Aim User Input And Output

Joel Swank
Rockaway, OR

AIM 65 is Rockwell's entry into the SBC (Single Board Computer) market. It provides many advanced features not usually found in inexpensive systems. Among these is the capability for device independent I/O. Along with this is the capability for the user to add his own I/O device or devices. The AIM user manual devotes three pages to this feature. The information here is accurate but insufficient. Since I have recently interfaced a disk system to my AIM via the user I/O function, I have had the opportunity to investigate this feature thoroughly.

Whenever the AIM firmware receives a command requiring input or output, it calls a routine to determine which device to use. Two routines are used for this purpose: WHEREI for input and WHEREO for output. WHEREI and WHEREO prompt the user for the device to be used. The user enters a one character device code: 'P' for the printer, 'T' for tape, etc. The WHERE routine then sets a flag in memory (INFLG or OUTFLG). The selected device then becomes the active input device (AID) or the active output device (AOD). Subsequently the AIM input or output routine checks INFLG or OUTFLG and calls the appropriate input or output routine for the chosen device. When the user chooses the 'U' code an indirect jump is performed to a vector on page one. There is a vector for user input and one for user output. These vectors must be set to point to the I/O drivers for the user device. The user routine is called once from The WHERE subroutine to perform initialization for the device and once from the input or output routine for each character to be processed. So that the user routines can distinguish between a call for initialization and a call for I/O the carry flag of the processor status register is clear for a call for initialization and set for a call for I/O. Also the output routine must pull the character to be output from the stack. The user routines return to the AIM monitor via an RTS (ReTurn from Subroutine) instruction. The above is all explained in the AIM manual. There are a few other important considerations though.

The manual does not mention the fact that the user routines must preserve the contents of the registers. The input routine must preserve X and Y, and the output routine must preserve X, Y, and A. The AIM monitor provides two subroutines for saving the X and Y registers on the stack, and

```
937A            ;****************************************
937A            ;*
937A            ;*       ENTRY FOR USER OUTPUT
937A            ;*
937A            ;****************************************

937A B01D    USERO  BCS OUTCHR       ;CARRY SET FOR OUTCHAR
937C 209EEB         JSR PHXY         ;SAVE REGS
937F 20F797         JSR OPNCHK       ;IS FCB OPEN?
9382 D03B           BNE AOPN         ;YES, ERROR
9384 20E69A         JSR FILEO        ;GET FILE NAME
9387 A015           LDY #NUMSEC      ;CLEAR NUMBER OF SECTORS
9389 A900           LDA #0
938B 91F8           STA (FCBPTR),Y
938D 201298  OPENU  JSR OPEN         ;OPEN FILE
9390 B017    OPNBK  BCS ERROUT
9392 20619B  NOIERO JSR ROLIN        ;RESTORE ZERO PAGE
9395 20ACEB  UBACK  JSR PLXY         ;RESTORE REGS
9398 60      SKIPUO RTS

9399            ;   OUTPUT ONE CHARACTER TO DISK

9399 68      OUTCHR PLA              ;GET OUTPUT BYTE
939A C9FF           CMP #$FF         ;DELETE FFs
939C F0FA           BEQ SKIPUO
939E 209EEB         JSR PHXY         ;SAVE REGS
93A1 209B9B         JSR POINTO       ;INIT PTRS
93A4 20C799         JSR PUTCHR       ;SEND CHARACTER
93A7 90EC           BCC UBACK

93A9            ; ERROUT : PUT ERROR MESSAGE

93A9 48      ERROUT PHA              ;SAVE ERROR CODE
93AA 20FEE8         JSR LL           ;RESET I/O FLAGS
93AD 2094E3         JSR CKER00       ;DISPLAY ERROR
93B0 A924           LDA #'$'         ;DISPLAY ZAPDOS ERROR CODE
93B2 20ECE9         JSR OUTALL
93B5 68             PLA
93B6 2046EA         JSR NUMA         ;IN HEX
93B9 20619B  JRET   JSR ROLIN        ;RESTORE ZERO PAGE
93BC 4CA1E1         JMP COMIN        ;BACK TO MONITOR

93BF A90C    AOPN   LDA #OPFILE      ;OPEN FILE ERROR
93C1 D0E6           BNE ERROUT

93C3            ;****************************************
93C3            ;*
93C3            ;*       ENTRY FOR USER INPUT CALL
93C3            ;*
93C3            ;****************************************

93C3 B00F    USERI  BCS INCHR        ;CARRY SET FOR INPUT
93C5 209EEB         JSR PHXY         ;SAVE REGS
93C8 20279B         JSR INITB        ;INIT HI ZERO PAGE
93CB 208E9B         JSR POINTI       ;INIT PTRS
93CE 20E69A         JSR FILEO        ;GET FILE NAME
93D1 4C8D93         JMP OPENU        ;GO OPEN

93D4            ;   GET A CHARACTER FROM DISK

93D4 209EEB  INCHR  JSR PHXY         ;SAVE REGS
93D7 208E9B         JSR POINTI       ;INIT POINTERS
93DA BA             TSX
93DB BD0401         LDA $0104,X      ;WHERE CALLED FROM?
93DE C9D6           CMP #$D6         ;ASSEMBLER OPEN CALL?
93E0 F004           BEQ ASOP         ;YES, GO GET NEXT FILE
93E2 C9D4           CMP #$D4
93E4 D018           BNE GETIT        ;NO, JUST GET NEXT CHAR
93E6            ;  ASSEMBLER WANTS A NEW FILE
93E6 A920    ASOP   LDA #' '
93E8 8D3DA4         STA DIBUFF+5     ;MARK END OF FILE NAME
93EB            ;       ENTRY FOR PL65 ,DFILE ROUTINE
93EB 20549B  PL65IN JSR ROLOUT       ;SAVE ZERO PAGE
93EE A900           LDA #0           ;CLEAR BUFFER INDEX
93F0 8522           STA CURCHR
93F2 A006           LDY #DRIVE
93F4 B1F8           LDA (FCBPTR),Y   ;RETRIEVE DRIVE# FROM FCB
93F6 8523           STA DRVSAV
93F8 201D98         JSR REOPEN       ;OPEN NEW FILE
93FB 4C9093         JMP OPNBK        ;FINISH

93FE 204899  GETIT  JSR GETCHR       ;A CHARACTER
9401 9092           BCC UBACK        ;NO ERROR



9403 C90E           CMP #ENDOFI      ;END OF FILE?
9405 F08E           BEQ UBACK        ;YES, CONTINUE
9407 D0A0           BNE ERROUT       ;NO, MUST BE ERROR
```

