BDS C Console I/O: Some Tricks and Clarifications

Leor Zolman
BD Software
Cambridge, Massachussetts

In this document I will attempt to remove some of the mystery behind the CP/M console I/O mechanisms available to BDS C users. When the major documentation for BDS C (i.e. the User's Guide) was being prepared, I had mistakenly assumed that users would automatically realize how the "bdos" and "bios" library functions could be used to perform all CP/M and BIOS functions, especially direct console I/O (by which the system console device may be operated without the frustrating unsolicited interception of certain special characters by the operating system.) In fact, the use of the "bios" function for such purposes might only be obvious to experienced CP/M users, and then only to those having assembly language programming experience with the nitty-gritty characteristics of the CP/M console interface. Let's take a look at what really happens during console I/O...

The lowest (simplest) level of console-controlling software is in the BIOS (Basic Input/Output System) section of CP/M. There are three subroutines in the BIOS that deal with reading and writing raw characters to the console; they are named `CONST' (check console status), `CONIN' (wait for and read a character FROM the console), and `CONOUT' (send a character TO the console). The way to get at these subroutines when you're writing on the assembly language level is rather convoluted, but the BDS C library provides the `bios' function to make it easy to access the BIOS subroutines from C programs. To check the console status directly, you use the subexpression `bios(2)', which returns a non-zero value when a console character is available, or zero otherwise. To actually get the character after `bios(2)' indicates one is ready, or to wait until a character is ready and then get it, use `bios(3)'. To directly write a character `c' to the console, you'd say `bios(4,c)', but note that the BIOS doesn't know anything about C's convention of using a single '\n' (newline) character to represent a logical carriage-return/linefeed combination. The call `bios(4,'\n')' will cause ONLY a single linefeed (ASCII 0x0A) character to be printed on the console.

Making sure that all console I/O is eventually performed by way of these three BIOS subroutines is the ONLY way to both keep CP/M from intercepting some of your typing and insure the portability of programs between different CP/M systems. (1)

The BDOS (Basic Disk Operating System) operations are the next higher level (above the BIOS) on which console I/O may be performed. Whenever the standard C library functions `getchar' and `putchar' are called, they perform their tasks in terms of BDOS calls...which in turn perform THEIR operations through BIOS calls, and this is where most of the confusion arises. Just as there are the three basic BIOS subroutines for interfacing with the console, there are three similar but "higher level" BDOS operations for performing essentially the same tasks. These BDOS functions, each of which has its own code number distinct from its BIOS counterpart, are: "Console Input" to get a single character from the console (BDOS function 1), "Console Output" to write a single character to the console (BDOS function 2), and "Get

---

1. Even so there's no way to know what kind of terminal is being used--so "truly portable" software either makes some assumptions about the kind of display terminal being used (whether or not it is cursor addressable, HOW to address the cursor, etc.) or includes provisions for self-modification to fit whatever type of terminal the end-user happens to have connected to the system.

Console Status" to determine if there is a character available from the console input (BDOS function 11). The BDOS operations do all kinds of things for you that you may not even be fully aware of. For instance, if the BDOS detects a control-S character present on the console input during a console output call, then it will sit there and wait for another character to be typed on the console, and gobble it up, before returning from the original console output call. This may be fine if you want to be able to stop and start a long printout without having to code that feature into your C program, but it causes big trouble if you need to see EVERY character typed on the console, including control-S. A little bit of thought as to how the BDOS does what it does reveals some interesting facts: since it must be able to detect control-S on the console input, the BDOS must read the console whenever it sees that a character has been typed. If the character ends up not being a control-S (or some other special character that might require instant processing), then that character must be saved somewhere internally to the BDOS so that the next call to `Console Input' returns it as if nothing happened. Also, the BDOS must make sure that any subsequent calls made by the user to `Get Console Status' (before any are made to `Console Input') indicate that a character is available. This leads to a condition in which a BDOS call might say that a character is available, but the corresponding BIOS call would NOT, since, physically, the character has already been gobbled up by the BDOS during a prior interaction with the BIOS.

If this all sounds confusing, bear in mind that it took me several long months of playing with CP/M and early versions of the compiler before even I understood what the hell was going on in there. My versions of `getchar' and `putchar' are designed for use in an environment where the user does NOT need total direct control over the console; given that the BDOS would do some nice things for us like control-S processing, I figured that I might as well throw in some more useful features such as automatic conversion of the '\n' character to a CR-LF combination on output, automatic abortion of the program whenever control-C is detected on input or output (so that programs having long or infinite unwanted printouts may be stopped without resetting the machine, even when no console input operations are performed), automatic conversion of the carraige-return character to a '\n' on input, etc. One early user remarked that he would like `putchar' to be immune from control-C; for him I added the `putch' library function, which works just like `putchar' except that control-C's would no longer stop the program. Much later it became evident that neither `putchar' nor `putch' suffice when CP/M must be prevented from ever even sampling the physical console input. At this point I added the `bios' function, so that users could do their I/O directly through the BIOS and totally bypass the frustrating character-eating BDOS.

