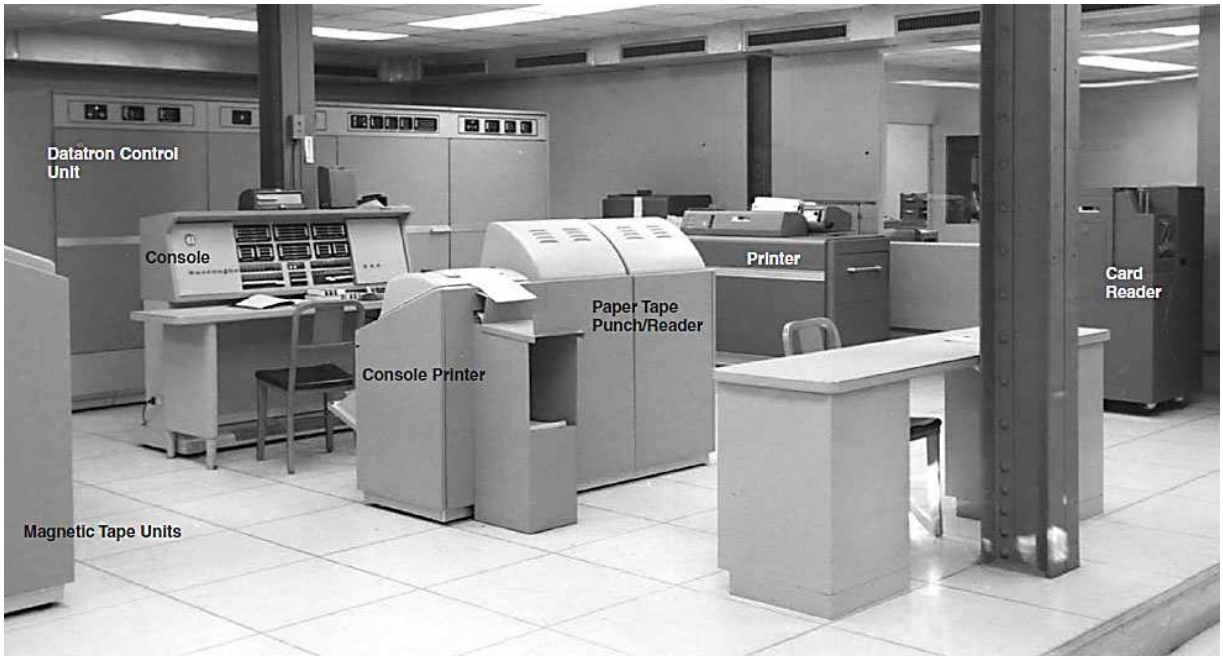
Simulating the Burroughs B220 on an Apple //e

Michael J. Mahon



What's a B220?

The Burroughs 220 was the last commercial vacuum tube computer. It was the successor to the B205, which used a magnetic drum for its main memory. The B220 provided a needed technology update by adopting the then-new magnetic core random-access memory while retaining the existing vacuum tube logic modules.

It's likely that you've already seen the console of a B220 with its prodigious quantity of neon lights, since it has often been cast as "the computer" in many TV shows and movies—with all its neon lights replaced with more photogenic incandescent lights.

Its maximum memory size was 10,000 words, each having ten 4-bit BCD data digits and a BCD sign digit. In the pre-ASCII, pre-lower case days, each machine encoded alphanumeric characters differently, with six bits being a common choice for binary machines, and two BCD digits for decimal machines. Each signed, ten-digit B220 word could hold five alphanumeric characters, indicated by a sign digit value of 2. Positive numbers had an even sign digit, usually 0, and negative numbers had an odd sign digit, usually 1. Other sign values were used for control purposes, such as flagging instructions whose addresses should be relocated upon loading.

S	1	2	3	4	5	6	7	8	9	0

B220 Word Format

The B220 was intended both for commercial markets, with its BCD data representation, and for scientific and engineering uses, with its hardware-supported floating-point arithmetic. Like its B205 predecessor, the B220 had an index register, the B register, named after the "B box" of

the Manchester Mark I computer. Index registers were still quite new and greatly simplified operations on tables, arrays of data, and linked lists, and also enabled the B220 to relocate library programs on-the-fly as they were loaded.

