Ingo Cyliax

MC68030 Workstation

The Hardware

Part 1 of 3

What do you do when components are no longer sold? Ingo built his own system. It was the only cost-effective way he could get a 68k platform to teach assembly-language programming when Motorola's MC68000 ECB was discontinued.

The MC68030-workstation project started when our faculty group realized that there was no good 68k platform for teaching assembly-language programming. The traditional MC68000 ECB from Motorola is no longer sold, and the evaluation kits for the 683*xx* MCUs weren't exactly right for some tasks—not to mention that they're overpriced. So, I suggested building our own teaching platform.

We quickly decided this new system had to be 68k based, since many textbooks use this processor as one of the architecture examples. It was also decided that it needed an ISA-compatible expansion bus as well as a scan-code-based keyboard and some kind of memory-based display interface.

Why ISA bus on a 68k-based system? ISA bus is a very cost-effective system, so it fit our limited budget. Since most of the PCs in the department are being continuously upgraded, I was able to rescue many 8-bit VGA and Ethernet cards from our surplus bins.

I also considered "cleaner" buses such as PCI and VME, but could not justify buying cards for these buses when free cards were available.

In addition to the ISA-bus slots, I added three custom "cpubus" connectors which provide access to the unbuf-

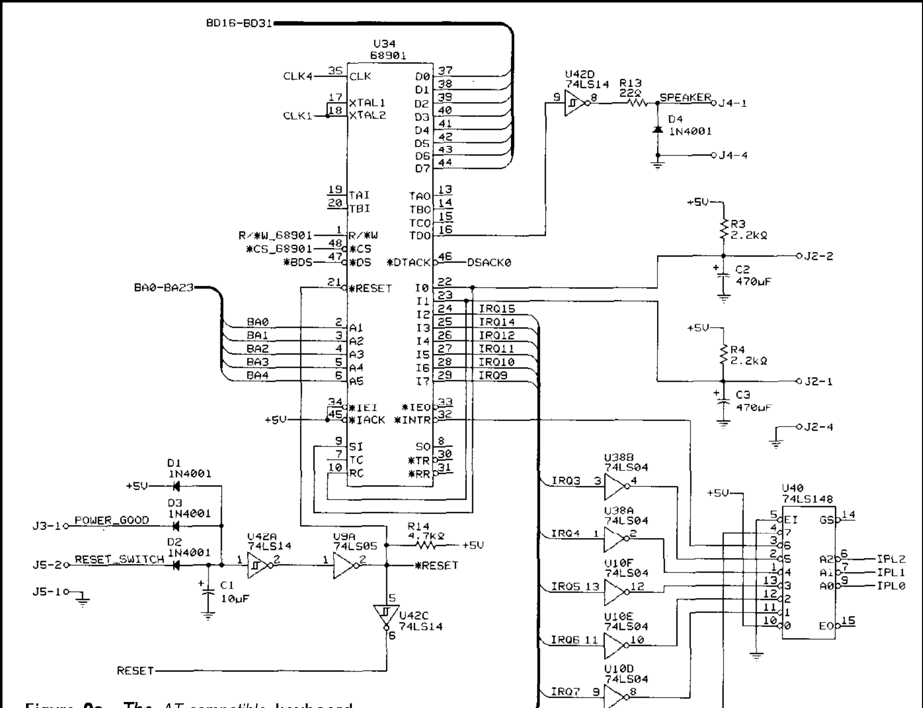


Figure **2a—*The*** *AT-compatible* **keyboard interface is provided by the MC68901 *Multifunction*** *Peripheral Controller* ***(MFP) (U34). It*** *also interfaces to the speaker and consolidates some of the* ***16-bit*** *ISA-bus interrupts. The* ***74LS148 (U40)*** *encodes the interrupt requests info the interrupt /eve/ for the CPU.*

fered CPU signals. So, coprocessors and fast memory can be added to this system.

I also decided to use LSI/MSI TTL components when possible. This way, every part in the system is identifiable, so it should be easy for students to infer what each component does from standard datasheets.

There are two exceptions. One is a PAL for decoding the bank selects for the DRAM memory. Also, I used an MC68901 MFP-a companion I/O chip to the 68k line. It contains some timers, parallel I/O, and the serial port used for the keyboard interface.

Since this machine is essentially built from scratch and I wanted to follow the open methodology in hardware and software, this meant writing many components from scratch and using freely available tools if the source code was available. Later in the series, I'll describe the software-development tools I used and the software I wrote for this system.

By the way, the schematics, PCB artwork, and program sources are

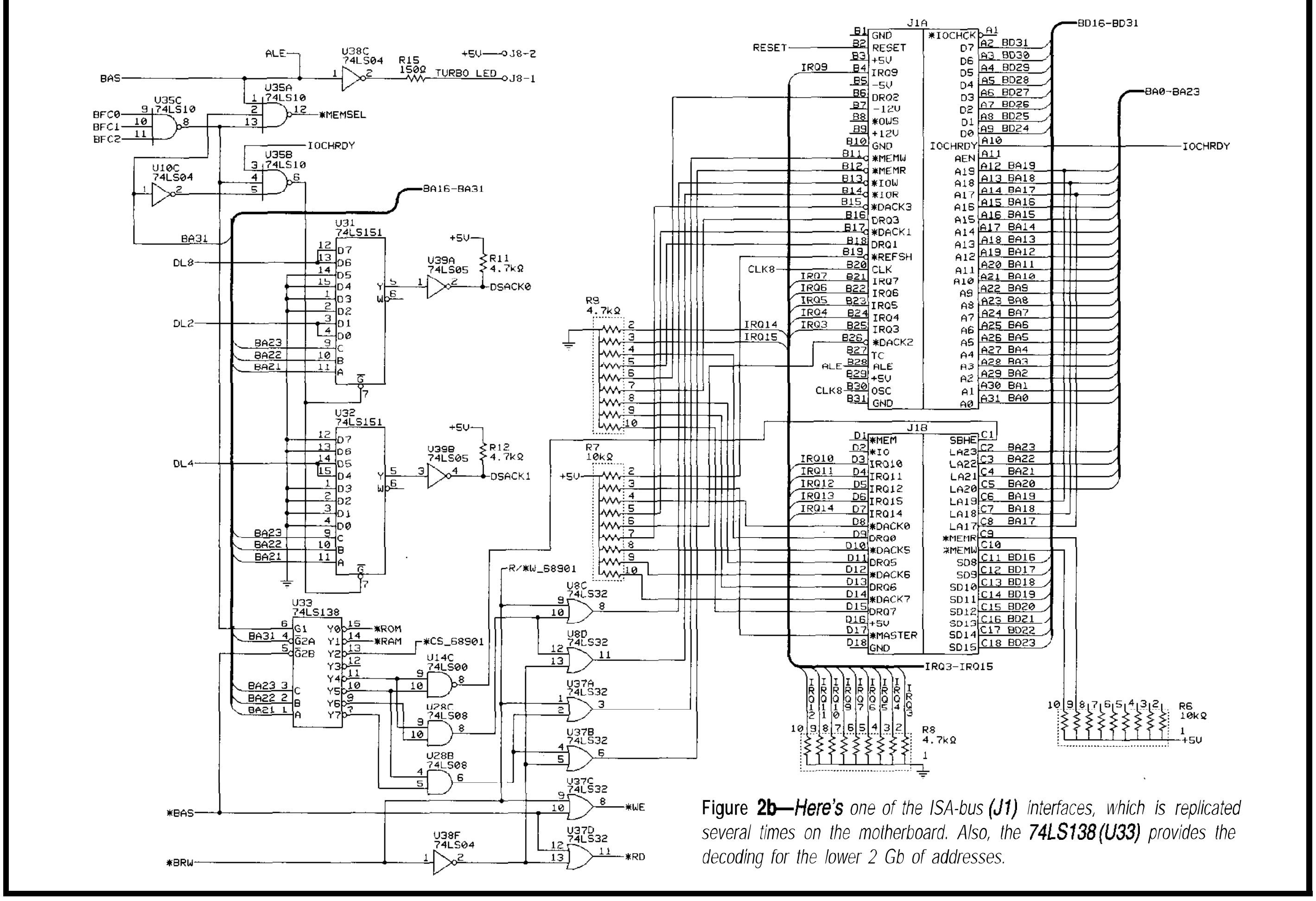


Figure **2b—*Here's*** *one of the ISA-bus* ***(J1)*** *interfaces, which is replicated several times on the motherboard. Also, the* ***74LS138 (U33)*** *provides the decoding for the lower 2 Gb of addresses.*

Address	Size	Description
00000000-000fffff	8	64-Kb EPROM
00100000-001fffff	8	32-Kb SRAM
00200000-002fffff	8	MFP
00300000-003fffff		illegal
00400000-004fffff	16	ISA-bus I/O
00500000-005fffff	16	ISA-bus memory
00600000-006fffff	8	ISA-bus I/O
00700000-007fffff	8	ISA-bus memory
00800000-7fffffff	–	repeat above (0-7fffff)
80000000-803fffff	32	4-Mb DRAM
00400000-ffffffff	32	repeat DRAM

Table I--Since *the* MC68030 processor *has no I/O* instructions, *the ISA* bus is broken up into *four* address *maps.* Each causes *a different bus cycle to occur on the ISA bus.*

available via ftp, and you're free to use them for noncommercial purposes.

HARDWARE OVERVIEW

The system can be roughly divided into four major groups-the CPU, ISA bus, DRAM interfaces, and onboard I/O. There's only one onboard I/O device on this system-the AT-compatible keyboard interface.

I chose the MC68030 for my CPU. It has a flexible bus interface and an integrated MMU module. Since it has dynamic bus sizing, it was easy to provide interfaces for 8-/16-bit ISA bus, 32-bit DRAM, and 8-bit EPROM and SRAM with minimal logic.

The processor asks for an object, and the system responds with the port size available. If the system port size is smaller than the object to fetch, the internal bus interface in the MC68030 automatically sequences and muxes the data to the appropriate place.

The MC68030 has two bus termination modes. The asynchronous termination mode, indicated by asserting a combination of DSACKO and DSACK1, is used when dynamic bus sizing is required.

Its timing is also compatible to the 68k bus cycle. It is asynchronous because it assumes the data and DSACKx signals are asserted asynchronously and have to be synchronized to the CPU clock before processing.

