Using SetFunction() in a Debugger

By Ewout Walraven

The Amiga OS consists of a set of libraries (and devices), which reside in ROM or on disk. These libraries provide a set of routines which are shared by the Amiga tasks (hence the name *shared* library).

The way in which an Amiga library is organized allows a programmer to change where the system looks for a library routine. Exec provides a function to do this: SetFunction(). The SetFunction() routine redirects a library function call to an application-supplied function (Although this article doesn't address it, SetFunction() can also be used on Exec devices). The *SetPatch* utility uses SetFunction(). *SetPatch* is a program which replaces some OS routines with improved ones, primarily to fix bugs in ROM libraries.

Normally, programs should not attempt to ''improve'' library functions. Because most programmers do not know exactly what system library functions do internally, OS patches can do more harm than good. However, a useful place to use SetFunction() is in a debugger. Using SetFunction(), a debugger can reroute library calls to a debugging function. The debugging function can inspect the arguments to a library function call before calling the original library routine (if everything is OK). Such a debugging function doesn't do any OS patching, it merely inspects.

SetFunction() is also useful for testing an application under conditions it does not encounter normally. For example, a debugging program can force a program's memory allocations to fail or prevent a program's window from opening. This allows a programmer to find bugs that only arise under special circumstances. Some programs that use SetFunction() for debugging purposes are *IO_Torture*, *Memoration* and *MungWall*. A real watchdog is *Wedge*, which, as its name implies, allows you to install a wedge for practically every function of a standard library and inform you about the register values passed to the function. These types of debugging tools helped debug release 2.0 of the OS and found bugs and 1.3 dependencies in commercial applications.

Although useful, SetFunction()ing library routines poses several problems. First of all, the wedge routine will have to be re-entrant, like all Exec library functions. Secondly, there is always a problem with removing the wedge. If another task has SetFunction()ed the same library routine as the debugger (a very real possibility), it is not normally possible to remove the first wedge, since the other task depends on the presence of your task's code. This would force your task to hang around, waiting for the other task(s) to remove their wedges. You also need to know when it is safe to unload your debugging code. Removing it while another task is executing it will quickly lead to a hopelessly crashed system.

For those of you who might be thinking about writing down the ROM addresses returned by SetFunction() and using them in some other programs: *Forget It.* The address returned by SetFunction() is only good on the current system at the current time. Blindly jumping into ROM *will* cause your programs to break.

Exec Library Structure

When a library is opened for the first time, a library node structure, a jump table, and a data area are created in RAM.

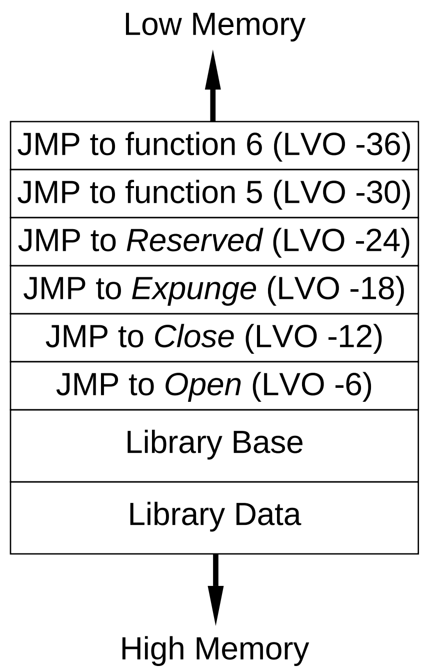

Fig. 1
An Exec Library Base in RAM

The library node structure address is the base address of the library. OpenLibrary() returns this base address. The library's jump table, which directly precedes the library node in RAM, consists of six byte long entries containing a jump instruction (JMP) to a corresponding library function. The jump table is initialized when Exec opens the library. Each function's entry in the jump table (also known as a vector) is always a constant (negative) offset from the library base. These fixed negative offsets are known as Library Vector Offsets (LVO). Note that the first four function vectors are reserved for use by Exec. They point to standard library functions for opening, closing, and expunging the library, plus there is space reserved for a fourth function vector. The base address of a library is determined dynamically when the library is loaded into RAM. See the Exec introduction chapter in the *ROM Kernel Manual: Libraries and Devices* for more information on libraries.