B220 input/output devices were paper tape readers and punches, printers, card readers and punches, and magnetic tape units. It could also use a novel magnetic tape device containing 50 loops of magnetic tape, each 50 feet long--an early mass-storage device with significantly lower latency than a magnetic tape.

I've included scans of three handy books that describe the B220 and its instruction set and peripherals in great detail for those who would like to learn more, and perhaps even write some B220 programs to get a feel for it. The first, **B220 Operational Characteristics**, provides a detailed overview of the system and each instruction. The second, **An Introduction to Coding the Burroughs 220**, covers the instruction set from the point of view of a programmer. The third, **B220 Operating Procedures**, is a detailed guide to using the system, with special attention to the various I/O devices.

Why a B220 Simulator?

The B220 was one of the first computers that I programmed. In 1963 it had already been superseded as Caltech's primary computer system, by an IBM 7090. When the campus computing load moved from the B220, it became a "research computer" for which students and faculty could sign up (on a blackboard) for blocks of time. Since this seemed the ideal environment for learning a machine and experimenting, I was immediately drawn to it. Though I didn't realize it, it was essentially a room-sized personal computer!

When I "met" the Caltech B220, it didn't have a symbolic assembler. It did have an Algol 58 variant compiler, known as BALGOL, and a "numeric" assembler that allowed "sections" of numeric machine language programs to be combined using section-relative addresses—neither of which provided the kind of programming experience I wanted. (Much later I learned that several B220 symbolic assemblers existed, but they were not available in the Caltech environment.)

So I came to write my first assembler. It was a crash course in scanning and table searching on this venerable machine, and I learned a lot about "wordy" computers. My "Symbolic-Numeric Assembler Program" or SNAP ("Numeric" because it also supported the older numeric assembler pseudo-ops) became pretty popular among the small group of Caltech people still programming the B220. I used it to write interactive graphics programs for visualizing and analyzing the voluminous data from nuclear physics experiments.

Sixteen years later, in 1980, I purchased my first Apple][+ computer. As I programmed it and used it, I realized that the experience I was having on a desk in my home was very similar to my experiences with the Burroughs 220—minus all the blinking lights—and I loved it!

Over the years, I thought several times about writing a simulator for the B220 on my Apple][+. There was enough memory, since the Caltech B220 had just 5000 words—about 30,000 bytes. And the 6502's BCD mode gave me hope that simulating the instructions could be reasonably efficient. The sticking point for me was the decimal address decoding.

I could only see a choice between the memory-wasting method of using the B220's BCD addresses as binary addresses, which would result in almost 40% of the memory being unused, or the extremely slow approach of converting the BCD addresses to binary on each storage

access. Then, recently, I realized how I could do it all with tables, quickly and compactly, and I began coding B220SIM on the Apple II almost immediately!

The Fetch Routine

Of course, there is another way of providing a dense BCD-to-binary address mapping that is also quite efficient: indirect mapping through tables. The natural mapping of B220 word memory to 6502 memory is to allocate 6 bytes to hold the B220's 11 BCD digits, with one wasted digit—the high order digit of the byte containing the sign digit. This mapping causes the BCD address part of each instruction to occupy two bytes, one containing the upper two digits and one containing the lower two digits.

Un-used	S	Variant				Op		Address			
-	S	1	2	3	4	5	6	7	8	9	0

B220 Instruction Format

In this scheme, converting a 4-digit B220 BCD address to the binary address of a 6-byte "word" requires:

1. Checking the B220 address for invalid BCD digits,
2. Converting the B220 address to binary,
3. Multiplying by 6 (bytes per B220 word), and
4. Adding the base address of simulated B220 memory.

That seems like a pretty tall order to do quickly on a 6502, but it turns out that with a couple of tables, one indexed by the two low-order BCD digits and the other indexed by the two high-order digits, plus some careful arranging, the complete binary address formation can be performed by a 6502 in just 38 cycles, including range and validity checking!

The table-lookup method of converting addresses combined with a straightforward BCD opcode lookup table pretty well handles the common "instruction fetch" cycle of the B220. There's some additional logic related to "run" mode versus "single step", error halts, periodic simulated register display, and B-register address modification, but that's all pretty straightforward.

Take a look at the simulator's 'fetch' routine on pages 16-18 of the B220SIM listing to see the result.

