

# Micrium

Empowering Embedded Systems

## **μC/Shell**

**V1.03.00**

**User Manual**

[www.Micrium.com](http://www.Micrium.com)

## **Disclaimer**

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

## **Copyright notice**

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of Micrium. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2007-2010; Micrium, Weston, Florida 33327-1848, U.S.A.

## **Trademarks**

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

## **Registration**

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: [licensing@micrium.com](mailto:licensing@micrium.com)

## **Contact address**

### **Micrium**

949 Crestview Circle  
Weston, FL 33327-1848  
U.S.A.

Phone : +1 954 217 2036

FAX : +1 954 217 2037

WEB : [www.micrium.com](http://www.micrium.com)

Email : [support@micrium.com](mailto:support@micrium.com)

## Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
V1.00	2007/08/30	SR	Released first version.
V1.01	2010/04/02	ITJ	
V1.02.00	2011/07/12	FBJ	
V1.03.00	2011/10/04	FBJ	

# Table Of Contents

<b>Table of Contents</b> .....	<b>iv</b>
<b>1 Introduction</b> .....	<b>1</b>
1.01 Required modules .....	1
<b>2 Directories and Files</b> .....	<b>2</b>
<b>3 Using <math>\mu</math>C/Shell</b> .....	<b>4</b>
3.01 Commands, callbacks, and data types .....	5
3.02 $\mu$ C/Shell startup code .....	6
3.03 $\mu$ C/Shell example use .....	9
3.04 $\mu$ C/Shell module configuration .....	10
3.05 $\mu$ C/Shell internal details .....	11
<b>4 <math>\mu</math>C/Shell API Reference</b> .....	<b>13</b>
<b>5 <math>\mu</math>C/Shell Callback Functions Reference</b> .....	<b>20</b>
<b>A <math>\mu</math>C/Shell Licensing Policy</b> .....	<b>23</b>
<b>B References</b> .....	<b>24</b>

# Chapter 1

## Introduction

**μC/Shell** is a stand-alone module allowing a string containing a command and its argument to be parsed and executed. Though not directly tied with **μC/TELNETs**, it has been developed in order to provide a shell for this product. However, it could be used with other applications.

This document describes how to configure and use the **μC/Shell** module in a **μC/OS-II** environment.

### 1.01 Required modules

The current version of **μC/Shell** requires the **μC/LIB** module. Please refer to the release notes document for version information.

## Directories and Files

The code and documentation of the **μC/Shell** module are organized in a directory structure according to “AN-2002, μC/OS-II Directory Structure”. Specifically, the files may be found in the following directories:

**\Micrium\Software\uC-Shell**

This is the main directory for **μC/Shell**.

**\Micrium\Software\uC-Shell\Doc**

This directory contains the **μC/Shell** documentation files, including this one.

**\Micrium\Software\uC-Shell\Cfg\Template**

This directory contains a template of **μC/Shell** configuration.

**\Micrium\Software\uC-Shell\Source**

This directory contains the **μC/Shell** source code. This protocol is implemented in two OS independent files:

shell.c  
shell.h

The location of the directory containing the example sample code application is dependent of the evaluation board and contains those files:

`app.c`

Application code.

`shell_cfg.h`

Example configuration file.

`includes.h`

Master include file used by the application.

`net_cfg.h`

**μC/TCP-IP** configuration file.

`os_cfg.h`

**μC/OS-II** configuration file.

`cpu_cfg.h`

**μC/CPU** configuration file.

## Using $\mu$ C/Shell

Before going into an example of  $\mu$ C/Shell usage, a few design concepts have to be explained. Since  $\mu$ C/Shell is not associated with any particular product, modules in need of a shell facility (such as  $\mu$ C/TELNETs) interact with it by means of an application callback function. This way, those modules are able to use or not to use the shell in a totally transparent manner.

