# mcu-uuid-console

**Simon Arlott**

**Dec 04, 2022**

# CONTENTS

CHAPTER

# ONE

# DESCRIPTION

Microcontroller console shell library

CHAPTER

TWO

# PURPOSE

Provides a framework for creating a console shell with commands. The container of commands (`uuid::console::Commands`) can be shared across multiple shell instances. Thread-safe (for log messages only) on the ESP32.

CHAPTER

# THREE

# DEPENDENCIES

- mcu-uuid-common
- mcu-uuid-log

Refer to the `library.json` file for more details.

CHAPTER

FOUR

# CONTENTS

## 4.1 Features

A static loop function consolidates the execution of all active shells.

### 4.1.1 Flexible command definition

Shells support a context stack so that multiple layers of commands can be implemented, and flags to support multiple access levels.

Commands can be composed of multiple words and have a fixed list of required/optional arguments per command.

### 4.1.2 Command line prompt

Text encoded using the US-ASCII character set can be entered with basic basic line editing (backspace, delete word, delete line). All standard line endings (CR, CRLF and LF) are supported.

Both command names and arguments (where the command returns a list of potential arguments) can be tab completed and spaces can be escaped using backslashes or quotes.

Password entry (without echo) can be performed using a callback function process.

Blocking commands can be performed using a callback function to execute asynchronously and can read from the underlying input stream.

The `Shell` class is customisable to allow the prompt, banner, hostname and context text to be replaced. The `^D` (end of transmission) character can be made to execute implied commands (e.g. `logout`).

### 4.1.3 Logging

Acts as a log handler in order to output log messages without interrupting the entry of commands at a prompt.

### 4.1.4 Session

An idle timeout can be configured to automatically stop the shell if it is waiting at a prompt for too long.

## 4.2 Usage

```
#include <uuid/console.h>
```

Create a `std::shared_ptr<uuid::console::Commands>` and populate it with the commands to be available on the shell.

Create a `std::shared_ptr<uuid::console::Shell>` referencing the `Serial` stream and the commands list. Call `start()` on the instance and then `uuid::console::Shell::loop_all()` regularly. (The static set of all shells will retain a copy of the `shared_ptr` until the shell is stopped.)

### 4.2.1 Example (Digital I/O)

```
#include <Arduino.h>

#include <memory>
#include <string>
#include <vector>

#include <uuid/common.h>
#include <uuid/console.h>

using uuid::read_flash_string;
using uuid::flash_string_vector;
using uuid::console::Commands;
using uuid::console::Shell;

void setup() {
        std::shared_ptr<Commands> commands = std::make_shared<Commands>();

        commands->add_command(flash_string_vector{F("pinMode")},
                flash_string_vector{F("<pin>"), F("<mode>")},

                [] (Shell &shell, const std::vector<std::string> &arguments) {
                        uint8_t pin = String(arguments[0].c_str()).toInt();
                        uint8_t mode;

                        if (arguments[1] == read_flash_string(F("INPUT"))) {
                                mode = INPUT;
                        } else if (arguments[1] == read_flash_string(F("OUTPUT"))) {
                                mode = OUTPUT;
                        } else if (arguments[1] == read_flash_string(F("INPUT_PULLUP")))
↪{
                                mode = INPUT_PULLUP;
                        } else {
                                shell.println(F("Invalid mode"));
                                return;
```

(continues on next page)

