001

Hello World

Operating Systems

Lab001 Hello World

We need to install the arm-none-eabi-gcc compiler. This compiler is essential for cross-compiling code for ARM Cortex-M and Cortex-A processors in a bare-metal environment (without an operating system). Below are detailed instructions to install arm-none-eabi-gcc on macOS, Linux, and Windows.

For macOS

Method 1: Using Homebrew

Homebrew is a popular package manager for macOS that simplifies the installation of software.

Step 1: Install Homebrew (If Not Already Installed)

Open the Terminal application and run:

/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

Follow the on-screen instructions to complete the installation.

Step 2: Install the ARM Cross-Compiler

Run the following command to install the ARM GCC toolchain:

brew install armmbed/formulae/arm-none-eabi-gcc

Note: If you encounter any issues with the above formula, you can try installing from another tap:

brew tap PX4/px4
brew install gcc-arm-none-eabi

Step 3: Verify the Installation

Check if the compiler is installed correctly:

arm-none-eabi-gcc --version

You should see output similar to:

arm-none-eabi-gcc (GNU Arm Embedded Toolchain ...) ...

Method 2: Download Pre-Built Binaries from ARM

Step 1: Download the Toolchain

Visit the official ARM developer website to download the GNU Arm Embedded Toolchain:

- GNU Arm Embedded Toolchain Downloads

Select the macOS version (.pkg file) appropriate for your development needs.

Step 2: Install the Toolchain

- Run the downloaded .pkg installer.
- Follow the installation prompts.

Step 3: Update Your PATH Environment Variable

Add the installation directory to your PATH. Assuming it installs to /usr/local/bin:

export PATH="/usr/local/bin:$PATH"

You may want to add this line to your ~/.bash_profile, ~/.zshrc, or ~/.bashrc file, depending on your shell.

Step 4: Verify the Installation

arm-none-eabi-gcc --version

For Linux

Method 1: Using Package Manager (Ubuntu/Debian)

Step 1: Update Package Lists

sudo apt-get update

Step 2: Install the ARM Cross-Compiler

sudo apt-get install gcc-arm-none-eabi gdb-arm-none-eabi

Note: On some systems, the package may be named gcc-arm-none-eabi or binutils-arm-none-eabi. Adjust accordingly.

Step 3: Verify the Installation

arm-none-eabi-gcc --version

Method 2: Download Pre-Built Binaries from ARM

Step 1: Download the Toolchain

- Visit the GNU Arm Embedded Toolchain Downloads.
- Choose the Linux 64-bit tarball (.tar.bz2 file).

Step 2: Extract the Archive

Navigate to the directory where you downloaded the file and run:

tar -xjf gcc-arm-none-eabi-*-x86_64-linux.tar.bz2

Step 3: Move to an Appropriate Location

sudo mv gcc-arm-none-eabi-*/ /opt/gcc-arm-none-eabi

Step 4: Update Your PATH Environment Variable

Add the toolchain to your PATH by editing ~/.bashrc or ~/.bash_profile:

export PATH="/opt/gcc-arm-none-eabi/bin:$PATH"

Reload your shell configuration:

source ~/.bashrc

Step 5: Verify the Installation

arm-none-eabi-gcc --version

For Windows

Method 1: Using the Official Installer

Step 1: Download the Installer

- Go to the GNU Arm Embedded Toolchain Downloads.
- Download the Windows version (.exe installer).

Step 2: Run the Installer

- Double-click the downloaded .exe file.
- Follow the installation wizard steps.
- Accept the license agreement.
- Choose the installation directory (e.g., C:\Program Files (x86)\GNU Arm Embedded Toolchain).

Step 3: Add Toolchain to the System PATH

- Open Control Panel > System and Security > System.
- Click on Advanced system settings.
- Click on Environment Variables.
- Under System variables, select Path and click Edit.
- Click New and add the path to the bin directory of your installation, e.g.:

C:\Program Files (x86)\GNU Arm Embedded Toolchain\bin

- Click OK to close all dialogs.

Step 4: Verify the Installation

Open Command Prompt and run:

arm-none-eabi-gcc --version

Method 2: Using MSYS2

MSYS2 provides a Unix-like environment on Windows and includes package management.

Step 1: Install MSYS2

Download and install MSYS2 from msys2.org.

Step 2: Update Package Database and Core System Packages

Open the MSYS2 MSYS terminal and run:

pacman -Syu

Close and reopen the MSYS2 terminal, then run:

pacman -Su

Step 3: Install the ARM Cross-Compiler

pacman -S mingw-w64-x86_64-arm-none-eabi-gcc

Step 4: Add to PATH (Optional)

Add the following to your Windows PATH environment variable:

C:\msys64\mingw64\bin

Adjust the path if you installed MSYS2 in a different location.

