ATmega32U4 Flashing firmware for USB

# Required Hardware

###  AVR USBASP ISP Programmer with 10 Pin Cable

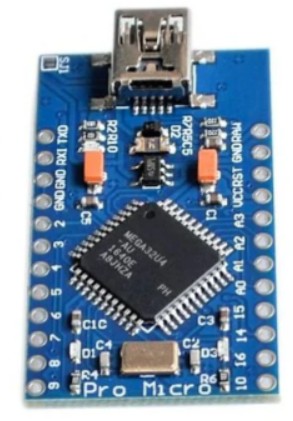


**fig: 1 AVR USBASP ISP Programmer**

The AVR USBasp ISP Programmer with a 10-pin cable is used to upload USB

firmware to Atmel ATmega32U4 devices. This programmer allows you to easily flash firmware directly onto the microcontroller without removing it from your project. It's compatible with various programming software and is simple to

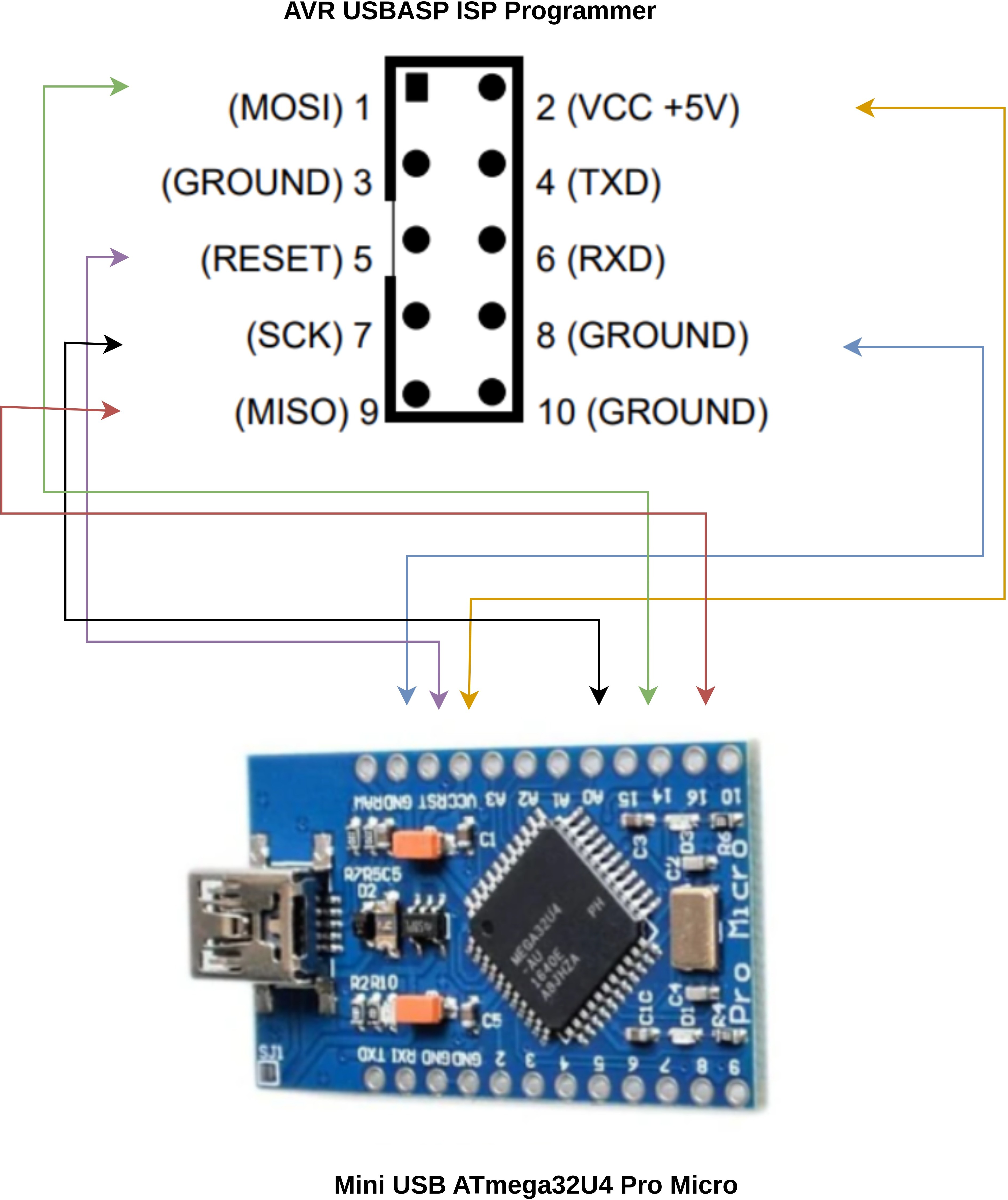
use.The 10-pin cable provides a strong connection for programming, helping you update and develop your projects with the ATmega32U4 quickly and easily.