From the caller point of view, once the commands have been developed and the initialization performed, all that is needed to do is a call the main  $\mu$ C/Shell execution function:

```
CPU_INT16S  Shell_Exec (CPU_CHAR      *in,  
                      SHELL_OUT_FNCT  out_fnct,  
                      SHELL_CMD_PARAM *pcmd_param,  
                      SHELL_ERR       *perr);
```

This function parses the 'in' parameter, a NUL terminated string containing a complete command line (command name, followed by possible arguments being separated by spaces), just like this one:

```
App_Test -a -b -c readme.txt
```

Once parsed, that is once the command name and its arguments have been extracted,  $\mu$ C/Shell looks into its command tables for a command match (in this case App\_Test is the name of the command), and invokes it.

Note that the Shell\_Exec() function also has a 'out\_fnct' argument, which is a pointer to a callback that handles the details of responding to the requester. In other words, if called by  $\mu$ C/TELNETs, then  $\mu$ C/TELNETs has to provide the details of the response; if called by a UART, the UART should handle the response. Finally, the 'pcmd\_param' is a pointer to a structure containing additional parameters for the command to use.

For more details on this function, please proceed with the next section.

## 3.01 Commands, callbacks, and data types

**µC/Shell** commands (i.e. commands placed in a ‘command table’) all have this prototype:

```
CPU_INT16S  My_Cmd (CPU_INT16U      argc,
                  CPU_CHAR         *argv[],
                  SHELL_OUT_FNCT   out_fnct,
                  SHELL_CMD_PARAM  *pcmd_param);
```

where ‘argc’ is a count of the arguments supplied and ‘argv’, an array of pointers to the strings which are those arguments. As for the return value, it is command specific, and will be used as the return value for the Shell\_Exec() function. However, in case of an error, SHELL\_EXEC\_ERR should be returned.

Commands are also defined by the SHELL\_CMD\_FNCT data type:

```
typedef CPU_INT16S  (*SHELL_CMD_FNCT) (CPU_INT16U      ,
                                       CPU_CHAR         **,
                                       SHELL_OUT_FNCT   ,
                                       SHELL_CMD_PARAM  *);
```

As mentioned in the preceding section, each command is responsible for responding to its requester, and this is done with the help of the last parameter: the pointer to the output function. This function has the following prototype:

```
CPU_INT16S  My_Out_Funct (CPU_CHAR      *pbuf,
                          CPU_INT16U   buf_len,
                          void          *popt);
```

where ‘pbuf’ is a pointer to a response buffer having a length of ‘buf\_len’. The third parameter, ‘popt’, is an optional argument used to provide implementation specific information (port number, UART identification, etc.). As for the return value, it is suggested to return the number of data octets transmitted, SHELL\_OUT\_RTN\_CODE\_CONN\_CLOSED if the link has been closed, and SHELL\_OUT\_ERR for any other error.

The output function is also defined by a data type, SHELL\_OUT\_FNCT:

```
typedef CPU_INT16S  (*SHELL_OUT_FNCT) (CPU_CHAR      *,
                                       CPU_INT16U   ,
                                       void          * );
```

Finally the ‘pcmd\_param’ is used to pass additional information to the command. The current implementation has provision for the current working directory, as well as an option parameter used by the output function :

```
typedef struct shell_cmd_param {
    void      *pcur_working_dir;
    void      *pout_opt;
    CPU_BOOLEAN *psession_active;
} SHELL_CMD_PARAM;
```

Note that future implementation could add members to this structure to support more parameters.

## 3.02 $\mu$ C/Shell startup code

We provide you with an example (i.e the application code) use of  $\mu$ C/Shell which is found in `app.c` and it was written to provide a startup example on how to use the capabilities of the  $\mu$ C/Shell module. `app.c` simply initializes  $\mu$ C/OS-II,  $\mu$ C/TCP-IP and  $\mu$ C/Shell, and creates a few tasks and other kernel objects that will give the user information about the state of the system. Note that you DO NOT need an RTOS like  $\mu$ C/OS-II or a TCP/IP stack like  $\mu$ C/TCP-IP to use  $\mu$ C/Shell.

Before you can use  $\mu$ C/Shell, the following has to be performed:

1. Develop/create your command(s)
2. Implement output functions (if needed)
3. Initialize  $\mu$ C/Shell

This section of the manual will give you some examples of the above steps. Note that some sections of the source code have been removed or modified to help focus on the  $\mu$ C/Shell module use.