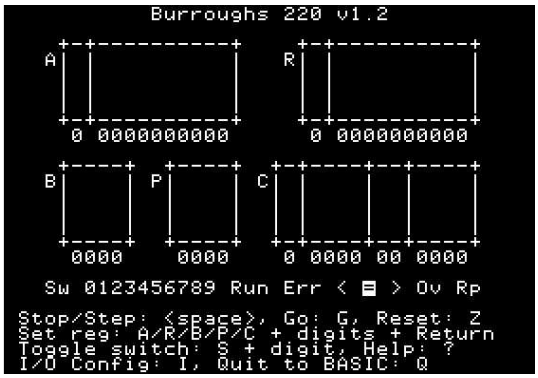
The Execute Routines

In general, there is an execute routine for each opcode. Since several B220 instructions are minor variations of each other, some routines are combined, or share significant implementation code. It's a good thing that some combination is possible, because with ProDOS and BASIC.SYSTEM resident there is less than 4KB left for the execute routines after allocating 30K of simulated B220 memory and about 2KB for the fetch, display, and human interface routines for the simulator.

Initially, I only planned a “skeleton” implementation—just enough instructions to convince myself that it would work. But, like many other projects, it took on a life of its own, and I set about fleshing it out with, first, all basic instructions without the floating-point instructions, then, all but the mag-tape instructions, and currently, all but the least-used mag-tape instructions and the Cardatron instructions. The latest version, 1.2, includes an I/O configuration screen to associate devices with named ProDOS files.

You can see all the instruction execute routines in the B220SIM listing starting at page 32.

Actual B220 panel (simulated portions highlighted)



B220 simulator panel

Implementation

Although the B220 is a decimal, word-oriented architecture, many of its instructions permit operations on “partial fields” of words—contiguous groups of digits treated as unsigned integers. Such fields are specified by two decimal digits, one of which designates the low-order “start” digit of the field, from digit 1 to 10 (represented as 0), and one of which is the length of the field extending to the left, again from 1 to 10, but never extending beyond the sign digit of the word.

S	1	2	3	4	5	6	7	8	9	0

B220 Word Format (04 field highlighted)

The fact that the simulator groups digits two to a byte implies that the “even numbered” digits are low-order digits in a byte, while the odd numbered digits are high-order in a byte. Since partial fields can start and end at any digit, digit alignment may require a one-digit (four-bit) shift. Similar shifts are required for multiply, divide, and, of course, shifts, as well as several instructions that use a two-digit immediate in the instruction as data to operate on an arbitrary partial field.

For efficiency, it is desirable to minimize the number of partial-byte (one-digit) shifts, and to maximize full-byte (two-digit) shifts, since they do not require iterated 6502 bit shift instructions, only data movement.

For example, multiply is implemented by first making a one-digit-shifted copy of the multiplicand (x10), and then handling two digits of multiplier per iteration, with all subsequent shifting done by the byte.

Floating-Point

The B220 was unusual in providing both BCD data representation for commercial applications and floating-point arithmetic instructions for scientific applications. Floating-point add and subtract are implemented directly, while floating multiply and divide are implemented by using the fixed-point multiply and divide routines as subroutines, just as in the actual B220.

Simulator I/O

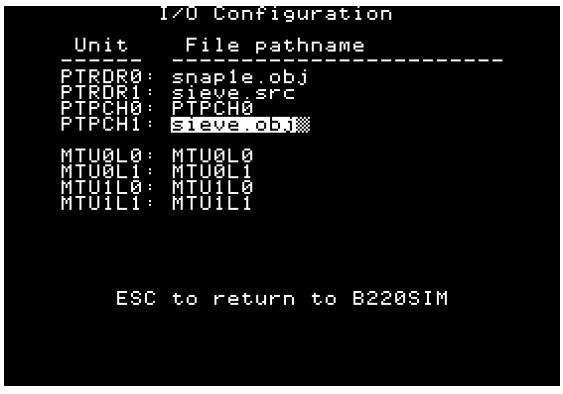
Although the B220 could attach to a pretty wide variety of I/O devices, the Caltech machine was intended for scientific applications and so had a relatively limited I/O configuration. It had one 1000 character-per-second paper tape photoreader, one 60 cps paper tape punch, one 10 cps teletype printer, and one electrostatic printer that could print about 200-300 lines per minute, depending on line length. It also had two B220 mag-tapes, which could transfer about 1900 B220 words per second. I based the simulator I/O configuration on the Caltech machine.

