



Bridgewater State University Virtual Commons - Bridgewater State University

Honors Program Theses and Projects

Undergraduate Honors Program

5-8-2018

The Restoration of UNIX: Emulating UNIX version 1.0 on a 16-bit DEC PDP 11/20

John J. Gilmore Jr.

Follow this and additional works at: http://vc.bridgew.edu/honors_proj

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Gilmore, John J. Jr.. (2018). The Restoration of UNIX: Emulating UNIX version 1.0 on a 16-bit DEC PDP 11/20. In *BSU Honors Program Theses and Projects*. Item 285. Available at: http://vc.bridgew.edu/honors_proj/285
Copyright © 2018 John J. Gilmore Jr.

This item is available as part of Virtual Commons, the open-access institutional repository of Bridgewater State University, Bridgewater, Massachusetts.

The Restoration of UNIX: Emulating UNIX version 1.0 on a 16-bit DEC PDP 11/20

John J. Gilmore Jr.

Submitted in Partial Completion of the
Requirements for Commonwealth Honors in Computer Science

Bridgewater State University

May 8, 2018

Dr. Michael Black, Thesis Advisor

Dr. John Santore, Committee Member

Dr. Haleh Khojasteh, Committee Member



The Restoration of UNIX: Emulating UNIX version 1.0 on a 16-bit DEC PDP 11/20

Author: **John Gilmore**

Mentor: **Professor Michael Black**

A thesis submitted in the partial pursuit of Departmental Honors, and the fulfillment of a Bachelor's of science in Computer Science.

Bridgewater State University

Bartlett College of Science and Mathematics Department of Computer Science

April 2018

Abstract

Next year, the one of the world's first Operating Systems (OS) UNIX will celebrate its 50th birthday. This relic of the past is objectively one of the most important creations that has ever influenced the field of Computer Science. However, mysteries of this artifact were discovered in 1995. Two engineers, Paul Vixie and Keith Bostic, dug deep enough to find several DEC tapes "under the floor of the computer room [at Bell Labs]" (UNIX Past) which held several original binary files for UNIX. With much help from several individuals, these two engineers were able to reverse engineer the binary tapes they found and 15 years later, put up a repository for this ancient system on GitHub. There it has stayed, waiting for someone to use it like it was meant to be used, on a DEC PDP 11/20, the computer UNIX v1 was initially written for. Currently, the PDP 11/20 can send and receive data via a homemade serial emulator we have written in Java, and the PDP 11/20 has been loaded with a binary file for the BASIC programming language. Once the entry point for BASIC can be determined, we will start to write code to emulate paper tape readers to assist in loading UNIX v1 into the PDP 11/20's memory. In the end, we hope to have a fully functional system running the one of the world's first Operating Systems.

Acknowledgments

I need to thank my mentor Professor Michael Black for helping guide me through this project. He helped answer questions that I had while being more of a guide than a teacher. He engaged me in conversations about where we should be going rather than just giving me a straight answer because this is new territory for us both. I would bring him my ideas on how to continue, and he would guide me in the correct direction. I also need to thank my peer and associate Francis Duffy. He and I worked extensively on many of the problems detailed in this work. Our discussions often led to the solutions necessary to move forward. Undoubtedly, without them, I could not have made as much progress as I did.

Table of Contents

- 1. Part 1: Introduction, page 5**
 - a. Section 1.1: The PDP 11/20, page 5**
 - b. Section 1.2: UNIX, page 6**
 - c. Section 1.3: Methodology , page 9**
- 2. Part 2: The PDP 11/20, page 12**
 - a. Section 2.1: The Architecture, page 12**
 - b. Section 2.2: The Instruction Set, page 16**
- 3. Part 3: Methodology, page 20**
 - a. Section 3.1: Simh, page 20**
 - b. Section 3.2: Communication, page 21**
 - c. Section 3.3: The Bootstrap Loader, page 26**
- 4. Part 4: Conclusion, page 30**
 - a. Section 4.1: Further Work, page 30**
- 5. Appendix A: Assembly Programs, page 32**
- 6. Appendix B: Serial Emulator, page 34**
- 7. Appendix C: Instruction Set, page 41**
- 8. Bibliography, page 48**

Part 1: Introduction

Section 1.1: The PDP 11/20

The DEC PDP 11/20 is a 16-bit minicomputer, the word “mini” being a bit of a misnomer for the modern day, as the PDP 11/20 accompanied with all its peripherals could take up an entire room. The PDP 11/20 is an updated version of some of DEC’s earlier models of PDP computers, and is the first 16-bit machine created by DEC. The PDP 11/20 had above average performance for its time, but it is undoubtedly starting to show its age. Several primary systems that we have tried to use have been malfunctioning, and it is indeed on its last legs.

In the dawn of the 1970’s, DEC was already a well-defined company after its launch of the 12-bit PDP 8 (“DEC”). During this time, computers were still in their infancy, and something as simple as a 12-bit computer (compared to today’s 64-bit computers) was a major technological breakthrough, but this was surely not the final accomplishment in the progress of computer architecture. In 1970 the DEC PDP 11/20 was released, and being one of the first 16-bit computers of its time, the PDP 11/20 quickly became the most successful DEC product of all time (“DEC”).



Figure 1: An image of the original PDP 11/20 that was sold in 1970. The main interface beign a series of LED's and switched shown on the middle right. Above that are the paper tape I/O devices.

The 11/20 CPU model was designed with eight 16-bit data “containers” called registers. The first six being general purpose registers that one could use for data manipulation and storage, the sixth register being the stack pointer, and register seven being the program counter. The stand-alone memory size was only 64 kilobytes; which is a stunningly small number compared to the size of memory today. The astoundingly low memory cache caused several issues throughout this research project. A single program you download from

the internet in the modern day is at least a few megabytes, hundreds of times bigger than the entire

memory on the PDP 11/20. The PDP 11/20 was constructed with over 70 unique instructions ranging from something as simple as an add operation, to as complex as subroutine calls.

When the PDP 11/20 was initially sold, it came along with several necessary peripheral devices. These included a Teletype I/O which could be used for a more natural interaction with the user and paper tape readers which could be used to load complex and lengthy programs into main memory as shown in **Figure 1**. In these paper tapes are where an assembler for UNIX may be loaded and read into

the PDP 11/20 as a series of 16-bit words. Our setup is bare bones in comparison. We have the computer and its interface without any additional peripherals (the PDP 11/20 owned by my mentor Michael Black shown in **Figure 2**). Nowadays it is a



Figure 2: The PDP 1120 interface that was used in this project. It does not include any peripherals and is strictly the main interface.

miracle we even had a PDP 11/20 to use for our research as it would be nearly impossible to find paper tape readers that would be still intact, and in good enough condition to use to load UNIX.

Section 1.2: UNIX

UNIX was the first operating system that made computing feasible for those who did not have an in-depth understanding of how computers worked. Without UNIX, the world today may have been vastly different. UNIX is the spiritual ancestor of all operating systems including the Windows, and MAC Operating systems. The Linux OS was created from a directed transition from UNIX v7. Without UNIX as a catalyst, computers may still only be something used behind the scenes instead of having a mass appeal.

UNIX version 1 was created at AT&T's Bell Laboratories sometime in early 1969 by Ken Thompson, and Dennis Ritchie (UNIX Past), but UNIX version 3, built in 1973, was the first version of the

OS to reach the public. It was received with an unexpectedly high positive response, and researchers, engineers, government agencies, undergraduate institutions, and many others started to use UNIX to improve the performance of their work. Soon enough, the open source OS was being manipulated into hundreds of different versions to improve performance and increase the interaction with the user. "In the early 1980's, the market for UNIX systems had grown enough to be noticed by industry analysts and researchers. Now the question was no longer 'What is the UNIX system?', but instead, 'Is a UNIX system suitable for business and commerce?'" (UNIX Past). The OS started to be used by people who knew less and less about the inner working of the computer, as the quality of the user interface got higher and higher until the 7th and final version of the system was released in 1979. At this point, Bell Labs ceased to create newer versions of UNIX, and it eventually evolved into the operating systems we have come to know today. However, something seems to be missing. Once UNIX version four was released, its mass appeal skyrocketed. In the excitement of the era, fewer and fewer people knew about where this powerful system originated; the first and second versions of UNIX never saw the mass market, and all that is left of these precious artifacts are seemingly just rumors. Not even the creators could recreate it from the ground up. The single assembler that was created for the OS seemed to have disappeared without any trace, and the humble beginnings of this revolutionary technology were utterly lost.

In 1995, the UNIX heritage society (TUHS) was founded for the purpose to try and preserve these late editions of UNIX, and several other ancient, and valuable artifacts. It quickly gained access to many newer versions of old systems such as the binaries for UNIX v5, manuals for UNIX v4, the entire repository for UNIX v7, and several former computer systems. Eventually, the organization got into contact with Dennis Richie to shine some light on the affairs of ancient UNIX systems. Unfortunately, not even the creator of UNIX could determine where the tapes for UNIX v1 had ended up (Toomey). After a significant amount of lobbying by TUHS, they gained access to old Bell Labs research facilities where miraculously two engineers Paul Vixie and Keith Bostic found several DEC tapes "under the floorboards

in the computer room.” These tapes included many exciting finds, but the most interesting involved that of binary files for UNIX v1 itself (Toomey).