The SetFunction() routine replaces an LVO address with a new address which points to the wedge routine. SetFunction() returns the old vector address, which the wedge routine can use to call the original library function from within the wedge. Note that if another task SetFunction()s the same library function, SetFunction() returns the address of *your* debugging routine (to the second task) as the old vector. At that point your task can no longer exit since that would mean that that other task has an invalid pointer to your function and will most likely crash the system when it tries to use your function.

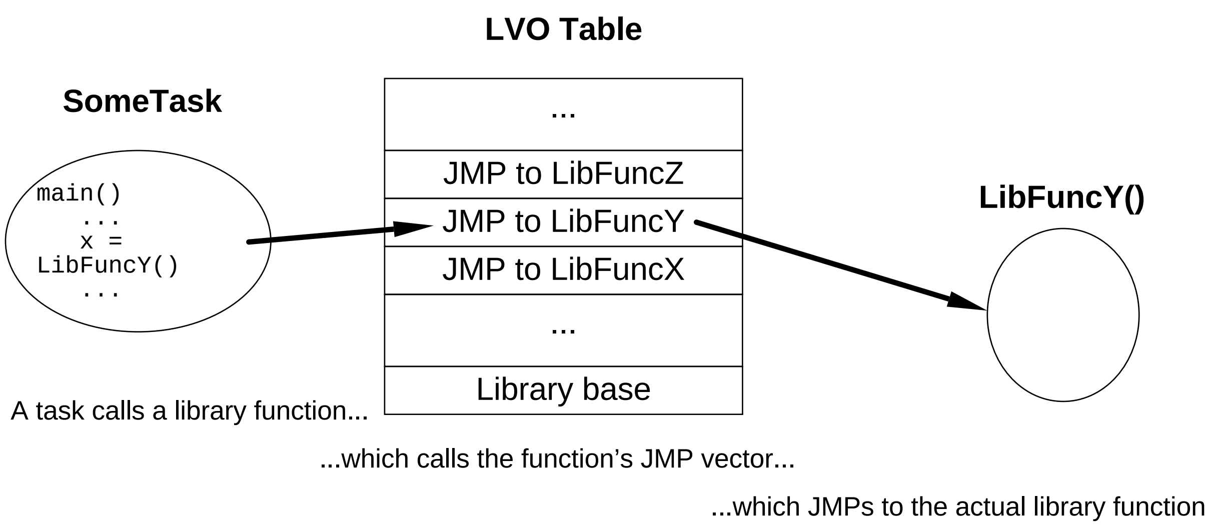

Fig. 2
Calling a Library Function

There is a way around this problem. Instead of SetFunction()ing a library function with the address of your wedge code, build your own jump table and use the addresses of its entries as arguments to SetFunction() calls. This method allows you to unload your code when you want to because if another task SetFunction()s your routine, that task will get a pointer to a jump table entry, not your routine.

