Porting the Unix Kernel

Christopher K. Hettrick

University of Victoria
Department of Computer Science
CSC490
Supervised by Dr. Bill Bird

ABSTRACT

This report describes the process of porting a variant of the Unix kernel from the MIPS architecture to the Arm architecture. A heavily modified 2.11BSD version of the Unix kernel called RetroBSD is used as a case study, and is the basis of this development. The goal of this project is to run this ported kernel on both a simulator and on a physical embedded development board. An additional portion of this work is devoted to adapting the large-scale codebase of RetroBSD to more modern and sustainable development standards that will facilitate future ports to other platforms and architectures.

15 December 2020

Table of Contents

Porting the Unix Kernel

Christopher K. Hettrick

University of Victoria
Department of Computer Science
CSC490
Supervised by Dr. Bill Bird

1. Introduction

Porting the MIPS32® M4K® architecture to the Arm® Cortex®-M4 architecture.

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

2. Relevant History of BSD

RetroBSD is a semi-modernized version of 2.11BSD targeted to the PIC32MX7 MIPS-based microcontroller.[1] The early history of RetroBSD has been lost. It can only be concluded that RetroBSD was started some time in 2011, or perhaps some time even before that. The earliest post on the RetroBSD forum was from August 15, 2011. The project could have started much earlier than the creation of the forum. The project was started and lead by Serge Vakulenko, a systems programmer who started working at MIPS Technologies in 2011.[2]

2BSD is a family of operating systems for the DEC™ PDP-11 derived from Research UNIX and developed at the University of California at Berkeley. 2.11BSD has a long lineage going back to the first release of 2BSD on May 10, 1979.[3] 2BSD is a direct descendant of the Sixth Edition of Research UNIX, commonly known as V6 UNIX. 2.8BSD incorporated features from the Seventh Editition of Research UNIX, 32V UNIX, and 4.1BSD. The 2BSD line of software distributions continued on until the most recent release of 2.11BSD in 1991.[4] This release was a celebration of the 20th anniversary of the PDP-11. It is the culmination of the many efforts to port features from 4.3BSD and 4.3BSD-Tahoe — which run on the DEC VAX — to the PDP-11. Patches to 2.11BSD have been sporadically available since the initial release in 1991 from the long-time maintainer Steven Schultz. The most recent patch level is 469 and was released on April 14, 2020.[5] RetroBSD was started from patch level 431, which was released on April 21, 2000. It is from this version that all RetroBSD development began.

DiscoBSD derives from the most recent commit to the RetroBSD codebase, which is revision 506 from February 17, 2019.[6]

3. Hardware

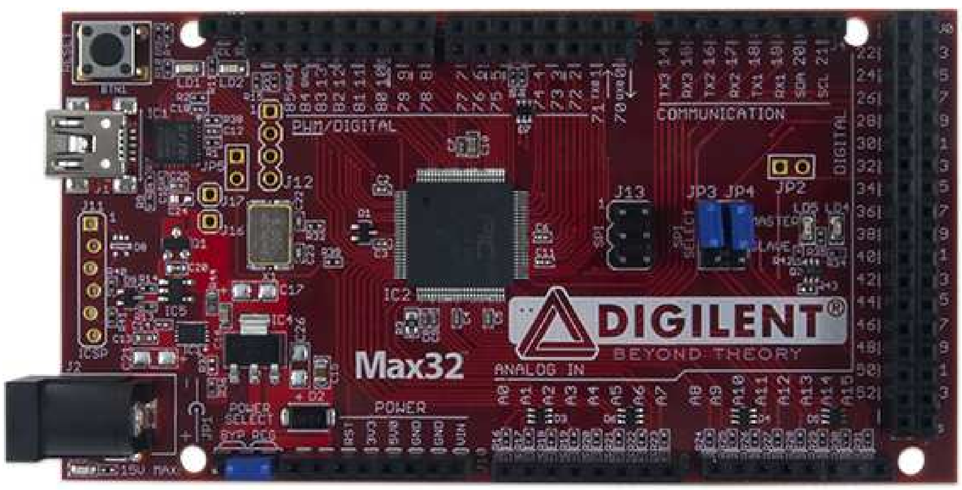
The defining features of the target hardware for RetroBSD and DiscoBSD are that they are RAM-constrained, have 32-bit processors, and do not have a memory management unit (MMU). The lack of an MMU rules out any possibility of virtual memory, which is a critical component in most major operating systems. A secondary feature of the target hardware is that their processors have the ability to protect kernel code from user code with a memory protection unit. This feature was not explored in this project, but is a viable focus of additional study.

