Marvell Design IP

Firmware Command Module

Software Development Kit

Revision 1.1

FOR CUSTOMER USE ONLY

Doc. No. NDCN-PA101226
February 24, 2012
CONFIDENTIAL
Document Classification: Restricted Distribution

Document Conventions

	**Note:** Provides related information or information of special importance.
 Caution	**Caution:** Indicates potential damage to hardware or software, or loss of data.
	**Warning:** Indicates a risk of personal injury.

For more information, visit our website at: www.marvell.com

Table of Contents

List of Tables

1 Overview

1.1 Summary

The command module (cmd) is used by command creators to register commands into a hierarchy, perform actions, and produce output and/or results. Users of commands may issue the registered command to perform the specified action or to get the specified results. Historically, this interface has been over a serial “debug” console but it may also be accessed through other I/O connections such as USB or networks. Commands may also be executed internally without the need for any I/O. Note that cmd is now distinct and separate from “debug.”

1.2 What is Cmd?

1. An enhancement of Debug's `dbg_register_cmd()` system.
2. An enhancement/replacement of ACL commands and parser

1.3 Enhancement of Debug's dbg_register_cmd()

1. Commands are registered into hierarchy.
2. Commands are registered with searchable/usable meta-data (usage, descriptions, and notes).
3. Commands may be handled by “plug-able” backend script engines (such as TCL).
   - 2 script engines are provided (TCL, and a simple, minimal engine).
4. Commands may produce both output (like debug) and results (debug does not allow this).
   - Output is similar to that of commands registered via `dbg_register_cmd()` except output goes to the proper I/O not necessarily to the debug console.
   - Results may be used by the script engine to nest commands or “pipe” commands together.
     - Set x [expr {[servo getpos] × 2}].
     - `servo getpos` is a command that produces results (presumably a number).
5. Isolates the command aspect of debug into its own module. Cmd is for command hooks implemented as C routines. Other “features” of debug remain in Debug.

1.4 Modification, Enhancement Replacement of ACL Commands

1. Text based, human readable – Generally do not require linkage with USBSend (or other).
2. Easily available over consoles (serial, network, etc.).
3. Re-entrant parsing (that is, multi instance capable).
   - There may be implications where implementers may have assumed single context.

1.5 Cmd Is Not Restricted to Any, Single Script Engine (for example, TCL)

1. It abstracts the use of a script engine so that clients do not need to change.
2. It can use the primitive debug style engine (no variables, math, procedures, flow control, etc.).

3. It can use TCL 6.7.
4. It could be extended with backend glue to other script engines (for example, jimTCL, Python, Lua, etc.)
5. There is only one backend at a time right now – System wide.
   - Clients of the proc_api (cmd implementers) do not care which backend is in use by the system.

1.6 Cmd Is Not a Debug System

1. It is intended to enhance debug and development.
2. It is intended to support product test (QA) and production (mfg).

1.7 Cmd Is Not Removed from Released Builds

1. Just like ACL, it is meant to be used into production.
2. Some commands may be conditionally included but the system is left in.
3. It is left to the implementer to decide what commands to register under what build conditions.
4. Developer commands (that are not intended to be released at manufacturing) must be wrapped with a C preprocessor macro

```
#ifdef HAVE_DEVELOPER_CMDS
// registration, usage, etc. of cmds
#endif
```

2 How to Create Commands

2.1 API

First, add the following line to the file:

```
#include “cmd_proc_api.h”
```

Next, create a command handler to perform the user’s operation. It must have the following prototype:

```
typedef int (*cmd_c_proc)(int argc, char* argv[]);
```

This prototype must be familiar as it matches exactly the `main()` prototype of all C programs.

Last, give the command a name and register it in its hierarchy. The registration command prototype is as follows:

```
int cmd_register(const char* cmdname, const char* desc, const char* usage,
const char* notes, cmd_c_proc  proc);
```

For example, if there is a hierarchy of command for “photo” and the user’s command publishes a photo's width in pixels, the user might register it like this:

```
static int photo_cmd_width_cb( int argc, char *argv[] );
void photo_cmd_init( void )
{
int cmd_res;
cmd_res = cmd_register( “photo width”, “get the width, in pixels, of a
photo”, “<path to photo filename>”, NULL, photo_cmd_width_cb );
ASSERT( CMD_OK == cmd_res );
}
```

