UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Toomas Soome

Porting and Developing a Boot Loader

Master's Thesis (30 ECTS)

Supervisor(s): Meelis Roos

Tartu 2018

Porting and Developing a Boot Loader

Abstract:

This paper describes the project to replace outdated boot program in illumos project with alternative one, allowing to provide better support for modern and future computer systems and having an architecture to support extending and improving the implementation.

Keywords:

Boot Loader, Operating System, FreeBSD, illumos

CERCS: P175 Informatics, systems theory

Boot programmi portimine ja arendamine

Lühikokkuvõte:

See magistritöö kirjeldab projekti, mille eesmärk oli asendada illumos projektis aegunud boot programm alternatiivsega, mis võimaldab paremini toetada kaasaegseid ja tuleviku süsteeme ning mille arhitektuur võimaldab parenduste ja täienduste kirjutamist.

Võtmesõnad:

Boot Loader, Operating System, FreeBSD, illumos

CERCS: P175 Informaatika, süsteemiteooria

Table of Contents

1 Introduction 5
1.1 What is this all about 5
2 Boot Loader 6
2.1 Boot Loader Components 6
Access to the Storage 7
User Interaction 7
Scripting 7
3 Operating System Integration 8
3.1 Packaging the Software 8
3.2 Installing the Boot Loader 8
3.3 beadm and libbe Interface 9
pkg and bootadm 9
reboot 10
3.4 Emulating the Boot Loader 10
4 Platform Support 11
4.1 BIOS Support 11
4.2 UEFI Support 11
5 Implementation 12
5.1 Building the Boot Loader 12
5.2 Adding Multiboot 2 Support 13
5.3 Adding New Features to the Boot Loader 16
Heap Size 16
Updating Ficl 16
Processing the Command Line 16
Rewriting the Block Cache 16
Build gptzfsboot Against libstand 18
Changes to the UFS File System 18
Changes to the dosfs 18
Changes to the ZFS Reader 18
5.4 Scripting for illumos 19
Boot Menu 20
Boot Environments 22
5.5 Chain Loading 23
6 Building Support for UEFI in the Kernel 25

7 The Console 26
7.1 The Console in the Boot Loader 26
Drawing Text on the Framebuffer 28
Terminal Emulation 29
Graphics Primitives for Boot Loader 31
7.2 The Console in the Illumos Kernel 31
The Mini Driver for Early Boot 32
Framebuffer Device Driver 33
Loading Fonts 34
TEM and Unicode Chars 35
Things to Do 35
8 Conclusions 36
9 References 37
Appendix 38
I. License 38

1 Introduction

1.1 What is this all about

The illumos project is a fork of OpenSolaris operating system open sourced by Sun Microsystems. The project is providing base code for operating system and is used as core for several public and private distributions. The original code did include a boot loader implementation based on a version of Grub, now labelled as Grub Legacy. Unfortunately, this implementation was limited to support BIOS systems only and neither its architecture nor license is allowing easy development.

While I was investigating the alternative implementations, the initial choice was to try out Grub 2, since the illumos was already using previous version of the Grub boot loader, it felt reasonable choice. While Grub 2 is the most feature rich boot loader around, its licensing does not allow code sharing with other illumos components. Additionally, the architecture of Grub 2 does not really fit well with boot environment technology used by illumos based systems, which is making the future updates more complex and possibly error prone – the problems we really would like to minimize with operating system updates.

From other possible choices, based on features, license and history of cooperation between illumos and BSD developers, for porting, I did select the FreeBSD boot loader.

In chapters 2 – 4 we will describe the main components of boot loader, what it means to integrate the boot loader with the specific operating system and how this all affects the operating system development. The boot loaders used to be (and still are) very specifically built and tuned to hardware and operating system details and the common approach we see in articles and book excerpts is about presenting some low-level assembly text or some sample about programming some controller. The need to share multiple operating systems on the same system and to provide flexible and easy way to select options how the operating system should be started, has resulted with boot loaders implementing more complex user interfaces.

In this paper I have used a lot of references to the actual source code, freely available in public source repositories. While the online source repository can be quite volatile and there will be changes, we also get the history of the changes and the interested reader can see how the development is happening over the time.

In chapters 5 – 7 we will walk through the most prominent aspects and features of the actual implementation, first from implementing the port of the FreeBSD boot loader to the illumos, then the necessary changes and updates to the illumos operating system to take advantage of the new boot loader and added features. All the updates described in chapters 5 – 7 have been implemented by me.

Among the additional features is also the option to start the operating system on top of UEFI firmware and this option did result with the need to build entire new interface for the operating system console.

The goal of this work is to select and implement the new boot loader for illumos system, and to update the illumos to support the new features.

2 Boot Loader

First of all, we have question of why we need the boot loaders at all and what is the generic architecture for boot loaders.

Today's computer systems in general are pre-loaded with code to perform the initial initialization and diagnostics for the hardware and to provide basic functionality to load and start the user software – usually the operating system. This software is stored in ROM part of the system memory and has many limitations. In a similar way, the interface to load the user software is also limited and is only providing very basic functions for programs to interact with the machine.

With few exceptions, the operating system, once the kernel is loaded and initialized, generally does not access or interact with system firmware and the user software used to load the kernel, is no longer needed. Such conditions do direct us towards modular software solution, where we use simple programs to load and start more complicated ones in stages and once next stage is started, the previous is discarded from the memory. Such staging is usually called boot loading. The example boot loading with staging is listed in Figure 1.

1. PC BIOS is done with POST and the system is set to 16-bit real mode.
2. PC BIOS will load sector 0 (512 bytes) from boot disk into memory address $0000:7C00 and will jump on this address to execute the loaded code.
3. The loaded 512 bytes is actually mix of the MBR partition table and boot program, which can only be very simple and usually will load from partition boot program area the next stage and will jump into it. Previously loaded MBR code is discarded.
4. Partition boot program abilities depend on how much space was dedicated to it – it may be smart enough to be able to load and start the operating system kernel directly, or it may only have basic features to load and start yet another stage. Usually the PC boot loaders switch the machine into 32-bit protected mode at this stage.
5. Finally, we have operating system kernel loaded into the memory, the necessary operating system specific preparations complete and we can pass control to the kernel.
6. Once the operating system kernel is running, the boot loader stage(s) loaded in previous steps are discarded and the resources are re-used for the operating system needs.

Figure 1. Stages in Boot Loading

Depending on the design decisions and intent, the boot loader may or may not provide features for user interaction via user interface. If implemented, such interface is used for diagnostics and for temporary configuration changes to work around errors and issues while loading the operating system.

2.1 Boot Loader Components

The purpose for boot loader is to implement loading and starting of the operating system, therefore the smallest set of features needed is the ability to read the operating system kernel files from the media. However, usually we would like to have some amount of information about the progress. Incidentally we can look on boot loader as special case of operating

system and we can use the same, already familiar design principles. Such approach will allow us to use already familiar API for programming and to apply well proven concepts.

Access to the Storage

Access to the storage is the most important part of the boot loader, because that's how we can access and load the other components. The access to the data alone is not the only criteria, however. Performing the read operations is usually quite time expensive and in worst case, our loading process can be slowed down significantly.

The storage itself can also be presented in different forms: local hard disks, USB storage, network-based storage. Providing the access to the different media types may also depend on the system firmware, the access to specific media via firmware functions may only be granted if we actually did set the system to boot from that media.

User Interaction

The bare minimum is to support output into some sort of console – either system local display, or serial console or some form of remote management solution. It is not uncommon to not provide any boot loader specific output unless there is some error condition.

In many cases the boot loaders do implement support for both output and input from user. This way the system can provide the environment for user, making it possible to set or select system specific features and to perform diagnostics and repair procedures.

Scripting

When boot loader does implement the user interaction, it also is quite common that the commands and control mechanisms are implemented to make it possible to create boot loader configuration scripts. As we are running in very limited environment, such scripting language has limited features, and is mostly used to create boot menus and features to select alternative options.

3 Operating System Integration

Integrating the boot loader with the Operating System means not only to provide the binaries to be installed on boot media, but also providing means to manage the installation of the boot loader, management of the boot options and features, maintenance and testing. Let's walk over the basic integration steps using illumos as an example.

3.1 Packaging the Software

The role of the packaging is to make software and its updates available for the system. In illumos we provide boot loader in pkg:/system/boot/loader package and use the package system features to provide updated versions available for updating the software. It is important to understand how the software provisioning and updates work in specific system, because this can be critical for maintenance. In case of illumos, the common case is to create a new clone of the system and apply the updates on that image, then boot the system using that newly created image and leave previous one behind. Such images are called Boot Environments (BE). As a common practice, during the [image] update, all applications executed during the update are only located on current BE, never on new BE.

Such behaviour will ensure the involved components will work as expected. Also, the currently installed management programs must be able to manage the system despite the new version being installed.

3.2 Installing the Boot Loader

The most basic or low-level tool to install the boot loader is the program named "installboot". This program is specifically written to deal with this specific boot loader implementation. The command line is defined as shown in Figure 2.

```
SPARC
  installboot [-fn] [-F zfs|ufs|hsfs] [-u verstr] bootblk raw-device
  installboot [-enV] -F zfs -i raw-device | file
  installboot [-n] -F zfs -M raw-device attach-raw-device

x86
  installboot [-fFmn] [-u verstr] stage1 stage2 raw-device
  installboot [-enV] -i raw-device | file
  installboot [-n] -M raw-device attach-raw-device
```

Figure 2. installboot command line

The installboot command does require the boot loader stage files and the device name of the root file system. Depending on the partition table and file system type, the correct target location for stage files is selected and the files are installed.