### fig: 1.1 Mini USB ATmega32U4 Pro Micro 5V 16MHz Board ModuleThe "Mini USB

**ATmega32U4 Pro Micro 5V 16MHz Board Module"** is a compact development board based on the ATmega32U4 microcontroller.

# Pin Connection



### fig: 1.2 Mini USB ATmega32U4 AVR USBASP ISP Programmer Pin Connection

VCC to VCC GROUND to GROUND RESET to RESET MISO to 14

MOSI to 16

SCK to 15

# Required Software

 **Firmware**

###  Kernel Driver

 **User Application**

 **Wireshark**

### The firmware named [00\_microcontroller\_firmware](https://github.com/Johannes4Linux/USB_Tutorial/tree/main/00_microcontroller_firmware) includes basic

**functionality for USB control transfers, along with bulk in and bulk out endpoints.**

**Control Transfer:** This is used for device setup and management, allowing you to send commands and receive status information.

**Bulk In Endpoint:** This endpoint is used for transferring data from the device to the host, ensuring reliable communication for larger data sets.

**Bulk Out Endpoint:** This endpoint allows data to be sent from the host to the device, enabling updates or commands.

You can modify this firmware to enhance or customize the USB communication according to your requirements.

### The [01\_kernelspace\_driver](https://github.com/Johannes4Linux/USB_Tutorial/tree/main/01_kernelspace_driver) refers to a basic driver that operates in kernel space for interfacing with hardware or devices in a Linux environment. Here are the key features and functions typically associated with this kind of driver:

**00\_usb\_hello** Introduced a basic Linux driver focused on writing to the Bulk Out endpoint. This serves as a foundational step for USB communication.

**01\_control\_msg** Expanded the functionality by adding a driver that can also write to the Bulk Out endpoint, enhancing control over data transfers.

**02\_bulk\_out\_msg** Further developed the Bulk Out endpoint capabilities, likely improving the handling and efficiency of data transmission.

**03\_bulk\_in\_msg** Added a driver for accessing the USB Bulk In endpoint, enabling data retrieval from the device. This marks a significant advancement,

allowing for full-duplex communication.

### In [02\_userspace\_apps](https://github.com/Johannes4Linux/USB_Tutorial/tree/main/02_userspace_apps), there are two applications:

 **Host Application** This program sends random characters to the USB device via the Bulk Out endpoint. The user can input a character, or the application generates one automatically.

 **Device Application** This runs on the device and responds to the host by sending a predefined string back through the Bulk In endpoint after receiving a character.

Together, these applications enable simple communication between the host and the device for testing and interaction.

 Wireshark is a powerful network protocol analyzer that can capture and

display USB communication data in real-time. By connecting your USB device and using Wireshark, you can monitor the data packets being sent and received, allowing you to verify that the communication is working properly.

This tool helps in debugging by providing insights into the USB protocol interactions, making it easier to identify issues or confirm correct data

transmission.

# Build code and flash

## USB Bulk In Endpoint Firmware Build Process

### Directory Structure:

Located in .

~/workspace/USB\_Tutorial-main/00\_microcontroller\_firmware/03\_bulk\_in\_ep

Contains the following files:

 The main firmware source code.

main.c

 Instructions for building the firmware.

