Writing Custom Commands in FreeBSD's DDB Kernel Debugger

BY JOHN BALDWIN

DDB is an interactive kernel debugger that can be used to inspect system state and control the running kernel. DDB was first developed as part of the Mach operating system. It was later ported to 386BSD from which it was inherited by various operating systems including FreeBSD, NetBSD, and OpenBSD. This article focuses on the implementation of DDB in FreeBSD.

DDB runs on the system console, and system execution is suspended while the debugger is active. This permits inspection of the system in a consistent state. DDB can be entered manually, but DDB is typically used after a kernel panic. FreeBSD's kernel can be configured to enter DDB after a kernel panic permitting a user or system administrator to examine the system state before rebooting. Debugging kernels built from FreeBSD's main branch do this by default.

DDB provides many features common to debuggers. It supports run control such as single stepping and breakpoints, and also supports hardware watchpoints on platforms with suitable hardware support. DDB includes several commands to display information about a system including stack traces and memory dumps.

Unlike many other debuggers, DDB does not understand type information and is not able to pretty-print structures or evaluate members of structures or unions in expressions. However, DDB can be extended by defining new commands. New commands can even be implemented in kernel modules which can be loaded after boot.

DDB Execution Context

DDB executes in a special context which differs from the normal kernel execution context in several ways:

- When DDB is active, the system is paused and borrows the execution context of the currently executing thread on each CPU. The normal kernel scheduler does not function in this context and the borrowed threads on each CPU are not permitted to context switch. This means that code execution in this context must not sleep or block on locks.
- If a fault or trap occurs during execution of a DDB command, the current thread will use `longjmp()` to resume execution in DDB's main loop.
- DDB accesses console devices directly for console input and output.

Due to these unique behaviors, implementations of DDB commands should adhere to the following guidelines:

- Commands should avoid side effects. If a fault occurs during command execution, there is no way to undo any side effects. The safest approach is to avoid side effects when possible.
- Commands should not use locks. Since execution is paused on all CPUs, the state of most data structures in the system will not be changing, so locks are not needed to synchronize with other CPUs. In addition, acquiring a lock is a side effect that will not be unwound if a command faults while holding a lock. In exceptional cases, where a command wishes to modify system state in a safe way, a command may use try locks. One command which does this currently is the kill command which can send a signal to a process.
- Commands should avoid complicated APIs. Higher level APIs often modify system state or contain other side effects such as acquiring locks.
- Most commands in DDB inspect system state without modifying it and output a human readable description of some portion of system state. Many of these commands are pretty-printers which print information about a specific data structure or list of structures.
- Commands must use DDB's APIs for console input and output. Mostly this means using **`db_printf()`** for output instead of **`printf()`**.

DDB provides a simple API for console output. The **`db_printf()`** function is similar to the normal kernel **`printf()`** and supports all of the same format specifiers. This function writes directly to console devices bypassing the system log device.

In addition, **`db_printf()`** includes simple pager support. Each time a newline is output to the console, **`db_printf()`** checks if the output should be paused. If so, **`db_printf()`** outputs a prompt on the console permitting the user to control how many lines are displayed before the next pause. Once the user has responded to the prompt, **`db_printf()`** returns. If the user requests the current command to quit (stop generating output), **`db_printf()`** sets the global variable **`db_pager_quit`** to a non-zero value. If a command generates output in a loop (for example, using a loop to walk a linked-list of data structures), the command should check **`db_pager_quit`** in each loop iteration and break out of the loop early if it is set.

Command Functions

Most DDB commands follow a simple syntax described in ddb(4):

```
command[/modifier] [address[,count]]
```

When a user enters a command at DDB's prompt, DDB parses this command line. The address and count fields are treated as expressions which can contain references to named symbols and many C arithmetic operators. The command and modifier fields are treated as simple strings. DDB uses the command field to locate a pointer to a C function. This C function is invoked to execute the command.

The functions implementing DDB commands use the following signature:

```
void fn(db_expr_t addr, bool have_addr, db_expr_t count, char *modif)
```

The **addr** argument contains the address the command should operate on. This can either be an explicitly-provided address or the address used with the previous command. The **have_addr** argument is true if the address was provided explicitly. The count argument contains the value of the **count** field. If the count field was not specified, **count** is set to -1. The **modif** argument is a pointer to a C string containing the modifier field. If a modifier was not specified, then **modif** will point to an empty string.

Command functions are associated with command names via internal tables maintained by DDB. DDB provides helper macros to abstract most of the details of registering new commands. Each macro accepts two arguments: the first argument is the name of the command, and the second argument is the name of the C function to associate with the named command. In addition to registering the linkage in the table, these macros also provide the C function declaration and should be immediately followed by the function body. Each macro is associated with a specific command table. The **DB_COMMAND** macro defines a new top-level command. The **DB_SHOW_COMMAND** macro defines a new command in the "show" table. The **DB_SHOW_ALL_COMMAND** macro defines a new command in the "show all" table. For example **DB_SHOW_COMMAND(bar, db_show_bar_func)** defines a new "show bar" command. It also defines a new C function, **db_show_bar_func**, which provides the implementation of this command. It is best practice, but not required, to name the C functions associated with a command using the pattern **db_<command>_cmd**.

