Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#94: Packing It In (and Out)

Written by: C.K. Haun <TR>
September 1990

This Technical Note discusses a potential problem with the Miscellaneous Tools routine `UnPackBytes`.

`PackBytes` and `UnPackBytes` are handy data compression and expansion routines built into the Apple IIGS System Software. Using them can dramatically reduce the amount of space your application uses on disk or in memory, but you need to understand how these calls work to avoid problems in your applications.

Buffer Size, Buffer Size, Buff, Buff, Buffer Size

There are some situations where the Miscellaneous Tools call `UnPackBytes` does not function as expected and can cause your application to loop infinitely while you're waiting for an unpacking process to finish.

The following packed data and code (in APW assembly) demonstrates the problem. It shows a small routine that unpacks data in two steps, simulating the situation in many applications where an arbitrary amount of data is unpacked in a variable amount of unpacking actions, depending on the results of the last unpack pass.

```
UnPackBuffer       ds    160                   ; area to unpack the data to
UnPackBufferPtr    dc    i4'UnPackBuffer'      ; pointer to unpacking buffer
UnPackBufferSize   ds    2
temp               ds    2

PackedData         dc h'FFFFFFFF'
```

```
EndPackData       anop
PackLength        dc i2'EndPackData-PackedData' ; how many bytes of packed data

* In packbytes format $FFFF means '64 repeats of the next byte ($FF) taken as 4 bytes' as
* described on page 14-39 of Toolbox Reference, so
* this data should unpack into 512 $FF bytes
* The following code loops infinitely

                 lda   #160                    ; Unpack buffer size
                 sta   UnPackBufferSize
UnPackLoop        pea   0                       ; return space
                 pushlong #PackedData          ; pointer to packed data
                 pea   2                       ; size of the packed data, unpack two bytes
                                               ; at a time
                 pushlong #UnPackBufferPtr     ; pointer to pointer to unpacking buffer
                 pushlong #UnPackBufferSize    ; pointer to word with the size of the
                                               ;  unpacking buffer
                 _UnPackBytes
                 pla                           ; returns 0 bytes unpacked
                 sta   temp
                 lda   PackLength
                 sec
                 sbc   temp                    ; subtracting it from our known
                 sta   PackLength              ; length of packed data
                 bne   UnPackLoop              ; this is always be non-zero
```

The problem is in the data and the buffer size. `UnPackBytes` is being told to unpack two bytes ($FFFF), which generate 256 bytes of unpacked data, into a 160-byte buffer. Instead of reporting an error with this condition, `UnPackBytes` instead just does nothing and passes back zero as the returned number of bytes unpacked. If you are relying on the unpacked byte count returned to control your unpacking loop, then you may encounter this problem.

`UnPackBytes` can be used to unpack in multiple steps, of course, but it cannot unpack a partial record. It cannot unpack 160 bytes of the 256 bytes specified in this record because `UnPackBytes` does not maintain any state information, so it must unpack full records or do nothing. If the buffer had been 256 bytes, this call would have succeeded.

The Fix

Fortunately, it's easy to avoid this situation if you know that it can exist. Simply, always supply `UnPackBytes` with a buffer that is big enough for it to unpack at least two bytes (a flag or count byte and a data byte). The largest value of a flag or count word possible is $FF, 64 repeats of the next byte taken as four bytes, which generates 256 unpacked bytes. So always give `UnPackBytes` a 256-byte long output buffer and you should never encounter this problem.

Check Your Current Applications

Please check your current applications to see if you could encounter this problem. One of the most likely places for this error to occur is in applications that process Apple Preferred (file type $C0, auxiliary type $0002) pictures. While most pictures currently available are screen-width or less (160 bytes or less per scan line), the Apple Preferred format and QuickDraw II both support pictures that are wider than the current Apple IIGS screen. If someone has created a picture with a `PixelsPerScanLine` value of 1,280 with a `ModeWord` of $0080, it would generate a scan line that was 320 bytes long. If a scan line in this hypothetical picture were all white, for example, the first two bytes of the packed scan line would be $FFFF, and applications that assume a standard maximum 160 bytes per scan line would not handle this correctly.