While this was undoubtedly an excellent milestone for TUHS, it takes a lot of reverse engineering to understand just what conditions these tapes needed to function. Unlike with Windows, you cannot just slip a CD into a CD reader and expect UNIX to pop up on a screen for the PDP 11/20 or even a modern computer. Certain pre-environments must be constructed, and it may need a various number of other peripherals to run. After these tapes were found, they were handed off to Warren Toomey of Bond University in Australia. After several years of arduous work which he details in his paper titled “The Restoration of Early UNIX Artifacts,” was able to reverse engineer the binary files, and restore UNIX version 1 to most of its former glory, but it is still mildly flawed (Toomey).

The most prominent issue that he had faced in restoring UNIX v1 was that the assembler that was provided on the tapes for UNIX v1 was written in an ancient dialect of the C language. This version of C was so incompatible with the modern C language that the kernel for UNIX which was found on the tapes could not run on the systems that he had at his disposal. In the end, to mitigate this issue, he was forced to use the UNIX v2 shell to communicate with the UNIX v1 kernel (Toomey). Therefore, the repository of UNIX v1 that can be found on GitHub is at its core the original version of UNIX, but alas, has some makeup of v2. This is where their research ends, and where ours begins. No one has ever tried to fully restore UNIX to running as it once did on a DEC PDP 11/20. It was once known that an assembler for the PDP 11/20 did exist in some way, but history was not as kind to it as it was to the tapes of the ancient UNIX artifacts found under Bell Labs. It is the goal of this research project to see UNIX come to fruition, and get UNIX to the state it once started in.

Section 1.3: Methodology

Throughout the past several months we have faced several problems and situations which strained our logic and reasoning abilities, which we failed to anticipate. Before starting work with the PDP 11/20, we had to get a reasonable basis for how the instruction set, and the process of how to program the PDP 11/20. We did some research into the old computer simulator Simh. Simh (as shown in **Figure 3**) (Simh) is a project dedicated to creating freeware simulators for hundreds of ancient, early, and modern computer systems. We

```
C:\Users\gilmo\Dropbox\pdp 11-20 Research Project\Simh\pdp11.exe
sim> dep 037706 000000
sim> dep 037710 012701
sim> dep 037712 177562
sim> dep 037714 012702
sim> dep 037716 000000
sim> dep 037720 012703
sim> dep 037722 000000
sim> dep 037724 012704
sim> dep 037726 177560
sim> dep 037730 160505
sim> dep 037732 105714
sim> dep 037734 100376
sim> dep 037736 111105
sim> dep 037740 162705
sim> dep 037742 000104
sim> dep 037744 142705
sim> dep 037746 177770
sim> dep 037750 006102
sim> dep 037752 006102
sim> dep 037754 006102
sim> dep 037756 006502
sim> dep 037760 005303
sim> dep 037762 020327
sim> dep 037764 000000
sim> dep 037766 10076
sim> dep 037766 100760
sim> dep 037770 010210
sim> dep 037772 062700
sim> dep 037774 000002
sim> dep 037776 000746
sim> run 037704
```

Figure 3: An example of running the Simh PDP 11/20 emulator. In the above instance, the bootstrap loader (which is discussed in detail in section 3.3) is loaded into memory addresses 037706-037770, and then executed.

were able to install a simulator for the PDP 11/20

and test several of its operations. Throughout the project, we would continue to use the Simh simulator for the PDP 11/20 to use the PDP 11/20 indirectly. Simh offers a more accessible, and more predictable interface, and allows for easier debugging of source code for assembly programs written for the PDP 11/20 (Simh).

With a good understanding of how to write source code for the PDP 11/20, we moved onto the unexpectedly arduous task of communicating with the PDP 11/20. After only a few weeks, we were able to determine how to communicate with the DL-11 serial card of the PDP 11/20 via

PDP-11/05 Signal	SCL connector	KLBE cable	40 pin flat cable	RS232 level converter	RS232 DSUB 9MATE	20mA N-LOCK
-15V	BB	U	17			
SERIAL OUT+ (20mA)	V	AA	23			5
CLK IN (TTL)	T	CC	25			
SERIAL IN- (20mA)	DD	S	15			3
READER RUN- (20mA)	R	EE	27			4
CLK DISAB (TTL)	N	HH	29			
SERIAL OUT- (20mA)	L	KK	31			2
+5V	C	TT	38	+5V		
SERIAL OUT H (TTL)	D	SS	37	TX	2 Tx/D	
READER RUN+ (20mA)	F	PP	35			6
SERIAL IN L (TTL) ^{1,2}	RR	E	05	RX	3 Rx/D	
20mA Interlock ¹	NN	H	07			
SERIAL IN+ (20mA)	LL	X	09			7
GND	A	VV	40	GND	5 GND	
GND	B	UU	39			
GND	UU	B	02			
GND	VV	A	01			

Figure 4: The diagram displaying the connections of the 40-pin berg cable. Even with a diagram like this for reference, the physical cable could be oriented the improper way to correlate to the above diagram; which could lead to a few issues if the wires for communication are attached improperly. Specifically, the lines we're interested in for serial communication are the A, D, and RR lines.

a 20-line Berg cable by deducing which lines were the Transmit (TX), Receive (RX), and Ground (GND)

lines for the Berg cable by using **Figure 4** (Interfacing). After another few weeks, we were able to write a program on the PDP 11/20 which would continuously send the character 'A' across the serial line (see Appendix A P.1 for the source code). This program allowed us to determine the validity of the TX line from the PDP to our computer.

Once the TX line was tested, it was time to check the RX line; which did not forfeit its secrets as easily. When trying to send data from our computer, and to the PDP 11/20 we noticed something strange happening. Whenever we would send a character, the character 'A' for instance, the PDP 11/20 would receive an entirely different, seemingly unrelated character. This issue persisted through several separate trials of test programs. No matter what we sent, how we sent it, or even how we tried to communicate with the PDP 11/20, for some reason every time the PDP 11/20 received something from us, some degeneration in what was presumably the DL-11 serial card would interfere with the data transfer.

We came to our first conclusion that perhaps it was the way that our laptop was transferring the data which was jumbling up the output. We needed some way to gain stricter controls over the data transfer, then the tools that the Arduino serial interface that we were using could supply to us. We decided that the most efficient solution would be to create a homemade serial emulator which we could program to use different settings, something that we could not do on the Arduino serial interface.

The research and development of our serial interfacing platform took priority over the next several weeks as we tried to determine the correct way to set up a peer to peer socket communication between our laptop, the serial port, and the PDP 11/20. Shortly, we were able to create a working serial monitor that was able to set up the communication (see Appendix B) (Poon). We tested the already known program on the PDP 11/20 which would continuously send the character 'A', and it seemed to work correctly still. However, once we started to put effort back into getting the RX line to work, no

matter how we manipulated the data being sent, it still seemed as if the PDP 11/20 was receiving random characters that did not correlate to the characters we were sending.

At this point, we were stumped on how to proceed. We had tried several solutions to the problem, which we had been hoping would fix the issue, but we seemed to be right back where we started. We next even went as far as taking out the DL-11 serial card to examine it. We looked at the schematics, determined what specific lines that crossed it were meant to do. A few things seemed out of place, but not even “fixing” the DL-11 solved the issue.

Finally, we noticed something strange. There seemed to be a peculiar pattern between the characters that we were sending. Looking at the ASCII table in **Figure 5**, one can get a better

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0 0	000	000	NULL	32 20	040	040	(Space	64 40	100	040	@	o
1 1	001	001	Start of Header	33 21	041	041)	!	65 41	101	041	A	A
2 2	002	002	Start of Text	34 22	042	042	*	"	66 42	102	042	B	B
3 3	003	003	End of Text	35 23	043	043	+	#	67 43	103	043	C	C
4 4	004	004	End of Transmission	36 24	044	044	,	\$	68 44	104	044	D	D
5 5	005	005	Enquiry	37 25	045	045	-	%	69 45	105	045	E	E
6 6	006	006	Acknowledgment	38 26	046	046	.	&	70 46	106	046	F	F
7 7	007	007	Bell	39 27	047	047	/	'	71 47	107	047	G	G
8 8	010	010	Backspace	40 28	050	040	(72 48	110	040	H	H
9 9	011	011	Horizontal Tab	41 29	051	041)		73 49	111	041	I	I
10 A	012	012	Line feed	42 2A	052	042	*	"	74 4A	112	042	J	J
11 B	013	013	Vertical Tab	43 2B	053	043	+	"	75 4B	113	043	K	K
12 C	014	014	Form feed	44 2C	054	044	,	"	76 4C	114	044	L	L
13 D	015	015	Carriage return	45 2D	055	045	-	"	77 4D	115	045	M	M
14 E	016	016	Shift Out	46 2E	056	046	.	"	78 4E	116	046	N	N
15 F	017	017	Shift In	47 2F	057	047	/	"	79 4F	117	047	O	O
16 10	020	020	Data Link Escape	48 30	060	048	0	0	80 50	120	048	P	P
17 11	021	021	Device Control 1	49 31	061	049	1	1	81 51	121	049	Q	Q
18 12	022	022	Device Control 2	50 32	062	050	2	2	82 52	122	050	R	R
19 13	023	023	Device Control 3	51 33	063	051	3	3	83 53	123	051	S	S
20 14	024	024	Device Control 4	52 34	064	052	4	4	84 54	124	052	T	T
21 15	025	025	Negative Ack.	53 35	065	053	5	5	85 55	125	053	U	U
22 16	026	026	Synchronous idle	54 36	066	054	6	6	86 56	126	054	V	V
23 17	027	027	End of Trans. Block	55 37	067	055	7	7	87 57	127	055	W	W
24 18	030	030	Cancel	56 38	070	056	8	8	88 58	130	056	X	X
25 19	031	031	End of Medium	57 39	071	057	9	9	89 59	131	057	Y	Y
26 1A	032	032	Substitute	58 3A	072	058	:	:	90 5A	132	058	Z	Z
27 1B	033	033	Escape	59 3B	073	059	;	:	91 5B	133	059	[[
28 1C	034	034	File Separator	60 3C	074	060	<	@	92 5C	134	060	\	\
29 1D	035	035	Group Separator	61 3D	075	061	=	@	93 5D	135	061]]
30 1E	036	036	Record Separator	62 3E	076	062	>	>	94 5E	136	062	^	^
31 1F	037	037	Unit Separator	63 3F	077	063	?	?	95 5F	137	063	_	_

Figure 5: The ASCII table of characters. The table extends further past 127 up to 255, but those characters include those rarely used, such as ì, ú, ä, æ, etc.

understanding of the pattern that was persisting.

