Loading a program from tape - Part 1: Introduction

I thought it might be fun to figure out how to load a program from paper tape. This exercise actually revealed a lot of very interesting information so I'm going to try and write a little mini-series of posts, covering in detail all of the different aspects of the paper tape loading process that I have discovered.

This first post is a high-level overview of how tape loading works and an example of how to load a program from a virtual tape in an emulated PDP-11.

Overview of approach

The general approach to loading a tape is as follows;

1. A bootstrap program is loaded into memory and executed.
2. The bootstrap program loads a tape loader program, known as the absolute loader.
3. The absolute loader is used to load the program from tape into memory.

Sample code

To demonstrate the process, I have written a demonstration program to load BASIC from paper tape into memory and run it. Here is the sample code:

```
set cpu 11/70,4M
;set realcons=localhost
set realcons panel=11/70
set realcons interval=8
set realcons connected

set ptr enable
set ptp enable

D 157744 016701
D 157746 000026
D 157750 012702
D 157752 000352
D 157754 005211
D 157756 105711
D 157760 100376
D 157762 116162
D 157764 000002
D 157766 157400
D 157770 005267
```

```
D 157772 177756
D 157774 000765
D 157776 177550

attach ptr DEC-11-L2PC-PO.ptap
go 157744

attach ptr DEC-11-AJPB-PB.ptap
go 157500
```

To execute this code, you'll need to download and place the two ptap files into the same folder as the boot.ini. You can find the links in the references section below.

Code walkthrough

These first four lines are the standard boilerplate to set up the CPU and panel display.

```
set cpu 11/70,4M
;set realcons=localhost
set realcons panel=11/70
set realcons interval=8
set realcons connected
```

The next two lines enable the paper tape reader and the paper tape punch.

```
set ptr enable
set ptp enable
```

The next series of lines deposit the bootstrap loader into memory. I am going to write an entire separate post where I describe line-by-line the operation of the bootstrap loader. For now, suffice to say that this code loads the absolute loader from tape.

```
D 157744 016701
D 157746 000026
D 157750 012702
D 157752 000352
D 157754 005211
D 157756 105711
D 157760 100376
D 157762 116162
D 157764 000002
D 157766 157400
D 157770 005267
D 157772 177756
D 157774 000765
D 157776 177550
```

The next line loads the virtual absolute loader tape into the paper tape reader.

```
attach ptr DEC-11-L2PC-PO.ptap
```

The next line executes the bootstrap loader by setting the program counter value to 157744, which is the first memory address of the bootstrap loader code above:

```
go 157744
```

When the bootstrap loader completes loading the absolute loader into memory the code halts to allow the operator to switch out the absolute loader tape and insert the tape that contains the program that is to be loaded. The operator would then press the continue button on the operator panel to execute the absolute loader and load the program from tape.

The next line loads the virtual BASIC tape into the paper tape reader.

```
attach ptr DEC-11-AJPB-PB.ptap
```

The final line of code executes the absolute loader to load the BASIC tape into memory.

```
go 157500
```

Output from running the code

If the code runs successfully, BASIC will start running and you will see output something like this:

```
*** booting tapeload1 ***
*** Start client/server ***
*** RPi 4 detected

PDP-11 simulator V4.0-0 Current  REALCONS build Aug 14 2019
Disabling XQ
Searching realcons controller "11/70" ...
Connecting to host localhost ...

HALT instruction, PC: 157500 (MOV PC,SP)

PDP-11 BASIC, VERSION 007A
*O
```

Loading a program from tape - Part 2: The Bootstrap Loader

Updated: Jan 29, 2021

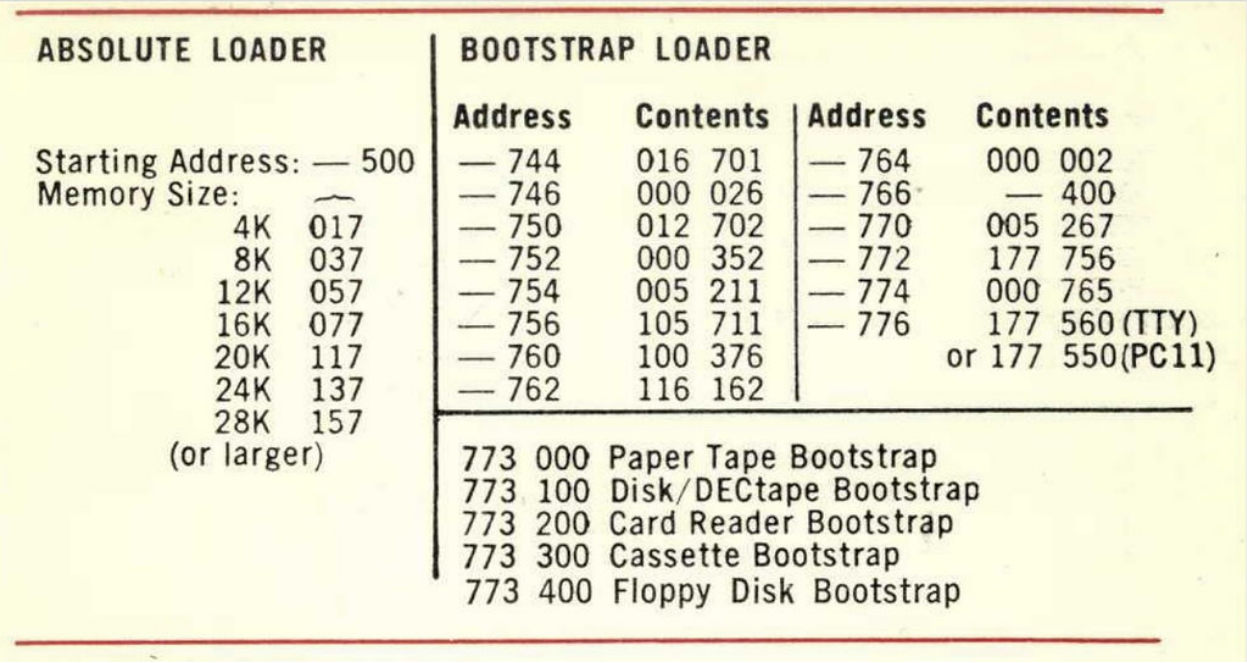
This post is a deep-dive on the PDP-11 bootstrap loader. It is a follow on from Part 1 of this series on loading a program from tape, which is recommeded reading before this post.

