|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | How Does The First Stage Loader Work? |  | **Altair Bootloaders** | | |
| |  | | --- | | *I spent quite some time figuring out how the Altair paper tape loading process works when I brought up Basic on my* [*Altairkit*](http://www.altairkit.com/) *8800. Hopefully, these pages will give a head start to anyone else who wants to know how it works.* | |
|  |
| |  | | --- | | Top of Form    Bottom of Form | |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **What Does the First Stage Bootloader do?**  MITS published several versions of the first stage bootloader, each one customized for the kind of program it was loading — 4k Basic, 8k Basic, etc. — as well as for the I/O card being used to load the program. The one we looked at earlier was for 4k Basic 3.2 running on port 1 of an 88-2SIO card.  They are all basically the same code with minor differences to handle the different hardware and to load the second stage bootloader into different locations in memory.  Here is the assembly language version of the bootloader we used earlier.   |  | | --- | | init: mvi A, 003 ; Reset the 2SIO port 1  out 020  mvi A, 025 ; 021 = 2 stop bits, 025 = 1 stop bit  out 020  ; Initialize HL to 0faeH for 4k Basic,  ; 1faeH for 8k Basic, 2faeH for Extended Basic  lxi H, 0FAEH  loop: lxi SP, stack ; initialize the stack pointer  in 020 ; Wait for a character to arrive  rrc  rnc  in 021 ; get the character  cmp L ; Is it the same as the value in L?  rz ; yes - ignore it and loop back to get another char  dcr L ; no - decrement the counter in L  mov M, A ; store the received character in RAM @ HL  rnz ; loop to get another until L = 0  pchl ; jump to the start of the program we just read.  stack: dw loop ; prime the stack with the  ; address of the top of the loop. |   The job of the first stage loader is to read data from a paper tape into memory using the smallest number of bytes possible so that you can easily enter it on the front panel switches. Consequently, it uses some odd techinques to minimize the size of the program, and it's very dumb about what it does. It does no error checking at all, it just wants to load the smarter second stage loader (called the checksum loader in the MITS manuals) and hand off to it as soon as possible.  Let's look at the code.   |  | | --- | | init: mvi A, 003 ; Reset the 2SIO port 1  out 020  mvi A, 025 ; 021 = 2 stop bits, 025 = 1 stop bit  out 020 |   This is straightforward and simply initializes port 1 on the 2SIO card.     |  | | --- | | ; Initialize HL to 0faeH for 4k Basic,  ; 1faeH for 8k Basic, 2faeH for Extended Basic  lxi H, 0FAEH |   Here we initialize the HL register pair to point to the location where we're going to store the second stage loader. The 2nd stage loader is stored in reverse on the tape, so HL starts at the last byte of the storage area and is decremented by one for each byte read. When L reaches 0 we know we've read the entire 2nd stage loader.     |  | | --- | | loop: lxi SP, stack ; initialize the stack pointer  in 020 ; Wait for a character to arrive  rrc  rnc |   This is the top of the read loop. The first thing we do is initialize the stack pointer to point to a location at the end of the program. That location is already primed with the address of the start of the loop so that all we have to do to go back to the top of the loop is execute a return instruction. One byte instead of the three it would take for a jump. Every time we get back to the beginning of the loop, we reinitialize the stack pointer to that we can get back here again.  This first part of the loop just monitors the status register of the 2SIO card to see if there is an incoming character ready to be read. When one is available, we fall through to the next part of the loop to pick it up.     |  | | --- | | in 021 ; get the character  cmp L ; Is it the same as the value in L?  rz ; yes - ignore it and loop back to get another char  dcr L ; no - decrement the counter in L  mov M, A ; store the received character in RAM @ HL  rnz ; loop to get another until L = 0 |   How does the 1st stage loader know whether the incoming character is part of the 2nd stage loader or just random noise from the beginning of the tape? Ahead of the 2nd stage loader on the tape there is a string of bytes called the lead in string. Each byte in the lead in string is a hex AE (octal 256), the same value that L is initialized to, so we just keep reading bytes until we come to the first one that doesn't match L, then we start storing bytes and decrementing L until L = 0.  Prior to Basic V3.2, a different second stage loader was used that was slightly smaller than this one. The lead in string for the earlier loader consisted of bytes containing hex 7D (octal 175). If you wanted to read one of those earlier tapes, you would have to change the initialization value for HL to something like hex 0F7D. There was also a later loader that came out around V4.0 that had a lead in string of hex C2 (octal 302).     |  | | --- | | pchl ; jump to the start of the program we just read. |   Finally, when L reaches 0, we have read the entire 2nd stage loader and HL is pointing at the first byte of the program. We fall through the RNZ instruction to this one which swaps the value of HL and PC and execution continues with the first byte of the second stage loader. | | |  |  |  | | --- | --- | --- | | [http://www.solivant.com/icons/left.gif](http://www.solivant.com/altair_bootloaders/index.php?album=altair_bootloaders&pagen=1) | 3 of 5 | [http://www.solivant.com/icons/right.gif](http://www.solivant.com/altair_bootloaders/index.php?album=altair_bootloaders&pagen=3) | | | [[Creative Commons License](http://creativecommons.org/licenses/by-nc/3.0/us/) [http://www.solivant.com/common/sig.png](http://creativecommons.org/licenses/by-nc/3.0/us/)](http://creativecommons.org/licenses/by-nc/3.0/us/) | |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | The Second Stage Loader |  | **Altair Bootloaders** | | |
