AETracker Programmatic Interface

Written by C.K. Haun
RavenWare Software

NOTE: This paper describes advanced features of AETracker. These features are *not* necessary for regular AppleEvent tracking.

> You only need to read this information if you would like to modify your application to talk directly to AETracker.
> You do *not* need to read this for most uses of AETracker

This document describes the programmatic interface to AETracker, introduced in version 3.0.
Components in package;

AETracker 3.0	AppleEvent tracking INIT/Control Panel
AETracker.h	C interface definitions for AETracker
AETracker Monitor	Sample program demonstrating programmatic interface
AETracker Monitor.c	MPW C sample code
AETracker Monitor.r	Rez code for sample
AETracker Monitor.make	MPW make file for sample

AETracker is a powerful tool for tracking and debugging AppleEvent activity. However, in earlier versions all control of the tracking mechanism came from the keyboard or the Control Panel, and output options were limited.

For version 3.0, a programmatic interface has been added to the AETracker INIT to address these weaknesses.
The interface allows you to toggle AETracker on and off, direct AETracker to use a specific file for output, re-direct parse text to your routine, and redirect AppleEvent calls to your routine.
Using these calls, you can code a debug version of your program that turns AETracker on and off at the particular section of your code that you know a bug is occurring at, allowing you to get a record of *just* the section of AppleEvent activity you want, instead of all the traffic AETracker normally reports.
You can also specify the output file AETracker should use, this allows you to use a name that is more meaningful to you.

AETracker also allows you to re-vector the text output from AETracker. This allows you to make a 'live' AppleEvent monitoring window or application, or you can capture the output in a file of your own format or type. This redirection capability is not exclusive, you can redirect AETracker output to yourself and still have AETracker record to it's standard file.

And finally, AETracker will also vector AppleEvent calls to your routine. This enables you to take advantage of AETracker's _Pack8 patch without having to write one yourself. You can break on specific AEM routines, record parameters, and potentially modify parameters. Again, this is not necessarily an exclusive vector, you can have AETracker do it's normal processing after you inspect the calls.

WARNING: Do not use the re-vectoring routines unless you know what you are doing! When you use the revectoring routines you are now part of a system level patch, potentially operating in someone else's A5 world and process time. Vectoring the parsing routines allow you to change parameters to actual AEM calls. There is a great potential for mistakes unless you are careful..

How does it work?
When AETracker installs itself, it now installs a Gestalt selector to allow others to contact it. When you make the gestalt call
`Gestalt(kAETGestaltSelector,&returnValue);`

AETracker returns a pointer to it's external interface routine in the Gestalt call return value.

From then on, you call that routine to ask AETracker for services.

There are currently 7 AET external routines

```
enum {
kAETToggle =0,
kAETStatus,
kAETInterceptOutput,
kAETReconnectOutput,
kAETInterceptParse,
kAETReconnectParse,
kAETSetPrefs,
kAETEmergencyReset
};
```

This list will probably grow in future releases of AET, stay tuned. Also, your input is vital for getting new features, if you have needs or suggestions please pass them along. Each call will be discussed in detail.

Calling AETracker
The interface to AETracker is defined as

```
typedef pascal OSErr (*AETrackExternProcPtr)(AETrackExtParamPtr theParm,short selector);
```

Where `AETrackExtParamPtr` is a pointer to the structure defined as

```
typedef struct AETrackExtParam{
short fileRefNum; /* returned file refNum
FSSpec outFile; /* input/output file specifier
union{
externOutputProcPtr outputPtr; /* re-vectoring routines */
externParseOutputProcPtr outputParsePtr;
}procs;
union{
AETrackerStatusBlockPtr statusData;
Ptr externDataPtr;
}dataPtr; /* data needed for some calls */
long refCon; /* refCon for your use (passed back to you by AET) */
unsigned short flags; /* control flags for some calls */
}AETrackExtParam;
typedef  AETrackExtParam *AETrackExtParamPtr,**AETrackExtParamHdl;
```

So a sample call to AETracker would look like this, using an AET toggle with no file specifier as an example;

```
AETrackExternProcPtr AETCall;
AETrackExtParam myAETParamBlock;
OSErr myErr = noErr;
Gestalt(kAETGestaltSelector,(long *)&AETCall);  /* get pointer to AET interface routine
*/
myErr = AETCall(&myAETParamBlock,kAETToggle);  /* call it */
```

Note: The AET interface routine does not move, AET is not dynamically positioned. You do not need to call the Gestalt routine before every call, once in your application is sufficient. All further samples in this paper assume that you have already made the Gestalt call.

