Hints for Successful Debugging

"There is always a method.... even to madness!"

adapted from Badri Roysam

The following comments apply equally to hardware and software.

Perhaps the single most important scientific principle that is relevant to the problem of debugging a system is "divide and conquer". Basically, think of a way that you can divide your system into two or more parts in such a way that you can test each little part independently of the rest. For instance, if you have an EVB (Evaluation Board) connected to a prototyping board, disconnect the two and test the EVB all by itself. Using a logic probe, oscilloscope and/or a logic analyzer, check that it is indeed putting out or correctly interpreting the signals that your hardware expects. Separately, test each part of the hardware in turn. For instance, you can easily watch the signals going into the EVB on a logic state analyzer. As soon as you identify one part of the overall system as being incorrect, consider dividing it further. Repeat until you have isolated the problem (e.g., bad chip).

Minimizing the Need to Debug

Remember the parable about the watchmakers (At the end of this article)? What it tells us is that "divide and conquer" is not only a good strategy for locating problems, but is also a great strategy for minimizing the need to debug in the first place!

To take advantage of this great principle, we must look ahead a bit. Basically, you should build a large and complex system in little parts, each of which are **explicitly** tested. Then, put these little parts together to build larger systems. Each time you put two things together to build something larger, **explicitly test the combination**. This way, you can be sure that only functioning building blocks are going into your system, and that each new building block works correctly with the rest. Importantly, if you do something wrong, you have something simple to debug. Furthermore, you can always undo what you did, and start again from an earlier point where things worked.

Reading the manuals carefully helps you avoid making silly mistakes.. the sort of mistakes that cost you dearly later in the form of long and frustrating hours of debugging.

Invest some time learning to use the oscilloscope and logic state analyzer. These tools can save you a lot of time later. For example, whenever you connect a hardware device, say the magnetic card reader, to the AVR, you can see exactly what signal is going into the AVR. This will help you quickly determine whether you are dealing with a software

problem or a hardware one. Thats so much better than staring at the system for hours and trying random ideas ("hacking").

Hardware Debugging Checklist

Here are some common mistakes that people make:

1. Forgetting to apply power to all chips.

2. Wiring chips/devices backwards. This is particularly common when dealing with the send and receive wires in serial communication. The easiest way to detect this type of situation is to see if some parts are drawing too much current and "bogging down" a signal. In extreme cases, parts may even overheat. Electrolytic capacitors are often wired incorrectly by students, partly because the markings are sometimes hard to see.

3. Messy wiring. Do your wiring systematically. For example, you can gain a lot by following simple color conventions. Use red wires for +5V, black for ground, yellow for the data bus, green for the address bus, and blue for all control signals. It will be advantageous for you to draw not only a logic diagram (necessary for grading!) but also a chip placement diagram. Start thinking about where your chips will go BEFORE you start building the circuit. This will help to keep your wires shorter and more uniformly laid out which will allow your circuit to perform better, and also make it easier to work with.

Miscellaneous Hardware Debugging Hints

One way to locate hardware errors is to **follow a signal** through the system. For instance, if a signal is going into a circuit, but is not coming out the output end the way it should, the circuit is suspect - very simple fact once you actually do this test. In many simple instances, a logic probe can be used to follow a signal path by looking for a characteristic blinking pattern. In some cases, we can use a pulse generator to produce a **test signal** that we can inject into our circuit and then follow its progress through our circuit. For more complex signals, we can use a logic state analyzer to follow time-varying signals.

READ the material at the beginning of the TTL data book. There is a lot of good general information here that will be of great help to you in your designs. Believe me, it will save you a good deal of debugging time.

Place a capacitor (about 0.01 uF to 1 uF) in parallel across the power and ground pins of EVERY chip. This helps keep power transients OUT of your logic where they can create some VERY weird problems. Such capacitors are called "Supply Bypass Capacitors".

To put a logic "1" or "high" on a pin, connect a l k ohm resistor in series between the pin and power. Don't connect it directly.

Note: This does not apply to the **power** pin of the chip, which should be connected directly to +5V.! (OR 3.3V, depending on the logic family!)

Keep your wires as SHORT as possible. A long wire acts as an antenna, which introduces crosstalk between signals. This problem only gets worse as the clock speed increases because the wires start to appear like transmission lines to the system. This is especially troublesome when dealing with asynchronous systems.

TTL logic is made for a wide variety of applications. There may be a chip that will enable you to implement your design in a much more economical manner. By this I mean fewer chips, and fewer chips means fewer wires, so more free time. Look at the data book index. In the index there are many chips that appear to do exactly the same thing. Actually, these chips have different features that you can (and should) exploit.

Software Debugging Hints

Build your software **hierarchically,** and in a **modular** fashion, like the successful watchmaker. This will minimize the need to debug in the first place.

Generally, things work better when you start out with a carefully conceived flow chart or state diagram representing your project. Be sure to discuss as many "what if?" scenarios with your partners. It is amazing just how many problems are revealed at this time. In fact, trying to explain your flow chart to a non-CE-student (e.g. a roommate, spouse, buddy or parent) is a great idea. The most successful charts are very concise and easy to understand. If they seem messy and hard to understand/explain, consider starting over.

One way to arrive at clean flowcharts is to organize your system design as a series of flowcharts at different levels of detail. One "high-level" flowchart should give an overall view of your software organization. This flowchart also helps define major software modules. You should try and make these modules such that each of them can be tested independently.

Once you have a clean and clear flowchart or state diagram, put it up on the wall in front of you as you go about writing your code. When you come in to the lab, put it up on the bench. This will keep you organized.

