Developer Guide

Sven van der Meer

Table of Contents

1. API

1.1. Commands

1.1.1. GetCommandID

Returns the identifier (name) of a command for a given input string.

Arguments	Return (print)
$1: the input string to test for a command identifier	Success: long form of the command Error: empty string

```
id=$(GetCommandID "string")
```

1.2. Config

1.2.1. WriteRuntimeConfig

Writes runtime configuration file. The file name is taken from `CONFIG_MAP["FW_L1_CONFIG"]`. The file is removed, and then all configuration maps are written into a new file.

Arguments	Return
none	none

```
WriteRuntimeConfig
```

The written maps are:

- General Configurations
  - CONFIG_MAP - configuration
  - CONFIG_SRC - setting source
  - FW_PATH_MAP - paths for the framework
  - APP_PATH_MAP - paths for an application
  - CHAR_MAP - the map of characters (UTF-8)
  - COLORS - the map of ANSI color codes
  - EFFECTS - the map of ANSI text effects
- Options

    - DMAP_OPT_ORIGIN - options and their declaration origin
    - DMAP_OPT_SHORT - short option names
    - DMAP_OPT_ARG - option arguments
- Exit Status
    - DMAP_ES - map of exit status declarations
    - DMAP_ES_PROBLEM - exit status problem identifiers (internal, external)
- Commands
    - DMAP_CMD - command declarations
    - DMAP_CMD_SHORT - command short names
    - DMAP_CMD_ARG - command arguments
- Parameters
    - DMAP_PARAM_ORIGIN - parameter origin (framework or application)
    - DMAP_PARAM_DECL - parameter declaration file
    - DMAP_PARAM_DEFVAL - parameter default value
    - DMAP_PARAM_IS - parameter *is* relationship, e.g. *is* a directory
- Dependencies
    - DMAP_DEP_ORIGIN - dependency origin (framework or application)
    - DMAP_DEP_DECL - dependency declaration file
    - DMAP_DEP_REQ_DEP - dependency requires another dependency
    - DMAP_DEP_CMD - dependency test command
- Dependency Runtime
    - RTMAP_DEP_STATUS - test status
- Tasks
    - DMAP_TASK_ORIGIN - task origin (framework or application)
    - DMAP_TASK_DECL - task declaration file
    - DMAP_TASK_SHORT - short task name
    - DMAP_TASK_EXEC - task script location and name
    - DMAP_TASK_MODES - task modes
- Task Requirements
    - DMAP_TASK_REQ_PARAM_MAN - required mandatory parameters
    - DMAP_TASK_REQ_PARAM_OPT - required optional parameters
    - DMAP_TASK_REQ_DEP_MAN - required mandatory dependencies
    - DMAP_TASK_REQ_DEP_OPT - required optional dependencies
    - DMAP_TASK_REQ_TASK_MAN - required other tasks, mandatory
    - DMAP_TASK_REQ_TASK_OPT - required other tasks, optional

    - DMAP_TASK_REQ_DIR_MAN - required mandatory directories
    - DMAP_TASK_REQ_DIR_OPT - required optional directories
    - DMAP_TASK_REQ_FILE_MAN - required mandatory files
    - DMAP_TASK_REQ_FILE_OPT - required optional files
- Tasks Runtime
    - RTMAP_TASK_STATUS - task load status
    - RTMAP_TASK_LOADED - loaded tasks
    - RTMAP_TASK_UNLOADED - unloaded tasks
- Scenarios
    - DMAP_SCN_ORIGIN - scenario origin (framework, application, or path)
    - DMAP_SCN_DECL - scenario declaration file
    - DMAP_SCN_SHORT - short scenario name
    - DMAP_SCN_EXEC - scenario script location and name
    - DMAP_SCN_MODES - scenario modes
    - DMAP_SCN_REQ_TASK_MAN - scenario required tasks, mandatory
    - DMAP_SCN_REQ_TASK_OPT - scenario required tasks, optional
- Scenario Runtime
    - RTMAP_SCN_STATUS - load status
    - RTMAP_SCN_LOADED - loaded scenarios
    - RTMAP_SCN_UNLOADED - unloaded scenarios
- Runtime Maps
    - RTMAP_REQUESTED_DEP - requested dependencies
    - RTMAP_REQUESTED_PARAM - requested parameters
- Description Maps
    - DMAP_CMD_DESCR - commands
    - DMAP_DEP_DESCR - dependencies
    - DMAP_ES_DESCR - exit status codes
    - DMAP_OPT_DESCR - options
    - DMAP_PARAM_DESCR - parameters
    - DMAP_TASK_DESCR - tasks
    - DMAP_SCN_DESCR - scenarios