Makefile

 Directory containing additional library files.

lib/

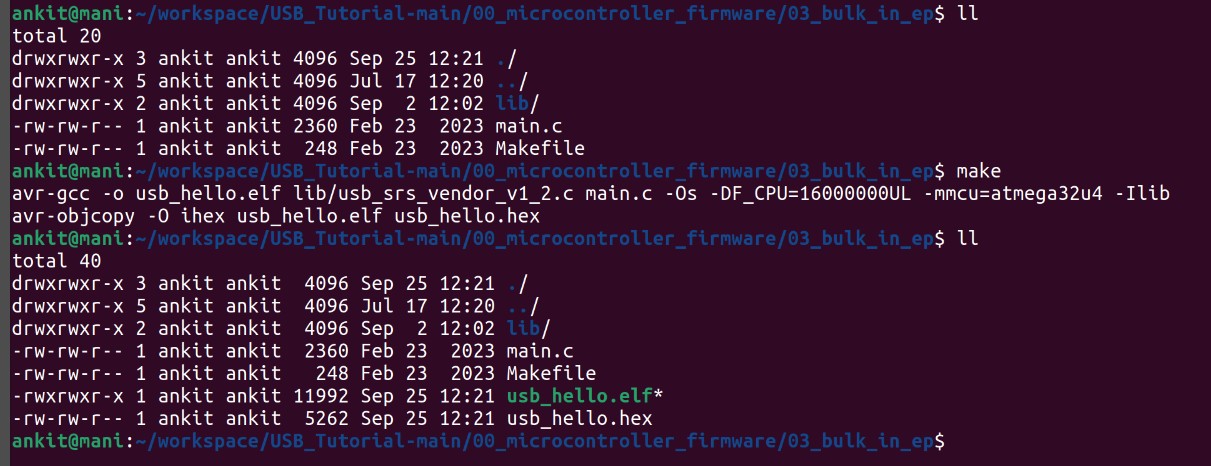
### Build Steps:

 **Make Command**:

Run in the terminal to compile the firmware.

make

The command used:



make

connect the AVR USBASP ISP Programmer and flash the firmware on device

### Command:

sudo avrdude -c usbasp -p m32u4 -U flash:w:usb\_hello.he

**Components**:

  Runs the command with superuser privileges, necessary for accessing hardware directly.

**sudo**

  The program used for programming AVR microcontrollers.

**avrdude**

  Specifies the programmer being used. In this case, commonly used USB programmer for AVR chips.

**c usbasp**

usbasp

is a

  Specifies the type of microcontroller. Here, ATmega32U4.

**p m32u4**

m32u4

refers to the

  This option indicates a memory operation:

**U flash:w:usb\_hello.hex**

 Specifies that you are working with the flash memory of the microcontroller.

**flash**

 Stands for "write."



**w**

 The HEX file to be written to the microcontroller's flash

**usb\_hello.hex**

memory.

## Output Explanation

 :

**avrdude: AVR device initialized and ready to accept instructions**

Indicates that the programmer successfully communicated with the microcontroller.

 :

**Device signature = 0x1e9587 (probably m32u4)**

Confirms the identity of the microcontroller as ATmega32U4 by reading its unique signature.

 :

**NOTE: "flash" memory has been specified, an erase cycle will be performed**

Indicates that before writing new data, the existing flash memory will be erased.

 :

**avrdude: erasing chip**

The process of erasing the flash memory has started.

 :

**reading input file "usb\_hello.hex"**

is reading the specified HEX file to gather the data to be written.

avrdude

 :

**input file usb\_hello.hex auto detected as Intel Hex**

Confirms that the format of the HEX file is recognized, ensuring compatibility.

 :

**writing flash (1864 bytes):**

Specifies that the firmware of size 1864 bytes is being written to the microcontroller.

 :

**Writing | ################################################## | 100% 0.72s**









Indicates that the writing process completed successfully, with a visual progress bar.

:

**avrdude: verifying flash memory against usb\_hello.hex:**

After writing, the program verifies that the written data matches the original HEX file.

:

**1864 bytes of flash verified**

Confirms that all written bytes were verified correctly against the HEX file, ensuring data integrity.