Running the bootstrap loader

The sole purpose of the bootstrap loader is to load the absolute loader into memory from an external media, such as from a paper tape. When the bootstrap loader was being used with PDP-11 hardware the bootstrap loader have been loaded by either being manually entered at the operator panel or loaded from a boot ROM. In a modern simulated environment the bootstrap loader is entered into the configuration file.

Configuring the bootstrap loader

If you take a look, for example, at the PDP-11 Programming Card, you will see that the bootstrap loader code is there. However, the code is provided in the form of a sort of template that you need to adjust according to your requirements:



ABSOLUTE LOADER

Starting Address: — 500
Memory Size:

4K	017
8K	037
12K	057
16K	077
20K	117
24K	137
28K	157

(or larger)

BOOTSTRAP LOADER

Address	Contents	Address	Contents
— 744	016 701	— 764	000 002
— 746	000 026	— 766	— 400
— 750	012 702	— 770	005 267
— 752	000 352	— 772	177 756
— 754	005 211	— 774	000 765
— 756	105 711	— 776	177 560 (TTY)
— 760	100 376		or 177 550 (PC11)
— 762	116 162		

773 000 Paper Tape Bootstrap
773 100 Disk/DECtape Bootstrap
773 200 Card Reader Bootstrap
773 300 Cassette Bootstrap
773 400 Floppy Disk Bootstrap

Firstly, note that the "Contents" of the bootstrap loader are listed but the memory addresses at which the bootstrap loader should be deposited are incomplete. That's because the location for the bootstrap loader depends on how much physical memory was contained in the machine you were loading the bootstrap loader onto.

On the left hand side of the image above you can see a "Memory Size" table. If your PDP-11 had 4K of memory, you used addresses for the bootstrap loader starting with "017". If your PDP-11 had 8K of memory, all of your addresses start with "037", and so on.

When I am emulating the PDP-11, I use 4M of physical memory so I use "157" for all addresses, so all of the "hyphens" in the bootstrap loader listing are replaced with "157" (or whatever is the appropriate choice in your case). In my case, for example, the first instruction of the bootstrap loader ("016 701") is deposited at memory address "157 744". Note that you also need to use the appropriate offset to complete the instruction word at address "- 766".

The reason why this is done is that it was intended that the bootstrap loader would be deposited at the highest available memory address locations, which will obviously depend on how much memory is available.

The other thing to note is that the final word of the bootstrap loader, in my case at memory address "157 776", has two possible values; "157 560" for TTY and "157 550" for PC11. This is to configure where you want the bootstrap loader to load the absolute loader from. You use "157 560" to load the absolute loader via a TTY and you use "157 550" to load the absolute loader from paper tape (i.e. from a PC11 tape reader device). This value is the memory address of the Control and Status Register (CSR) of the device from which the absolute loader will be loaded.

The bootstrap loader

So, here is my bootstrap loader. The first item in each row is a memory address and the second is the value to be placed at that memory address. Note that, as discussed above, all addresses start with "157" and that the last word has the value "157 550" (the CSR of the paper tape reader).

```
157744 016701
157746 000026
157750 012702
157752 000352
157754 005211
```

```
157756 105711
157760 100376
157762 116162
157764 000002
157766 157400
157770 005267
157772 177756
157774 000765
157776 177550
```

Code walkthrough

Let's take the code instruction-by-instruction and see how it works.

```
157744 016701
157746 000026
```

The first two octal digits in the first word ("01") are a MOV instruction. The second two octal digits ("67") indicate that the source value to be moved can be found at an offset from the Program Counter. The offset is found in the next word ("000026").

When an instruction is being executed, the content of the location pointed to by the PC is loaded and then the PC is incremented before the instruction is actually executed. Similarly, when an offset value is being read the value is read and the PC is incremented before the value is used. Therefore, the PC always points at the word **after** the instruction that is currently being executed. In this case, as the MOV instruction is being executed, the PC will have the value 157750.

The offset here is 26 (octal), meaning that the source operand for the MOV instruction will be found at address 157776. As mentioned above, this value is the address of the control and status register (CSR) of the input device, in this case the paper tape reader.

The final two octal digits of the instruction ("01") are the destination operand, which in this case is register R1.

In summary, these first two words represent "MOV 26(PC), R1".

```
157750 012702
157752 000352
```

These next two octal words are another MOV instruction, as indicated by the first two octal digits in the first word ("01"). The next two octal digits ("27") reflect the source operand of the

move instruction. In this case they represent the immediate value pointed to by the PC. Remember that at the time the MOV instruction is being executed, the PC will already have been incremented, so it will point at address 157752. In other words the, source operand is the immediate value 352 (in octal).

The destination in this case, represented by the final two octal digits of the MOV instruction ("02") is register R2.

Therefore, these two words represent the instruction "MOV #352, R2".

```
157754 005211
```

The next instruction is an increment word instruction, indicated by the four octal digits "0052". The operand to be incremented is represented by the final two octal digits ("11"). In this case, the value to be incremented is the value in the memory address that is contained in register R1. Put another way, this word represents the instruction INC (R1).

