ISIS, a real operating system

In 1974, I started on the design of a disk operating system, intended for the upcoming MDS multibus chassis. I had already designed and built a DOS at my previous company, and I had some idea what I wanted to do with the new DOS, and what I wanted to avoid.

My original plan was to have an internal hard drive of maybe 5 mb, along with a removable floppy disk or cassette tape for offline storage and distribution. I didn’t think a floppy was sufficient to do the job of supporting a full microprocessor development cycle. But, Marketing had different ideas. A hard disk was costly, it would require two different controllers, yadda, yadda, yadda.

The floppy disk had been developed by IBM as a distribution media for their big mainframe controllers. Their model was that an IBM tech would visit a customer site, use the floppy disk to install a patch or an upgrade to an installed controller, and then restart the controller with the upgrades. At no time was a floppy disk to be relied upon for secure data storage. If one failed, make a new one.

IBM competitors scrambled to find out how the floppy disk worked, and IBM made that as hard as possible. I was told by the Shugart salesman that the first IBM design info they managed to get had the drive rotating counter-clockwise, and they started a design of something similar, When the IBM production units showed up, they spun clockwise, thus setting the competition back by months, since they would have to redo the entire drive chassis. This was the device selected by Intel to use to create a microprocessor development system, over my objections. I may have the drive directions reversed, but you can get my point.

When we started the design, we looked at the track and sector layout: 77 tracks, with 22 128 bytes sectors per track, about 200k of storage. Note that the track count is a product of 2 prime numbers (11 and 7), and that the sector count is also a product of 2 prime numbers (11 and 2). This weird feature would cause untold complexity in any sector allocation algorithm, since it required 16 bit multiplies and divides to map that sector/track map into a continuous string of sectors, the kind of thing a file system wishes to work with. If you do the math, you can quickly determine that those multiplies and divides will always take a maximum amount of time, since you can’t do any clever shifts and adds based on the data, the data is always ugly, due to the prime numbers.

We (Jim and I, the entire ISIS project team) had long discussions trying to come up with an elegant solution, and we failed. Table lookups took up too much space, and calculations took too much time. We finally just used a 16 bit track/sector pair and created an allocation bitmap that padded out the odd sector count to 32 (I think), and filled in the dummy sectors bits as already used. This had the undesirable effect of storing the IBM mistakes directly in the file system, and made it difficult to add a hard drive to later versions of ISIS. I was no longer with Intel at that time, but came back as a contractor to work on that issue.

PIFS

Ridiculous drive layout aside, we took delivery of several Shugart drives and a tech went to work hooking it up on his bench. Once he got to the point where he could see data coming off the drive on his scope, he called me over to take a look at what he had accomplished. He had built a small wire-wrap board that used an Intel USART chip to read the stream of bits off of a floppy track, block it into 8 bit bytes, and then generate an interrupt, where his test program grab the byte and stuffed it into a array in memory, and then did it again. What you had was a primitive disk controller. With some software, you could set up a write command and it would write out a sector of bytes in the same manner. This was not a sophisticated disk controller by any stretch of the imagination, far from it. Basically, you could start writing a sector 0, track 0, and it would keep writing and incrementing until if reached the end of storage at track 76, sector 21. It did not support any kind of random access.

At the same time, there were many contentious meetings with Marketing over the amount of effort and budget needed to produce a ‘real’ disk operating system, versus what a ‘microprocessor development engineer really needed’. One of the Marketing people remarked that the tech already had the disk controller working. Jim and I decided we would test that idea.

We used the ROM BIOS that I had built over the past year, adding a PROM of code that mapped the tech’s crude disk controller writes into the one of paper tape punch channels and reads into one of the one of the paper tape reader channels. When hooked up to the text editor I had written in PL/M, I could write a small program in memory, save it by ‘punching’ it to floppy disk ‘tape’, save it, and then read it back.

We called this the Papertape Information on Floppy Storage, or PIFS, and presented it to Marketing. They asked about more than 1 file on the disk, our answer was “No, can’t do that”, it was really just a mapping of a string of bytes to/from a floppy. Marketing reluctantly agreed that this was shooting too low, and that Intel could be embarrassed if they attempted to market this solution. We obviously agreed. They asked what we called this piece of work, and we told them “PIFS”, which stood for the “Pie In The Face System”, since it would look like something the 3 Stooges had brought to market. Marketing decided to let us go ahead and design something more substantial.

Despite all the aggravation provided by Marketing, I really enjoyed writing ISIS. We decided early on that we would write the entire system as a single PL/M program, since all we had to work with was the PLM8 compiler (The PL/M80 version would become available before product release), and neither of us relished 8kb of assembler code. So, we were very careful with resources. We dropped all character I/O out of the DOS, and used the ROM monitor to handle keyboard and display chores. We did not use the 8080 interrupt system, because using it caused a code explosion due to the requirement for multiple buffers and other resources like low memory interrupt vectors. We ended up with a single track byte buffer which handled all I/O.

The final ISIS memory footprint was 12 kb, which made running an 8k assembler in a 16k ram system an impossibility. Marketing had a melt-down. After a great display of angst, Marketing finally agreed to ship a 32k ram minimum ISIS dual-floppy system. For this, they demanded that I produce an assembly language version of the OS, which could fit in 8kb (because PL/M was sooo ‘inefficient’). This version would be sold with a single floppy disk (still in the dual chassis, but with a blank plate covering the missing second drive).

The Marketing team presented this dual strategy solution to the sales force, and someone (my hero), said “But what happens if your dog pisses on your boot disk? You can’t make a copy!” The 16k ram requirement stayed, but the single drive idea was dropped. I didn’t see that error repeated until the Macintosh in 1984. But, the Mac was marketed as a toy, and they got away with it.