1.3. Console

1.3.1. ConsoleMessage

Prints a message to the console (standard error). The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_QUIET"]` for the loader, `CONFIG_MAP["SHELL_QUIET"]` for the shell, or `CONFIG_MAP["TASK_QUIET"]` for tasks. If the setting for quiet is *off*, it prints the message. Otherwise it does not print the message.

Arguments	Return
$1: the message	none

```
ConsoleMessage "message"
```

1.3.2. ConsoleIsMessage

Returns the message status. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_QUIET"]` for the loader, `CONFIG_MAP["SHELL_QUIET"]` for the shell, or `CONFIG_MAP["TASK_QUIET"]` for tasks.

Arguments	Return (print)
none	1 for *on*, 0 for *off*

```
if ConsoleIsMessage; then ...; else ...; fi
```

1.3.3. ConsoleIsPrompt

Returns shell-prompt status from `CONFIG_MAP["SHELL_SNP"]`. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_QUIET"]` for the loader, `CONFIG_MAP["SHELL_QUIET"]` for the shell, or `CONFIG_MAP["TASK_QUIET"]` for tasks.

Arguments	Return (print)
none	1 for *on*, 0 for *off*

```
if ConsoleIsPrompt; then ...; else ...; fi
```

1.3.4. ConsoleFatal

Prints an error message with *[Fatal]* tag if the level for *fatal* is set and increases the error counter. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_LEVEL"]`and `LOADER_ERRORS` counter for the loader, `CONFIG_MAP["SHELL_LEVEL"]` and `SHELL_ERRORS` counter for the shell, or `CONFIG_MAP["TASK_LEVEL"]` and `TASK_ERRORS` counter for tasks.

Arguments	Return (print)
$1: message prefix, e.g. script name with colon $2: the error message	none

```
ConsoleFatal " →" "fatal error message"
```

1.3.5. ConsoleError

Prints an error message with *[Error]* tag if the level for *fatal* is set and increases the error counter. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_LEVEL"]` and `LOADER_ERRORS` counter for the loader, `CONFIG_MAP["SHELL_LEVEL"]` and `SHELL_ERRORS` counter for the shell, or `CONFIG_MAP["TASK_LEVEL"]` and `TASK_ERRORS` counter for tasks.

Arguments	Return (print)
$1: message prefix, e.g. script name with colon $2: the error message	none

```
ConsoleError " →" "error message"
```

1.3.6. ConsoleResetErrors

Resets the error counter, i.e. sets it to *0*. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which counter to reset: `LOADER_ERRORS` for the loader, `SHELL_ERRORS` for the shell, or `TASK_ERRORS` for tasks.

```
ConsoleResetErrors
```

1.3.7. ConsoleHasErrors

Returns *true* if the counter has errors (i.e. is larger than *0*) or false if it does not have errors (i.e. is *0*). The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which counter to use: `LOADER_ERRORS` for the loader, `SHELL_ERRORS` for the shell, or `TASK_ERRORS` for tasks.

Arguments	Return
none	*true* (0) if errors *false* (1) if no errors

```
if ConsoleHasErrors; then ...; ...; fi

if ConsoleHasErrors; then ...; else ...; fi
```

1.3.8. ConsoleWarnStrict

Prints a strict warning message. If the application is not in strict mode, those messages are considered warnings. Here, the message will be printed with the tag *[Warn/Strict]* and the warning counter will be increased. If the application is in strict mode, those messages are considered errors. Here, the message will be printed with the tag *[Error/Strict]* and ere, the error counter will be increased. In *ansi* print mode, *Warn* is yellow and *Error* is red. The function uses `CONFIG_MAP["RUNNING_IN"]` and `CONFIG_MAP["STRICT"]` to determine which counter to increase.