Remember that R1 contains the address of the CSR of the paper tape reader. Incrementing this value sets the lowest bit of the CSR to 1, which enables the device.

```
157756 105711
157760 100376
```

The next two instructions are a loop to wait for the device to become ready. The paper tape reader is significantly slower than the CPU (at least at the time when physical hardware was being used rather than emulation!) and so the CPU needs to wait until the paper tape reader has a byte of data ready to be loaded into memory.

The first word is a TSTB instruction (given by the octal values "1057"). This value will set the flags in the Processor Status Word (PSW) based on the value of the operand. The operand in this case is the value in the memory address that is contained in register R1, given by the octal digits ("11"). Again, recall that this is the CSR of the paper tape reader.

Bit 7 of the CSR value represents the DONE status bit of the paper tape reader. At the same time, the highest bit of any signed binary value represents the sign of that value with a "1" representing a negative value and a "0" representing a positive value. Therefore when bit 7 of the low byte of

the CSR is set to 1, that will mean that the byte has a negative value whereas if bit 7 is zero, the CSR will have a positive value.

That brings us onto the second of these instructions, the branch instruction, which is a BPL instruction, or branch if positive, given by the octal digits ("100"). This instruction will test the value of the "N" bit in the PSW. If the value tested, in this case the CSR of the paper tape reader, has a negative value, the N bit will be set, otherwise the N bit will not be set.

The branch instruction will branch if the value of the CSR was positive, meaning that bit 7 was not set, which in turn means that the paper tape reader does not yet have a value ready to be loaded into memory.

The location to branch to if the value of the CSR is positive is given by the offset portion of the branch instruction, which has the octal value 376. Without digressing into a whole discussion about how 2's complement numbers are structured, suffice to say that this is a branch offset of -2.

Recall that the PC will have been incremented before the branch instruction is executed, so it will have the value 157762. Therefore the -2 offset, two words back from the value of the PC, will branch back to address 157756, which is the TSTB instruction again.

In other words, these two instructions loop until bit 7 of the CSR of the paper tape reader is set, indicating that the paper tape reader has a value ready to be loaded into memory.

```
157762 116162
157764 000002
157766 157400
```

These next three words represent a single MOVB (move byte) instruction, as reflected by the first two octal digits of the first word ("11").

The source operand of the move instruction is specified by the second two octal digits ("61"). This value means that the source is the value of a memory location obtained by adding the value in the R1 register (i.e. the CSR of the paper tape reader) plus the value specified in the next word of the instruction ("000002"). In other words the value "157 552", which is the data buffer register of the paper tape reader.

The destination operand of the move instruction is specified by the final two octal digits ("62") of the first word. This value means that the destination is the value of a memory location obtained by adding the value in the R2 register (which currently contains the octal value 352) plus the value specified in the final word of the instruction ("157400"). This gives a destination memory address of "157752".

Note that this value is actually one of the addresses that make up the bootstrap loader program itself. In fact, it is the memory address location of the value placed in register R2 and just used to determine the memory address location where the byte read from the paper tape is to be stored.

```
157770 005267
157772 177756
```

This next instruction is an INC (increment) instruction, as indicated by the octal value "0052". The location to be incremented is specified as a PC offset in the next word. The value "177756" is "-18" in 2's complement. The value in PC, after the location offset word has been read is "157774". Subtracting 18 from this gives a memory address of "177752".

So, in summary, this instruction increments the value in memory location "177752".

```
157774 000765
```

The final instruction in the bootstrap loader program is an unconditional. The branch instruction is "0004", which makes the branch offset 365 in octal, or -11 in decimal. So, after reading this instruction the PC will have the value "157776". Branching back 11 means subtracting 11 words (or 22 bytes) from the memory location, which means that this instruction branches back to memory location "157750".

```
157776 177550
```

The final byte of the bootstrap loader is the address of the CSR of the input device, in this case the CSR of the paper tape reader.

Summary of code

Here is a summary of the bootstrap loader:

1. Move address of paper tape reader CSR into R1
2. Move destination address offset into R2

3. Enable the paper tape reader and wait for a byte to become available
4. Move a byte from the paper tape reader buffer into the memory address 157400 + the destination address offset in R2.
5. Increment the destination address offset
6. Loop back to number 2

The sharper-eyed reader might have noticed that this code appears to be an infinite loop. So, how does it ever stop reading bytes from the paper tape? That's the subject of the next in this series of posts.

Loading a program from tape - Part 3: Loading the Absolute Loader

This post describes the process by which the bootstrap loader loads the absolute loader. It is part of a series of posts, so it won't make much sense unless you read Part 1 and Part 2 first.

The code of the absolute loader is read byte-by-byte by the bootstrap loader and placed into memory. For the purposes of explaining the loading of the absolute loader, I will distinguish three parts to the process of loading the absolute loader, which I refer to as the beginning, the middle and the end.

The beginning

The first part of the loading process sets up the address into which the absolute loader will be loaded. Here are the first few octal values of the absolute loader:

```
351 351 351 075 000 000 306 021 246 051
```

Note that the first three octets have the same value, 351. This is called the tape leader. Let's take a look at what happens when these bytes are read in.

The key thing to watch when figuring out the operation of the bootstrap loader is the value contained in memory address "157752", which is the address offset, added to the base value of "157400", to determine the location to store the byte read from tape.

Initially this has the value 352, so when the first byte is read from tape (octal 351), the value is stored in the location 157752 (which is 157400 + 352). In other words, the value 352 at address 157752 in the bootstrap loader program is overwritten with 351. After reading and storing a value, the next instruction in the bootstrap loader program increments the value at address 157752, restoring the value to 352 again.

