March 1979

CLOAD
MAGAZINE, inc.
BOX 1267
GOLETA, CA
93017
March 79

March, eh? We started this magazine with a March issue, last year. As my somewhat volatile memory serves me, we got that issue out about the first of April. Yessir, we may fall short in a few areas here and there, but we are consistant.

Announcements:

There's only one announcement this month, and that is that yours truly firmly planted his foot in his mouth last month. The cassette loader circuit modification we announced was not (and therefore is not) the one Radio Shack is using to improve level II loading. If you can't load level II tapes at all, talk to your local Radio Shack dealer for the official mod. If you like to - or at least don't terribly mind - hammering on your hardware, this is a good modification. As with all user modifications, installation of this one will void any warranty now in effect, and Radio Shack will not take too kindly to servicing it afterwards (and reasonably so).

A general remark in passing: if it works, don't fix it! Don't descend on your dealer to get the very latest circuitry unless you really need it.

This last week we fielded a request or two to explain the disassembler program on our February issue. As usual, I'd like to approach the subject from far off, so as to get a good head of steam built up by the time we hit it.

Machine code is that esoteric stuff that the Z-80 (and nobody else) finds palatable. The code is in the form of bytes of data stored in memory, either ROM-type or RAM-type. Each instruction (usually one byte long) is a pattern which tells the Z-80 chip to do something, like move the contents of one internal register to another. How does one go about writing machine code? Let's look at some history of the art of machine language programming. People started out writing it armed with a pencil, paper, and a list of a computer's operations. (An example of an operation: increment the contents of the internal register called the "accumulator" by one). Each operation in this list had a code associated with it, imaginatively called the "operation code", or "opcode". (Example: the opcode for incrementing the accumulator is 3C hex). Programming was the process of selecting operations in some appropriate order, and writing down the sequence of opcodes. The computer understood opcodes. [When the Z-80 runs across a 3C hex (00111100 binary, 60 decimal) in memory during an instruction fetch cycle, it will increment the accumulator]. These pioneers then entered their program by throwing "front panel" switches in a certain order, forcing associated memory cells to copy the information on the switches. I can remember writing a program of about a thousand steps, and entering it this way. This is called "insanity". The pioneers eventually developed a better input device (punchcards - yecch!) and I eventually scraped up enough cash to purchase a keyboard.

But I digress. Someone, back in the dim ages of computing, came up with a program called an "assembler". What it did was take mnemonic code and translate it into machine code. (Mnemonic - from a Greek word "mindful" - an example: INC A). The programmer, who had by now memorized the various operations of the computer at hand, now could submit a program in a more human form. Like a stack of punchcards - yecch! However, they could now be read by humans as well as the computer. The assembler program would read the card input and translate it into machine code.

The computer would then either execute the program (equivalent to RUNning it) or return to some other task, like putting it on punchcards in machine code form (object code). Fantastic. That's what an assembler does. It translates Swahili into Bengali. A disassembler, logically enough, translates Bengali back into Swahili.

So? Why would anyone want to disassemble machine code? One obvious reason would be to steal it. If one were to disassemble the machine code in the TRS-80's ROMs one would have some powerful software utilities (the original mnemonic code is securely locked up, I can assure you). That's not a good reason, however. In the first place, theft is not a particularly moral thing. The BASIC interpreter and all its copies and disassemblies are morally, ethically and legally the property of Tandy Corporation. In the second place, it's more trouble than it's worth. Turns out that a disassembler can never do a perfect job of disassembling machine code - there are places where it just plain gets lost. Also, and most importantly, there are these things called "comments". These are little marginal notes that the programmer puts alongside the mnemonic list to explain what's happening and why. They don't end up in the machine code - and therefore they can't be disassembled. Without them, the code is hundreds of times more difficult to understand.

So much for lifting the code to sell it. One logical reason to disassemble code is if the original is somehow lost (which happens a maximum of ONCE). Another reason (the most useful one in this case) is to learn how a program was written, much like LISTing a CLOAD magazine program. Those of you who commence looking around in the ROMs will find that the code is extremely efficient. Another reason is to debug code. All debuggers are disassemblers of a sort. Yet another reason is interfacing code. When the TRS-80 was announced, one of the first ideas for its use was for a local traffic engineer. He needed a device to collect data such as time, fuel, and distance continuously and automatically while driving through a city. This data was at that time all recorded by hand and punched onto cards (yecch!) to submit to a large program running on a large computer. The TRS-80 was the perfect solution, but level I BASIC didn't have the power we needed. This meant that the software associated with the project had to be written in machine code. The only problem was that the keyboard, screen and cassette handler areas of the BASIC interpreter program were not documented at that time (October 1977), and we had to control these items with our program. We also had to figure out the form of the level I floating-point numbers and find the location of the A(n) array in memory. So - we disassembled their level I ROMs to find these parts, using a much more powerful disassembler than the one we published. Two quick notes: We won't publish the more powerful one, it wouldn't run on the TRS-80 anyway; We won't make Radio Shack's code available, please don't ask.