The synchronous bus cycle, indicated by asserting the STERM signal, shows that the data and STERM are synchronous to the processor clock and can be processed without an extra synchronization cycle. Also, the STERM signal indicates that the port size is 32 bits. STERM allows the fastest possible bus timing on the MC68030.

The MC68030 also has a burst-mode operation to fill its cache from page-mode DRAMs using an extension of the synchronous-bus-cycle mode. But since this system is low end, this mode wasn't implemented. The data and instruction caches also work well with the asynchronous and nonburst-mode synchronous-bus-cycle modes.

Given the flexible bus interface, the MC68030 is easy to design with. I ended up using asynchronous-bus-cycle modes when addressing the ISA bus (8- and 16-bit ports), PROM/SRAM (8-bit port), and onboard I/O resources (8-bit port). I used synchronous cycles for speed when addressing the system's DRAM (32-bit port).

A 74LS138 (8:1 selector) decodes the I/O, ISA bus, PROM, and SRAM address space in the lower 2-Gb address space. The higher order address bit selects the DRAM which, even though 4 Mb is implemented, repeats in the 2-Gb upper address space.

By placing the DRAM in the upper address space, I was able to decode the 8-bit PROM in the lower memory where the process fetches its reset vector and initial stack pointer. Table 1 shows the address map of the whole system.

A shift register generates differently timed bus acknowledge signals. These are selected using a 8: 1 selector-one for each DSACKx signal-to generate different wait states for each decoded object. Hence, the ISA-bus I/O cycle has to be longer than an SRAM access.

Another shift register generates a bus-error exception if no bus acknowledgement is received. This feature implements a bus timeout for references to illegal address ranges.

All that remains to interface to the 68030 is the interrupt interface and a single-phase 1 6-MHz clock, which is derived from a TTL clock module.

If you need I/O...

Add these numbers up:

80C552 a '51 Compatible Micro
40 Bits of Digital I/O
8 Channels of 10 Bit A/D
3 Serial Ports IRS-232 or 422/485)
2 Pulse Width Modulation Outputs
6 Capture/Compare Inputs
1 Real Time Clock
64K bytes Static RAM
1 + UVPROM Socket
5 12 bytes of Serial EEPROM
1 Watchdog
1 Power Fail Interrupt
1 On-Board Power Regulation

It adds up to real I/O power!

That's our popular **OEM 552SBC-40**, priced at just $299 in single quantities. Not enough I/O? There is an expansion bus, too! Too much I/O? We'll create a version just for your needs, and pass the savings on to you! Development is easy, using our Development Board: The **552SBC-50** Development board with ROM Monitor for $349.

New Versions of the 8031SBC

Our popular 803 1SBC can now be shipped with your favorite 8051 family processor. Models include **80C5 1 FA, DS80C320, 80C550, 80C652, 80C154, 80C851** and more. Call for pricing today!

Truly Low-cost In-Circuit Emulator

The DryICE Plus is a low-cost alternative to conventional ICE products. Load, single step, interrogate, disasm, execute to breakpoint. Total price for the base unit with most pods is a low $448. Call for brochure, or **World Wide Web** at **www.hte.com.**

Call for your custom product needs. Quick Response.

HTE

HiTech Equipment Corp.
9672 Via Excelencia
San Diego, CA 92126
(Fax: (619) 530-1458]

Since 1983

—— (619) 566-l 892 ——

VISA MasterCard

Internet e-mail: info@hte.com
World Wide Web: **www.hte.com**

#133

MC68030's interrupt system is also very flexible. (What did you expect?) It uses multilevel priority-based interrupts (IRQ1–7) to request an interrupt. IRQ7 is the highest level and is not maskable. The external interrupter asserts an interrupt by encoding it into the three interrupt signals IPL[0:2].

In this design, I used a 74LS148 (8-l priority encoder). Figure 2a shows the interrupt circuitry and ISA-bus and keyboard interfaces, while Figure 3a shows the CPU.

Once the MC68030 sees the interrupt request by noting that one or more of IPL[0:2] are asserted, it responds to the interrupt if the interrupt-request level is higher than its current interrupt mask or if it's a level 7. It starts an interrupt-acknowledge cycle that resembles a regular bus cycle, except that a special bus cycle is indicated with the FC[0:2] function codes.

The interrupt-acknowledge cycle can be terminated by a peripheral by supplying an external interrupt vector and asserting one of the bus termination signals or by asserting AVEC. If the cycle is terminated with AVEC, it uses a predetermined internal interrupt vector.

A large real-time system may use many interrupt vectors to tell the processor what the source of the interrupt was and how to handle it without spending much time hunting down the cause. In this design, it's acceptable to spend time polling devices for the interrupt cause, especially since ISA-bus devices don't generate interrupt vectors.

Generating the autovector bus termination was easy. I simply decoded the function code FC[0:2] to indicate an interrupt-acknowledge cycle and assert AVEC.

The integral MMU was an interesting option, since it enables the use of this platform in a serious OS-type class project (e.g., virtual memory or memory management in modern OSs). In fact, I developed a version of Minix that uses the MMU, which the students can dissect and explore. No external hardware is needed to enable the use of MMU functions on the MC68030.

The ISA-bus interface uses four of the select lines to form the address decoder to implement four address maps. Each address map indicates one of the possible ISA-bus cycles.

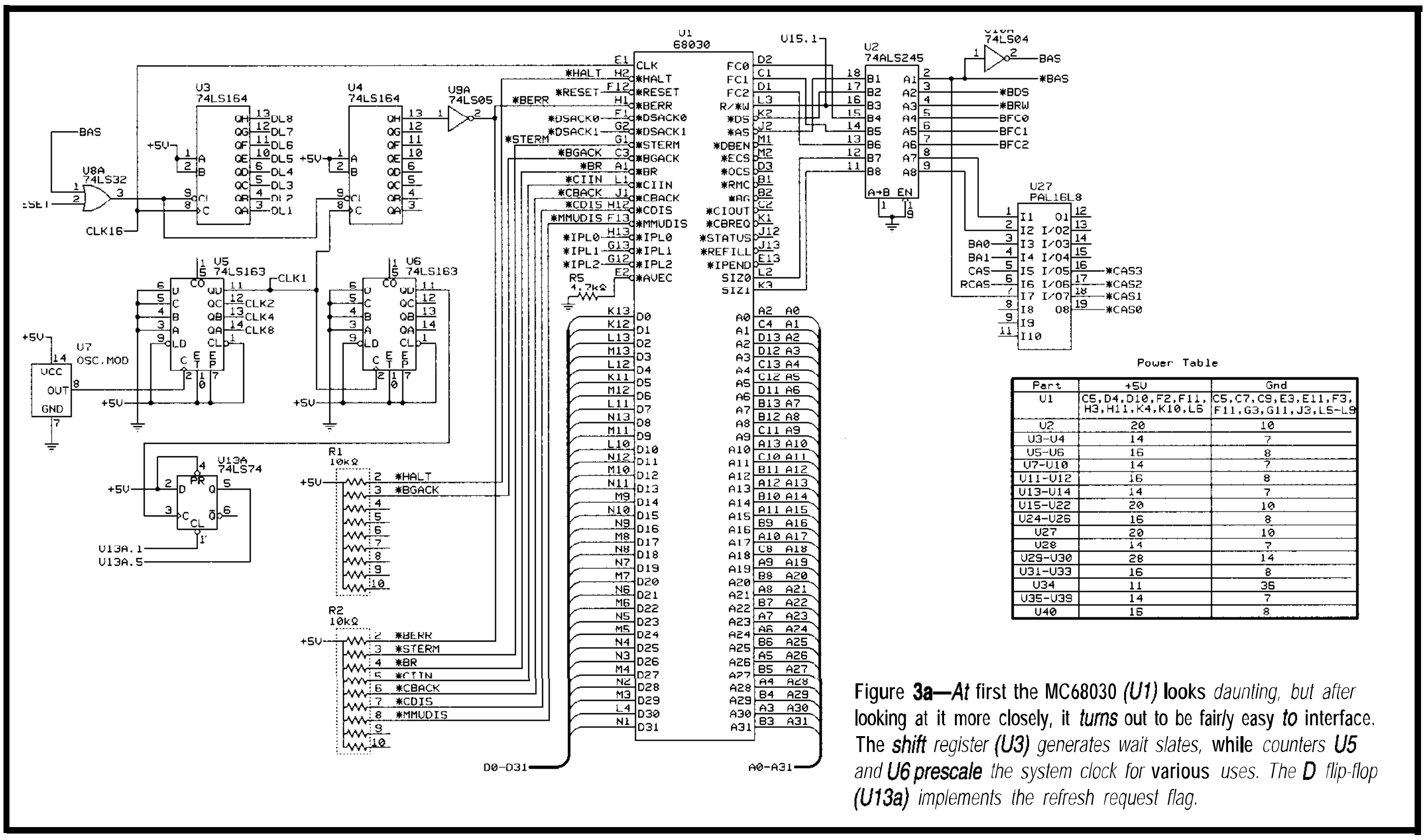
There are 8- and 16-bit memories and 8- and 16-bit I/O accesses. The memory accesses use the signals MEMRD and MEMWR to indicate a memory read and write on the ISA bus. An I/O cycle uses the IORD and IOWR signals.

Figure 2b shows how the ISA-bus interface is implemented. An ISA-bus card can also extend a bus cycle by asserting the signal IORDY. It was necessary to implement this signal, since many VGA cards stretch the CPU bus cycle to deal with memory contention to the video memory.

In addition to expanding the interrupt system by implementing the 16-bit expansion interrupts, the MC68901 Multifunction Peripheral (MFP) handles the AT-compatible keyboard interface.

The MFP expands the number of interrupts by using six of the eight general-purpose I/O pins as external interrupt pins. They can be programmed to be edge as well as level sensitive. The MFP interrupt controller makes it easy for software to find the interrupt source by providing some registers that indicate pending interrupts.

There are also facilities for masking any of the interrupts and clearing pending interrupts. Table 2 shows the interrupt map of this system. As you can see, many of the interrupts are handled by the MFP and are presented as a single interrupt request at level 6 to the CPU.