3.1. PIC32 Development Board

The default development board that was used for the design and development of RetroBSD is the Max32 board, produced by Diligent. It employs a PIC32MX795F512 32-bit MIPS-based microcontroller. The processor runs at 80 MHz, has 512KB of flash program memory, and 128KB of SRAM data memory. It is powered either through the external power connector or the onboard USB connector. A UART terminal is achieved through the USB connection. This board does not have an onboard SD card, but one can be made available through an external Arduino-compatible shield. The SD card connects to the SPI2 port of the microcontroller. 83 general purpose I/O (GPIO) pins and some onboard LEDs are available. The board offers many more peripherals than outlined here, but they have no relevance to this project.

Programming and debugging of the board is achieved through the use of the PICkit3 in-system programmer/debugger and the connected USB cable. The MPLAB IDE software is used on a Windows system to load the firmware into the flash memory in the microcontroller.

An image of the Max32 development board is shown below:



3.2. STM32 Development Board

The default development board that is used for the design and development of DiscoBSD is the STM32F4Discovery board, produced by STMicroelectronics. A fully compatible revised edition of the board has been released under the model name STM32F407G-DISC1.

XXX Words

XXX Words

XXX Words

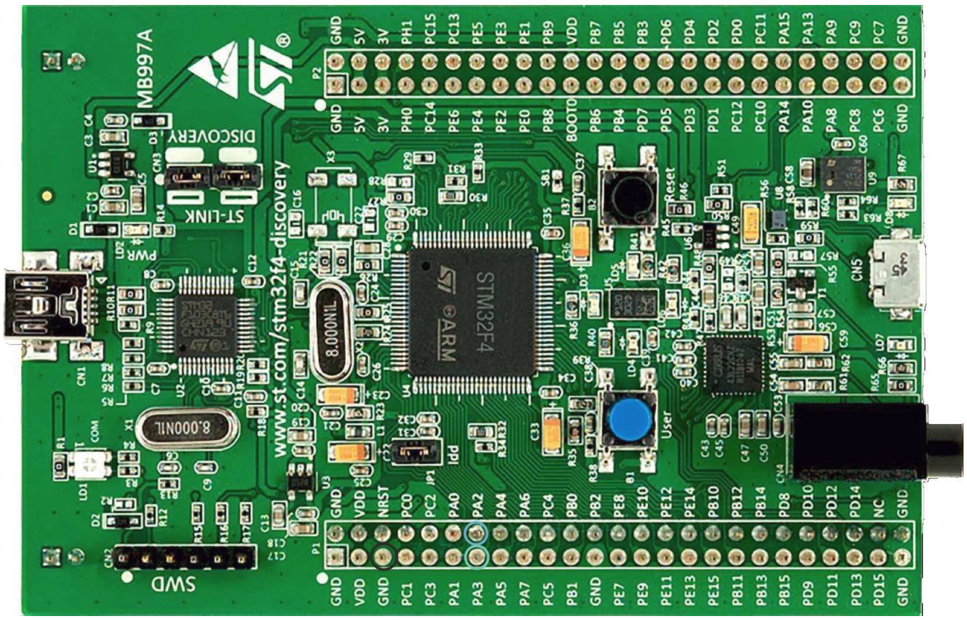
XXX Words

XXX Words

XXX Words

XXX Words

An image of the STM32F4Discovery development board is shown below:



