TECHNOTE: Controlling Apps with Synthesized Events,
or jGNEFilter — the Untold Story

by Pete Gontier <gurgle@apple.com>
and Mark Cookson <mcookson@apple.com>
Apple Developer Technical Support (DTS)

Until now, jGNEFilter has been "under documented," with only vague mentions appearing in *Technote TB 11*. jGNEFilter is the name of a mechanism by which programs can obtain access to each `EventRecord` just before the event is sent to the caller of `GetNextEvent` or `WaitNextEvent`.

Using jGNEFilter, your programs can customize most event-driven interaction with the user, including but not limited to such things as monitoring keystrokes, and programmatically simulating some kinds of user activity. Also, without

being an application or driver, your program can arrange to be called periodically at a time when it's safe to call Memory Manager (and the high-level managers which depend on Memory Manager).

Developers who would like to make use of jGNEFilter — or developers who are already bravely making use of it even in the face of inadequate documentation — should read this Technote.

jGNEFilter Fundamentals

The interface to jGNEFilter is, unfortunately, rather primitive. The key to the whole thing is a single long word in low memory (at address 0x029A, to be precise). This long word, if non-zero (and it's almost always is non-zero; more on that later), is the address of a routine that `GetNextEvent` calls to filter events. It's that simple.

The process of installing a jGNEFilter routine amounts to saving off the old filter routine address and installing a new one. There is no arbitration for access to this memory location; programs must be very careful to access it according to the calling conventions, as explained in the next section of this Technote. Otherwise, the system may begin to misbehave in mysterious ways and other jGNEFilter routines may not get the access to events they need to function properly.

As is always the case with low memory, you should access jGNEFilter only through the LM accessor functions declared in the Universal Headers' <`LowMem.h`>, which in this case are `LMGetGNEFilter` and `LMSetGNEFilter`.

All jGNEFilter routines should call any previous routine. This policy alone is responsible for the formation of a "chain" of jGNEFilter routines. This is roughly the same idea as a trap patch. Calling the previous jGNEFilter is essential to the proper operation of the Mac and cannot be omitted. Exactly when in your routine to call the next routine in the chain is up to you.

jGNEFilter Calling Conventions

jGNEFilter calling conventions are inherently 68000-oriented and don't conform to the calling conventions of any high-level language.

For PowerPC filter routines, use `NewGetNextEventFilterProc` and `CallGetNextEventFilterProc`, both of which make use of a special-case routine descriptor that makes writing the routine's interface in a high-level language simple.

For 68K filter routines, it's probably not impossible to write a jGNEFilter routine entirely in a high-level language (assuming you're using a reasonably modern compiler), but it's probably more effort than it's worth. Instead, you'll probably want to use a few lines of assembly glue to call a routine written in a higher-level language.

On entry to your jGNEFilter glue, register A1 will contain the address of the event record to be filtered. Register D0 will contain a word which is the proposed return value for `GetNextEvent`. The word at offset 4 from register A7 (just above the return address) will *also* be the proposed return value for `GetNextEvent`. The difference is that D0 is an input value and the stack word is an output value. The stack word will be returned to the caller of `GetNextEvent`. Initially, the word in D0 and the word on the stack are (or should be, assuming there are no buggy jGNEFilter functions higher in the chain) the same.

It's important to note that the values of both registers A1 and D0 must be set before calling the next routine in the chain or returning. This isn't anything particularly special to jGNEFilter routines (as opposed to trap patches and other such things), but it warrants emphasis because it's an easy thing to forget.

By far, the trickiest part of calling the previous jGNEFilter routine is knowing when and how to set the value of the word on the stack. If your routine is going to *jump to* the next routine in the chain (using a 68K JMP instruction or its equivalent), you need to make sure the stack value is what you would have returned in case the next routine in the chain chooses not to change it. If your routine is going to *call* the next routine in the chain (using a 68K JSR instruction or its equivalent), you should push a word onto the stack before calling the next routine in the chain and pop the word after the routine returns. You can then use the return value from the next routine in the chain routine to help you decide what return value to put on the stack when your routine returns.

In any case, you should set register D0 to the value you want the next routine in the chain to use as input.

These lines, slightly modified from the "jGNE Helper" sample on the Developer CD Series Tool Chest Edition, perform the correct sequence of assembly instructions to allow a C function to do the brunt of the work.

```
MOVE.W    D0,-(A7)      // push pre-result for C
MOVE.L    A1,-(A7)      // push event record pointer for C
JSR       myGNE         // do the real work (in C)
MOVE.L    (A7)+,A1      // restore event record pointer
ADDQ.L    #2,A7         // pop pre-result
                        // the post-result (from C) is in D0
ASL.W     #8,D0         // bump C boolean to (Lisa) Pascal format
MOVE.W    D0,4(A7)      // stash result where caller expects it
```

jGNEFilter “Gotchas”

The following section discusses the two important edge cases that you need to take into consideration when installing a jGNEFilter.

The Low Memory Context Switching Gotcha

Most low memory global variables are swapped in and out of low memory on a process-by-process basis. (Their values live in the Process Manager’s storage while they’re swapped out.) This was done in the early days of MultiFinder to appease applications that assumed they owned the whole machine.

jGNEFilter is not one of the low memory globals which is swapped. Consequently, it’s possible for an application to install a filter routine and get access to events which are about to be passed to other applications. (Mostly, such filters get access to all relevant events destined for the foreground application but only null and update events destined for background applications, because these events are generally the only events which background applications receive.)