Listing 1 is the source to a simple command named "double". This command multiplies the address provided by the user by 2 and outputs the result. Listing 2 shows some use cases of this command. The output from the third case may be surprising, as 32 times 2 is certainly not 100. The reason for this behavior is that DDB parses integer values with a default base of 16 (controlled by DDB's internal **$radix** variable). In base 16, 32 evaluates to the decimal value of 50.

```
DB_COMMAND(double, db_double_cmd)
{
        if (have_addr)
                db_printf("%u\n", (u_int)addr * 2);
        else
                db_printf("no address\n");
}
```

Listing 1: Source for the "**double**" command

```
db> double
no address
db> double 4
8
db> double 32
100
```

Listing 2: Sample output for the "**double**" command

Commands with Custom Syntax

DDB commands do not have to use the simple syntax given above. Command functions can choose to support other syntaxes. Commands request this by passing an additional flag

when registering commands. A separate set of macros accept command flags as a third argument: `DB_COMMAND_FLAGS`, `DB_SHOW_COMMAND_FLAGS`, and `DB_SHOW_ALL_COMMAND_FLAGS`.

Two flags are available to control command line parsing. `CS_MORE` indicates that a command mostly follows the simple syntax, but that the command supports more than one address. When this flag is specified, the main loop of DDB will still parse the command line as normal, but it will not discard any remaining tokens from its lexer before invoking the command function. This allows the command function to parse additional options on the command line. The second flag, `CS_OWN`, indicates that the command function performs all of the parsing itself. When this flag is specified, the main loop of DDB stops parsing the command line after reading the name of the command. The command function uses DDB's lexer to parse the rest of the command line. Regardless of which flag is specified, the command function must call `db_skip_to_eol()` to discard remaining tokens from the current command line before returning.

DDB provides a few functions to parse command line arguments. `db_expression()` parses an arithmetic expression. This can consume multiple words of input and supports the full DDB expression syntax including symbol resolution and various C operators. If no more command line arguments were available, `db_expression()` returns 0. If an expression was successfully parsed, then `db_expression()` returns a non-zero value and stores the result of the expression in the value pointed to by its sole argument. If `db_expression()` encounters a syntax error while parsing an expression, it prints a message and aborts the current command via `longjmp()`. Command functions should avoid any side effects while calling `db_expression()` since they can't be unwound if the user provides invalid input.

Two other functions provide a lower level interface to DDB's lexer. `db_read_token()` parses the next token from the command line and returns a constant identifying the type of token parsed. The constants are named `t<TYPE>` and are defined in `<ddb/db_lex.h>`. Most of the constants are associated with C operators and other special tokens, but a few are useful for custom commands. `tEOL` is returned when the end of the command line is encountered. `tEOF` is returned for invalid input such as a number that contains invalid characters. `tIDENT` is returned when a word (identifier) is parsed. A copy of the word is saved in the global variable `db_tok_string`. `tNUMBER` is returned when a numeric value is parsed. The value is saved as an integer in the global variable `db_tok_number`. Note that DDB's lexer assumes that any word beginning with a decimal digit is a number, and that any word beginning with an alphabetic character, underscore, or backslash is an identifier. `db_unread_token()` inserts a single token to be returned by the next call to `db_read_token()`. The value passed to `db_unread_token()` is one of the `t<TYPE>` constants. Normally this function is used to put back the token just read from `db_read_token()` if the returned token was invalid or unexpected.

DDB provides two additional functions to handle parsing errors. `db_error()` prints out a caller-supplied message, flushes the lexer state, and invokes `longjmp()` to abort the current command and return to DDB's main loop. `db_flush_lex()` just flushes the lexer state discarding the current command line. `db_flush_lex()` can be used if a more detailed error message is desired or to unwind additional state if `longjmp()` is undesirable.

Listing 3 is the source to a command named "`sum`". This command computes a sum of all of the expressions given on the command line. It uses the `CS_MORE` flag and uses `db_expression()` in a loop to parse additional expressions from the command line. Listing 4

shows some sample output from this command. Note that in the third case, **db_expression()** parsed the expression "**9 * 3**" and returned the value **27** to the loop in **db_sum_cmd()**.

```
DB_COMMAND_FLAGS(sum, db_sum_cmd, CS_MORE)
{
        long total;
        db_expr_t value;

        if (!have_addr)
                db_error("no values to sum\n");

        total = addr;
        while (db_expression(&value))
                total += value;
        db_skip_to_eol();
        db_printf("Total is %lu\n", total);
}
```

Listing 3: Source for the "**sum**" command

```
db> sum 1
Total is 1
db> sum 1 2 3
Total is 6
db> sum 9 * 3 4
Total is 31
```