Buzzword time - The mnemonic list we spoke of back there is generally called "source code", or "the source" (sorry, Mr. Michener). The machine code that is generated from the source is called the "object code" or "machine code", and the program which does the translation is the assembler. The language that the programmer writes in is called "assembly language" or "assembler", though it's common for programmers to claim they are writing a machine code program, as opposed to a BASIC or FORTRAN one. (It's just a sympathy play - sounds like more work). True machine code programming, which is still done for some very, very short programs, usually involves writing in assembly language and translating it with the old quill pen and parchment. This is (logically) called "hand assembling". One might ask if programs exist which translate a high level language into machine code, and the answer is yes, they are called "compilers". The language FORTRAN is usually treated in this manner. The terms "source code" and "object code" apply here just as in assembly. The language BASIC is usually implemented as an "interpreter". This is an arrangement where the source code (the BASIC program) is translated and executed on a line-by-line basis. There is no object code. If the program is to be distributed, it must be distributed as source code, and this is one of the prime reasons that significant programs are exceptionally hard to find in BASIC.

So much for last month's issue. There are two items for this month. The first is the turns counter values on the label. When all seemed to be going well on our duplicator, our tape cassette supplier decided that life was too boring. They slipped us a box or two of tape with a somewhat different thickness. The tape is really quite good, it's just that there is an error in the turns count on about 10 percent of the copies. On a level II TRS-80 with a CTR-41 recorder, the error starts out small and gets larger. On a level I TRS-80 with a CTR-41 recorder, the error starts out large and doesn't get much better. On a computer which uses a CTR-80, the turns count value has no meaning whatever. We are looking into both aspects of this problem. For now, the logical thing to do is write down the turns count of your recorder on the cassette label if there is much disparity.

Also on this issue, I'd like to describe "worm". This is a fill- in- the-screen activity by a very production oriented worm. When it asks for a level, that's level of difficulty. I'd recommend 0 until you get the hang of things. You get to steer the worm, and the idea is to keep from crossing over your path. When you find that you have "painted in" an area and can't move to another area, you steer in that direction and activate the machine gun ("enter" in level I, space bar in level II). I said the worm was production oriented. This will clear a channel through your previous path. You run out of ammunition after five bursts. You run out of room shortly thereafter, and receive a score for your surefooted performance. When you get embarrassed by your extremely high score, increase your level of difficulty to even things up.

To start out this month's hardware soliloquy, I'd like to recap what we've covered in the last few issues. We started this topic by discussing the concept of what would be appropriate to control, given that there is a computer around to do it with. At this time there is no specific application - everything we're covering applies to all implementations, so there is no need. We then addressed the subject of safety and stressed it rather heavily - even though the majority of CLOAD subscribers will probably never design or build a computer controller, it is important to be aware of the concept of controlled failure. Next we published a schematic diagram of a controller circuit, and described the concept of a schematic diagram. The last thing we covered last month was a short description of how the computer responds to an output instruction. In this issue I'd like to explain this last topic in a bit more detail.

The first item of explanation is an introduction to the timing diagram - the set of squiggly lines at the end of this article. The old saying of a picture being worth a thousand words is as true in the field of electronics as anywhere else. There is an instrument, called an oscilloscope, which has been designed to draw a graph of voltage versus time. The 'scope looks a great deal like a television set, and works on the same general principle, but the similarity ends there - most conspicuously in the area of price (or, a picture is worth a thousand bucks). The squiggly lines that this instrument draws are a clear view of what is going on inside a circuit - so clear that most circuit descriptions draw what are called "waveforms" - what an oscilloscope would draw if one were hooked up.

The set of lines on the last page are a set of waveforms. In this case they are the waveforms of the signals present on the expansion port when an OUT instruction is being executed. The particular OUT instruction could be either OUT 131,43 in level II BASIC or an assembly instruction such as OUT A,83H with 2B hex in the accumulator. As we mentioned, the squiggly lines represent a graph of voltage against time. The vertical position of each line represents the voltage present at that point, with hi = logical 1 = high voltage = up (don't laugh - the RS-232 standard defines up as being down). The horizontal direction of all lines represents time, and it is read from left to right as time marches on. Correction - as time zips by. Check the scale of time, remembering that one microsecond is one millionth of a second, and that one nanosecond is approximately the time it takes for light to travel from this page to your eyes.

All the lines are stacked one above the other so that one can see the time relationship between them. The clock is the equivalent of the drummer on the ancient, slave powered war galleys - it tells the ten thousand transistors inside the Z-80 when to stroke. The lower eight address lines - A0 through A7 - form the port number just after the rising edge of the first clock pulse. Before that time, they are used by another instruction and might be high or low (we signify this by the crosshatching). The address decode circuitry that is hanging out there (in our case, the 74LS30 and 74LS04) recognizes this as the pattern that it should respond to. Half a drumbeat later, the data to be transmitted appears on the data bus. After the data has had time to become stable, the Z-80 puts out two signals that the TRS-80 combines to form the $\overline{OUT}$ signal. The $\overline{OUT}$ signal is active low - it is low if and only if there is data to be output. For the entire time that $\overline{OUT}$ is active, both the address and data patterns stay put. This assures that our external circuit remains selected and can grab the data anytime this line is low. As a matter of interest, we actually grab the data during a relatively short "window" (the boxes drawn in the center of T3). For the input function, all the waveforms are similar, except that the $\overline{IN}$ line becomes active instead of the $\overline{OUT}$ line, we put the data the computer is requesting on the data bus during the $\overline{IN}$ pulse time, and the Z-80 grabs the data sometime during the same "window".

The data byte now disappears into the deep, dark recesses of the 8255. The way our circuit is set up, the (decimal) port addresses are as follows: 128 is port "A", 129 is port "B", 130 is port "C" and 131 is the control port - the pattern that we put there controls the configuration of the other 3 ports. The pattern 43 decimal in this control port sets bit 5 in port "C". Next issue we'll do something a bit more exciting.

Ralph McElroy
Publisher