4. Simulators and Emulators

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

4.1. PIC32 VirtualMIPS Simulator

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

XXX Words

An image of RetroBSD booting in the VirtualMIPS simulator is shown below:

```
$ cd discobsd/tools/virtualmips/
$ ./pic32

VirtualMIPS (version 0.06-retrobsd)
Copyright (c) 2008 yajin, 2011-2015 vak.
Build date: Nov  1 2020 14:07:14

Using configure file: pic32_max32.conf
ram_size: 128k bytes
boot_method: Binary
flash_type: NOR FLASH
flash_size: 492k bytes
flash_file_name: ../../sys/pic32/max32/unix.bin
flash_phy_address: 0x1d000000
boot_from: NOR FLASH
sdcard_port: SPI2
sdcard0_size: 340M bytes
sdcard0_file_name: ../../sdcard.img
start_address: 0x9d001000
uart1_type = console
--- Start simulation: PC=0x9d001000, JIT disabled

2.11 BSD Unix for PIC32, revision G19 build 1:
     Compiled 2020-11-01 by chris@trp.my.domain:
     /home/chris/compsci/github/CSC490/discobsd/sys/pic32/max32
cpu: 795F512L 80 MHz, bus 80 MHz
oscillator: HS crystal, PLL div 1:2 mult x20
spi2: pins sdi=RG7/sdo=RG8/sck=RG6
uart1: pins rx=RF2/tx=RF8, interrupts 26/27/28, console
uart2: pins rx=RF4/tx=RF5, interrupts 40/41/42
uart4: pins rx=RD14/tx=RD15, interrupts 67/68/69
sd0: port SPI2, pin cs=RC14
gpio0: portA, pins ii---ii-iiiioiii
gpio1: portB, pins iiiiiiiiiiiiiiii
gpio2: portC, pins i-ii-------iiii-
gpio3: portD, pins --iiiiiiiiiiiiii
gpio4: portE, pins ------iiiiiiiiii
gpio5: portF, pins --ii--------i-ii
gpio6: portG, pins iiii--i-----iiii
adc: 15 channels
pwm: 5 channels
sd0: type I, size 348160 kbytes, speed 10 Mbit/sec
sd0a: partition type b7, sector 2, size 102400 kbytes
sd0b: partition type b8, sector 204802, size 2048 kbytes
sd0c: partition type b7, sector 208898, size 102400 kbytes
phys mem  = 128 kbytes
user mem  = 96 kbytes
root dev  = (0,1)
swap dev  = (0,2)
root size = 102400 kbytes
swap size = 2048 kbytes
/dev/sd0a: 1446 files, 11972 used, 90027 free
Starting daemons: update cron


2.11 BSD UNIX (pic32) (console)

login: root
Password:
Welcome to RetroBSD!
erase ^?, kill ^U, intr ^C
# shutdown -h now
Shutdown at 14:15 (in 0 minutes) [pid 16]


        *** FINAL System shutdown message from root@pic32 ***

System going down IMMEDIATELY


System shutdown time has arrived
# killing processes... done
syncing disks... done
halted
9d019014: wait instruction with interrupts disabled - stop the simulator.

--- Stop simulation
$
```

4.2. QEMU-based Arm Cortex-M Emulator

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

5. Host Development Environment

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

5.1. Development Tools on OpenBSD

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

5.2. Development Tools on Linux

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

6. Kernel Operation Overview

Coverage of the kernel operation will be limited to the relevant issues for this project. System startup, process creation, and process management will be covered in outline in this section. For example, signals, communication facilities, and the filesystem will not be covered, but are, nonetheless, important facilities of any kernel.

The kernel gets loaded into RAM by reset and bootstrap code in the system startup sequence, and then execution is passed to it. It sets up the *swapper* process (PID 0), which the kernel will eventually become. The kernel then hand-crafts the first new process (PID 1) which will be the *init* process. The *init* process is the ancestor, and parent process, of all future processes in the system. Once *init* is created by a kernel-specific form of `fork()`, then the kernel becomes the *swapper* and manages scheduling processes.

In a roundabout and convoluted way, the *init* process loads the program `/sbin/init` from the filesystem and it is set executing. The *swapper* process eventually schedules the *init* process and runs it, which runs the `/sbin/init` executable. `/sbin/init` spawns a shell to interpret the commands in `/etc/rc`, then forks a copy of itself to invoke `/libexec/getty`, which further invokes `/bin/login` to log a user on. Upon a successful login, `/bin/login` uses a call to `exec()` to overlay itself with the user's shell. The system is now in the position that general *user mode* programs can now be run by users through their shell, and they will be scheduled and executed by the kernel *swapper* process.