B220 magnetic tapes are a bit unusual. Each drive mounts one tape, but each tape can be recorded or read in two independent “lanes”—independent, that is, except for positioning, since the lanes are parallel on the tape. So, for now, consider that each tape unit is like two units in terms of capacity.

I decided to implement both paper tape and mag-tape as ProDOS binary disk files, with the data being simply the B220 memory image of the record. This design choice greatly simplified I/O handling, allowing all reads and writes to be handled by BASIC.SYSTEM commands to BSAVE and BLOAD, with the address specified by the A parameter, length by the L parameter, and the offset into the file specified by the B parameter. This approach also has the advantage that only one file is ever open at a time, reducing buffer space requirements.

I also chose to provide the simulator with an additional paper tape reader, since I had always wanted one for the boot device and one for a data reader. So B220SIM provides two paper tape readers: PTRDR0 and PTRDR1, two paper tape punches: PTPCH0 and PTPCH1, and two magnetic tapes, each with two lanes: MTU0L0, MTU0L1, MTU1L0, and MTU1L1.

Each of these units can be associated with a ProDOS file using the I/O Configuration screen. If no file name is configured, the default file name is the unit name, e.g.: “PTPCH0”, for example:



The I/O Configuration screen allows the user to change the ProDOS filename associated with each I/O unit from its default, which is the unit name. Editing is rudimentary but sufficient. The up and down arrow keys select the desired unit file name, and left arrow and delete keys delete the character to the left of the cursor. Typing appends characters. Escape returns to B220SIM.

The only B220 output instruction that results in formatted text output is SPO (Supervisory Print-Out). It prints to the 4-line "help" region at the bottom of the simulator's screen. If desired, SPO output can be redirected to a printer (or a text file if running on an Apple II emulator) by typing "PR#1" at the Applesoft BASIC prompt.

One paper tape read instruction, PRB, has unusual, but very useful, behavior. Instead of specifying a fixed number of words to read in, it reads paper tape until it encounters a specially flagged word which, instead of being stored in memory, is passed immediately to the C (instruction) register to be executed. This permits "scatter loading" of code and data, and typically finishes with a flagged branch to the start of the program, making it the B220's natural boot instruction.

Since this I/O is not count-controlled, the "safe" way to simulate it would be to read the paper tape file into a buffer and scan for flagged words, transferring only the desired data into simulated B220 memory. Rather than spend RAM on a large buffer, I chose to read the data directly into B220 memory and scan it there. This could result in as many as 100 words of memory being inadvertently overwritten, but since the "scatter load" read instruction is virtually always used only for program loading to ascending addresses, only unused memory is put at risk and no adverse effects occur. Arbitrarily scattered loading would require a more faithful implementation.

Since RAM was getting scarce in the simulator, I also took some shortcuts when implementing the rather extensive set of mag-tape instructions. To simplify things, I only implemented as much as was required to support the SNAP assembler, which means that each mag-tape record is 100 B220 words long (maximum length) regardless of the block sizes specified in Mag-tape Initial Write instructions. SNAP (and most other Caltech B220 programs) used 100-word records almost exclusively, so this was an easy compromise resulting in a much simpler, smaller mag-tape implementation.

Testing the Simulator

By the time that I'd implemented almost all the instructions, I realized that I didn't have much B220 code to run on the simulator, and, in particular, no machine diagnostic to test it. Then I remembered that I had kept listings of two versions of my SNAP assembler: SNAP 1C and SNAP 1E, versions about five months apart in 1964. So my "only problem" was getting these listings in machine-readable form so that I could, first, extract the machine code for SNAP and, second, extract the SNAP source so I could bootstrap the assembler on the simulator.

My first hope was that I could scan the listing and use OCR to recover the needed data. As any of you who have tried this approach know, OCR is not very useful for dot matrix printouts of "columnar" assembly listings. OCR was a major disappointment.

So I bit the bullet and decided (as others have before me) that my best recourse was to re-type the listing and hope to keep typos under control. I succeeded in retyping the listing over about three weeks, in both long and short bursts of effort. Surprisingly, there were only a couple dozen typos in 35 pages! But several of them were of the "oh" versus "zero" type, both in the machine code column and in the source, and finding them required patience as they revealed themselves during debugging.