(continued from previous page)

```
                }

                pinMode(pin, mode);
                shell.printfln(F("Configured pin %u to mode %s"),
                                pin, arguments[1].c_str());
        },

        [] (Shell &shell, const std::vector<std::string> &current_arguments,
                        const std::string &next_argument)
                        -> const std::vector<std::string> {
                if (current_arguments.size() == 1) {
                        /* The first argument has been provided, so return
                         * completion values for the second argument.
                         */
                        return {
                                read_flash_string(F("INPUT")),
                                read_flash_string(F("OUTPUT")),
                                read_flash_string(F("INPUT_PULLUP"))
                        };
                } else {
                        return {};
                }
        }
);

commands->add_command(flash_string_vector{F("digitalRead")},
        flash_string_vector{F("<pin>")},

        [] (Shell &shell, const std::vector<std::string> &arguments) {
                uint8_t pin = String(arguments[0].c_str()).toInt();
                auto value = digitalRead(pin);

                shell.printfln(F("Read value from pin %u: %S"),
                                pin, value == HIGH ? F("HIGH") : F("LOW"));
        }
);

commands->add_command(flash_string_vector{F("digitalWrite")},
        flash_string_vector{F("<pin>"), F("<value>")},

        [] (Shell &shell, const std::vector<std::string> &arguments) {
                uint8_t pin = String(arguments[0].c_str()).toInt();
                uint8_t value;

                if (arguments[1] == read_flash_string(F("HIGH"))) {
                        value = HIGH;
                } else if (arguments[1] == read_flash_string(F("LOW"))) {
                        value = LOW;
                } else {
                        shell.println(F("Invalid value"));
                        return;
                }
```

(continues on next page)

(continued from previous page)

```
                        digitalWrite(pin, value);
                        shell.printfln(F("Wrote %s value to pin %u"),
                                        arguments[1].c_str(), pin);
                },

                [] (Shell &shell, const std::vector<std::string> &current_arguments,
                                const std::string &next_argument)
                                -> const std::vector<std::string> {
                        if (current_arguments.size() == 1) {
                                /* The first argument has been provided, so return
                                 * completion values for the second argument.
                                 */
                                return {
                                        read_flash_string(F("HIGH")),
                                        read_flash_string(F("LOW"))
                                };
                        } else {
                                return {};
                        }
                }
        );

        commands->add_command(flash_string_vector{F("help")},
                [] (Shell &shell, const std::vector<std::string> &arguments) {
                        shell.print_all_available_commands();
                }
        );

        Serial.begin(115200);

        std::shared_ptr<Shell> shell;
        shell = std::make_shared<uuid::console::Shell>(Serial, commands);
        shell->start();
}

void loop() {
        uuid::loop();
        Shell::loop_all();
        yield();
}
```

#### Output

```
$ help
pinMode <pin> <mode>
digitalRead <pin>
digitalWrite <pin> <value>
help
$ digitalRead 2
Read value from pin 2: LOW
$ digitalRead 3
Read value from pin 3: HIGH
$ pinMode 4 OUTPUT
Configured pin 4 to mode OUTPUT
$ digitalWrite 4 HIGH
Wrote HIGH value to pin 4
$ pinMode 5 OUTPUT
Configured pin 5 to mode OUTPUT
$ digitalWrite 5 LOW
Wrote LOW value to pin 5
$
```

### 4.2.2 Example (WiFi network scan)

```
#ifdef ARDUINO_ARCH_ESP8266
# include <ESP8266WiFi.h>
#else
# include <WiFi.h>
#endif

#include <memory>
#include <string>
#include <vector>

#include <uuid/common.h>
#include <uuid/console.h>

using uuid::read_flash_string;
using uuid::flash_string_vector;
using uuid::console::Commands;
using uuid::console::Shell;

void setup() {
        std::shared_ptr<Commands> commands = std::make_shared<Commands>();

        commands->add_command(flash_string_vector{F("wifi"), F("scan")},
                [] (Shell &shell, const std::vector<std::string> &arguments) {

                        int8_t ret = WiFi.scanNetworks(true);
                        if (ret == WIFI_SCAN_RUNNING) {
                                shell.println(F("Scanning for WiFi networks..."));
```

(continues on next page)