Whenever we sent over the character ‘A’ (decimal value 65), the PDP 11/20 would receive the character ‘}’ (decimal value 125). If the character ‘B’ (decimal value 66) were sent then, the PDP 11/20 would receive the character ‘{’ (decimal value 123).

A ‘C’ (decimal value 67) would show as a ‘y’ (decimal

value 121). This pattern of a one to one relationship would continue for any important characters that we may want to send. This degeneration seemed to imply some functional relationship between the character which we sent, and the character that the PDP 11/20 would receive. This discovery was reinvigorating because it implied that the error occurring inside of the PDP 11/20 was not random, but rather systematic. Furthermore, if there was a defined functional relationship, then that means that we can reverse the function to send the actual characters we want to send. To mitigate the issue we were able to devise an equation that when we wanted to send a character to the PDP 11/20, the serial

emulator would be coded to change that character into the character that would produce the original character back on the PDP 11/20's side.

Having mitigated the most substantial issue we had run into so far, we turned towards our next step in our process. To load UNIX onto the PDP 11/20, there would need to be tape drives which the OS can poll for specific results throughout its operation. Unfortunately, we do not have any peripherals that we can use with the PDP 11/20, so any peripherals that UNIX may need to run would have to be simulated. Simulating such intricate, and old hardware would be near impossible to do with just the PDP 11/20 due to the complexity of programming it, and its small memory size. Therefore, we needed a higher-level programming language to code to emulate the peripherals that we needed. Somehow, we would need to load BASIC (one of the world's first high-level programming languages) into the PDP 11/20's core memory. We quickly began development on a loader written in the PDP 11/20 assembly language which would take a binary file from our laptop and store the entire file into concurrent memory addresses in the PDP 11/20. We were able to create just that (see Appendix A P.2), and by the end of our first semester, were able to load what we believed to be the binary file for BASIC onto the PDP 11/20.

Part 2: The PDP 11/20

Section 2.1: The Architecture

The central systems behind the inner architecture of a computer have not significantly changed since their inception in the mid 1900's. Computers have simply gotten smaller, and faster. Usually a computer is operated between the connections between the different components which make up the computers architecture. These components communicate through a single bus, also known to the PDP 11/20 as the UNIBUS.

Inside of a bus lies many wires which carry data from component to component (this sits below the components seen in **Figure 6**). The number of the wires inside of the UNIBUS is up to the specifications of the computer. A 16-bit machine like the PDP 11/20, for instance, has a single UNIBUS which has 16 individual wires which run along the main components of the system, carrying essential data from master and slave components. Any peripherals that one may want to connect to the PDP 11/20 will be in communication with the rest of the computer through the UNIBUS as well. This relationship allows for a well-organized system with top priority task handling. Based on which components wish to send data, they must first request control of the UNIBUS from the previous “master” of the UNIBUS. This master/slave relation (which can be seen in **Figure 7**) gives each device that can request use of the UNIBUS a specific priority. When two devices wish to access the



Figure 6: The internal structure of the PDP 11/20. Each interface card is attached to the UNIBUS below. Interestingly, the UNIBUS runs along the top part of the chassis, and the cards hang down from it. To examine the inside of PDP 11/20, it must first be flipped upside down.

UNIBUS at the same time, the device with the higher priority is given control of the system. This priority-

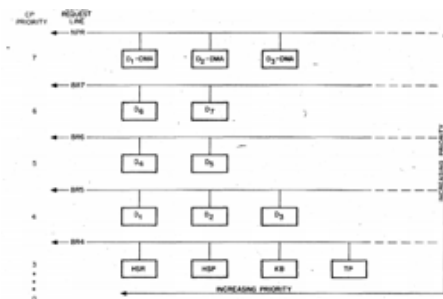


Figure 7: A depiction of the master/slave relationship which takes place between the UNIBUS on the PDP 11/20, core, and peripheral devices.

based over-ride system controls the flow of data and detracts from mis-interpolation of data (Olsen).

Every system in the PDP 11/20 is assigned a unique memory address for the user to be able to gain control of it via the UNIBUS and allows a wanted peripheral, register, device, memory location, etc. to send information. For instance, for the

user to poll the central processor, the reference address is 10008. If the user wishes to poll the Teletype Transfer line, the reference address 177566 is used. This is an extraordinarily powerful tool as it allows the user to write programs which can poll TX and RX lines, and manipulate data going out, or coming in (Olsen).

The processor is also given access to several registers which can be used to store, transfer, and manipulate data. The user works heavily with these registers to perform most of any tasks they may want to complete. These registers can almost be thought of variables in a higher-level language which allows a programmer to store data and control it by referring to that variable in certain situations. The PDP 11/20 comes with eight general purpose registers (r0, r1,...r7). The registers r6 and r7 are exceedingly unique. The register r6 is used can be used as a way to poll the process stack; which holds all processes that the PDP 11/20 must complete. In this way, a user could push a new instruction onto the stack and have that take priority over any other instruction that the PDP 11/20 was about to execute. A user also now could pop a process off the stack to stop a specific task from happening. These abilities can be potent when used in specific situations. Furthermore, the register r7 holds the value of the program counter (PC); which keeps track of the executed code in memory and interacts heavily with the central processor of the PDP 11/20. Whenever an instruction gets popped off the stack after successfully executing, the value stored in the PC changes (usually by increasing by instruction), and the central processor polls the PC to determine the next instruction to perform. Having a register with this specialty can be as powerful as the stack pointer. Manipulating r7 allows the user to change the sequential run of a program and provides the ability to create subroutines; which can be entirely separate programs which can be run inside of a more extensive program. Referencing the PC can also give the user the ability to manipulate data that is anywhere within plus or minus 256 memory addresses (Olsen).

The PDP 11/20 is also equipped with a central processor status register (shown in **Figure 8**) (Olsen). Bits 5-7 of this register store the current priority of the processor, and bits 0-5 store conditional codes that can be flipped on or off based on the outcome of the previous instruction run. In general, for most instructions, if the result of the last instruction resulted in a bit to be

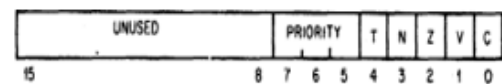


Figure 8: The CPU status word for the PDP 11/20, showing the displacement of bits within the memory address.

carried from an arithmetic operation, then the first bit of the status register will be set; this is given the condition designation 'C'. If the result of the instruction resulted in an arithmetic overflow (AKA if the result of an ADD instruction was greater than 32767 then this large of a number cannot be stored within 16 bits), then the second bit of the status register will be set; this is given the condition designation 'V'. If the result is zero, then the third bit of the status register will be set; this is given the condition designation 'Z'. The fourth bit is set if the result of the operation was negative; this is given the condition designation 'N'. Lastly, the fifth bit of the status register is purely for debugging purposes. It can be set by the user at any point in a program and will cause a processor trap instruction which will hang up the program (Olsen).

The PDP 11/20 which was used in this project contains over 32,000 memory addresses with instructions and memory address locations represented as octal digits. For instance, the octal address 36215 could be expressed as the decimal number 15501, or the binary number 11110010001101. Referencing these addresses can be accomplished through the primary interface of the PDP 11/20 by turning the red and black switches on the front panel of the PDP 11/20 (which can be seen back in **Figure 2**). Every address has the capability of storing a byte of data. For this reason, a single 16-bit instruction is stored within every two memory addresses, making it impossible to save instructions directly to odd-numbered addresses. This strange storage of instructions can often cause confusion because many times numbers in instructions will be divided or multiplied by two, the reason that of the word size of 16-bits. Furthermore, a significant amount of memory addresses is reserved for particular purposes. For example, the addresses between 000000 and 00370 are strictly reserved for interrupt vectors (which is important to note because we ran into an issue which described in greater detail in Part 3). The top 4096 addresses are designated to peripheral device management registers (Olsen).

