Apple II Technical Notes

Developer Technical Support

Apple IIGS #2: Transforming I/O Subroutines for Use in "Native" Mode

Revised by: Pete McDonald
November 1988
Written by: Pete McDonald
October 1986

This Technical Note outlines a number of techniques useful when transforming Apple II I/O subroutines for use in the "native" Apple IIGS environment.

The Apple IIGS execution environment represents quite a departure from the environment to which the average Apple II developer is accustomed. This fact results in a number of unique problems when one attempts to convert existing Apple II applications for use in the "native" Apple IIGS environment. (Note: If you intend to let your application remain an eight-bit "classic" Apple II application, then you can ignore the information this Technical Note presents.)

I/O subroutines which depend upon critically timed code present some of the biggest conversion problems due to two major issues. In the native IIgs environment, you cannot guarantee that there will be memory available in a given bank, and I/O locations are not available in every bank.

There are a number of possible solutions to this problem. Which ones you

should use depend upon what the program in question is doing. This Note attempts to describe some of the problem situations and possible solutions.

Examine the 6502 code segment below. It serves no useful purpose, other than to illustrate a simple manifestation of the problem. Assume `IoLoc` is a location in the $C000 – $CFFF range of memory.

```
Loop        LDA         IoLoc
            DEY
            BPL         Loop
```

Because the $C000 – $CFFF range of memory in bank 2 or higher contains RAM instead of I/O circuitry unless hardware shadowing is enabled, if you place the fragment above in one of these banks, it will have no effect on the I/O device you intend it to control.

There are two possible solutions in this case. Either change the instruction `LDA IoLoc` so it uses long addressing, thereby forcing the CPU to reference the the proper bank. (Note: The problem with this is the long version of `LDA` requires an extra CPU cycle to execute. If the code segment is timing critical, then this method is likely to be unacceptable.) Alternately, in the timing-critical case, we could set the data bank register before entering the loop which would mean the `LDA IoLoc` would take the same number of cycles as it did previously, thus leaving the timing loop unchanged.

These solutions seem pretty easy; therefore, you know there is a catch. The catch, unfortunately, is that most code is not isolated as in the example. Specifically, code commonly tries to load from or store to some location in memory other than the I/O location at the same time it is trying to access the I/O location.

Take, for example, the following fragment:

```
Loop        LDA         Data,y
            STA         IoLoc
            DEY
            BPL         Loop
```

In this example, we assume that the label `Data` refers to some kind of table which normally resides in the same bank as the program. Now if you set the data bank register to access I/O locations, then the reference to `Data` will also reference the same bank as the I/O; this solution is likely not acceptable. One thing you can do is move the data table to the direct page (zero page for 6502 programmers), but now the `LDA Data,y` instruction will take one less cycle to execute. There is a solution, although it is a little complicated. If we set the direct page register to a non page-aligned location, then we effectively apply a one-cycle penalty to all direct page references and solve our problem.

Of course, nothing is ever as simple as it seems. What happens to references to other direct page locations that expect to operate without the one-cycle penalty? To properly address this question, I would need much more space than I have here, so in lieu of further examples, I offer some general information. (As an aside, I used these techniques to transform the old "Apple II Disk II formatter module" for use in any bank of memory in the native IIGS environment. I accomplished this using, almost exclusively, editor find and replace commands, and I finished in hours instead of the days which would have been required to completely rewrite the program.)

In addition to the techniques already covered, there are a few other things which may be necessary to complete a transformation (they were necessary in the case of the formatter module).

As I already mentioned, one problem is what to do in the case where a program references I/O, local program-bank data, and the zero-page. In this case, significant rewrites could be required, but not necessarily.

In the case of the disk formatter, it turned out that some modules used both normal zero-page addressing and normal 16-bit absolute indexed addressing. Since the transformation process dictates that we change 16-bit absolute addressing to direct-page addressing with a non page-aligned direct page, there could have been a problem had both uses of the direct page been timing critical. Fortunately, by treating each module of the program separately, when I needed both types of addressing, only one was critical. The solution was to set the direct page to a non page-aligned value in some modules and to a page-aligned value in others. There are some minor logistical issues when a direct page's base address can be at either $xxx0 or $xxx1, the biggest of which is keeping track of which is in effect at a given point and knowing to reference the label as label, label+1, or label-1, depending upon the particular case.

With the formatter transformation, there was one other major issue: there are not direct-page versions of all the 16-bit absolute addressing modes (i.e., one cannot convert 16bitaddress,x to 8bitaddress,x). In the case of the formatter, I was able to solve this by reversing all the register use (i.e., all `LDY` instructions became `LDX` instructions, all `STY` instructions became `STX` instructions, etc.).

There are still a number of other ways in which one can approach these issues; one that comes to mind would be using some form of the new stack-relative addressing modes to yield yet another range of semi-independently accessible addresses.

The real point of this Technical Note is that with a little thought and effort, one can successfully convert a large subset of likely configurations for use in the native IIGS environment without major rewrites. The bottom line is to be creative!

Further Reference

- *Programming the 65816 Including the 6502, 65C02, and 65802* (Eyes/Lichty)
- *Apple IIGS Firmware Reference*