I typed in the SNAP 1C listing, the only one containing both object code and source statements, as an ASCII TXT file, so I wrote an Apple II program, CVT2B220—Applesoft BASIC assisted by 6502 assembly code—to extract either the machine code portion or the source statement portion of the file and convert it to a B220 "paper tape image" for the simulator. (This required

preparatory step of converting an ASCII text file to a B220 tape image is oddly reminiscent of my Caltech ritual of converting punched card decks to B220 tapes!)

When the SNAP 1C object code was extracted and run on the simulator, after some debugging of the simulator and the "toolchain", it successfully assembled some small test cases. I then turned it loose on its own source—and it worked (again, after finding some typos in the source). When I compared the fresh object file to the extracted one, they were identical!

I now had a fully bootstrapped SNAP 1C assembler running on the simulator.

SNAP 1E

The SNAP 1E source listing (without object) only differed from the SNAP 1C source in about a hundred lines, so I decided to start with the 1C source file and edit it manually into the 1E source. That process was pretty straightforward, and, in the process, I found and fixed another handful of typos—in the **comments** this time—of both source files.

I then assembled SNAP 1E source with the SNAP 1C object code and it successfully produced fresh SNAP 1E object code, unseen since 1966, when Caltech decommissioned the B220.

I completed the bootstrap by reassembling SNAP 1E using the SNAP 1E object code, and it produced identical object code—success!

Simulator Speed

The B220 was not fast by today's standards. Its basic clock speed was 200kHz, and data was transmitted in the machine as serial streams of 4-bit digits. Fetching an instruction from memory to the C register required 90 microseconds. A fixed-point ADD executed in 185 or 245 microseconds, depending on the need to decomplement the result.

Therefore, if we could fetch and execute a B220 instruction in about 200 cycles of a 1MHz 6502, we would achieve approximate speed parity with a B220. This was an encouraging result.

Unfortunately, there are quite a few "rough edges" to be smoothed out, and the current simulator runs at approximately a third of B220 speed with a 1MHz processor. The good news is that a moderately accelerated Apple (8MHz) runs considerably **faster** than the native B220. Most Apple II emulators run at an equivalent 6502 clock speed of tens of megahertz, so B220SIM really flies under emulation!

Future Plans

The B220 Simulator has met all my short-term expectations, running SNAP 1E and my various test and demo programs. It certainly shows that an Apple II computer can effectively simulate one of the last early "mainframes". I have also unearthed my assembly listing of the B220 BALGOL compiler, which is over one hundred pages long, in a long-lost assembly language. I'm not thrilled to type it all in, but I am considering how I might reconstruct it as running code.

When the Caltech B220 was decommissioned, I managed to rescue the library mag-tape, which contained numerous relocatable library routines, plus the SNAP assembler and the BALGOL compiler and its libraries. I'm quite sure that no working B220 tape drives exist, but, in principle,

this ¾" 12-track tape could be "read" using special equipment, allowing recovery of a substantial collection of B220 code. This remains "an idea in search of a project", but it is a possibility!

Resurrecting BALGOL would entail a more complete implementation of mag-tapes, which would require more free memory. This could mean getting rid of BASIC.SYSTEM and implementing the simulator as a SYSTEM program running directly on ProDOS. Alternatively, the simulator could also use the auxiliary memory space of a 128KB Apple //e. This is another avenue that I will explore going forward. This approach would provide enough memory space that the interface to a DEC 340 CRT display, as implemented at Caltech, could be supported.

Using B220SIM

B220SIM is a standalone binary program that runs under ProDOS and BASIC.SYSTEM. It is started by typing "-B220SIM" at the Applesoft prompt.

When the simulated "panel" is displayed, the simulator is ready to run. Since its memory is initially cleared to zeroes, a B220 program must be entered before it can do anything besides execute Halt instructions (00). Programs and data may be entered manually, by placing the desired instruction or data into the A or R register and putting an appropriate Store instruction into the C register, then "stepping" one instruction by keying the space bar. This gets quite tiresome for more than a few words, so programs are usually loaded from paper tape unit 0, that is, from unit PTRDR0.

After starting B220SIM, type "I" to enter the I/O Configuration screen, where you can enter the boot file name for the PTRDR0 unit. You can also configure the file names for data input (PTRDR1) and data output (usually PTPCH1). When the I/O configuration is complete, press ESCape to return to the simulator.

To boot the simulator, enter "00000040000" into the C register (leading zeroes are unnecessary if the C register is already 0), then press "G" (Go) to start the simulator, which will load the program and immediately start running it.