Power Table

Part	+5V	Gnd
U1	C5,D4,D10,F2,F11, H3,H11,K4,K10,L6	C5,C7,C9,E3,E11,F3, F11,G3,G11,J3,L5-L9
U2	20	10
U3-U4	14	7
U5-U6	16	8
U7-U10	14	7
U11-U12	16	8
U13-U14	14	7
U15-U22	20	10
U24-U26	16	8
U27	20	10
U28	14	7
U29-U30	28	14
U31-U33	16	8
U34	11	36
U35-U39	14	7
U40	16	8

Figure **3a**—*At* first the MC68030 *(U1)* **looks** *daunting, but after* looking at it more closely, it *turns* out to be fairly easy *to* interface. The ***shift*** *register (U3) generates wait slates,* **while** *counters U5 and* ***U6 prescale*** *the system clock for* various *uses. The D flip-flop (U13a) implements the refresh request flag.*

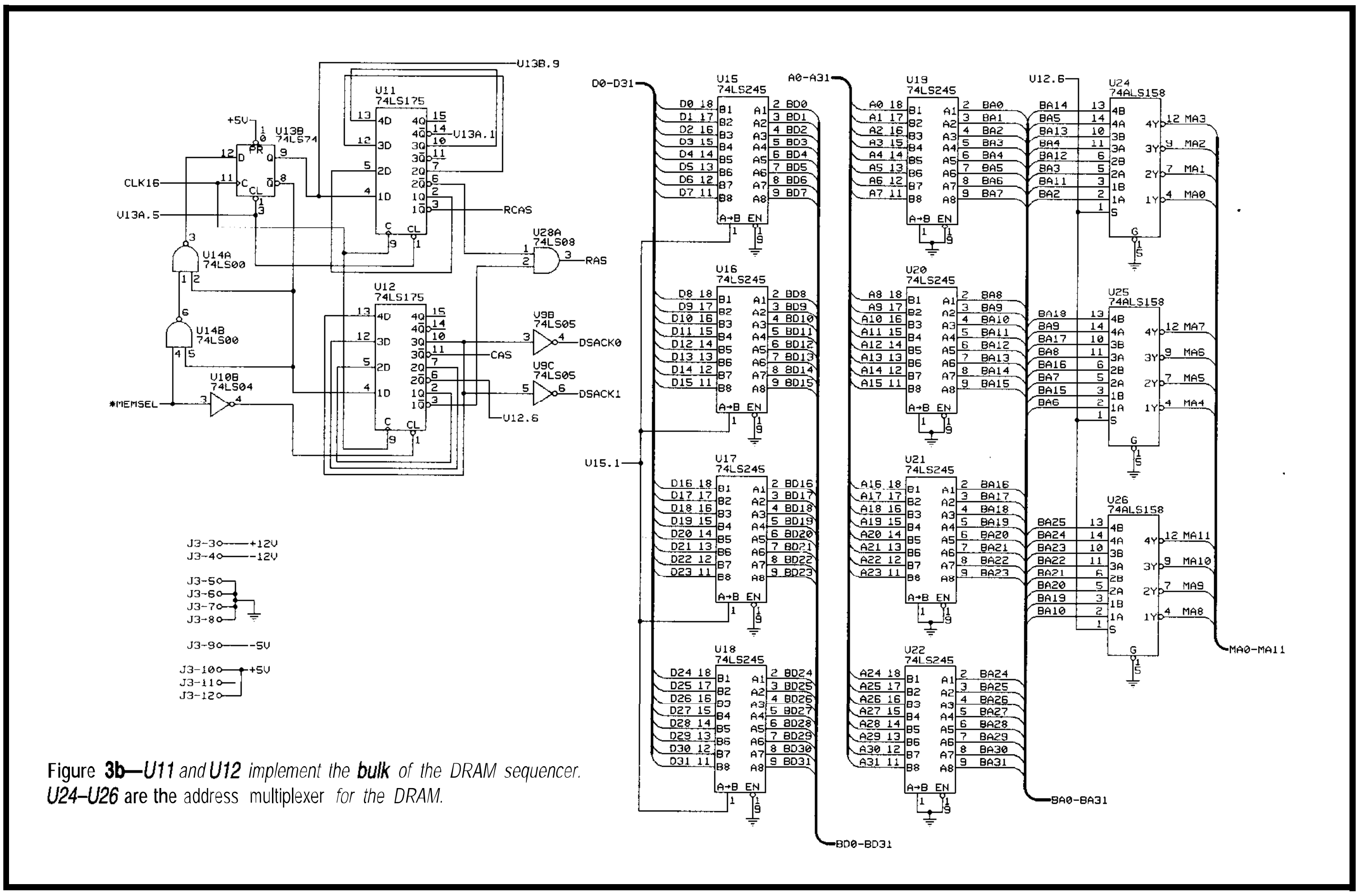


Figure 3b—*U11 and U12 implement the* **bulk** *of the DRAM sequencer. U24–U26* are the address multiplexer *for the DRAM.*

We know INK

STANFORD • RYERSON POLYTECHNIC • ROCHESTER INST. OF TECHNOLOGY • TUFTS • HARVARD • PURDUE • DEVRY INST. OF TECHNOLOGY • PENN STATE • CORNELL • CALIFORNIA STATE • U.S. NAVAL ACADEMY • MILWAUKEE SCH. OF ENGR. • CASE WESTERN RESERVE • MERCER • AALBORG • AMARILLO C. • APPALACHIAN • MIT • STATE U. • U. OF ARIZONA

Do You?

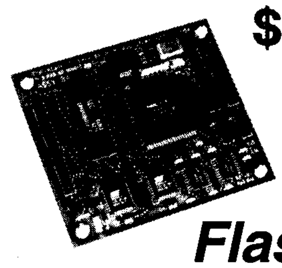
If you're an engineering professor or teach qualified technical students, we'd like to give you and all of your students free subscriptions to *Circuit Cellar INK*. Contact:

Rose Mansella • Circulation Coordinator • Circuit Cellar INK
4 Park St. • Vernon, CT 06066 • (860) 875-2199 • Fax: (860) 871-0411
rose.mansella@circuitcellar.com • www.circuitcellar.com

EMBEDDED DOS
CONTROLLERS AT 8051 PRICES

Jse Your PC Development Tools
No More Crash & Burn EPROM Technology

$195 Qty 1 Price

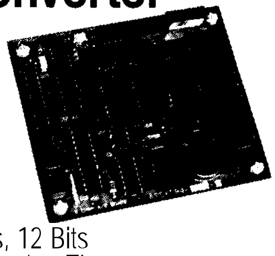


Flashlite
DOS *Single* Board Computer
with 512 k FLASH Memory disk drive

✓ 10 Mhz/8 Mhz CPU ✓ 2 Timers
✓ 512 k bytes RAM ✓ 4 Interrupt Lines
✓ 512 k/256 k FLASH ✓ 8 Analog Inputs
✓ 2 Serial Ports ✓ X-Modem File Transfer
✓ 24 Parallel I/O Lines
INCLUDES DOS & Utilities

USE YOUR TURBO C COMPILER OR QUICKBASIC COMPILER SAVE TIME, MONEY AND HEADACHES

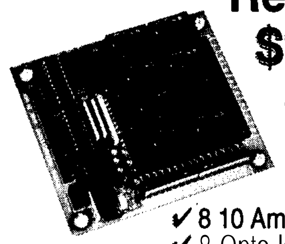
A/D Converter
$95 Qty 1 Price



✓ 8 Channels, 12 Bits
✓ 6 µs. Conversion Time
✓ Clock/Calendar Option
✓ Includes Drivers & Apps.

GET YOUR EMBEDDED CONTROLLER PROJECT RUNNING *FAST!* WITH THESE ACCESSORIES

Relay I/O
$139 Qty 1 Price



✓ 8 10 Amp Relays
✓ 8 Opto-Isolated Inputs

JK microsystems
Cost Effective Controllers for *Industry*
TO ORDER (510) 2364151
FAX (510)236-2999—email: jkmicro@dsp.com
Visit our WEB site—www.dsp.com/jkmicro
1275 Yuba Ave., San Pablo, CA 94808

#134

Level	Source	Function
ipl7	"break" button	NMI
ipl6	MFP(15)	ISA-bus IRQ9
ipl6	MFP(14)	ISA-bus IRQ10
ipl6	MFP(13)	Timer A (60 Hz)
ipl6	MFP(12)	KBD receive
ipl6	MFP(11)	KBD error
ipl6	MFP(8)	Timer B
ipl6	MFP(7)	ISA-bus IRQ11
ipl6	MFP(6)	ISA-bus IRQ12
ipl6	MFP(5)	Timer C
ipl6	MFP(4)	Timer D
ipl6	MFP(3)	ISA-bus IRQ14
ipl6	MFP(2)	ISA-bus IRQ15
ipl6	MFP(I)	KBD data
ipl6	MFP(0)	KBD clock
ipl5	direct	ISA-bus IRQ3
ipl4	direct	ISA-bus IRQ4
ipl3	direct	ISA-bus IRQ5
ipl2	direct	ISA-bus IRQ6
ipll	direct	ISA-bus IRQ7

Table 2—*These are all the* possible interrupt sources on the *MC68030*. Many of the interrupts are routed through the MC68901 *MFP*, which *prioritizes all* of its *interrupt* sources *(0-15) and uses* interrupt priority level 6 *to notify the CPU.*

The MFP implements the keyboard interface by using its internal USART. The USART has a receive clock input that enables a lx baud-rate clock, even in asynchronous mode.

This feature is perfect for the AT keyboard interface, since it uses a clock signal to indicate when the serial data needs to be sampled. The data format for the AT keyboard scan codes is 1 start bit, 8 data bits, even parity, and 1 stop bit. Figure 2b shows how the keyboard interfaces to the MFP.

To make the design more interesting and useful for larger projects, I added DRAM. A single 72-pin SIMM module doesn't take up much real estate, but since it uses 50-mil staggered pins, it's a little harder to prototype with. The DRAM interface port size is 32 bit and uses a 74LS163 shift register to sequence the CAS and RAS, which are presented to the PLD for further decoding.

I used 74LS150 2: 1 muxes to multiplex the address bus. Since the 68030 needs to be able to write byte data to memory, I implemented a bank decoder in the PLD to assert the correct combination of CAS*x* selects and satisfy any possible data-transfer situation.

Figure 3b shows how the DRAM subsystem interfaces with the CPU. Table 3 shows the truth table for the byte-selection logic implemented by the PLD.