:

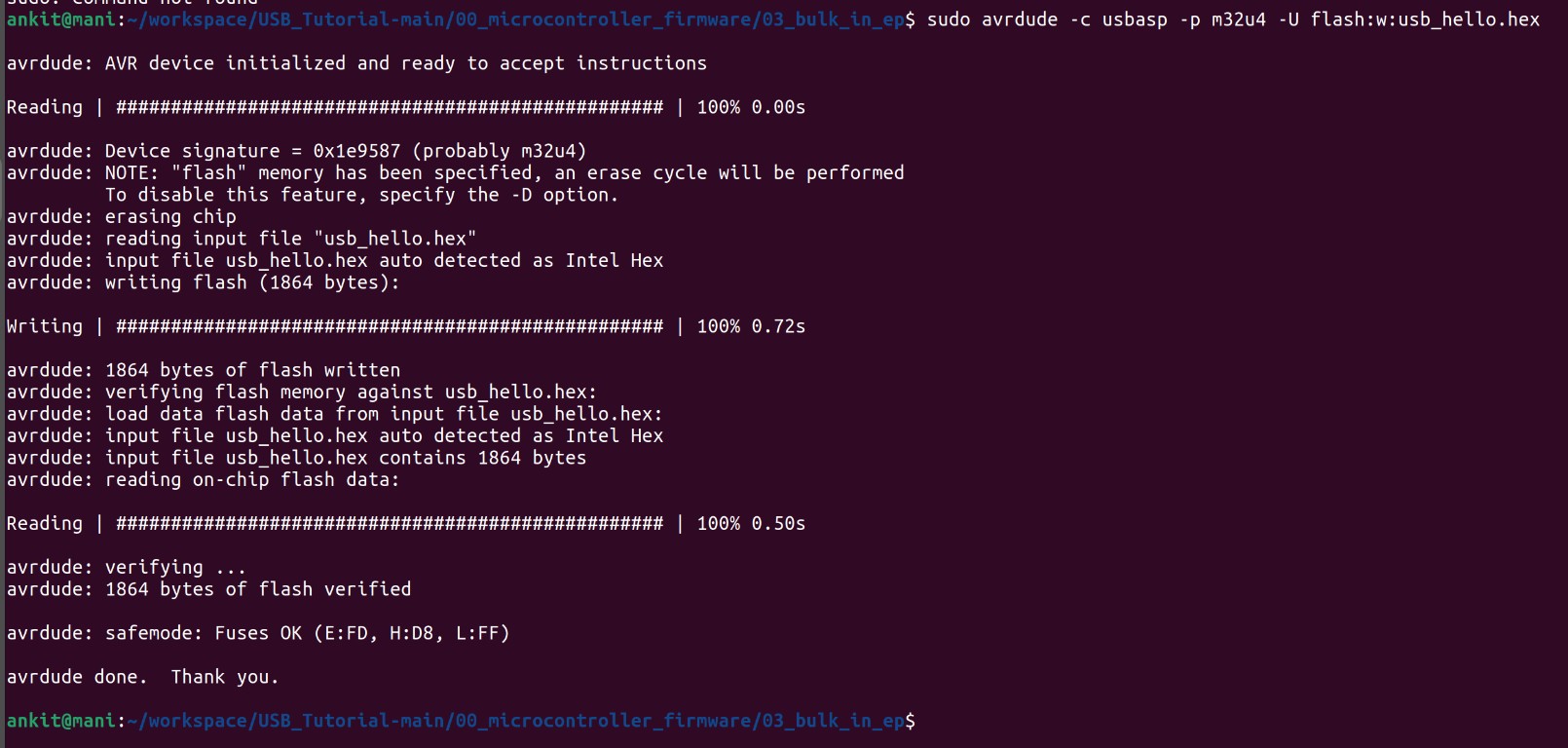
**safemode: Fuses OK (E:FD, H:D8, L:FF)**

Indicates that the fuse settings (configuration bits) for the microcontroller are correct.

:

**avrdude done. Thank you.**

The process is complete, and there were no errors during the upload.



## Directory Structure

You are working in the directory:

~/workspace/USB\_Tutorial-main/01\_kernelspace\_driver/02\_bulk\_o ut\_msg

The contents include:

 Contains build instructions for compiling the driver.

**Makefile**

 The source code for your USB device driver.

**my\_usb\_devdrv.c**

## Build Process

###  Run Make Command:

make

 **Make Output Breakdown**:

**make -C /lib/modules/6.5.0-44-generic/build M=/home/ankit/workspace/USB\_Tutorial-**

This command tells

:

to compile the driver against the current

**main/01\_kernelspace\_driver/02\_bulk\_out\_msg modules**

make

kernel's source located at

/lib/modules/6.5.0-44-generic/build

module's source in your specified directory.

