this script watch out for the scripts with a backtick. Some versions of the script have a backtick (‘) at the beginning of a early comment line. The script will not execute unless the backtick is replaced with a semicolon (;).