Section 2.1: The Instruction Set

Programming as it exists today is drastically different than what someone would consider programming in the early 1970's. In general, most programmers use higher level languages such as C++, Python, Java, etc. while very few use assembly languages such as Windows X86 assembly language. Before the creation of these smoother, easier to use, higher level languages, the only way to program a computer was strictly just using the assembly language of the computer. Higher-level languages that most use now do the same thing, just much more superficially. Higher level languages allow the user to use assembly language in a more concrete and streamlined way. Everything you program breaks down into assembly language, then straight into machine code with which the computer can understand. As such, something that could take two lines to write in Java could take as much as five or six lines to write in assembly language. Having to program straight assembly language into the PDP 11/20 can be a hassle for this reason, and because there is a significantly higher amount of human interaction with the PDP 11/20 interface when programming assembly into there is a higher margin for human error. When you are writing programs that are hundreds or thousands of lines long, you do not want to have a mistake somewhere that would be close to impossible to find. For these reasons, we needed to get BASIC onto the PDP 11/20, but the instruction set is an extremely integral part of daily interactions with the PDP 11/20. Several subtle nuances must be brought to light to understand how the PDP 11/20 operates.

The PDP 11/20 has over 80 individual instructions that can be used in conjunction with each

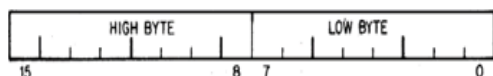


Figure 9: A diagram describing how an instruction is divided into two separate bytes of information, taking up two memory addresses, one for each byte.

other to do just about anything the heart desires. An

instruction written to the PDP 11/20's memory is in the form

of **Figure 9**. Two types of instructions can be written into

memory, single operand instructions, and double operand instructions. Single operand instructions

usually involve manipulating a single value stored in a register. An example of this can be seen in the INC

instruction in Appendix C. This instruction manages a single register by incrementing its contents by one. Double operand instructions manipulate two registers at the same time, such as the MOV instruction which can also be seen in Appendix C. This instruction can be used in several ways, but primarily involves moving the contents from another address or another register, into a different register.

Every instruction follows the general form of: Operation (OP) code, source mode, a source register, destination mode (if applicable), and the destination register (if applicable). Where the sections involving destination, registers are only used in double operand instructions. Because of the lack of

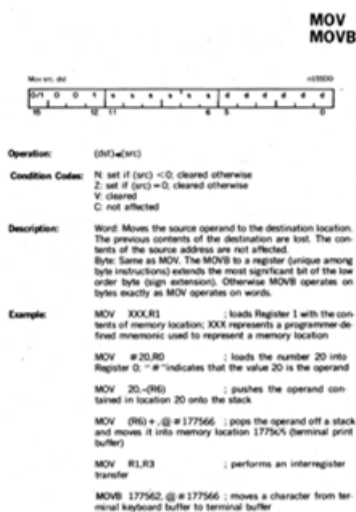


Figure 10: This diagram gives a more in depth and technical description of the MOV instruction. For a different description of this instruction, refer to Appendix C.

these fields in single operand instructions, the OP field is usually a 12-bit number. Whereas, the OP field in double operand instructions is often a 6-bit number. The OP field itself is like a nametag for the instruction; every individual instruction has a unique OP code which specifies which instruction you are using in each instance.

Individualizing instructions is, of course, is important because otherwise, the computer would confuse itself between two different instructions which may be written the same way (Olsen).

The PDP 11/20 has a vast repository of instructions at a user's disposal, many being only usable in very niche situations and techniques. There are a few, however, which are incredibly vital and make up roughly 70% of any program you may need to create on the PDP 11/20, and this paper would be remiss if it did not mention how they are used, and how you write them. These two dominant instructions are the MOV and BR commands (refer to Appendix C for a more comprehensive report on the instructions used throughout this project). Without these instructions, it would be close to impossible to do anything significant on the PDP 11/20. The MOV instruction (whose word diagram can be seen in **Figure 10**), as its name suggests, moves data from one location to another. This instruction is exceptionally dynamic and can be

used to transfer any data in any situation. The user can move data between registers, immediate numbers into registers, addresses onto the stack, the PC into a register, and several other cases. The BR (which stand for branch, and whose word diagram can be seen in **Figure 11**) instruction is significantly more complicated. BR lets you change the current address stored in the PC and lets you create what would be the equivalent of a for or while loops in a higher-level language. What makes it so complex is the several different manipulations that this single instruction can take on; based on various parameters you can write branch instructions that execute when different condition bits (N, Z, V, C) from the CPU status word are set. If the last instruction causes the result to be zero and sets the Z bit to be one, then you can make a branch that will branch on that condition. If the instruction causes a carry, and the C bit is set, then you can write a branch that branches on that situation. You can write conditional branches which check the values of two registers, and if they are equal then branch, or perhaps if you want to branch when their values are not identical; you can do that as well. There is no limit to how a user can manipulate the branch instructions to create a truly dynamic program that can do nearly anything (Olsen).

The past few paragraphs have explained in detail a few of the intricacies that go into programming the PDP 11/20. However, these are only pieces towards a larger whole. There are eight different ways (of which we will describe the most frequently used three), you can write the mode field of the source and destination registers. This secondary instruction field describes how you want to access the data from the register that it is referring to. The three primary modes that are used the most often are 00, 01, and 02. Mode 00 works by taking the data inside of a specified register and using that as the data to be manipulated. For instance, if we use the ADD instruction, and r0 contains 000002 and r1 contains 000001, and we want to add the actual numbers that are inside of r0 and r1 then we write,

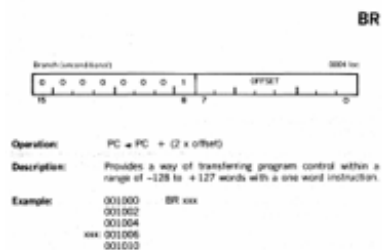


Figure 11: A more technical description of the BR instruction. A different description can be seen in Appendix C.

060001. The first two digits (06) are the OP code specific to the double operand ADD instruction; the third digit is the addressing mode of 0 which addresses on register 0 which is indicated by the fourth bit. The fifth digit is the same and points to an addressing mode of 0 and refers to register one which is shown by the last numeral. In this situation, the instruction would add the contents of r0 and the contents of r1 and store it into r1.

The addressing mode of 01 is like that of the zeroth addressing mode. This addressing mode is still used on a register, and it still refers to the value in that register. The difference, however, is that the 01 addressing mode looks at the value inside of the register that it refers to, and what is inside of that register, in this situation, should be another memory address. The instruction will then use the data inside of that deferred memory address as the data used in operation. This mode is used the most when working with peripheral devices, as this project relied on heavily. As a peripheral device is referred to by a memory address, we can manipulate data coming in or going out of individual ports using the peripheral address, and the 1st addressing mode. An example of this addressing mode is given in the MOV instruction. Let us say we want to take in data from the serial RX port (177566), and r0 contains said memory address. We can write the instruction 011001 to MOV data from the RX line into the r1 register. Where the first two digits 01 are the OP code for the MOV instruction, the third digit of 1 is the mode referring to the zeroth register, and store that data into the first register.

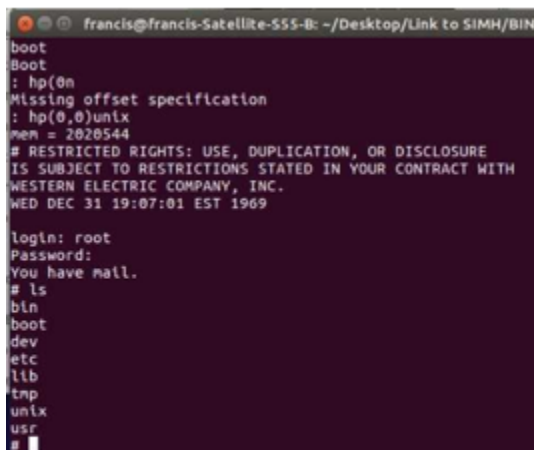
Lastly, for addressing mode 02, the PDP 11/20 will auto increment the data in the specified register, and use that auto incremented data as the data in our operation. This is a robust mode to use in conjunction with the PC register. In this way, you can increase the PC and use data in a successive memory address in an operation. This type of instruction is the only way to move immediate numbers straight into memory without direct user interaction. An example of this would be using the MOV instruction to move an immediate into r1. The successive instructions of 012701, 000004 will move the

immediate number 4 into r1. Where 01 is the OP code for MOV, the third and fourth digits indicate an auto-increment addressing mode on r7 (which is the PC) and move the data directly into r1.

Part 3: Methodology

Section 3.1: Simh

Simh is an instrumental, and versatile program that is of paramount importance when working with old and ancient computer systems. After the turn of the millennia, ancient computer systems like the PDP 11/20 had already been out of production for several decades, and because of their age, most surviving systems were deteriorating rapidly. Simh itself was designed for the sole purpose of immortalizing these old systems, and it contains repositories for hundreds of old systems, not just the



```

francis@francis-Satellite-555-B: ~/Desktop/Link to SIMH/BIN
boot
Boot
: hp(0n
Missing offset specification
: hp(0,0)unix
mem = 2020544
# RESTRICTED RIGHTS: USE, DUPLICATION, OR DISCLOSURE
IS SUBJECT TO RESTRICTIONS STATED IN YOUR CONTRACT WITH
WESTERN ELECTRIC COMPANY, INC.
WED DEC 31 19:07:01 EST 1969

login: root
Password:
You have mail.
# ls
bin
boot
dev
etc
lib
tmp
unix
usr
#

```

Figure 12: A demonstration run of UNIX v7 being run on the Simh PDP 11/20.