The same cycle is repeated for the other two leader bytes.

The next byte (octal 075) is read in and replaces the value 352 at address 157752. After reading and storing the value, the next instruction in the bootstrap loader increments the value at address 157752, resulting in the value 76. At this point, the operation of the bootstrap loader changes.

The next byte is read in (octal 000) and stored at address 157476 (which is 157400 + 76). The value at address 157752 is incremented and now has the value 77.

The next byte is read in (another octal 000) and stored at address 157477 (which is 157400 + 77). The value at address 157752 is again incremented and now has the value 100 (octal remember!).

We have now read in one full instruction; address 157476 contains the word "000 000", which is the HALT instruction.

The middle

Notice when we read in the two bytes that made up the HALT instruction, the bytes are read in little endian order. In other words, the least significant byte is read in first and the most significant byte second.

Let's do one more example. Remember that the value in 157752 is now 100.

The next byte read in is octal 306. This is stored at address 157500 (157400 + 100) and the value at 157752 is incremented to 101. The next byte read in is octal 021, which is stored in address 157501, and the value in 157752 is incremented to 102.

We have now read in another full instruction. Let's work out what it is.

We read 306 (binary 1100 0110) into the low byte and 021 (0001 0001) into the high byte. Merging these into a single word in binary, and arranging into sets of three bits we get "0 001 000 111 000 110", which in octal is 010706.

The first two octal digits ("01") represent the MOV instruction. The source operand can be found in the second two octal digits ("07"), which represents register 7, or the PC. The destination operand is found in the final two octets ("06"), which represent register 6, or the SP.

In summary, this instruction is "MOV PC, SP".

This process is repeated for almost all of the remaining bytes on the paper tape, representing the middle section of the loading process. The last few bytes have a different meaning, which will be discussed in the final section.

The end

Skipping ahead to the last few bytes in the absolute loader, we find:

```
301 035 026 000 302 025 373 353 000 000 000 000 000
```

At this point in the load, the value in memory address 157752 is 344, which means that the next byte will be written to address 157744 (which is 157400 + 344). Note that this address overlaps with the memory location where the bootstrap loader is stored, so these remaining bytes are going to overwrite some of the bootstrap loader.

```
301 035 026 000 302 025
```

These six bytes represent three word instructions that will overwrite the first three words of the bootstrap loader as follows:

```
157744 016701
157776 000026
157750 012702
```

If you refer back to the listing of the bootstrap loader program you will see that these are identical to the first three words of the bootstrap loader, so these bytes are overwritten with identical values.

Having replaced these values, the value in address 157752 is now 352, meaning that the next byte value is going to be read into address 157752. So, the next byte, 373, is read in and replaces the value 352 at address 157752. The next instruction in the bootstrap loader then increments this value, increasing it to 374.

The bootstrap loader loops again and reads in the final byte, with the value 353. This value is stored at address 157774 (157400+374). This is the location of the branch instruction at the end of the bootstrap loader program.

If we take a look at the structure of the unconditional branch instruction, the high byte consists of the value 0000 0001 in binary and the low byte consists of the offset to branch by. The offset value is a signed 2's complement value so that branches can branch forward with a positive offset or backwards with a negative offset.

So, when the low byte at address 157774 is overwritten, the branch instruction remains but the offset is changed. The value 353 is a negative offset of -25, which means that the bootstrap loader will branch backwards by 25 words, to address 177724.

This is how the bootstrap loader infinite loop is ended.

Restoring the bootstrap loader

When control jumps out of the bootstrap loader loop, a couple of instructions are executed to restore the bootstap loader program so it can be used again if needed.

These are the instructions, starting at address 157724, that restore the bootstrap loader:

```
157724 012767
157726 000352
157730 000020
157732 012767
157734 000765
157736 000034
157740 000167
157742 177532
```

The first instruction:

```
157724 012767
157726 000352
157730 000020
```

replaces (with a MOV instruction) the value 352 into address 157752 which, as this code is executing is found at address PC+20.

The second instruction:

```
157732 012767
157734 000765
157736 000034
```

replaces (again with a MOV instruction) the value 765 into address 175774 which is found at address PC+34.

The final instruction is an unconditional jump:

```
157740 000167
157742 177532
```

The jump offset is found in the address 157742. It is a negative value, reflecting an offset of -246 (octal), which will jump to address 157476.

Halting and waiting

When control jumps to 157476, remember this was the very first instruction read from the paper tape, which was a HALT.

The processor now halts and waits for the operator to change the tape and insert the tape that will be loaded by the absolute loader program. When the operator has changed the tape, the press continue on the panel and the absolute loader program will execute and load the program from the tape.

Loading a program from tape - Part 4: The Absolute Loader

Updated: Jan 22, 2021

This is the fourth part in a series on loading a program from paper tape. To understand what's going on, I recommend you read Part 1, Part 2 and Part 3 before reading this post.

Recap of where we are

Let's summarise where we are. So far we have;

1. loaded the bootstrap loader
2. run the bootstrap loader to load the absolute loader
3. restored the bootstrap loader so it can be used again if needed
4. halted waiting for the operator to change the tape

If any of that doesn't make sense, you really need to read the previous parts of this series.

Now we want to use the absolute loader to read in a program from tape. The absolute loader expects paper tape data to be formatted in a particular way, as described in the next section. After that, we'll take a detailed look at how the absolute loader works.

