2. U_BOOT_CMD

NXKR_YoungSikYang

Exported on 02/27/2024

Table of Contents

1 How to add and delete a custom command in u-boot

1.1 Write code for the custom command

`path/to/u-boot/common/hello.c`

```
#include <common.h>
#include <command.h>

static int do_print_hello(cmd_tbl_t * cmdtp, int flag, int argc,
                          char * const argv[]) {
        printf("Yang Hello world!\n");
        return 0;
}

U_BOOT_CMD(
    hello,          // Command name
    1, // Max number of arguments
    0,                  // Repeatable
    do_print_hello,       // Command function
    "print \"Yang Hello world!(C)\"", // Command description
    "Usage: hello" // Help text
);
```

1.2 Add to the compile list

`common/Makefile`

```
obj-$(CONFIG_CMD_HELLO) += hello.o
```

This mean "Include the `hello.o` in the build based on the configuration option `CONFIG_CMD_HELLO` "

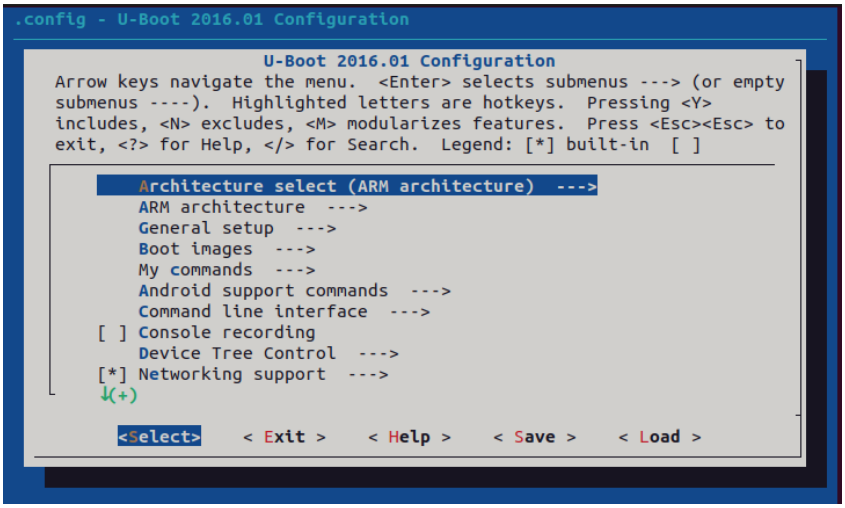
`common/kconfig`

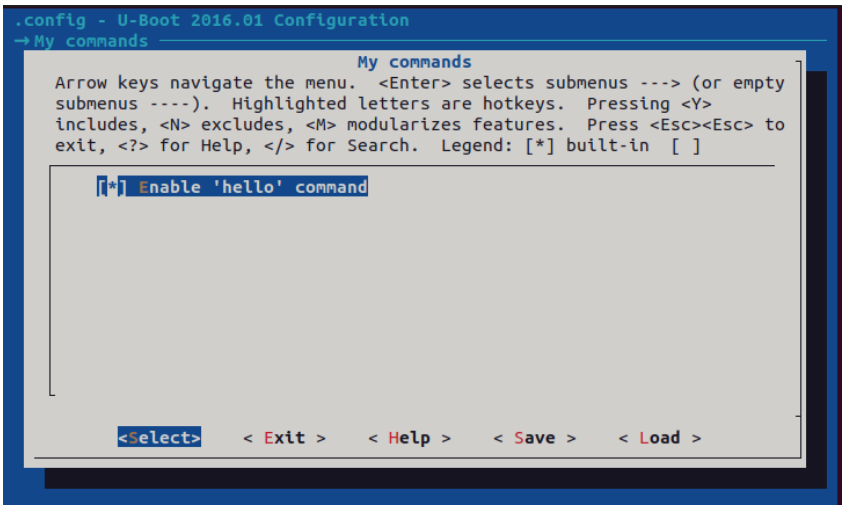
```
menu "My commands"

config CMD_HELLO
        bool "Enable 'hello' command"
        default y
        help
```

```
            print "Hello world!(Kconfig)"
endmenu
```

