BBL
Programming Logic Manual

David Todd
Computing Center
Wesleyan University
Middletown, CT

August 1992

Introduction

BBL is an installable software device driver for MS DOS that displays on the screen, in big block letters, any characters sent to it. It is a memory-resident (800 bytes of memory) program that can be used by any program or utility as an output device, much as `CON:` might be used for input or output.

This manual explains how BBL is organized so that others can customize or revise it as needed.

Design Philosophy

BBL was implemented in response to a request from a faculty member in Chemistry for a video display system that would let him display on a DOS-based computer the results of real-time measurements from experiments in classroom demonstrations. The system was to replace a system he had used a Commodore PET that he had retired. The new system was to be an AT&T 6300 computer with GPIB interface, and he was adapting his BASIC code to that system. He needed a display system that would let him iterate the loop {read-data, compute values, display results}.

I looked on the FTP-able archives for a "bulletin board" program with source code that I could adapt, but found none and concluded that I would have to develop the code.

I realized that it would be prudent to develop the code so that it could be used from any language and be portable to other DOS machines (those 6300's can only last so long!). It seemed to me that the best way to implement the system would be to write it as a device driver — make it a small, portable piece of code that, once installed, could be accessed by programmers using their favorite language with no concern for the underlying support code.

I tried to develop the code as an assembly-language shell around a Turbo-C core, but I have been unable to get that to load correctly. I'll distribute that if I ever figure out how to get it working.

Since I made some design decisions that other programmers might want to alter, I decided to write this PLM (Programmer's Logic Manual) to guide others in changes they might want to make and note the reasons for my choices. Some of the basic decisions were:

- Use the 8x8 font ROM built into the video adapter of most (all?) DOS machines. Rationales: from a distance, the 8x8 ROM, displayed as 8x8, 16x16, or 24x24 cells of block characters on the screen should be sufficiently readable; embedding the character pixel patterns in the code would increase the size of the resident device driver unnecessarily; the code would be less portable if it were designed to use a base 16x16 pixel pattern in 43x132 EGA display mode, for example (in particular, it wouldn't work on the AT&T 6300 for which the code was intended!). However, the core display routines use 16-bit masks and retrieve 16-bit quantities as pixel patterns. The character width is an assembly parameter, and adaptation to 16x16 pixel patterns should not be too difficult.

- Use BIOS INTs to display — don't access video memory directly. Rationale: code portability was paramount. Video memory seems pretty standardized, but this didn't seem worth the risk. The display loop is very well defined, and the display INT is well identified: easy to change to direct addressing if speed is an issue.

- Use 8086-compatible code. Rationale: there were only a few situations where shifts or multiplies with immediate-mode arguments would have made the code shorter and faster. I didn't bother to insert conditional-assembly options, either, because the penalties are pretty small.

- Implement commands through the input character stream rather than use the device-driver IOCTL feature. Rationale: few casual programmers (for whom this was intended) would want to master the IOCTL calling sequence in their preferred high-level language; only a few special symbolic characters are made undisplayable by using them as command-introducers (and they are parameterized); and the IOCTL mode of control can be introduced easily, if desired, by adding the IOCTL code to invoke the control routines and changing the command-interpreting state tables.

- Implement command-character interpretation through a state machine. Rationale: Once implemented, the state tables can be changed easily to implement new commands or rearrange the way they're processed; processing within the device driver is fast.

Code Organization

The code skeleton is pretty standard device-driver code (see Lai's book, for example). I put the equates at the beginning for easy editing. The STRUC definitions used by the BBL -specific code follow. Then the STRUC definitions and equates used in standard device-driver code (some not needed for this program have been left in).

The device-driver "strategy" section is standard code. The command dispatching upon invocation of the device-driver "interrupt" code is also standard.

This particular device does not hook into the interrupt system, so there are no timing issues and no places at which the interrupts must be disabled.

There are few commands to implement. The *Initialization* command and *Output* command are really the only ones that require code. The others are ignored; the code for handling them is standard.

The State Machine

The processing of characters sent to BBL is handled by a state machine. The incoming character is classified by a type ("token type") that indicates if the character can be interpreted as a command or command parameter or is simply to be displayed. The state machine is a matrix with rows identified by states and columns identified by the token type of the incoming character; the elements of the matrix identify the action to be taken and resulting next state when a token of the specified type is processed when the machine is in each particular state.

In version 1 of BBL, the commands available, invoked by the incoming character stream, are:

⟨**FF**⟩ clear screen & move cursor to home

⟨**BS**⟩ move cursor back one character position (same line)

⟨**CR**⟩ move cursor to beginning of line

⟨**LF**⟩ move cursor down one line (or scroll up)

⟨**VT**⟩⟨**digit**⟩ position cursor on line ⟨digit⟩; e.g. ⟨VT⟩1 positions cursor on line 1, no change in column position

⟨**HT**⟩⟨**digit**⟩ position cursor on column ⟨digit⟩; e.g., ⟨HT⟩3 positions cursor on column 3, no change in line position

⟨**SO**⟩⟨**digit**⟩ with ⟨digit⟩ = [0—1—2] means set display mode to 0 (3rows x 10char/row), 1 (1x5), or 2 (1x3).