Absolute loader tape format

The absolute loader tape format is extremely simple. The tape is made up of blocks of data, with each block being structured in the following way:

1. The first byte has the value 001
2. The second byte has the value 000
3. The next two bytes are the number of bytes in the block, including the header, stored little endian. This is referred to as the byte count.
4. The next two bytes are the memory address to load the block into, stored little endian. This is referred to as the load address.

The remainder of the block contains data bytes, with the exception of the final byte, which is a checksum to verify the content of the block.

If the block has a byte count of 6, in other words the block contains no data, the load address is interpreted as the address to jump to in order to start the program. This address should always be even. If it is odd the absolute loader will halt.

The absolute loader

Here is the full listing of the absolute loader. I've split it into a few sections so it's a bit less intimidating.

```
;the halt instruction
157476 000000

;setup the absolute loader
157500 010706
157502 024646
157504 010705
157506 062705
157510 000112
157512 005001
157514 013716
157516 177570
157520 006016
157522 103402
157524 005016
157526 000403
157530 006316
157532 001001
157534 010116

;seek to the beginning of a block then read block header
157536 005000
157540 004715
157542 105303
157544 001374
157546 004715
157550 004767
157552 000074
157554 010402
157556 162702
157560 000004
157562 022702
157564 000002
157566 001441
157570 004767
157572 000054
157574 061604
157576 010401

;read in the rest of the block
157600 004715
```

```
157602 002004
157604 105700
157606 001753
157610 000000
157612 000751
157614 110321
157616 000770

;subroutine to read a byte
157620 106703
157622 000152
157624 105213
157626 105713
157630 100376
157632 116303
157634 000002
157636 060300
157640 042703
157642 177400
157644 005302
157646 000207

;subroutine to read a word
157650 012667
157652 000046
157654 004715
157656 010304
157660 004715
157662 000303
157664 050304
157666 016707
157670 000030

;subroutine to check jump address and jump to the loaded program
157672 004767
157674 177752
157676 004715
157700 105700
157702 001342
157704 006204
157706 103002
157710 000000
157712 000700
157714 006304
157716 061604
157720 000114
157722 000000

;this is the code that restores the bootstrap loader
;(discussed in the previous post)
157724 012767
157726 000352
157730 000020
157732 012767
157734 000765
157736 000034
157740 000167
```

```
157742 177532

;this is the code that overwrites the bootstrap loader
;(discussed in the previous post)
157744 016701
157746 000026
157750 012702
373
353
```

In that form it is still obviously completely incomprehensible but actually as we work through it you'll see that it's not too bad, because it consists of a few subroutines and when you figure it out it all makes a lot of sense.

The HALT instruction

The first part of the absolute loader is the halt instruction:

```
;the halt instruction
157476 000000
```

After the bootstrap loader has been restored, control jumps to this instruction and the CPU is halted. The reason for this was that the operator of the PDP-11 would need to remove the absolute loader paper tape from the paper tape reader and put in the paper tape containing the program that was to be loaded.

When the paper tape had been changed over, the operator would press continue on the panel and that would step forward to the next instruction which began to execute the absolute loader and read in the tape.

Setting up the absolute loader

The next section of the code sets up the absolute loader.

```
;setup the absolute loader
157500 010706
157502 024646
157504 010705
157506 062705
157510 000112
157512 005001
157514 013716
157516 177570
157520 006016
```

```
157522 103402
157524 005016
157526 000403
157530 006316
157532 001001
157534 010116
```

Let's walk through this instruction-by-instruction.

```
157500 010706
157502 024646
```

The absolute loader uses subroutines, which requires storing return addresses on the stack. These first two instructions set up the stack pointer.

The first instruction is a MOV (octal "01"). The source operand is register 7, the program counter (octal "07") and the destination operand is register 6, the stack pointer (octal "06").

This assigns the stack pointer to be equal to the value of the program counter. Remember that the instruction is read and the progam counter is incremented before the instruction is executed. Therefore, at the time that this instruction is being executed, the value of the program counter is 157502.

There are two instructions in the program that precede this memory location; the MOV instruction just executed and the HALT instruction. Recall from my primer on the use of the stack pointer, that the stack pointer is decremented before it is used so we want the stack pointer to initially point to the memory address of the first instruction of the program 157476.

Therefore, we need to decrement the stack pointer twice to point at the address of the HALT instruction (157476). The second instruction (at address 157502) performs a very neat trick of decrementing the stack pointer twice with a single instruction.

The instruction is "CMP -(SP), -(SP)". This is a compare instruction (octal "02") but we don't care about the result of the compare, all we care about is the side-effect of gathering the two operands. The source operand (octal "46") means decrement by one word and then read the stack pointer. The destination is also octal 46, meaning decrement by one word and then read the stack pointer again. These two values will now be compared, but that doesn't matter. The upshot is that

the stack point has been decremented twice and now has the value 157476. This was achieved with a single instruction, which is pretty cool.

```
157504 010705
157506 062705
157510 000112
```

The absolute loader's execution involves quite a lot of reading bytes from tape, so these next instructions set up the register R5 with the memory address of the read byte subroutine.

The first instruction MOVs (octal "01") the value from the source operand (octal "07"), which is the program counter (register 7) to the destination operand (octal "05"), which is register R5. When this instruction is executed the value 157506 will be moved into R5.

The next two words ADD (octal "06") the value of the source operand (octal "27"), which is the immediate value in the subsequent word (i.e. octal 112) to the destination operand (octal "05"), which is register R5. When this instruction completes, the value in R5 will be 157620, which is the address of the subroutine that loads a byte from paper tape. I'll describe how this subroutine works below.

```
157512 005001
```

This instruction clears the value of register R1.

