Implementing Full Device Cloning on the NVIDIA Jetson Platform

March 2, 2018

Calvin Chen
Linh Hoang
Connor Smith

Advisor: Professor Mark Claypool
Sponsor - Mentor : NVIDIA - Jimmy Zhang

Abstract

NVIDIA's Linux For Tegra development team provides support for many tools which exist on the Jetson-TX2 embedded AI computing device. While tools to clone the user partitions of the Jetson-TX2 exist, no tool exists to fully backup and restore all of the partitions and configurations of these boards. The goal of this project was to develop a tool that allows a user to create a complete clone of the Jetson-TX2 board's partitions and their contents. We developed a series of bash scripts that fully backup the partitions and configurations of one Jetson-TX2 board and restore them onto another board of the same type.

Table of Contents

1 Introduction

The Jetson-TX2 is a low power, high performance embedded artificial intelligence supercomputer utilized in many modern technologies. Because of its compact size, many manufacturers have utilized the Jetson-TX2 in products such as drones, cameras, robots, and gaming devices. The Jetson board comes prepackaged with Linux for Tegra along with other tools users may need to start developing their technologies [16].

There is a need for a cloning tool for the Jetson-TX2 that would allow users to easily and effectively backup board configurations, partitions, and bootloaders from one target board and restore them onto another target board. Although it is possible to manually set board configurations, the process is time consuming. This tool would allow developers and consumers to duplicate board configurations efficiently along with user partitions and bootloaders. This would allow developers to clone many boards with the same specifications for rapid testing and development.

This project assisted NVIDIA to create a backup and restore tool that allows users to fully clone Jetson-TX2 boards. We first conducted background research to familiarize ourselves with the Jetson platform and to learn more about the tools necessary to complete this project. We then collaborated with NVIDIA to design and implement a solution through the duration of the MQP. After implementing our basic design, we added additional features to allow for more user customization while running the scripts. Then, we executed a series of tests to ensure that the backup and restore scripts functioned properly and created the exact copy we desired. The final product which we presented to NVIDIA was two scripts, one for backup and one for restore, and a more powerful initial ramdisk (initrd) that allows for the cloning process to be completed. The initrd mode allowed the boards to maintain integrity during the backup and restore process by preventing files from being modified while executing the scripts.

This report first delves into the background research the team completed to gain a better understanding of the Jetson-TX2 in Chapter 2. In Chapter 3, we described the process our team took to design and implement the cloning tool. Chapter 4 elaborates further upon the backup and restore scripts and the modifications made to the initial ramdisk, as well as documenting use cases of the tool. Chapter 5 discusses the ways in which NVIDIA can further improve upon the tool. Chapter 6 provides a conclusion to the work we have done at NVIDIA.

2 Background

In order to effectively complete the project, we researched the Jetson-TX2 board, Linux technologies, and supporting tools.

2.1 Preliminary Research

We researched several technologies and concepts to gain insight into how we would complete our task. The main topics that we focused on were flashing, booting, and file systems; all of which are important components to completing the project. This section delves into the details of each of the topics.

2.1.1 UEFI Booting and Flashing

The Unified Extensible Firmware Interface (UEFI) is a specification that defines a software interface between an operating system and proprietary or open source platform firmware. UEFI is developed to replace the Basic Input/Output System (BIOS) firmware interface, which is used on all IBM PC-compatible computers [2].

The interface defined by the EFI specification includes data tables that contain platform information, as well as boot and runtime services that are available to the operating system and its loader. Figure 1 shows the position of the EFI in the software stack of a computer. UEFI firmware provides some advantages over the traditional BIOS scheme [3]. These include being able to support large disks up to two terabytes in size with a GUID partition table (GPT) format. Additionally, the firmware is CPU independent and provides a flexible pre-operating system environment that includes network capability. Its modular design and backward and forward compatibility make it especially useful.

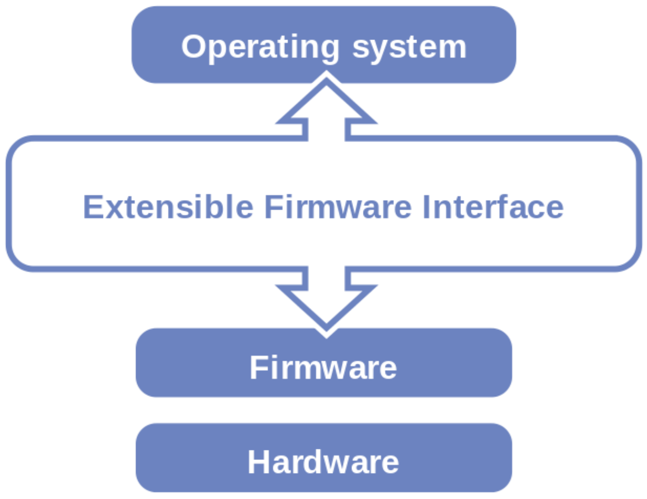


Figure 1: EFI in computer [1]

A device which supports UEFI boot usually has a storage partition formatted according to the UEFI, which is typically a File Allocation Table (FAT) file system. UEFI can run the *UEFI binaries* which resides on the EFI partition. UEFI binaries include bootloaders, drivers and applications. For example, GNU GRUB is an operating system loader which resides on the EFI partition, whose features include a way to choose the loading operating systems and change the boot loading image. Figure 2 provides an example of how EFI binaries can be run during the bootloading process.

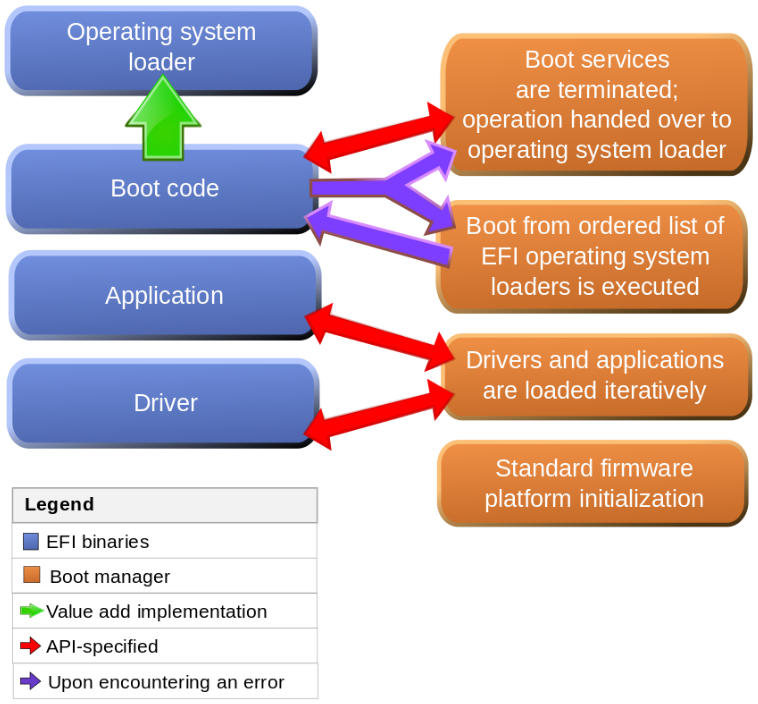

Operating system loader
Boot services are terminated; operation handed over to operating system loader
Boot code
Boot from ordered list of EFI operating system loaders is executed
Application
Drivers and applications are loaded iteratively
Driver
Standard firmware platform initialization
Legend
EFI binaries
Boot manager
Value add implementation
API-specified
Upon encountering an error

Figure 2: UEFI interactions with drivers [1]

The NVIDIA Jetson-TX2 boots using the UEFI booting process. Broadly, the TX2 loads the Boot Configuration Table (BCT) from NVRAM, the on-board eMMC, and runs the bootloader, whose location is specified in the BCT. The Tegra bootloader supports two boot modes: cold boot mode and USB recovery mode (RCM). The Jetson-TX2 can be booted into RCM mode using a sequence of button inputs. In this mode, the board can be booted from binaries loaded from another computer or from the network and can be flashed with a new OS. NVIDIA provides a bash script, `flash.sh`, with the Tegra BSP to interact with the RCM mode. The script can read and restore partitions on the TX2, flash a new Linux Kernel onto the board, or set up the board for a network boot. These functions are internally provided by a compiled program.