Of course, you have to refresh all the rows in the DRAM once every 2 ms. This task is accomplished by dividing the system clock down to 4 us. That gives one total refresh cycle in 2.048 ms, which I thought was close enough.

The 4-µs clock edge triggers a 74LS74 flip-flop to indicate a pending refresh request. Once the current CPU cycle finishes, a separate shift register will time a CAS before RAS refresh cycle and present the PLD with an RCAS signal to indicate that this is a refresh cycle, while the next CPU is blocked.

Once the refresh cycle is done, the refresh-request flip-flop is cleared using an async clear input. It's unlikely that a refresh request is skipped, since a CPU cycle and one refresh cycle take less than 4 µs.

I also considered that someday I might want to add a floating-point processor to the system, but I didn't want to clutter the design. So, I added a CPU bus instead.

This bus essentially brings all the CPU pins into a 96-pin DIN connector. There's also a signal that inhibits the onboard decoder and enables the CPU card to decode address spaces which shadow the onboard resources.

MOTHERBOARD DESIGN

The wire-wrap version of the motherboard was done on a special wire-wrap proto board we developed at IUCS for chip testing and system-level prototyping. This board-the Logic Engine—contains a parallel port interface that enables a PC to set and read 128 bits of I/O, program timers, and so forth.

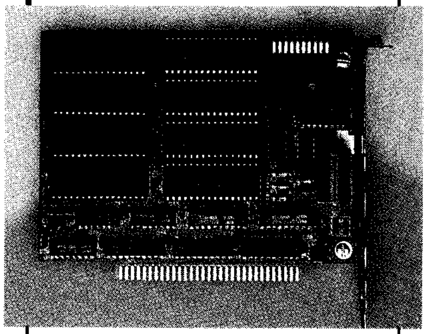
For this project, we didn't use this interface and build the system as a stand-alone prototype. I actually created two wire-wrap prototypes.

The first was a 68020 system that included a prototype ISA-bus interface to demonstrate the feasibility of writing software that could interface with PC peripherals (e.g., the VGA graphics card and floppy interface).

The 68020 system was short-lived, since it only implemented 8-bit devices

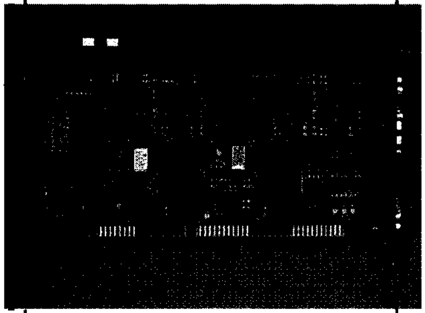
VMAX®

QUALITY PRODUCTS
RESPONSIBLE SERVICE
RELIABLE DELIVERY



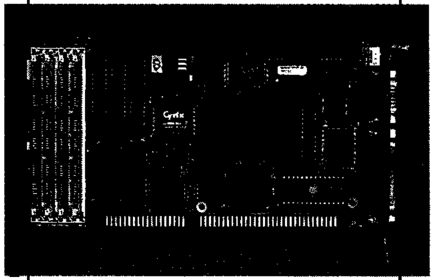
ENHANCED SOLID STATE DRIVE — $89.00

5 Flavors: 2 or 3 Drives, 2M, 4M, 32M, 2M with Post LEDs
Either Drive Boots, FFS included
1/2 Card, XT Height, Customs too



MULTI PORT I/O BOARD— $90.00

4 Serial Ports, 2 Bi-Parallel Ports, 2 Drive IDE, 2 Floppy Interface, 16 Bit Standard, 8 Bit Adaptable, Low Profile 4.2", Full Cable Set



486 66MHz SINGLE CARD COMPUTER — $335"

Up to 2.5MegFlash/Sram drive
Compact-XT height ½ card size
Industry Standard PC-l 04 port
L2 cache to 64K—DRAM to 16Meg
Dual IDE/Floppy connectors

All Tempustech VMAX® products are PC Bus Compatible. Made in the U.S.A., 30 Day Money Back Guarantee
*Qty 1, Qty breaks start at 5 pieces.

TEMPUSTECH, INC.

TEL: (800) 634-0701
FAX: (941) 643-4981
E-Mail: cpusales@tempustech.com
I-Net: www.tempustech.com

Fax for fast response! 295 Airport Road Naples, FL 34104

#135

adrl	adr0	sizl	siz0	cas3	cas2	cas1	cas0
0	0	0	0	1	1	1	1
0	1	0	0	0	1	1	1
1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1
0	0	0	1	1	0	0	0
0	1	0	1	0	1	0	0
1	0	0	1	0	0	1	0
1	1	0	1	0	0	0	1
0	0	1	0	1	1	0	0
0	1	1	0	0	1	1	0
1	0	1	0	0	0	1	1
1	1	1	0	0	0	0	1
0	0	1	1	1	1	1	0
0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1
1	1	1	1	0	0	0	1

Table **3**—*This truth table for the byte-selection* circuitry *for the DRAM interface is **implemented** in the **PLD**. For **each** combination of address location and **size** request, the appropriate CAS **selects fo** the **DRAM** must **be** generafed.*

and had no DRAM. After testing some of my assumptions about the ISA-bus interface on this system, I proceeded to design and build a wire-wrap prototype of a 68030 system.

I documented the design as I went along and was able to design in steps that made the design more modular and kept my confidence high that I could get everything to work.

Contrast this approach to designing the whole system and then building a wire-wrap prototype only to discover that many things you assumed would work didn't. It's much easier to debug an incremental design.

Once the design was working and verified to match the documentation, I passed it on for PCB layout. The board was laid out to ensure that all the mounting holes, ISA-bus slots, and power connectors could fit in a standard baby-AT case.

While this choice enables us to use ultra-cheap PC cases with power supplies, I think it makes for an unattractive machine. It looks just like a PC! This situation has, of course, evoked many funny situations when an unsuspecting user gets a totally foreign boot-monitor prompt instead of the familiar Windows look.

After the PCB layout was complete, we spent a nervous two weeks waiting for the first samples to come back from the board house. We distracted ourselves by working out the logistics of ordering components to build 20 of these systems and by putting some finishing touches on the software.

The first board worked, although it had a few erratas that we incorporated in the final layout. To date, the final layout has only had to have one errata in three years of service.

HARDWARE WRAPUP

Even though the design is fairly complex, we ended up with 20 out of 20 working systems. This process shows that wire-wrap prototype designs running at 16 MHz are no major challenge as long as you use well-designed prototype boards that have a ground plane.

Next month, I'll talk about the monitor software and how it boots, as well as some of the issues involved with programming ISA-bus peripherals in a 68k. ❑

*Ingo **Cyliax is a research engineer in the Analog VLSI and Robotics Lab and teaches hardware design in the computer-science department at Indiana University. He also does software and hardware development with Derivation Systems, a San Diego-based formal-synthesis company. You may reach** Ingo **at** cyliax@EZComm.com.*

SOURCES

Schematics and PCB artwork for this system can be found at <ftp.cs.indiana.edu/pub/goo/mc68030>.

REFERENCES

L.C. Eggbrecht, ***Interfacing to the IBM Personal Computer,*** SAMS, Carmel, IN, 1990.
Motorola, MC68030 ***Enhanced** 32-**bit Microprocessor User's Manual,*** Phoenix, AZ, 1990.
Motorola, MC68901 ***Multi-Function Peripheral,*** Phoenix, AZ, 1988.
Motorola, ***Motorola Memory Databook,*** Phoenix, AZ, 1989.
E. Solari, ***ISA & EISA: Theory and Operation,*** Annabooks, San Diego, CA, 1992.

I R S

416 Very Useful
417 Moderately Useful
418 Not Useful

MICRO SERIES

Ingo Cyliax

MC68030 Workstation

The Boot PROM Monitor & Device Drivers

Part 2 of 3

Stranded by Motorola's decision to discontinue an affordable MC68000 eval board, Ingo built a 68k platform to teach programming in assembly. After covering the hardware last month, he looks at the boot-PROM monitor and how to build device drivers.

Last month, I described the hardware architecture for a 68030-based workstation used in our computer-architecture lab at Indiana University. In this installment, I want to discuss the onboard monitor software on this machine as well as some issues that arise when writing device drivers for I/O devices that are onboard or attached to this system's ISA bus.

But, let's start with what the monitor does. I'll cover how it initializes all the I/O devices and the processor to a consistent, usable state.

Once this is done, I'll show you how the monitor implements a simple command-line-based user interface. This enables the student to download and boot code from the network; boot from a floppy or IDE drive; view and change memory, I/O, and processor registers; and debug programs.

In our typical lab configuration, these machines have a VGA card and monitor, floppy and IDE disks, an AT keyboard, two RS-232 serial ports, one parallel port, and an Ethernet card. The monitor supports all these devices.

MONITOR

The monitor resides in the system's boot PROM and is activated at reset. It was written entirely from scratch—

mostly in C, with some of the start-up and exception processing routines in 68k assembly language.

The complete monitor may seem fairly Spartan since it doesn't implement fancy command-line editing, history functions, or even a programming/scripting language. However, if you consider that it fits in a single 32-Kb EPROM, you'll realize it gets a fair amount of bang for the buck.

Let's look at the monitor's operation and features in more detail. At reset, the 68030 expects to load the system stack pointer at location 00000000 (hex) and the initial PC at location 00000004. Besides these vectors, many exception vectors reside below 00000400.

The reset vector causes the CPU to start executing the cold-reset start-up code. The first thing the start-up code does is to disable interrupts. While the interrupts are already off after a real reset, this task enables any code to use the reset vector (which is in a known location) to cause software to give up control and reset the system.

Once the interrupts are off, the monitor checks whether it's already running from DRAM by checking the actual running address in the PC against the desired address the monitor was linked for. If the addresses don't match, the monitor copies the desired address to DRAM (at physical address 80000000 [hex]) and resumes operation there.