Step 5: Verify the Installation

Open MSYS2 MinGW 64-bit terminal and run:

arm-none-eabi-gcc --version

Additional Notes

• Administrator Privileges: Some steps may require administrator or root privileges, especially when installing software or modifying system-wide settings.
• Version Compatibility: Ensure that all components (compiler, debugger, etc.) are compatible in terms of versions.
• Multiple Versions: If you have multiple versions of arm-none-eabi-gcc installed, make sure the correct one is being used by checking your PATH variable.

"Hello World"

Creating a "Hello World" program for the BeagleBone Black at the bare-metal level is an excellent way to learn about low-level programming and hardware interaction. In this guide, we'll create a minimal program that outputs "Hello World" via the serial console using BeagleBone Black's UART0 interface.

Prerequisites

- **Cross-Compiler Toolchain**: An ARM cross-compiler (e.g., arm-none-eabi-gcc) installed on your development machine. This allows you to compile code for the ARM Cortex-A8 processor in the BeagleBone Black.
- **Serial Communication Software**: A terminal program such as minicom, screen, Tera Term, or PuTTY to view serial output and interact with the U-Boot bootloader. You can use CoolTerm on Mac computers.
- **USB-to-Serial Cable**: Connect your computer to the BeagleBone Black's serial debug port.
- **BeagleBone Black Board**: The hardware platform you'll be programming.

Hardware Setup

1. Connect the Serial Debug Port

- **Use a USB-to-Serial Adapter**: Connect your computer to the BeagleBone Black's serial debug header (UART0).
- **UART0 Pin Assignments**:
    - **BeagleBone Black UART0 Header (J1)**:
        - **Pin 1**: Ground (GND)
        - **Pin 4**: UART0 Receive (RXD)
        - **Pin 5**: UART0 Transmit (TXD)

Note: Ensure that you cross-connect the RXD and TXD lines between the BeagleBone Black and your USB-to-Serial adapter:

- **BeagleBone Black TXD → USB-to-Serial Adapter RXD**
- **BeagleBone Black RXD → USB-to-Serial Adapter TXD**
- **Alternative**: Some BeagleBone Black revisions provide a micro-USB port for serial communication. If available, you can use a micro-USB cable instead.

2. Set Up Serial Communication Software

- **Configure Your Terminal Program** with the following settings:
- **Baud Rate**: **115200**
- **Data Bits**: **8**
- **Parity**: **None**
- **Stop Bits**: **1**
- **Flow Control**: **None**
- **Open the Serial Connection** to the BeagleBone Black. You should be able to see U-Boot messages when the board is powered on.

Writing the "Hello World" Program

We will write two source files: an assembly startup file (startup.s) and a C program (hello.c). We will also use a linker script (memmap) to control the memory layout.

1. Create the Assembly Startup Code (startup.s)

This code sets up the initial stack pointer and provides minimal startup routines.

```
/* startup.s */

.section .text
.syntax unified
.code 32
.globl _start

_start:
   ldr sp, =_stack_top      @ Initialize stack pointer
   bl main                  @ Call main
   b hang                   @ Infinite loop

hang:
   b hang

.globl PUT32
PUT32:
   str r1, [r0]
   bx lr

.globl GET32
```

```
GET32:
   ldr r0, [r0]
   bx lr

.section .bss
.align 4
_stack_bottom:
   .skip 0x1000          @ Reserve 4KB for stack
_stack_top:
```

Explanation:

- **_start**: Entry point of the program. Initializes the stack pointer and calls main.
- **PUT32 and GET32**: Simple functions to write to and read from memory-mapped I/O registers.
- **Stack Space**: Reserves 4KB for the stack between _stack_bottom and _stack_top.

2. Write the Main Program in C (hello.c)

This code sends the “Hello World” string via UART0.

```
/* hello.c */

extern void PUT32(unsigned int, unsigned int);
extern unsigned int GET32(unsigned int);

#define UART0_BASE    0x44E09000
#define UART_THR      (UART0_BASE + 0x00)   // Transmitter Holding Register
#define UART_LSR      (UART0_BASE + 0x14)   // Line Status Register
#define UART_LSR_THRE  0x20                 // Transmit Holding Register Empty

void uart_send(unsigned char x) {
   while ((GET32(UART_LSR) & UART_LSR_THRE) == 0);  // Wait for THR empty
   PUT32(UART_THR, x);
}

void uart_puts(const char *s) {
   while (*s) {
      uart_send(*s++);
   }
}

int main(void) {
   uart_puts("Hello World\n");
   while (1);  // Prevent exit
```

```
    return 0;
}
```

Explanation:

- uart_send: Sends a single character over UART0 by writing to the Transmitter Holding Register (THR).
- uart_puts: Sends a null-terminated string by calling uart_send for each character.
- main: Calls uart_puts to send “Hello World” over UART0 and then enters an infinite loop.