### Listing 3-1 - Output function

```
CPU_INT16S App_TestShellOut (CPU_CHAR *pbuf, (1)
                                CPU_INT16U buf_len,
                                void *popt)
{
    APP_TRACE_DEBUG((pbuf)); (2)
    APP_TRACE_DEBUG(" executed.\n\r");
    return (buf_len); (3)
}
```

L3-1(1) Function implementing the 'output' facility. This function MUST have the prototype specified in section 2.01.

L3-1(2) This implementation simply outputs 'pbuf', using the trace mechanism (typically the console output).

L3-1(3) Returns the number of positive data octets transmitted (no error).

## Listing 3-2 - Command

```
CPU_INT16S  App_TestCmd (CPU_INT16U      argc,                (1)
                        CPU_CHAR        *argv[],
                        SHELL_OUT_FNCT   out_fnct,
                        SHELL_CMD_PARAM  *pcmd_param)
{
    CPU_INT16U  cmd_namd_len;
    CPU_INT16S  output;
    CPU_INT16S  ret_val;

    cmd_namd_len = Str_Len(argv[0]);
    output       = out_fnct(argv[0],                (2)
                        cmd_namd_len,
                        pcmd_param->pout_opt);

    switch (output) {
        case SHELL_OUT_RTN_CODE_CONN_CLOSED:
        case SHELL_OUT_ERR:
            ret_val = SHELL_EXEC_ERR;
            break;

        default:
            ret_val = output;
    }

    return (ret_val);                (3)
}
```

L3-2(1) Function implementing a test command.

L3-2(2) Use the output function to display the command name.

L3-2(3) The return value is command specific, with the exception of SHELL\_EXEC\_ERR in case of an error.

### Listing 3-3 - Initialization of module

```
static SHELL_CMD AppShellCmdTbl[] = (1)
{
    {"App_test", App_TestCmd},
    {0,          0          }
};

void App_InitShell (void)
{
    CPU_BOOLEAN success;
    SHELL_ERR   err;

    APP_TRACE_DEBUG("Initialize Shell ... ");

    success = Shell_Init(); (2)
    if (success == DEF_OK) {
        APP_TRACE_DEBUG("done.\n\r");
    } else {
        APP_TRACE_DEBUG("failed.\n\r");
        return;
    }

    APP_TRACE_DEBUG("Adding Shell command table ... ");

    Shell_CmdTblAdd("App", App_ShellAppCmdTbl, &err); (3)
    if (err == SHELL_ERR_NONE) {
        APP_TRACE_DEBUG("done.\n\r");
    } else {
        APP_TRACE_DEBUG("failed.\n\r");
    }
}
```

L3-3(1) Declare and populate a SHELL\_CMD structure table that will hold the 'App' shell commands. The first member of this structure is the command name, and the other member a pointer to a function implementing the command itself.

This command table MUST have its last entry set to '0'.

L3-3(2) Initializes **µC/Shell** internal variables.

L3-3(3) Add the AppShellCmdTbl module command table to the Shell.

Once **µC/Shell** has been initialized, the only thing left to do is to call the `Shell_Exec()` function, like depicted above.

### Listing 3-4 - Example use

```

void App_TestShell (void)
{
    SHELL_ERR      err;
    SHELL_CMD_PARAM cmd_param;
#ifdef APP_FS_EN
    FS_DIR         *pdir;
#endif

    APP_TRACE_DEBUG(("Testing Shell, executing command ...\n\r"));

#ifdef APP_FS_EN
    pdir = FS_OpenDir("");
    cmd_param.pcur_working_dir = (void *)pdir;
#else
    cmd_param.pcur_working_dir = (void *)0;
#endif
    cmd_param.pout_opt = (void *)0;

    Shell_Exec("App_test -a -b -c", &App_TestShellOut, &err);           (1)

    switch (err) {
        case SHELL_ERR_NONE:
            APP_TRACE_DEBUG(("Command executed, no error.\n\r"));
            break;

        case SHELL_ERR_NULL_PTR:
            APP_TRACE_DEBUG(("Error, NULL pointer passed.\n\r"));
            break;

        case SHELL_ERR_CMD_NOT_FOUND:
            APP_TRACE_DEBUG(("Error, command NOT found.\n\r"));
            break;

        case SHELL_ERR_CMD_SEARCH:
            APP_TRACE_DEBUG(("Error, searching command.\n\r"));
            break;

        case SHELL_ERR_ARG_TBL_FULL:
            APP_TRACE_DEBUG(("Error, too many arguments\n\r"));
            break;

        default:
            break;
    }
}