Listing I-i *de_cmd() tries to read or write a block of data from the IDE hard disk by first seeking the location and then reading or writing the disk data through the data port.*

```
ide_cmd(un,cmd,hd,cy,se,buf,len)
int un, cmd, hd, cy, se;
short *buf;
int len; {
  int i,s,st;              /* busy? */
  while(((st=inp(IDE(7)) & (SBSY|SRDY))) != SRDY);
    outp(IDE(2),len/512); /* sec count */
    outp(IDE(3),se);       /* sector */
    outp(IDE(4),cy&0xff); /* cylinder low byte */
    outp(IDE(5),cy>>8);    /* cylinder high byte */
    outp(IDE(6),0xa0 | (un <<4) |
          (hd & 0xf) );    /* secsize,unit,head */
    /* wait for drive to become ready */
    while(!(inp(IDE(7)) & SRDY));
      outp(IDE(7),cmd);    /* command */
      /* do a write */
      if(cmd == 0x30){     /* wait for drive to accept data */
        while(!(inp(IDE(7)) & SDRQ));
        len = len / 2;
        i = 0;
        while(i < len){
          outpw(IDE(0),*buf++);
          i++; }
        i = i*2;           /* wait for completion of write here? *
        if((st = inp(IDE(7))) & SBSY){
          while((st = inp(IDE(7))) & SBSY);}
        if(st & 0x01){     /* an error occured */
          printf("ide_cmd: Error status %x error %x\n",
                  st,inp(IDE(1)));
          return(-1); } }
    else{
      if((st = inp(IDE(7))) & SBSY){
        while((st = inp(IDE(7))) & SBSY);}
      if(st & 0x01){       /* an error occured */
        printf("ide_cmd:Error status %x error %x\n",st,inp(IDE(1)));
        return(-1); }
      i = st:
      if(len){             /* is drive ready to send data? */
        while(!(inp(IDE(7)) & SDRQ));
          len = len / 2;
          i = 0;
          while(i < len){
            *buf++ = inpw(IDE(0));
            i++; }
          i = i*2;}}
  return(i); }
```

The code also changes the base of the exception vector table to start in DRAM. There are several reasons for this.

Since the boot PROM is an 8-bit device, execution speed is enhanced if it's copied and run from 32-bit-wide DRAM. Also, DRAM makes it easy to alter the exception-vector table and patch the monitor code to add support for new devices and fix bugs if needed.

After the monitor and exception table is relocated, the assembly-language start-up code is done and the C code is called. If it's the first time into the monitor code after a real hardware reset, the monitor initializes all the needed I/O devices and prepares itself to boot from the IDE drive unless the user enters a character to interrupt the auto-boot sequence.

If the auto-boot sequence is interrupted in time, the monitor enters a command-line-based interface so the user can execute monitor routines to boot from an alternate device or debug a program. If the monitor is entered from either a software-initiated reset or an exception that isn't trapped by another program, the auto-boot sequence is bypassed and the command loop entered directly.

The code also prints out the current state of the PC and status register and what exception caused control to be transferred to the monitor. This scenario occurs when a user program encounters an exception (e.g., an address/bus error or divide-by-zero) or when an unexpected interrupt occurs.

Let's look at what it takes to boot the system from one of the boot devices. The monitor is supported for booting from an AT-compatible IDE drive and floppy drive on the ISA bus. It also knows how to boot from Ethernet using one of two possible 8-bit Ethernet card types (also on the ISA bus).

Booting from a disk device is relatively straightforward. The monitor reads the first sector from the floppy or IDE drive to location 80008000 (hex) and extracts the number of blocks and location on the media to start reading the boot image.

The format of the boot sector looks like Table 1. After loading the boot sector, the monitor continues to read

1F0	Data Register
1F1	Error Register
1F2	Sector Count Register
1F3	Sector Number Reqister
1F4	Cylinder Register Low
1F5	Cylinder Register High
1F6	Drive/Head Reqister
1F7	Status/Command Register

Figure 1—*To perform* a *disk I/O* operation, *the sector location* address *must be* programmed before a command can be given. The *Data register is 16* bits, whereas *the* resf of *the* registers can be *accessed via 16- or 8-bit I/O instructions.*

the boot image until all specified blocks are read.

Once the boot image is in memory, the monitor does the equivalent of a subroutine call starting at address 80008000. There, a branch instruction should cause execution to continue somewhere in the boot image, depending on the application.

The Ethernet boot code is similar. It uses standard Internet protocols to determine the boot image's location and filename on the network and uses another standard protocol to download the image to location 80008000.

Once loaded, it executes a subroutine call to the start of the boot image. This way, the same boot image can be loaded from floppy/IDE and Ethernet without changes to the image. In Part 3, I'll discuss the protocols used in loading the boot image over the Ethernet.

Once control is given to the boot image (usually some kind of operating system), you only get back to the monitor if an unexpected exception occurs that the OS can't handle or if the OS gives control back to the monitor.

When students are learning to write code for this system, we need to load code images and allow debugging from the monitor code. The student normally assembles and compiles code on a workstation and manages to get the code image into system memory for debugging. (More on how the program image gets there next time.)

The monitor's debugging facilities are Spartan so the student is exposed to a different environment than they're used to. Since the lab's purpose is to teach computer-science students computer architecture, we want them to see low-level machine constructs. We hope they learn how abstract programming constructs in C/C++ programs are implemented.

They'd also never experience the joy of doing a manual stack trace to figure out where their program bombed. Remember, this lab is as close as most computer-science students get to programming real hardware these days.

After they load a code segment into memory, they can use the monitor's `go` command to start executing at a specified address. They may also set a breakpoint anywhere in their code.

When the processor encounters a breakpoint, which is implemented with a software interrupt (`trap`) instruction, an exception control jumps back to the monitor where the context of the running code is saved. The user can then examine and modify registers as well as memory and continue executing their code. The user can also trace their program's execution, for which the 68030 has a special hardware facility.

Another useful feature of the monitor PROM is the ability to save more than context. This feature enables the student to save the context of an OS, if active, and the context of the program being debugged. There are two default contexts for this purpose and a shortcut command that restores the OS context and continues execution.

DEVICE DRIVERS

Of course, the monitor also has to implement device drivers for several devices that are present. I already discussed the boot sequence, so let's look at some of the low-level routines used to implement the device driver for the boot devices.

The IDE controller contained on the disk drive uses a parallel port on the IDE interface card to connect to the ISA bus. Last month, I talked about how the ISA-bus interface is implemented.

To understand how the software works, you only need to know that the ISA bus is divided into four different memory areas depending on what type of ISA-bus bus cycle is required. These are 8- and 16-bit memory space access and 8- and 16-bit I/O access.

The IDE interface uses 8-bit I/O accesses to communicate with its registers. The register map and I/O locations for the first IDE drive can be seen in Figure 1. The data is transferred through a single 16-bit I/O register using programmed I/O.

Listing 1 starts the sequence of steps necessary for reading from and writing to the IDE disk. The routine `ide_cmd` sets up the IDE controller with the desired disk address by programming the appropriate register.

When reading, initiate the command and poll the status register until the disk drive is idle. Then, check the status register for any errors. If none are found, read the data from the data register.

For writing, the data should be transferred through the data register before issuing the `write`. The data is simply read or written to the data register until the required amount is transferred.

The floppy driver is a bit more complicated since some timing issues are involved. The floppy driver is another reason why we want to execute from 32-bit memory and why the monitor was moved to DRAM.

It would have been possible to write a floppy driver that ran from 8-bit memory. But, it would have required extensive assembly-language programming to make it work fast enough during the data-transfer phase.

The data-transfer phase involves the most critical timing. There's only a single register to buffer the data to and from the floppy disk.

Address	Description
000 (hex)	Branch to start
004 (hex)	Starting block number of boot image
008 (hex)	Number of blocks in boot image
00c–1ff	Not used by monitor

Table 1 --The *boot sector contains information for the* boot loader about *where to find the boot image and how long if is.*

3F2	Operations Control Register
3F4	Master Status Register
3F5	Data Register
3F7	Data-Rate Register

Figure 2—*The* Operations register *resets the controller and* spindle-motor *control, while the* Data-Rate register programs *the* data rate *and density desired. Most of the control is accomplished by sending commands to and reading status packets from the Data register.*

If we're too slow in reading the data, the data coming from the floppy disk overruns the data register, causing an error. If we're too slow in writing the data to the floppy, the data register is empty when data needs to be serialized to the floppy disk, resulting in a data underrun. In a real PC architecture, the DMA offloads the CPU from these timing requirements.

Motor control brings up another complication on the floppy controller. The floppy drive has a motor to drive the spindle. However, if it runs all the time, the media can become worn out.

So, the motor needs to be started before any floppy operation can proceed. It also needs to be stopped sometime after the last floppy operation is done.

Other than that, the floppy works mostly like the IDE drive. The driver calculates the sector number, cylinder, and head number from the block number requested, sets these parameters into the appropriate register, and requests a seek operation.

Listing 2-f *7oppy_read() seeks **the** specified location on the disk and then transfers **the** data from **the** floppy data port to **memory.** Since **the** 68030 motherboard doesn't implement **DMA,** programmed **I/O** is required **to transfer** the data.*

```
floppy_read(drv,block,buf,len)
int drv;
unsigned block;
register unsigned char *buf;
int len; {
  int sec;
  int trk;
  int hd;
  register int n;
  unsigned char resp[7];
  unsigned short x;
  register unsigned char i;
  floppy_seek(drv,block);
  trk = block / 18;
  sec = (block % 9);
  hd = (block / 9) % 2:
  len = (len > ((9-sec)*512))?(9-sec)*512:len;
  n = len;
  fdc_out(0x66);   /* read command MFM,SK */
  fdc_out(drv|hd<<2); /* drive & head select */
  fdc_out(trk);    /* cylinder */
  fdc_out(hd);     /* head address */
  fdc_out(sec+1); /* sector address */
  fdc_out(2);      /* sector size 512 */
  fdc_out(9);      /* end of track */
  fdc_out(0x1b);   /* gap */
  fdc_out(0xff);   /* data length, only if sector size == 0 */
  while(n){        /* anything happening? */
    while(!((i = inp(FDC(4))) & 0x80)); /* still executing */
      if((i & 0xe0) == 0xe0){
        *buf = inp(FDC(5));
        buf++;
        n--; }     /* oops, not executing, must be response data */
      else if((i & 0xe0) == 0xc0)
        break; }
    for(x=0;x<7;x++){
      while((inp(FDC(4)) & 0xe0) != 0xc0);
        fdc_in(&resp[x]); }
    if((resp[0] & 0xc0) || n){
      if(!n && (resp[0] & 0xf8) == 0x40 &&
        (resp[1] == 0x10|| resp[1] == 0x80) && resp[2] == 0x00)
        got0 out;
      puthex2(i);
      putchar(' ');
      for(x=0;x<7;x++)
        puthex2(resp[x]);
      putchar(' ');
      puthex4(n);
      putchar('\n');
    return(-1); }
  out:
    return(len  n); }
```

Tight Budget '51 Emulation

8051 Family Emulator is truly Low Cost!

The DrylCE Plus is a modular emulator designed to get maximum flexibility and functionality for your hard earned dollar. 805family processor pods that are low in price. Features include: Execute to breakpoint, Line-by-Line Assembler, Disassembler, SFR access, Fill, Set and Dump Internal or External RAM and Code, Dump Registers, and more. The DrylCE Plus base unit is priced at a meager $299, and most pods run only an additional **$149.** Pods are available to support the 8031/2, 8751/2, 80C154, 80C451, 80C535, 80C537, 80C550, 80C552/62, 80C652, 80C85 1, 80C320 and more. Interface through your serial port and a comm program. Call for a brochure or use INTERNET. We're at **info@hte.com** or **www.hte.com.**

Tighter budget? How about *$149* emulation?