Arguments	Return
$1: message prefix, script name with colon $2: the warning/error message	none

```
ConsoleWarnStrict " →" "did not find file $FILE"
```

1.3.9. ConsoleWarn

Prints a warning message. The message will be printed with the tag *[Warn]* and the warning counter will be increased. If the application is in strict mode, those messages are considered errors. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which counter to increase: `LOADER_WARNINGS` for the loader, `SHELL_WARNINGS` for the shell, or `TASK_WARNINGS` for tasks.

Arguments	Return
$1: message prefix, script name with colon $2: the warning message	none

```
ConsoleWarn " →" "did not find file $FILE"
```

1.3.10. ConsoleResetWarnings

Resets the warning counter, i.e. sets it to *0*. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which counter to reset: `LOADER_WARNINGS` for the loader, `SHELL_WARNINGS` for the shell, or `TASK_WARNINGS` for tasks.

```
ConsoleResetWarnings
```

1.3.11. ConsoleHasWarnings

Returns *true* if the counter has warnings (i.e. is larger than *0*) or false if it does not have warnings (i.e. is *0*). The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which counter to use: `LOADER_WARNINGS` for the loader, `SHELL_WARNINGS` for the shell, or `TASK_WARNINGS` for tasks.

Arguments	Return
none	*true* (0) if warnings
	false (1) if no warnings

```
if ConsoleHasWarnings; then ...; ...; fi

if ConsoleHasWarnings; then ...; else ...; fi
```

1.3.12. ConsoleInfo

Prints an information message. The message will be printed with the tag *[Info]*.

Arguments	Return
$1: message prefix, script name with colon	none
$2: the message	

```
ConsoleInfo " →" "I am doing something now"
```

1.3.13. ConsoleDebug

Prints a debug message. The message will be printed with the prefix ` >` in bold.

Arguments	Return
$1: the message	none

```
ConsoleDebug "I am doing something now"
```

1.3.14. ConsoleTrace

Prints a trace message. The message will be printed with the prefix ` >` in italic.

Arguments	Return
$1: the message	none

```
ConsoleTrace "I am doing something now"
```

1.3.15. ConsoleIsDebug

Returns the message status. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_LEVEL"]` for the loader, `CONFIG_MAP["SHELL_LEVEL"]` for the shell, or `CONFIG_MAP["TASK_LEVEL"]` for tasks.

Arguments	Return (print)
none	1 for *on*, 0 for *off*

```
if ConsoleIsDebug; then ...; else ...; fi
```

1.3.16. ConsoleIsTrace

Returns the message status. The function uses `CONFIG_MAP["RUNNING_IN"]` to determine which setting to use: `CONFIG_MAP["LOADER_LEVEL"]` for the loader, `CONFIG_MAP["SHELL_LEVEL"]` for the shell, or `CONFIG_MAP["TASK_LEVEL"]` for tasks.

Arguments	Return (print)
none	1 for *on*, 0 for *off*

```
if ConsoleIsTrace; then ...; else ...; fi
```

1.4. Execute

1.4.1. ExecuteTask

Executes a task.

Arguments	Return
$1: full command line for the task	none

The first word from the argument is taken as the task name. This can be the task’s short or long name. The task must be loaded and available for execution.

This function will print extra information (header and footer, execution time calculations) for most tasks. The exception here are all *standard* tasks known to not need header and footers: *list-**, *describe-**, *setting*, *manual*, *statistics*, and *wait*. Also: any task execution that includes the arguments `-h` or `--help` will not see header and footer printed. For the task *wait*, an additional calculation of the actual wait time is displayed.

```
ExecuteTask "$SARG"

ExecuteTask "list-tasks -AT"
```

1.4.2. ExecuteScenario

Executes a scenario.