We always do install stage1 file as MBR (or in case of GPT, PMBR) and partition boot block and stage2 is either stored into unused space before partition, special dedicated boot partition if there is no unused space before partition or in case of ZFS, into the dedicated boot block area in ZFS label [1, p. 14].

The purpose of stage1 is to locate and read into the memory the next stage and start the loaded stage.

Stage1 is designed to read specified number of blocks from given 64-bit Logical Block Address (LBA). The LBA and number of blocks denotes the location and the size of stage to be loaded. Having the location recorded will free us from browsing different types of

partition tables and the size is needed because we are storing the next stage directly on disk, without a file system. If the MBR code is installed, it is always set to read and start partition boot code. While the MBR update is optional, we always install or update the partition boot block. This approach will assure we will always have recorded the correct location and size of next stage.

The purpose of stage2 is to implement file system reader to read the stage3 from the root file system and start it.

The stage2 is constructed to have built in version information, to make it possible to identify the binary to make it possible to decide if the installed binary will need update or more importantly, to make sure we will not accidentally downgrade it. The versioning is implemented as attached last block at the end of the stage2 file and the location is stored in "fake" multiboot 1 structure. The structure is fake because we do not use it to build multiboot 1 boot protocol but is only used to provide private information.

The structure was picked because of built in identification (magic code) and for its internal storage space. We will also store the root file system partition start LBA into this header, this will help the stage2 to identify the root file system even if the stage itself was stored in a separate dedicated partition (or even in the space outside the partitions).

The purpose of stage3 is to load and start the operating system kernel and to provide user interface to manage the boot process.

The installboot command will discover the related disk partition(s) and locations where the stages are to be installed. Then it will perform the version check for stage2. By default, we only allow newer version to be installed. And finally, the stage programs are updated with LBA and size values and are written on dedicated areas on the disk.

3.3 beadm and libbe Interface

Updates to Operating System will normally create new clone of the system image, new boot environment (BE) and the command beadm is there to help to manage the BE's. It will provide ability to list, create, rename, delete and activate BE's. The actual implementation is provided by libbe interface and can be used by other system applications.

Part of the BE activation is also boot program installation. libbe will execute the installboot command for every disk device used for root file system where this BE is stored. This will ensure we have up to date boot program setup on the disk(s).

Finally, the list of the BE's is maintained in the ZFS pool root dataset in the file <poolname>/boot/menu.lst. Even as we can read the BE dataset names directly from the pool configuration, having the file-based list does allow us to have more control over the order of the entries and to add custom types of entries. Note the /boot/menu.lst is not the same as one used for Grub.

pkg and bootadm

The package management utility pkg is another consumer for libbe interface. While performing the package update, it will call libbe to create new BE, then the update is performed on new BE and lastly, the new BE will be activated and along with activation, the boot programs are installed as described above.

bootadm command is the tool to manage boot archive (miniroot with essential files and drivers to support kernel start up before the root file system is mounted), boot menus and to

provide simplified method for user to install the bootloader. bootadm is relying on libbe implementation to provide the boot loader installation.

reboot

The reboot command in illumos allows to set temporary, one-time boot settings for the kernel. As the x86 BIOS platform does not provide a mechanism to implement such feature, the boot loader temporary configuration (/boot/transient.conf) is used. The reboot mechanism will create and record transient configuration, and on next boot, the file is removed before system will enter the multi-user run-level. The implementation is actually using internal mechanism implemented by bootadm command.

3.4 Emulating the Boot Loader

Important part of the development is support for testing. Testing boot loader and the scripts used by the boot loader can be time consuming and taunting task and it would be nice to have ability to perform automated tests. To ease up the task, we do provide the userland program, capable of interpreting scripts run by the boot loader, the configuration and this will allow us to perform a set of tests on boot loader scripts without actually booting the system.

The boot loader is using the Forth scripting engine implementation “ficl”, and we use the same source with additional emulation functions to provide the interpreter for boot loader scripts and setup. The same engine is also used internally by the libbe and bootadm components to make it possible to produce information about current settings.

4 Platform Support

The current illumos based systems do support two architectures, x86 PC systems and SPARC based systems. The current state of the boot loader port is only focusing on x86 platform and the SPARC port is left for future. The modern x86 systems are equipped with two alternative system firmwares – traditional BIOS and UEFI, and the industry is moving towards switching to UEFI. Such move is implemented by providing BIOS emulation layer inside UEFI firmware and then later, pure UEFI systems will be left. This transition on x86 world does mean that it is absolutely essential to develop UEFI support for operating systems and for boot loaders.

4.1 BIOS Support

The BIOS based systems have been around for decades and one might ask, what is there to develop. Curiously enough, with birth of the UEFI CSM mode for BIOS emulation, all kind of bugs are (re)introduced in firmware and the developers have challenges to keep the boot loaders functional.

New features have added and suddenly the old solutions are not usable any more – features like new disks with 4kn sector size.

4.2 UEFI Support

Providing functional UEFI support is key for survival for any current Operating System. There are many challenges building such support. First of all, the boot loader has to be designed to support UEFI. UEFI firmware does provide the whole set of APIs to provide an access to the hardware, but also the API to access services provided by the firmware itself (UEFI Boot Services and UEFI Runtime Services) and by other components of the system. At the same time, there is line of specification updates and the developer has to make decisions about how old version should be supported.

Also, the Operating System has to implement some degree of UEFI support. As bare minimum, the boot loader installation procedures have to be adjusted and the early kernel initialization can't rely on any BIOS specific features – as there are none.

For better support, the support to access UEFI Runtime Services feature should be provided for both kernel and userland.

5 Implementation

This chapter will describe the steps and details of the actual implementation of the boot loader itself and the changes done in the operating system.

5.1 Building the Boot Loader

Building normal userland application does involve using support files such as headers, libraries with frameworks and API and other components to support the application to run inside the operating system.

Boot loaders are running before there is any operating system up and they need to be built in specific way to avoid all the dependencies from the operating system services. The application built in such a way is called a standalone application or a standalone program.

Fortunately for us, the standalone program can be entirely self-contained package of files and build rules, and this will make porting the standalone program relatively easy.

Therefore, the first step was to collect the boot loader source, relevant header files, build the make rules, packaging rules and have the first binary built.

The challenge at this stage was to create the make rules, integrate the build logic with the rest of the build system and to create the packaging.

In the illumos source tree, the boot loader source is located in usr/src/boot subtree[1].

The original FreeBSD bootloader[2] is providing multiple variants of the various stages, the reasons are to provide some functional differences, also there are historic reasons how the FreeBSD boot setup is done.

For illumos port, my decision was to reduce the list of files to minimum. This did allow to implement automation of the boot loader installation, but the cost is slightly limited functionality. For example, not providing stage2 binary defaulting to serial output means that some of the early boot messages will be missed, but since the console output can be set via configuration files, such loss is perhaps acceptable.

So, we do build the list of the standalone binaries shown in the Table 1.

The BIOS system is using file pmbr for MBR code (stage 1) and for partition boot code (still stage 1).

The file gptzfsboot used in BIOS system as stage 2 boot program to implement disk boot. The purpose is to detect the root file system, read out the stage 3 from it and start the stage 3. The gptzfsboot does read config file allowing to set console device and does provide very simple boot prompt allowing to specify boot device and the name of the stage 3 program.

The file zfsloader is used in BIOS system as stage 3 boot program to implement the user interface and to perform loading and starting of the operating system kernel.

There are also two special cases, files cdboot and pxeboot, implementing the boot support for ISO 9660 (CD/DVD) and network boot. The special cases are needed to make sure those boot programs are small enough to be usable.

To support UEFI systems, there are two boot programs implemented as files boot1.efi and loader.efi. boot1.efi is used as stage 2 boot program, is stored in EFI System Partition (ESP) and is loaded by the UEFI firmware. boot1.efi will then locate, load and execute the

[1] https://github.com/illumos/illumos-gate/tree/master/usr/src/boot
[2] https://svnweb.freebsd.org/base/head/stand/

loader.efi. To support cd and network boot, the UEFI platform is using loader.efi directly. In near future, the boot1.efi will get removed as technically it is not really needed and we could use loader.efi in ESP directly.

Table 1. List of the Boot Programs

File in /boot directory	Description
boot1.efi	Stage 2 file for UEFI system
loader.efi	Stage 3 file for UEFI system
cdboot	boot program file for cd9660 image
gptzfsboot	Stage 2 file for BIOS system
pmbr	Stage 1 file for BIOS system
pxeboot	Boot program file for network boot (PXE boot)
zfsloader	Stage 3 file for BIOS system

5.2 Adding Multiboot 2 Support

Now when we can build the standalone boot loader, we need to take steps to teach the boot loader to load the illumos kernel, prepare the necessary data for the kernel start and actually start the kernel. To do all this, firstly, we would need to understand what is really happening when we are preparing and starting the kernel.

In illumos, the kernel is ELF binary. To load the ELF binary there are two options – either we interpret the ELF headers and set up the memory layout as described by the ELF header or we load the whole file as one binary blob and let the special program to handle the memory layout and setting up the components from the ELF file.