3. Create the Linker Script (memmap)

This script defines the memory layout for the program, specifying where code and data sections are placed in memory.

```
/* memmap */

MEMORY
{
    ram : ORIGIN = 0x82000000, LENGTH = 256K
}

SECTIONS
{
    . = ORIGIN(ram);

    .text :
    {
        *(.text*)
        *(.rodata*)
    } > ram

    .data : AT(ADDR(.text) + SIZEOF(.text))
    {
        __data_start__ = .;
        *(.data*)
        __data_end__ = .;
    } > ram

    .bss :
    {
        __bss_start__ = .;
        *(.bss*)
        *(COMMON)
```

```
    __bss_end__ = .;
  } > ram

  _stack_bottom = __bss_end__;
  _stack_top = ORIGIN(ram) + LENGTH(ram);
}
```

Explanation:

- **Memory Region**: Defines a memory region named ram starting at 0x82000000 with a length of 256KB.
- **Sections**:
  - .text: Contains the code (including .rodata), placed at the start of the RAM.
  - .data: Contains initialized data.
  - .bss: Contains uninitialized data.
- **Stack Placement**:
  - Defines _stack_bottom and _stack_top to set up the stack space.

Compiling the Program

Use the ARM cross-compiler toolchain to compile and link your program.

1. Clean Previous Builds

```
rm -f *.o hello.elf hello.bin hello.list
```

2. Assemble startup.s

```
arm-none-eabi-as --warn --fatal-warnings startup.s -o startup.o
```

3. Compile hello.c

```
arm-none-eabi-gcc -c -mcpu=cortex-a8 -mfpu=neon -mfloat-abi=hard -Wall -Werror -O2 -nostdlib -nostartfiles -ffreestanding hello.c -o hello.o
```

Explanation:

- -mcpu=cortex-a8: Specifies the target CPU.
- -mfpu=neon -mfloat-abi=hard: Specifies the floating-point unit.
- -Wall -Werror: Enables all warnings and treats them as errors.
- -O2: Optimization level 2.
- -nostdlib -nostartfiles -ffreestanding: Compiles for a freestanding environment without standard libraries or startup files.

4. Link the Object Files

Link the object files using the linker script memmap
arm-none-eabi-ld -T memmap startup.o hello.o -o hello.elf

5. Generate Binary File

arm-none-eabi-objcopy hello.elf -O binary hello.bin

6. Disassemble the ELF File (Optional)

Disassemble the ELF file to verify addresses
arm-none-eabi-objdump -D hello.elf > hello.list

Loading the Program onto the BeagleBone Black

We will use U-Boot to load and execute the program via the serial console.

1. Connect to the BeagleBone Black via Serial Console

Ensure that your serial connection is established and that you can see U-Boot messages.

2. Interrupt the Boot Process

When the board starts, press any key in the serial console to stop the autoboot process.

You should see the U-Boot prompt:

U-Boot#

3. Load the Binary via YMODEM

At the U-Boot prompt, use the loady command to prepare for a YMODEM file transfer.

=> loady 0x82000000

- **Explanation**: This command tells U-Boot to receive a file via YMODEM and load it into memory at address 0x82000000.

4. Start the YMODEM Transfer

In your terminal program, initiate a YMODEM file transfer and select hello.bin.

- **For** minicom:
  - Press Ctrl+A then S to bring up the send menu.
  - Choose ymodem protocol.
  - Select hello.bin to send.
- **For** Tera Term:
  - Go to File → Transfer → YMODEM → Send.
  - Select hello.bin to send.
- **For** PuTTY **Users**:
  - PuTTY does not support file transfers directly. Use an alternative terminal program like Tera Term or ExtraPuTTY.

5. Wait for the Transfer to Complete

U-Boot will display progress information. Once the transfer is complete, it will show the total size and start address.

6. Execute the Program

At the U-Boot prompt, execute the program using the go command:

```
=> go 0x82000000
```

- **Explanation**: This tells U-Boot to start executing code from address 0x82000000.

Viewing the Output

After executing the go command, you should see Hello World printed in your serial console.

```
Hello World
```

Emulating ARM Bare-Metal Code with QEMU

(Optional)

While QEMU doesn't have a specific machine model for the BeagleBone Black, you can emulate a similar ARM platform to test your bare-metal code. We'll use the VersatilePB machine provided by QEMU, which is sufficient for running simple ARM code like the "Hello World" example.

