DRAWING BOARD

Some reader suggestions concerning our SSAVI descrambler and something completely different.

ROBERT GROSSBLATT

Over the years, a lot of the projects we've worked on together have needed oddball decoders. I've said over and over again that my preferred solution is an EPROM. I've used EPROM's for everything from custom character generators to state detectors for weird numbers. If you've got the time and patience to work out a gates-only solution, you might improve your logical thinking skills, but it will take you a lot longer to get something working, it will make PC board layout a lot more complicated, and it will lock you into a particular design. EPROM's are more versatile because any modifications to the hardware in the design can be accommodated simply by programming some new code in the EPROM.

When you're in the middle of designing some hardware, a gates-only decoder might seem more attractive if you can't program an EPROM right then and there. But if you do a lot of hardware design, an EPROM programmer is just as essential as an oscilloscope.

I'm mentioning this because since we went through the basics of a SSAVI descrambler, I've received a lot of mail with alternatives to the EPROM decoding scheme I used to detect lines 24 and 257. Since it seems that a lot of you out there either prefer to do stuff with gates or don't have access to EPROM programmers, I'm going to pass along some of the decoders I've received.

All the decoders that were sent in are built with standard gates, so you should have no trouble getting the parts. Even though I have the greatest faith in my readers, I'd be a bit remiss if I didn't tell you that I haven't tried these circuits myself. You should experiment with them before you lock them into your decoder design.

The first one is from David Siegel of Livonia, Michigan and the schematic is shown in Fig. 1. It's a pretty slick design in that it's built with only three chips: two dual 4-input NOR gates and one dual 4-input AND gate.

The second decoder is from Chris Carson of Ottowa, Ontario. His design is a bit more complicated, but that's ok. Remember that more complexity makes a design more interesting. As you can see in Fig. 2, one nice feature is that only one pin is used for the line indicator. That can be handy if the rest of your descrambler wants the start and end of the vertical interval to be indicated on a single line.

Most of the circuits I received came from people in the northern part of the U.S. and Canada, so I can only guess that having to spend more time indoors during cold weather must have advantages. My apologies to the rest of you who sent me solutions—there are limits to the room I have here. My special thanks to both Chris and Dave for their designs—I know from my own experience that they took a lot of time to produce them.

If you come up with something interesting for video descrambling, drop me a note and I'll pass it along. Remember that there's strength in numbers. And now...

Something completely different

One of the considerations I use to choose the topics for this column is the amount of good material available on the subject. After a bit of

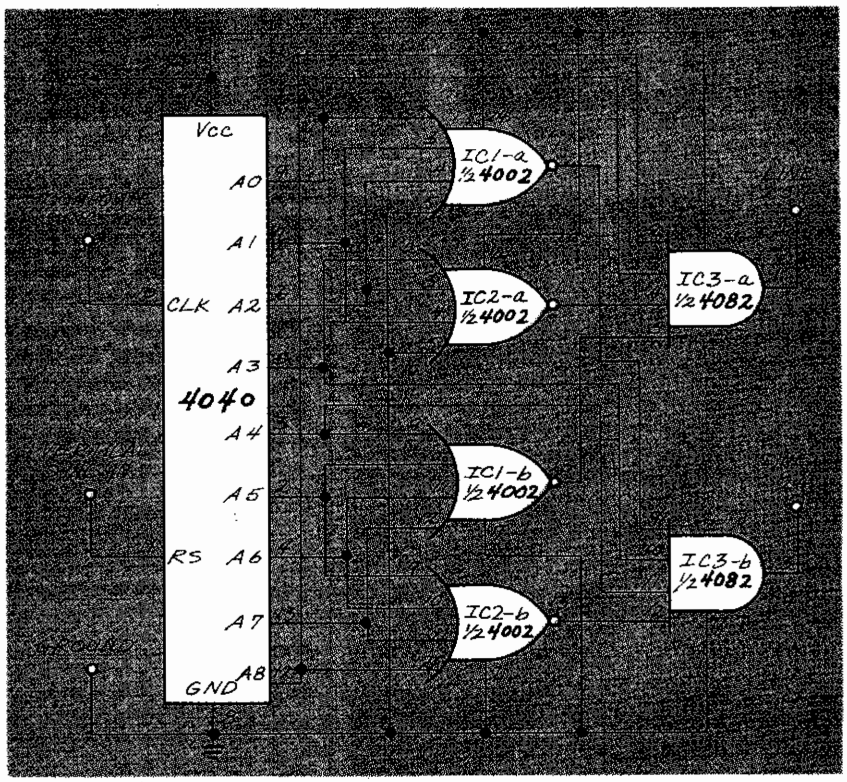

FIG. 1—MANY READERS SENT IN ALTERNATIVES to the EPROM decoding scheme I used to detect lines 24 and 257. This one uses only three chips.