For Grub boot loader, the boot process was built to load the whole 64-bit kernel file as binary blob. The special program is identified by the Program header with type PT_LOAD, where the virtual and physical address are set to same value. The load address is determined by the PT_LOAD type Program Header, where virtual address and physical address are the same, and by subtracting the ELF header size.

To describe how the kernel is to be loaded, what information the boot loader should pass to the kernel and what other files should be loaded is implemented via boot protocol. Some boot protocols are private for a specific system and developed keeping in mind just the needs of this operating system. However, there has also been an attempt to build generic purpose boot protocol, usable by different operating systems – the multiboot boot protocol. The first version is referred as multiboot 1 and second version as multiboot 2.

To pass boot time data to kernel, the Grub Legacy in illumos is using multiboot 1 boot protocol [2], but unfortunately the multiboot 1 boot protocol is rather limited and does not support UEFI platform. The multiboot 2 boot protocol [3] was an attempt to address the shortcomings of multiboot 1 boot protocol and we can support both variants with little effort.

Therefore, the first step in actual development was to add support for the multiboot 2 boot protocol for both the kernel and to the boot loader.

For the boot loader, the support means adding the necessary checks to detect if the kernel actually does support the multiboot 2 boot protocol and to implement the code to provision the needed data structures for passing to the kernel. And finally, we need to implement the trampoline code to jump from boot loader into the kernel when we are about to start the loaded kernel. If the kernel does support multiboot 2 boot protocol, we will prefer it. If the platform is UEFI and the kernel does not support multiboot 2 boot protocol, we will abort loading with an error.

The implementation of multiboot 2 specification in boot loader is in the file "multiboot2.c"[3]. There we implement the verification of the kernel and building the multiboot 2 tag list to be passed down to kernel via multiboot info package. Additionally, we will create virtual module to pass boot loader environment variables down to kernel to be used as source for setting up the properties (see also section 5.4).

The multiboot 2 boot protocol is used to boot both BIOS and UEFI platform. In case of BIOS, we control the whole system and we can place the loaded kernel and modules to final locations while loading and we can share the multiboot trampoline with multiboot 1 boot method. In case of UEFI, the UEFI firmware is having the control over the system and before we can switch off the UEFI Boot Services, we have to follow the rules set by UEFI specification. To implement kernel and module loading in UEFI system, we must allocate memory and read the data into the allocated temporary memory. Once we switch off the UEFI Boot Services, we can move data into final location and start the kernel.

The problem with moving the data is that our boot loader code or allocated memory may be (partially) overlapping with the final locations, also the system memory map may have unusable areas in target address space. To solve this issue, we will construct the mapping for loaded data, where we record the address of the buffer where the data was loaded, target physical address and the size. After we switch off the UEFI Boot Services, we will copy the needed functions and relocation info into the safe memory and call the trampoline.

The first step done in the trampoline is to copy the loaded data into the final locations. Since the illumos kernel startup is built keeping in mind the requirements from the previous Grub, in case of UEFI64, we need to switch from 64-bit long mode to 32-bit protected mode, so the next step of the trampoline will do the switch and then we will jump to kernel. The setup leading to trampoline is done in multiboot2_exec() and the trampoline itself is in multiboot_tramp.S[4]. The UEFI32[5] case is simpler in sense that after the trampoline is done with data relocation, we can jump to the kernel.

For the kernel, the support means providing the multiboot 2 header structure embedded into the first 32KB of the kernel file and implementing the functions to parse the multiboot 2 information provided by the boot loader and to provision the data into native data structures. To support both boot protocols, we need to build the parsing and data extraction functions to handle both protocol data structures. Example about detecting the multiboot version and setting up the scene in the early kernel is shown in the Figure 3. The support functions to

[3] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/common/multiboot2.c

[4] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/efi/loader/arch/amd64/multiboot_tramp.S

[5] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/efi/loader/arch/i386/multiboot_tramp.S

process the multiboot 2 information tags are implemented in the file dboot_multiboot2.c[6]. The multiboot info is processed in dboot_startkern.c[7] in order to extract the needed essential information.

```
/*
 * Set up basic data from the boot loader.
 * The load_addr is part of AOUT kludge setup in dboot_grub.s, to support
 * 32-bit dboot code setup used to set up and start 64-bit kernel.
 * AOUT kludge does allow 32-bit boot loader, such as grub1, to load and
 * start 64-bit illumos kernel.
 */
static void
dboot_loader_init(void)
{
#if !defined(__xpv)
        mb_info = NULL;
        mb2_info = NULL;

        switch (mb_magic) {
        case MB_BOOTLOADER_MAGIC:
                multiboot_version = 1;
                mb_info = (multiboot_info_t *)(uintptr_t)mb_addr;
#if defined(_BOOT_TARGET_amd64)
                load_addr = mb_header.load_addr;
#endif
                break;

        case MULTIBOOT2_BOOTLOADER_MAGIC:
                multiboot_version = 2;
                mb2_info = (multiboot2_info_header_t *)(uintptr_t)mb_addr;
                mb2_mmap_tagp = dboot_multiboot2_get_mmap_tagp(mb2_info);
#if defined(_BOOT_TARGET_amd64)
                load_addr = mb2_load_addr;
#endif
                break;

        default:
                dboot_panic("Unknown bootloader magic: 0x%x\n", mb_magic);
                break;
        }
#endif  /* !defined(__xpv) */
}
```

Figure 3. Detecting the Multiboot version

For this time, we only provide support for multiboot boot protocols, even as there also exists the native boot protocol which is currently only used to pass information from the multiboot protocol to the next layer in the kernel. The reason to implement only the multiboot protocol support is that we need to preserve compatibility for some time, so users will have time for transition from Grub to the new boot loader.

[6] https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/i86pc/dboot/dboot_multiboot2.c
[7] https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/i86pc/dboot/dboot_startkern.c

5.3 Adding New Features to the Boot Loader

Once we have the ability to load and start the kernel, the next steps are about polishing the details and adding features to improve the user experience and the operating system support.

Heap Size

The heap size was limited to very small (3MB) amount of the memory, and I did increase it to 64MB. The reason was to provide more space to cache and to support additional features.

Updating Ficl

The boot loader stage 3 is currently using Ficl[8] scripting engine. Ficl is an implementation of Forth[9] programming language. The current latest Ficl is version 4.1 and I did update illumos port to use it, replacing the previous version 3.03.

The motivation for this update was ability to use compressed softcore (the forth dictionary with core words implemented in forth and built into the ficl binary), the Ficl 4 is providing LZ77 compressed softcore. I did replace LZ77 with LZ4 as later does offer slightly better compression ratio and we will get smaller payload. Also, in the boot loader, we will get LZ4 decompressor virtually for free because it is also used by the ZFS, so the code is already there.

Additionally, some extra forth words were added to complement forth words for double numbers and list of bug fixes posted in Ficl forum but not yet integrated.

We build Ficl as interpreter component in boot loader stage 3, but also as runtime library (libficl-sys.so.4.1.0) and command executable (ficl-sys) in illumos userland. The runtime library does additionally implement loader emulation support allowing us to interpret boot loader scripts in illumos. Such boot loader emulation is needed to ease the development of boot loader scripts, but it will also allow us to extract information from boot loader configuration.

Processing the Command Line

To offer better user experience, the boot loader was updated to use linenoise[10] command line editor. Linenoise was chosen because it's implementation is very small and lightweight while providing simple line edit, cursor movement and history support. However, it was quickly discovered that this implementation is not really good for use on slow serial lines and while it is still in use in boot loader, I have set a goal to replace it with better solution. This part of the development is definitely set as task in the future.

For ficl-sys, we use tecla[11] command line editor instead. Userland binary program does not have the limitations of the standalone binary and since tecla is already used by other components in the system, we are re-using the shared components.

Rewriting the Block Cache

One of the first discoveries while testing the new boot loader was that reading the disk was rather slow, despite making use of the block cache (bcache[12]) to speed things up. It appeared

[8] http://ficl.sourceforge.net/ficl.html
[9] http://www.forth.org
[10] http://github.com/antirez/linenoise
[11] http://www.astro.caltech.edu/~mcs/tecla/
[12] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/common/bcache.c

that the implementation was probably good enough for floppy devices, but not adequate for other media.

The ideas to solve bcache issues were:

1. Use simple hash table based on the cache size and block number.
2. Create per-device cache instance. The devices are enumerated during the stage initialization and this way we can keep caches for file systems like ZFS, where we need to access (potentially) multiple disk devices.
3. Implement simple read ahead strategy. Most of the reads in boot loader are sequential and often for whole files – so we would benefit from read ahead. To keep things simple, we will not wrap "around" the cache with read ahead. Also, we do set maximum number of blocks to read ahead to 256. This number is partly arbitrary and based on experiments to see how much read ahead would give the benefit. Another reason is, reading small files will have little help from reading ahead too many blocks.

The read ahead feature did also reveal a whole set of issues with some systems. First of all, we found we need to discover the disk size and handle the IO checks properly. This is important because some systems will not handle the reads past disk end and will start to behave erratically.

Secondary issue related to disk size detection is about the fact that there are systems which do provide bogus information about the disk size. Fortunately, we have quite a good workaround – we can read and set disk size based on information in disk partition table. With GPT (EFI) we do have the disk size recorded. In case of other partition table types, we will get the disk size indirectly from partition declarations.