###  Compiler Warning:

, using the

 Indicates that

**warning: the compiler differs from the one used to build the kernel**

the GCC version you are using to compile the driver may not exactly match the version used to compile the kernel. This could potentially lead to compatibility issues, although it often works fine.

###  Compiling the Driver:

**CC [M] /home/ankit/workspace/USB\_Tutorial-**

The source file

.

my\_usb\_devdrv.o

:

is being compiled into an object file

**main/01\_kernelspace\_driver/02\_bulk\_out\_msg/my\_usb\_devdrv.o**

my\_usb\_devdrv.c

**LD [M] /home/ankit/workspace/USB\_Tutorial-**

:

**main/01\_kernelspace\_driver/02\_bulk\_out\_msg/my\_usb\_devdrv.ko**

Links the object file to create the kernel module ( be loaded into the kernel.

 **BTF Generation**:

file), which can

 Indicates that the

.ko

**Skipping BTF generation... due to unavailability of vmlinux**

BPF Type Format BTF generation is skipped because the necessary vmlinux file (which contains kernel symbols) is not available. This is typically not critical for building the driver.

###  Directory Listing After Build:

ll

Shows the new files generated by the build process:

 The compiled kernel module ready to be loaded into the

**my\_usb\_devdrv.ko**

kernel.