Steps to Test the "Hello World" Example on QEMU

1. Install QEMU

First, install QEMU on your development machine.

- **On Ubuntu/Debian:**

```
sudo apt-get update
sudo apt-get install qemu-system-arm
```

- **On macOS (using Homebrew):**

```
brew install qemu
```

Installing QEMU on Windows

There are several methods to install QEMU on Windows:

1. Using the Official Windows Binaries
2. Using MSYS2
3. Using Windows Subsystem for Linux (WSL)

I'll provide detailed steps for each method so you can choose the one that best fits your requirements.

Method 1: Installing Official QEMU Windows Binaries

The QEMU project provides pre-built Windows binaries that you can download and use directly.

Step 1: Download QEMU for Windows

• Visit a Trusted Source: While the official QEMU website doesn't provide Windows binaries, you can download them from reputable sources like:
qemu.weilnetz.de: Offers 64-bit Windows binaries.

Step 2: Download the Installer or Zip File

• Choose the Latest Version: Download either the installer (.exe) or the zip archive (.zip).
Example: qemu-w64-setup-20240816.exe (version and date may vary).

Step 3: Install QEMU

• Using the Installer:
• Run the downloaded .exe file.
• Follow the installation prompts.
• Select the components you wish to install.
• Choose the installation directory (e.g., C:\Program Files\qemu).
• Using the Zip Archive:
• Extract the contents to a directory of your choice (e.g., C:\qemu).

Step 4: Add QEMU to the System PATH

1. Press Win + X and select System.
2. Click on Advanced system settings.
3. In the System Properties window, click Environment Variables.
4. Under System variables, select Path and click Edit.
5. Click New and add the path to QEMU's bin directory (e.g., C:\Program Files\qemu or C:\qemu).
6. Click OK to close all dialogs.

Step 5: Verify the Installation

- Open Command Prompt:
- Press Win + R, type cmd, and press Enter.
- Run:

qemu-system-arm --version

- You should see output displaying the QEMU version information.

Method 2: Installing QEMU via MSYS2

MSYS2 provides a Unix-like environment on Windows, including a package manager to install software like QEMU.

Step 1: Install MSYS2

- Download the Installer:
- Visit https://www.msys2.org/.
- Download the installer (msys2-x86_64-*.exe).
- Run the Installer:
- Follow the installation prompts.
- Install MSYS2 to the default location (e.g., C:\msys64).

Step 2: Update the Package Database and Core Packages

- Open the MSYS2 MSYS terminal from the Start menu.
- Run the update commands:

  pacman -Syu

- If prompted, press Y to proceed.
- Close and reopen the MSYS2 terminal if instructed.

- Run the update again:

pacman -Su

Step 3: Install QEMU

- In the MSYS2 terminal, install QEMU:

  pacman -S mingw-w64-x86_64-qemu

- Press Y when prompted to confirm the installation.

Step 4: Add MSYS2 QEMU to the System PATH (Optional)

- If you want to use QEMU from the regular Command Prompt, add the MSYS2 mingw64\bin directory to your system PATH:
- Path to add: C:\msys64\mingw64\bin

- Follow the same steps as in Method 1 to edit the Environment Variables.

Step 5: Verify the Installation

- Open the MSYS2 MinGW 64-bit terminal.
- Run:

  qemu-system-arm --version

- You should see QEMU version information.

Method 3: Using Windows Subsystem for Linux (WSL)

If you're using Windows 10 (version 2004 and later) or Windows 11, you can install WSL to run a Linux environment directly on Windows.

Step 1: Enable WSL

- Open PowerShell or Command Prompt as Administrator.
- Run:

  wsl --install

- This command installs WSL with the default Linux distribution (Ubuntu).
- Restart your computer when prompted.

Step 2: Install QEMU in WSL

- Open the Ubuntu terminal from the Start menu.
- Update package lists:

sudo apt-get update

- Install QEMU:

sudo apt-get install qemu-system-arm

- Press Y to confirm the installation.

Step 3: Verify the Installation

- Run:

qemu-system-arm --version

- You should see QEMU version information.

Modify the Code for QEMU

Since the hardware peripherals and memory addresses differ between the BeagleBone Black and the emulated machine in QEMU, you'll need to adjust your code accordingly.

a. Update the Memory Addresses

- Linker Script (linker.ld):

Change the starting address to the RAM base address for the VersatilePB machine, which is 0x00010000.

```
ENTRY(_start)

SECTIONS
{
  . = 0x00010000; /* Starting address in RAM */

  .text :
  {
    *(.text*)
  }

  .data : AT (ADDR(.text) + SIZEOF(.text))
  {
    *(.data*)
  }

  .bss :
  {
    __bss_start = .;
    *(.bss*)
    __bss_end = .;
  }
```

b. Update UART Base Address and Registers

• UART Base Address:
The VersatilePB machine uses the PL011 UART at base address 0x101f1000.