Toggling AETracker

AETracker has traditionally been turned on or off with a special key combination. This had some obvious drawbacks

- You usually had to track more information than you really wanted.
- AETracker was dependent on one GetNextEvent call being executed before tracking actually started.

Toggling AET programattically solves both of these problems. By adding toggle calls right before and after the code you are having difficulties with, you get to track just what you need.

Also, when you toggle programatically, you can specify a file to open instead of the default tracking file.

This call is a *toggle*. If AETracker is off, this turns it on. If AETracker is on, this turns it off. It is your responsibility to keep track of the toggle state.

To simply turn AETracker on without specifying a file;

```
myErr = AETCall(nil,kAETToggle);  /* call it */
```

To specify that AETracker use a different file from the default, you pass an FSSpec for the target file, and set the appropriate flag, as follows;

```
StandardPutFile("\p","\pmytrack",&myReply);  /* ask for a file name */
if(myReply.sfGood){                          /* if a different file requested */
myAETParamBlock.outFile = myReply.sfFile;    /* move the FSSpec into parameter block
*/
myAETParamBlock.flags = kAETUsePassedFileFSSpec;} /* set flag */
myErr = AETCall(&myAETParamBlock,kAETToggle);       /* call AETracker */
```

If a parameter block is passed to AETracker to start tracking, AETracker will fill in the file reference number and an FSSpec describing the file it has opened.

Note: When AETracker is toggled off, it clears *any* settings you may have made. It clears

vectoring and goes back to the standard target file. AET does this to protect itself, the only thing you need to remember is that you must reset anything you have changed before turning on AETracker.

Checking AET Status

You can check the status of AETracker at any time. This call returns AET's activity state, any file parameters, and state of AET interception.
There is an additional parameter block used for this call, passed in the `statusData` parameter of the standard parameter block. It is defined as;

```
typedef struct AETrackerStatusBlock {
short AETActive;  /* AET currently tracking yes/no */
short fileInterceptActive;  /* file output being intercepted yes/no */
short fileExclusiveIntercept; /* file output being vectored only to external routine*/
externOutputProcPtr fileInterceptRoutine;  /* routine file output being sent to */
short parseInterceptActive;   /* parse being intercepted */
short parseExclusiveIntercept;  /* exclusively? */
externParseOutputProcPtr parseInterceptRoutine;  /* routine location */
}AETrackerStatusBlock;
typedef AETrackerStatusBlock,*AETrackerStatusBlockPtr,**AETrackerStatusBlockHdl;
```

Note: The state parameters are returned in `short`, 16 bit values instead of traditional one-byte booleans. The effect is the same, all bits on means true, all bit off means false. I wrote AET in assembly, and I ***hate*** single byte access on the 680x0 so I won't use them. Bear with my idiocyncracies, it works just fine this way.

```
AETrackerStatusBlock myAETStatus;
AETrackExtParam  myAETParamBlock;
myAETParamBlock.dataPtrs.statusData = &myAETStatus;
myErr = AETCall(&myAETParamBlock,kAETStatus);
```

> **Caution:** It's a good idea to make this call before making any other AET calls. Someone else may already be intercepting the AET, or the user may have turned AET on from the keyboard.

Replacing File Output Vector

AETracker sends all the text output it produces to one place, usually a file called 'AETrack-{tickcount}'. This may not always be suitable for you, you may want to redirect the text output to another file or a window.
AETracker 3.0 allows you the ability to intercept the text stream it produces.
When you intercept the stream, you also have the choice of whether to turn off AETracker's regular output, or to have your intercept in place and also have AETracker do the standard output. This allows you to create a 'real-time' picture of Apple Event Manager traffic in a window, while AETracker continues to log a permanent record it it's standard file.
You supply a procedure pointer to AETracker, and whenever AETracker prepares to write text to it's text file, it will first call your routine with the text information.
A call to re-vector the standard AET output looks like;

```
myAETParamBlock.procs.outputPtr = SampleIntercept;
/* pass my A5 so it will come back to me later */
myAETParamBlock.refCon = SetCurrentA5();
myErr = AETCall(&myAETParamBlock,kAETInterceptOutput);
```

Where the procedure pointer you supply as `outputPtr` is defined as;

```
typedef pascal OSErr (*externOutputProcPtr)(ParmBlkPtr paramBlock,long refCon);
```