Our **$149** DrylCE model is what you're looking for. **Not** an evaluation board - much more powerful. Same features as the DrylCEPlus, but limited to just the 803 1/32 processor.

You can afford it!

So, if you're still doing the **UV Waltz** (Burn-2-3, Erase-2-3). or debugging through the limited window ROM emulators give, ***call us now*** for relief! Our customers say our products are still the **best** Performance/Price emulators available!

Look into our Single Board Computer solutions, too!

HTE
HiTech Equipment Corp.
9672 Via Excelencia
San Diego, CA 92126
[Fax: (619) 530-1458]

Since 1983

(619) 566-I 892

VISA MasterCard

Internet e-mail: info@hte.com
World Wide Web: **www.hte.com**

#136

Once completed, the data can be read or written to the data register. But, you need to make sure the floppy is ready by read-testing the data-request status bit. Listing 2 shows the code for reading one sector from the floppy, while Figure 2 shows the I/O register layout.

ETHERNET CONTROLLERS

Now that you know how to control the floppy and IDE drive, what about an Ethernet device? Ethernet controllers are a bit more complex even at the hardware level. Let's examine the code that sends and receives a packet on one of the controllers.

Ethernet runs at 10 Mbps, which turns out to be a little better than 1 MBps. Since many Ethernet packets are not for this node, it's wasteful to get the processor to read every packet. It would keep an 8-bit ISA-bus system 100% utilized whenever there's traffic.

To reduce the I/O requirements, Ethernet cards employ some kind of buffer to store at least one received packet. If this packet isn't destined for this station, the write point to the packet memory is simply reset.

Once a packet is received, the Ethernet card signals the processor with an interrupt request or a bit in the status register that it's done. Listing 3 shows the code for receiving a Ethernet packet from an Ethernet card.

Sending packets is easy. The CPU copies the packet to be transmitted to the card's transmit buffer and tells it to start. If the Ethernet card transmits the buffer, it simply indicates that it's done and interrupts.

Transmitting the packet can take a long time. The card has to find an idle period on the Ethernet and then try to transmit. If another station tries to transmit, a collision occurs and both cards retry after a randomized timeout.

These actions are transparent to the software, but you need to be aware of them, since it may take a while before the transmit request completes. Check out the transmit routine in Listing 4.

KEYBOARD

In Part 1, I talked about the physical interface of the keyboard, which involved how the data and clock interface to the USART in the 68901 MFP chip.

Listing 3-we8003 read() pulls a received Ethernet frame from the Ethernet card's infernal memory. Since frames are stored in a circular queue on the card, pull the oldest frame and advance the tail pointer for another.

```
we8003read(ifp,buf,count,timer)
struct ifconfig *ifp;
unsigned char *buf;
int  count:
int *timer; {
  int len;
  unsigned long x;
  struct we8003 *we = (struct we8003 *)ifp->ifc_addr;
  unsigned char *mem;
  int head,tail,i,stat,resid;
  #define ltail   (*(unsigned char *)(ifp->ifc_addr2 + 0x5ff))
  len = 0:
  x = 1000* *timer;
  mem = (unsigned char *)ifp->ifc_addr2;
  while(x && !len){
    while(!((stat = we->we_p0_isr) & 0x05) && x)
      x--;
    if(!x) break:
      if(stat & 0x04){
        i = we->we_nic_reg[13];
        i = we->we_nic_reg[14];
        i  = we->we_nic_reg[15];}
  tail = we->we_p0_bnry;       /* first packet */
  we->we_cmd = 0x42:           /* ps=l,  running */
  head = we->we_pl_curr;       /* next free */
  we->we_cmd = 0x02:           /* ps=0, running */
  /* printf("isr %x hd %x tl  %x lt %x\n",stat,head,tail,ltail); */
  if(ltail)
    tail = ltail;
  while(tail != head){
    i = tail<<8;
    len = mem[i+3]<<8;
    len |= mem[i+2];
    tail = mem[i+1];
    /*printf("  stat %x nxt %x len %x\n", mem[i],mem[i+1],len); */
    len -= 4;                  /* lop off CRC */
    len = len > count ? count : len;
    /* need to check for wrap--ped buffers */
    resid = WE_MEMSIZ-i-4;
    if(resid < len){
      bcopy(&mem[i+4],buf,resid);
      bcopy(&mem[0x600],&buf[resid],len-resid);  }
    else
      bcopy(&mem[i+4],buf,len);  }
    ltail = tail:
    tail--:
    if(tail < 6)
      tail = 0xlf;
    we->we_p0_bnry = tail:
    we->we_p0_isr = 0xff;  }
  return(len);  }
```

Listing 4—Compared to reading Ethernet frames, sending them is straightforward. Just copy it to the Ethernet card's internal memory, and tell it to go. Since the monitor uses polling, wait here to make sure if was sent OK.

```
we8003write(ifp,buf,count)
struct ifconfig *ifp;
unsigned char *buf;
int count; {
  struct we8003 *we = (struct we8003 *)ifp->ifc_addr;
  unsigned char *mem = (unsigned char *)ifp->ifc_addr2;
  count = count > 60 ? count : 60;
  bcopy(buf,mem,count);
  we->we_p0_tbcr0 = count & 0xff;
  we->we_p0_tbcrl = count >> 8;
  we->we_cmd = 0x06;        /* transmit, running */
  while(!(we->we_p0_isr & 0x0a));
    we->we_p0_isr = 0x0a;
  return(count);  }
```

On the software side, the interface looks like a standard USART interface.

When a byte arrives from the keyboard, the receive-buffer fill bit gets set in the status register and an interrupt is generated when enabled. Reading the received byte from the data register resets the receive-buffer full condition and readies the USART for another byte.

It seems simple, but the received byte is only the scan code from the keyboard or, worse, one byte of a multiple-byte message. What the monitor really needs is an ASCII character that responds to the keycap legend on the keyboard.

The keyboard driver achieves this by looking up the scan code-one for each key-in a character table. The driver also tracks the state of the keyboard shift and control keys, which modify the ASCII code. Listing 5 explains all.

Listing 5—*The keyboard routine pulls bytes from the USART in the MC68901 (MFP).* ***These bytes are the scan codes*** **from** *the* **keyboard** *and need to be translated info ASCII codes using a look-up fable (i.e., co de tab [ ]). The receive routine a/so tracks the state of the shift/capslock and control keys.*

```
kbd_getc() {
  unsigned char c;
  while(!(c=kbd_stat())) ;
    return(c); }
kbd_stat() {
  unsigned char *port = KBD:
  unsigned char x,ox;
  extern struct codetab codetab[];
  static int brk=0;
  if(!(port[0x15] & 0x80))
    return(0);
  x = port[0x17];
  if(x == SC-BREAK) {
    brk = 1:
    return(0); }
  if(x > 0x7f){
    brk = 0;
    return(0); }
  if(brk){
    brk = 0;
    ox = codetab[x].code;
    switch(ox){
      case CC_LSHFT
      case CC_RSHFT: shft = 0: break;
      case CC_CTRL: ctrl = 0;  break:
      default: break; }
    return(0); }
  ox = codetab[x].code;
  switch(ox){
    case CC_LSHFT:
    case CC_RSHFT:    shft = 1; break;
    case CC_CTRL:     ctrl = 1;  break;
    case CC-CAPS:     caps = caps?0:1; break;
    default: if(ctrl)
               return(codetab[x].cchar);
             if(shft|| caps)
               return(codetab[x].schar);
             return(codetab[x].uchar); }
  return(0); }
```

VGA DRIVER

Of course, I saved the hardest until last-the VGA driver. It's hard because each VGA chip and card has a different low-level configuration register set.

In the PC world, this situation is handled by calling the appropriate initialization and mode change routines from the BIOS on the card itself. The code is usually proprietary, and I'd need an Intel instruction emulator for my system to execute the routines.

I ended up writing a driver for only a limited number of possible VGA chipsets, including the Paradise and Western Digital 8-bit VGA chipsets for which I already had extensive datasheets. Other chips/cards may also work if they behave like a Western Digital chip.

The monitor only needs a character-based interface, so I initialize the VGA chip to its CGA mode, which emulates the 6845-based CGA card. The monitor then has a character-based frame buffer with 8 KB of video RAM. To speed up scrolling, the monitor's character-output routine uses the extra display memory to page the display.

UNTIL NEXT MONTH...

I hope I've given you an idea of how involved even a simple debugging monitor can become-without even trying.

Next month, I'll discuss the software-development environment the students and I use to write software. I'll also tell you about some of the things we've accomplished with this system. ❑

Ingo ***Cyliax is a research engineer in the Analog VLSI and Robotics Lab and teaches hardware design in the computer-science department at*** *Indiana* ***University. He also does software and hardware development with*** *Derivation* ***Systems, a San Diego-based formal-synthesis company. You may reach*** *Ingo* **at** *cyliax@EZComm.com.*

REFERENCES

Motorola, ***Programmer's Reference*** *Manual,* Motorola Literature Distribution, Phoenix AZ, 1992.

Western Digital, 1992 ***Devices,*** *Sys-***tem Logic, Imaging, Storage,** Western Digital Corp., Irvine CA, 1992.

SOFTWARE

Complete source code for the monitor can be found at <ftp.cs.indiana.edu/pub/goo/mc68030>.