The kernel uses a full swap policy wherein there can only be one process running in RAM at a time, in addition to the always-present kernel *swapper* process. The processes not currently running will be swapped out to the *swap area* on the disk, which in this case is a filesystem partition on the mounted SD card. The reasoning for this policy is that the available RAM to the system is not large enough to support multiple in-core processes. This is a defining, and unavoidable, constraint of DiscoBSD.

7. System Startup

After a system hardware reset, the kernel gets loaded into RAM from Flash by initial reset code and execution begins at the kernel's entry point, which eventually arrives at the kernel's `main()` function. Machine dependent (MD) peripherals are set up and initialized. The kernel's various data structures and services are initialized. The filesystem is mounted and set up. The *init* process is created and forked. The kernel process becomes the *swapper* to schedule all system processes. The code for `/sbin/init` is loaded from the filesystem into user memory and the *init* process "returns" to location zero of the code in user memory to execute it. The specifics of how all this happens is covered in the following subsections.

7.1. Bootstrapping and Linker Script

The default bootloader in STM32F4xx microcontrollers is set by the `BOOT0` (held low by default) and `BOOT1` (held high by default) pins. This selects the main Flash memory as the boot space, starting at address `0x00000000`.

There are two linker scripts that concern this operating system: one for the kernel and one for user executables. The former will be discussed in this section.

A linker script is a specifically formatted file that instructs the linker — as the last step of the compilation process — on how to lay out the various sections of the executable. This amounts to placing kernel code in the read-only *.text* section, initialized data in the read and write *.data* section, and specifying where the *.bss* section is located for uninitialized data and variables. The stack pointer is also placed accordingly, normally at the end of RAM for the full-decending stack on the Arm Cortex-M4. The stack pointer is defined by the label *_estack* and it is located at the end of RAM at address `0x20020000`.

A trimmed down version of the kernel's linker script is as follows:

```
MEMORY {
    FLASH (r x)  : ORIGIN = 0x08000000, LENGTH = 1024K
    RAM   (rwx)  : ORIGIN = 0x20000188, LENGTH = 32K - 0x188
    U0AREA(rw!x) : ORIGIN = 0x20008000, LENGTH = 3K
    UAREA (rw!x) : ORIGIN = 0x20008C00, LENGTH = 3K
}

/* Higher addresses of the user mode stacks. */
u0    = ORIGIN(U0AREA);
u     = ORIGIN(UAREA);
u_end = ORIGIN(UAREA) + LENGTH(UAREA);

_estack = 0x20020000;

ENTRY(Reset_Handler)

SECTIONS {
    .text : {
        KEEP(*(.isr_vector))
        *(.text*)
        *(.rodata*)
    } > FLASH
    _etext = .;

    .data : AT (_etext) {
        _sdata = .;
        *(.data*)
        . = ALIGN(8);
        _edata = .;
    } > RAM

    .bss : {
        . = ALIGN(8);
        _sbss = .;
        *(.bss*)
        *(COMMON)
        . = ALIGN(8);
        _ebss = .;
    } > RAM
}
```

All execution starts at `ENTRY(`*label*`)` where *label* is *Reset_Handler* on DiscoBSD (historically *start*). In Arm Cortex-M4, the first 32 bits (first word) of the executable is actually the address of the stack pointer, and the second word is the address of *label*. This is handled by the linker. *label* refers to a label in the architecture-specific assembly language startup code. This code will be covered in the next section.

7.2. Assembly Language Startup

The assembly language startup code differs greatly between MIPS and Arm. The MIPS startup code is entirely contained in the file `/sys/pic32/startup.S`, whereas Arm and STM has standardized on an elaborate set of files that are common amongst each family of microcontrollers. These standardized files are available from STMicroelectronics, the microcontroller vendor for STM32F407xx devices.

The following files are required by Arm for CMSIS functions:

- `cmsis_gcc.h`
- `core_cm4.h`
- `core_cmFunc.h`
- `core_cmInstr.h`
- `core_cmSimd.h`

The following files are required by STM for processor and SysTick initialization:

- `startup_stm32f407xx.s`
- `stm32_assert.h`
- `stm32f407xx.h`
- `stm32f4xx.h`
- `stm32f4xx_it.c`
- `stm32f4xx_it.h`
- `system_stm32f4xx.c`
- `system_stm32f4xx.h`

The Arm file that contains the label *Reset_Handler* is `/sys/stm32/startup_stm32f407.s` and is the file that starts all execution. This file is specific to STM32F407xx microcontrollers. Other microcontrollers in the STM32F4xx family have similar startup files, named in a comparable way.