1.2.1 Result

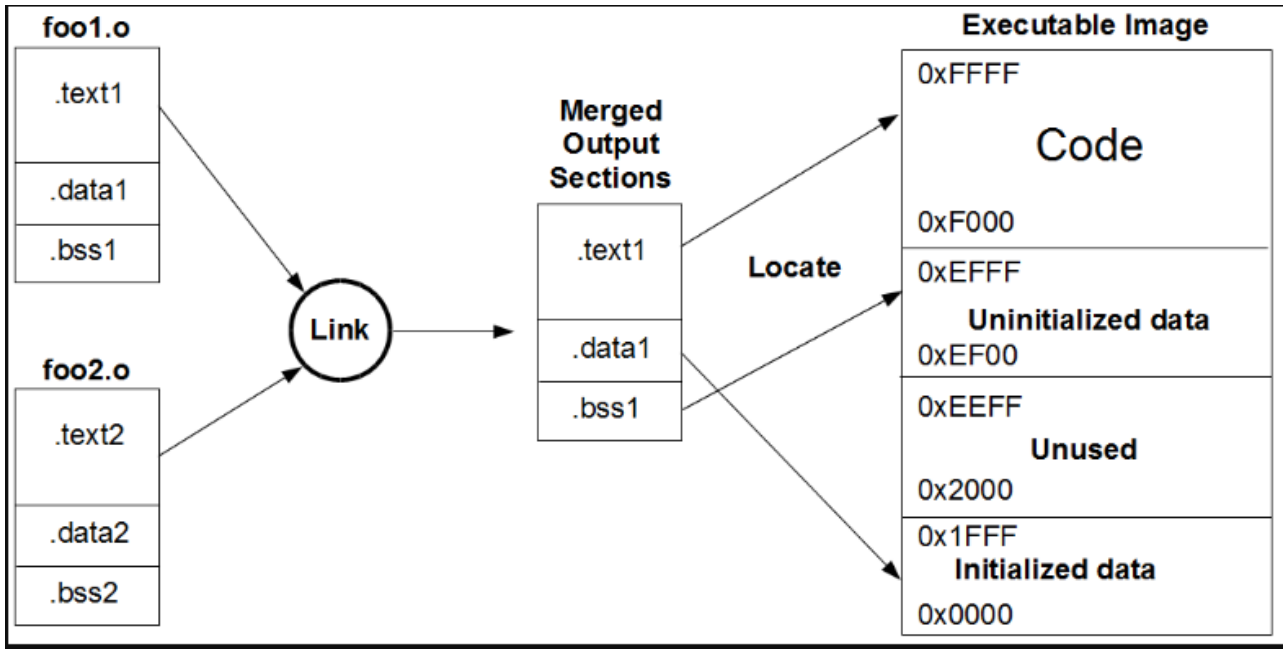




1.3 Rebuild U-Boot and flash it onto the board to check if the command has been successfully added

```
bitminer# hello
Yang Hello world!
```

2 LD script



2.1 What is Linker?

The linker combines multiple object files (produced by the compiler) into a single executable.

2.2 What is an LD Script?

An LD script provides the linker with instructions on how to assemble compiled source files (.o files) into a final executable.

- It specifies the memory layout of the output file, including where sections(chunks of code or data) should be placed in memory, the entry point of the executable, and how to resolve external references.
- The `.lds` file extension denotes a linker script file used by GNU's LD linker

2.3 Why Use an LD Script?

- LD scripts are particularly useful in embedded OS, where precise control over the memory layout of the compiled code is needed.
    - For example, there may be a need to place code or data at specific addresses to match the hardware requirements or bootloading procedures.

2.4 What is Relocation?

Relocation is a process performed by the linker to adjust the addresses of symbolic references(such as variable names or function names) within the code and data sections so that they point to specific memory locations.

2.5 Why is Relocation Needed?

- This process is necessary because, during compilation, the exact addresses of functions, variables, or other data might not be known.
- The compiler generates object code with placeholders(such as variable names or function names) for addresses that cannot be determined until later. The process of relocation involves updating these placeholders with actual addresses.

example.lds

```
/* Example linker script */
MEMORY
{
  ROM (rx) : ORIGIN = 0x08000000, LENGTH = 256K /* Memory region with read and
execute permissions (rx) for code*/
  RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K /* Memory region with read, write,
and execute permissions (rwx) for data */
}

SECTIONS
{
  /* Define the .text section (code) */
  .text : {
    *(.text) /* Include all .text sections from input files */
    *(.rodata) /* Include read-only data */
    . = ALIGN(4); /* Align to 4-byte boundary */
  } > ROM

  /* Define the .data section (initialized data) */
  .data : {
    *(.data) /* Include all .data sections from input files */
    . = ALIGN(4);
  } > RAM AT > ROM /* Place in RAM but initialize from ROM */

  /* Define the .bss section (uninitialized data) */
  .bss : {
    *(.bss)
    *(COMMON)
```

```
    . = ALIGN(4);
  } > RAM

  /* Define a symbol at a specific address */
  _special_address = 0x20001000;
}
```