PDP 11/20. When researching ways to emulate anything on old hardware, it almost seems like the sentence “let us start by trying this on Simh” is an industry standard. Thus, initially after proposing this project, we were given the task of running the latest version of UNIX (version 7), on the Simh PDP 11/20. Fortunately, this has been done seemingly hundreds of times in the past, and it was not hard to find a generous amount of resources on the task.

A guide which we used by John D. Pressman on “How to Emulate UNIX V7 Using Simh (2015)” was particularly useful. We quickly realized that when using software so near to a computer systems architecture, we should be using Linux rather than any other operating system, as it gives the user access to tools which go much more in-depth than what Windows or Mac OS can provide. We created a dual boot for Ubuntu on our laptop to use Simh with the most efficiency. We were quickly able to

download, and unzip a directory for UNIX v7, simulate the PDP 11/20, and boot up UNIX v7 on the PDP 11/20 simulation (shown in **Figure 12**).

Undoubtedly, Simh is the most important tool we have used throughout the journey of getting UNIX v1 on the physical PDP 11/20. When we first started, we did not realize just how vital Simh would



Figure 13: The DL11 serial interface card which is stored in the chassis of the PDP 11/20, and handles communication between the PDP, and an external peripheral device.

be for writing assembly programs for the PDP 11/20. Debugging code on the PDP 11/20 is notoriously tricky and challenging as we quickly found out. Simh allowed us to be able to analyze code which we wrote by being able to step through our programs line by line, and analyze the registers, the current state of the computer, and make sure each line was doing what we

expected. Admittedly, we could

do this on the PDP 11/20, but

reading binary, and translating that binary into octal digits in one's head can give a large margin for human error, Simh gave visualization to what we were writing and allowed us to work as efficiently as possible.

Section 3.2: Communication

The PDP 11/20 uses a DL11 Serial Interface card (shown in **Figure 13**) to accomplish the task of communicating with peripheral devices. In the corner of one of the quadrants of the DL11, there is a port for a 40-pin

berg cable (the cable itself is shown in **Figure 14**) which would standardly be used to communicate with a teletype printer or a paper tape reader. In our situation, we would use the berg cable, along with a serial adapter cable (shown in **Figure 15**) which has wires for ground, 5 Volts, TX, and RX, to accomplish the tricky act of communication. Initially, it was hard to determine which specific pins correlated to the



Figure 14: The 40-pin berg cable classically would allow communication between the PDP 11/20 and a peripheral device.

input ports on the berg cable. Because the berg cable is manufactured with the intention of being able to be used in communication between almost any two devices, there were several ports on the cable which were unnecessary, and not needed to what we needed to accomplish. All we needed for the serial adapter was the ground, TX, and RX ports, which are only 3 of the 40 ports we could choose from. On the website *RETROCOMP*, we found an article which contained an image which significantly aided our ventures in finding the correct ports for the pins to be attached to (“Interfacing with a PDP 11/05”).



Figure 15: The serial adapter is attached to the ground, TX, and RX lines on the berg cable, and is used to allow communication between our laptop, and the PDP 11/20.

Now being able to set up a direct communication line between our laptop and the PDP 11/20, we began work on researching the assembly language of the

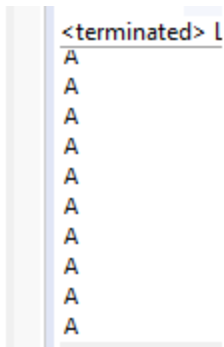


Figure 16: A small snippet of what the user would see in the console as the PDP 11/20 is rigorously pushing the character ‘A’ through the transmit line.

PDP 11/20 to create a program which could send the character ‘A’ across the transmit line. The simple program shown in Appendix A.P.1 was a quick creation that we knew would be able to accomplish the task needed. Testing the program resulted in a successful confirmation that the TX line on the PDP 11/20 worked how it should and would send the character ‘A’ constantly to our computer (as shown in **Figure 16**). The logical next step was to test that the RX line worked properly as well. Creating another program (seen in Appendix A.P.0) which would take a character sent from our laptop, and then the PDP 11/20 would send back the next alphabetically ordered character back to our laptop, so if the character

‘A’ was sent, then the character ‘B’ should be received back. We loaded the program into memory, tested it, and unfortunately, there was an immediate issue. The character which we would receive back was nowhere close to the character we were expecting. If the character ‘A’ were sent, then we would receive the character ‘}’, and always receive the character ‘}’; which in a way was a good thing because at least the behavior was not random, but somehow systematic. Furthermore, if we went through the

ASCII table and sent the character 'B', we would get again a seemingly random character on an entirely different part of the ASCII table.

Something was apparently wrong with the RX line. We went through a considerable amount of different ideas to try and resolve the issue: changing the program in the PDP 11/20 in case there was something wrong with it, however, further testing of the program on Simh showed that it did work properly on a more streamlined system, so the program was ruled out. We changed the berg cable in case the RX line on it was damaged or corroded somehow, this just led to verifying that the berg cable was indeed not the issue. We did the same with the serial adapter and came to the same conclusion. We dug into some research on the connections on the DL11 card, and thought we found some issues, tried re-soldering connections that we hoped would fix the problem, but again this led to no substantial discoveries, we even went as far as to apply for a grant, and buy a new DL11 card, but the new card gave the same output as always. We decided to try one last ditch effort which we had been discussing but had not tried yet because it would be a considerable detour away from the goal of the project; we were going to create a homemade serial console.

Up until this point we had been using a serial console created for an Arduino. We had no reason to believe that their software was corrupt in any way but creating a console could give us a lot more freedom to change and manipulate parameters such as baud rate, stop bits, parity, and data bits. For the PDP 11/20, the values of these parameters should be 1200 (because this was the prevailing baud rate from 1970 computers), 1 (because whenever a serial communication is set up, you need to have a bit which indicates the end of a line), 0 (the PDP 11/20 did not have a use for a parity bit), and 8 (because the PDP 11/20 is a 16 bit, 2 byte computer, which can only have words of size

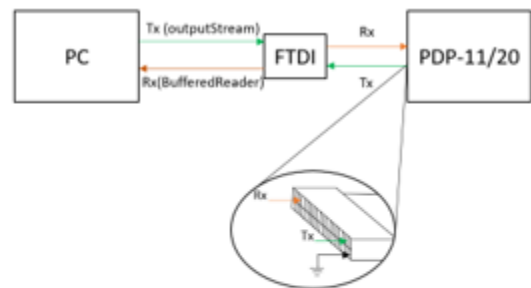


Figure 17: This diagram displays the communication setup between the PDP 11/20 and our laptop.

8 or 16), respectively. Changing any of these initial parameters could solve our issue. We discovered a guide online on how to create a homemade serial emulator using Java created by Henry Poon (Poon). With his guidance, we were able to develop a serial console (Appendix B) which gave us access to change any of the input parameters mentioned earlier (particularly on lines 180-195 is where we can manipulate these parameters). **Figure 17** shows the process of how the serial console initially sets up a communication with a specific COM port on our laptop, then sets up a buffered reader and an output stream to read information coming from the PDP 11/20 and to write data to the PDP 11/20 respectively. The console then prompts the user if they want to send a binary file or send a single character. Once the user has posted all the information they want, the console will wait for data coming from the PDP 11/20 and display anything received by the user.

Finally, after creating an entire serial console, we used it to see if the RX line would work based on various initialized parameters. To our dismay, it did not seem to help at all. We tried running several tests with several different combinations of the parameters, and no matter how we set up the connection, the character we received was incorrect. The only values of input parameters which gave a reasonable output (other outputs would result in bizarre characters that did not even show up on the ASCII table, they looked almost like Egyptian hieroglyphs) were the ones that seemed obvious to begin with; 1200 for baud rate, 1 for stop bits, 0 for parity, and 8 for data bits. Having invested so much time in creating a serial console and knowing that it was our last hope; made it extremely disheartening when it did not work. All our attempts to fix the issue pointed to the same terrifyingly daunting conclusion; that there was something innately, and profoundly wrong with an integral part of our PDP 11/20; attempting to fix that large of an issue would have been way out of the scope of this project. Everything seemed lost; until we noticed something interesting.

Up until this point, we had only been sending capital case letter characters; such as 'A', 'B', 'C', etc. We decided to send other characters in the ASCII table just to see if we could find anything

interesting, and that is just what we found. We noticed that when we sent certain other characters, we would get back 'A', 'B', and so on. Furthermore, no two sent characters would send back the same character. This pattern meant that somewhere in the ASCII table, every transmitted character had a relation to every received character, implying some sort of one to one function. We excitedly realized that there must be some way to manipulate a character before leaving our computer so that when the PDP 11/20 received the character, the PDP 11/20 would perform its integrity error, and transform it into the character we were trying to send all along. The only thing left to do was determine what this function was.

Sent	Received	Sent ASCII decimal value	Received ASCII decimal value
]	A	125	65
[B	123	66
Y	C	121	67
w	D	119	68
u	E	117	69
s	F	115	70
q	G	113	71
o	H	111	72
m	I	109	73
k	J	107	74
i	K	105	75
g	L	103	76
e	M	101	77
c	N	99	78
a	O	97	79
_	P	95	80
]	Q	93	81
[R	91	82
Y	S	89	83
w	T	87	84
u	U	85	85
s	V	83	86
q	W	81	87
o	X	79	88
m	Y	77	89
k	Z	75	90

Table 1: A table organizing the strange occurrence of the manipulation of the characters as they're sent across the receive line to the PDP 11/20.