The absolute loader will load blocks of instructions from the paper tape into locations specified by the load address in the header of each block. However, the operator can specify with the panel switches an offset address, or base address, to be added to the load address specified in each block. The remainder of the setup code is concerned with configuring this offset address. Ultimately, the offset address calculated by the code is stored in the memory address pointed to by the stack pointer.

```
157514 013716
157516 177570
```

Firstly, the value of the Control Switches and Display word is read and stored in the address pointed to by the stack pointer. This is achieved with a MOV instruction (octal "01") from the absolute address 177570 (the memory address of the Control Switches and Display word) into the address stored in R6, the stack pointer.

```
157520 006016
157522 103402
```

The next instruction rotates the content of the memory address pointed to by SP to the right (octal "0060" represents the ROR instruction). The lowest bit of the memory address pointed to by SP will be stored in the carry bit.

Since the offset address must be even, the lowest bit cannot form part of the address and it is therefore used as a flag to indicate whether or not to use an offset address. If the lowest bit is not set, then the offset address is not used. If the lowest bit is set, the offset address is used.

The second instruction tests the status of the carry bit (using a BCS instruction, represented by octal "1034") and if the carry bit was set it jumps ahead to address PC+2. If the carry bit is not set the branch does nothing.

```
157524 005016
157526 000403
```

If the carry bit is not set, we do not use an address offset and these two instructions are executed.

First we clear the value in the memory address pointed to by SP (octal "0050" represents CLR) and then branch ahead by three instructions to address 157536, which is the beginning of the instructions to read a block from tape.

```
157530 006316
157532 001001
157534 010116
```

Otherwise, the carry bit was set, in which case we are using an address offset and these three instructions are executed.

Remember we rotated the value pointed to by SP to the right to test whether the low bit was set? The first thing we need to do is restore the address offset, and this is done with an arithmetic shift left, or ASL instruction (represented by octal "0063"). This is performed on the content of the memory location pointed to by SP. A zero bit will be moved into the lowest order bit, ensuring that the address offset value is even.

Next there is a branch to determine whether the resulting value is zero. If the resulting value is not zero, control jumps ahead by one instruction (to address 157536) to the code that will begin

reading a block from the tape. This is achieved with the BNE instruction, represented by octal 001.

Finally, in case the value in the memory address pointed to by SP is zero, the value of R1 is moved into this address and execution continues through to the code that reads a block from tape. The only situation I can find where this code may be executed is where a jump block has been read but it has an odd address in it. In that case the code halts and if the operator presses continue then control branches back up to this setup code again, to the instruction that reads the Control Switch and Display word again.

In that situation R1 will contain the next address into which a byte read from tape should have been loaded, so it seems that this branch is used to continue loading when an invalid jump block has been read in. I suspect this technique may have been used in situations where a single program was being loaded from multiple paper tapes.

The instructions to read a block from tape

The next section of the absolute loader reads a block from tape. The section is really the core of the program.

```
;seek to the beginning of a block then read block header
157536 005000
157540 004715
157542 105303
157544 001374
157546 004715
157550 004767
157552 000074
157554 010402
157556 162702
157560 000004
157562 022702
157564 000002
157566 001441
157570 004767
157572 000054
157574 061604
157576 010401

;read in the rest of the block
157600 004715
157602 002004
157604 105700
157606 001753
157610 000000
```

```
157612 000751
157614 110321
157616 000770
```

The first instruction of this section clears the value in register R0:

```
157536 005000
```

Register R0 is used to hold the running checksum as the block is being read in.

```
157540 004715
157542 105303
157544 001374
```

It is possible that there may be leader bytes on the tape, for example some zero bytes before the first block starts. Therefore, these instructions reads bytes from the tape and loop until a byte containing 001 is identified, which is the first byte of the block header.

The first instruction reads a byte from the tape. This is performed by performing a JMP (octal "004") to the address contained in the register R5 (represented by octal "15), while storing the return address in register 7 (octal "7"). As mentioned earlier, the register R5 contains the address of the read byte subroutine. The resulting byte is stored in register R3.

The next instruction decrements the byte (octal "1053") contained in register R3. If the value in R3 was 1, then this decrement will result in a zero value in R3. For any other value, the result of the decrement will be non-zero.

The final instruction branches if the result of the decrement was not zero to address 157536 to repeat the process and read in another byte. When the value returned from the read was 1, the code continues:

```
157546 004715
```

This is another JMP PC, (R5) instruction, which reads another byte from the tape. This is the second byte of the block header, which is always zero, so this byte is read and then ignored.

```
017550 004767
017552 000074
```

The next two words are a another JMP instruction, storing the return address in register 7, as represented by octal 0047. The jump address is given as an offset of 74 from the current value of

the program counter. This jumps to the subroutine to read and assemble a word from the paper tape. The result is stored in register R4.

```
157554 010402
```

This instruction moves the result reading the word, contained in R4, into R2.

```
157556 162702
157560 000004
```

The next instruction subtracts 4 from the value held in register R2. The SUB instruction is represented by the octal 16. The value to subtract is found in the next word, as reflected by bhe source operand of 27. This value is to be subtracted from the destination operand, which is the value in register R2.

The four bytes being subtracted from the byte count represent the first four bytes of the block; the 001 byte, the 000 byte and the word containing the byte count.

```
157562 022702
157564 000002
```

The next instruction checks whether the remaining byte count, as contained in R2, is 2.

The octal value "02" is the CMP instruction. The source operand is found in the next word, as reflected by the source operand of 27. This is compared to the destination operand, register R2.

```
157566 001441
```

The next instruction is a BEQ, branching if the CMP instruction determines that the two operands are equal (strictly speaking, if the difference between the two operands equals zero). If the operands are equal, this means that the block has a size of six and therefore that the load address represents the jump address to start the application.