2.2 Standard Practices

1. The `printfs` must only use the `cmd_printf()` function inside of a command. Users of the `dbg_printf()` call (and related macros) will not send output back the command channel/context unless the command is issued via the debug/serial module. Using `cmd_printf()`ensures the user’s command output is sent back to the channel that the command was issued under (for example, `USBCmd`, `ewscmd`, etc.)
2. There are 3 possible return codes from cmd handlers. They are `OK`, `ERROR`, and `USAGE_ERROR`. The last is a special case of `ERROR` that has the same effects within a script except that it also causes any provided usage information to be displayed to the user. Commands that return 1 of the 2 error codes halt script execution at that point (unless the script engine allows errors to be caught and handled such as TCL's catch command).
   - If the command is going to return `CMD_ERROR`, the user performs a `cmd_printf()` indicating for what reason the command is failing.
   - If the cmd is returning `CMD_USAGE_ERROR`, it may still want to provide specific details about what sort of usage error was encountered but this may be overkill in simplistic situations.
   - When returning, `CMD_OK` and `cmd_append_result()`are provided to the caller of the function as results. These results can be used in additional script processing. A command need not return results (think of this as a C function which returns void).

3 Important Points and Differences from the Old dbg_register_cmd System

- CMD has built-in sub-command handling. Problem solved: No more massive string comparison handling, this is built-in. Sub-cmds are easy to re-order in the hierarchy without modifying handler/callback functions.
  - argv[0] always contains the entire path to the command (top-level cmds all the way down to the sub-cmd name).
  - A command can switch its hierarchy without modifying the handler code.
- CMD has built-in "help." Description, usage, and notes (cmd meta-data) are all part of the cmd registration.
  Problem solved: consistent usage, description when commands are entered incorrectly or when the user is searching for a command to use. All commands now have help.
  - Type `help <cmd>` to see command description and usage.
  - If the user types a command incorrectly or uses improper parameters, usage is automatically displayed.
- CMD has built-in "search." Currently, only command and sub-command names are searchable. By having the cmd system "aware" of the command hierarchy and all of the "strings" used to "navigate" to specific sub-command, the CMD system can provide generic searching that was not possible previously.
  Problem solved: All commands at all levels can be globally searched. If the user knows only a portion of the command the user wants to use, even if he/she do not know what subsystem it is in, the user can easily find it.
  - Type `search <string1> [<string2>]` ... to find all commands that contain string1 and optionally those that contain string2, string3, etc. anywhere in their command hierarchy.
  - Commands that include words such as "network," "networking," or "net" are all listed when "search net" is specified.
  - Multiple words may be searched for. If, for example, the user was uncertain what name was used to refer to a specific type of flash part, he/she might do a search such as "search bspi bootspi flash" and all pertinent commands would be listed in response.
- TCL (if desired) is pluggable into the backend of cmd.
  Problem solved: CMD implementers do not need to think about TCL and CMD, the user receives both when he/she implements their cmd automatically.
- "Results" available for scripting back-ends, like TCL.
  Problem solved: enhancement.
  - cmd_append_result() API is available to produce a result that may be used in a TCL script, assigned to a variable, etc.
  - An example of a command that does this: `memory sysmem`.

4 Command Tips

- Utilize the "help" command to aide command usage.
- As the user writes new commands, look at the plethora of examples to follow.
- Fill out as much meta-data as possible.
  - Keep description and usage as short as possible and be as explicit and detailed as possible in notes.
  - Usage should follow the established conventions and list each required and optional parameter in positional order (see below).
  - None of the meta-data should contain control characters (carriage returns, new lines, tabs, etc.). These text fields will be reflowed as necessary by the built-in help system.
- Use as deep of sub-cmd registration to enable automatic help/usage.
- The built-in command "search" is useful to find commands that contain a search string: `search <search string>`.
- The built-in command "help" is useful to see description and usage of a command: `help <command [sub-cmd]>`.
- Syntax nuances:
  - `<parameter>` indicates the name of a required parameter
  - `[<parameter>]` indicates the name of an optional parameter
  - `[<parameter>=<value>]` indicates the name of an optional parameter and its default value.
  - `arg1|arg2|arg3` indicates that arg1, arg2 or arg3 may be used as the argument.