The table on the right shows what we discovered. A clear pattern can be seen where the sending characters value is decreasing by two every time, but the received characters value is incremented by one every time. Now knowing this instead of fixing the actual issue of what was happening inside of the PDP 11/20, we mitigated the problem and were able to come up with a function of $y = \frac{x-127.5}{-.5}$. This function takes the character you want to send as input and changes it into a different character which when sent will be received by the PDP 11/20 as the original character you were trying to communicate. If we use the character 'A' as an example which has a decimal value of 65, then $y=125$ which, referring to the table, when the character '[' is sent (which has decimal value 125), then an 'A' is received by the PDP 11/20. Implementing this function into our serial console (which can be seen on line 40 of the serial console source code), we were able to communicate with the PDP 11/20 successfully and get the program of the PDP 11/20 receiving a character and sending the next sequential character back to work!

Section 3.3: The Bootstrap Loader

Having accomplished the task of communication; the next step was to acquire a binary file for BASIC (one of the world's first high-level languages) and load it onto the PDP 11/20. A higher-level programming language is something that could help us in achieving our final goal of loading UNIX v1 onto our PDP 11/20. We would be able to write programs in BASIC more easily than in the PDP 11/20's assembly language. We would be able to write an absolute loader for UNIX, to load UNIX into the PDP 11/20's core memory. Typically, loading BASIC onto a PDP 11/20 would be a reasonably easy task. When the PDP 11/20 was still being produced, paper tape images of BASIC could be obtained separately, and with a paper tape reader, the PDP 11/20 could easily read information from the paper tape, follow

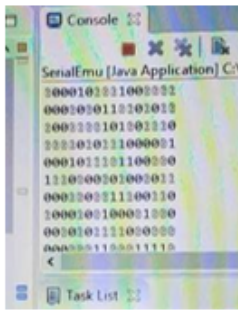


Figure 18: The above image demonstrates an instance of sending over the entirety of a binary file for the BASIC programming language. As each line is sent across to the PDP 11/20, it is also printed to the console to give a visual confirmation of its passage.

instructions given, and load BASIC. However, paper tape readers are excessively hard to find, and the ones that do exist do not come cheap. Without the necessary equipment on hand, we began to develop a bootstrap loader which could accomplish the task of loading BASIC onto the PDP 11/20.

To tackle the intricate task of creating our bootstrap loader, we must first determine how to send the information via the homemade serial console and decide whether we need to worry at all about the integrity error that was tormenting our PDP 11/20. Once we obtained a binary file of BASIC, we would need to open that file via our serial console and send over the individual characters in pieces. The PDP 11/20 stores an entire word as a 16-bit, six octal

digit word. While sending information over, we would have to split up a single 16-bit line and separate it into six octal digits. For instance, as shown in **Figure 18**, the number 1100111010001000 would be equivalent to the octal number 147210. Once translated into an octal digit, the serial console sends one of the following characters: 'D', 'E', 'F', 'G', 'H', 'I', 'J', or 'K'. Each of these characters correlates to a number

from 000 to 111. Thus, if the 3-bit binary number that is going to be sent is 000, then the serial console will first transform that number into a 'D' and send it (this process can be seen on lines 31-85 of the serial console source code in Appendix B). The reason that we start at 'D' instead of 'A' is that the integrity error that plagues our PDP 11/20 still will not receive the character 'C'. Thus, we started a little further down the alphabet. Also, the reason we did not use the characters '1', '2', '3', etc. to represent 000,001,010, etc. was because of the equation mentioned earlier in section 4.2. When we send the 'D', or 'E', etc, across we apply the function that will change the sent characters value, and then send the character. If we apply that function to the characters '1', '2', etc. then the characters that they would be translated into do not exist on the primary ASCII table. The characters reside only in the extended ASCII table, and just to be safe to avoid errors, we stuck with only translated characters that exist on the primary ASCII table.

An assembly program which would have to take in a string of binary digits from a serial port, and somehow store them in memory, promised to be a complex task; especially keeping in mind the integrity error that our PDP 11/20 was demonstrating. After developing a few versions, and ironing out several bugs, the loader (seen in Appendix A.P.2) was able to fully load a binary file of BASIC onto our PDP 11/20. **Figure 19** shows how the loader works visually, and **Figure 20** shows how information is sent from the serial console. The loader first initializes several registers: The register r0 is set to be the address register; which will start at address 0 and will be incremented every time we store an instruction in it. The register r1 is set up as the keyboard monitor device; which is the device which will receive the incoming characters from the laptop. The register r2 gets set up as an accumulator; which will start empty and will be filled piece by piece with the bits which have been converted from the characters sent by the laptop. The register r3 is set up

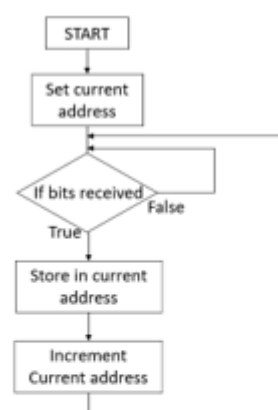


Figure 19: A very superficial diagram showing how that bootstrap loader on the PDP 11/20 reads in a binary file sent from our laptop.



Figure 20: The above flowchart depicts how our serial console sends the information from our BASIC binary file from our laptop to the PDP 11/20

as the for-loop iterator; which will act as a variable with which we can check to see if a condition is met over time. The register r4 is set up as the keyboard status register; which will be used to check if the TX line is ready to be read from or has anything waiting to be read. Lastly, r5 will be the register which will store the character sent over by the laptop. We clear out r5, just to make sure that there is nothing left in there from a previous for loop iteration, and we then hang up the program, waiting for something to be transmitted across to the keyboard status device of r4. Once something is detected, we move the lower end byte from the TX line into r5. We only want the lower byte just in case an unexpected transmitting error occurred and resulted in error bits propagating into the 15th or 16th bit. Next, we convert the character which was just taken in from the laptop into a 3-bit binary number by subtracting 104 from it. 104 is the ASCII value for the character 'D', thus, if the character 'D' is taken in, then subtracting 104 from it will result in 000. The same can be said for the character 'E' which is ASCII value

would be able to store this whole file on this computer, even with a complex system created by us do to so. At this point, we had to go back to the drawing board in looking for a different BASIC file which would hopefully satisfy the storage requirements for the PDP 11/20. With luck, we were able to find not only a different paper tape image of BASIC (seen in **Figure 21**) but also homemade absolute and bootstrap loaders which were made to accompany the paper tape (Par). The accompanied loaders for BASIC allowed us to quickly load and run the BASIC paper tape on simh rather smoothly, and we were able to get a prompt asking us for user input; which was somewhat promising. We were able to type in lines of BASIC code, send it over to the simulated PDP 11/20 via a telnet connection, and receive back the proper output for that given BASIC program. Having confirmed it to work on Simh, it was hopefully going to be an easy task of using our loader, and serial console, to load that BASIC binary file onto the physical PDP 11/20, hit start, and then be prompted to send over code written in BASIC. However, as one may be able to guess, things were not that simple. After manipulating the paper tape acquired from PCjs.org to be appropriately read by the serial console; we tried executing the program the same way we did on the simulated PDP 11/20, but this resulted in a near-immediate crash of the program. Unfortunately, this is the current state of the matter. Several diagnostic tests have been run on the PDP 11/20, and the binary file which has been loaded onto the PDP 11/20, and so far, each test has been inconclusive as to what may be causing the immediate crash.

Part 4: Conclusion

Section 4.1: Further work

With BASIC fully loaded and working correctly, the next steps are vast. The task next task moves away from the PDP 11/20 itself and would require a user to create an absolute loader for UNIX v1 using BASIC. An absolute loader would need to be intricately more complicated than the bootstrap loader which was designed to be a bootstrap loader for BASIC. One would have to decipher the ancient files

that have been on GitHub for so many years on how to use, compile, and execute UNIX v1 on a PDP 11/20. The logical step would be to undoubtedly try and create an executable for UNIX v1 written in the PDP 11/20's assembly language, load that onto the Simh PDP 11/20 using a program written in BASIC and execute it to load UNIX v1 onto Simh. In theory, this should work, and then be able to be accomplished on the physical PDP 11/20, but in practice, there are no doubts that several complications will arise somewhere.

In its heyday, the PDP 11/20 was a highly sophisticated, complex, and beautiful piece of human ingenuity. While the core concept of the computer has not changed much since the time of the PDP 11/20, computers have noticeably gotten smaller, and faster, but that does not mean that we should not admire these works of the past. Old technologies such as these have led us to new, creative, and meaningful innovations which have furthered the development of our society. Restoring old computers is not something that should just be done on a whim, or without much thought; it is something that should, and desperately needs to be accomplished. If one cannot understand what has got us to this point in history, then how can anyone make assumptions about the future? This ideology hints at the true, massive scope of this project. When working with such an ancient machine, several unexpected issues are expected to be encountered, and with such minimal resources at hand, can be excruciatingly painful to fix. Even with a clear path in mind, this project is hard to have a definitive timetable in mind. The unfortunate truth is that while this project has done the job of carving out a path to its destination, the PDP 11/20's glory has yet to be achieved. The building blocks have been put back in place for future students, and seekers of knowledge to dive themselves into the world of these complex architectures.