The boot process and flashing are especially important to our project because we needed to determine the best way to safely backup files so that files were not being backed up while also being modified. The `flash.sh` script was especially helpful to take inspiration from while determining that RCM boot was the best boot type to use.

2.1.2 Tegra Linux Driver Package

The Tegra Linux Driver Package is a tarball distributed by NVIDIA containing a set of file systems, software binaries and scripts to configure and manage the bootloader, the boot configuration, and the Linux system on the Jetson devices. In this package, the parts which are relevant to our project include the `flash.sh` utility and the initrd binary.

`Flash.sh` is the main script in the driver packages, which comprises of many functionalities. Two of the functionalities are crucial to our project: enabling RCM boot mode and building the initial ramdisk (initrd). In the driver package, there is a precompiled initrd binary. This binary does not provide enough functionalities to the backup and restore processes, so we have to inject more binaries into the initrd. RCM Boot and initrd are explained in further detail in section 2.2.

2.1.3 Network File System

A network file system is a file system abstraction over a network that allows a client to remotely access files over a network. It works similarly to the local file system, but does not require the client and server to be on the same physical machine, and allows for multiple users to access the same file system. NFS is generally only used when both the client and server are running Linux.

Figure 4 shows the general architecture of NFS. NFS follows the client-server model of computing. The server is where the file system is implemented and shared, and the clients then attach to the server. The clients themselves implement the user interface that displays the shared file system, which is mounted within the client's local file space.

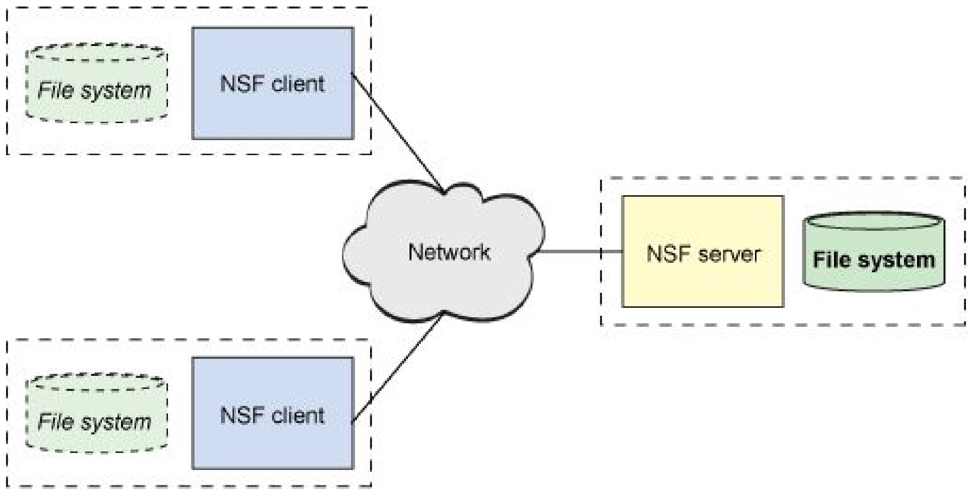


Figure 4: NFS Client-Server Architecture [5]

Some of the newer versions of NFS (versions 4 and 4.1) have introduced key elements to NFS that make it simpler and more powerful. NFSv4 simplified the process for mounting and other aspects of file management by removing support for UDP as a transport protocol so that TCP is the main form of transport. Additionally, version 4 provided more support for various other operating systems. NFSv4.1 introduced the concept of parallel NFS (pNFS) that increases the scalability of NFS and allows for better performance. Figure 5 shows the architecture among pNFS, and its differences from typical NFS.

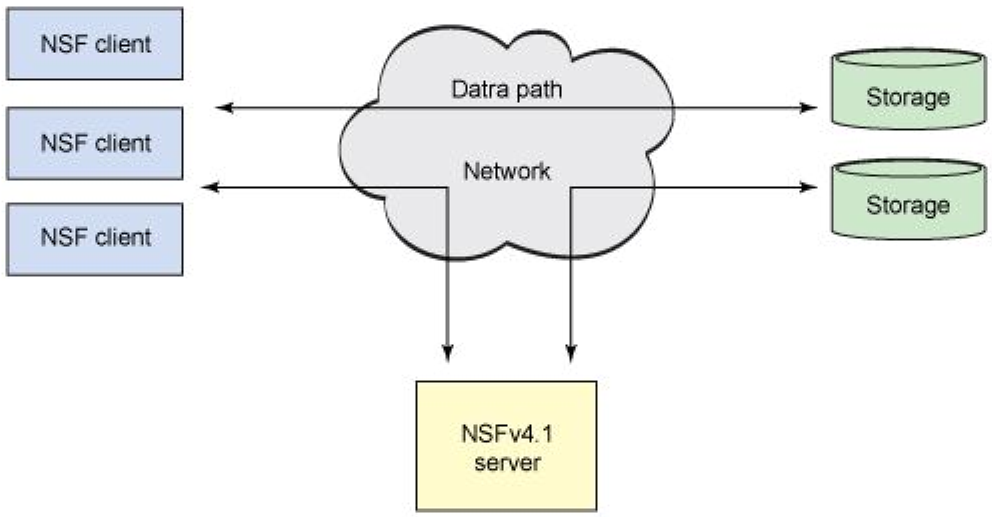


Figure 5: pNFS Architecture [5]

When a client requests file access, the server responds with a layout that describes the mapping of the file to the storage devices. Once the client has access to the mapping, it can directly access the storage without requiring further contact with the server. This allows for increased scaling and performance.

For the Jetson-TX2, NFS is one utility that is used to flash a version of Linux onto the board. Due to the limitations of our project, we were unable to create a way to support NFS flashing with the backup and restore scripts. However, it is possible to do this type of flashing, and this is further elaborated on in the Future Work chapter.

2.1.4 Jetson TX2 Bootloader

The Jetson TX2 bootloader is divided into several components specifically targeted for the Jetson [4]. Bootflow of the bootloader starts from the BootROM which is passed onto Microboot and then onto the Tegraboot. Each of these bootloaders completes necessary tasks before initializing the next step in the Bootflow process. Figure 6 shows the shows the flow of control within the boot software.