To use bcache, we need to initialize it at first by setting total cache size in blocks and the block size, such as listed in Figure 4. Secondly, we need to allocate our chunk of cache when

```
    /*
     * Initialise the block cache. Set the upper limit.
     */
    bcache_init(32768, 512);


bd_open()
{
  . . .
  BD(dev).bd_open++;
        if (BD(dev).bd_bcache == NULL)
            BD(dev).bd_bcache = bcache_allocate();
  . . .
}

bd_close()
{
  . . .
  BD(dev).bd_open--;
  if (BD(dev).bd_open == 0) {
    bcache_free(BD(dev).bd_bcache);
    BD(dev).bd_bcache = NULL;
  }
  . . .
}
```

Figure 4. Setting up the Block Cache

we open the disk for read and return our chunk on disk close. As we do not want to limit the disk use for single open-close session, we use reference counting and only return the cache chunk on last close. To actually use bcache, we have bcache strategy call implementing the cache and we will provide it the real strategy call to use to read the data from the media.

And finally, some related numbers. Normal illumos boot does load kernel and boot archive (miniroot file system). In case of SmartOS, the boot_archive is as large as about 160MB. Loading boot archive with such a size from USB2 stick took about 15 minutes before the update. After the update, the load time is down to 30-35 seconds.

Build gptzfsboot Against libstand

The libstand is library in boot loader implementing the set of base functions in the boot loader.

The original gptzfsboot was built with custom functions to reduce its size, at the price of having duplicate code to maintain. Since we do not have strict size limits for stage 2, we can use generic libstand and share the code base with stage 3. This will simplify building the updates to both stages.

Changes to the UFS File System

Both FreeBSD and illumos have UFS file system, but since both operating systems have been developed on separate paths, there are some differences developed over the time. FreeBSD actually has two version of UFS, version 1 and version 2, and in fact, the version 1 is still quite close with illumos version. From point of view of how the on-disk structures are set, the differences are in some areas of the superblock and i-node. After adjusting the superblock and i-node definitions, virtually the same code can be used to read the illumos UFS.

Changes to the dosfs

The dosfs is the implementation of FAT 12/16/32 file system in the boot loader. The implementation was using little caching and the reads were quite slow. I did implement the first round of updates and other developers did the follow-up work to fix some bugs created by me and introducing further optimizations.

Changes to the ZFS Reader

To support illumos kernel to boot from ZFS root, we need to provide boot dataset name or object ID and physical path of the device, extracted from pool label on the disk. We implement zfs_bootfs() function to extract the needed information and to export it via boot loader environment. The difficulty here is to make sure we do provide the information about usable and most recent member of the pool, or the kernel ZFS implementation will reject the pool.

The original ZFS reader implementation was reading from the disk only the first instance of the pool label, but there is in total four copies of the pool label on each member disk device. Two copies are located at front to the pool data and another two copies are located at the end of pool data area, providing the redundancy. Because it is possible that some of the copies have failed to receive an update with most up to date information about the pool configuration, we must process all copies of the label and select one with most recent information.

The other updates are about enabling support for the extra features provided by the ZFS and which were disabled before. Notably support for gzip decompression – since we do need to support decompression of the boot archive, we do have gzip code in boot loader already and

therefore adding gzip decompression support for ZFS is virtually free. For technical reasons, implementing the support for the large blocks is trivial – we only need to be prepared to allocate large blocks. And the sha512 hashing algorithm is easy to add next to existing sha256.

Also, the salt-based checksum (edonr and skein) support is prepared but not yet integrated into illumos – however the support for skein is already integrated into the FreeBSD.

5.4 Scripting for illumos

Important part of boot loader user interface is implemented in support scripts, that includes preparation to load the kernel and menu interface. In current implementation, the scripting language used is Forth.

The kernel of the illumos is using set of variable-value pairs to define initial setup of related components. This set of variables is implemented via emulated eeprom interface and provided via text file "/boot/solaris/bootenv.rc".

Our goal is first to read in the variable value pairs from the emulated eeprom, this will provide us the default values for variables. Then we need to allow boot loader environment to update the variables and values, and lastly, we provide the pairs to the kernel. In boot loader, we present the configuration variable and value pairs in form of environment variables.

Fortunately for us, the boot loader support scripts already implement the parser for reading variable – value pairs. We only need to add support to handle the constructs used in "/boot/solaris/bootenv.rc" file. This update is implemented as word "include_bootenv" in the script, installed as "/boot/forth/support.4th"[13].

The boot loader does provide multiple locations of configuration files to support setting and updating the variables:

1. /boot/solaris/bootenv.rc
2. /boot/defaults/loader.conf
3. /boot/loader.conf
4. /boot/loader.conf.local

Additionally, I did add support for configuration snippets from /boot/conf.d/* and finally, the /boot/transient.conf.

The /boot/defaults/loader.conf is designated location for system wide defaults, provided by the operating system distribution. This file usually does provide some important default values and commented out examples.

The /boot/loader.conf and /boot/loader.conf.local can be used to override defaults in a well-known location. This is convenient way to implement local instances or setting non-default values when we cannot use wildcards, such as in case of tftp based boot (as tftp does not implement support for directory listing).

The /boot/conf.d/* is convenient way to separate the settings into the different custom files.

And finally, the /boot/transient.conf is processed last, therefore its content will override any previous setting and this configuration file is used by transient boot, when we want to change the system behaviour for one time only. The /boot/transient.conf file is removed automatically when the system will enter svc:/milestone/multi-user:default state.

[13] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/forth/support.4th

This multiple configuration file setup will allow us to use very flexible changes, but it also complicates processing the values via management programs. To solve this problem, the management software, such as implementation in libbe and in command bootadm, are calling libficl-sys to interpret the current configuration and then the information can be extracted from the command environment.

To simplify handling the boot options in the boot menu system, we also do translate set of kernel options to variables and vice versa. When the boot loader is reading and parsing the configuration files, the special variable, "boot-args" can be set with kernel command line switches and the switches there are translated to individual variables. When the kernel will be started by boot command, the variables are translated back to kernel switches and the resulting options list is passed to the kernel as kernel command line.

This approach will allow us to implement the boot loader options menu just to manipulate with environment variables. The update is implemented as words "set-boot-args" and "parse-boot-args" in file "/boot/forth/loader.4th"[14].

Boot Menu

To support illumos, the boot options submenu is updated to allow the common kernel options to be set or changed. This does include options controlling the OS console, ACPI options and some others.

The "Select Boot Environment" menu in FreeBSD implementation is reading the list of the boot environment datasets from the ZFS pool and is presenting the list in the menu. In illumos, we maintain the list of the boot environments in the file located in the pool root dataset, and named as "/boot/menu.lst", changed from presenting ZFS datasets to process and present the content from the file "zfs:poolname:/boot/menu.lst". This does allow us better control over BE order and we can add custom entries.

The entry in menu.lst file had syntax as following:

title *title name*

bootfs *pool/dataset1/dataset2*

This entry does declare the entry for boot environment from given dataset. We did update this construct to add keyword **chain**, to allow creating an entry for chain-loading, and we did allow to specify the disk specification like "diskX:" or "diskXpY:" to be provided for chain loading or to be used as non-zfs boot device.

[14] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/forth/loader.4th

With added features, let's have look on example of the boot environment menu setup. The content of the file "/rpool/boot/menu.lst" is shown in Figure 5.

```
title Disk1
chain disk1:
title Disk2
chain disk2:
title Disk3_ufs
bootfs disk3s1a:
title oi-487
bootfs rpool/ROOT/oi-487
title oi-488
bootfs rpool/ROOT/oi-488
```

Figure 5. Example of the menu.lst File

We can get information about boot menu from the operating system by using the command "bootadm list-menu", the output of this command based on the contents of the menu.lst file above is shown in the Figure 6.

```
$ bootadm list-menu
the location for the active menu is:
/rpool/boot/menu.lst
INDEX NAME      DEVICE            TYPE   DEFAULT
0     Disk1     disk1:            chain     -
1     Disk2     disk2:            chain     -
2     Disk3_ufs disk3s1a:         bootfs    -
3     oi-487    rpool/ROOT/oi-487 bootfs    -
4     oi-488    rpool/ROOT/oi-488 bootfs    *
```

Figure 6. The listing of the boot menu

And finally, we will get the Boot Environments menu in the boot loader, and by selecting the corresponding menu entry, the boot loader will either activate the selected boot environment or will execute the chain loading based on device name. The example screenshot is shown in the Figure 7.