The structure of the code in `startup_stm32f407xx.s` is as follows (shortened for brevity):

```
.global Reset_Handler

Reset_Handler:
    ldr sp, =_estack     /* Set stack pointer. */

    /* Code to copy .data segment from flash to SRAM. */

    /* Code to fill .bss segment with zeros. */

    bl SystemInit        /* Init system clock. */

    bl main              /* Call main() in kernel. */

    /* Once main() returns here as PID 1: */
    /*   enter user mode, */
    /*   run icode at address zero (to exec /sbin/init). */
    /* This is described in Section 7.4. */
```

Exception handlers and interrupt service routines are defined and handled in `stm32f4xx_it.c`. The Arm-required `SystemInit()` function, which is called from the startup assembly code shown above, is defined in `system_stm32f4xx.c`. The various header files have defines for the standard Arm environment. Once the startup assembly code calls the `main()` routine, the kernel proper is running C code and will start the kernel initialization process.

7.3. Kernel Initialization

Kernel initialization is completely contained in the file `init_main.c`, which is where the `main()` routine is located. The kernel starts in *kernel mode*.

The `startup()` routine initializes machine dependent (MD) peripherals. `startup()` is defined in `/sys/stm32/machdep.c` and is highly specific to the processor architecture and the available peripherals on the target board. For example, this is where LEDs and GPIO pins are initialized.

Kernel autoconfiguration is performed with a call to `kconfig()`, which probes for all the devices available to the system at boot time. This is a dynamic process, and as such, allows flexibility in the presence of optional devices. The absense of any required standard device will cause the kernel to panic. Kernel configuration is explained in more detail in Section 8.

The system process structure (*struct proc*) for PID 0 is set up. Each process in the system has an entry in the process table in the kernel. The process table is implemented as an array of *struct proc* entries. The process structure must always remain in main memory, no matter what state the process is currently in.

The init user structure (*struct user*) is set up. The user structure is quite unique. There are two instances of the user structure: *u0* and *u*, which are declared in the linker script. *u0* is dedicated to PID 0, the *swapper* process. *u* is the user structure of the in-core active process. The user structure of any process not currently in a *runnable* state is swapped out.

Next, signals are initialized. The kernel's various data structures, tables, and protocols are initialized. Well-known inodes are set up. The kernel clock is set up. Services are attached to the kernel. Detailed coverage of these topics is beyond the scope of this report.

The root filesystem is mounted. If no root filesystem is found, the kernel will panic. The swap file on the root filesystem is opened and cleared. If no swap file is found, the kernel will also panic. Timeout driven kernel events are started. Finally, the root filesystem is set up.

The next section will continue the kernel initialization with the final task of setting up a working kernel: getting `/sbin/init` to run.

7.4. Getting to /sbin/init

Continuing on in the `main()` routine, and following the set up of the root filesystem, the *init* process is created by the kernel-specific version of `fork()` called `newproc()`. The kernel process (as the parent process) officially becomes the *swapper* to schedule all system processes by calling the `sched()` routine, which never returns. The child process of the fork is the *init* process. In the *init* process, the code for a small assembly language routine called `icode` is copied from the kernel image to the start of user memory.

The routine is effectively the same as the following program:

```
main()
{
    char *argv[2];

    argv[0] = "init";
    argv[1] = 0;
    exit(execv("/sbin/init", argv));
}
```

The last task in the `main()` routine is for the *init* process to "return" to location zero of the code in user memory and execute it. In effect, the return is from the branch to `main()` in the startup assembly code, and is a *thunk* to run the `icode` just copied out. This process has been been, rightly so, described as "somewhat enigmatic" by John Lions in his famous Commentary on UNIX 6th Edition. The call to `execv()` replaces the image of the *init* process with the userland image of `/sbin/init`, which is loaded from the mounted root filesystem. It is especially important to understand that `/sbin/init` is running in *user mode*, not in *kernel mode*, as a regular user process.