Appendix A: Assembly Programs

For information on what any of the instructions in the following programs do, refer to Appendix C.

P.0

Address	Machine Code	Instruction	Comment
000500	012701	mov [pc+],r1	put receive register into r1
000502	177562	177562	receive register address
000504	011102	mov [r1],r2	put what's in the receive into r2
000506	005202	inc r2	r2=r2+1
000510	012701	mov [pc+],r1	put transmit register into r1
000512	177566	177566	transmit register address
000514	010211	mov r2,[r1]	send it back out

P.1

Address	Machine Code	Instruction	Comment
000600	012701	mov [PC+], r1<---	move 'A' into r1
000602	000101	'A'	
000604	012702	mov [PC+], r2	move TX port into r2
000606	177566	177566	(TX data port)
000610	010112	mov r1, [r2]	move whats in r1 to the TX line
000612	000772	branch -6<-----	go back 6 lines

This program will move the character 'A' into register 1, and send into to the address which is stored in r2 (hence why the indirect addressing mode is invoked in the mov r1,[r2] instruction). On our laptops' side we would be able to see a constant stream of 'A's coming across the serial port.

P.2

Address	Machine Code	Instruction	Comment
037704	012700	mov [pc+],r0	start the address at 0
037706	000000	0	
037710	012701	mov[pc+],r1	put keyboard monitor into a register
037712	177562	177562	
037714	012702	mov[pc+],r2 <-----	set up the accumulator register
037716	000000	0	
037720	012703	mov[pc+],r3	set up the "for loop" iterator
037722	000000	0	
037724	012704	mov[pc+],r4<---	move keyboard status register into r4
037726	177560	177560	
037730	160505	sub r5,r5	clears out register to be used again
037732	105714	TSTB[r4]<--	is keyboard ready to take something in?
037734	100376	b -2>-----	if not try it again
037736	111105	movb [r1],r5	what's in keyboard monitor into r5
037740	162705	sub [pc+],r5	subtract 104 from r5
037742	000104	104	

037744	142705	BICB [pc+],r5			and r5 with 7 to get rid of any error bits
037746	177770	177770			
037750	006102	rol r2			mov bits in r2 over to left 3 times
037752	006102	rolr2			
037754	006102	rol r2			
037756	060502	add r5,r2			add the new character to r2
037760	005203	lnc r3			increase the iterator
037762	020327	cmp r3,[pc+]			do iterator-6. If result != 0 then branch
037764	000006	6			
037766	100760	BMI -19>-----			if result is not 0, branch
037770	010210	mov r2,[r0]			store the accumulator into an address
037772	062700	add [pc+],r0			increase the address for next time by 2
037774	000002	2			
037776	000746	b -24>-----			restart

P.2 is a more complicated program which is used as an absolute loader to load the BASIC bin file onto the PDP 11/20. Refer to section 4.3 for an in-depth description of its design, and how it's executed.

Appendix B: Serial Emulator

```

1 public class SerialEmu {
2     static SerialPort port = null;
3     static CommPortIdentifier portId = null;
4     static Scanner scan = new Scanner(System.in);
5     static BufferedReader fromPDP = null;
6     static PrintStream toPDP = null;
7     static int reversebits = 0, instructionLength = 8, counter = 0, readFromFile=1, Wait = 0;
8     static Boolean keepsending;
9     public static void main(String[] args) throws IOException, InterruptedException {
10        System.out.println("Welcome to the Serial Connection Emulator (SCE) for the PDP
11/20, proceeding with setup...");
12        setup();
13        System.out.println("Setup was succesful!");
14        System.out.println("Would like to send something or receive something? (s,r): ");
15        String answer=scan.next();
16        if(answer.equals("s")) {
17            keepsending=true;
18        }
19        else {
20            keepsending=false;
21        }
22        //main program loop
23        DateTimeFormatter timeStampPattern = DateTimeFormatter.ofPattern("yyyy MM dd
24        HH mm ss ");
25        DataOutputStream writer = new DataOutputStream(new FileOutputStream("Files\\" +
26        timeStampPattern.format(java.time.LocalDateTime.now()) + "log.txt"));
27        while(keepsending){
28            char[] charArray = null;
29            if(readFromFile==0) {
30                System.out.println("What would you like to send?");
31                //answer is what the user wants to send
32                answer = scan.next();
33                charArray = answer.toCharArray();
34            }
35            else {
36                //going to send a file
37                System.out.println("Enter the directory of the file you wish to send: ");
38                String Directory = "C:/Users/gilmo/eclipse-
39        workspace/SerialEmu/Files/BASICOctallnBinary.txt";
40                File file = new File(Directory);
41                BufferedReader reader = new BufferedReader(new FileReader(file));
42                String text = reader.readLine();
43                charArray = text.toCharArray();
44            }
45            System.out.println(charArray.length/16);

```



```

66         singleCharacter = 'H';
67         singleCharacter = (char) ((singleCharacter-
127.5)/-.5);
68     }
69     else if(ThreeBits.equals("101")) {
70         singleCharacter = 'I';
71         singleCharacter = (char) ((singleCharacter-
127.5)/-.5);
72     }
73     else if(ThreeBits.equals("110")) {
74         singleCharacter = 'J';
75         singleCharacter = (char) ((singleCharacter-
127.5)/-.5);
76     }
77     else if(ThreeBits.equals("111")) {
78         singleCharacter = 'K';
79         singleCharacter = (char) ((singleCharacter-
127.5)/-.5);
80     }
81     toPDP.print(singleCharacter);
82 }
83 }
84 }
85 }
86 else {
87     for(int i = 0; i<charArray.length; i++) {
88         char singleCharacter = charArray[i];
89         singleCharacter = (char) ((singleCharacter-127.5)/-.5);
90         toPDP.print(singleCharacter);
91     }
92 }
93 }
94 System.out.println((charArray.length)/16 + " instructions stored");
95 ArrayList<Character> received = new ArrayList<Character>();
96 delay();
97 //Test input stream and read from serial
98 received = receiveData();
99 System.out.println("received data");
100 for(int i = 0; i<received.size(); i++) {
101     char thingToWrite = received.get(i);
102     writer.write(thingToWrite);
103 }
104 System.out.println("Keep sending?");
105 String response = scan.next();
106 if(response.equals("N")) {
107     break;
108 }
109 while(!keepsending) {

```

```

110         String received;
111         System.out.println("receiveing...");
           //Test input stream and read from serial
112         received = receiveData().toString();
113         System.out.println(received);
114         if(received.isEmpty()) {
115             System.out.println("Finished receiving, check file!");
116             break;
117         }
118     }
119     writer.close();
120 }
121 public static ArrayList<Character> receiveData() throws InterruptedException{
           //function to take input from the PDP
           //create a buffer first
122     char[] cbuf = new char[100];
123     for(int i=0; i<cbuf.length; i++) {
124         cbuf[i]=' ';
125     }
136     ArrayList<Character> EndArray = new ArrayList<Character>();
137     try {
138         fromPDP.read(cbuf);
139     } catch (IOException e) {
140         e.printStackTrace();
141     }
142     for(int i = 0 ; i<cbuf.length; i++){
143         if(cbuf[i]!=' ') {
144             EndArray.add(cbuf[i]);
145         }
146     }
147     return EndArray;
148 }
149 public static void setup(){
           //sets up a connection name
150     String name = "PDP";
151     while(true){
152         while(true){
           //user enters a port like "COM6", and this finds all available COM ports and
           //checks them against what the user entered, but has been hard coded to "COM6"
153             System.out.println("Please enter the communication port "
                + "you believe your serial device is "
                + "connected to.");
154             System.out.print(">> ");
155             String wantedPortName = "COM6";//scan.next();
156             Enumeration portIdentifiers = CommPortIdentifier.getPortIdentifiers();
157             System.out.println("Printing com ports...");
158             while (portIdentifiers.hasMoreElements()){
                //this prints out all the possible COM ports

```

```

159         CommPortIdentifier pid = (CommPortIdentifier)
portIdentifiers.nextElement();
160         System.out.println(pid.getName());
161         if(pid.getPortType() == CommPortIdentifier.PORT_SERIAL
//checks all serial ports with requested port
&& pid.getName().equals(wantedPortName)){
162             portId = pid;
163             break;
164         }
165     }
166     if(portId == null){
//if it wasn't found then restart the connection
167         System.err.println("Could not find serial port at " + wantedPortName +
"...restarting");
168         continue;
169     }
170     break;
171 }
//cast commportidentifier portId to type SerialPort and attempt to open
172 try{
//sets up the found port as a serial port
173     port = (SerialPort)portId.open(name,10000);
174     break;
175 }catch(PortInUseException e){
176     System.err.println("Port already in use: " + e + "restarting...");
177 }
178 }
//setting parameters of serial interface. 1200 is baudrate, you can vary instruction length to still
setup correctly to different machine
//stopbits of 1 for every case, and no parity for every case
179 try {
180     if(instructionLength==8){
181         port.setSerialPortParams(1200, SerialPort.DATABITS_8,
182             SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
183     }
184     if(instructionLength==7){
185         port.setSerialPortParams(1200, SerialPort.DATABITS_7,
186             SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
187     }
188     if(instructionLength==6){
189         port.setSerialPortParams(1200, SerialPort.DATABITS_6,
190             SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
191     }
192     if(instructionLength==5){
193         port.setSerialPortParams(1200, SerialPort.DATABITS_5,
194             SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
195     }
196 }catch (UnsupportedCommOperationException e) {