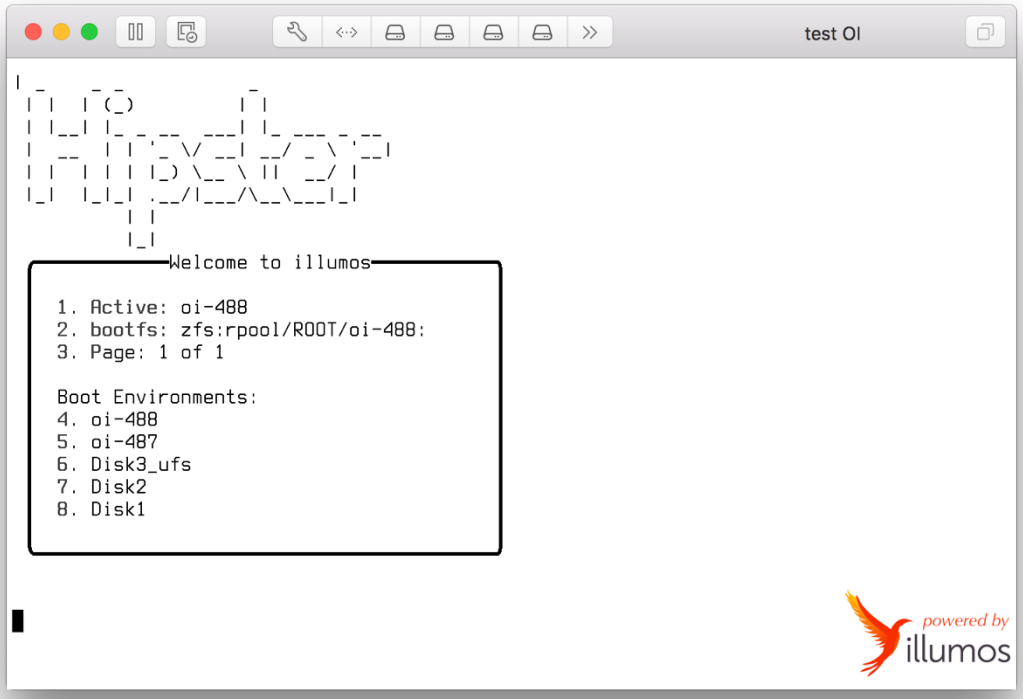

Figure 7. Boot Environments menu

Boot Environments

The boot environment is device with root file system. In our case it is also important fact that boot environment also does contain the /boot directory with boot loader scripts and configuration settings. To implement the boot environment switch, we need to release the currently allocated resources (memory and unload the possibly loaded modules), then set the device to load new files, and read in the configuration from the new device. The boot loader does implement two alternate ways to switch the boot environment. First option is to use the select boot environment menu entry, second option is to use boot loader command

line command “beadm”. The beadm command does implement options to get a listing of boot environments and option to activate the given boot environment.

The boot environment management is implemented partly in “/boot/forth/menu-commands.4th”[15] and mostly in “/boot/forth/beadm.4th”[16]. The “set_bootenv” word in Figure

```
\
\ Select a boot environment
\

: set_bootenv ( N -- N TRUE )
        dup s" bootenv_root[E]" 13 +c! getenv
        s" currdev" getenv compare 0= if
                s" zfs_be_active" getenv type ."  is already active"
        else
                dup s" set currdev=${bootenv_root[E]}" 27 +c! evaluate
                dup s" bootenvmenu_caption[E]" 20 +c! getenv
                s" zfs_be_active" setenv
                ." Activating " s" currdev" getenv type cr
                s" unload" evaluate
                free-module-options
                unset_boot_options
                s" /boot/defaults/loader.conf" read-conf
                s" /boot/loader.conf" read-conf
                s" /boot/loader.conf.local" read-conf
                init_bootenv

                s" 1" s" zfs_be_currpage" setenv
                s" be-set-page" evaluate
        then

        500 ms                  \ sleep so user can see the message
        be_draw_screen
        menu-redraw
        TRUE
;
```

Figure 8. Select BE from menu

8 is implementation of the logic described above. We set the special variable “currdev”, then release the previously allocated resources and read in the configuration from new device, and finally, redraw the screen.

5.5 Chain Loading

Chain loading is a process where we load the next boot loader and execute it. Since we support both UEFI and BIOS platforms, we will look here both cases.

We need chain loading for two reasons – first, it is another useful feature for users to make life a bit easier, and secondly, it will give us another tool for testing, so we can start our components from the controlled environment.

For UEFI boot loader, we have the firmware API support [4, p. 242] to use. For the purpose of starting new application for UEFI program, we have two callbacks from the UEFI BootService:

1. LoadImage()

[15] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/forth/menu-commands.4th
[16] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/forth/beadm.4th

2. StartImage()

To prepare, we need to allocate the memory and read the new binary into the memory, then pass this memory as argument for LoadImage() call.

Then we need to set up the LoadOptions and DeviceHandle for the loaded image and as an final step, we use StartImage() to pass the control to new program. The UEFI system does not replace existing program in the memory. If the newly loaded program will exit, our application will continue the execution.

For BIOS case we have no specific API to use. We need to load the chainloaded boot loader to address $0000:7C00, switch the machine to 16-bit real mode and jump to address $0000:7C00.

To simplify our case, we assume that we only chainload 512-byte MBR or partition boot record or its equivalent in file and the code there will take care about the rest.

This assumption does allow us to use small amount of low memory and preserve our boot loader in case something will go wrong during the loading.

We will load the chainload block into unused memory location at 1MB line (physical address 0x100000), set up the relocator code block with proper values and copy it to physical address 0x600 and will jump to physical address 0x600. This part of the implementation is in file "chain.c"[17].

As the first step, the relocator code has to set up new Global Descriptor Table (GDT), so we won't blow ourselves up with fault while moving the data into the place and overwriting the parts of our current boot loader.

Once we are safe, we will move our 512B data block into the target memory area (starting from $0000:7C00).

When the data is in place, we will switch the system to 16-bit real mode. The switch itself is quite well described in the osdev.org wiki[18]. In addition to the steps described in the wiki, we also need to update the interrupt controller and update the A20 gate.

The relocator code is located in the file "relocater_tramp.S"[19]. Note, we also use the same relocator to load and start linux kernel – although it is created specifically keeping in mind the syslinux case and to support standalone applications, such as memtest86.

[17] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/i386/loader/chain.c
[18] https://wiki.osdev.org/Real_Mode
[19] https://github.com/illumos/illumos-gate/blob/master/usr[4]/src/boot/sys/boot/i386/libi386/relocater_tramp.S

6 Building Support for UEFI in the Kernel

From point of view of starting the kernel in UEFI platform, we have several challenges to work out. In this chapter, we will only attend the details from the point of view of the operating system kernel.

Firstly, we do not have BIOS services to use to discover our hardware resources. The services include interrupt calls and BIOS Data Area. This means, the kernel of our operating system has to identify the system platform early and avoid all BIOS specific services and data. UEFI platform does provide vital system data via EFI System Table [4, p. 106], which is passed by boot loader to the kernel with multiboot info tag for EFI:

```
typedef struct multiboot_tag_efi64 {
        uint32_t mb_type;
        uint32_t mb_size;
        uint64_t mb_pointer;
} multiboot_tag_efi64_t;
```

From the operating system point of view, the most interesting parts of EFI System Table are pointer to Runtime Services and Configuration Table. This is because Console input and output related pointers and Boot Services pointer will become unusable once the Boot Services are switched off. Such switch off usually will be done right before giving control to the loaded kernel. The UEFI Runtime Services are described in [4, p. 267] and the access to Runtime Services is managed by the operating system kernel.

The Configuration Table consists of GUID/pointer pairs for individual vendor tables, such as ACPI [5, p. 121] and SMBIOS [6, p. 22] tables. The exact number and types of tables depend on the vendor and the system.

To prepare the ground to make it possible to distinguish if we are running on BIOS or UEFI and to discover the resources provided by the system, we need to check if the multiboot_tag_efi64 or multiboot_tag_efi32 is present in multiboot information tag list and extract the information. The UEFI system may run either in 64-bit mode or 32-bit mode, and since the configuration is using firmware native mode pointers, it is important to recognize the mode.

Once we have detected and set up the basic configuration, it is quite easy to implement configuration checks based on the current platform.

To find all cases in source where the BIOS data is accessed, two methods was used. Firstly, the simple search for BIOS interrupt call usage. Since the BIOS interrupt mechanism is quite specific, it was simple task to identify all the cases. Secondly, we had to detect the access to BIOS Data Area (BDA). Fortunately, the BDA is located in low memory – within the first megabyte. To detect the access to BDA, I did remove the memory mapping to the first megabyte, which did trigger the kernel panic on any access to this memory area. This technique did help to identify all cases of known BDA accesses.

Cleaning up and isolating the platform specific code made it possible to have the ability to start up the kernel and the operating system on UEFI system, even as we do not yet have achieved the full support for the platform. The next steps are to build the support for local console, Runtime Services, and secure boot.

Since the access to the system console is vital, we will dedicate next chapter to the console.

7 The Console

The common solution for console access is via local console, that is the monitor and the keyboard attached to the system. In case of server systems, the console is often implemented via serial connection. Since we already have usable serial console, we will only focus on local console.

By local console we mean the input and output facilities providing the interface for human interaction and with ability to control the system starting from the early startup. The console consists of output device, usually screen connected to the graphics card, and input device(s), keyboard and pointing device.

The current loader and illumos console implementation is using VGA text mode on x86 BIOS platform, however, the UEFI firmware does not provide traditional VGA text console, because the specification does not require the VGA BIOS to be present. The common practice used in UEFI systems is to provide SimpleTextOutput protocol to implement the text output to the screen. SimpleTextOutput Protocol is implementing the text terminal interface, usually stacked on top of Graphics Output Protocol [4, p. 509], which is using linear frame buffer as the output device. Since the UEFI console related protocols are only available when UEFI Boot Services are enabled, we cannot access SimpleTextOutput protocol nor Graphics Output Protocol from the operating system kernel. Therefore, we need to implement linear framebuffer-based console for our operating system.