7.5. Getting to the User's Shell

As shown in the previous section, the *init* process starts up the `/sbin/init` userland program, and exits if the call to `execv()` fails. This makes the presence of `/sbin/init` vital to the system bootstrapping procedure.

`/sbin/init` forks itself and spawns a shell to interpret the commands in `/etc/rc`, which performs various tasks such as filesystem consistency checks, and starting up daemon processes like `/sbin/cron` and `/etc/update`. `/sbin/init` then forks a copy of itself for each terminal device that is marked for use in the file `/etc/ttys`. Each copy of `/sbin/init` invokes `/libexec/getty` to manage signing on to the system. `/libexec/getty` eventually reads in a user's login name from its terminal and invokes `/bin/login` to complete the login sequence. Once the user password check is complete, `/bin/login` uses an `exec()` call to overlay itself with the user's shell (normally `/bin/sh`, the standard Bourne shell).

The system is now, finally, in a state to be commanded by users in the usual way.

8. Kernel Configuration

The kernel configuration program `/tools/kconfig/kconfig` is used to configure a kernel, based on the `Config` file in the build directory, namely `/sys/stm32/f4discovery/Config`. The support files `Makefile.kconf`, `devices.kconf`, and `files.kconf` in the `/sys/stm32` directory are used in the configuration process. Cursory coverage of `kconfig` will be outlined below, while detailed information is available from the `kconfig` documentation.

The purpose of `kconfig` is to generate a `Makefile`, which is used to compile a specific kernel. `Makefile.kconf` is a template `Makefile` that has default build rules and directives, as well as anchors to attach generated build rules. The specific source files used to build the kernel are retrieved from the file `files.kconf` by matching both standard kernel files and optional device drivers. `devices.kconf` contains a list of block devices and their major numbers for the filesystem.

A basic kernel configuration is possible with the following `Config` configuration file:

```
architecture     "stm32"                  # Processor architecture
cpu              "STM32F407xx"            # Processor variant
board            "F4DISCOVERY"            # Board type
ldscript         "f4discovery/STM32F407XG.ld" # Linker script

options          "CPU_KHZ=80000"          # CPU core osc freq
options          "BUS_KHZ=80000"          # Peripheral bus freq
options          "BUS_DIV=1"              # Bus clock divisor

config           unix     root on sd0a    # Root filesystem
                          swap on sd0b    # Swap partition

device           uart1                    # Serial UART port 1
options          "CONS_MAJOR=UART_MAJOR"  # UART1 as console
options          "CONS_MINOR=0"           # /dev/tty0

controller       spi2                     # SD card

device           sd0      at spi2 pic RC14 # SD card select pin
options          "SD_MHZ=10"              # SD card speed 10 MHz
```

Note that the full functionality of STM32-specific configuration has not yet been added to `kconfig`. A fully working `Makefile` that is able to compile the DiscoBSD kernel, using the above configuration defines, has been created by hand.

9. Userland

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

10. Build System

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

10.1. Multi-Architecture Features

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

11. Project Difficulties

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

12. Future Work

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

13. Conclusion

XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words
XXX Words

References

1. Serge Vakulenko, *Homepage of RetroBSD*, http://retrobsd.org (Last updated August 28, 2015). Accessed November 28, 2020.
2. Serge Vakulenko, *Homepage of Serge Vakulenko*, http://vak.ru (Last updated April 12, 2017). Accessed November 29, 2020.
3. The PDP Unix Preservation Society, Warren Toomey, (ed), *Details of the PUPS Archive*, https://minnie.tuhs.org/PUPS/archive_details.html (Last updated February 21, 1996). Accessed November 28, 2020.
4. Steven M. Schultz, *Announcement of Second Distribution of Berkeley PDP-11 Software for UNIX Release 2.11*, https://www.krsaborio.net/bsd/research/1991/0314.htm (January 1991). Accessed November 28, 2020.
5. Steven M. Schultz, *Patch level 469 for 2.11BSD Distribution*, https://www.tuhs.org/Archive/Distributions/UCB/2.11BSD/Patches/469 (April 14, 2020). Accessed November 28, 2020.
6. RetroBSD, *RetroBSD Autobuilder Revision 506*, http://retrobsd.org/wiki/autobuild.php?rev=506 (February 17, 2019). Accessed October 31, 2020.