(continued from previous page)

```
                                /* This function will be called repeatedly on every
                                 * loop until it returns true. It can be used to
                                 * wait for the outcome of asynchronous operations
                                 * without blocking execution of the main loop.
                                 */
                                shell.block_with([] (Shell &shell, bool stop) -> bool {
                                        int8_t ret = WiFi.scanComplete();

                                        if (ret == WIFI_SCAN_RUNNING) {
                                                /* Keep running until the scan completes
                                                 * or the shell is stopped.
                                                 */
                                                return stop;
                                        } else if (ret == WIFI_SCAN_FAILED || ret < 0) {
                                                shell.println(F("WiFi scan failed"));
                                                return true; /* stop running */
                                        } else {
                                                shell.printfln(F("Found %u networks"),␣
↪ret);
                                                shell.println();

                                                for (uint8_t i = 0; i < (uint8_t)ret;␣
↪i++) {
                                                        shell.printfln(F("%s (%d dBm)"),
                                                                       WiFi.SSID(i).c_
↪str(),
                                                                       WiFi.RSSI(i));
                                                }

                                                WiFi.scanDelete();
                                                return true; /* stop running */
                                        }
                                });
                        } else {
                                shell.println(F("WiFi scan failed"));
                        }
                }
        );

        Serial.begin(115200);

        std::shared_ptr<Shell> shell;
        shell = std::make_shared<uuid::console::Shell>(Serial, commands);
        shell->start();
}

void loop() {
        uuid::loop();
        Shell::loop_all();
        yield();
}
```

### Output

```
$ wifi scan
Scanning for WiFi networks...
Found 3 networks

Free Public WiFi (-87 dBm)
Hacklab (-30 dBm)
ALL YOUR BASE ARE BELONG TO US (-44 dBm)
$
```

CHAPTER

FIVE

# RESOURCES

## 5.1 Change log

### 5.1.1 Unreleased

### 5.1.2 3.0.0 – 2022-12-04

Remove `StreamConsole` and the need for virtual inheritance.

#### Changed

- Remove `StreamConsole` and the need for virtual inheritance to combine a custom `Shell` with `StreamConsole`. All of the shells operate on streams so it's an unnecessary complexity.

  This allows a `static_cast` to be used on the `shell` parameter to command functions instead of forcing the use of `dynamic_cast`. Applications can be compiled with `-fno-rtti` and still usefully extend the `Shell` class.

### 5.1.3 2.0.1 – 2022-12-03

Fix list of available commands.

#### Fixed

- Iterators over available commands may return an unavailable command if it is at the beginning of the commands in the current context.

### 5.1.4 2.0.0 – 2022-11-26

Provide the next argument to the argument completion function.

#### Changed

- Provide the next argument (the one being completed) to the argument completion function. This makes it possible to do filesystem lookups based on what has been provided instead of having to traverse the entire filesystem.

### 5.1.5 1.0.1 – 2022-11-06

Fix potential deadlock when outputting log messages.

#### Changed

- Use `PSTR_ALIGN` for flash strings.

#### Fixed

- Deadlock if a message is logged from `display_prompt()` and the shell is a log handler for that message.

### 5.1.6 1.0.0 – 2022-10-29

Be thread-safe (for log messages) where possible.

#### Added

- Indicate whether this version of the library is thread-safe or not (`UUID_CONSOLE_THREAD_SAFE` and `uuid::console::thread_safe`).

#### Changed

- Make the library thread-safe (for log messages only) when supported by the platform.

### 5.1.7 0.9.0 – 2022-07-12

Support for iterating over available commands.

#### Added

- Support for iterating over all available commands in a shell.

### 5.1.8 0.8.0 – 2022-02-19

Support for command flags that must be absent.

#### Added

- Support for commands that are only available when specific flags are absent. This makes it easier to have user and admin versions of commands that would otherwise conflict with each other.

### 5.1.9 0.7.6 – 2022-02-19

Tab completion bug fixes.

#### Changed

- Tab completion now shows an empty line as a suggestion when the current command is an exact match but it also has longer partial matches. Suggested commands will always be output and be less eager to immediately skip to a single longer command.

#### Fixed

- Tab completion now takes into account additional matching commands with longer names when there is a single command with a shorter name between them (`a` will no longer complete to `a b` if `a c d` is also present).
- Always order suggested commands by insertion order instead of the length of its name.

### 5.1.10 0.7.5 – 2021-04-18

Upgrade to PlatformIO 5.

#### Changed

- Use PlatformIO 5 dependency specification.