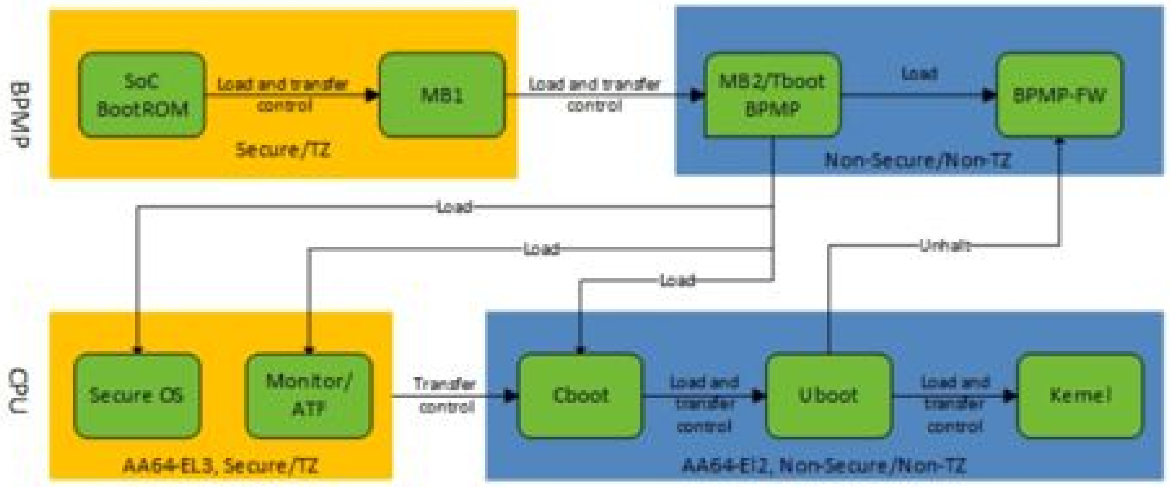


Figure 6: Flow of Control in Boot Software [4]

Bootflow is initialized with the BootROM. BootROM is a hard-wired component on the Tegra board. It is used to initialize Boot Media which contains many copies of the BootROM

Boot Configuration Table (BR-BCT). The BR-BCT stores the configuration parameters that the BootROM uses to initialize hardware. Additionally, BootROM also initializes Microboot1, and then pass control over to Microboot1.

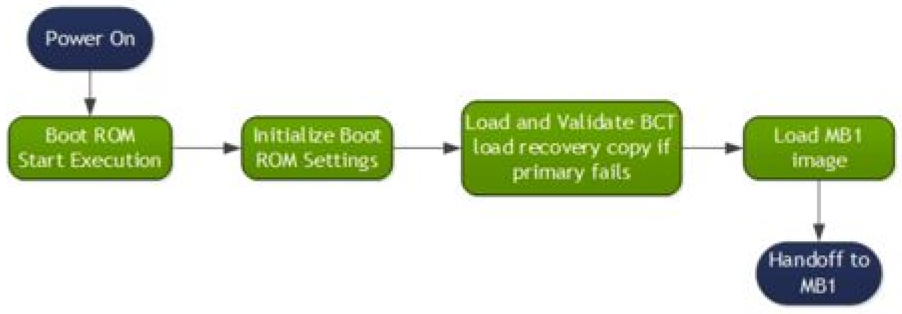

Power On
Boot ROM Start Execution
Initialize Boot ROM Settings
Load and Validate BCT load recovery copy if primary fails
Load MB1 image
Handoff to MB1

Figure 7: BootROM flow [4]

Microboot1 (MB1) is an extension of the BootROM and provides flexibility to alter the boot sequence. Since it is a component of BootROM, MB1 is encrypted by an NVIDIA-owned key. MB1 is responsible for many tasks such as firmware initialization and loading the next stage of the Bootflow, TegraBoot.

TegraBoot is split into two portions, TegraBoot-BPMP (MB2) and TegraBoot-CPU. Furthermore, TegraBoot-BPMP is split into two variants, one of which is used for cold boot and another that is used for recovery boot. It is responsible for loading and initializing firmware components and completing CPU initialization. Once the CPU initialization is complete, the bootloader passes control over to the BPMP-FW. The TegraBoot also supports flashing in addition to RCM which is the MB2 recovery boot.

Another component of Bootflow is CBoot. CBoot is the primary CPU bootloader utilized on mobile platforms in a cold boot. CBoot is what boots the kernel. It supports display, boot logo, and verified boot. CBoot is based on the Little Kernel (LK) open source bootloader [4]. CBoot also utilizes the interrupt and scheduling frameworks of LK. Once the CBoot completes the loading process, it will pass control over to the U-Boot which is the default bootloader for the Tegra Linux Driver Package.

The deficiency in the default Jetson-TX2 bootloader is that because it is a custom bootloader, it does not support the use of Initial ramdisk. Because of this, our project required us to identify alternative boot processes that would allow the use of Initial ramdisk.

2.2 Linux Terminology, RCM boot and Initial ramdisk

A key part of understanding how to best accomplish our project is being able to comprehend some of the major components of Linux and its boot modes. This section will cover several terms that are important to Linux and our project, as well as detailing the importance of initial ramdisk and the RCM boot mode.

2.2.1 Linux Terminology

File system: In computing, a file system is used to control how data is stored and retrieved. The Linux operating system create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. The file system starts at root directory, and every file is under the root. [10].

Mount : Linux operating system assign a device name to each device. The action to specify the location on the file system to gain access to files on another device is called mounting [9].

Devtmpfs : Devtmpfs is a special file system which is populated with device nodes by the Linux kernel. Device nodes are interfaced to the device drivers that appear in a file system as if it were an ordinary file [11].

Procfs : According to the Linux kernel documentation, Procfs is a special file system in Linux operating systems that presents information about processes and other system information in a hierarchical file-like structure [11].

Sysfs : According to the Linux kernel documentation, Sysfs is a special file system provided by the Linux kernel that exports information about various kernel subsystems, hardware devices, and associated device drivers from the kernel's device model to user space through virtual files [11].

Configfs : According to the Linux kernel documentation, Configfs is a special file system that provides the converse of sysfs functionality. Where sysfs is a file system-based view of kernel objects, configfs is a file system-based manager of kernel objects, or config_items [11].

Systemd : Systemd is a suite of basic building blocks of the Linux system. It provides a system and service manager that run as the first process and start the rest of system [12].

GUID Partition Table (GPT): A partition table is a binary format table describing the partition layout of a disk. The GPT is a standard for the layout of the partition table as parts of the UEFI standard [14]. On a disk using GPT, two copies of the partition table exist, one at the beginning of the disk and one at the end of the disk. Figure 8 outlines the general format of the GPT.