It may sound like a cliche, but it is true. Most of us don't do a sufficiently good job of commenting our code. The rule of thumb to use in evaluating the value of your comments is to show your code to another person, and see if he/she can understand what you wrote. If they don't, you flunk the "commenting exam"! Consider starting over!!

When you are not using interrupts, it makes sense to put in carefully chosen print statements throughout your code.

How to get rid of those Interrupt Programming Blues

When you're using interrupts, use the following checklist:

Did you clear the flag in the ISR? This is a common problem. The symptoms of the program are that it appears to "hang" because it is servicing the same interrupt again and again.

Many problems with interrupt based programs arise from an invalid stack.

Did you setup the stack correctly? You need adequate stack space, and you cannot afford to have the stack pointer off by even one byte!!

Systematically look over all your subroutines. Are they saving and restoring exactly the same number of bytes? Remember that if your SP is off by a just one byte, your program will crash sooner or later.

Did you use the correct address for the ISR?

Does your ISR end with the "RTI" instruction? Putting an RTS will cause your stack to get corrupted. In general, don't mix up interrupt service routines and subroutines!

When you are dealing with interrupts, especially the serial communication interrupts, you cannot use the usual technique of sprinkling lots of print statements in your code. In this case, you can use "indirect" methods for debugging your programs. For example, to find out if your ISR has been visited, you can set a memory location that you could query with a "memory display" command later. To find out how many times the ISR has been visited, make the ISR increment the byte in memory each time it is visited.

Another way to debug a program that uses interrupts is to take advantage of hardware output devices. For example, you can quickly interface one or more LEDs to one of the output ports, and turn them on and off each time something happens. For instance, if you turned on the LED each time an interrupt service routine was executed, and turn it off just before an RTI is executed, you can readily diagnose problems such as repeated interrupts, ISRs that invalidate the stack, etc.

List out all the subroutines that are called by your interrupt service routines. Then, check them line by line to see if any of them are using specific memory locations using either direct, indexed, or extended addressing. Such subroutines, called "non-re-entrant" subroutines are problemmatic

Is your ISR taking too long? In general, your ISRs should not take so long that you will miss an interrupt. It is a good idea to keep them as short as possible.

Do you have an invalid memory map? Check the listing for each piece of code to see what memory locations it is using, and make sure there are no unwanted overlaps.

The following story was told by Nobel Laureate Herbert A. Simon in his book "The Sciences of the Artificial," MIT Press, 1969.

Reading this story will help you understand hardware and software debugging better.

The Evolution of Complex Systems

Let me introduce the topic of evolution with a parable. There once were two watchmakers, named Hora and Tempus, who manufactured very fine watches. Both of them were highly regarded, and the phones in their workshops rang frequently - new customers were constantly calling them. However, Hora prospered, while Tempus became poorer and poorer and finally lost his shop. What was the reason?

The watches the men made consisted of about 1,000 parts each. Tempus had so constructed his that if he had one partly assembled and had to put it down - to answer the phone, say - it immediately fell to pieces and had to be reassembled from the elements. The better the customers liked his watches, the more they phoned him and the more difficult it became for him to find enough uninterrupted time to finish a watch.

The watches that Hora made were no less complex than those of Tempus. But he had designed them so that he could put together subassemblies of about ten elements each. Ten of these subassemblies, again, could be put together into a larger subassembly; and a system of ten of the latter subassemblies constituted the whole watch. Hence, when Hora had to put down a partly assembled watch in order to answer the phone, he lost only a small part of his work, and he assembled his watches in only a fraction of the man-hours it took tempus.

It is rather easy to make a quantitative analysis of the relative difficulty of the tasks of Tempus and Hora: Suppose the probability that an interruption will occur while a part is being added to an incomplete assembly is p. Then the probability that Tempus can complete a Watch he has started without interruption is (1 - p)^1000 - a very small number unless p is 0.0001 or less. Each interruption will cost, on the average, the time to assemble 1/p parts (the expected number assembled before interruption). On the other hand, Hora has to complete 111 subassemblies of ten parts each. The probability that he will not be interrupted while completing any one of these is (1 - p)^10, and each interruption will cost only about the time required to assemble five parts.

Now if p is about 0.01 - that is, there is one chance in a hundred that either watchmaker will be interrupted while adding any one part to an assembly - then a straightforward calculation shows that it will take Tempus, on the average, about four thousand times as long to assemble a watch as Hora.

We arrive at the estimate as follows:

1. Hora must make 111 times as many complete assemblies per watch as Tempus; but

2. Tempus will lose on the average 20 times as much work for each interrupted assembly as Hora (100 parts, on the average, as against 5); and

3. Tempus will complete an assembly only 44 times per million attempts (0.99^1000 = 44 x 10^6), while Hora will complete nine out of ten (0.99^10 = 9 x 10^-1). Hence Tempus will have to make 20,000 as many attempts per completed assembly as Hora. (9 x 10^-1)/(44 x 10^-6) = 2 x 10^4. Multiplying these three ratios, we get

1/111 x 100/5 x 0.99^10/0.99^1000 = 1/111 x 20 x 20,000 ~4,000.

Comments: Interestingly, this story also gives us some insight into the phenomenal success of Japan's industry. They use Hora's methods in building their products. Furthermore, these ideas are intimately involved in many phenomena in the world economy, and the evolution of life on earth! It is also related to an interesting new method for optimization - Evolutionary Optimization. Read Simon's book for more details!