If the block has a size of six, the code branches to PC+41. This jumps to the subroutine to jump to the code just read from the tape.

```
157570 004767
157572 000054
```

The next two words are a another JMP instruction, storing the return address in register 7, as represented by octal 0047. The jump address is given as an offset of 54 from the current value of

the program counter. This jumps to the subroutine to read and assemble a word from the paper tape. The result is stored in register R4.

```
157574 061604
```

Next, any relocation offset from the setup phase of the absolute loader is added to the value in register R4. The ADD instruction is represented by the octal value "06". The source operand is represented by the value "16".

The "1" means register deferred addressing and the "6" means register R6. Register deferred addressing means that R6 does not contain the operand, rather it contains the memory address which contains the operand. R6 is, of course, the stack pointer. Therefore, the source operand is the value pointed to by the stack pointer.

The destination operand is represented by the value "04". The "0" means register addressing and the "4" means register R4. Register addressing means that the operand is the register itself, in this case R4.

```
157576 010401
```

This instruction moves the value from register R4 into register R1. So now, R1 contains the starting destination address for the bytes to be read from the remainder of the block.

```
157600 004715
```

This is another JMP PC, (R5) instruction, which reads another byte from the tape. The resulting byte is stored in register R3.

```
157602 002004
```

The last instruction of the subroutine that reads a byte from paper tape is to decrement R2, which contains the number of bytes remaining to be read in the block. This instruction tests whether the value in R2 has been decremented to zero. If it has not, in other words there are more bytes remaining, then code jumps forward to PC+4 (i.e. to address 157614).

Otherwise there are no bytes remaining, so the code moves on to validation of the block checksum.

```
157604 105700
157606 001753
```

The first of these two instructions tests the value in R0 with a TSTB instruction (represented by octal "1057"). This will set the flags in the PSW based on the value found in R0.

The second instruction is a BEQ instruction (given by an octal base value of "0014"), plus an offset. The offset in this case is 353, which is the 2's complement representation of -25 octal.

Together this means if the value in R0 is zero, which indicates that the checksum for the block was valid, then the code loops back to address 157536, which is the address of the beginning of the set of instructions to read a block from. This will begin reading in the next block from paper tape.

```
157614 110321
```

This is the instruction that actually stores the byte read from paper tape into its destination address in memory.

This instruction moves the byte contained in register R3 to the address specified in R1. The value in R1 will be auto-incremented after the value of R3 has been stored.

Note that this is a byte instruction (MOVB, given by octal "11") as opposed to a word instruction (MOV, which would be given by octal "01"). Therefore when R1 is being auto-incremented, it will be incremented by 1 representing one byte increase as opposed to 2 in the case of a MOV instruction, which would represent 1 word increase.

This auto-increment means that the register R1 now contains the value into which the next byte is to be stored when it is read from paper tape.

```
157616 000770
```

The final instruction in this section, having stored the byte at the relevant location in memory, branches back to read in another byte.

This is an unconditional branch instruction, given by the base octal value of "000400". The offset is 370, which is the 2's complement representation of the negative value -10 in octal. This will therefore loop back 10 word instructions to address 157600 to read and process another byte.

The remaining sections of the absolute loader are a set of supporting routines that help with different aspects of the code just described.

The instructions to read a byte from tape

This block of instructions is the subroutine to read a byte from the paper tape. Here it is:

```
;subroutine to read a byte
157620 106703
157622 000152
157624 105213
157626 105713
157630 100376
157632 116303
157634 000002
157636 060300
157640 042703
157642 177400
157644 005302
157646 000207
```

This code is quite similar to the bootstrap loader, so it should make a lot of sense as we go through it.

```
157620 016703
157622 000152
```

This instruction moves the address of the CSW of the paper tape reader into R3. The MOV instruction is represented by octal "01". The source operand, represented by octal 67, is the value at an address specified by the offset in the next byte (octal 152) added to the program counter, giving an address of 157776, which is the final address of the original bootstrap loader. This value is moved into R3.

```
157624 105213
```

This instruction increments the byte at the address pointed to by the value in R3. In other words, it sets the low bit of the CSW, which enables the paper tape reader.

```
157626 105713
157630 100376
```

These next two instructions loop until the paper tape reader indicates that there is a byte ready by setting the DONE bit.

The first instruction tests the low byte of the address contained in R3 using the TSTB instruction. The second instruction branches back to the TSTB instruction if the value is positive using a BPL instruction.

The DONE bit is bit 7 of the low byte of the CSW. When this value is set, this represents a negative value, so when the DONE bit is set, the value will not be positive and control will pass through the branch.

```
157632 116303
157634 000002
```

When the paper tape reader sets the DONE bit, these next instructions read the byte value from the paper tape reader buffer register (CSW + 2) and store it in register R3.

```
157636 060300
```

The value just read in and stored in R3 is added to R0 by this instruction. R0 is used to accumulate the checksum by adding each byte read to the running value in R0.

```
157640 042703
157642 177400
```

This next instruction clears all of the bits of R0 except those in the lowest 8 bits, representing the byte just read in from the paper tape reader.

This is achieved through the use of the BIC, or bit clear instruction with the mask value of 177400 (which in binary is 1111 1111 0000 0000).

```
157644 005302
```

Register R2, which represents the number of bytes remaining to be read in the current block, is decremented.

```
157646 000207
```

The last instruction returns from the subroutine, restoring the PC from the value stored on the stack.

The instructions to assemble a word

The next subroutine reads in two bytes and assembles them into a word. Here it is:

```
;subroutine to read a word
157650 012667
157652 000046
157654 004715
157656 010304
157660 004715
157662 000303
157664 050304
157666 016707
157670 000030
```

The first instruction saves the return address to a temporary location:

```
157650 012667
157652 000046
```

When a subroutine is invoked using JSR, the stack pointer will be decremented and the current program counter will be pushed onto the stack. Therefore, the stack pointer points at the return address, which is the value of the program counter to be returned to when this subroutine is finished.