At any time, you can press the space bar to stop the running program, press it again to single-step through the program, or press "G" to resume continuous execution. While the simulator is stopped, you can alter any register by typing its name ("A", "R", "B", "P", or "C"), followed by any number of digits. As each digit is typed, the register is shifted left and the new digit enters at the right. To "accept" the value in the register, press RETURN. This provides an easy way to examine and change the simulated processor state, just as the registers could be changed manually at the real B220 console.

The B220 also has 10 switch inputs that can be used to control the running program. The simulated switches are indicated by the title "Sw" and a string of the ten digits. If a digit is inverse, that switch is on, and if a digit is displayed normally, that switch is off. Any switch can be toggled by typing "S" followed by the desired digit.

If it is desired to zero the registers of the simulated B220 and "rewind" all the simulated paper and magnetic tape units, type "Z".

While the simulator is stopped, you can also type "Q" to quit to Applesoft. This can be used to allow disk commands to be typed to, for example, CATALOG files. Most BASIC commands will fail, since B220SIM takes over much of page zero. To return to the simulator with its memory intact, type "mtr", then "803g". Some registers may have been altered during the interlude, so if

any are important, like the P register, you should note its value prior to “Q” and then restore it manually after returning to B220SIM.

All of these commands are summarized in a 4-line help window below the simulated B220 panel. These lines are also where Supervisory Print-Out (SPO) instructions write their output during simulation. If the help lines have been scrolled off, typing “?” will re-display them.

An Example: Assembling and Running SIEVE

It may be helpful to go through the process of using B220SIM to assemble and run a small program written in SNAP called SIEVE.

The first step is to start BASIC.SYSTEM and use CVT2B220 to convert SIEVE.TXT to SIEVE.SRC, which is the B220-coded source file to be read by SNAP1E. CVT2B220 is versatile enough to extract the source file from a listing file, and it has a number of options to facilitate this. Type in “SIEVE.TXT” as the input file, and “SIEVE.SRC” as the output file. Then specify an “alpha” conversion (“2”), and skip 15 columns before converting, producing 11 B220 word records as output (again, this is the input format expected by SNAP). In a few seconds, CVT2B220 will complete, having created “SIEVE.SRC” containing the desired SIEVE source.

If you want to produce a text listing of the assembly, type “PR#1”, then start the simulator by typing “-B220SIM”. After the simulator loads, you will see the simulated panel of the B220.

To run SNAP1E to assemble the source, configure the B220 boot file (“PTRDR0”), and the SNAP1E source input file (“PTRDR1”) and object code output file (PTPCH1). Type “I” to enter the I/O Configuration screen and enter “SNAP1E.OBJ” as the file name associated with “PTRDR0”, the B220SIM boot device. Also enter “SIEVE.SRC” for PTRDR1 and “SIEVE.OBJ” for PTPCH1. Type ESCape to return to the simulator.

Enter the boot instruction into the C register by typing “C” followed by enough zeroes (six is plenty, but none are needed if the C register is zero), “4”, and four more zeroes, followed by ENTER. The C register should now be “00000040000”, which is a “Paper tape Read and Branch” (PRB) instruction used to boot the file associated with PTRDR0.

Now type “G” (Go), and SNAP1E will load and come to an “O-O” halt (C = 0 6996 00 6996). This is the normal halt before starting an assembly. To start the assembly, type “G” again, and SNAP will do a two-pass assembly, with the listing being produced in the second pass.

When the second pass completes at the “O-O” halt, type “Q” to quit the simulator. “CAT” will show that a new file, “SIEVE.OBJ” has been produced. This is the SIEVE object file.

To boot this file in B220SIM, type “I” to reconfigure PTRDR0 to “SIEVE.OBJ”, and set the C register to “00000040000” by typing “C”, followed by the digits, followed by RETURN. Then type “G” to run SIEVE. SIEVE will stop after every 100 primes unless simulated console switch 0 is turned on. Typing “S” followed by “0” will toggle switch 0. Prime numbers are written, 100 at a time, to file “MTU1L0” in internal B220 format until you get tired of running it! You can configure a non-default file name for MTU1L0 in I/O Configuration if you wish.

You can view a B220 format file by writing a short B220 program to read in a block of numbers and then print them out using the SPO opcode, which converts to ASCII text. I leave this as an exercise for the reader. ;-)

Have fun! Now, write your own B220 programs!