#define UART0_BASE 0x101f1000

• Update UART Registers in main.c:
Adjust your UART register definitions to match the PL011 UART.

```
#define UART0_BASE 0x101f1000

#define UART_DR     0x00  // Data Register
#define UART_FR     0x18  // Flag Register
#define UART_FR_TXFF 0x20 // Transmit FIFO Full

volatile unsigned int * const UART0 = (unsigned int *)UART0_BASE;

void uart_putc(char c) {
   // Wait until there is space in the FIFO
   while (UART0[UART_FR / 4] & UART_FR_TXFF);
   UART0[UART_DR / 4] = c;
}

void uart_puts(const char *s) {
   while (*s) {
      uart_putc(*s++);
   }
}

void main() {
   uart_puts("Hello World\n");
   while (1); // Infinite loop to prevent exit
}
```

c. Adjust the Startup Assembly (startup.S)

Ensure that the stack pointer is set to a valid memory address within the emulated RAM.

```
    .section .text
    .global _start

_start:
    b reset

reset:
    ldr sp, =0x00020000   @ Set up stack pointer (within RAM)
    bl main               @ Call the main function
loop:
    b loop                @ Infinite loop to prevent exit
```

3. Compile the Program

Use the ARM cross-compiler to build your program.

```
arm-none-eabi-gcc -c startup.S -o startup.o
arm-none-eabi-gcc -c main.c -o main.o
arm-none-eabi-ld -T linker.ld startup.o main.o -o hello_world.elf
```

- Explanation:
  - startup.S: Assembly startup code.
  - main.c: The main program.
  - linker.ld: Linker script adjusted for QEMU's VersatilePB machine.

4. Run the Program in QEMU

Execute your program using QEMU with the appropriate machine and options.

```
qemu-system-arm -M versatilepb -nographic -kernel hello_world.elf
```

- Option Breakdown:
- -M versatilepb: Specifies the VersatilePB machine model.
- -nographic: Disables graphical output; redirects serial I/O to the console.
- -kernel hello_world.elf: Loads your compiled ELF binary as the kernel.

5. View the Output

You should see the following output in your terminal:

```
Hello World
```

Understanding the Modifications

• Memory Addresses:
The memory map of the VersatilePB machine differs from the BeagleBone Black. Adjusting the starting address in the linker script and the stack pointer ensures that your code and data are placed in valid RAM locations.

• UART Registers:
The UART peripheral on the VersatilePB is different from that on the BeagleBone Black. Updating the base address and register offsets allows your code to interact correctly with the emulated UART device.

Limitations and Considerations

• Hardware Differences:
The VersatilePB machine does not emulate the BeagleBone Black's hardware exactly. For simple programs like "Hello World," this is acceptable, but more complex hardware-specific code may not work without further adjustments.

• Instruction Set Compatibility:
The VersatilePB uses an ARM926EJ-S processor (ARMv5 architecture), while the BeagleBone Black uses an ARM Cortex-A8 (ARMv7-A). For basic ARM instructions used in simple programs, this difference is negligible.

• Peripheral Support:
If you plan to emulate more advanced peripherals or need closer hardware matching, consider using other QEMU machine models or simulators.

Advantages of Testing on an Emulator

• No Hardware Dependency:
Develop and test your code without needing the physical BeagleBone Black hardware.

• Faster Development Cycle:
Quickly iterate on your code and test changes immediately.

• Debugging Capabilities:
Utilize powerful debugging tools to diagnose and fix issues.

Limitations

• Hardware Accuracy:
Emulators may not perfectly replicate the behavior of the actual hardware, especially for complex peripherals.

• Performance Differences:
Execution speed and timing may differ from real hardware.

• Learning Curve:
Setting up and configuring emulators requires additional effort and understanding.

A few comments for Mac Users.

If you're on macOS, and pressing Ctrl-a followed by x isn't exiting QEMU as expected. This issue can occur because macOS terminals or shells might handle control characters differently, or they might intercept certain key combinations for their own use.

Here is an alternative method to exit QEMU on macOS:

Method 1: Change the QEMU Escape Character

By default, QEMU uses Ctrl-a as the escape character when running in -nographic mode. You can change this escape character to one that doesn't conflict with macOS key bindings.

Steps:

1. Choose a New Escape Character
   Let's use Ctrl-t (which corresponds to ASCII code 0x14).

2. Start QEMU with the -echr Option
   qemu-system-arm -M versatilepb -nographic -echr 0x14 -kernel hello_world.elf
   • -echr 0x14 sets the escape character to Ctrl-t.