An application installing a jGNEFilter function does not pose a problem until it’s time to quit the application and/or uninstall the filter. Since the jGNEFilter routine address is just a long word in low memory, there’s no way to prevent a second application from reading it and installing a new filter routine address. This second application would expect to call what it perceives to be the next filter routine in the chain. When the time came for the first application to quit, it would restore the “next” routine address low memory, over-writing the filter routine address of the second application. Suddenly, the second application would be excluded from the filter chain and would stop functioning properly. The situation would get even worse if the second application were to call its “next” filter routine address, because that code would have disappeared when the first application quit.

The solution for applications that want to install a jGNEFilter is to install it indirectly via a “jump island” in a 6-byte pointer block in the system heap. Disassembled, the jump island looks like this:

```
JMP XXXXXXXX ; where XXXXXXXX is the address of your filter routine
```

You declare a struct for this:

```
#if PRAGMA_ALIGN_SUPPORTED
#   pragma options align=mac68k
#endif

typedef struct
{
```

```
    unsigned short     jmp;
    void               *addr;
}
tJumpIsland, *tJumpIslandP;

#if PRAGMA_ALIGN_SUPPORTED
#   pragma options align=reset
#endif
```

After calling `NewSysPtr` to allocate the block, you set `jmp` to 0x4EF9 and `addr` to the address of your filter routine (or `GetNextEventFilterUPP`). Remember to flush the instruction cache after performing this magic. (See Technote HW 06 for details on flushing the instruction cache.) When you want to uninstall your filter routine, simply set `addr` to the previous filter routine address. *Don't* dispose the pointer block and *don't* call `LMSetGNEFilter`, so that other programs continue to function. Do remember to flush the instruction cache again.

The Unexpectedly NIL Filter Routine Address Gotcha

Since the system uses jGNEFilter to do housekeeping such as servicing the Notification Manager queue, one might expect jGNEFilter to always be non-NIL. However, this is not the case. Some third-party programs have taken it upon themselves to set the jGNEFilter routine address to NIL temporarily for their own nefarious purposes. Always be ready to compensate for this. Compensating might be as simple as testing the address before calling it.

jGNEFilter Limitations

Compared to other mechanisms, jGNEFilters have remarkably few limitations. jGNEFilter routines can allocate or move memory (directly or indirectly), call the Toolbox, perform file I/O, launch applications, etc. The limitations are:

Not Every Event

Your jGNEFilter function will not receive every event returned to GetNextEvent and WaitNextEvent.

MacOS 8

The jGNEFilter mechanism will almost certainly be unavailable under Mac OS 8. You should abstract as much of your event-handling code as possible to minimize maintenance if and when MacOS 8 provides a similar mechanism.

Not a Process

Any system call that relies on the current process to establish some sort of unique identity is not going to work very well. The reason is that a jGNEFilter routine can be called while any process is current.

For example, the AppleEvent Manager uses the current process to identify the sender of an AppleEvent. A jGNEFilter routine can *send* AppleEvents, as long as the current process has its `modeHighLevelEventAware` bit set, but it can't receive them, and that includes queued reply events. It might be tempting to set up AppleEvent handler routines while a given process is current, but that's likely to cause you big compatibility problems in the long term, if not right away — just don't do it.

If you need a process, consider starting up a background-only application, either via LaunchApplication or by changing your 'INIT' file to an 'appe'. You can find more information on background-only applications in Technote PS 02.

Not an Event Loop

jGNEFilter is not a loop but a filter. Consequently, a jGNEFilter routine does not have the freedom to define the way in which it handles events. It handles them when the system dictates. If your routine jumps to the next routine in the chain,

Your routine might make Event Manager calls which result in another call to jGNEFilter; these calls include `GetNextEvent` and `WaitNextEvent` (because `WaitNextEvent` calls `GetNextEvent`).

The bottom line is your routine may need to set and/or test a re-entrancy flag to avoid infinite recursion.

International Keystrokes

"Fake" `keyDown` events are *not* sent through the jGNEFilter chain. This is a known bug. It should manifest only when multi-byte script packages, such as the Japanese Language Kit, are installed.

If your program needs access to keystrokes and will be sold into a market where WorldScript is in heavy use, you may be better off patching `WaitNextEvent` than writing a jGNEFilter routine. (Yes, it's shocking to see DTS speaking in a favorable light about a trap patch, and it pains us to write it, but it's the plain truth.) Unfortunately, this kind of patch is difficult to write.

Text Services Manager Bugs

In the presence of a Text Services Manager input-method window (usually

called, simply, a TSM window), some `mouseDown` events may not be sent to the jGNEFilter and in fact may appear to "pass through" a TSM window into whatever is behind it. Unfortunately, there is no official workaround for this known bug.

Summary

The jGNEFilter is a powerful mechanism that adds new functionality to your code, enabling you to watch the system as whole. If you're trying to monitor or modify the user's interaction with the system, jGNEFilter is the place to start. If you're simply trying to achieve a more flexible environment for your periodic tasks — i.e., allowing them to allocate memory or do file I/O, jGNEFilter may also be for you. There are some limitations, as explained in this Technote, but the unique advantages of jGNEFilter outweigh its disadvantages.

Further Reference

- *Technote TB 11*
- *Technote HW 06*
- *Technote PS 02*

For sample code, see:

```
Dev.CD Aug 96 TC
  Tool Chest
    OS Utilities
      jGNE Helper
```

Acknowledgments

Thanks to Matt Slot for sanity-checking and other rewardingly useful suggestions.