After ISIS was released, written in PL/M, consuming 12k of ram and supporting an assembler and text editor (also written in PL/M), all fitting into a 32kb ram configuration, I did produce a hand-built assembly language version. I found that the PDP-10 timesharing system also supported a Simula compiler, which is a nice Algol-like language, so I used that to build a disassembler to convert the generated PL/M code back to assembler, which I then hand-tuned to squeeze 12k bytes into an 8k bag. I had to drop a random access feature from the file system, but neither the assembler or text editor depended on that feature. We sold two of those systems.

ISIS did very well in the market and had a stunning 0 bug reports over the first year.

For reasons unknown to me, Intel had reduced the work being done by Gary Kildall, the developer of the PL/M cross compiler. If I had to make a supposition, I would say the new software manager wanted tight control over all software work being done by and for Intel Corp, and independent consultants made him experience loss of control. That manager was “wound a bit tight”.

Gary did manage to get one of the early production disk controllers we built for the ISIS project, trading for something Marketing wanted done, and the development of CP/M proceeded in parallel with our work on ISIS. Gary and I discussed possible solutions for doing file allocation and I/O, and how to manage the disk controller, and such like, but each DOS was developed separately.

That done, it was time to make the PL/M compiler run on ISIS.

But first, the 8080 vs. The Motorola 6800.

Intel Marketing was worried about the 6800, because it presented a much more sane instruction set than the 8080, and that was hard to argue against. So, Intel started touting the 8080’s speed relative to the 6800.

There was a particular free-lance technical writer who wrote articles showing how the 6800 could beat an 8080 in an assembly language face-off. My second level manager was livid about this guy, and challenged him to a code-off. They agreed to a small set of computing tasks that each CPU would have to do, and then they would tote up the cycles used. The loser would have to buy dinner for the winner at an expensive French restaurant in San Francisco. This manager came out of his office down the hall, shouting about this tech writer and said that he needed someone to write the winning code for the 8080, and looked directly at me. Cripes.

To make a long story shorter, I would point out that the fastest way to move 16 bits in an 8080 is to push it on the stack. The stack pointer is a 16-bit register, whereas all the other general registers were 8-bit pairs (BC, DE, HL). Since one of the algorithms in the contest was a string copy, I simply pushed 2 bytes at a time onto the stack, ignoring the fact that you can’t actually use a trick like that if you also want to have subroutines. But, we won, and the tech-writer had to take my wife and I to a nice dinner.

The competition continued over time, with the 8085 being developed as basically an 8080 with some process improvements that gave it a higher clock speed. When the design was presented to the software group, it turned out that the chip guys had added a couple of instructions to the chip-set, and wanted us to add those instructions to the assembler. The instructions had little real value, so we absolutely refused to support the new instructions, and they still remain buried in the old 8085 chips. I felt we got some revenge for the ‘dog pee DOS’ of the past.

And now, ISIS-II, the whole shebang

The Software Group at Intel really wanted to take programming languages further to support the microprocessor chip war, so first, we had to have a real, native 8080-based compiler. In addition to the significant effort to build the compiler, that also implied a lot of other pieces, like a linker, an object module definition, and a new assembler that could generate the same kind of object module as the compiler. Add to that the need to support the In Circuit Emulator under development, and we had a big, complex project.

I figured I had it quite easy, since all I would have to do was add support to the ISIS loader to handle new object formats, and also recompile all of ISIS with the new compiler. No big deal. Unfortunately, the management decided they needed a single Release Chief to make sure all the parts got done on time and integrated properly into the whole. That turned out to be my job.

We had contracted with an outside software house to build a new compiler, and that was going well. These guys had a good compiler-writing system and methodology, and they proved that they could make a compiler run successfully in the 64k address space of an 8080. The new compiler was a joy to work with, and Jim and I made short work of the port of ISIS to the new compiler system.

A good friend of mine, Bruce, got the job of developing the Object Module Format (OMF) for the 8080, and the upcoming 8086. This piece of work would live on for years, since MS-DOS used it with little or no modifications. It was a good piece of work and it was updated in 1985 to support the 80386 32-bit flat address mode. It probably was updated again when the 64-bit architecture came along, but I wasn’t in the loop at the time.

The OMF gave us a good roadmap for the design of the linker, which knitted object modules together, and the locater, which processed the relocatable code into a fixed memory image, with a defined entry point, since that was how ISIS handled load modules. The locater was also built to enable mapping of a linked module into a set of EEPROM images, so they could be programmed by a prom-burner.

The assembler proved to be more of a headache than some of the other more complex programs. It became the single biggest risk to meeting our delivery dates. I checked with the 2 person team working on the assembler, and one of them started complaining about floppy disk errors causing loss of data and therefore causing the time slip. We had probably a dozen engineers writing code all-day long, and these two were having by far the most problems. I looked around their shared office, and saw diskettes, out of their storage sleeves, spread all over any horizontal surface. Sitting nearby was an ash-tray, full of cigarette butts (this is 1975). I pointed out that KEEPING THE DISKS IN THEIR SLEEVES was a good idea, and that SMOKING NEARBY AND DROPPING ASHES ON THE DISK SURFACE, was the probable cause of their error rate issues. I let it be known that that would stop now and that they had better clean up their office. They did bring the assembler in on time, just, but it continued to produce more bug reports than the rest of the release combined.

The In-Circuit-Emulator (ICE) was probably the most complex software code in the ISIS-II release. It was all new, and handling issues of high-speed demands and complex timing. I felt very lucky that the development team was a seasoned group of developers who knew their jobs, and they delivered on time.

Being a Release Team Leader was not always fun.