The refCon field you pass in this call is very important. When AETracker calls your routine, you will *not* know what application is currently executing and making the AppleEvent call. In many, or all, cases the current value of the A5 register will reflect the A5 world of the application making the AEM call, *not* your application! This means that, unless you supply a reference to your own A5, you cannot access any of your application globals, call functions in other segments, operate on any of your windows, and many other normal things. To assist you in doing what you want to, AETracker allows you to supply a refCon when you install a intercept routine. This can be a simple numeric value (as in this case, the actual value of your A5), or a pointer or handle to a more complicated structure. Every time AETracker calls your procedure it will pass this refCon value to you for your use.

> **Caution:** Again, at a *minimum* you should pass your A5 value as a refCon. You can cause serious damage to other applications if you do not set up your proper application environment during an intercept. Please see the supplied sample code for an example of how to do this.

The parameter block passed to your routine is identical to the parameter block used by the file system routine `PBWrite`, in fact, it is a duplicate of the parameter block the AETracker will be using to write the information out to it's own file. However, you are free to use it in any way you wish, in most cases all you will care about are the `ioBuffer` and `ioReqCount` fields. But again, this is a *duplicate* of the parameter block AETracker will be using, so you are free to make any changes to this parameter block you wish.

You have one other option for file intercepting, you can request an *exclusive* intercept of AETracker output.
Setting the `kAETDisconnectRegularOutput` flag in the passed parameter block
```
myAETParamBlock.flags = kAETDisconnectRegularOutput;
```
tells AETracker that you will be handing all the text output. AETracker will not write anything to it's normal output file.

Reconnecting File Output Vector

At any time, you are free to stop intercepting file output. You can do this without any regard for the toggled state of AETracker, anytime you wish to stop intercepting text output.

> **Note:** This does not turn AETracker off. AETracker will still send it's normal text to it's output file.

It's a simple, no parameters call;
```
AETCall(nil,kAETReconnectOutput);
```

Intercepting AppleEvent Routine Parsing Vector

You can intercept the actual AppleEvent calls in roughly the same way you intercept file output.
By installing a parse vector, AETracker will call your application on every AppleEvent call, telling you which call is being made and giving you access to the parameters for that call. This gives you the ability to do more data analysis than AETracker does, or to replace AETracker's parsing routines.

Warning: This call gives you direct access to the parameters to any AppleEvent Manager call. This means that you could (accidentally or purposefully) modify AppleEvent calls in progress. This can, obviously, be very dangerous, please do *not* use this capability without fully understanding what you are doing.

The call to do this looks like

```
myAETParamBlock.procs.outputParsePtr = SampleParseIntercept;
/* pass my A5 so it will come back to me later */
myAETParamBlock.refCon = SetCurrentA5();
myErr = AETCall(&myAETParamBlock,kAETInterceptParse);
```

Where the parsing intercept routine you pass is defined as

```
typedef pascal OSErr (*externParseOutputProcPtr)(Ptr AEParmPtr,long AESelector,long
refCon);
```

`AESelector` is a long word containing the selector number of the routine currently being called ($0D17 for AESend, for example). The AppleEvent manager selectors are all short integers, but AETracker passes you a long. This is done because of the AETracker tail-patch. When your routine is called from the head-patch, or entry, point of the AppleEvent routine the selector code will be the actual selector passed to the AppleEvent manager. However, when the routine returns and the tail-patch side of the parsing occurs, the selector will contain the constant `kAETGestaltSelector`, this is to let you know that a routine is completing. It is your responsibility to keep track of which routine was last entered.

The `refCon` field is the same refCon you pass in at the `kAETInterceptParse` call, that long will be passed to your parse routine every time it is called.

Caution: Again, at a *minimum* you should pass your A5 value as a refCon. You can cause serious damage to other applications if you do not set up your proper application environment during an intercept. Please see the supplied sample code for an example of how to do this.