7.1 The Console in the Boot Loader

The boot loader is implementing a simple terminal emulator for UEFI and for BIOS, the entry points are efi_term_emu()[20] for UEFI and vidc_term_emu()[21] for BIOS case. Both are implementing the same terminal type with the same set of features. Both are using the text output interfaces behind the scenes – SimpleTextOutput or BIOS INT 10h function family.

From design point of view, as we do get the framebuffer from UEFI, we would also like to have option for framebuffer for BIOS case, so we could have uniform user interface. Fortunately for us, we can use Vesa BIOS Extensions (VBE) for just that purpose, specifically, to set up linear frame buffer on BIOS platform.

For the VBE setup, we read the supported mode list from the monitor via EDID [7] and will try to use the default mode suggested by the monitor. If EDID is not available, we default to 800x600x32 to follow the suggestion from UEFI specification [4, p. 586].

The mode can be changed from loader prompt or from script by the “framebuffer” command, as shown in Figure 9.

The implementation for VBE specific graphics detection, setup and initialization is in the file vbe.c[22]. This instance is based on source from NetBSD and is quite heavily modified for our purposes.

The approach to provide VBE framebuffer for BIOS platform does also mean some necessary side changes. When chain loader is used, we need to switch system back to VGA text mode before jumping to new boot loader. In a similar way, we need to detect if our loaded kernel can support framebuffer console or not. Fortunately, at this time, we can use few simple checks. If the kernel does not support multiboot 2 boot protocol, we switch to text mode before jumping to the kernel. If the kernel does support multiboot 2 boot protocol, we

[20] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/efi/libefi/efi_console.c
[21] https://github.com/illumos/illumos-gate/blob/master/usr/src/boot/sys/boot/i386/libi386/vidconsole.c
[22] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/i386/libi386/vbe.c

can check if the framebuffer information is requested by the kernel via providing MULTIBOOT_TAG_TYPE_FRAMEBUFFER in the multiboot header tag information request packet. If the MULTIBOOT_TAG_TYPE_FRAMEBUFFER is not present, we switch to VGA text mode before jumping to the kernel.

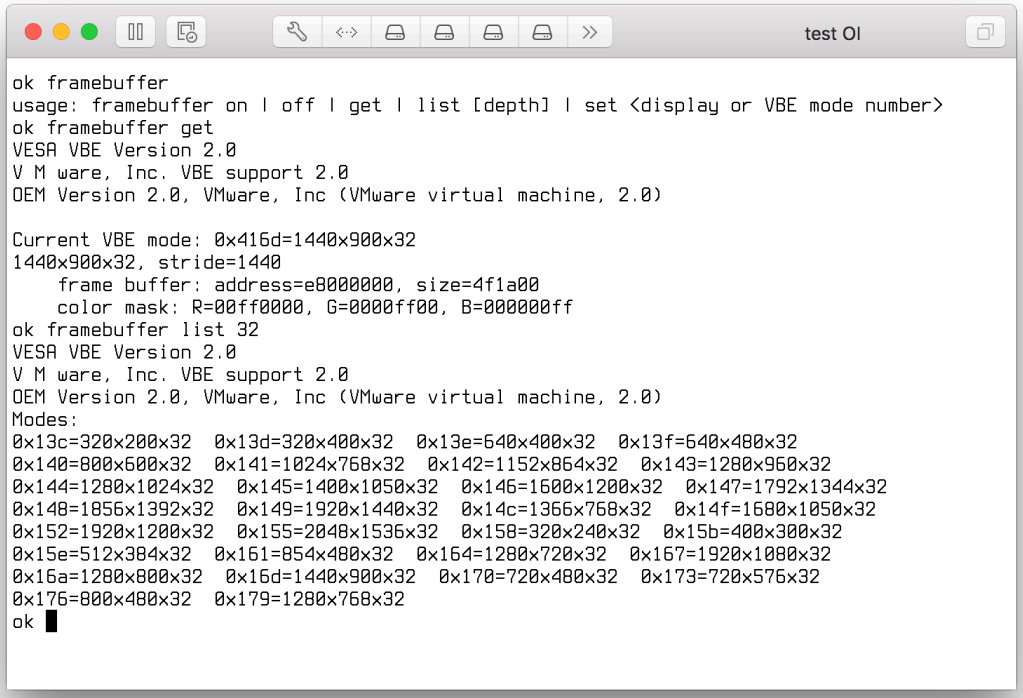

Figure 9. framebuffer command use cases

The function to set text mode is quite simple, first we set the mode by calling BIOS INT 10h, then we set text attribute mode and finally, we set the values in global variables to reflect the state of the system. We record the VBE state and the framebuffer specific information, which is used later to set up the multiboot2 framebuffer information tag. The excerpt of the function is presented in Figure 10.

In case of UEFI platform, we actually do have two protocols implementing the graphical console interfaces. The initial EFI[23] specification did define UGA (Universal Graphics Adapter) interface, but in UEFI specification it was replaced by GOP (Graphics Output

[23] https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface - Graphics_features

Protocol) and now most systems only provide GOP mechanism. By my knowledge, only early intel-based MacBook laptops from Apple did provide the UGA interface.

Even as we have very basic code to detect and set up the UGA based framebuffer, since the UGA is not commonly used, we are not implementing proper support for UGA at this time.

For GOP, we are also implementing the "framebuffer" command[24], and it is implementing the same options and features as done for VBE counterpart. The difference is that GOP interface does only provide support for 32-bit depth.

The multiboot 2 request package is processed by for loop in multiboot2_loadfile() function[25]. If we have framebuffer info requested, we record this fact in have_framebuffer boolean variable and later can use it in multiboot2_exec() function[26] to decide if and how we need to provision the related information. The described above mechanism does allow us to provide the framebuffer and related information to the kernel when it is actually needed.

By implementing the VBE based linear framebuffer, we have graphical console interface, usable for both UEFI and BIOS platforms, also both platforms have option to use text console (with exception that in case of UEFI, the text console is actually built on GOP) and now we can start to build next layer to support the console output.

```
/* Actually assuming mode 3. */
void
bios_set_text_mode(int mode)
{
        int atr;

        v86.ctl = V86_FLAGS;
        v86.addr = 0x10;
        v86.eax = mode;                         /* set VGA text mode */
        v86int();
        atr = vga_get_atr(VGA_REG_ADDR, VGA_ATR_MODE);
        atr &= ~VGA_ATR_MODE_BLINK;
        atr &= ~VGA_ATR_MODE_9WIDE;
        vga_set_atr(VGA_REG_ADDR, VGA_ATR_MODE, atr);

        vbestate.vbe_mode = 0;                  /* vbe is disabled */
        gfx_fb.framebuffer_common.framebuffer_type =
            MULTIBOOT_FRAMEBUFFER_TYPE_EGA_TEXT;
        /* 16 bits per character */
        gfx_fb.framebuffer_common.framebuffer_bpp = 16;
        gfx_fb.framebuffer_common.framebuffer_addr =
            VGA_MEM_ADDR + VGA_COLOR_BASE;
        gfx_fb.framebuffer_common.framebuffer_width = TEXT_COLS;
        gfx_fb.framebuffer_common.framebuffer_height = TEXT_ROWS;
        gfx_fb.framebuffer_common.framebuffer_pitch = TEXT_COLS * 2;
}
```

Figure 10. Set VGA text mode

Drawing Text on the Framebuffer

The linear framebuffer will provide us the array of pixels defining the drawable area with coordinates (0,0) at the upper left corner and (width – 1, height – 1) at lower right corner. To set the pixel in this area, we need to calculate the offset for the pixel bytes using the pixel

[24] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/efi/loader/framebuffer.c - L712
[25] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/common/multiboot2.c
[26] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/common/multiboot2.c

coordinates and the colour depth, then write the colour value to the memory using the calculated offset.

The simple option to draw the text is to use bitmap font, where the glyph is described by array of bytes, where bit value 0 indicates the pixel not set, and bit value indicates the pixel set. Using bitmap glyph description combined with foreground and background colours, we can draw individual chars.

For fonts to use I have selected the fonts from terminus family[27], which is fixed width bitmap font with rich set of glyphs, supporting about 120 language sets and sizes 6x12, 8x14, 8x16, 10x18, 10x20, 11x22, 12x24, 14x28 and 16x32.

Additionally, the traditional Sun console font, Gallant[28], was transformed to bdf format, some glyphs added and made available as ISO10646-1 (Unicode) font.

With such a rich selection of glyphs and sizes, we cannot build all the supported fonts into the boot loader binary, so the decision was made to only use 8x16 built in font and only to cover the glyphs from ASCII set. This will make sure we can present the information on the screen even if we will experience read errors while reading support files. Also, we do compress the built-in font with LZ4 compression, the resulting compressed font bitmap is using only 1007 bytes.

The initial framebuffer-based console setup will uncompress the built-in font and will set up the needed data structures to make it possible for console implementation to start using the built-in font.

Once the boot loader startup has detected the storage resources, we will perform the secondary initialization for the console by reading the index of available fonts from the disk, then we will select the best fitting font for the current framebuffer resolution and we will read into the memory the complete font for the selected size.

Similar autodetection and font reload will happen when the user will change the screen resolution.

The user can also use manual font selection by either using “fontload” command or by setting screen-font variable.

