Writing Custom Commands in FreeBSD’s DDB Kernel Debugger

John Baldwin
EuroBSDCon
September 18, 2022
Overview

• Introduction to DDB
• DDB Execution Context
• Simple Commands
• Commands with Custom Syntax
• Custom Command Tables
What is DDB?

• Interactive kernel debugger
  • Runs on system console
  • Interrupts system execution
• Developed in Mach and ported to 386BSD
• Provides run control (stepping, breakpoints, watchpoints)
• Simplistic memory display
• Simple way to inspect system after a panic
• Supports custom commands
  • Can be defined in modules
DDB Execution Context

• Kernel context with special rules
• No blocking or sleeping
• Faults while a command is running cause the command to be aborted
  • Return to main loop via longjmp()
• Lower-level console
DDB Command Guidelines

• Commands should avoid side effects
• Commands should not use locks
  • Try lock if you must, but those can still leak on fault
• Avoid complicated APIs
• Custom commands generally are pretty printers
• Use DDB API for output
DDB Console Output

• `db_printf()`
  • Use this instead of `printf()`
  • Direct console driver access without syslog
  • Pager support

• `db_pager_quit`
  • Break out of loops generating console output if this is set
Command Functions

• General command syntax (see ddb(4)):
  • command[/modifier] [addr][,count]

• Debugger parses command line and invokes per-command function
  • void fn(db_expr_t addr, bool have_addr, db_expr_t count, char *modif)
    • addr holds an address to operate on
    • have_addr is true if addr was explicit
    • modifier is the optional modifier field (empty string if not present)
    • count is optional count field or -1
Helper Macros

- Macros define linker set entry in command table and start of function definition and are followed by function body.
- `DB_COMMAND(foo, db_foo_cmd)` defines the “foo” command implemented by a C function named `db_foo_cmd`.
- `DB_SHOW_COMMAND(bar, db_show_bar_cmd)`
- `DB_SHOW_ALL_COMMAND(baz, db_show_all_baz_cmd)`
- Function name pattern of `db_<command>_cmd` is common practice but not required.
Simple Command Example

DB_COMMAND(double, db_double_cmd)
{
    if (have_addr)
        db_printf(“%u\n”, (u_int)addr * 2);
    else
        db_printf(“no address\n”);
}
Commands with Custom Syntax

• Two flags are available to control command line parsing
  • CS_MORE – command accepts more than one address
  • CS_OWN – command does all command line parsing

• Flags are passed to DB_*COMMAND_FLAGS() macros

• After parsing, commands must call db_skip_to_eol() to discard remaining command line tokens before returning
Parser Functions

- **int db_expression(db_expr_t *expr)**
  - Parses an arithmetic expression (including symbol name resolution)
  - Returns false for EOL and true if an expression was parsed
    - Prints message and aborts command via `longjmp()` for expression syntax error

- **int db_read_token()**
  - Returns `tFOO` constant defined in `<ddb/db_lex.h>`
  - `tIDENT`: string saved in `db_tok_string`
  - `tNUMBER`: integer saved in `db_tok_number`

- **db_unread_token(int token)**
  - Put back unexpected/invalid token
Handling Errors

• `db_error(const char *msg)`
  • Prints `msg` if non-NULL, flushes lexer state, and uses `longjmp()` to abort command

• `db_flush_lex()`
  • Flushes lexer state, can be used if `longjmp()` is undesirable
Example Command using CS_MORE

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);
}
Example Command using CS_OWN

```c
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));
}
```
Custom Command Tables

• DDB command tables are a special type of command
  • db_show_table command handler
  • Variable of type struct db_command_table
    • Really a <sys/queue.h> LIST_HEAD

• Not as well abstracted (have to use “internal” macros currently)

• New tables must be a child of an existing table
  • db_cmd_table – top level commands
  • db_show_table – “show” commands
  • db_show_all_table – “show all” commands
Example Table

/* 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");
}
Example Pager-aware Command

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

    for (rs = ring;;) {
        ...
        db_printf("\n");
        if (db_pager_quit)
            break;
    }
}
Example Pager-aware Command

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

    for (rs = ring;;)
    {
        ...
        db_printf("\n");
        if (db_pager_quit)
            break;
    }
}
Conclusion

- Most custom commands pretty-print structures treating `addr` argument as a pointer
- Several examples in the tree, just grep for `DB.*_COMMAND` or `db_printf`
- Demo kernel module available at [https://github.com/bsdjhb/ddb_commands_demo](https://github.com/bsdjhb/ddb_commands_demo)
- Questions?