restoring them without affecting the A register. These are PHXY and PLXY. I found need to investigate these two interesting subroutines when I tried to enter PLXY via a JMP (Jump) instruction instead of a JSR (Jump to SubRoutine) instruction. Whenever a subroutine call is the last instruction of another subroutine I usually enter it with a JMP instruction instead of a JSR followed by a RTS. This saves one byte and normally works the same. This does not work for PHXY or PLXY because of the way they manipulate the stack. When a subroutine is entered via JSR, the return address is the last two bytes on the stack. Anything that the subroutine pushes on the stack must be pulled off before it can return properly. To get around this problem PHXY and PLXY use a third subroutine called SWSTAK (SWap the STAcK). SWSTAK swaps the 2 bytes that are 2 locations back on the stack with the 2 bytes that are 4 locations back on the stack. So, PHXY pushes the X and Y registers onto the stack and calls SWSTAK. SWSTAK swaps the X and Y bytes with the return address for PHXY and then returns to PHXY. PHXY then returns to its caller with the X and Y register values next on the stack. PLXY works just the opposite. It first calls SWSTAK to swap the saved X and Y registers with its return address. It then pulls X and Y off the stack and returns. If PLXY is entered via a JMP instead of a JSR the stack is not in the expected condition and PLXY ends up returning to the address contained in the saved X and Y registers giving unpredictable results. As long as they are used properly, PHXY and PLXY can be used by the user I/O routines to save and restore the X and Y registers.