Listing 4: Sample output for the "**sum**" command

Listing 5 contains the source to a "**show softc**" command. This command accepts the name of a device as a single command line argument. If the device is found, the command prints out the value of the pointer to the device's **softc** structure. This structure contains the per-device information maintained by the device's driver. This command uses the **CS_OWN** flag to request full control of command line parsing. It uses **db_read_token()** to fetch the device name from the command line. If a valid device name is given, a **tIDENT** token will be returned with the device name saved in **db_tok_string**. Listing 6 shows some sample output for this command.

```
DB_SHOW_COMMAND_FLAGS(softc, db_show_softc_cmd, CS_OWN)
{
        device_t dev;
        int token;

        token = db_read_token();
        if (token != tIDENT)
                db_error("Missing or invalid device name");
```

```
        dev = device_lookup_by_name(db_tok_string);
        db_skip_to_eol();
        if (dev == NULL)
                db_error("device not found\n");
        db_printf("%p\n", device_get_softc(dev));
}
```

Listing 5: Source for the "`show softc`" command

```
db> show softc 4
Missing or invalid device name
db> show softc foo0
device not found
db> show softc pci0
0xffff800039380f0
```

Listing 6: Sample output for the "`show softc`" command

Custom Command Tables

A DDB command table contains a list of commands. Additional tables can be defined by a special command in an existing table. This permits building a tree of command tables. Commands that define new tables do not specify a function to use as their command handler. Instead, tables must define and initialize a variable of type struct `db_command_table` which will contain a linked-list of commands belonging to the table. A pointer to this table is associated with the command entry in the parent table. This variable should be named using the pattern `db_<name>_table`. At the time of writing, there are not nicely abstracted macros similar to `DB_COMMAND` which permit defining new tables. Instead, new tables must be defined using an "internal" macro `_DB_SET`. Commands belonging to this table must either be defined by the "internal" macro `_DB_FUNC` or by defining a new helper macro similar to `DB_SHOW_COMMAND` which wraps `_DB_FUNC.`

Listing 7 contains the source for a "demo" table along with two commands belonging to this table. The listing starts by defining a `db_demo_`table variable to contain the list of DDB commands belonging to the new table. The `_DB_SET` invocation adds the "demo" command to the top-level table similar to `DB_COMMAND`. Note that the third argument to `_DB_SET` (which normally contains a pointer to the function handler) is `NULL`, but that the last argument to `_DB_SET` contains a pointer to the new table. The rest of the listing defines two simple commands belonging to this new table. The second and third arguments to `_DB_FUNC` are similar to the two arguments given to `DB_COMMAND`. The fourth argument identifies the parent table the new command belongs to. The fifth argument contains flags such as `CS_MORE` or `CS_OWN`, and the final argument should be `NULL`. The first argument to both `_DB_SET` and `_DB_FUNC` should be the name of the parent table with a leading underscore and any spaces replaced by underscores. If the parent table is the main table, use "`_cmd`". Listing 8 shows sample output for these commands.

```
/* Holds list of "demo *" commands. */
static struct db_command_table db_demo_table = LIST_HEAD_INITIALIZER(db_demo_table);
```

```
/* Defines a "demo" top-level command. */
_DB_SET(_cmd, demo, NULL, db_cmd_table, 0, &db_demo_table);

_DB_FUNC(_demo, one, db_demo_one_cmd, db_demo_table, 0, NULL)
{
      db_printf("one\n");
}

_DB_FUNC(_demo, two, db_demo_two_cmd, db_demo_table, 0, NULL)
{
      db_printf("two\n");
}
```

Listing 7: Source for the "**demo**" table commands

```
db> demo
Subcommand required; available subcommands:
one                two
db> demo one
one
db> demo two
two
```

Listing 8: Sample output for the "**demo**" table commands

Pager-Aware Command

Our last sample command provides an example of honoring DDB's output pager. Most pager operations such as continuing for a page or for a single line are handled internally by the pager implementation in **db_printf()**. However, if the user requests that the pager stop, the global variable **db_pager_quit** is set to a non-zero value as noted earlier. Commands which generate output in a loop should check this variable and abort any loops if it is set. Listing 9 contains an abbreviated sample command which checks **db_pager_quit**. The command is an implementation of the Internet "**chargen**" service. It generates lines of output to the screen in a continuous loop until the user terminates the loop by requesting an exit via the pager. The main takeaway from this listing are the last two lines of the main loop which break out of the loop if **db_pager_quit** is set.

```
DB_COMMAND(chargen, db_chargen_cmd)
{
      char *rs;
      int len;

      for (rs = ring;;) {
            …
            db_printf("\n");
            if (db_pager_quit)
```

```
                Break;
        }
}
```

Listing 8: Abbreviated source for the "`chargen`" command

Conclusion

DDB provides a fairly simple framework for adding new commands. New commands can even be added post-boot by loading kernel modules containing new commands. There are many examples of custom commands in FreeBSD's source tree which can also be used as a reference when developing new commands. These can be found by searching for `DB.*_COMMAND` or `db_printf`. In addition, a kernel module containing all of the commands from this article can be found at https://github.com/bsdjhb/ddb_commands_demo.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Concord, California with his wife, Kimberly, and three children: Janelle, Evan, and Bella.