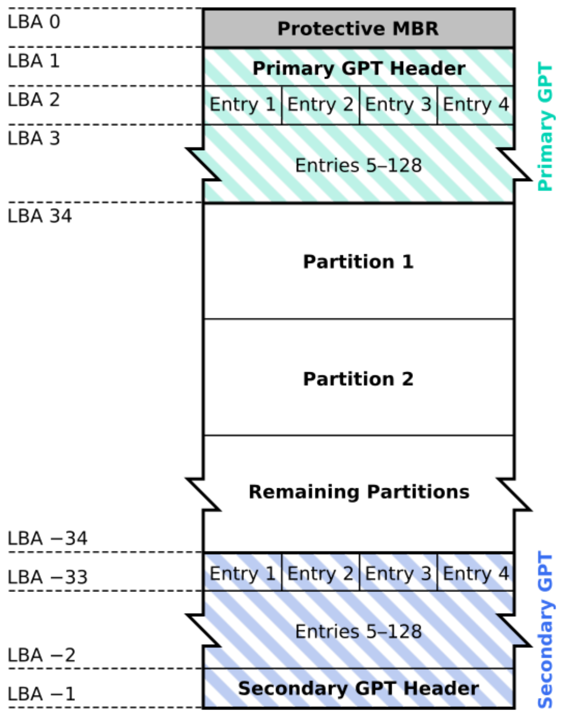
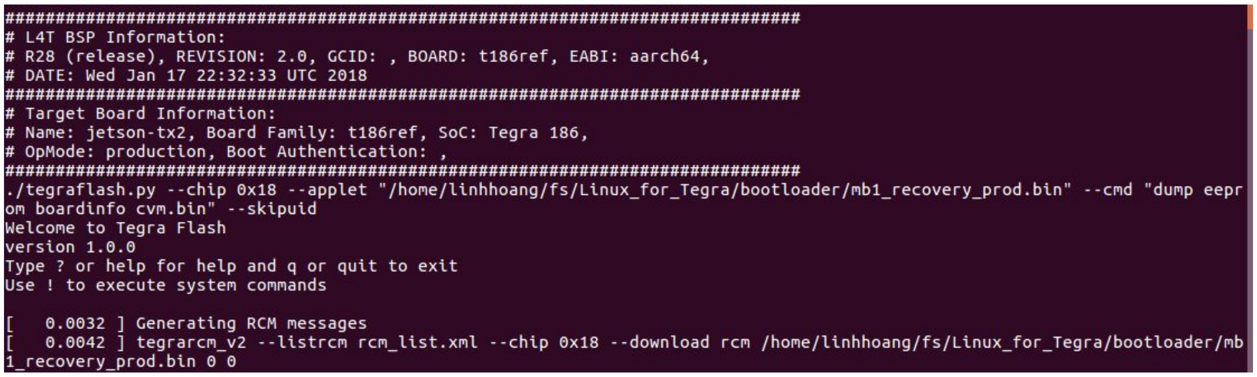


Figure 8: GUID Partition Table Scheme [15]

2.2.2 USB Recovery Mode (RCM) Boot

RCM boot is one of the boot modes on the Jetson-TX2 device. In the boot mode, the target Jetson board receives a binary through its device-mode USB port and loads it into its SDRAM. This binary is then executed on the devices. Figure 9 demonstrates how the Linux For

Tegra package generates the RCM boot signal.



```
###############################################################################
# L4T BSP Information:
# R28 (release), REVISION: 2.0, GCID: , BOARD: t186ref, EABI: aarch64,
# DATE: Wed Jan 17 22:32:33 UTC 2018
###############################################################################
# Target Board Information:
# Name: jetson-tx2, Board Family: t186ref, SoC: Tegra 186,
# OpMode: production, Boot Authentication: ,
###############################################################################
./tegraflash.py --chip 0x18 --applet "/home/linhhoang/fs/Linux_for_Tegra/bootloader/mb1_recovery_prod.bin" --cmd "dump eepr
om boardinfo cvm.bin" --skipuid
Welcome to Tegra Flash
version 1.0.0
Type ? or help for help and q or quit to exit
Use ! to execute system commands

[   0.0032 ] Generating RCM messages
[   0.0042 ] tegrarcm_v2 --listrcm rcm_list.xml --chip 0x18 --download rcm /home/linhhoang/fs/Linux_for_Tegra/bootloader/mb
1_recovery_prod.bin 0 0
```

Figure 9: Generating RCM messages for Jetson TX2

Because RCM boot is the fastest and only way to safely boot into initrd on the Jetson-TX2, it is the boot type that is used by our backup and restore scripts.

2.2.3 Initial Ramdisk

Initial ramdisk (initrd) is a scheme for loading a temporary Linux file system into memory, and is commonly used in the Linux startup process. Initrd is usually a *cpio* archive of a small Linux file system, compressed using popular compression algorithms such as *gzip*. This archive is combined with the Linux kernel and sent to the device using RCM boot mode. The kernel running on the SDRAM of the device will mount the archive to '/' (root directory).

In the normal Linux boot up scheme, the '/init' in initial ramdisk runs and mounts *sysfs*, *procfs*, and *devtmpfs*. When these special file systems are successfully mounted, the Linux system mounts the user file systems, which contain all the user's software and data, to the root ('/') directory. On Ubuntu Linux, *systemd* is initiated and continues to construct the user's Linux environment. Figure 10 shows the standard bootflow of the initial ramdisk on a normal Linux system.

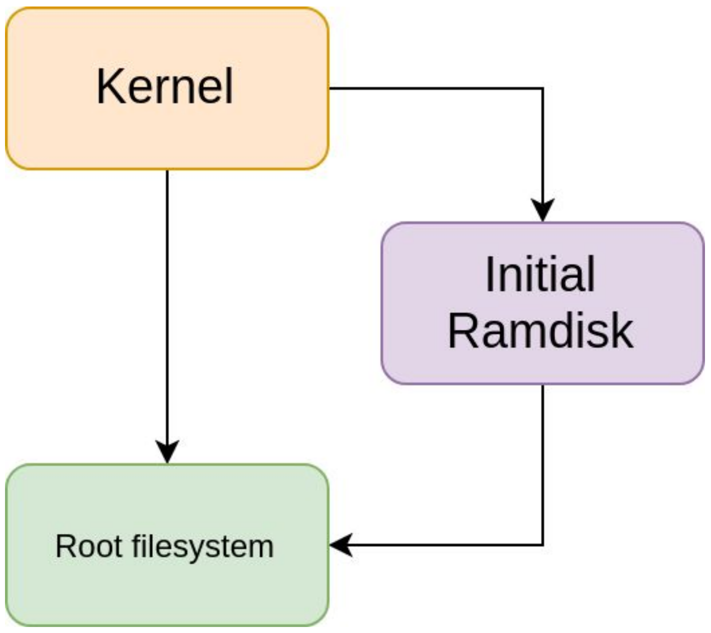


Figure 10: Initial Ramdisk Bootflow on a normal Linux system

The backup and restore environment for our project requires an initial ramdisk that is properly set up to facilitate the execution of these scripts.

2.3 Associated Technologies

In order to complete our project, we researched various technologies that may help us create an efficient tool to backup and restore Jetson boards. The following sections documents our findings.

2.3.1 Linux tools

Linux provides a wide variety of open source commands that can be used during the backup and restore processes. Several examples include:

fdisk - read the GPT of a block device

dd - read and copy a block device using logical block addresses.

mkfs - format a block device into an ext4 file system

tar - generate a copy of an entire file system

All of these tools are used in our backup and restore processes. *fdisk* is used to assist with the backing up of the GPT. *dd* and *tar* are both used for copying partition images. *mkfs* is used for formatting a device into a file system, so *tar* can then restore the file system image onto that device.

2.3.2 Serial Console

A console is the text entry and display device for system administration messages. A Linux shell is a user interface that provides access to an operating system's services. A serial port is a serial communication interface through which information transfers in or out one bit at a time. For our project, we created a console running a Linux shell controlled through the serial port so that the users can interact with the backup and cloning environment.

The Jetson-TX2 kernel provides drivers for two types of serial ports which can be used to enable a serial console (a console runs on a serial port) . The two types of serial ports are:

- Universal asynchronous receiver-transmitter (UART) serial ports
- Universal Serial Bus (USB) device ports

UART serial ports on the Jetson-TX2 support a serial console comprised of the UART-designated GPIOs and J10 debug header (used internally in NVIDIA). The Jetson-TX2 also has a USB device mode port that accepts commands from connected devices.