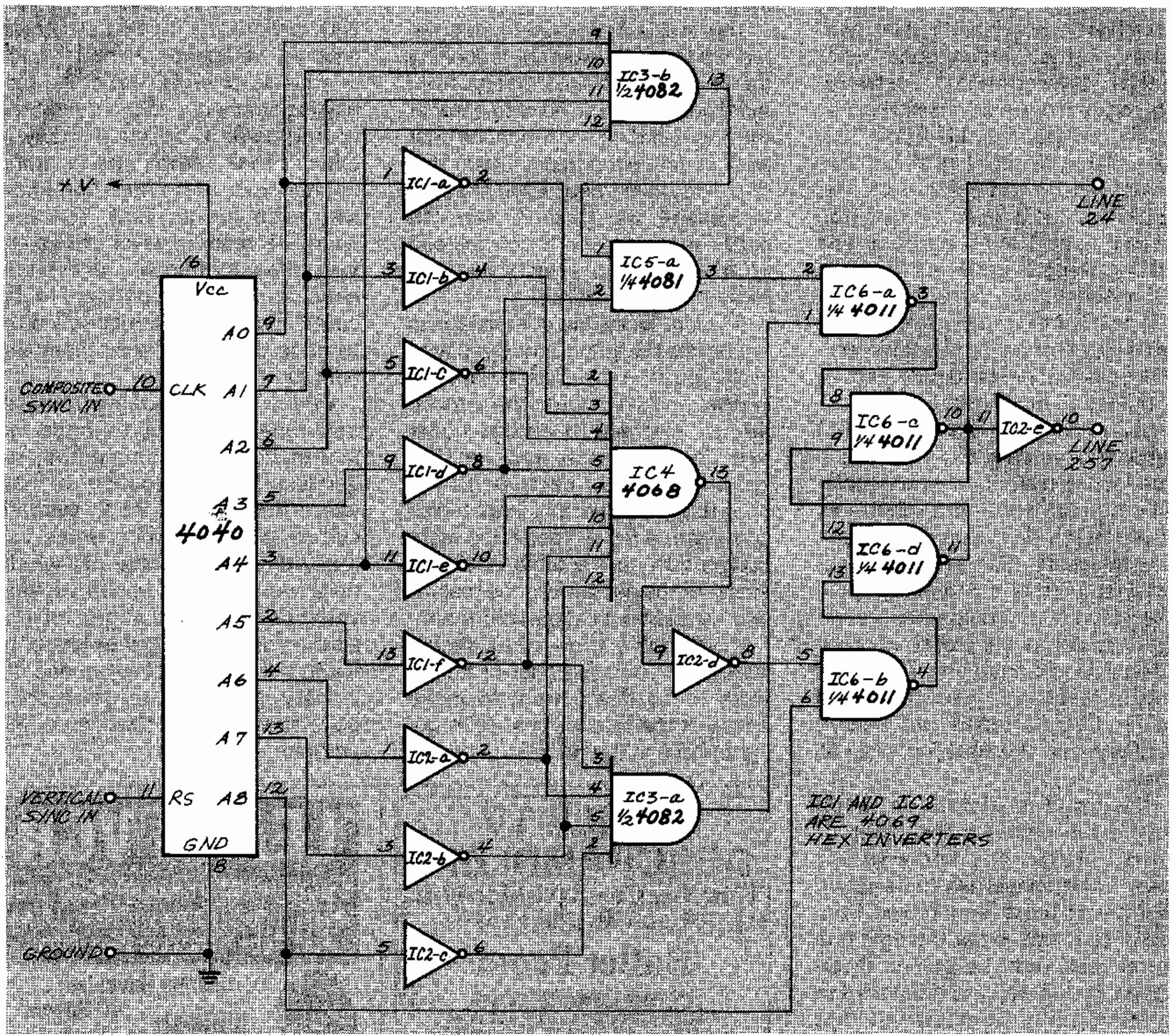

FIG. 2—THIS DESIGN IS MORE COMPLICATED, but only one pin is used for the line indicator, which is handy if the rest of your descrambler wants the start and end of the vertical interval to be indicated on a single line.