### 5.1.11 0.7.4 – 2021-01-17

Fixes for uncontrolled ordering of static object lifetimes.

#### Changed

- Use less memory by not using empty or single character literal strings.
- Don't unregister log handler explicitly in the destructor, this is now handled by the logging library.

#### Fixed

- Make registration of shells safe during static initialization.
- Make use of the built-in logger instance safe during static initialization.

### 5.1.12 0.7.3 – 2019-09-22

Bug fixes.

#### Fixed

- Output an error message if the shell has no commands.
- Avoid running a shell loop if it has already stopped.

### 5.1.13 0.7.2 – 2019-09-17

Logout improvements on remote shells.

#### Changed

- Automatically stop the shell on end of transmission character if an idle timeout is set.

### 5.1.14 0.7.1 – 2019-09-16

Tab completion bug fixes.

#### Fixed

- Problem with tab completion when the partial match commands have arguments and the longest common prefix is returned.
- Incorrect partial tab completion matches when the command line has a trailing space.

### 5.1.15 0.7.0 – 2019-09-15

Add idle timeout.

#### Added

- Configurable idle timeout.

#### Fixed

- Use move constructors on rvalues.

### 5.1.16 0.6.0 – 2019-09-03

Bug fixes and additional configuration options.

#### Changed

- Remove `get_` and `set_` from function names.
- Move maximum command line length and maximum log messages to getter/setter functions.

#### Fixed

- Remove messages from the log queue before processing them.
- Problems with tab completion of commands and arguments when there are multiple exact matches or there is a single shortest partial match with multiple longer partial matches.

### 5.1.17 0.5.0 – 2019-08-31

Fix escaping of command line argument help text.

#### Changed

- Avoid copying command line arguments when executing commands.
- Executed commands can now modify their arguments.
- Use `std::vector` instead of `std::list` for most containers to reduce memory usage.

#### Fixed

- Don't escape command line argument help text.

### 5.1.18 0.4.0 – 2019-08-30

Support for printing all currently available commands.

#### Added

- Support for printing all currently available commands.

#### Changed

- Move trailing space handling into instances of the `CommandLine` class.

#### Fixed

- Support tab completion of empty arguments.

### 5.1.19 0.3.0 – 2019-08-28

Support for empty arguments using quotes.

#### Added

- Support for empty arguments using quotes (`""` or `''`).
- Move command line parsing/formatting to a `CommandLine` utility class.

### 5.1.20 0.2.0 – 2019-08-27

Support blocking commands that execute asynchronously.

#### Added

- Support for blocking commands that execute asynchronously and can read from the underlying input stream.
- Example serial console for ESP8266/ESP32 WiFi features.

#### Changed

- The default context is now optional when creating a `Shell` (it defaults to 0).
- Commands can now be created with a default context and flags of 0.

#### Fixed

- Don't set private member `prompt_displayed_` from virtual function `erase_current_line()`.
- Don't try to write empty strings to the shell output.
- Workaround incorrect definition of `FPSTR()` on ESP32 (#1371).
- Create a copy of `va_list` when outputting with a format string so that it can be used twice.

### 5.1.21 0.1.0 – 2019-08-23

Initial development release.

#### Added

- Reusable container of multi-word commands that can be executed, with a fixed list of required/optional arguments per command.
- Shell context to support multiple layers of commands.
- Shell flags to support multiple access levels.
- Minimal line editing support (backspace, delete word, delete line).
- Text input in the US-ASCII character set.
- Support for entry of spaces in arguments using backslashes or quotes.
- Support for CR, CRLF and LF line endings on input.
- Tab completion for recognised commands/arguments.
- Logging handler to output log messages without interrupting the entry of commands at a prompt.
- Password entry prompt.
- Customisable `Shell` class:
  - Replaceable prompt text.
  - Optional banner, hostname and context text.
  - Support for the `^D` (end of transmission) character with implied command execution (e.g. `logout`).
- Support for `Stream` (`Serial`) consoles.
- Loop function to consolidate the execution of all active shells.
- Example serial console for Arduino Digital I/O features.