The Linux kernel created by NVIDIA supports a serial console on the UART GPIOs in the default setting. However, for the USB device port, the driver has to be configured to support Communication Device Class (CDC) Abstract Control Model (ACM) USB [13] , a vendor-independent publicly documented protocol that can be used for emulating serial ports over USB.

The initial ramdisk, created from the NVIDIA-customized kernel, does not support creating a console on the USB device port. Thus it is necessary to use *getty*, a Linux utility to create additional consoles for the Linux kernel environment, to set up a console on the USB device port. After running *getty*, a console is set up so that the users can interact with the backup and restore program.

3 Methodology

Prior to our arrival at NVIDIA, we conducted background research pertinent to the project. NVIDIA's internal documentation and open-source files for the Jetson-TX2 were helpful in determining the best approach to our project. Additional relevant information to our project is fully detailed in Chapter 2. Once on site, we started working alongside Jimmy Zhang, our mentor, to develop a design which we implemented during our time at NVIDIA. Through this process we identified necessary elements of the backup and restore process that needed to be addressed in order to deliver a viable end product. Figure 11 provides the layout of our process for the MQP.

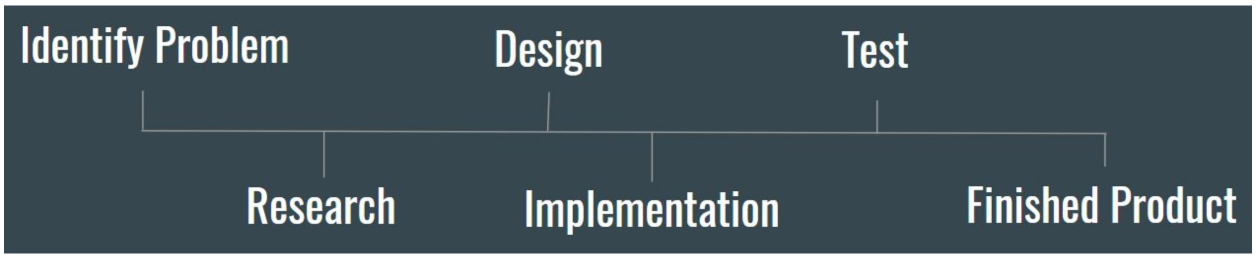


Figure 11: Project Process

The important requirements for our project were to make sure the files are safe from modification during the backup and restore processes and that all partitions and configurations are copied properly from one board to another. By backing up configurations and bootloaders from the user space, it leaves the possibility for the backup board to be altered as the backup is occuring. In order to account for this, we looked into utilizing initrd mode on the Jetson-TX2. As described in the previous chapter, initrd is a scheme for loading a temporary Linux file system into memory. Initrd would allow the scripts to backup and restore Jetson-TX2 boards while ensuring integrity on both target boards. However, because initrd is used to load the full Linux kernel in the bootup process, it contains minimal development tools. To account for this, we devised a method to inject Linux tools into the initrd mode to perform the commands needed to backup and restore. Some of the essential binaries we packaged into the existing initrd are *fdisk, dd, mkfs,* and *tar*.

As we created a more powerful and robust initrd mode, we also started writing a set of bash scripts to handle the backup and restore process. On the backup end, the script first needed to retrieve GPT information from the target Jetson-TX2 board. To complete this, the script utilized *fdisk* to parse the GPT table and stored it as a local variable. Necessary data such as partition name, start sector, and sector length were retrieved to properly create backups of each partition. The script would then utilize *dd* to create backups of each partition with the exception of ext4 filesystem partitions as *dd* is not as efficient in backing up such a large file. Instead, the script would utilize *tar* to backup ext4 extensions. In addition to cloning each partition, the backup script also creates an index file which stores information such as board specifications and partition information. The index file would also contain checksums for each partition which is used to ensure the integrity of the two target boards.

After designing how the backup script would function, we determined that the easiest way to structure the restore script would be to effectively reverse the functionality of the backup script. We decided that using a USB through initrd mode would be the safest way to run the script. Because tools like *dd* and *tar* have already been injected into the custom initrd for the backup script. It was most logical to focus on utilizing these utilities on the restore side of the cloning process, as well. To ensure that each partition is restored properly, we parse through the index file by first handling the board specification, then the GPT, followed by all of the individual partitions.

After we had developed working versions of each script and adjusted the initrd to allow these scripts to function properly, we then added extra options and utility within the scripts to allow for more customization. Once these options were implemented, we tested the full product we had developed. Additionally, Jimmy Zhang did some testing of his own to ensure that our scripts functioned properly. His extensive knowledge of bash script functionality and the Jetson-TX2 allowed for better and alternative methods of testing the scripts that we as a group may not have thought of.

Together with the help of our background research and the collaborating done at NVIDIA, we successfully created two scripts and a modified initrd as our final product. Sections 4.1, 4.2, and 4.3 further detail how the backup script, restore script, and initrd changes work, respectively.

4 Product

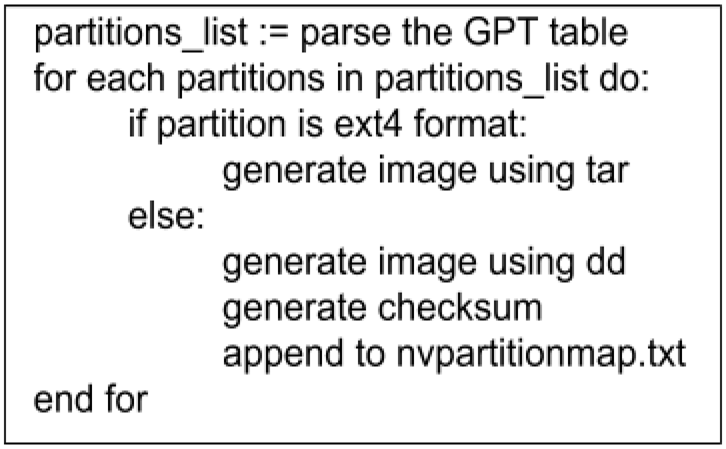
The following section details the components of the project that our team completed to create a final operable product. In order to complete our goal on this project, we developed the following three components:

1. Backup Script
2. Restore Script
3. Initrd

Together these components function as a unit and are a safe and efficient method of backing up and restoring the full configurations of one Jetson-TX2 board onto another board so that they are identical to one another. The backup and restore scripts both require the modified initrd to function properly.

4.1 Backup Script

We developed a method to fully backup a target Jetson-TX2. The Jetson-TX2 board contains three storage devices: Bootloader device 1, Bootloader device 2, and the main storage (eMMC). Six types of backup images are needed to help with backing up the three devices. These six types of backup images are Bootloader 1, Bootloader 2, the GPT, backup GPT, the main Linux file system (APP) and other configurations and bootloader partitions (Partition 1). Figure 12 is pseudocode used to represent the general structure of the backup script.



```
partitions_list := parse the GPT table
for each partitions in partitions_list do:
        if partition is ext4 format:
                generate image using tar
        else:
                generate image using dd
                generate checksum
                append to nvpartitionmap.txt
end for
```

Figure 12: Pseudocode for backup script

Bootloader 1 and Bootloader 2 are two storage devices that contain the necessary binaries to configure and boot up the Jetson-TX2. They are typically very small, so *dd* is used to back them up directly, generating Bootloader 1 backup images and Bootloader 2 backup images.