`AEParmPtr` The third value passed to your parse routine is the location of the parameters passed to the AppleEvent call. This is a pointer to the 'bottom' of the stack, so you can access any parameters passed to the routine as offsets from this value. For example, if you want to see the class of AppleEvent being created in a call to AECreateAppleEvent,

```
pascal OSErr AECreateAppleEvent(AEEventClass theAEEventClass,
                                AEEventID theAEEventID,
                                const AEAddressDesc *target,
                                short returnID,
                                long transactionID,
                                AppleEvent *result)
= {0x303C,0x0B14,0xA816};
```

your parse routine would look something like

```
OSType createdType;
if(AESelector == 0x0B14){ /* is AECreateAppleEvent being called? */
createdType = *((OSType *) (AEParmPtr + 18));
}
```

since the class of the AppleEvent is passed 18 bytes above the bottom of the stack. When the tail-patch is passed to you (indicated by `kAETGestaltSelector` in the selector long passed to you) this pointer does not (of course) point to the passed parameters, since they have been cleaned up. Instead, it is now a pointer to a short integer, the error code passed back by the AppleEvent routine.

As with the file intercept routine, you can exclusively intercept the parse.

Setting the `kAETDisconnectRegularParse` flag in the passed parameter block
`myAETParamBlock.flags = kAETDisconnectRegularParse;`
tells AETracker that you will be handing all the parsing.

Reconnecting AppleEvent Routine Parsing Vector

At any time, you are free to stop intercepting the parse. You can do this without any regard for the toggled state of AETracker, anytime you wish to stop .

Note: This does not turn AETracker off. AETracker will do it's normal parsing.

It's a simple, no parameters call;

```
AETCall(nil,kAETReconnectParse);
```

Overriding AET Preferences

```
/* This is the preferences structure that AETracker uses. */
/* Can be modified and passed to kAETSetPrefs */
struct PrefsStruct {
long keyMask1;
long keyMask2;
short collectFrom;
short collectLevel;
short modKeys[4];
long actionKey;
short infoLevel;
short disabled;
short findMBug;
};
```

`keyMask1` is the mask AETracker uses to check for the modifier keys needed to toggle AET.

`keyMask2` Hmmm. Looking at the AETracker source code, this doesn't look like it does anything at all.

`collectFrom` determines if the starting PSN should be the only app to check. If ZERO, then all calls will be tracked. If <> ZERO, then tracking will only be performed for the PSN that was frontmost when AETracker was toggled.

`collectLevel` is the selector range value, set by the everything/limited/pick 'em radio buttons in the CDEV. Values here go from 0 (everything) to 2 (pick 'em).

`modKeys` is merely a small array for convience in the CDEV, it has no effect on the INIT.

`actionKey` is the long that contains the key code that AETracker is looking for as a toggle. AETracker looks at the key *code*, not the ASCII code, so the value at $0000XX00 is signifigent.

Note: Only that byte should be set. bits set in other parts of the long word will cause AETracker to ignore the keystroke.

`infoLevel` is the setting changed by the minimum/more/maximum radio buttons in the CDEV. The value is from 0 to 2, with 0 being minimum and 2 being maximum

`disabled` Tells AETracker's INIT code not to install AETracker. This is not useful in this call, since AETracker is already installed

`findMBug` is a boolean that tells AETracker to try and find the label for the routine that is currently calling the AppleEvent manager.

Note: The effects of this call are not permanent. The preferences resource in AETracker will not be changed by this call, this is only for runtime effect.

To affect permanent changes you must use the Control Panel.

Emergency Resetting

There comes a time in every application when things Go Wrong. If you are programtically controlling AETracker and things get too weird for you, you can make the call

```
myErr = AETCall(nil,kAETEmergencyReset);
```

and AETracker will reset all it's operations. Tracking will be turned off, all the intercept routines will be cleared, and the preference changes you have made will be cleared.

AETracker Errors

There are three errors that may be passed back to you when you call the AETracker external entry point

```
kAETBadSelectorErr =8080,
kAETBadParamBlockErr,      /* something weird in the PB you passed (missing or
incomplete) */
kAETAlreadyInterceptedErr /* someone has already intercepted AETracker. */
```

`kAETBadSelectorErr` is returned if the selector you passed is not in the range AETracker supports. No action has taken place.

`kAETBadParamBlockErr` is returned if the call needed a parameter block and you forgot to pass one, or if there was a missing field in the parameter block you did pass. No action has taken place.

`kAETAlreadyInterceptedErr` is returned if you try and set an intercept routine when there is already an intercept in place. Use the Status call to check for this, and if you want to override an intercept routine another application may have installed you will need to make the appropritate reset call. No action has taken place.

Errors you return

Both the file intercept and parse intercept routines you supply require you to return an error code. Currently, AETracker does not have a sophisticated error interface. If you pass anything back other than `noErr` (0), AETracker will consider that to be a critical emergency, and will call Emergency Reset itself, turning everything off.