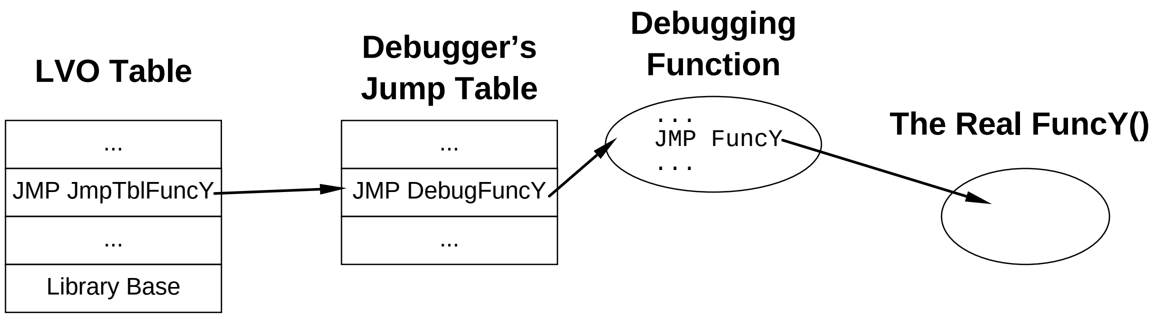

Fig. 3
Using SetFunction() with a Jump Table

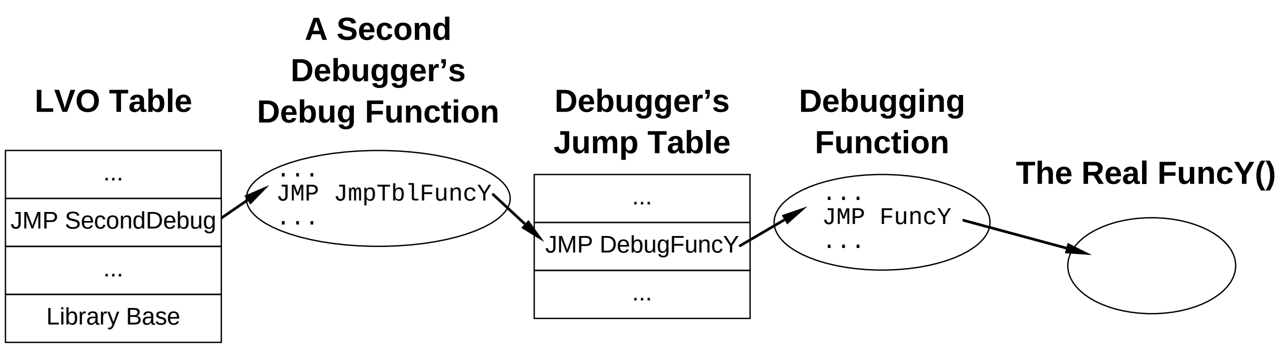

Fig. 4
Another Debugger SetFunction()s an Entry in the Jump Table

Now when you want to exit, all you need to do is replace the entries in your jump table which point to your functions with the original function vectors which were returned by SetFunction(). By not freeing the memory allocated for the jump table your task can exit any time, regardless of other tasks which SetFunction()ed the same library routines. The other task(s) will never know what happened.

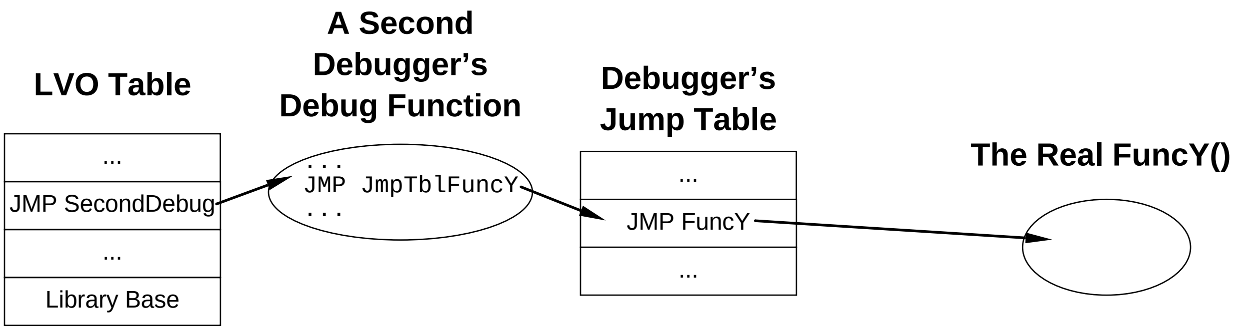

Fig. 5
Original Debugger Removes its Debugging Routine

The next time the debugger is executed, it looks for the jump table it left behind and replaces the entries in it with pointers to its own functions. Incidentally, this is an easy way to provide a mechanism to determine if your debugging program has already been installed.