This instruction (which is "MOV (SP)+ @46(PC)") reads the value at the stack pointer (which is the return address) and then increments the stack pointer. The value is then stored at a temporary location which is PC+46 (i.e. memory address 157722).

```
157654 004715
```

This is another JMP PC, (R5) instruction, which reads another byte from the tape. The resulting byte is stored in register R3. This will be the low order byte of the word that is being assembled by this subroutine.

```
157656 010304
```

This instruction moves the value in R3 into R4.

```
157654 004715
```

We then execute another JMP PC, (R5) instruction, which reads another byte from the tape. The resulting byte is stored in register R3. This will be the high order byte of the word that is being assembled by this subroutine.

```
157662 000303
```

This is a SWAB R3 instruction, meaning that the bytes in the register R3 will be swapped, making the low order byte the high order byte, and vice versa. This converts the byte just read, which would have been stored in the low order byte of R3 into a high order value.

```
157664 050304
```

This is a BIS R3, R4 instruction. This will set the bits in R4 that are set in R3. In other words, this has the effect of ORing R3 (the low order byte) and R4 (the high order byte). The resulting value is stored in R4.

```
157666 016707
157670 000030
```

Finally, this instruction moves the PC value that was stored in a temporary value at the beginning of this subroutine (located at PC+30) into the program counter. This returns from the subroutine.

The instructions to jump to the code just read in from tape

The last subroutine in the absolute loader is used to check, and then jump to, the address specified in a jump block (a block with a length of six). This will execute the code that has been read in from paper tape. The jump address must be even, otherwise the execution of the absolute loader will halt. Here is the code:

```
;subroutine to check jump address and jump to the loaded program
157672 004767
157674 177752
157676 004715
157700 105700
157702 001342
157704 006204
157706 103002
157710 000000
157712 000700
157714 006304
157716 061604
157720 000114
157722 000000
```

Recall that this code is only executed when the block loading code above determines that the block has a length of six. Four of the six bytes (the two header bytes and the byte count word) have already been read by the block loading code.

Therefore, the first thing this code does is read a word is read from the paper tape - this is the jump address.

```
157672 004767
157674 177752
```

This is achieved by JMPing to the adddress PC-26. The -26 is specified by the word value 177752, which is the 2's complement representation of that value. The result is stored in R4.

```
157700 105700
```

Next, one more byte is read. This is achieved using a JMP PC, (R5) instruction, which reads another byte from the tape. The resulting byte is stored in register R3. This is the checksum byte, which is not included in the byte count value.

```
157700 105700
```

As bytes are being read from tape, the running checksum value is stored in R0. When the checksum byte has been read in, if there are no checksum errors, the value in R0 should be zero. This instruction, therefore checks the value of R0 using a TSTB instruction. This will set the value of the flags in the PSW based on the value in R0.

```
157702 001342
```

If the value in R0 is non-zero this means there has been a checksum error, so this instruction is "BNE @-36PC". In other words, branch if the zero bit in the PSW is not equal to zero to address PC-36. This branches to the HALT instruction at address 157610, which represents the halt state arising from a checksum error.

```
157704 006204
157706 103002
157710 000000
157712 000700
```

At this point we have read the jump address and we have confirmed that there is not a checksum error. The next step is to test whether or not the jump address is even. If it is not, the absolute loader should halt. This is achieved by these next instructions.

The first instruction performs an ASR (octal "0062"), which means arithmetic shift right, on the value in register R4. This shifts all bits in the register one location to the right. The lowest bit in the register is moved into the carry flag.

If the number originally stored in register R4 was even, then the value in the carry flag should be zero. If the number originally stored in register R4 was odd, then the value in the carry flag should be one.

We therefore test the carry flag with the next instruction, "BCC 2(PC)". This instruction says branch if the carry bit is clear (i.e. if the value in R4 was even) to PC+2. In other words skip the next two instructions and carry on (remember PC already points at address 157710 as this instruction is executing).

If the carry flag was set, this branch will do nothing and the execution carries on to the next instruction with is HALT (octal "000000"). If the operator presses continue, control jumps back to the setup code (address 157536) by way of the branch instruction (octal "000700").

```
157714 006304
157716 061604
157720 000114
```

At this point we know that the jump address is valid because it passed the checksum test and the evenness test. Therefore all that remains is to jump to that address.

Remember, however that we shifted R4 to the right to test whether the value was odd or even. Therefore, we now need to shift R4 to the left to restore the value.

The first instruction (octal "0063") with the operand ("04") means "ASL R4", or arithmetic shift left of the value in R4. This will shift all bits in R4 one position to the left and add a zero at the lowest bit position. The adding a zero at the lowest bit position is fine because we know at this point that the jump address is even, which means a zero belongs in the lowest bit position anyway.

Next, we add the address in the location pointed to by the stack pointer to R4. This adds any address offset configured by the operator to R4. This is achieved by the octal "061604" which represents the instruction "ADD (SP), R4".

Lastly, we jump to the memory location contained in the register R4, which is achieved by the instruction "000114", or JMP ("0001") to the register deferred value in R4, in other words to the location pointed to by R4. This will begin executing the code that has been read in from the paper tape.

```
157722 000000
```

This last word is used as a temporary storage location by the subroutine that reads in and assembles a word. It is used to store the return address for use after the subroutine completes.

And that's it! The remaining code relates to the restoration of the bootstrap loader, and that was discussed in the previous post on loading the absolute loader.