3. Use the New Escape Sequence to Exit
   Press Ctrl-t, then x to exit QEMU.

Explanation: Ctrl-t is less likely to be intercepted by macOS terminal applications.
The -echr option allows you to specify an escape character that suits your environment.

Booting the "Hello World" Program from an SD Card on the BeagleBone Black

Now that your "Hello World" program is working when loaded via U-Boot over the serial console, you can configure the BeagleBone Black to boot your program directly from a microSD card. This method allows the board to run your program automatically on power-up without manual intervention.

Prerequisites

- **Compiled Binary**: You have already compiled your program and have the hello.bin file.
- **microSD Card**: A microSD card (at least 4GB is recommended).
- **Card Reader**: A microSD card reader connected to your development machine.
- **U-Boot Files**: Access to the MLO and u-boot.img files compatible with the BeagleBone Black.
- **Cross-Compiler Toolchain**: The arm-none-eabi- toolchain installed on your development machine.
- **Serial Communication Software**: A terminal program like minicom, screen, Tera Term, or PuTTY for serial console access (optional but recommended for debugging).

1. Building U-Boot (on macOS)

Building U-Boot involves setting up the correct development environment, cloning the U-Boot source code, configuring it for your target (BeagleBone Black), and compiling it. Here's a step-by-step breakdown:

a. Install Homebrew (If Not Already Installed)

Homebrew is a package manager for macOS that simplifies the installation of software.

1. Open Terminal.
2. Install Homebrew by running:

   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

3. Follow the on-screen instructions to complete the installation.
4. Ensure Homebrew is in your PATH:

After installation, you might need to add Homebrew to your PATH. Add the following line to your shell configuration file (~/.zshrc for Zsh or ~/.bash_profile for Bash):

```
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zshrc
source ~/.zshrc
```

Replace ~/.zshrc with ~/.bash_profile if you're using Bash.

b. Install Required Tools and Dependencies

You'll need several tools and libraries to build U-Boot and compile your bare-metal program.

1. Update Homebrew:

   ```
   brew update
   ```

2. Install Essential Packages:

   ```
   brew install gcc arm-none-eabi-gcc pkg-config openssl@3 git make
   ```

   - gcc: GNU Compiler Collection for building U-Boot.
   - arm-none-eabi-gcc: ARM cross-compiler for bare-metal development.
   - pkg-config: Helps in managing compiler and linker flags for libraries.
   - openssl@3: Provides SSL/TLS libraries required by U-Boot.
   - git: Version control system to clone the U-Boot repository.
   - make: Build automation tool.

3. Verify Installations:
   Ensure that the installations were successful by checking their versions:

   ```
   gcc --version
   arm-none-eabi-gcc --version
   pkg-config --version
   openssl version
   git --version
   make --version
   ```

Example Output:

```
gcc (Homebrew GCC 12.2.0) 12.2.0
arm-none-eabi-gcc (GCC) 12.2.0
pkg-config 0.29.2
OpenSSL 3.0.7  7 Sep 2023
git version 2.41.0
GNU Make 4.3
```

c. Clone the U-Boot Repository

1. Choose a Directory for the U-Boot Source Code:

Navigate to your preferred development directory or create a new one:

```
mkdir -p ~/development/u-boot
cd ~/development/u-boot
```

2. Clone the U-Boot Repository:

```
git clone https://source.denx.de/u-boot/u-boot.git
```

3. Navigate into the Cloned Repository:

```
cd u-boot
```

4. Check Out a Specific Release (Optional but Recommended):

It's often best to build from a stable release rather than the latest commit.

```
git checkout v2023.10
```

Replace v2023.10 with the desired stable version. You can list available tags with:

```
git tag
```

d. Configure Environment Variables

Properly setting environment variables ensures that the build system can locate necessary tools and libraries.

1. Set HOSTCC to Use clang:

This tells the build system to use clang for compiling host tools.

export HOSTCC=clang

Verify:

echo $HOSTCC

Expected Output:

clang

2. Configure pkg-config to Locate OpenSSL:

Ensure that pkg-config can find OpenSSL's .pc files.

export PKG_CONFIG_PATH="/opt/homebrew/opt/openssl@3/lib/pkgconfig:$PKG_CONFIG_PATH"

Create a Symlink for openssl.pc (If Necessary):
U-Boot may look for openssl.pc. Create a symbolic link if it doesn't exist.

ln -sf /opt/homebrew/opt/openssl@3/lib/pkgconfig/openssl@3.pc /opt/homebrew/opt/openssl@3/lib/pkgconfig/openssl.pc

