Personalized Bootloader and first Hello World

Author:

Armando Emmanuel Correa Amorelli

Zapopan, Jalisco, May 30, 2023

Table of Contents

Introduction

Boot managers, on rare occasions, adapt to the specific needs of a product due to their generic nature that provides support for different architectures. U-Boot takes things to the next level as a result of its development process as an open-source software project that requires adjustments to become usable software for a commercial product.

In the following activity, customization of U-Boot software will be exercised as an introductory approach to its development cycle.

Objective

Modifying the source code of the boot manager by extending its coding for a specific purpose.

First Hello World

Base Header Files

For creating our first hello world, we first need to include 2 different header files, <include/common.h> and <include/exports.h>

common.h

This header includes a set of header files that contain necessary definitions, declarations, and functions for various aspects of U-Boot development, including configuration options, error handling, time operations, string manipulation, I/O operations, kernel-related operations, display options, and formatting strings. These files are the follow:

1. `config.h`: This header file typically contains configuration options and settings specific to the U-Boot build. It allows you to customize various aspects of U-Boot, such as enabled features, hardware configurations, and default settings.

2. `errno.h`: This header file defines various error codes that are used to indicate different types of errors in the system. It provides a standardized set of error constants that can be used by U-Boot and other software components.

3. `time.h`: This header file provides functions and structures for working with time and date-related operations. It includes functions to retrieve the current time, manipulate time values, and convert between different time representations.

4. `linux/types.h`: This header file defines various data types commonly used in Linux and U-Boot, such as u8, u16, u32, s8, s16, s32, which represent unsigned and signed integers of specific sizes. It provides portable definitions for these types across different platforms.

5. `linux/printk.h`: This header file provides functions and macros for printing messages to the console or system log. It includes the printk() function and related macros that are used for logging debug messages, warnings, and errors in the kernel or U-Boot code.

6. `linux/string.h`: This header file contains functions for manipulating strings. It provides functions for copying, comparing, searching, and modifying strings. These functions are commonly used for various string operations in U-Boot.

7. `stdarg.h`: This header file provides facilities for working with functions that accept a variable number of arguments. It includes macros and types needed to handle functions like printf() that can accept a variable number of arguments.

8. `stdio.h`: This header file contains declarations for standard input/output operations. It provides functions and macros for reading and writing data to streams, such as printf(), scanf(), and file I/O functions like fopen(), fclose(), etc.

9. `linux/kernel.h`: This header file provides various kernel-related definitions and macros. It includes macros for defining data alignment, memory barriers, bit manipulation, and other low-level operations commonly used in kernel or U-Boot code.

10. `asm/u-boot.h`: This header file contains definitions and structures related to the U-Boot boot process and communication with the Linux kernel. It includes information about the boot parameters passed from U-Boot to the kernel and other boot-related structures.

11. `display_options.h`: This header file defines display-related options and configurations for U-Boot. It includes settings for video modes, framebuffers, resolutions, and other display-related parameters.

12. `vsprintf.h`: This header file provides functions and macros for formatting strings with variable arguments. It includes functions like vsprintf() and vsnprintf() that are used for formatting strings with a format specifier and a variable number of arguments.

export.h

On the other hand, this file contains a few functions that can be used.

Here is a description of the purpose of each function:

1. `get_version()`: Returns the version number of the program or component.

2. `getc()`: Reads a character from the input source (e.g., console, UART, etc.).

3. `tstc()`: Tests whether a character is available for reading from the input source.

4. `putc(const char)`: Writes a character to the output destination (e.g., console, UART, etc.).

5. `puts(const char*)`: Writes a null-terminated string to the output destination.

6. `printf(const char* fmt, ...)`: Formats and prints a string with a variable number of arguments.

7. `install_hdlr(int, interrupt_handler_t, void*)`: Installs an interrupt handler for a specific interrupt number.

8. `free_hdlr(int)`: Frees the installed interrupt handler for a specific interrupt number.

9. `malloc(size_t)`: Allocates memory dynamically from the heap.

10. `free(void*)`: Frees memory previously allocated by malloc().

11. `__udelay(unsigned long)`: Delays the execution for a specified number of microseconds.

12. `get_timer(unsigned long)`: Returns the elapsed time in milliseconds since a specific reference point.

13. `vprintf(const char *, va_list)`: Formats and prints a string with a variable argument list.

14. `simple_strtoul(const char *cp, char **endp, unsigned int base)`: Converts a string to an unsigned long integer.

15. `strict_strtoul(const char *cp, unsigned int base, unsigned long *res)`: Converts a string to an unsigned long integer with error checking.

16. `env_get(const char *name)`: Retrieves the value of an environment variable.

17. `env_set(const char *varname, const char *value)`: Sets the value of an environment variable.

18. `simple_strtol(const char *cp, char **endp, unsigned int base)`: Converts a string to a long integer.

19. `strcmp(const char *cs, const char *ct)`: Compares two strings and returns the difference.

20. `ustrtoul(const char *cp, char **endp, unsigned int base)`: Converts a string to an unsigned long integer.

21. `ustrtoull(const char *cp, char **endp, unsigned int base)`: Converts a string to an unsigned long long integer.

22. `i2c_write (uchar, uint, int , uchar* , int)`: Writes data to an I2C device.

23. `i2c_read (uchar, uint, int , uchar* , int)`: Reads data from an I2C device.

24. `mdio_get_current_dev(void)`: Retrieves the current MDIO bus device.