```

L3-4(1) Invoke the `Shell_Exec()` function responsible for parsing and calling the specified command. In this case, passing 'App\_Test' will result in the function `App_TestCmd()` to be called (see L2-3(1)).

## 3.04 **μC/Shell** module configuration

The **μC/Shell** module has to be configured according to your specific needs. A template configuration file (`shell_cfg.h`) is included in the module package (see Chapter 1, Directories and Files), and this file should be copied and added to your project. Here is the list of the values and description for each of the configuration variable. However, keep in mind that future releases of this module might include more configuration options.

```
#define SHELL_CFG_CMD_TBL_SIZE 3
```

Size of the command module table. Once this table is full, it is not possible to add any more command module table, unless `Shell_CmdTblRem()` is called. This should be defined to the total amount of module registering command table in **μC/Shell**.

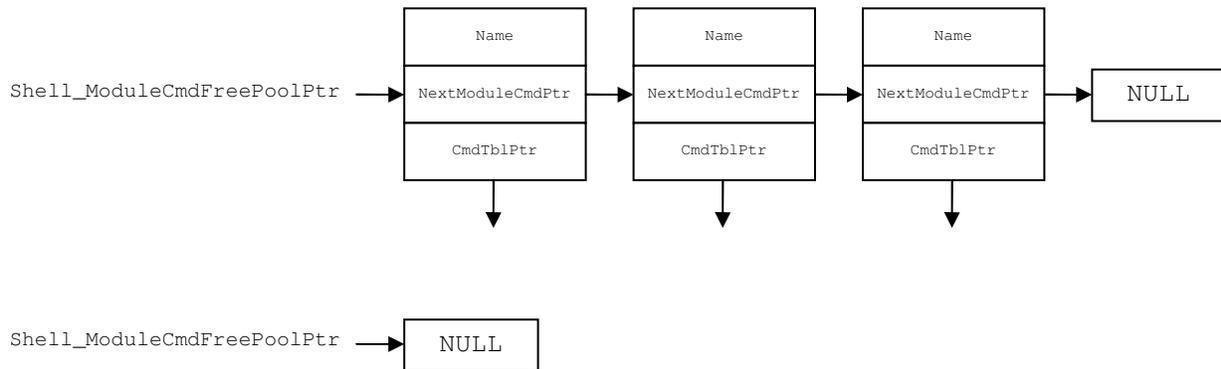
```
#define SHELL_CFG_CMD_ARG_NBR_MAX 5
```

Maximum number or argument(s) a command may pass on the string holding the complete command.

```
#define SHELL_CFG_MODULE_CMD_NAME_LEN_MAX 6
```