Breaking into someone else's code, especially when they've taken measures to make it as difficult as possible for you to do, means you have to have a systematic approach to the job and know exactly what you're looking for. If a program is looking for something from the disk or the keyboard, there are a limited number of instructions that can be used. The op codes are the sign posts that will help you figure out where the copy protection comes into play and how to get around it.

If you're going to be successful in removing the copy protection from a piece of software, you have to deal with the problem in a logical manner. The steps to follow are:

1. Be familiar with the software. The instructions that make up a piece of software are there to make things happen on the screen, make sounds come from the speaker, and get things from the keyboard. Even before you look at the code, you can get a good idea of how the code is structured by carefully noting the order in which things are happening.

2. Be familiar with your tools. No one, not even the person who wrote the code, can read the kind of uncommented source code produced by a debugger and follow it like the plot of a cheap novel. Comments and labels must be added to explain things normally understood only by computers. Each debugger has its own unique way of interpreting hex and deciding how it should be translated into source code. An ASCII string can be decompiled into some of the strangest source code imaginable, but if you know the behavioral quirks of your debugger, it's easier to understand.

3. Be familiar with the operating system. Programmers are free to do whatever they want as long as they limit their activity to data that's already in memory. When they want to get something from the keyboard or disk, however, there are only a few commands they can use. The 80XXX series of microprocessors are interrupt-driven. That means any searching, it occured to me that there has been virtually nothing published on the "underground" subject of "unprotecting" software. Some years ago I did a basic tutorial on the subject for the Apple but, alas, that computer has all but vanished from the face of the earth. The Apple IIgs had a lot of promise but, even though I think Apple is still making the computer, it's been all but abandoned by the software development community. And, unfortunately, I know a lot of people who shelled out an extraordinary amount of money to buy them. Oh well.

My primary interest is in the PC family and I suspect that the same is true for the majority of you people. Although software protection has pretty much disappeared in business software, it's still used extensively to protect computer games. The advent of inexpensive hard disks as well as the growth of game sizes has changed the nature of copy protection in the last several years. Original disks are always readable and the files are easy to copy.

The trend in protection these days is the infamous document, or "doc" check. You know what I mean—the manual has some pages that are columns of numbers from which you have to get the right one to proceed with the game. Those pages, by the way, are often printed in some color combination that makes it virtually impossible to duplicate them on a copying machine.

Getting rid of that kind of copy protection differs from the old methods in that you have to find the place in the code that's calling the document check and eliminate it. Since manufacturers don't provide the source code for the software, you have to work your way through pages and pages of uncommented source code and raw hex.

Before we start on this, let me warn you that our discussion you have a working knowledge of DOS and some familiarity with computer programming in general. Although there are some good debuggers around that make the job easier, don't forget that "easier" is a relative term. If you can't understand the information that's being displayed, it doesn't matter how informative it is (or isn't).

The basics of doc-check removal are simple. The program is looking for one piece of information from the manual that you have to enter at the keyboard. It then checks that against data either read in from disk or already in memory. If the check is successful, you go on to save the earth—or whatever. If not, instead of blasting into space you get blasted back to DOS.

If you're lucky, the comparison is made with the actual string you type in. All you have to do is find the table in the program with the stored strings and you're almost home free. That technique, however, isn't too common now because it's too easy to get around, and large tables can waste a lot of space. It's more common to have a table of checksums for the numbers in the manual and do the verification with them.

If you don't know what checksums are, you have a lot of homework to do if you want to make sense of the discussions we're going to have in the future.

No matter how familiar you are with programming or how much time you've spent hunting for bugs in a jungle of code, you'll find that making sense out of uncommented source code is, to put it mildly, a difficult task. When you're going through your own code, you know why each line is there. You also know the order in which things are supposed to happen, and when each routine is supposed to be called. That makes it relatively easy to trace the flow of a program and hunt for glitches.

When you're looking through someone else's source code, no matter how well commented, regardless of the language, or how logical the structure, it's always difficult to get to the point where you really understand how the whole thing works. When you're working your way through pages of unlabelled op codes, it takes longer to figure out what's going on, and it's virtually impossible to get a complete handle on every aspect of the program.