If we have successfully loaded the full font, to ensure the consistent console output, we will pass the loaded font image to kernel as a multiboot 2 module, otherwise kernel will use the font built into kernel the same way as described above and will use its own built in mechanisms to load the full font data.

The glyph drawing code source is shared between the boot loader and the kernel[29], the functions to draw the glyph are font_bit_to_pix8(), font_bit_to_pix16(), font_bit_to_pix24() and font_bit_to_pix32().

Terminal Emulation

The terminal emulator in original loader code does have duplicated implementation (see section 7.1) and is quite simplistic. However, to provide consistent console experience, we would like to have the boot loader and kernel to have as close as possible behaviour for console terminal handling. To reduce the duplication, I did replace the old terminal

[27] http://terminus-font.sourceforge.net/
[28] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/data/consfonts/Gallant19.bdf
[29] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/common/font/font.c

emulation implementation in the boot loader with the code from the illumos kernel, tem.c[30]. However, the replacement is not just one to one copy, but somewhat simplified approach as we do not have to deal with threads and synchronization.

This approach does provide us some more features for terminal emulation, the tools to provide the meaningful state of the console for the kernel terminal emulator, allowing us to build consistent look for transition from boot loader to the operating system and we can use single instance to implement both UEFI and BIOS platform support and to support both text and graphical mode console.

The entry point to terminal emulator is tem_write() but before we can use tem, it has to be initialized and activated. The tem initialization is implemented in platform specific modules, efi_console.c[31] and vidconsole.c[32].

The tem is using call-backs mechanism to implement the screen related functions:

```
typedef struct tem_callbacks {
        void (*tsc_display)(struct tem_vt_state *, term_char_t *, int,
            screen_pos_t, screen_pos_t);
        void (*tsc_copy)(struct tem_vt_state *,
            screen_pos_t, screen_pos_t, screen_pos_t, screen_pos_t,
            screen_pos_t, screen_pos_t);
        void (*tsc_cursor)(struct tem_vt_state *, short);
        void (*tsc_bit2pix)(struct tem_vt_state *, term_char_t);
        void (*tsc_cls)(struct tem_vt_state *, int, screen_pos_t, screen_pos_t);
} tem_callbacks_t;
```

The tem call-backs are to implement text or pixel mode functions, tem call-backs in turn are using call-backs from visual_io interface to actually manipulate the display to draw the text, copy the regions of the display and manipulate the cursor:

```
struct vis_polledio {
        struct vis_polledio_arg *arg;
        void    (*display)(struct vis_polledio_arg *, struct vis_consdisplay *);
        void    (*copy)(struct vis_polledio_arg *, struct vis_conscopy *);
        void    (*cursor)(struct vis_polledio_arg *, struct vis_conscursor *);
};
```

The visual_io call-backs are requested by tem_info_init(), and the visual_io call-backs are passed by console interfaces depending on if we are using text mode or framebuffer mode. The same request will also provide the console interface the call-back to notify tem about mode change when the mode is switched between text and framebuffer or when the graphics resolution is changed.

The user can influence the tem by switching between text and framebuffer mode, or by changing framebuffer resolution, or by selecting the font manually. The mentioned actions do change the terminal resolution. Additionally the user can set variables “tem.inverse”, “tem.inverse-screen” to inverse the text or screen colours and “tem.fg_color” and “tem.bg_color” to set foreground and background colours for the text. The settings mentioned above are also passed to kernel and the kernel tem will use the values to set itself up.

[30] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/common/tem.c
[31] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/efi/libefi/efi_console.c

[32] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/i386/libi386/vidconsole.c

The framebuffer specific visual_io callbacks gfx_fb_cons_display(), gfx_fb_cons_copy() and gfx_fb_cons_clear() are implemented in gfx_fb.c[33]. The current implementation is manipulating directly the data in the mapped frame buffer memory, the future work would need to investigate the implementation based on GOP Blt() function, because it is possible we only get the software based GOP implementation, without the access to the memory mapped framebuffer.

Graphics Primitives for Boot Loader

To support using graphical elements on the screen, we do implement small set of forth[34] words to be used with framebuffer-based console. The words are:

```
fb-setpixel (x, y -- )
fb-line (x0, y0, x1, y1, width -- )
fb-bezier (x0, y0, x1, y1, x2, y2, width -- )
fb-drawrect (x0, y0, x1, y1, flag -- )
fb-putimage (addr, len -- flag )
term-drawrect (x0, y0, x1, y1 -- )
```

All words are implemented as wrappers for C functions, implemented in the gfx_fb.c[35]. The implementation is based on Bresenham algorithm modification described in [8].

- fb-setpixel is the word to set the single pixel.
- fb-line will draw the straight line between two endpoints and with given width.
- fb-bezier will draw bezier line with one middle point (the current implementation is limited and needs fixing).
- fb-drawrect will draw the rectangle defined by upper left and lower right corner and optionally filled by foreground colour.
- fb-putimage will put the PNG format image to the lower left corner of the screen. Eventually the fb-putimage should allow to specify the location and the size for the image, for this we would need to implement the image scaling function, something like bilinear interpolation[36]. For better image handling, the screen drawing is also using alpha blending, but only for 32-bit depth case.
- term-drawrect will draw the rectangle with rounded corners using the text terminal coordinates and using line width proportional to the glyph size.

This set of framebuffer related words does allow us to draw the basic menu and will serve the demonstrational purposes.
Finally, while preparing to start kernel, we will build the multiboot2 module from boot loader environment variables, and one step of the process will call tem_save_state() function to make sure we have cursor position recorded.

7.2 The Console in the Illumos Kernel

At first, we need to explain the basic components and steps involved with kernel startup. Since legacy Grub boot loader was only supporting loading and starting 32-bit kernel, special 32-bit component was developed to extract information from multiboot 1 boot protocol,

[33] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/common/gfx_fb.c
[34] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/common/ficl/loader.c
[35] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/boot/sys/boot/common/gfx_fb.c
[36] https://en.wikipedia.org/wiki/Bilinear_interpolation

to set up the needed data, including the kernel segments, to switch to 64-bit mode and to start the actual kernel. This 32-bit module is called dboot (direct boot) module, and is needing the console access for diagnostic messages, when kernel is started with prom_debug=true or map_debug=true or kbm_debug=true properties.

To start the real kernel, dboot code does jump to "locore" part of the kernel. Once the jump is done, the dboot code is never used or needed.

The locore code will start to initialize the needed data structures and to prepare the kernel for loading drivers, starting up other CPU(s) and to mount the root file system. During this setup, the console input and output is needed for diagnostic messages (see dboot above), for kernel debugger (when booted with -kd command line switch) and to print out the kernel name, version and copyright notice. All this console input and output is performed by early boot mini drivers for keyboard, display (or serial port). Only after the kernel has prepared to be able to load drivers, we will get the real console driver up and running and after that, the early boot console code is not used any more.

The dboot module is built to share the console code source with early kernel startup, but in binary form, we have three instances of console code:

1. The 32-bit code in dboot.
2. The 64-bit early boot code (source[37] shared with dboot).
3. The driver used with live kernel.

The Mini Driver for Early Boot

The dboot and early boot (locore) are needing the console mostly for diagnostics, but since we do not have the full access to all the resources yet, we only implement very simple skeleton drivers to support early boot. We already have mini drivers for keyboard and VGA text display, but we need to add a mini driver for the framebuffer.

The most basic screen support is about to be able to put char(s) on the screen and scroll the screen. However, it is not entirely enough for our needs, because we also need to support the kernel debugger and for this we need a bit more screen handling.

To simplify the task, much simplified terminal emulator is implemented with call-backs to display management.

```
/* Console device callbacks. */
typedef struct bcons_dev {
   void (*bd_putchar)(int);
   void (*bd_eraseline)(void);
   void (*bd_cursor)(boolean_t);
   void (*bd_setpos)(int, int);
   void (*bd_shift)(int);
} bcons_dev_t;
```

The call-backs implement putchar() to put char to screen, eraseline() to clear the line from cursor to end of the line, cursor() to hide or unhide the cursor, setpos() to move the cursor to given position, shift() to shift the content of the line to left or right.

The boot terminal emulator is implemented in boot_console.c[38], with entry function btem_parse().

[37] https://github.com/illumos/illumos-gate/tree/master/usr/src/uts/i86pc/boot
[38] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/i86pc/boot/boot_console.c

To provide better support for kernel debugger, the keyboard mini driver was updated to emit escape sequences for arrow keys.

The VGA mini driver is relying on either built in font or font loaded by boot loader, so we do not have to worry about font in case of VGA text output.

The framebuffer-based console implementation is in boot_fb.c[39]. To access the framebuffer, we have to extract the framebuffer related information from multiboot information tag, if it exists, then we can build the needed data structures.

Also, we will check if the font was passed in multiboot module and set it up in font list. If the font was not passed by the boot loader, we will fall back to built-in font, as at this stage, we have no way to access other media.

Once the locore boot has set up the method for memory allocation, it will also allocate the shadow buffer for frame buffer console, allowing us to operate on RAM and get the performance boost by avoiding the accesses to slower memory in graphics card.

Note that in early boot we have no mechanism to switch the console mode or resolution, we only rely on what we inherit from the boot loader.