The eMMC is typically 32 GB and is divided into many partitions based on the functionalities and requirements. Thus, a backup image is created for each partition in the eMMC. First, *fdisk* and *awk* are used to parse the partition layout. From the partition layout, the script identifies the types of backup images and fetch the necessary parameters for the *dd* and *tar* command to generate the backup images. From the eMMC, the script generates the remaining four types of backup images. The backup images for the GPT and the backup GPT are constructed from the GPT partition table of the eMMC. The GPT backup images are vital for the restore process, so they are generated first. The script then backs up the Partition 1 and APP images. By default, partitions that make up Partition 1 are backed up individually. The APP partition is copied using *tar* due to its size and thus is separated from Partition 1, which is backed up using the *dd* command. Because of this, the APP is marked with a *tz* flag to signify tar being used.

In order to keep track of the backup partitions, the script creates a comma-separated value index file that keeps track of partition information. The default version of this file is called `nvpartitionmap.txt.` Each entry in the index file contains a file name, partition name, start sector, number of sectors, flags, and sha256. Figure 13 shows how an individual line in the index file is formatted and provides an example of one of these line with sample values.

Schema :<file name, partition name, start sector, number of sectors, flags, sha256>

Input: <mmcblk0p1.tar.gz,mmcblk0p1,0,58720256,tz,d95126d4d....2539ea>

Figure 13: Index file schema and sample input

The index file is stored alongside the backup and is then utilized on the restore script to properly transfer partitions to the target board. Furthermore, the index file also stores the target board specifications as the first line of the file. The script only works for Jetson boards that are identical in board specifications. In order to ensure that data transfer is completed properly, sha256 checksum values are included in the index file which is checked against to ensure the two target boards are identical when restoring. This is further elaborated upon in the Restore Script section. Another important factor to ensure the integrity of the two target boards is the sector data. Since the restore boards need to place the backed up partitions in the same memory sectors as the backup board, it is necessary to store the start location and the size of each partition. In order to retrieve sector locations, the backup script uses *fdisk* to extract the GPT into a local variable which is then parsed through as needed.

By default the restore script operates without any special flags. However it was designed to accept optional flags which allow the user to customize the script behavior. The options are as follows:

-h | --help : display usage

-s : squash mode

-z : do not use tar to backup ext4 partitions

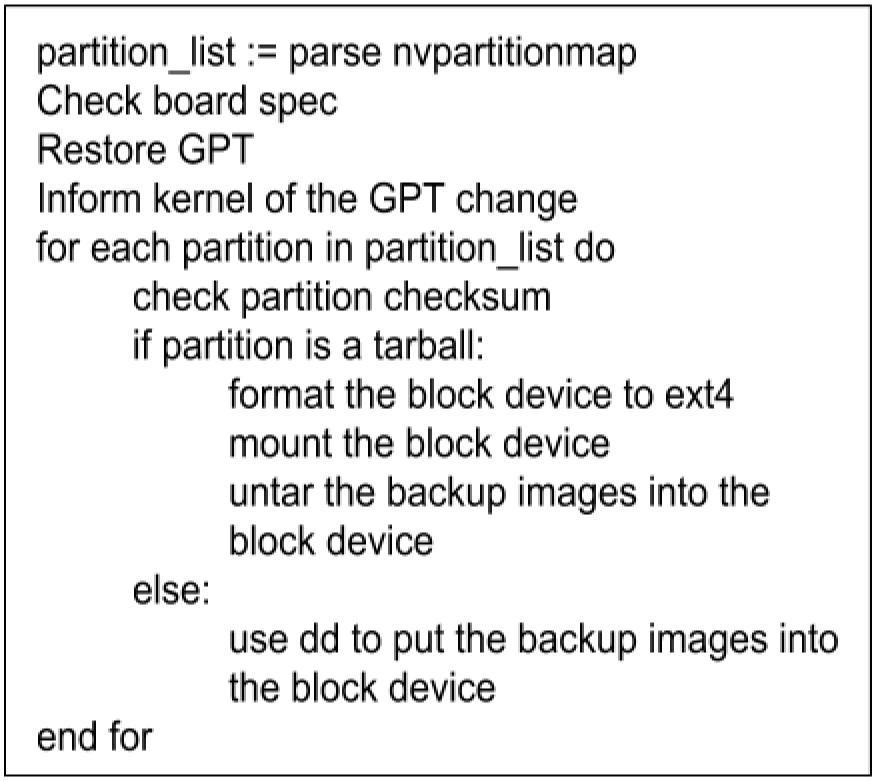
-f : force mode

-d | --device <device_name> : specify block device to store backup

Squash mode, denoted by a -s, allows for the user to squash all partitions before and after the file system partition into two files. As mentioned prior, the partitions that make up Partition 1 are backed up individually. However, with squash mode, the partitions are backed up onto two images, `partition1.img` and `partition2.img`. The -z option allows the user to solely utilize *dd* to backup all partitions including APP. Force mode is an option implemented to allow users to run the script without prompts.

4.2 Restore Script

The restore script effectively reverses the procedure that the backup script follows. This process requires that each partition that is written to the index file be flashed fully onto the target Jetson-TX2 board. The first field the restore script looks for is the board specification, identified as board_spec. Once this information is identified, the script confirms that the board_spec of the backup board is the same as that of the target board. After the board types are confirmed, the script searches for the GPT information in the index file. While the GPT information is generally on the second line of the index file, right after the board specification, it is possible that a user modifying the index file could move it. The GPT is required to be the first partition flashed because the partx command must be run after it is installed. This command updates the kernel knowledge of all of the partitions and relies on the GPT to run. Figure 14 is pseudocode used to represent the general structure of the backup script.



```
partition_list := parse nvpartitionmap
Check board spec
Restore GPT
Inform kernel of the GPT change
for each partition in partition_list do
        check partition checksum
        if partition is a tarball:
                format the block device to ext4
                mount the block device
                untar the backup images into the
                block device
        else:
                use dd to put the backup images into
                the block device
end for
```

Figure 14: Pseudocode for restore script

After the GPT is flashed, the restore script continues iterating through the index file and flashes the corresponding partitions for each line in the file. Most partitions use the *dd* command to copy the files from the mount device to the board. However, the APP partition requires the use of the *tar* command to be properly flashed, as the size of this partition causes the *dd* commands execution to be much slower. The APP partition is marked with the *tz* flag in the fifth column of the index file so the script knows that the *tar* command should be used.

There are also several customizations the user can specify when running the script. While the default name of the index file is `nvpartitionmap.txt`, the user can specify an alternate file to use. Any file name is allowed and can be specified at any argument position in the command, as long as the file extension is *.txt*. Runtime options include:

- -h | --help: display usage
- -a | --automation: suppress calls for user input
- -d=: specify mount device

The user can specify the device for mounting. While the default device is *sda1*, a different device can be specified. If the specified device and *sda1* are both not found, the script instead uses *sda*.

If the user specifies *-h* or *--help*, no flashing occurs, and the usage message will be displayed for the user. The user can also specify *-a* or *--automation* to forego viewing the message that requires user input that asks if the USB is plugged in and if it contains the backup files.