Verify pkg-config Can Locate OpenSSL:

pkg-config --cflags openssl

Expected Output:

-I/opt/homebrew/Cellar/openssl@3/3.4.0/include

3. Set Compiler and Linker Flags for Host Tools:

These flags ensure that the host compiler (clang) can locate OpenSSL headers and libraries.

export HOSTCFLAGS="$(pkg-config --cflags openssl)"
export HOSTLDFLAGS="$(pkg-config --libs openssl)"

Verify the Flags:
echo $HOSTCFLAGS
echo $HOSTLDFLAGS

Expected Output:

-I/opt/homebrew/Cellar/openssl@3/3.4.0/include
-L/opt/homebrew/Cellar/openssl@3/3.4.0/lib -lssl -lcrypto

e. Clean Previous Builds

Before starting a fresh build, it's crucial to clean any previous build artifacts that might cause conflicts.

1. Run make clean:

   make clean

   Note: If you encounter errors like rm: SPL: is a directory, manually remove problematic directories.

2. Manually Remove the SPL Directory (If Present):

   rm -rf SPL

3. Run make mrproper:
   This command performs a more thorough clean, removing all generated files including configuration files.

   make mrproper

   Caution: make mrproper will delete your .config file. You'll need to reconfigure after this step.

f. Configure U-Boot for BeagleBone Black

1. Set the Default Configuration for BeagleBone Black:

   make am335x_evm_defconfig

   Explanation:

   - am335x_evm_defconfig is the default configuration for BeagleBone Black (AM335x SoC).
   - This command generates the .config file based on the specified configuration.

2. Verify Configuration (Optional):

You can inspect the generated .config file to ensure it has the expected settings.

```
less .config
```

g. Build U-Boot

With the environment set up and configuration in place, proceed to build U-Boot.

1. Start the Build Process:

   ```
   make V=1 all
   ```

   Explanation:

   - V=1 enables verbose output, which is helpful for debugging.
   - all specifies that you want to build all targets.

2. Alternatively, Pass Environment Variables Inline:

   ```
   make V=1 all HOSTCC=clang HOSTCFLAGS="$(pkg-config --cflags openssl)"
   HOSTLDFLAGS="$(pkg-config --libs openssl)"
   ```

3. Monitor the Build Output:

Ensure that U-Boot compiles without errors. Key output files to look for include:

- MLO: Secondary Program Loader (SPL).
- u-boot.img: Main U-Boot image.
- u-boot.bin: Binary version of U-Boot.

Example Successful Build Output:

```
CC     common/version.o
AR     common/built-in.o
LD     u-boot
OBJCOPY u-boot-nodtb.bin
MKIMAGE u-boot.img
...
```

Final Build Confirmation:
After the build completes, verify that the necessary files are present:

ls -l MLO u-boot.img u-boot.bin

Expected Output:

```
-rw-r--r--  1 user  staff  123456 Jul 10 12:34 MLO
-rw-r--r--  1 user  staff  234567 Jul 10 12:34 u-boot.img
-rw-r--r--  1 user  staff  345678 Jul 10 12:34 u-boot.bin
```

2. Preparing the SD Card for Booting

Once U-Boot is built, you'll need to prepare an SD card with the necessary bootloader files to boot your BeagleBone Black.

a. Format the SD Card

Warning: Formatting will erase all data on the SD card. Ensure you've backed up any important data.

1. Insert the SD Card into your Mac's SD card reader.
2. Identify the SD Card Device:

   diskutil list

   Example Output:

   ```
   /dev/disk2 (external, physical):
     #:                       TYPE NAME                    SIZE       IDENTIFIER
     0:      FDisk_partition_scheme                        *16.0 GB    disk2
     1:               Windows_FAT_32 BOOT                   15.9 GB    disk2s1
   ```

   - Note: In this example, /dev/disk2 is the SD card. Ensure you correctly identify your SD card to avoid data loss.

3. Unmount the SD Card:

   diskutil unmountDisk /dev/disk2

   Replace /dev/disk2 with your SD card's identifier.

4. Erase and Format the SD Card as FAT32:

   sudo diskutil eraseDisk FAT32 BOOT MBRFormat /dev/disk2

   Parameters:

   - FAT32: Filesystem type.
   - BOOT: Volume name (you can choose any name).
   - MBRFormat: Master Boot Record partition scheme.

   Example Output:

   Erasing all partitions on disk2
   Completely erased disk2
   Created partition BOOT as FAT32 with 15.9 GB.

b. Copying U-Boot Files to the SD Card

1. Mount the SD Card (If Not Automatically Mounted):

   diskutil mountDisk /dev/disk2

   • The SD card should now be accessible at /Volumes/BOOT/.