Caveats

There are some things to keep in mind when using SetFunction(). The scheme described above can force a second task to hang around forever if it SetFunction()ed a routine before you, since your debugger will not normally release its handle on the second task's function. Whenever possible, install jump table based debuggers before any other SetFunction()ing program (but after *SetPatch* of course).

Some debuggers interpret the return address of the caller. When a debugger jumps (JMP) to (what it thinks is) the original function, there will be no problem. However, if a debugger performs a JSR to a second debugging function which interprets the return address, the second debugging function will receive the first debugging function's return address (the one that performed the JSR) rather than the return address of the application that called the library function in the first place. This can confuse the second debugger. Two good examples of this are *Scratcher* and *MungWall*, which both SetFunction() FreeMem(). *MungWall* looks at the return address of the caller. Since *Scratcher* calls the old FreeMem() function with a JSR instruction, it would mislead *MungWall* if run after it. Preferably, debuggers that interpret the return address should be started *after* other debuggers.

Although it is not common, some library functions call other library functions and depend on certain scratch registers to contain valid values. SetFunction()ing one of these functions is likely to change the values in these scratch registers, leading to problems. Because these dependencies are not always documented, you might innocently run into one. *Scratcher* is an excellent tool for finding such dependencies.

In the past, some system functions did not have a JMP vector in their entry in the LVO table. Instead, the actual function was in the jump table. SetFunction() will not work on such a function.

Any debugging routine should be careful not to call the function it has patched with SetFunction(), either directly or indirectly. Doing so will likely cause a stack overflow and crash the machine. This may seem a bit obvious until you consider how easy it is to indirectly call a system routine. Many system functions are not atomic. They have to call lower level system functions. If you call a higher level system functions in the debugging code and you have SetFunction()ed one of the routines the high level function uses, the machine will probably crash from a stack overflow.

Using SetFunction() on disk-based libraries and devices requires a little extra care. Unlike a ROM library, libraries (and devices) loaded into RAM can be expunged when memory gets low. To prevent the system from expunging a library (or device) you have SetFunction()ed, either keep the library (or device) opened, or use SetFunction() to patch the library's expunge function.

Remember that when you install a wedge, another program can call it almost instantly. Because of this, the wedge should be completely set up before you install it.

Note that *dos.library* is now a standard library and can be SetFunction()ed as of V36. Before V36 you would have to Forbid(), get the six original bytes of the entry in the function vector table, install the new vector, perform a SumLibrary() and then Permit().

If it is necessary to put debugging code into a Forbid() or Disable() state, keep it in that state for as little time as possible to limit system slowdown. Remember that you cannot Disable() for more than 250 microseconds. Be sure to read the Disable()/Enable() Autodocs before using them.

An Example Debugger

The usage of SetFunction() is shown by the example debugging program at the end of this article, *ISpy*. *ISpy* uses a semaphore to gain access to its jump table. This jump table contains pointers to the wedge routines. When executed, each wedge routine puts a shared lock on the semaphore to indicate that the code is being executed. To get an idea of who is calling the debugger entries, *ISpy* uses a little assembler stub to load a4 with the address of the stack of the caller and calls a C routine where the actual (simple) argument checking is done. When *ISpy* is signalled to exit, it tries to get an exclusive lock on the semaphore in a Forbid()en state. If this succeeds, it can safely assume its code is not being executed at the moment and can therefore place the original function vectors in the jump table and exit, leaving the semaphore behind. This semaphore is also used to check whether *ISpy* is already installed. If so, the new instance will exit immediately. Because of the use of shared semaphore locks, this program will only run with V37. By using a global counter (which is incremented each time a function is entered and decremented when it is exited) *ISpy* can be adapted to V33. Because of the way *ISpy* is set up, it is very easy to add argument checking front ends for functions, and have multiple versions of *ISpy* for different libraries.

Memoration and *Scratcher* by Bill Hawes. *IO_Torture* by Bryce Nesbitt. *Wedge* and *DevMon* by Carolyn Scheppner. *MungWall* by Ewout Walraven (inspired by *Memwall* by Randell Jesup and *MemMung* by Bryce Nesbitt).