Special failsafes and extra precautions were implemented to prevent any unwanted circumstances from occurring. A special *cleanup* function ensures all devices were unmounted and the boot partitions are reverted back to read-only status. If for some reason the script exits before completion, this function is called. In addition to the script exiting if board types do not match, there are a few other instances in which the user would see an early exit. If the user inputs "no" as a response to the question asking if the USB is plugged in, the script does not execute flashing. Also, if a sha256 checksum value in the index file does not match the its value acquired from the corresponding partition image file, the script exits. Finally, if the GPT does not exist in the index file, no flashing occurs, either.

4.3 Initial ramdisk (initrd)

Unlike the traditional Linux boot scheme, the role of the initial ramdisk for our set of scripts is to provide a Linux environment for executing the backing up and restoring task of the Jetson-TX2. By using the initial ramdisk, the user Linux file system on the Jetson-TX2 is not corrupted while generating the backup image; and the restoring task can be executed directly on the Jetson devices. Figure 15 provides a general overview of the modifications that were made to the initial ramdisk to better facilitate the execution of the backup and restore scripts.

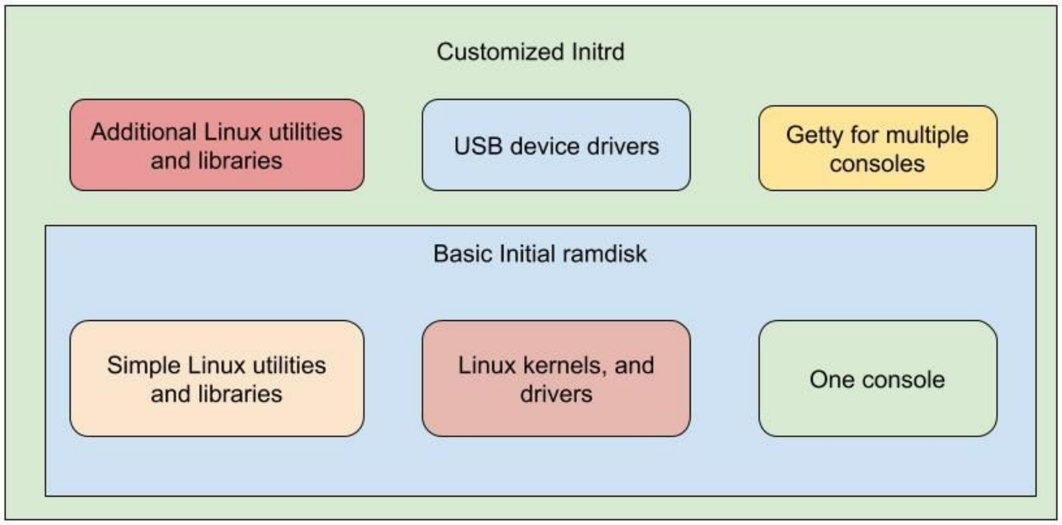

Customized Initrd
Additional Linux utilities and libraries
USB device drivers
Getty for multiple consoles
Basic Initial ramdisk
Simple Linux utilities and libraries
Linux kernels, and drivers
One console

Figure 15: Customized Initial ramdisk architecture

The existing initrd is injected with additional binaries in order to create a more powerful Linux environment for the backup and restore scripts to run. This is done by using Linux utilities *gunzip* and *cpio* to unpack the initrd. The additional Linux utilities and corresponding libraries from the provided L4T file systems are then copied into the initrd. The initrd is then repackaged using *cpio* and *gzip*. The '/init' script is also modified in the initrd to change the kernel USB gadget driver to enable ACM and start a serial console on the device mode micro USB port using *getty*. This allows the users to be able to control and interact with the backup and restore process on TX2.

The kernel USB gadget driver starts in the normal Linux system booting process which does not exist inside initial ramdisk, so the '/init' script is modified to enable the driver. We created a bash script which mounts and changes the setting of the drivers in *configfs*. *Configfs* is a file system-based manager of the driver, so the bash script changes the configuration by writing text files to the right locations in the configuration folder. After enabling ACM on the USB device mode port, a *getty* process is run to enable serial console.

4.4 Script Development Guidelines

During the development of our scripts, we followed a few guidelines to improve the product for the user. We identified a clear use case for our set of scripts to tailor these scripts to better fit what a user would want to do with them. We needed to maintain integrity with the user and ensure that their devices and files would be safe from external interaction during the cloning process. Additionally, if our colining scripts were to be used, they needed to run significantly faster than the alternative cloning approach, which would be booting up and configuring a new board.

4.4.1 Use Case

The main purpose for our product is to provide an efficient, effective, and safe way to clone all of the configurations, data, and partitions from one Jetson-TX2 to another. Ideally, users working with a Jetson-TX2 board could quickly configure many Jetson boards with the same configuration. Our product should reduce wait times during the development process, and allow developers to spend time on more important matters. For example, a customer may need create a fleet of artificially intelligent robots, all of which are powered by the Jetson-TX2.

Users can also utilize the tool to generate backup images that can be stored on a drive separate from the Jetson board. While many tools like this already exist, none have been specifically developed for the Jetson, so this would be a safe and native method of backing Jetson boards.

4.4.2 Integrity

One of the key components of creating the final product is ensuring that the scripts maintain integrity and prevent outside interference that could harm the Jetson-TX2 or its partitions. The use of sha256 checksum calculations allows the script to confirm that only proper images will be flashed onto the board. If a user modifies the sha256 value in the index file, the corresponding partition image will not be flashed because these checksum values will no longer match. Likewise, if a partition image file is modified after backing up, this checksum value will no longer match the value stored in the index file. This prevents fake backup images from being flashed onto a target board. Additionally, there are several failsafes to prevent incorrect flashing. One such failsafe is to ensure that the target backup board and the target restore board both have the same specifications since partitions are written into the same memory sectors as the backup board. Refer to section 4.2 for greater detail on the board specification.

4.4.3 Performance

A major aspect of our project that we monitored throughout the development phase was how the scripts performed. Specifically, we wanted to ensure that our backup and restore scripts run quickly so it would not disrupt the workflow of users. Initially, both scripts took upwards of thirty minutes to complete. A big factor of this was due to misaligned time stamps on the backup board and the restore board. As a result, usage of the *tar* command led to the the console outputting numerous timestamp warnings which greatly reduced the performance of the scripts. To alleviate this, a flag to remove the warnings was added to the *tar* commands in both the backup and restore scripts. Figure 16 shows the runtime of both the backup and restore scripts on a Jetson-TX2 with a 3.3 GB file system.