```



```

    //a catch statement just in case something went wrong. Most likely the user entered
    something that wasn't 8,7,6,or 5 for the Databits
197     System.out.println(e + "Invalid System Parameters, stopping the connection...");
198     System.exit(1);
199 }
    //the checker is just to check to see if neither the buffer of the reader are set up. If the checker
    is 1 that means that one of them
    //didn't get setup properly, and it shuts down the connection because there's no point in having
    a connection that you can't use
200 int checker = 0;
201 try{
    //sets up a Buffered Reader to read what the PDP 11 is sending us
202     fromPDP = new BufferedReader(new InputStreamReader(port.getInputStream()));
203 }catch (IOException e){
    //in case something went wrong with setting up the buffer
204     System.err.println("Can't open input stream: write-only");
205     fromPDP = null;
206     checker = 1;
207 }
    //attempts to initiate toPDP print, errors are handled
209 try {
    //sets up a print steam to the PDP 11
210     toPDP = new PrintStream(port.getOutputStream(), true);
211 } catch (IOException e) {
212     if(checker==1){
213         //if the checker is 1 from not setting up the Buffer, then it just shuts down the
        connection
214         System.err.println("Could not open input or output streams, stopping the
        connection...");
215         System.exit(1);
216     }
217     System.err.println(e + "Can't open output stream: read-only");
218     toPDP = null;
219 }
    //closes the connection when the program stops
220 if (toPDP == null) toPDP.close();
221 if (fromPDP==null)
222     try {
223         fromPDP.close();
224     } catch (IOException e) {
225         e.printStackTrace();
226     }
227 if (port == null) port.close();
228 }
229 public static void delay() {
230     int delay = 0;
231     for(double i = 0; i<1000000; i++){
232         for(double j = 0; j<1000; j++){

```

```
233         delay++;
234     }
235 }
236 }
237 }
```

The above program is that of the serial emulator that was created for communicating with the PDP 11/20. Refer to section 4.2 for an extensive description of its design and execution.

Appendix C: Instruction Set

The following section is dedicated to be a reference for every instruction used by the assembly program written in Appendix B. You can find an in-depth description of the MOV and BR instructions in section 2.1. These are the two most commonly used instructions, but the vast amount of a program is made up of instructions other than these. Therefore, if any questions should arise as to what a specific instruction does, attention should be directed here.

ADD

0	1	1	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: (Src) + (Dst) store into (Dst)

Condition Codes:

Z: set if result = 0; cleared otherwise

N: set if result < 0; cleared otherwise

C: set if there was a carry from the most significant bit of the result; cleared otherwise

V: set if there was arithmetic overflow as a result of the operation, that is, if both operands were of the same sign and the result was of the opposite sign; cleared otherwise

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.

BICB

1	1	0	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: \sim (Src) and (Dst) store into (Dst)

Condition Codes:

N/A

Description: The BICB instruction clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the sources are unaffected.

BMI

1	0	0	0	0	0	0	1	Offset	Offset	Offset	Offset	Offset	Offset	Offset	Offset
---	---	---	---	---	---	---	---	--------	--------	--------	--------	--------	--------	--------	--------

Operation: (loc) stored into (PC) if N=1

Condition Codes:

NA

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation. The Offset is treated as a signed two's complement displacement to be multiplied by 2 (because of the fact that each instruction in a 2-byte instruction) and added to the program counter. The PC then points to the next word in sequence. The effect is to Cause the next instruction to be taken from the address "loc", located up to 127 words backwards or forwards (+-256 bytes).

BR

0	0	0	0	0	0	0	1	Offset	Offset	Offset	Offset	Offset	Offset	Offset	Offset
---	---	---	---	---	---	---	---	--------	--------	--------	--------	--------	--------	--------	--------

Operation: (loc) stored into (PC)

Condition Codes:

NA

Description: Provides a way of transferring program control within a limited range with a one-word instruction. The Offset is treated as a signed two's complement displacement to be multiplied by 2 (because of the fact that each instruction in a 2-byte instruction) and added to the program counter. The PC then points to the next word in sequence. The effect is to Cause the next instruction to be taken from the address "loc", located up to 127 words backwards or forwards (+-256 bytes).

CMP

0	0	1	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: (Src) – (Dst) store into (Dst) (particularity it's (Dst) + ~(Src) + 1 store into (Dst))

Condition Codes:

Z: set if result = 0; cleared otherwise

N: set if result < 0; cleared otherwise

C: set if there was a carry from the most significant bit of the result; cleared otherwise

V: set if there was arithmetic overflow as a result of the operation, that is, if both operands were of the same sign and the result was of the opposite sign; cleared otherwise

Description: Arithmetically compares the source and destination operands. Affects neither the Src or Dst operands. The only reason to do this is if on the next line you have a JMP or BR command which is activated under certain pretenses of the condition codes.

INC

0	0	0	0	1	0	1	0	1	0	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	---	---	---	---	---	---	-----	-----	-----	-----	-----	-----

Operation: (Dst) + 1 store into (Dst)

Condition Codes:

Z: set if result = 0; cleared otherwise

N: set if result < 0; cleared otherwise

C: not affected

V: set if (Dst) used to hold 077777; cleared otherwise

Description: Adds one to the destination

MOV

0	0	0	1	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: (Src) store into (Dst)

Condition Codes:

Z: set if (Src) = 0; cleared otherwise

N: set if (Src) < 0; cleared otherwise

C: not affected

V: cleared

Description: Moves the source operand to the destination location. The previous contents of the destination are lost. The contents of the source are not affected.

MOVB

0	0	1	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: N/A

Condition Codes:

N/A

Description: Moves high order byte from source register into Destination register

ROL

0	0	0	0	1	1	0	0	0	1	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	---	---	---	---	---	---	-----	-----	-----	-----	-----	-----

Operation: $(Src) - (Dst)$ store into (Dst) (particularly it's $(Dst) + \sim(Src) + 1$ store into (Dst))

Condition Codes:

Z: set if all bits of the result word are 0; cleared otherwise

N: set if the high order bit of the result word is set, and thus rolled over; cleared otherwise

C: loaded with the high order bit of the destination

V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)

Description: Rotates all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into bit 0 of the destination.

SUB

1	1	1	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: $(Src) - (Dst)$ store into (Dst) (particularly it's $(Dst) + \sim(Src) + 1$ store into (Dst))

Condition Codes:

Z: set if result = 0; cleared otherwise

N: set if result < 0; cleared otherwise

C: set if there was a carry from the most significant bit of the result; cleared otherwise

V: set if there was arithmetic overflow as a result of the operation, that is, if both operands were of the same sign and the result was of the opposite sign; cleared otherwise

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected.

TSTB

0	0	1	0	Src	Src	Src	Src	Src	Src	Dst	Dst	Dst	Dst	Dst	Dst
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Operation: $(Dst) - 0$

Condition Codes:

Z: set if result = 0; cleared otherwise

N: set if result < 0; cleared otherwise

C: cleared

V: cleared

Description: Examines the high order bit of the high order byte in the Src register. Used simply just to set the condition codes Z and N. Mostly used to check status registers to see if they're ready to be used.

Bibliography

Beiser, Johan, et al. "Restoration of 1st Edition UNIX Kernel Sources from Bell Laboratories." Github, Online Repository, 4 May 2008, github.com/jserv/unix-v1.

"DEC PDP 11/20." History of Computers and Computing, Birth of the Modern Computer, Electronic Computer, PDP-11, History-Computers.com, history-computer.com/ModernComputer/Electronic/PDP-11.html.

Par, Jeff. PCjs: DEC PDP-11 BASIC, www.pcjs.org/apps/pdp11/tapes/basic/.

Poon, Henry. "Serial Communication in Java with Example Program." Henry Poon's Blog, 22 Jan. 2016, blog.henrypoon.com/blog/2011/01/01/serial-communication-in-java-with-example-program/.

"Interfacing with a PDP 11/05: Sorting the Wires." RETROCMP, RETROCMP, retrocmp.com/how-tos/interfacing-to-a-pdp-1105/144-interfacing-with-a-pdp-1105-sorting-the-wires.

Olsen, Kenneth H. PDP 11/20 Handbook. Digital Equipment Corporation, 1971.

Pressman, John D. "How To Emulate Unix V7 Using Simh (2015)." Jdpressman.com, 27 Nov. 2015, [www.jdpressman.com/2015/11/27/how-to-emulate-unix-v7-using-SIMH-\(2015\).html](http://www.jdpressman.com/2015/11/27/how-to-emulate-unix-v7-using-SIMH-(2015).html).

"SimH." The Computer History Simulation Project, 3.9, The Computer History Simulation Project, 3 May 2012, simh.trailing-edge.com/.

Toomey, Warren. "The Restoration of Early UNIX Artifacts." Usenix.com, www.usenix.net/legacy/events/usenix09/tech/full_papers/toomey/toomey.pdf.

"UNIX Past." Open Group, http://www.unix.org/what_is_unix/history_timeline.html.