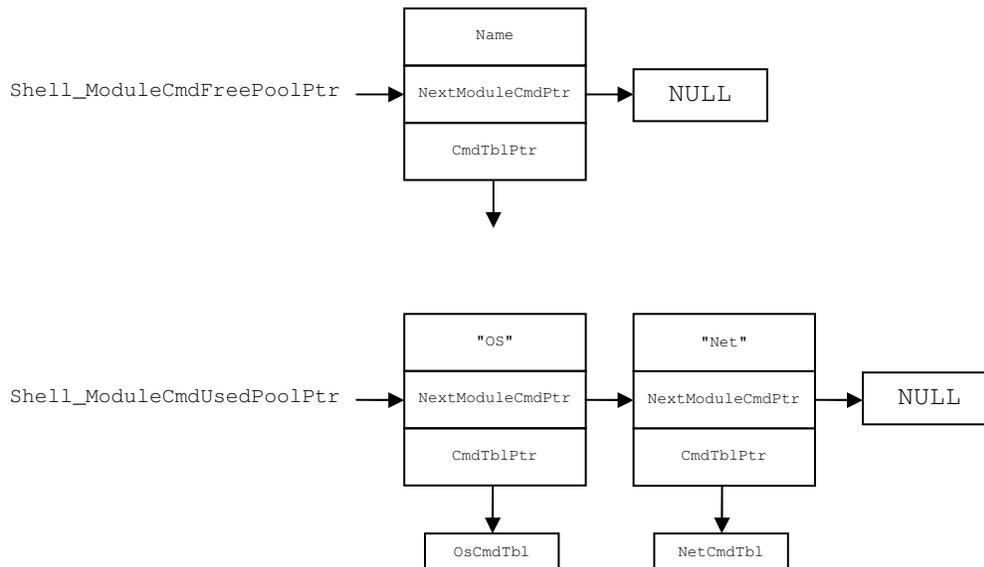
Maximum length for module command name, including the termination NUL character.

At initialization time, that is when the `Shell_Init()` function is called, two module command pools are being created: the free and the used. Right after initialization, no module command are being used, so all of the `SHELL_CFG_CMD_TBL_SIZE` module command are located into the free pool, and the used pool is empty, like displayed below (`SHELL_CFG_CMD_TBL_SIZE` set to 3 in this example).



**Figure 3-1 - Pools after initialization**

Adding module command tables to the shell with `Shell_CmdTblAdd()` results in a free module command being taken from that pool, initialized, and taken into the used pool. Below is a representation of the pools after two module command tables have been inserted.



**Figure 3-2 - Pools after modules insertion**

When the `Shell_Exec()` function is being called in order to parse a line and execute a command, the lists of module commands have to be searched to find a match. Since the module command tables are inserted in a way analog to a stack, the search begins with the last addition. For instance, if the 'OS' table has been inserted just after the 'Net' one, command search will always look at the 'OS' command table, then proceed with the 'Net' command table if a match has not been found.

Two searches are necessary to locate a command. First, the correct module command table has to be found based on the command prefix, and then the corresponding command inside that table is looked for. The second search also starts with index '0' of the command table, and increments that index by '1' until a match is found.

As mentioned at the beginning of this chapter, the command name and arguments passed on the command line are separated by space characters. It is the responsibility of the commands to interpret and extract those arguments. For instance, this command:

```
App_Test -a
```

would result in '-a' to be passed as the argument for the 'App\_Test' command. If the dash needs to be removed, it has to be performed by the command itself. Note however, that **µC/LIB** functions are available to assist with that.

## **μC/Shell API Reference**

This chapter provides a reference to the **μC/Shell** API. Each of the user-accessible services is presented in alphabetical order. The following information is provided for each of those services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of the returned value(s)
- Specific notes and warnings on using the service
- A usage example

## Shell\_CmdTblAdd()

```
void Shell_CmdTblAdd (CPU_CHAR *cmd_tbl_name,  
                     SHELL_CMD cmd_tbl[],  
                     SHELL_ERR *perr);
```

File	Called from
shell.c	Application

Allocates and initializes a module command, and inserts a command table into it.

### Arguments

cmd\_tbl\_name      Pointer to character string representing the name of the command table.

cmd\_tbl            Command table to add.

perr                Pointer to variable that will receive the return error code from this function :

SHELL_ERR_NONE	No error.
SHELL_ERR_NULL_PTR	Argument 'cmd_tbl' passed a NULL pointer.
SHELL_ERR_MODULE_CMD_EMPTY	Command table empty.
SHELL_ERR_MODULE_CMD_ALREADY_IN	Command table already added, or command table name already used.
SHELL_ERR_MODULE_CMD_NONE_AVAIL	NO available module command to allocate.
SHELL_ERR_MODULE_CMD_NAME_NONE	No module command name found.
SHELL_ERR_MODULE_CMD_NAME_TOO_LONG	Module command name too long.
SHELL_ERR_MODULE_CMD_NAME_COPY	Copy error.