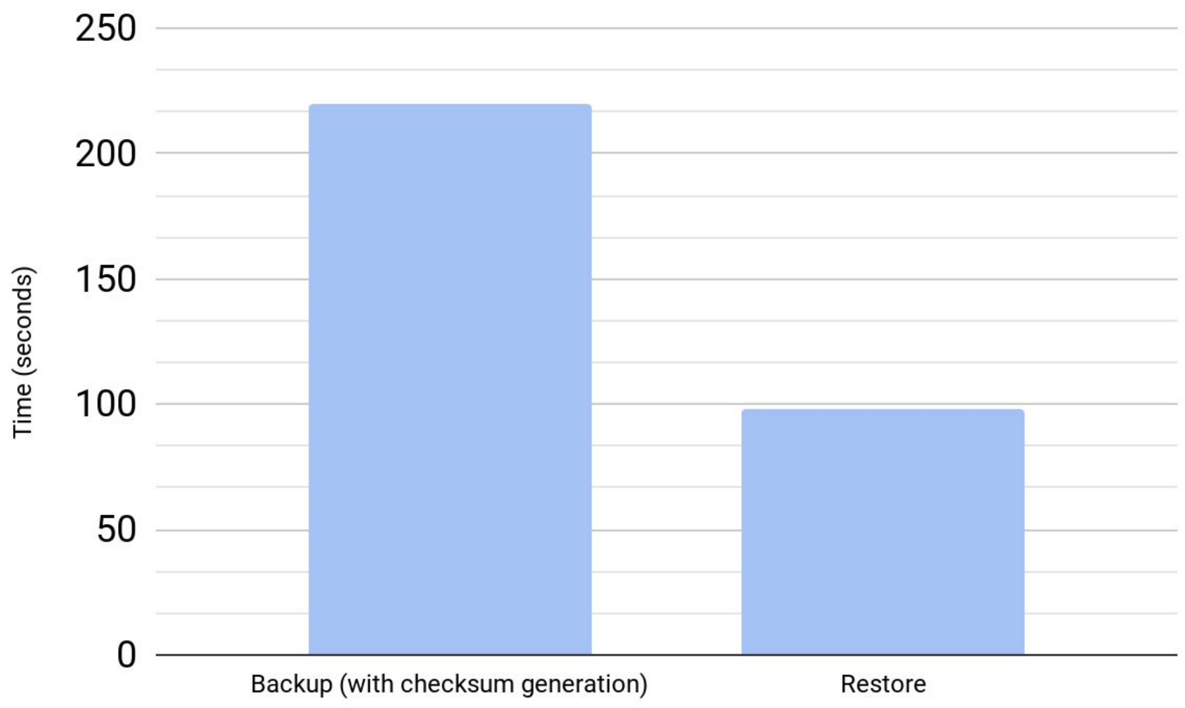


Figure 16: Script Runtime (in seconds) on TX2 with 3.3 GB File System

The backup script requires more time to run due to the checksum generation that occurs. Reducing the runtime for both scripts allows a user to have the partitions of a Jetson board cloned from the backup board to the target board in just over five minutes. If the backup board used for cloning has a different amount of partitions than a newly flashed backup board, these times may vary.

5 Future Work

Throughout the development of our set of scripts, we discovered several places that could be improved upon in the future. Most of these potential improvements involve ways that will make the cloning process more efficient and easier to set up.

Currently, the only method for properly and safely backing up the Jetson board is to do it from initrd mode. However, we think it would be helpful to add functionality to allow the backup and restore scripts, as well as `flash.sh`, to run using NFS boot. The system would have to be frozen after establishing a connection so that no files are modified during the backup process.

Adding a series of hotkeys or keypress combinations to activate a default backup or restore from a plugged in USB drive would cut down on any additional time to set up the script for execution, and allow for the tool to be more portable. A guide to create a keyboard shortcut that would run an NVIDIA-made autorun script could be included in a future release.

It may be beneficial to make the cloning process as quicker. Currently, there exists some redundancy in the scripts that handle slightly different functions. Cutting down on these redundancies could potentially reduce runtime. Additionally, a bottleneck analysis should be performed by identifying the places in the code where the scripts spend the most time. Once these lines have been identified, the code there should be optimized to increase performance.

Another area of potential improvement to make the series of cloning scripts more robust is to add an 'enable_ums' option. This option would enable the Mass Storage Gadget of the USB device mode drivers. The backup could then be stored on the host computer instead of on a USB plugged into the Jetson board. Unfortunately, the USB driver only accepts one file as a parameter at a time. As a result, the exposed block devices of the host computer would have to be rotated.

6 Conclusions

The Jetson-TX2 is useful because it is an embedded artificial intelligence supercomputer that allows developers to create smart cameras, robots, and gaming devices. The cloning tool we created will allow users to quickly duplicate board configurations, user partitions, and bootloaders to help in the development and testing process.

We worked alongside engineers to develop a Jetson-TX2 cloning tool that would be used both internally and by consumers. Presented with the task of creating such a tool, we looked into the layout of Jetson-TX2 boards to gain a better understanding of the partitions we would need to backup and restore. Using this knowledge, we created a product design which would allow us to successfully backup user partitions, bootloaders, and configurations of target boards. As we developed the scripts, we also created a more powerful initrd mode in parallel which the scripts would run on. The use of the initrd mode to backup and restore helps ensure the integrity of the backed up and restored images. The backup script individually backs up partitions and stores the image information into an index file. The index file is then utilized by the restore script to determine the proper memory sectors that each partition is restored into. Through the development phase of this project, we strived to create efficient and effective scripts. This is reflected in the backup and restore tasks being completed in just over five minutes, these scripts have a high level of flexibility that allows the users to enable optional flags to utilize the backup and restore process as needed.

Over the course of this project, we have gained a large amount of knowledge on the inner workings of the Jetson-TX2 and how bash scripts should be properly written to ensure that they are easy to use and protect the user’s information and partitions. We would like to thank NVIDIA for giving us the opportunity to work on this project and provide a useful tool for people who use the Jetson-TX2 is their daily work.

7 Bibliography

[1] Unified Extensible Firmware Interface. (2017, December 13). Retrieved March 2, 2018, from https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

[2] Vincent Zimmer, Michael Rothman, and Robert Hale. (May 10, 2007). EFI Architecture. Retrieved March 2, 2018, from http://www.drdobbs.com/embedded-systems/efi-architecture/199500688

[3] UEFI Firmware, (n.d.). Retrieved March 2, 2018, from http://technet.microsoft.com/en-us/library/hh824898.aspx

[4] Linux For Tegra (December 13, 2017). Retrieved March 2, 2018, from https://developer.nvidia.com/embedded/linux-tegra

[5] Jones, M. (2010, November 10). Network file systems and Linux. Retrieved March 2, 2018, from https://www.ibm.com/developerworks/library/l-network-filesystems/index.html

[6] NVIDIA Jetson Modules and Developer Kits for Embedded Systems Development. (n.d.). Retrieved March 2, 2018, from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modul es/

[7] Time Machine. (n.d.). Retrieved March 2, 2018, from https://www.apple.com/airport-time-capsule/

[8] Team., D. (n.d.). Clonezilla. Retrieved March 2, 2018, from http://clonezilla.org/

[9] FreeBSD Documentation - Mounting and Unmounting filesystem (n.d). Retrieved March 2, 2018, from https://www.freebsd.org/doc/handbook/mount-unmount.html

[10] Linux File System - The Linux Documentation Project (n.d). Retrieved March 2, 2018, from https://www.tldp.org/LDP/intro-linux/html/chap_03.html

[11] Linux Kernel Documentation (n.d.). Retrieved March 2, 2018, from https://www.kernel.org/doc/Documentation/filesystems/

[12] Systemd System and Service Manager. (n.d.). Retrieved March 2, 2018, from https://freedesktop.org/wiki/Software/systemd/

[13] CDC_ACM (n.d). Retrieved from http://wiki.openmoko.org/wiki/CDC_ACM

[14] UEFI specifications (n.d). Retrieved March 2, 2018, from http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_7_A%20Sept%20.pdf

[15] GUID Partition Table Scheme [Digital image]. (n.d.). Retrieved March 2, 2018, from https://commons.wikimedia.org/wiki/File%3AGUID_Partition_Table_Scheme.svg

[16] Embedded Systems Development. (n.d.) Retrieved March 2, 2018, from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/