03_ShellCommands.md

RIOT Shell und Command Handler

Um RIOT interaktiv zu verwenden können wir die RIOT Shell verwenden. Um in der RIOT Shell Befehle auszuführen müssen im Projekt die Module `shell` und `shell_commands` hinzugefügt werden. Dazu wird die Makefile angepasst:

```
# Name der Anwendung
APPLICATION = commands-tutorial

# Wenn beim Aufruf von "make", kein Board angegeben wurde, verwende
# "native"
BOARD ?= native

+  # Hinzufügen der benötigten Module shell und shell_commands
+  USEMODULE += shell
+  USEMODULE += shell_commands

# Pfad zur RIOT installation, in diesem Fall befindet sich RIOT im
# User Home verzeichnis
RIOTBASE ?= ${HOME}/RIOT

include $(RIOTBASE)/Makefile.include
```

Die Command Struktur

In der "shell.h" Datei wird die shell_command_t Struktur definiert, diese wird benötigt um Commands in der RIOT shell zu registrieren:

```
/**
 * @brief               A single command in the list of the supported commands.
 * @details             The list of commands is NULL terminated,
 *                      i.e. the last element must be ``{ NULL, NULL, NULL }``.
 */
typedef struct shell_command_t {
        const char *name; /**< Name of the function */
        const char *desc; /**< Description to print in the "help" command. */
        shell_command_handler_t handler; /**< The callback function. */
} shell_command_t;
```

Ein shell_command_handler_t ist dabei ein Function-Pointer zu einer Funktion, die ein int zurückgibt und Zwei Argumente annimmt:

- `int argc` Die Anzahl der Argumente
- `char** argv` Die Liste der Argumente, `argv[0]` ist dabei der Name des aufgerufenen Befehls, `argv[argc]` ist `NULL`

Ein Shell Callback Handler gibt bei erfolgreicher Bearbeitung `0` zurück.

Beispiel eines Shell Callback Handler

```
int test_command_handler(int argc, char** argv) {
        printf("Test command: ");

        for (int i = 0; i < argc; i++) {
            printf("%s ", argv[i]);
        }
```

```
        printf("\n");
        return 0;
}
```

Der obige Command Handler schreibt die Angegeben Argumente auf die Konsole, mit der "Test Command: " Präfix.

Registrierung von Befehlen und Starten der Shell

Um einen Befehl zu Registrieren definieren wir zunächst ein Array, welches 2 Commands enthält:

- Unseren Test command
- Einen command, dessen member ***alle*** auf `NULL` gesetzt sind, um die Liste zu terminieren. (Ähnlich wie das 0-Byte bei C-Strings)

```
shell_command_t commands[] = {
        { "test", "RIOT Shell test command", test_command_handler },
        { NULL, NULL, NULL }
};
```

Nachdem wir dieses Array definiert haben, müssen wir nur noch den Line Buffer erstellen und die Shell starten. Der Line-Buffer ist Speicher für den eingegebenen Text auf der Shell:

```
char line_buf[SHELL_DEFAULT_BUFSIZE];
shell_run(commands, line_buf, SHELL_DEFAULT_BUFSIZE);
```

Hier die vollständige `main.c` Datei:

```
#include "shell.h"
#include "shell_commands.h"

#include <stdio.h>

int test_command_handler(int argc, char** argv) {
        printf("Test command: ");

        for (int i = 0; i < argc; i++) {
            printf("%s ", argv[i]);
        }

        printf("\n");
        return 0;
}

int main(void)
{
        shell_command_t commands[] = {
            { "test", "RIOT Shell test command", test_command_handler },
            { NULL, NULL, NULL }
        };

        char line_buf[SHELL_DEFAULT_BUFSIZE];
        shell_run(commands, line_buf, SHELL_DEFAULT_BUFSIZE);

        return 0;
}
```

Das `commands` -Array und der Line Buffer sind beide *Stack-allocated*, d.h. nachdem die `main` Funktion terminiert, wird der Speicherbereich gelöscht. Da aber `shell_run` eine Unendlichschleife aufruft, wird dies nicht vorkommen.

Das Programm lässt sich über `make all term` builden und starten, durch Eingabe von `help` kann man nun alle vorhandenen Befehle auflisten:

```
Command                 Description
---------------------------------------
test                    RIOT Shell test command
reboot                  Reboot the node
version                 Prints current RIOT_VERSION
pm                      interact with layered PM subsystem
```

Nach der Eingabe des Befehls `test 1 2 3 4` sollte nun folgende Ausgabe zu sehen sein:

```
Test command: test 1 2 3 4
```

Zurück zum Index

Zurück zu Teil 2 Programmaufbau

Weiter zu Kapitel 2: Krypto, AES-ECB