### Returned Values

None.

### Notes/Warnings

The 'cmd\_tbl\_ame' argument is the prefix of the commands in 'cmd\_tbl'. In order to speed up the command search, the shell first locate the appropriate table based on the prefix of the command. Hence, it is recommended that all commands in a table be named with the same prefix. For instance, **µC/TCP-IP** related command displaying statistics could look like :

```
Net_stats
```

while a file system command listing the current directory would be :

```
FS_ls
```

The names of those module commands are respectively 'Net' and 'FS'.

## Example

```
static SHELL_CMD App_ShellAppCmdTbl[] = {
    {"App_test", App_TestShellCmd},
    {0,          0}
};

void App_CmdTblAdd (void)
{
    SHELL_ERR  err;

    APP_TRACE_DEBUG(("Adding Shell command table ... "));

    Shell_CmdTblAdd("App", App_ShellAppCmdTbl, &err);
    if (err == SHELL_ERR_NONE) {
        APP_TRACE_DEBUG(("done.\n\r"));
    } else {
        APP_TRACE_DEBUG(("failed.\n\r"));
    }
}
```

## Shell\_CmdTblRem()

```
void Shell_CmdTblRem (CPU_CHAR *cmd_tbl_name,  
                     SHELL_ERR *perr);
```

File	Called from
shell.c	Application

Removes a command table from the shell.

### Arguments

`cmd_tbl_name` Pointer to character string representing the name of the command table.

`perr` Pointer to variable that will receive the return error code from this function :

`SHELL_ERR_NONE` No error.

`SHELL_ERR_NULL_PTR` Argument 'cmd\_tbl\_name' passed a NULL pointer.

`SHELL_ERR_MODULE_CMD_NOT_FOUND` Module command NOT found.

### Returned Values

None.

### Notes/Warnings

None.

### Example

```
void App_CmdTblRem (void)  
{  
    SHELL_ERR err;  
  
    APP_TRACE_DEBUG("Removing Shell command table ... ");  
  
    Shell_CmdTblRem("App", &err);  
    if (err == SHELL_ERR_NONE) {  
        APP_TRACE_DEBUG("done.\n\r");  
    } else {  
        APP_TRACE_DEBUG("failed.\n\r");  
    }  
}
```

# Shell\_Exec ()

```
CPU_INT16S Shell_Exec (CPU_CHAR *in,  
                      SHELL_OUT_FNCT out_fnct,
```

File	Called from
shell.c	Application

Parses and executes the command passed in parameter.

## Arguments

`in` Pointer to a CPU\_CHAR string holding a complete command and its argument(s).

`out_fnct` Pointer to 'output' function used by command.

`perr` Pointer to variable that will receive the return error code from this function :

SHELL_ERR_NONE	No error.
SHELL_ERR_NULL_PTR	Argument 'in' passed a NULL pointer.
SHELL_ERR_CMD_NOT_FOUND	Command NOT found.
SHELL_ERR_CMD_SEARCH	Error searching for command.
SHELL_ERR_CMD_EXEC	Error executing command.
SHELL_ERR_ARG_TBL_FULL	Argument table full and token still to be parsed.

## Returned Values

SHELL\_EXEC\_ERR If command executing error.  
Command specific return value Otherwise.

## Notes/Warnings

The command may generate some output that should be transmitted to some device (socket, RS-232 link, ...). The caller of this function is hence responsible for the implementation of such function, if output is desired..