25. `phy_find_by_mask(struct mii_dev *bus, unsigned phy_mask, phy_interface_t interface)`: Finds a PHY device based on the MDIO bus, PHY mask, and interface.

26. `mdio_phydev_for_ethname(const char *ethname)`: Finds a PHY device based on the Ethernet interface name.

27. `miiphy_set_current_dev(const char *devname)`: Sets the current MDIO bus device based on the device name.

28. `app_startup(char * const *)`: Starts up an application with command-line arguments.

Standalone App

Creating standalone app

For this point, first we had to create a C file with the previos headers. And a empty function with a different name from main, otherwise we would get a undefine command output when executing.

Next, we need to add inside the function, the following line:

```
app_startup(argv);
```

This is needed to make the code executable as a standalone app from U Boot.

Showing U Boot version

For this, we can use multiple ways, but the one we chose to implement it using the `get_version()` function paired with a `printf()`

```
printf("Actual U-Boot ABI version %d\n", (int)get_version());
```

Create A Makefile

For this point instead of creating a completely new Makefile, we choose to modify the existing one for standalone applications examples, by adding the following line

```
extra-y        += myApp
```

Running the binary

To be able to run the binary at the BBB, we first need to copy it to it. For this we can either use xmodem or tftp, both of them work fine. In this case, we choose to use tftp because we have already used it previously for loading the kernel and device tree. For this, we execute the following command

```
tftp 0x80300000 myApp.bin
```

Before running this command we need to make sure that the .bin file is located at our tftp directory.

Once the bin file has been copied to the BBB we can run it with the command `go 0x80300001`

Banner creation

To create an original banner for this bootloader, we chose to use some third-party software to generate ASCII art. Once we had this, we simply add enough `printf()` to print the whole banner.

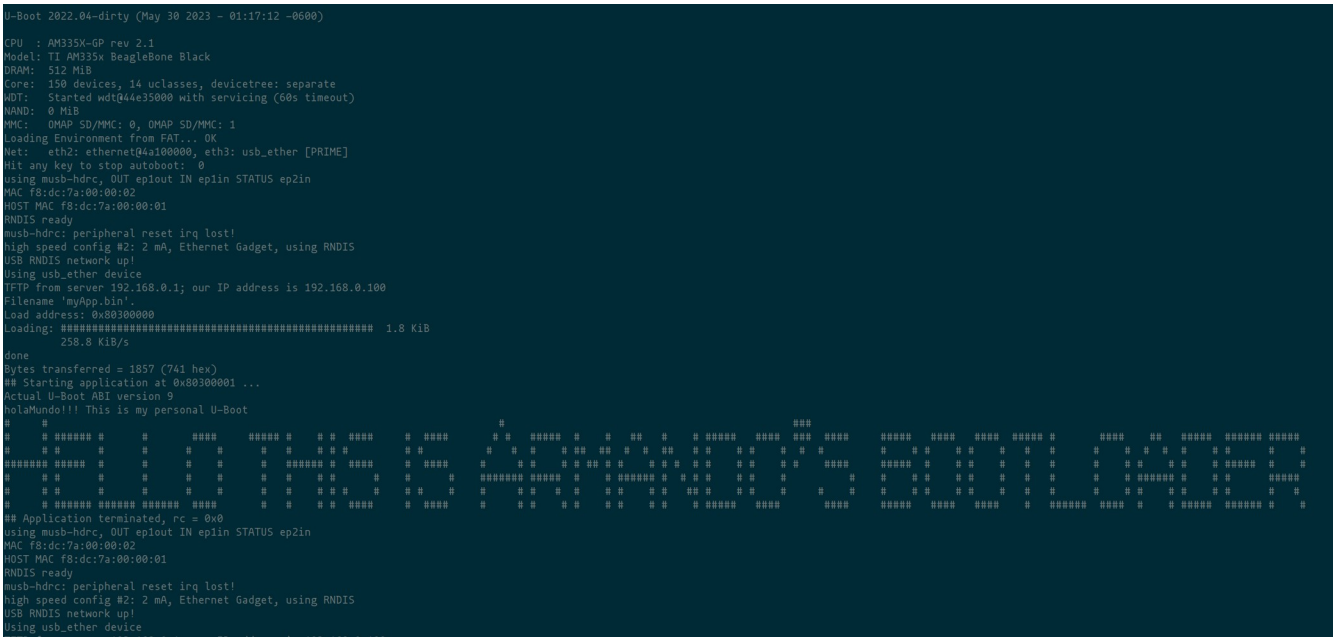
After this, we recompile and send the standalone app, and additionally, we chose to create a new environment variable at the bootloader to make it easier to display this banner every time we boot from the SD card.
For this we used the following commands

```
setenv tftp_myapp “tftp 0x80300000 myApp.bin; go 0x80300001;”

setenv bootcmd “run tftp_myapp; run tftp_custom; run findfdt; run
init_console; run finduuid; run distro_bootcmd”

saveenv
```

With this, we make sure that every time that we boot we load the binary file and execute it before loading the kernel and the file system. Getting, as a result, something like this



Conclusions

Being able to modify and personalize the bootload is, from my point of view, a powerful tool. This way you can create a bootloader for a specific purpose or give it some personality so it can be distinguished from the others.

Bibliography

[1]“The U-Boot Documentation — Das U-Boot unknown version documentation,” *u-boot.readthedocs.io*. https://u-boot.readthedocs.io/en/latest/ (accessed May 29, 2023).