**modules.order**

### Various

**,**

### and

**Module.symvers**



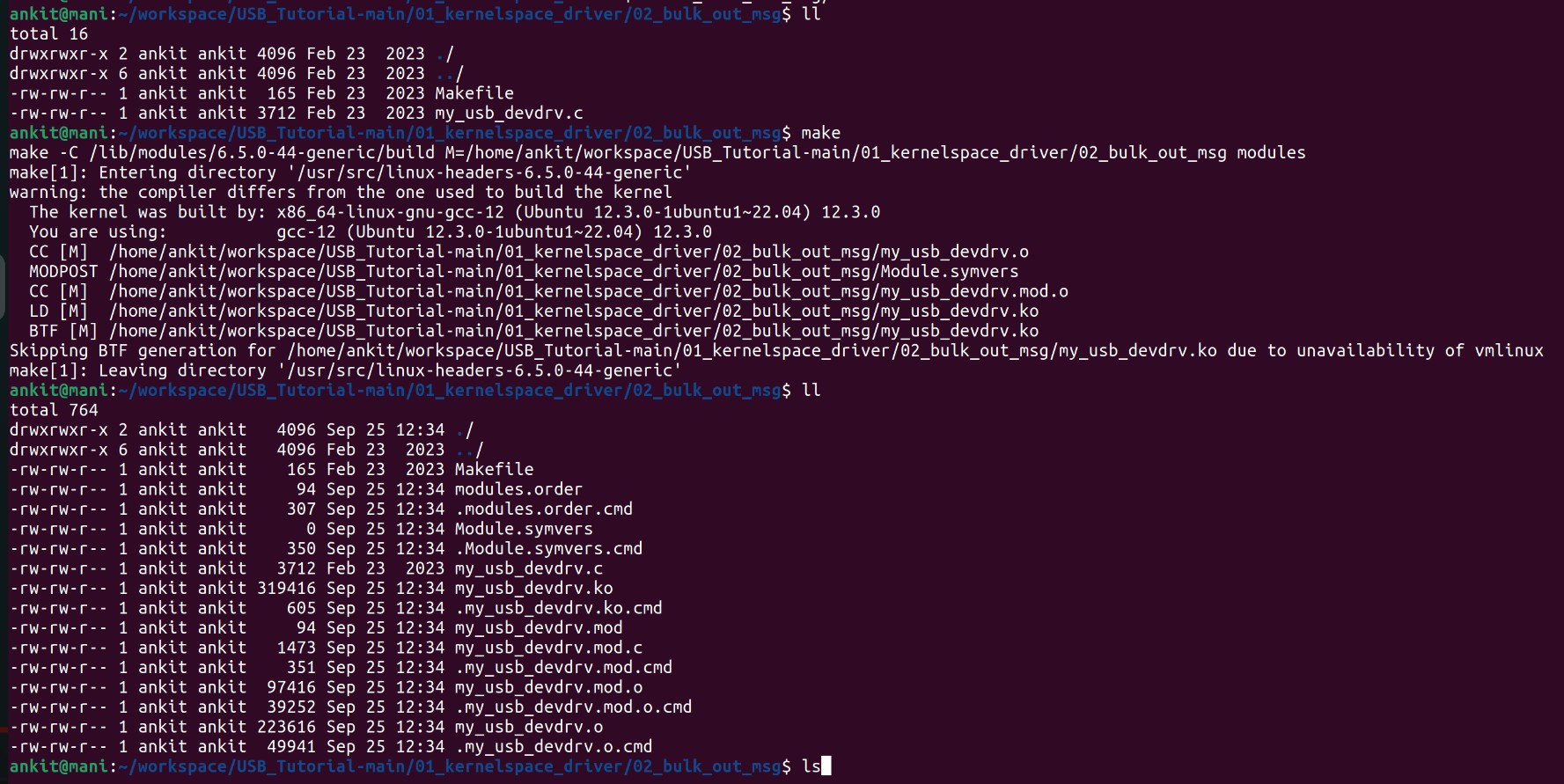
**.o**

 Files that help manage module dependencies.

**files** Intermediate and metadata files created during

**.mod**

compilation.



### Insert Module:

sudo insmod my\_usb\_devdrv.ko

This command uses

to insert the kernel module

into the

Linux kernel. Running it with kernel modules.

insmod

my\_usb\_devdrv.ko

sudo

provides the necessary permissions to load

lsmod | grep my\_usb\_devdrv

This command lists all currently loaded kernel modules and uses to filter the

grep

results for .

my\_usb\_devdrv

my\_usb\_devdrv

12288 0

 This is the name of your kernel module, confirming that it was successfully loaded into the kernel.

 This number represents the size of the module in bytes. In this case, is 12,288 bytes in size.

**my\_usb\_devdrv**

**12288**

my\_usb\_devdrv

 This indicates the number of instances of the module currently being used



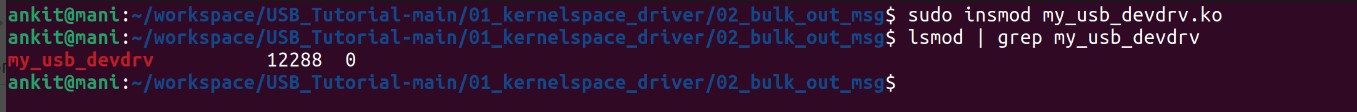
**0**



0

(i.e., how many processes are using this module). A value of other processes are currently using the module.

means that no



## 1. Change Directory User Application

cd /home/ankit/workspace/USB\_Tutorial-main/02\_userspace\_apps/ C/ankit

ll

This command shows the contents of your current directory, which includes:

 Source code file for your second application.

**ankit2.c**

 Source code file for your first application.

**ankit.c**

## Compiling the Application

compiled using:

ankit2.c

gcc ankit2.c -o ankit2 -lusb-1.0

 The GNU Compiler Collection, used to compile C programs.

 The source file you are compiling.

**gcc**

**ankit2.c**

 This flag specifies the name of the output executable file (in this

**o ankit2**

case,

).

 This flag links the USB library (libusb-1.0, allowing your application

ankit2

**lusb-1.0**

to interface with USB devices.

## Running the Application

sudo ./ankit2

 This command runs the program with superuser privileges, which is often necessary for USB device access.

## Output Explanation

**sudo**

The output shows repeated lines:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Received | data: | 46 | 72 | 6f | 6d | 20 | 53 | 6c | 76 |
| Received | data: | 46 | 72 | 6f | 6d | 20 | 53 | 6c | 76 |
| Received | data: | 46 | 72 | 6f | 6d | 20 | 53 | 6c | 76 |
| ... |  |  |  |  |  |  |  |  |  |

Each line indicates that your application has received data from the USB device.

The hexadecimal data (

46 72 6f 6d 20 53 6c 76

Slv", wrriten in firmware code.

For stop the data generating used

) corresponds to the string "From

to interrupt the program, which is a



^C

common way to stop a running command in the terminal.

