Q&A

Q: Does GadTools support custom string and integer gadget editing hooks?

A: Yes, here is the tag, which is new for V37:

```
#define GTST_EditHook GT_TagBase+55
```

Which is a custom string gadget edit hook (StringExtend->EditHook) for this gadget. GadTools will allocate the StringExtend->WorkBuffer for you. Defaults to `NULL`.

Also, two tags for string and integer gadgets were implemented in GadTools but weren't documented. See < *intuition/gadgetclass.h*>:

`STRINGA_Justification` - (new for V37) Controls the justification of the contents of a string gadget. Choose one of `STRINGLEFT`, `STRINGRIGHT`, or `STRINGCENTER` (Defaults to `STRINGLEFT`).

`STRINGA_ReplaceMode` (BOOL) - (new for V37) If `TRUE`, this string gadget is in replace-mode. Defaults to `FALSE` (insert-mode).

Q: I am using ReadArgs() to scan an argument list. When ReadArgs() finishes with the RDArgs structure, it leaves a value in rda_Buffer. Should it do that?

A: No, this is a bug. If you supply a value for rda_Buffer, it should `NULL` rda_Buffer, so you don't have to do it manually before passing an RDArgs struct back into ReadArgs().

Q: I'd like to be able to take a memory region that I previously allocated with the system's AllocMem() function, and then later, sometime after it's been used a bit, FreeMem() part of it (always the last part of it), to shrink the memory region. Is this OK?

A: This is *not* supported. While you might be able to make it seem to work, the OS does *not* support such behavior and our standard testing tools will not like this behavior. Do not do this. Such things are not possible if we want to eventually move to a more complex memory system (including virtual memory and/or protected memory).

Q: I start a program with `SYS_Asynch` and I have opened a file ''CON:0/0...'' as parameter for `SYS_Output`. The program starts and the console window opens, but if I enter a `CTRL-C` into the window, the main program gets the `CTRL-C` and not the program started via System(). Is there a way to make sure the program started via System() gets the `CTRL-C`?

A: All you have to do is set both input and output to the same file by setting `SYS_Input` to the CON: handle and setting `SYS_Output` to `NULL`. System() and Execute() only redirect signals from the input stream to themselves (with a few other minor conditions). They do this with the new `ACTION_CHANGE_SIGNAL` packet. On synchronous (`SYS_Synch`) System() calls, after the System() call returns, it redirects signals back to the parent.

Q: Is there an easy way to find the name of a volume (like Workbench:) from its device name (like DF0:)?

A: Yes...

```
if (fib=AllocMem(sizeof(struct
FileInfoBlock),MEMF_PUBLIC))
{
    if (lock=Lock("DF0:",ACCESS_READ))
    {
        if (Examine(lock,fib))
        {
            printf(
                "DF0: - Volume is %s\n",
                fib->fib_FileName);
        }
        UnLock(lock);
    }
    FreeMem(fib,sizeof(struct
FileInfoBlock));
}
```

Q: What is the correct way to determine whether a device node in the DosList is associated with a disk type device like a floppy or hard disk?

A: Under 1.3, try to lock the root of the device. Under 2.0, use IsFileSystem().

Q: How do you obtain information about the object a hard link references?

A: The following code obtains a lock on the object a hard link references:

```
BPTR l;
LONG success = FALSE;
char buff[256];

l = Lock(hardlinkname,SHARED_LOCK);
if (l)
    success =
        NameFromLock(l,
        &buff[0],
        sizeof(buff));
```

Q: I am working on a program that runs CLI based commands from a separate process that is launched from the main program. This process is given a configurable stack size. Execute() (under 1.3) or System() (under 2.0) use a default stack size of 4K. I can explicitly give System() a stack size but how do I change the stack size that Execute() uses?

A:

```
char cmdbuff[whatever];

sprintf(cmdbuff,
        "stack %ld\n%s",
        stacksize,
        command);
Execute(cmdbuff,...);
```

Q: SASC puts its own version of Ctrl-C checking into my code. Can I prevent it from doing that?

A: You can disable the SAS CTRL-C/D checking with:

```
#ifdef LATTICE
int CXBRK(void) { return(0); }
int chkabort(void) { return(0); }
#endif
```

Then, when you are waiting on signals, you `OR` in `SIGBREAKF_CTRL_C` as defined in *<dos/dos.h>*.

```
ULONG signals, winsig, timersig;

winsig = 1L << mywin->UserPort->mp_SigBit;
timersig = 1L << mytimerport->mp_SigBit;

signals = Wait(winsig   |
               timersig |
               SIGBREAKF_CTRL_C);

if (signals & SIGBREAKF_CTRL_C)
{
    cleanup and exit
}
```

You can also check the signal at busy times with SetSignal():

```
if (SetSignal(0,0) & SIGBREAKF_CTRL_C)
        {
            cleanup and exit
        }
```

Q: How do I use the CreateNewProc() `NP_ExitCode` *and* `NP_ExitData` *tags?*

A: `NP_ExitCode` points to a cleanup function for the new process that will get executed when the process exits. Before the exit code (Process->pr_ExitCode) is called, the OS will put the program return code in d0, and the value from the `NP_ExitData` tag (pr_ExitData) in d1. Note that the cleanup code stashes pr_Flags before calling pr_ExitCode, so don't even think about modifying them. pr_ExitCode should return the return code passed to it, or a different return code if you wish.

pr_ExitCode/Data really belong to the creator of the process. An application can easily use the equivalent to exit()/_exit()/XCEXIT()/etc. to do process cleanup; this is to give the parent some hooks into a child that is going away. This can include processes created via System(). One example is to send an ''I'm dead'' message to the parent. This is quite useful for things like CreateNewProc() with `NP_Entry`, so the parent knows when it's safe to exit, or for a System() call so you know when a copy has finished, etc.

Q: Is it safe to assume that the process that runs its ExitCode exists until after pr_ExitCode is finished?

A: Yes. The ExitCode runs as part of its process.

Q: Are there any restrictions on what the ExitCode can do or access?

A: Anything that is legal in the main body of the process's code is legal in pr_ExitCode.

Q: Do I win a prize for asking too many questions in a row?

A: No.

Q: What do the following fatal errors under 2.04 signify: #80000004, #80000002, and #80000008.

A: These particular fatal errors signify what they have always signified.

The lefthand ''8'' means dead-end fatal crash.

The low numbers at the right are standard 68000 exception numbers. Ignoring the lefthand digit (here an 8), any fatal error which is between 0000000 and 00000FF is a standard processor exception. Books on the 680x0 CPUs should document the standard processor exceptions.

Anyway, here are the common ones:

```
##      Name                 Explanation
02  Bus Error            - Referencing non-
     existent memory: bad pointers.
03  Address Error        - Referencing odd
     addr. as word/long on 68000.
04  Illegal Instruction  - Executing garbage:
     bad libbase, trashed stack
05  Divide By Zero       - Obvious.
08  Priviledge Violation - Executing
     priviledged Instr. or garbage (see 04).
0A  Line 1010 emulator   - Executing a word
     beginning with 1010 (see 04).
0B  Line 1111 emulator   - Executing a word
     beginning with 1111 (see 04).
```

Note that more complex (higher number) fatal errors are Amiga-specific and are (for the most part) defined in *<exec/alerts.h>*. You may have to OR a couple of them together to build the full alert number if you don't find the complex number as-is in *<exec/alerts.h>*.

Q: Are DOS ticks different on PAL and NTSC systems?

A: No. Ticks are a 50th of a second for either PAL or NTSC. See the AmigaDOS manual for details.