A problem I had with the output routine is detecting the end of the output stream. Some devices such as tape and disk need to have a termination or 'close' routine that is executed after all output is complete. This routine must write the last buffer or, as in the case of disk, update the directory. The AIM output routine gives the user routine no indication of when output is complete. There is no consistant way to determine this from the data itself. I solved this problem by using one of the AIM user function keys to execute the routine to close the output file. This means that I must remember to push that key after each use of user output. This is inconvenient but the only feasible way to solve the problem.

```
0000          ; PL65 .DFILE INTERFACE
0000          ;
0000          ;   PL65 DATA AREAS
0000          ;
0000          PLDRIV =$0158         ;PL65 SAVEA DRIVE NUMBER HERE
0000          PLBUFF =$014B         ;PL65 SAVES FILE NAME HERE

0000          ;   AIM ADDRESSES

0000          DIBUFF =$A438         ;DISPLAY BUFFER
0000          PHXY   =$EB9E         ;SAVE X AND Y REGS

0000          ;   ZAPDOS ADDRESSES

0000          POINTI =$9B8E         ;POINT TO INPUT FCB
0000          PL65IN =$93EB         ;ENTRY FOR DFILE INTERFACE
0000          FCBPTR =$F8           ;POINTER TO FCB
0000          DRIVE  =6             ;OFFSET TO DRIVE # IN FCB
0000          ;
0000                 *=$112
0112          ;F3 VECTOR
0112 4CD00F          JMP DFILE      ;

0115                 *=$0FD0
0FD0          ;
0FD0 209EEB   DFILE  JSR PHXY       ;SAVE REGS
0FD3 A206            LDX #6
0FD5 BD4B01   DLUP   LDA PLBUFF,X   ;COPY FILE NAME TO DIBUFF
0FD8 9D38A4          STA DIBUFF,X
0FDB CA              DEX
0FDC 10F7            BPL DLUP
0FDE A920            LDA #$20
0FE0 8D3EA4          STA DIBUFF+6   ;MARK END OF NAME
0FE3 208E9B          JSR POINTI     ;POINT TO INPUT FCB
0FE6 AD5801          LDA PLDRIV     ;GET PL65 DRIVE
0FE9 C9FE            CMP #$FE       ;ANY SPECIFIED?
0FEB F007            BEQ NODRV      ;NO, SKIP
0FED 6A              ROR A
0FEE 6A              ROR A          ;SHIFT TO 2 HI BITS
0FEF 6A              ROR A
0FF0 A006            LDY #DRIVE
0FF2 91F8            STA (FCBPTR),Y ;AND SAVE IN FCB
0FF4 4CEB93   NODRV  JMP PL65IN     ;ENTER ZAPDOS
0FF7                 .END
0FF7           ERRORS= 0000
```

```
0000          ;  TIMER BUG VERIFICATION PROGRAM

0000          DI1024 =$A497          ;TIME X 1024
0000          RINT   =$A485          ;TIME OUT
0000          COMIN  =$E1A1          ;RETURN TO MONITOR
0000          CUREAD =$FE83          ;INPUT A CHARACTER
0000          RED2   =$E976          ;ECHO A CHAR

0000                 *=$200

0200 2083FE   READ   JSR CUREAD      ;READ A CHAR
0203 48              PHA             ;SAVE IT
0204 A9FF     SET    LDA #$FF        ;255 X 1024
0206 8D97A4          STA DI1024      ;START TIMER
0209 2C85A4          BIT RINT        ;TIME UP ALREADY?
020C 1003            BPL LUP         ;NO, TRY AGAIN
020E 4CA1E1          JMP COMIN       ;EXIT ON TIMER ERROR
0211 2C85A4   LUP    BIT RINT        ;TIME UP?
0214 10FB            BPL LUP         ; NO, WAIT
0216 68              PLA
0217 2076E9          JSR RED2        ;ECHO CHAR
021A 4C0002          JMP READ        ;REPEAT
021D                 .END
021D           ERRORS= 0000
```

An even greater problem is how to handle end of file on input. My disk routines detect end of file and return a condition code, but there is no way to tell the AIM routine that there is no more data. Each different AIM program detects the end from the data in its own way. The 'L' command uses a zero length record; the editor uses two successive CRs (carriage returns); the assembler uses a .END statement followed by two CRs; BASIC uses a CTL-Z. Another inconsistancy in the 'L' command causes it to try to read 5 or 6 more characters from the final zero length record than the 'D' command wrote. The user input routine must compensate for this and provide pad characters or the 'L' command will not terminate properly. When the KBD/TTY switch is in the TTY positionn, and OUTFLG is set to 'U', the AIM CRLF routine inserts an LF(Line Feed) and a null (AIM uses hex FF for a null) after each CR. On input AIM does not expect these characters to be included. The 'L' command will ignore the LFs and nulls when inputting a line of data. The editor will ignore the LFs but not the nulls. The null between the two successive CRs that end the file cause the editor to fail to recognize the end and continue to request data. To solve this problem, the nulls must be deleted from either the input stream or the output stream. I chose to delete nulls from the output stream because this saves disk storage space.

The assembler requires that the source file be read twice, once for each pass. It is designed to read the source file from tape. Before starting pass 1 it saves the name of the tape file at location $A7. Before starting pass 2 it moves the name of the tape file back to the name buffer ($A42E) and to the display buffer ($A438). It then calls the tape open routine. If the source is coming from the 'U' device it moves the file name and then makes an extra call to the user input routine. However it does not indicate to the user input routine that the call is to open a file and not to read another character. The only way I could find to detect the extra call is to test the stack to see what page the call was from. The assembler also makes an extra call to the user input routine when it encounters a .FILE statement. The .FILE statement is used to link source files together so that programs too long for the editor buffer may be assembled. When the assembler encounters a .FILE statement it moves the file name to the name buffer and display buffer and makes a call to the user input routine. Again the only way to distinguish this call from a normal input call is by checking the stack. While investigating the .FILE statement I found an undocumented feature of the assembler. The .END statement may also contain a file name. If it does, then that file is used to start pass 2 instead of the one saved at location $A7. This allows pass 2 to start with a different file than pass 1. Of what use is