We also fetch the state of the tem from the boot loader environment passed to us via multiboot module, so we can set up the cursor and terminal foreground and background colours accordingly. Since the early boot console implementation is shared between dboot and locore, we can only use boot loader information from dboot console and we need to pass this information to locore via internal xboot_info[40] data structure, and later, when kernel tem is starting, it will read the early console state to provide the consistent console experience.

Framebuffer Device Driver

The current driver implementing the access to display devices in illumos is vgatext[41], which does implement only the text mode. The vgatext driver is attached to PCI Display Controller Class, VGA/XGA compatible controllers and is providing the visual_io interface functions via ioctl() interface and the visual_io interface is used by the kernel terminal emulator (tem).

Since the driver attachment is based on device identificator or class but not on device specific mode, we would need to extend the implementation in a way to handle both text and framebuffer modes.

The illumos is result of the fork from the OpenSolaris project and one side effect of the fork is that we also did inherit the half-implemented projects from OpenSolaris. One of such projects is gfx_private module[42], which is the starting point to implement the device mode specific functions in a form of API. Since the module is already there and currently unused, I did decide to pick it up and to continue the development from where it has been cut.

The very first step was to complete the migration of functionality from vgatext module to gfx_private and implement vgatext to be consumer for gfx_private. Such transformation was quite simple because most vgatext functionality was already present in gfx_private and all what was needed to be done was to add the missing parts and to make sure we have the same features implemented and the implementation has not changed. The transformed

[39] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/i86pc/boot/boot_fb.c
[40] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/intel/sys/bootinfo.h
[41] https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/intel/io/vgatext/vgatext.c
[42] https://github.com/illumos/illumos-gate/tree/master/usr/src/uts/i86pc/io/gfx_private

vgatext[43] is implemented to use gfx_private API and as a side effect, we have reduced the code duplication.

The next step is to build the visual_io implementation for the bitmap-based console to support the framebuffer mode. We have the console set in linear framebuffer mode by the system firmware (UEFI) or by the boot loader (VBE), we have the framebuffer memory address, framebuffer size and other related details and we implement entirely software-based rendering for framebuffer. Since we are using the same terminal emulator in boot loader and in the terminal, the visual_io call-backs in the kernel are also same as we already did see in the boot loader:

```
struct vis_polledio {
        struct vis_polledio_arg *arg;
        void    (*display)(struct vis_polledio_arg *, struct vis_consdisplay *);
        void    (*copy)(struct vis_polledio_arg *, struct vis_conscopy *);
        void    (*cursor)(struct vis_polledio_arg *, struct vis_conscursor *);
};
```

First, we need to have some "housekeeping" support. For graphics DRM module[44], we need to provide two functions:

1. gfxp_bm_register_fbops() – to register the frame buffer related operations provided by DRM module. At this time, we only receive and use the mode change call-back. The remaining call-backs should be implemented at later phases.
2. gfxp_bm_getfb_info() – to inform DRM module about the framebuffer mode (resolution, bits per pixel and colour depth).

For illumos module API, we need to provide attach() and detach() entry functions.

To handle both UEFI and VBE frame buffers, we implement colour depths 8, 15, 16, 24 and 32. To improve the performance, the screen related functions are using shadow frame buffer.

At this time, we do not have method to change the console mode. Whichever mode we get from boot loader, we have no way to change it. The only mechanism available is KDIOC ioctl functions to set text mode or graphics mode, and is used by X11 to notify the system if the console is in use or released, and accordingly, we notify the DRM layer via setmode() call-back.

Since we do not have the mechanism for console mode change, we also do not notify the tem about mode change. This functionality should be added at later development stages.

As we have created the functions to handle the bitmap framebuffer[45], we only need to add the switch mechanism to detect if we can use framebuffer or VGA text mode and provide the updated functions[46] for use by the consumers (such as vgatext driver).

Loading Fonts

In current implementation the kernel has three options to receive the font, from multiboot module, passed to kernel by boot loader or built in default font. For x86 platform, the built-in font is 8x16, to make it possible to load it to VGA adapter for text mode.

[43] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/intel/io/vgatext/vgatext.c
[44] https://github.com/illumos/gfx-drm
[45] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/i86pc/io/gfx_private/gfxp_bitmap.c
[46] https://github.com/tsoome/illumos-gate/blob/loader/usr/src/uts/i86pc/io/gfx_private/gfxp_fb.c

Since the kernel and the boot loader are sharing the source with font functions, the base mechanism for font loading is there, the actual file access mechanism is yet to be implemented (not too hard task as its already done in boot loader).

The VGA text mode is even more interesting task. Since we are using Unicode fonts, we need to select glyphs from specific code page, at this time the implementation is using CP437. Again, the support to set different code page is planned for future development – we need to implement ioctl mechanism to provide the character mapping table to be used instead of CP437. Of course, assuming the VGA [text mode] will disappear in not too distant future, there is a question if such mechanism is worth to be implemented.

TEM and Unicode Chars

We are now stepping a bit out from the boot loader topic, but not too much, since we just have borrowed the terminal emulator from the kernel for the needs of the boot loader, and even as we do not directly share the code, we still do track the functionality.

Also, we just did provision our kernel with Unicode font, so it would be nice we could actually take advantage of.

The tem was built on assuming 8-bit chars, and is using buffers for screen, foreground and background colours. With Unicode we get 32-bit chars, however, only 21 bits are actually used. This fact does allow us to use some optimization – we can merge the char value with tem text attributes.

The combined attributes will allow us to simplify the buffer management but also, we can use attribute values while working with font data.

Things to Do

The current implementation is still lacking the access to the EFI Runtime Services. To implement it, we need to implement first the support to maintain context for FPU, MMX and XMM registers in kernel. We also need to figure out how to deal with systems with UEFI32 firmware, but able to run 64-bit kernel. To call Runtime Services from 64-bit kernel, we would need to switch the machine to 32-bit protected mode first and back to 64-bit long mode after.

Since there is no access to the UEFI Runtime Services, we are also missing the userland tools to access the EFI boot manager variables.

For UEFI side, the secure boot support would be the next big target, but it is also more about the organizational work to maintain key management etc.

Other things are further possible development to implement better support and features. For example the boot loader could use a custom keyboard map. The primitives to handle the screen would also need improvements.

The menu system would need some improvement to make it easier to create custom entries and dialogs.

8 Conclusions

In this paper we did walk through the process of porting and developing the boot loader from FreeBSD to illumos. We gave an overview of the basic components of the boot loader, how those components do interact and how improving the parts can influence the whole.

We did add the multiboot 2 boot protocol support for boot loader and the illumos kernel, we did rebuild the console implementation for both boot loader and illumos, even as the actual integration is still in progress in illumos. We did enable the UEFI support for illumos and along with it a series of development opportunities.

The actual integration of this work with the official illumos source repository is still on-going process, we do have integrated the new boot loader and we have implemented the base UEFI support but there is still no framebuffer console support. However, the goals mentioned in the introduction are fulfilled.

The ported boot loader does provide the easy way to extend and add new functionality (such as to add new ZFS features). The base of the port and integration was integrated with illumos source and is already in use by several illumos based distributions and has received a lot of positive feedback. As any software project, there are also bugs and issues, users can report the issues via illumos (and FreeBSD, because many of the updates are integrated in both operating systems) bug reporting web-based interfaces.

This whole project has been lasting well over two years, during which we have added many improvements the code, ported changes between the FreeBSD and illumos; both platforms have not only won in terms of technology but there is also much more cooperation between people from both camps.

For me, this journey has been pleasant opportunity to refresh my memories and skills about different languages (C, assembler, forth), dig into the details of different technologies (from system firmware, filesystems, computer graphics, fonts etc) and to have chance to work with many wonderful people, without whom this whole work would have not been possible.

9 References

[1] Sun Microsystems, “ZFS On-Disk Specification,” 2006-08. [Online]. Available: http://www.giis.co.in/Zfs_ondiskformat.pdf. [Accessed 18 April 2018].

[2] B. Ford and E. S. Boleyn, “Multiboot Specification version 0.6.96,” Free Software Foundation, Inc., [Online]. Available: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html. [Accessed 19 April 2018].

[3] B. Ford and E. S. Boleyn, “Multiboot2 Specification version 2.0,” Free Software Foundation, Inc., [Online]. Available: https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html. [Accessed 18 April 2018].

[4] “UEFI Specification Version 2.7,” Unified Extensible Firmware Interface Forum, [Online]. Available: http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf. [Accessed 27 04 2018].

[5] “ACPI Specification Version 6.2,” Unified Extensible Firmware Interface Forum Member Pages , [Online]. Available: http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf. [Accessed 27 04 2018].

[6] Distributed Management Task Force, Inc., “System Management BIOS (SMBIOS) Reference Specification 3.1.0,” 2016-11-16.

[7] VESA, “VESA ENHANCED EXTENDED DISPLAY IDENTIFICATION DATA STANDARD,” 9 February 2000. [Online]. Available: http://read.pudn.com/downloads110/ebook/456020/E-EDID%20Standard.pdf. [Accessed 15 05 2018].

[8] A. Zingl, “A Rasterizing Algorithm for Drawing Curves,” 2012. [Online]. Available: http://members.chello.at/%7Eeasyfilter/Bresenham.pdf. [Accessed 16 05 2018].

Appendix

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Toomas Soome**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Porting and Developing a Boot Loader,

(title of thesis)

supervised by Meelis Roos,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **20.05.2018**