2. Navigate to Your U-Boot Build Directory:

   cd ~/development/u-boot/u-boot

   Adjust the path if your U-Boot directory is different.

3. Copy MLO and u-boot.img to the Boot Partition:

   cp MLO /Volumes/BOOT/
   cp u-boot.img /Volumes/BOOT/

   Verify the Copy:

   ls -l /Volumes/BOOT/

   Expected Output:

```
-rw-r--r--  1 user  staff  123456 Jul 10 12:34 MLO
-rw-r--r--  1 user  staff  234567 Jul 10 12:34 u-boot.img
```

4. Eject the SD Card Safely:

   diskutil eject /dev/disk2

   Replace /dev/disk2 with your SD card's identifier.

3. Deploying and Running Your "Hello World" Program

With U-Boot and your bare-metal program ready, the next steps involve copying the program to the SD card and configuring U-Boot to execute it.

a. Copying the Program to the SD Card

1. Mount the SD Card:
   Insert the SD card into your Mac. It should mount automatically at /Volumes/BOOT/.
   If not, manually mount it:

   diskutil mountDisk /dev/disk2

   Replace /dev/disk2 with your SD card's identifier.

2. Navigate to Your Bare-Metal Program Directory:

   cd ~/development/baremetal/hello

3. Copy the Compiled Binary to the Boot Partition:

   cp hello.bin /Volumes/BOOT/

4. Eject the SD Card Safely:

   diskutil eject /dev/disk2

   Replace /dev/disk2 with your SD card's identifier.

b. Configuring U-Boot to Load and Execute the Program

To instruct U-Boot to load your bare-metal program from the SD card and execute it, you can create a U-Boot script (boot.scr) .

1. Create a U-Boot Command Script (boot.cmd):
   Navigate to your U-Boot build directory or any temporary location:

   cd ~/development/u-boot/u-boot

   Create boot.cmd with the following content:

   fatload mmc 0:1 0x82000000 hello.bin
   go 0x82000000

   Explanation:

   • load mmc 0:1 0x82000000 /hello.bin: Loads hello.bin from the first partition of the first MMC device (SD card) into memory address 0x82000000.

   • go 0x82000000: Jumps to the loaded program's address to execute it.

2. Convert boot.cmd to boot.scr Using mkimage:

   mkimage -C none -A arm -T script -d boot.cmd boot.scr

Parameters:

- -C none: No compression.
- -A arm: Architecture set to ARM.
- -T script: Image type set to script.
- -d boot.cmd: Input file.
- boot.scr: Output script file.

3. Copy boot.scr to the Boot Partition:

   cp boot.scr /Volumes/BOOT/

   Verify the Copy:

   ls -l /Volumes/BOOT/boot.scr

4. Eject the SD Card Safely:

   diskutil eject /dev/disk2

4. Deploying and Running Your “Hello World” Program

With your SD card prepared and U-Boot configured to load your program, follow these steps to deploy and run your bare-metal “Hello World” program on the BeagleBone Black.

a. Insert the SD Card and Power On

1. Eject the SD Card from Your Mac (If Not Already Ejected):

   diskutil eject /dev/disk2

2. Insert the SD Card into the BeagleBone Black:
   - Locate the SD card slot on the BeagleBone Black.
   - Insert the SD card firmly.

3. Connect the Power Supply:
   - Power off the BeagleBone Black before inserting the SD card.
   - After inserting, connect the power supply to boot the device.

b. Monitor the Boot Process via Serial Console

1. Connect to the Serial Console:
   - Use a USB-to-serial adapter connected to your Mac.
   - Open your serial terminal software (e.g., CoolTerm, screen).

Example with screen:

screen /dev/tty.usbserial-XXXX 115200

Replace /dev/tty.usbserial-XXXX with your serial device identifier (found via ls /dev/tty.* before and after connecting the adapter).

2. Observe the Boot Messages:
   - You should see U-Boot booting messages.
   - If you set up boot.scr, U-Boot should automatically load and execute your program.

3. Verify Program Execution:
   - If Your Program Sends Output via UART:
     - You should see “Hello World” printed in the serial console.

5. Directory Structure on the SD Card

Understanding the directory structure on the SD card helps in organizing files and troubleshooting.

a. Boot Partition (FAT32):

Mounted at /Volumes/BOOT/ on macOS.

```
BOOT/
├── MLO
├── u-boot.img
├── boot.scr
├── hello.bin
```

- MLO: Secondary Program Loader (SPL).
- u-boot.img: Main U-Boot image.
- boot.scr: U-Boot script to load and execute the bare-metal program.
- hello.bin: Your compiled bare-metal “Hello World” program.