I promised some examples earlier, so let's get to it. First of all, here is a very rudimentary set of functions to perform the three basic console operations in terms of the `bios' function, with no special conversions or interceptions AT ALL (i.e., nothing like the '\n' --> CR-LF translations):

```
/*
        Ultra-raw console I/O functions:
*/

getchar()       /* get a character from the console */
{
        return bios(3);
}

kbhit()         /* return true (non-zero) if a character is ready */
{
        return bios(2);
}

putchar(c)      /* write the character c to the console */
char c;
{
        bios(4,c);
}
```

These ultra-raw functions do nothing more than provide direct access to the BIOS console subroutines. If you include these in your C source program, then the linker will use them instead of the standard library versions of the similarly named functions--provided that some direct reference to them is made before the default library file (DEFF2.CRL) is scanned. Usually, in programs where such functions are necessary, there will be many explicit calls to `getchar' and `putchar' to insure that the library versions aren't accidentally linked. A good example of a case where trouble might occur is when the entire program consists of, say, a single `printf' call followed by a custom version of `putchar'. Since the linker won't know that `putchar' is needed until after `printf' is loaded from the library, the custom version of `putchar' will be ignored and the old (wrong) version will be picked up from the DEFF2.CRL library file. The way to avoid such a problem is to insert, somewhere in the source file, explicit calls to any functions that are a) NOT explicitly called otherwise, and b) named the same as some library function. This isn't an expecially neat solution, but it gets the job done.

OK, with that out of the way, let's consider some more sophisticated games that can be played with customized versions of the console I/O functions. For starters, how about a set that performs conversions just like the library versions, detects control-C, and throws away any characters typed during output (except control-C, which causes a reboot)? No problem. What's needed is automatic conversion of '\n' to CR-LF on output; conversion CR to '\n' and ^Z to -1 on input with automatic echoing; and re-booting on control-C during both input and output.

```
/*
        Vanilla console I/O functions without going through BDOS:
        (`kbhit' would be the same as the above ultra-raw version)
*/

#define CTRL_C 0x03     /* control-C */
#define CPMEOF 0x1a     /* End of File signal (control-Z) */

getchar()       /* get a character, hairy version  */
{
        char c;
        if ((c = bios(3)) == CTRL_C) bios(0);   /* on ^C, reboot  */
        if (c == CPMEOF) return -1;             /* turn ^Z into -1  */
        if (c == '\r') {                /* if CR typed, then    */
                putchar('\r');          /* echo a CR first, and set */
                c = '\n';               /* up to echo a LF also */
        }                               /* and return a '\n'    */
        putchar(c);                             /* echo the char  */
        return c;                               /* and return it  */
}

putchar(c)      /* output a character, hairy version  */
char c;
{
        bios(4,c);              /* first output the given char */
        if (c == '\n')          /* if it is a newline, */
                bios(4,'\r');   /*      then output a CR also */
        if (kbhit() && bios(3) == CTRL_C)       /* if ^C typed, */
                bios(0);                        /* then reboot  */
}                               /* else ignore the input completely */
```

Now, if you wanted to have control-S processing and a push-back feature (the two are actually quite related, since you must be able to push back anything except control-S that might be detected during output), you could add some external "state" to the latest set of functions and keep track of what you see at the console input. Once this is done, though, you're probably better off going back to the original library versions of `getchar' and `putchar', which let the BDOS handle all that grungy stuff.

Incidentally, CP/M version 2.x has a new BDOS function which supposedly makes it easier to perform some of the direct console I/O operations that required the BIOS calls for CP/M 1.4. While this might be useful for people having CP/M 2.x, it would render any software developed using the new BDOS feature autistic when run on CP/M 1.4 systems. Please keep that in mind if you ever write any software on your 2.x system for use on other (perhaps non-2.x) systems.

So far, everything I've talked about has been in terms of the BIOS, and applies equally to all CP/M systems. Unfortunately, there is one console operation often needed when writing real-time interactive operations that is not supported by the BIOS, and thus there is no portable way to implement it under CP/M. What's missing is a way to ask the BIOS if the console terminal is ready to ACCEPT a character for output. An example of the trouble this omission causes is evident in the sample program RALLY.C; the case there is that the program must be able to read input from the keyboard at any instant, and cannot afford to become tied up waiting for the terminal when the amount of data being sent to it has caused the X-ON/X-OFF protocol to lock up the program until a character can be sent. Given that the only "kosher" way

to send a character to the console is through the CONOUT BIOS call, and that such a call might at any time tie up the program for longer than is tolerable, the only recourse is to bypass CONOUT completely and construct a customized output routine in C that can be more sophisticated. This is done in RALLY.C, at the expense of non-portability for the object code; each user must individually configure his BDSCIO.H header file to define the unique port numbers, bit positions and polarities of the I/O hardware controlling his console. It would have been SO much easier if the BIOS contained just one more itty bitty subroutine to test console output status...but NoooooOOOOOOOOoooooo, they had to leave that one OUT so we have to KLUDGE it...

Sorry. I get carried away sometimes. Oh well...I hope this has helped to demystify some of the obscure behavior sometimes evident during console I/O operations. For the low-down on how the library versions of `getchar', `putchar', etc. really work, see their source listings in DEFF2.ASM. And if there's something you want to do with the console and can't figure out how despite this document, I'm always available for consultation (at least whenever I'm near the phone.)

Good luck.