Arguments	Return
$1: scenario ID, short or long	none

The scenario must be loaded and available for execution.

```
ExecuteScenario build-site
```

1.5. MVN Site

1.5.1. MvnSiteFixAdoc

Fixes HTML files generated from ADOC sources by the Maven site and Asciidoctor plugins. The problems fixed are:

*add a text to the HTML title of the page (empty otherwise) * add text to the active bread crumb list item (empty otherwise)

This function will take the file name (first argument, no extension) and use `sed` to add the text from the second argument to the HTML file. For the title, it assumes that there is a prefix in the title, so it adds `x2013; $2`.

Arguments	Return
$1: file name $2: text	none

```
MvnSiteFixAdoc target/site/developer-guide/api/mvn-site "API: MVN Site"
```

1.6. Print ANSI

The functions here will print text according to the current print mode in `CONFIG_MAP["PRINT_MODE"]`:

- *ansi* - with ANSI encoded colors or effects
- *adoc* - in AsciiDoc notation
- *text* - as plain text

- *text-anon* - as annotated text

By default, the functions take the current print mode. They can also be requested to use a specific, then forced, print mode.

The printed text will not contain a line feed.

1.6.1. PrintColor

Prints text in color in *ansi* mode, just text otherwise.

Arguments	Return
$1: color $2: message $3: forced print mode	none

```
PrintColor light-green "I am available"

PrintColor yellow "some problem here"
```

Supported colors are:

- black
- red
- green
- brown
- blue
- purple
- cyan
- light-gray
- dark-gray
- light-red
- light-green
- yellow
- light-blue
- light-purple
- light-cyan

1.6.2. PrintEffect

Prints text with an effect in *ansi* mode, plain text in *text* mode, and text with some annotation in *adoc* and *text-anon* mode.

Arguments	Return
\$1: effect \$2: message \$3: forced print mode	none

```
PrintColor bold "I am available"

PrintColor reverse "Name Description"
```

Supported effects are:

- *bold* - either bold, plain text or as annotation using `*`
- *italic* - either italic, plain text or as annotation `_`
- *reverse* - either reverse, or plain text

1.7. Prompt

Collection of functions for the shell prompt.

1.7.1. PromptSfMode

Prints the application flavor and the mode in brackets.

```
PromptSfMode
```

1.8. Prompt

Collection of functions for scenarios.

1.8.1. GetScenarioID

Returns the identifier (name) of a scenario for a given input string.

Arguments	Return (print)
\$1: the input string to test for a scenario identifier	Success: long form of the scenario Error: empty string

```
id=$(GetScenarioID "string")
```

1.9. Prompt

Collection of functions for the underlying system.

1.9.1. PathToSystemPath

Converts a given path to a system-specific representation. This can be important when running on hybrid systems. In Cygwin for instance, one can execute Windows programs that do not understand UNIx path. This function can convert the paths.

The system is taken from `CONFIG_MAP["SYSTEM"]`. If no special path conversion is implemented for the system, the original path is returned.

Arguments	Return (print)
$1: path to convert	converted or original path

```
VARIABLE=$(PathToSystemPath "path")
```

Currently supported conversions are:

- Cygwin: uses `cygpath -m` for the conversion

1.10. Prompt

Collection of functions for tasks.

1.10.1. GetTaskID

Returns the identifier (name) of a task for a given input string.

Arguments	Return (print)
$1: the input string to test for a task identifier	Success: long form of the task Error: empty string

```
id=$(GetTaskID "string")
```

1.10.2. BuildTaskHelpLine

Prints a single line for the task help screen with one argument and its settings. An argument is described by:

- a short form, use `<none>` if not applicable
- a long form, use `<none>` if not applicable
- an argument, use `<none>` if not applicable, if used the string will be converted to upper case spelling
- a description in form of a short tag line

Additionally, a length value an be given. The length is used to calculate the padding between short/long/argument and description. The padding default value is *24*.

Arguments	Return
$1: short options $2: long option $3: argument $4: description $5: length	none

```
BuildTaskHelpLine h help "<none>" "print help screen and exit" 25

BuildTaskHelpLine f file "FILE" "file to open in viewer" 25
```

1.10.3. TaskGetCachedHelp

Returns a file name with cached help screen for current print-mode, none if none found.

Arguments	Return (print)
$1: task ID, long form	Success: file name Error: empty string

```
CACHED_HELP=$(TaskGetCachedHelp "my-task")
```