Accordingly, the state machine has four states:

```
0:      waiting-for-character
1:      VT-waiting-for-digit
2:      HT-waiting-for-digit
3:      SO-waiting-for-digit
```

and the character set (256 chars) is classified by the following types:

```
0:      non-action char --- just display (any but the following)
1:      digit char {0..9}
2:      move-cursor {FF, BS, CR, LF}
3:      Hor-position {HT}
4:      Vert-position {VT}
5:      Set-mode {SO}
```

Note that the whole 256-char set has displayable characters in the 8x8 ROM, so all could be displayed except the ones used for control functions here. If you want to extend BBL to display the symbol located at the place in ROM addressed by one of these action chars, you'll need to add an escape char that can be used to prefix the control chars used above and modify the state tables accordingly.

With the definition of tokens types and states above, the following state table is used to define the actions and transitions:

Token Type State	char 0	digit 1	move 2	ht 3	vt 4	so 5
0	Emit, 0	Emit, 0	Act, 0	Noop, 2	Noop, 1	Mode, 3
1	beep, 0	VPos, 0	beep, 0	beep, 0	beep, 0	beep, 0
2	beep, 0	HPos, 0	beep, 0	beep, 0	beep, 0	beep, 0
3	beep, 0	beep, 0	beep, 0	beep, 0	beep, 0	SetMode, 0

The actions include:

```
Emit:    display the character coming in from the input stream
Act:     perform the action indicated by the cursor-moving command
beep:    error: sound bell, reset state
HPos:    position cursor horizontally
VPos:    position cursor vertically
SetMode: set mode for display.
```

Incoming characters are categorized by direct table addressing, using the value of the character as an index (actually, the value of the character, divided by two is the index since the 4-bit token-class values are stored two nibbles per byte). The token-class table ("TC") is built dynamically at initialization time (see below).

The state table is constructed as a matrix using a STRUC definition that gives the action procedure and next state associated with each ⟨current-state,token-class⟩ pair. This is a static matrix.

Using the state table makes the code easier to modify (just modify the state table and categorize the token type of any new command characters), and it makes the processing of commands in the input stream very fast (direct table addressing to find the routine to invoke and next state).

Code Description

The best way to understand the code is to read it, but this might help.

Initialization

Recall that device drivers are called once when loaded through the strategy section to pass a DOS pointer, then again through the interrupt section using the *Initialization* function code. These sections are called just once. The *Initialization* section can do whatever it needs to set up the device driver, then dispose of much of itself, and one-time code, by returning to DOS a pointer that indicates that the first byte of free space (reused by DOS) is at the address of the once-only code.

The initialization code for BBL has to process any command-line options, set up the pointer to the 8x8 ROM font, build the token-class table, and announce that it has been loaded.

The command-line processing, located in the initialization section, is straightforward recursive-descent code ... not pretty, but it works. I tried to leave a road map for future extension.

The address of the 8x8 ROM font is assumed to be accessible via BIOS calls (true for EGA and VGA, for Phoenix BIOS at least). The command-line options may specify some other choice, and the appropriate address is selected if the option was invoked.

The token-class table is set up so that all characters default to class 0, displayable. The token-class table must reside in the device driver during execution. The exceptions to the default settings are listed in a separate list in the initialization section. The initialization code processes the exceptions from that list into the memory-resident "TC" table, and the exceptions list is then discarded by the init code since it is in the "once-only" section of the program.

The banner announcing that BBL has been loaded is part of the once-only code and uses the standard DOS INT for string display.

Output: Character Processing

The device driver can be given a string of characters to process at once, so ther output code contains a loop to process all characters in the input buffer. For each of those characters, the following process is followed in a procedure invoked in the string-processing loop.

The output character is classified by using its value to select the correct nibble from the TC table addressed by the character value divided by two (left nibble for even-valued characters, right for odd-valued characters). The current state and token-class value are used to compute the matrix address of the correct state-table entry, the next state value is loaded and the action-procedure associated with this state and input character token-type is executed. Upon completion, control is returned to the invoking procedure.

Author, Copyright, and Distribution

BBL was written by David Todd, Chemistry Department and Computing Center, Wesleyan University, August, 1992; for John Sease, Chemistry Department. Author's address:

```
H. David Todd
Computing Center
Wesleyan University
Middletown, CT  06459
email via Internet: hdtodd@eagle.wesleyan.edu
```