## Example

```
void App_Exec (void)
{
    SHELL_ERR  err;

    APP_TRACE_DEBUG(("Testing Shell, executing command ...\n\r"));

    Shell_Exec("App_test -a -b -c", &App_TestShellOut, &err);

    switch (err) {
        case SHELL_ERR_NONE:
            APP_TRACE_DEBUG(("Command executed, no error.\n\r"));
            break;

        case SHELL_ERR_NULL_PTR:
            APP_TRACE_DEBUG(("Error, NULL pointer passed.\n\r"));
            break;

        case SHELL_ERR_CMD_NOT_FOUND:
            APP_TRACE_DEBUG(("Error, command NOT found.\n\r"));
            break;

        case SHELL_ERR_CMD_SEARCH:
            APP_TRACE_DEBUG(("Error, searching command.\n\r"));
            break;

        case SHELL_ERR_ARG_TBL_FULL:
            APP_TRACE_DEBUG(("Error, too many arguments\n\r"));
            break;

        case SHELL_ERR_CMD_EXEC:
            APP_TRACE_DEBUG(("Error, executing command.\n\r"));
            break;

        default:
            break;
    }
}
```

# Shell\_Init ()

CPU\_BOOLEAN Shell\_Init (void);

File	Called from
shell.c	Application

Initializes the shell.

## Arguments

None

## Returned Values

DEF\_OK Shell initialization successful.

DEF\_FAIL Otherwise.

## Notes/Warnings

The Shell\_Init() function must be called before the other Shell function are invoked. Shell\_Init() must also only be called once from product's application.

## Example

```
void App_Init (void)
{
    CPU_BOOLEAN success;
    SHELL_ERR err;

    APP_TRACE_DEBUG("Initialize shell ... ");

    Success = Shell_Init();
    if (success == DEF_OK) {
        APP_TRACE_DEBUG(("done.\n\r"));
    } else {
        APP_TRACE_DEBUG(("failed.\n\r"));
    }
}
```

## **μC/Shell Callback Functions Reference**

This chapter provides a reference to the **μC/Shell** callback functions. The following information is provided for each of those functions:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of the returned value(s)
- Specific notes and warnings on using the service

Remember that those functions are referenced by pointers, so their naming convention is left to the developer.

# Shell Output Function

```
CPU_INT16U Shell_OutFnct (CPU_CHAR *pbuf,  
                          CPU_INT16U buf_len,  
                          void *popt);
```

File	Called from
Application	Command

User implemented output facility for the shell.

## Arguments

pbuf	Pointer to buffer to output.
buf_len	Length of buffer
popt	Pointer to implementation specific additional parameter

## Returned Values

The number of positive data octets transmitted	if no error
SHELL_OUT_RTN_CODE_CONN_CLOSED	if link connection closed
SHELL_OUT_ERR	otherwise

## Notes/Warnings

None.

# Shell Command Function

CPU\_INT16S *Shell\_CmdFnct* (CPU\_INT16U *argc*,  
CPU\_CHAR \**argv*[],  
SHELL\_OUT\_FNCT *out\_fnct*)

File	Called from
Application	Command

User implemented shell command.

## Arguments

*argc* Argument count supplied to the function via *argv*.

*argv* Array of pointer to the strings which are those arguments.

*out\_fnct* Pointer to 'output' function.

## Returned Values

SHELL\_EXEC\_ERR if an error occurred  
Any other value otherwise

## Notes/Warnings

The return value for the command will indeed be used as the one for the *Shell\_Exec()* function..

# Appendix A

## **μC/Shell Licensing Policy**

You need to obtain an 'Object Code Distribution License' to embed **μC/Shell** in a product that is sold with the intent to make a profit. Each 'different' product (i.e. your product) requires its own license, but the license allows you to distribute an unlimited number of units for the life of your product. Please indicate the processor type(s) (i.e. ARM7, ARM9, MCF5272, MicroBlaze, Nios II, PPC, etc.) that you intend to use.

For licensing details, contact us at:

### **Micrium**

949 Crestview Circle  
Weston, FL 33327-1848  
U.S.A.

Phone : +1 954 217 2036  
FAX : +1 954 217 2037

WEB : [www.micrium.com](http://www.micrium.com)  
Email : [licensing@micrium.com](mailto:licensing@micrium.com)

# Appendix B

## References

***$\mu$ C/OS-II, The Real-Time Kernel, 2<sup>nd</sup> Edition***

Jean J. Labrosse  
CMP Books, 2002  
ISBN 1-57820-103-9

***Embedded Systems Building Blocks***

Jean J. Labrosse  
R&D Technical Books, 2000  
ISBN 0-87930-604-1