17 For Further Study

If you've worked your way through the book to this point, you will have a good understanding of how the Industry Standard Architecture bus operates, how to build useful ISA bus hardware, and write functional low-level firmware.

In this chapter, I'll discuss some aspects of the ISA bus that I either glossed over or simply omitted from the Firmware Development Board's hardware and firmware. These are relatively complex topics that don't have much relevance to current ISA bus designs, primarily because contemporary PC systems provide better ways to achieve the goals.

Crossing the 1 MB Line

Each ISA bus board can access 16 MB of memory using the 24 memory address bits that appear on each ISA connector. However, both the battery backed RAM and the Graphic **`LCD Refresh RAM`** on the Firmware Development Board decode only 20 bits and reside entirely below the 1 MB line. Why is that?

In short, back when the ISA bus was new and 80286 CPUs were state-of-the-art, the ISA bus matched the CPU's performance rather well and 16 MB of RAM was a *lot*. Neither condition holds true today.

Today, typical system boards accommodate 128 MB or more of RAM and even desktop operating systems require 32 MB for *normal* operation. Given the 24-bit address space available to an ISA bus board, its RAM must reside well below the middle of the system board RAM. But very few operating systems allow a block of special purpose RAM within the memory space above 1 MB. If you expect to use your ISA bus memory for, say, a big LCD buffer, you'll watch your operating system's data in action, not your own bits!

Even if you put general purpose RAM on that board, it will run at ISA bus speeds instead of the 60 ns access time found in system board RAM. Compare 60 ns with the times shown in Photo 1 of Chapter 13 and you'll see why using ISA bus RAM in a contemporary system is an Exceedingly Bad Idea.

You can assign and use system memory as you wish in an embedded system project, because you're writing the code yourself. Some BIOS setup routines can carve a hole in the system board RAM for an ISA bus board's memory, either by disabling the system's RAM at those addresses or remapping it to a different address. Just don't expect much help if you're trying to run a standard operating system as part of your embedded project. They simply don't do the things we need.

To gain access to any memory above 1 MB, your program must either run in protected mode or call routines that do. Nowadays, toolkits from Phar Lap Software and others make writing protected mode embedded programs relatively straightforward, but there's a whole lot of unexplored territory between the down-to-the-metal firmware we've been using and the code you write with a protected mode toolkit. If you go that route, spend some time making the detailed measurements you've seen in this book *before* assuming that protected mode operation provides the performance you need.

Of course, you could use the BIOS extended memory copying routines shown in Chapter 15, provided you work around their limitations. Most embedded programs cannot stand long periods without interruptions, so make sure you understand how your application will cope as the BIOS switches into and out of protected mode.

Check the references for information on the **-MemR** and **-MemW** bus signals that control access above 1 MB. The **LA17** through **LA23** address bits also have slightly different timing requirements that will affect your logic design.

Direct Memory Access

Everything we've done with the ISA bus requires direct supervision from the CPU, whether to read an I/O port, write data into memory, or respond to an interrupt. In effect, the CPU must execute at least one instruction per ISA bus operation. As it turns out, the system board and ISA bus include hardware that can transfer data between I/O ports and memory without involving the CPU, a process called DMA: Direct Memory Access.

The diskette controller uses DMA to transfer data between the diskette drive and memory, but, because the BIOS can handle only one task at a time, it puts the CPU into a tight loop until the DMA transfer finishes. That's not a particularly good use of the DMA hardware, but that's how the real mode BIOS works.

Many sound boards transfer digital audio data using the DMA hardware, leaving the CPU free to render game graphics (for example) at the same time. This works reasonably well, although the overhead involved at the beginning and end of each DMA transfer requires either two separate channels, an on-board FIFO buffer, or brief silent periods while the CPU sets up the next chunk of the sound file.

Regardless of the CPU speed, the ISA bus DMA hardware executes roughly one transfer per microsecond. Because the ISA bus can handle at most two bytes of data on each bus cycle, DMA transfers run at 1 or 2 MB/s. That's slightly less than the rates we've measured throughout the book, so, contrary to popular belief, using DMA does *not* move data at a particularly high speed!

If you're familiar with microcontrollers that include DMA support, you know that you can generally gain a significant speed advantage by block-moving memory with DMA hardware. On any PC faster than a '386, however, the *x*86 `REP MOVS` instruction outruns the system board's DMA hardware by a considerable margin, with considerably less overhead.

Surprise!

So… unless you have a compelling reason to use DMA, don't bother. If you absolutely must use DMA, consult Solari's book for the myriad details required to get it working on all possible systems. There are significant differences in the system board hardware available that will affect how you design your board.

The PCI bus is a better choice for systems with exceedingly stiff response time specs and high data transfer rates, despite the additional complexity of designs using that bus. If you find your project becoming encrusted by the PC Compatibility Barnacles on the ISA bus, the PCI bus may provide a clear path.

My back of the envelope estimate says that you can do about as well as ISA bus DMA with a moderately large FIFO buffer and a well written interrupt handler transferring data across the bus using `REP OUTS` and `REP INS` instructions. You'll avoid all the hassles of finding and using a DMA channel, too.

Busmastering Boards

Pin D17 on the ISA bus sports the label **`-Master`**. In principle, when an ISA bus board asserts that signal the system board relinquishes the bus and allows the board to drive the bus control lines. In practice, this doesn't work nearly as well as you'd expect, because the ISA bus was designed to support a single CPU on the system board that controlled a bunch of relatively dumb peripheral boards.

An ISA bus board that would become the master must first set up a DMA transfer using a DMA channel. When the DMA controller asserts that channel's **`DRQ`** line, the board responds by asserting the corresponding **`-DAck`** signal and then asserting **`-Master`**. At that point, the DMA controller has released the bus and will not proceed with the next DMA cycle.

A busmaster board must allow normal DRAM refresh cycles by either asserting **`-Refresh`** every 15.6 µs or releasing control of the bus. This can pose a problem if you expect to process large blocks of data at high speeds with a CPU on an ISA bus board. In any event, you certainly won't get the bus throughput you might expect.

I've worked with intelligent ISA bus boards on several projects and can say that the best bus interface is the simplest. If you can *possibly* get along with a small block of shared memory, similar to the Graphic **`LCD Refresh RAM`**, don't bother with a busmastering board. Even better, if you can pass data through a few I/O ports, you've reduced your board's footprint to the point where it's certain to work.

Unfortunately, that simplicity may not be feasible if your board must share and process huge blocks of data. In that case, rather than force-fit the project into the ISA bus, it's time to break out those PCI design tools and do the job right.

Release Notes

Gather your tools, collect some parts, and settle down for some construction!

Remember to check the Web page for more information and send me a note when you come up with a neat trick.

Now it's *your* turn…