SOURCE

MC68*xxx*
Motorola
MCU Information Line
P.O. Box ***13026***
Austin, TX 78711-3026
(512) 328-2268
Fax: (512) 891-4465
freeware.aus.sps.mot.com

I R S

425 Very Useful
426 Moderately Useful
427 Not Useful

MC68030 Workstation

MICRO SERIES

Ingo Cyliax

Cross-Development Environment and Downloading

Part 3 of 3

Ingo finishes his three-part series on pulling together an affordable 68000 development board. The final step—software—is based on tools freely available on the Internet. He can even boot the board over the network.

At a university, it's sometimes easier to justify spending money on hardware than software. And besides, people in computer-science departments tend to pride themselves in their ability to roll their own.

But, I didn't fancy spending the next five years writing and supporting yet another C compiler. Instead, I opted to look for freely available tools on the 'Net.

Several C compilers and 68000 assemblers are out there, so in this article, I discuss some of the packages I used. Of course, I still had plenty to do to set up a suitable environment, so I wasn't at all deprived of the experience to roll my own.

In this final installment of the MC68030 series, I start out with a look at some software-development tools and how network booting works. I also discuss some applications that were done on the MC68030 system described in Parts 1 and 2. One of the tools is a network application that permits students to log in to a Unix machine to edit and compile programs and then download them for execution and debugging.

COMPILERS AND ASSEMBLERS

The MC68030 comes from a long line of 68000 chips that was first introduced about 17 years ago. Naturally, a large collection of software has accumulated over the years. There are many assemblers, and almost all programming languages have been implemented on the 68000.

Originally, the computer-science faculty at Indiana University intended for students to program this system in assembly language only. For this, I chose asm68, which was written by Paul McKee.

This assembler is a stand-alone system that takes Motorola-format assembly-language files and generates a Motorola S-record hex file. It also generates a traditional assembly listing and a symbol table file.

asm68 comes in source-code form and is fairly easy to compile for other systems, since it doesn't require many services from the OS. I've built this assembler under Minix and Unix, and students also have built it under DOS on their PCs at home. Being able to run it under Unix lets students edit their source files in the familiar programming environment of our network of Sun workstations.

While asm68 is very compact and enables the students to explore architectural features of the 68030 system in lab, it doesn't allow multimodule programs.

Multimodule programs are when the student can use modules developed in an earlier lab (e.g., their own keyboard driver) and link them into a more sophisticated program. With asm68, the student has to cut and paste everything into one monolithic file and assemble it.

Besides the limitation of single-file modules, the instructors thought students would benefit from writing some of the high-level functionality in C, while maintaining exposure to the hardware interface through assembly-language modules. Once the low-level modules were written, the students would have more flexibility in writing more complex programs.

By mixing C and assembly language, students gain some insight in how high-level language constructs are implemented on a machine like the MC68030.

Listing 1--This *BOOTP* table *resides on the server. It contains the mapping between the Ethernet and Internet* addresses *as well as the name of the boot image to download.*

```
# Legend:
# first field - hostname (should be full domain name)
# hd - home directory
# bf - bootfile
# cs - cookie servers
# ds - domain name servers
# gw - gateways
# ha - hardware address
# ht - hardware type
# im - impress servers
# ip - host IP address
# lg - log servers
# lp - LPR servers
# ns - IEN-116 name servers
# rl - resource location protocol servers
# sm - subnet mask
# tc - template host (points to similar host entry)
# to - time offset (seconds)
# ts - time servers
#
# Be careful about including backslashes where they're needed. Weird
# things can happen when a backslash is omitted where one is intended
#
default:sm=255.255.255.0:hd=/tftpboot:bf=null:\
  :ds=0.0.0.0:ns=0.0.0.0:ts=0.0.0.0:to=18000:
rmc:tc=default:ip=198.88.16.3:ht=ethernet:ha=0000C034D810:bf=net030
reset:tc=default:ip=198.88.16.4:ht=ethernet:ha=0000C022DC10:bf=net030
rw:tc=default:ip=198.88.16.5:ht=ethernet:ha=0000C0945D14:bf=net030
ocs:tc=default:ip=198.88.16.6:ht=ethernet:ha=02608C754292:bf=net030
ipl2:tc=default:ip=198.88.16.7:ht=ethernet:ha=0000C0FE6214:bf=net030
ipl0:tc=default:ip=198.88.16.9:ht=ethernet:ha=0000C0138314:bf=net030
ipend:tc=default:ip=198.88.16.10:ht=ethernet:ha=0000C0DEE710:bf=net03
halt:tc=default:ip=198.88.16.11:ht=ethernet:ha=0000C0CE5016:bf=net030
fc1:tc=default:ip=198.88.16.13:ht=ethernet:ha=0000C0AC9A14:bf=net030
ecs:tc=default:ip=198.88.16.15:ht=ethernet:ha=0000C08D6514:bf=net030
ds:tc=default:ip=198.88.16.18:ht=ethernet:ha=0000C0E15E14:bf=net030
siz1:tc=default:ip=198.88.16.1:ht=ethernet:ha=02608C172716:bf=net030
siz0:tc=default:ip=198.88.16.2:ht=ethernet:ha=02608C751977:bf=net030
ipl1:tc=default:ip=198.88.16.8:ht=ethernet:ha=02608C754301:bf=net030
#fc2:tc=default:ip=198.88.16.12:ht=ethernet:ha=02608C754301:bf=net030
fc0:tc=default:ip=198.88.16.14:ht=ethernet:ha=02608C285527:bf=net030
```

GNU TO THE RESCUE

Luckily, since I was already using the GNU-C environment to develop the monitor for this system and other applications, this was a no brainer. I could use the same development environment being used to develop the firmware for the system in the students' labs.

Well, that was easier said than done. Let's first take a look at GNU-C (the compiler) and GNU-as (the assembler) to write software for the stand-alone software and operating system.

The GNU-C compiler has been around for quite some time. It was originally written by Richard Stahlman of the free-software foundation (FSF). In fact GNU, which stands for "GNU is Not Unix," is a whole tool suite of utilities that the FSF develops and gives away free.

Other popular utilities besides GNU-C are Emacs and Ghostscript. Due to the open philosophy, many people and organizations contribute to these tools by porting them to new environments, adding features and functionality, as well as fixing bugs.

The current GNU-C supports too many architectures to list here. It can be used as a system compiler for a particular architecture/operating system and, in some cases, is even better than the vendor-supplied compiler.

For example, Linux and FreeBSD, both freely available Unix-like operating systems, use GNU-C as the system compiler. Also, many software packages can compile with GNU-C, which makes it sort of a standard C dialect across many platforms.

One feature in particular that's not as well known to GNU-C users also makes it a good tool for embedded-systems programming. It enables GNU-C to be built as a cross-compiler and -assembler for many architectures.

I use GNU-C mostly as a cross-compiler for the 68000 and ColdFire, Motorola's new architecture (see Tom Cantrell's "Motorola Lights ColdFire," *INK* 77), on Sun workstations. However, the 68000 cross-compiler can be built for almost any OS.

As a C compiler, it behaves as you'd expect, compiling old-style C as well as ANSI C into an object module. It can also compile C++ and Objective C and even provides a pretty complete run-time support for the 68000 architecture (e.g., floating-point-math emulation libraries). GNU-C also has many optimization switches and can optimize code for almost all the 68000 variants.

GNU-C is quite amazing, but the assembler is interesting, too. GNU-as can be configured to work as a 68000 assembler, accepting both Motorola and MIT syntaxes-the two prevailing assembler syntaxes for the 68000 family. It can be configured to generate different object file formats (e.g., COFF), which are all supported by the GNU linker.

There is also a source-level debugger which has remote debugging capability. So at this point, I can compile C files, assemble 68000 assembly-language modules, and link them all together into a program on my Unix workstation. Nice, but how do I get it into the 68030 system?

There are several options. I can generate a Motorola S-record hex file and download it over one of the serial ports on the 68030 system. But, this method takes a long time for any program bigger than a few kilobytes.

I can also extract an image of the program and its data and copy it to a floppy that can be used to boot. This option is pretty nice, but it requires a floppy drive on the workstation that can write floppy disks in "raw" format.

One of the fastest and most convenient methods is network booting. In Part 2, I wrote about the features of the monitor on my 68030 system and its ability to boot from the network. Here's how it's done.

NETWORK BOOTING

Without getting too in-depth about Ethernet, let me describe what goes on. Last month, I described the packet-level driver needed to send and receive packets to and from Ethernet.

Each packet sent on the net needs to have a destination address, which in the case of Ethernet, is sometimes called the hardware address. The hardware address is a serial number that's unique for each Ethernet card.

Since hardware addresses are assigned by the manufacturer, they're not so useful when it comes to sending a packet to another machine, unless it happens to be on the same network segment. The hardware address has no information about how to route packets between network segments.

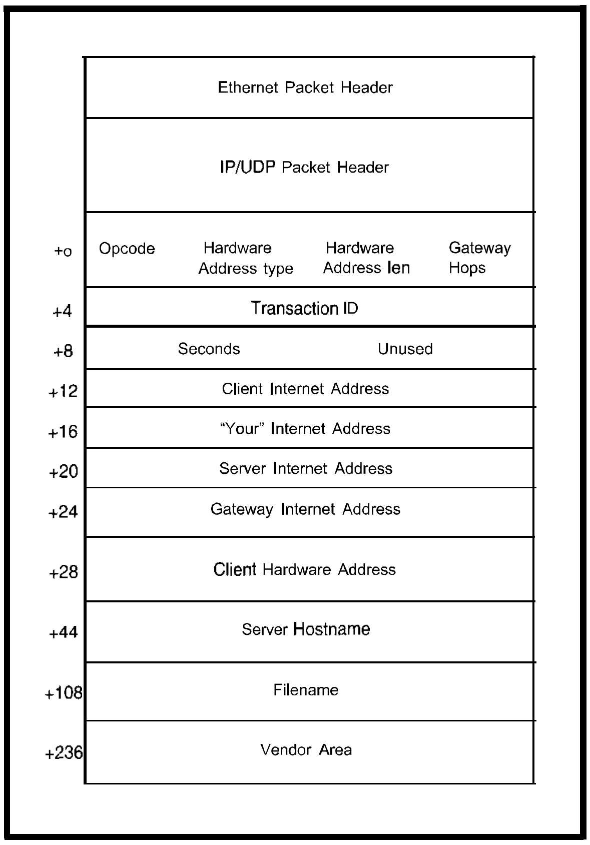


Figure 1—*The client broadcasts a BOOTP request (opcode = 1) on the Ethernet. The server then sends a reply (opcode = 2) to the client with all the information the client needs to boot a file from a server.*

RS-485 NOW!