| |  | | --- | | *I spent quite some time figuring out how the Altair paper tape loading process works when I brought up Basic on my* [*Altairkit*](http://www.altairkit.com/) *8800. Hopefully, these pages will give a head start to anyone else who wants to know how it works.* | |
|  |
| |  | | --- | | Top of Form    Bottom of Form | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | When the first stage loader has finished loading the second stage loader, it hands over control to that program. The second stage loader first checks the front panel switches to see what I/O device it should use, then starts reading bytes from that device looking for the main payload, in this case the Basic 3.2 executable. The payload is stored in a series of data blocks with checksums which allow the loader to verify that it is reading the tape correctly.  The possible types of data blocks are:   |  | | --- | | <begin/name record>  <prog load record>  <EOF record> |     **The Optional Begin/Name Record.**  The begin/name record block is optional, and, in fact, I haven't found any tapes that include it and the second stage loaders that I have seen don't even try to process it they just skip over it. If it exists, it begins with a byte containing the value octal 125, followed by a 3 byte program name field, followed by a variable length comment field and is terminated by an octal 015 byte.    **The Program Load Record.**  The program load record blocks are more interesting and contain the tape payload. There will be one or more prog load record blocks, each one containing up to 255 bytes of data. The format of a prog load record is:   |  | | --- | | 074 - sync byte (3Ch)  NNN - # bytes in record  lll - low byte of load address  hhh - high byte of load address  <NNN bytes> - NNN bytes of program data  ccc - checksum byte (sum includes lll & hhh) |   The block begins with an octal 074, followed by a 1 byte length field, and a 2 byte address in memory where this block should be stored. Next comes a variable number of bytes of payload data, the number of bytes for this block were specified in the length field. The block ends with a 1 byte checksum, which is simply the sum, truncated to 1 byte, of all the bytes of data in the block including the low and high address bytes.    **The EOF Record.**  After all the program load records there will be one EOF record consisting of an octal 170 followed by the two byte starting address of the program, which can differ from the first load address.   |  | | --- | | 170 - sync byte (78h)  lll - low byte of start address  hhh - high byte of start address |     **Error Handling.**  The second stage loader has no control over the paper tape being fed to it, so it can't recover from any errors it encounters, but it can warn you if it finds any. It verifies the checksum for each program load record and sends a stream of **C** characters to the terminal if it finds a mismatch. If it has problems storing any bytes in memory due to a memory error or simply running out of memory, it will send a stream of **M** characters. If it detects that it is being overwritten by the incoming program, it sends a stream of **O** characters    **Front Panel Switches.**  The second stage loader is designed to handle a range of I/O devices, so you have to set switches on the front panel to tell it how to read the tape. Bootloaders prior to Basic 3.2 used different switch settings than the newer loaders. I'll talk about only the 3.2 loader settings here. If you need to know the switch settings for an earlier loader and don't have access to the appropriate MITS BASIC manual, let me know and I'll add that information to this page.  Before you start the first stage loader running, you have to set the address switches to tell the second stage loader and the Basic program that it loads what device you are using to load the tape and what device your terminal is on.   |  |  |  | | --- | --- | --- | | **Terminal Device** | **Switches Up** | **Default Channel in octal** | | SIOA,B,C (not Rev 0) | None | 0,1 | | SIOA,B,C (Rev 0) | A14 | 0,1 | | 88-PIO | A13 | 0,1 | | 4IO | A12 | 20,21 (input) 22,23 (output) | | 2SIO | A11 A10 (up=1 stop bit, down=2 stop bits) | 20,21 | | | |  |  |  | | --- | --- | --- | | [http://www.solivant.com/icons/left.gif](http://www.solivant.com/altair_bootloaders/index.php?album=altair_bootloaders&pagen=2) | 4 of 5 | [http://www.solivant.com/icons/right.gif](http://www.solivant.com/altair_bootloaders/index.php?album=altair_bootloaders&pagen=4) | | | [[Creative Commons License](http://creativecommons.org/licenses/by-nc/3.0/us/) [http://www.solivant.com/common/sig.png](http://creativecommons.org/licenses/by-nc/3.0/us/)](http://creativecommons.org/licenses/by-nc/3.0/us/) | |