this feature? Sometimes it may be useful to have the first file of a program contain only label definitions (= directives). Since these statements only make entries in the symbol table and generate no code it is not necessary to read them again for pass 2. This feature can be used to save time and printer paper. You may have a file containing definitions for all the AIM subroutine addresses that you use for every program you write, but they still consume no extra space in the source file or the program listing. You can use the .END feature to omit the definition file from pass 2, but remember to set the program counter in the first file to be read in pass 2, instead of in the definitions file or assembly errors may result.

So, the user input must detect the special open calls by the assembler .FILE and .END statements, and the start of pass 2. This can only be done by checking the page of the calling routine. These calls are made from pages $D4 and $D6. It must also provide pad characters for the 'L' command. The user output routine must delete all null characters so that the editor can properly recognize the end of the file on input. As an example I have included listing 1 which is the user I/O interface for my disk system. The disk routines are not included. These routines work with all AIM commands that use the AID or AOD including 'L', 'D', the editor, the assembler, BASIC, and PL/65.

PL/65 also has a linked file feature. It uses the .TFILE statement to link tape files and the .DFILE statement to link disk files. The .TFILE statement causes the tape open routine to be executed. The .DFILE statement executes the disk open routine through the user F3 key vector. This makes the disk open interface much more straightforward than the way the assembler does it. PL/65 stores the file name at location $14B and the drive number at location $158 before calling the F3 key routine. Listing 2 is the routine needed to implement the .DFILE statement.

Another consideration is console communications. It may be necessary to request information from the user during user I/O. For example, my disk initialization routines prompt the user for the name of the file to be used. Care must be taken which AIM monitor subroutines are used at this time. Some communicate with the keyboard and display only, while others use the AID or AOD. Normally the keyboard and display are the AID and AOD so calling OUTPUT and OUTALL, for example, give the same results. After the WHERE subroutine is executed the AID or AOD has changed. If the user output initialization calls OUTALL instead of OUTPUT to display a character, it will end up calling itself. The results are unpredictable. There are two ways to solve this problem. Either be sure to call only keyboard/display routines or change the AID or AOD to the keyboard/display before attempting to communicate with the user, and restore the AID or AOD after communications is complete.

You can make your programs device independent also. To make a program device independent, you must call the WHERE subroutines before doing any I/O that is to be device independent. The data must be read or written with subroutines that use the AID or AOD. Here again care must be taken to use only AID and AOD subroutines and not the keyboard/display subroutines. Calling a wrong subroutine could cause part of the data to end up on the display or the program could hang up waiting for input from the keyboard. Subroutine LL can be used to return the keyboard and display to the AID and AOD. Even BASIC programs can change the AID and AOD by calling the WHERE subroutines with the USR function. This will cause BASIC input or output to be redirected. Of course when using some devices such as tape or disk there are close routines that must be executed to terminate output. Routine DU12 ($E511) does this for AIM tape output.

Another discovery that I made is that AIM's 6532 timer has the same bug as KIM's 6530 timer. KIM's 6530 has a bug that causes it to ignore a start command on the average of once every 256 starts. I chose to use the 6532 because it has a maximum interval of more than a quarter of a second, while the 67522 timer can only time up to 65 miliseconds. I was immediately suspicious of the 6532 when I saw that it works exactly like the 6530 and that the two chips have other similar features. So I wrote a small program to prove the prescence of this bug. Listing 3 is the result.

The 6530/32 timer, when not in use, is continuously counting down from $FF to 0 at the rate of the CPU clock signal. The bug occurs when the CPU trys to store a starting value in the counter register just as the count is passing zero. When this occurs the timer ignores the CPU. The result is an immediate time out the first time the program checks the timer. The routine in listing 3 proves that the 6532 is guilty. It first reads a character from the keyboard to get a random starting time for the timer. It then starts the timer for about a quarter second and checks to see if there is an immediate time out. If there is, it returns to the monitor. If there is not an immediate time out it enters a loop and waits for the timer to time out. Then it echos the character and repeats the sequence. If the timer is working correctly the program can never end, unless the escape key is used. It may take several hundred trys sometimes, but this program will always eventually catch the timer bug and return to the monitor.

This bug will cause occasional errors in my disk system if not circumvented. The way to circumvent this bug is to always use 2 successive stores