5 Using Commands

5.1 Examples on the Console

```
[ EXAMPLE OF SEARCHING FOR CMDs WITH "mem" ]
CMD==> search mem
memory test|sysmem|dump|mallinfo|limit|relevel|alloc|free|ualloc ...
memory sysmem—Retrieves the amount of free heap memory available in the
system


[ EXAMPLE FORMAT OF OUTPUT FROM "help" CMD ]
CMD==> help
The user should request help for one of the commands:
  base64      cmd      dbg     echo flowtext     gpio     help hwconfig
list   logger
  memory       os     peek     poke     puts   search     slog    stack
test  threads
    uart


[ EXAMPLE OF TOP-LEVEL CMD HELP ]
CMD==> help memory
memory test|sysmem|dump|mallinfo|limit|relevel|alloc|free|ualloc ...


[ EXAMPLE OF HELP WITH SUBCMD ]
CMD==> help memory sysmem
memory sysmem—Retrieves the amount of free heap memory available in the
system


[ EXAMPLE OF HELP WITH SUBCMD + PARAMETER ]
CMD==> help memory alloc
memory alloc <megabytes>—Allocate the desired amount of memory, specified
in Megabytes
  megabytes is the amount of memory to allocate in megabytes.  For example
if the user wants 1 megabyte, pass '1' as the parameter.


[ EXAMPLE OF USAGE ERROR ]
CMD==> peek
ERROR: Usage: peek <address> <num words> [<word size> = 1]
```

```
[ ANOTHER EXAMPLE OF HELP WITH TOP-LEVEL CMD (Note, the "notes" describe
the parameters of the "usage") ]
CMD==> help peek
peek <address> <num words> [<word size> = 1] -- Examine memory
  <address> should be aligned to correct boundary. Word size can be 1, 2 or
4.
  Default is 1.
```

5.2 Tools – Examples on the Host

5.2.1 USBCmd

If the user's product has USB, the user can use a new tool to issue commands. Many developers and test teams will find this very useful.

The reason for using `cmd_printf()` and `cmd_append_result()` will become more apparent as users use this tool.

The tool resides in the following location:

<sdk>/tools/usb_tools/

There are 2 versions (Linux vs. Windows) with the source code. Linux and Windows are supported (**USBCmd.linux** and **USBCmd.exe**).

This tool can issue cmds over the USB port from your host PC in the following manner:

[A command with results only]

```
./USBCmd.linux -c memory sysmem
96466716
[ A command with results and output ]
 ./USBCmd.linux -c test run oid
TEST BEGIN...
Run test case oid...
area = 254 (254) value = 253 (253) module = 255 (255) type = 252 (252)
Testing all OID combinations area = 4 (4) value = 14 (14) module = 0 (0)
type = 0 (0) area = 6 (6) value = 0 (0) module = 0 (0) type = 0 (0) OID
Memory Total Size In Bytes: 128 area = 6 (6) value = 2 (2) module = 0 (0)
type = 0 (0) OID Memory Left In Bytes: 91 Passed.
TEST PASSED.
1
[ Same command, hide output ]
./USBCmd.linux --nooutput -c test run oid
1
Use ./USBCmd -h to get usage.
```

The source code is available so it is easy to port the USBCmd source code and run commands via the user's favorite scripting engine: Python, Perl, Bash, etc.

5.2.2 sendfile_to_device/getfile_from_device

In the same **usb_tools** folder, there are 2 applications:

- `getfile_from_device`
- `sendfile_to_device`

Similar to USBCmd, there are both Linux and Windows versions as indicated by the extension.

These applications utilize the CMD system to get and send files to the target. This feature requires an Unified File System (UFS) and some UFS backend (FAT32 file system, NAND, etc.)

Usage:

```
 ./getfile_from_device.linux
usage: ./getfile_from_device.linux <devicefile> <hostfile>
./sendfile_to_device.linux
usage: ./sendfile_to_device.linux <hostfile> <devicefile>
```

A Revision History

Table 1: Revision History

Document Number	Revision	Description	Date
NDCN-PA101226	1.0	Initial release	10/3/2011
	1.1	Updated information related to developer commands, dbg_printf() output, and command tool location	2/23/2012

Marvell Semiconductor, Inc.
5488 Marvell Lane
Santa Clara, CA 95054, USA

Tel: 1.408.222.2500
Fax: 1.408.752.9028

www.marvell.com

Marvell. Moving Forward Faster