But That's Not All…

`UnPackBytes` has some other buffering problems of which you need to be aware. The size and location of the input buffer (the buffer containing your packed data) can also cause problems.

Note: These problems only occur if you are doing multipass unpacks. If you always unpack a packed data range in one pass (with one call to `UnPackBytes` for the whole data set) then you are not affected by these problems, and the restrictions described herein do not apply.

Multipass Restrictions

When performing a multipass unpack (as described on pp. 14-43..44 of the *Apple IIGS Toolbox Reference*, Volume 1) the packed data needs to follow two rules.

Rule 1: Your packed data buffer cannot cross a bank boundary.
Rule 2: Your packed data buffer needs to be at least 65 bytes longer than the actual size of the data.

These rules are required by a bug in `UnPackBytes`. When `UnPackBytes` begins to unpack a record, it checks the record data to see if there are enough bytes in the current source buffer to unpack the number of bytes requested in the record header (described on pg. 14-39 of the *Apple IIGS Toolbox Reference*, Volume 1). If there are not enough bytes left for the current record (i.e., the header says to process 63 bytes, and there are only 30 left in the buffer), `UnPackBytes` returns to the caller. The caller then adjusts the source buffer for the next pass based on the amount of actual bytes unpacked, so the bytes left over from the last pass get processed the next time.

The problem occurs when the partial record is close to the end of a bank. When `UnPackBytes` checks to see if there is enough data left in the buffer, the check is flawed when the real end of the buffer is near the end of a bank, and a complete copy of the partial record would extend into the next bank. `UnPackBytes` erroneously thinks that the record is complete, and happily unpacks the remaining actual packed data, plus random information from the next bank. It continues to unpack nonsense data until it fills the unpacking buffer and the number of bytes unpacked returned by the `UnPackBytes` call is greater than the `bufferSize` parameter passed as input.

To prevent this bug from occurring, you need to make sure that the buffer for the packed data is at least one record length away from the end of a memory bank. Since the largest packed data record is one flag byte and 64 data bytes, adding 65 bytes to the end of your buffer does the trick. This ensures that your packed data is 65 bytes away from the end.

Following is an example of a safe way to prepare your packed data buffer for multipass unpacking, in APW assembly:

```
* Some data space
myCallBlock  dc     i2'2'                ; two parameters
fileRefNum   ds     2                    ; file reference number
EOFreturned  ds     4                    ; file length returned by this call
myIDNumber   ds     2                    ; your application memory manager ID number
* assume that a packed data file is open, and it's a plain packed screen image, not over 32K
            jsl    $E100A8              ; ask GS/OS for the length of the data
            dc     i2'$2019'            ; Get_EOF call
            dc     i4'myCallBlock'
```

```
* Now we need a handle to read it into
            pha
            pha                         ; return space
            pea    0                    ; size, high word
            lda    EOFreturned          ; the actual size of the packed data
            sta    actualPackDataSize
            clc
            adc    #65                  ; ask for a handle 65 bytes longer than the data
            pha
            lda    myIDnumber           ; Memory Manager ID for your application
            pha
            pea    $8010                ; attrLocked and attrNoCross
            pea    0
            pea    0                    ; anywhere
            _NewHandle                  ; get the handle
```

Now you have a handle 65 bytes longer than your data that does not cross a bank boundary. You are ready to read in the data and perform a multipass unpack.

PackBytes Buffers Count Too

`PackBytes` can also cause you problems if you do not plan for the worst-case situation. Unlike the other toolbox compression routine `ACECompress`, `PackBytes` is **not** guaranteed to shrink the source data. In fact, your data size may actually grow after a `PackBytes` call.

If you pass a data stream of 64 bytes, all with different values, to `PackBytes`, `PackBytes` puts **65** bytes in your output buffer—the 64 original data bytes and the flag byte of $3F, indicating “64 bytes follow, all different.” Unless you preprocess or analyze your data before packing to avoid this situation, make sure your output buffer is large enough to hold the worst case situation, one additional byte generated for every 64 bytes passed to `PackBytes` for compression.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1-3
- File Type Note for File Type $C0, Auxiliary Type $0002, Apple Preferred Format