Standard RS-485 Network Software

Deploy a rugged, proven Cimetrics RS-485 solution in just days. Software, full documentation, development tools. All backed by outstanding service and support from Cimetrics.

- Process Control
- HVAC & Lighting Control
- Point-of-sale
- Transportation
- Fire & Security
- Telecommunication
- Medical Equipment
- Laboratory Data Acquisition

Comprehensive Microcontroller Master & Slave Support

PC (DOS/Win/NT)	68HC05	PIC16C74
8051	68HC11	PIC17C42
8096	68HC16	H8/300H
80C196	68302	TMS370
80C186EB	68332	Z180
	68333	

Cimetrics TECHNOLOGY

55 Temple Place • Boston, MA 02111-1300
Tel: 617-350-7550 • Fax: 617-350-7552
Website: www.cimetrics.com

#135

$200 4A C Compiler?

You heard right. A quality C compiler designed for the 8051 microcontroller family, just $200, including the Intel compatible assembler and linker. A great companion to our fine Single Board Computers, like those below. **CALL NOW!**

552SBC

80C552 a '5 1 Compatible Micro
40 Bits of Digital I/O
8 Channels of **10** Bit A/D
3 Serial Ports (RS-232 or 422/485)
2 Pulse Width Modulation Outputs
6 Capture/Compare Inputs
1 Real Time Clock
64K bytes Static RAM
1+ UVPROM Socket
512 bytes of Serial EEPROM
1 Watchdog
1 Power Fail Interrupt
1 On-Board Power Regulation

Priced at just $299 in single quantities. Call about our 552SBC C Development Kit, just **$448**.

99 MHz 8051!

Our popular 8031SBC can now be shipped with Dallas Semi's hyperactive **DS80C320**, an 8051 on steroids. Averaging 3x faster than the standard 51, your project can really scream! **Call or email for pricing and brochures today!**

Other versions of the **8031SBC** have processors with **on-chip** capture registers, EEPROM, IIC, **A/D** and more. Call or **email** for **a** list!

8031SBC as low as $49

Call for your custom product needs. Quick Response.

HiTech Equipment Corp.
9672 Via Excelencia
San Diego, CA 92126
(Fax: (619) 530-1458)

Since 1983

(619) 566-1892

Internet e-mail: **info@hte.com**
World Wide Web: **www.hte.com**

#137

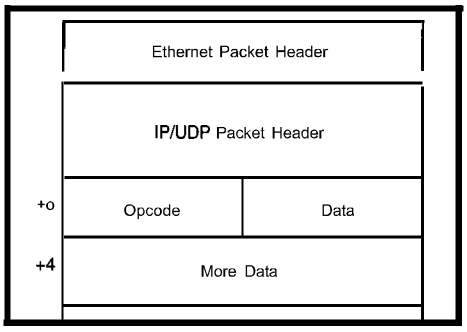


Figure 2—*The* client *sends a read* request packet *(opcode = 1), in which the filename is specified in the data. After the* **server** *responds with an acknowledge (opcode =* **4),** *the client asks for data* **blocks** *(opcode = 2).*

On the Internet, routing is done with the Internet address. An Internet address is composed of two parts-a network number and a host number. I'm not going to discuss routing issues any further here, so let's just say that Internet addresses are assigned by network administrators to make the routing between hosts possible.

But, how does a machine figure out the hardware address of a host it wants to send a packet to? And furthermore, how does the machine discover its own Internet address?

I'll answer the second question first. The machine already knows its own hardware address.

The monitor uses two protocols to do this. The boot protocol (BOOTP) is used to discover the Internet numbers of the machine and server as well as the filename of the boot image to load. Figure 1 depicts a BOOTP packet.

The BOOTP protocol is simple. The monitor fills out a BOOTP request packet and uses the packet driver to transmit it. A special broadcast address ensures that all machines on the segment can see this packet, since it has no idea which machines are present on the local Ethernet segment.

A BOOTP server host has a table that maps the hardware address from which the packet came into an Internet address and a filename. These are then sent back to the BOOTP client that originally broadcasted the request. Listing 1 shows the BOOTP table for our lab.

The client receives the packet and extracts its Internet address, the Internet address of the server, and the filename to download for the boot image. The monitor then uses the trivial file transfer protocol (tftp) to read the file from the boot server.

To discover the server's hardware address, the monitor broadcasts an address resolution protocol (ARP) request packet. This packet basically asks, "Hey! Who out there has Internet address *xx.xx.xx.xx*?"

All hosts on the local segment can hear this packet. If their Internet address matches the one requested, they respond with a packet giving the hardware address. Figure 2 illustrates a tftp packet, and Figure 3 gives you a look at the ARP-packet format.

The boot monitor can now contact the server using its hardware and Internet addresses and download the requested file via tftp. Our BOOTP and tftp servers are usually the same machine (i.e., a Sun workstation).

The tftp protocol is fairly simple. The client requests a certain file and downloads it a packet at time. Each packet has a sequence number, and it is resent if an acknowledge isn't received within a timeout interval.

This network-booting scheme has been around for a while, and Unix workstations from many vendors have the appropriate BOOTP and tftn server programs-as part of their normal software distribution.

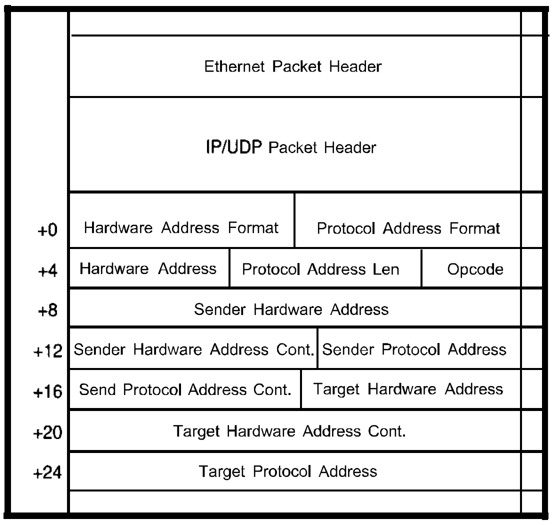


Figure 3—*When a host wants to find the hardware address for another host, it broadcasts a* **request** *packet (opcode = 1). The target host responds by* **filling** *in its hardware address and sending if back as a response (opcode =* **2).** *The protocol address is the* **Internet** *address.*

Even though this boot method is fairly popular and reliable, there's one problem that makes it unsuitable for downloading students' programs in the lab. There's no privacy. Since the boot client has no way to authenticate itself except by its hardware address, the system must have provisions for students to only be able to download their own programs.

Furthermore, tftp has no authentication method, so it essentially acts as a public-access server. Also, the boot monitor has no way for students to log in remotely to edit and compile programs. Something is clearly missing.

Here's where the network monitor comes in. It enables the student to log in to their Unix account to edit, compile, and download their code in a secure manner over the network.

The network monitor is based on the ka9q network OS. This software, originally written by Phil Karn, is used by many amateur radio operators to communicate using packet radio. It implements a TCP/IP protocol stack with several clients like telnet and ftp.

I adapted this program to run on my MC68030 system by stripping out unneeded features and interfacing it to some of the drivers I've written. The students boot the network monitor through the network using the BOOTP and tftp boot process. Once it's running, they can telnet to their Unix account anywhere on campus.

Once they're ready to debug, they can download the image of their program into the 32-Kb SRAM, which is guaranteed to be available on the 68030 system, by using the normal Internet ftp protocol. ftp lets the student log in to their own account and download only their files.

Once their program is downloaded into SRAM, they can interrupt out of the network monitor via a front-panel push button, which sends an NMI, and enter the boot-PROM-based debugging monitor. The student can also resume the network monitor, assuming their code didn't disrupt the saved context of the network monitor.

CLASSWORK

When students take the computer-architecture lab, they've generally only been exposed to an introductory C programming course. So, they have quite a lot to learn.

They start with the basics of 68000 assembly language and architecture by doing exercises in which they fill in missing code segments or explain the state certain registers are in after executing instructions. The first lab exercise consists of logging in to their account and assembling a small example which they then download into the machine using the network.

Once they master a subset of the 68000 assembly language, they quickly take off and start coding more complex programs. One of the first really hard labs is the interrupt lab, in which they have to write an interrupt-driven keyboard driver based on a polled keyboard driver from an earlier lab.

They then also learn to interface C modules to their assembly-language modules which implement the interrupt service routine. The labs' complexity increases until they culminate into the final lab. The final lab consists of taking various interrupt-based I/O drivers and timer routines and implementing a game (e.g., Tetris or Missile Command).

FUTURE DIRECTIONS

One of the things I'd like to do with this system is create a more integrated debugging environment. I might do that by implementing some kind of network-based debugging interface to the 68030 and integrating it with GNU's source-level debugger. Another possibility is to redesign the 68030 system using a newer processor technology that implements a hardware debugging port. Some interesting processors include Motorola's ColdFire and PowerPC.

This article concludes my series on the 68030 system I built for our computer-architecture lab. I hope that I've fueled some interest in the development of open architectures and structures that are suitable for academics and others as well.

Ingo Cyliax is a research engineer in the Analog VLSI and Robotics Lab and teaches hardware design in the computer-science department at Indiana University. He also does software and hardware development with Derivation Systems, a San Diego-based formal-synthesis company. You may reach Ingo at cyliax@EZComm.com.

SOFTWARE

The schematics, PCB artwork, and sources for both monitors are available at <ftp.cs.indiana.edu/pub/goo/mc68030>. The computer-architecture class using the 68030 system for the lab has a Web site at <www.cs.indiana.edu/classes/c335home.html>. To find out more about networking, including RFCs on the protocol discussed, check out <ftp.digital.com/pub/net/info>. Phil Karn's ka9q network OS can be found at <ftp.digital.com/pub/net/ka9q>.

REFERENCES

D. Comer, *Internetworking with TCP/IP, Principles, Protocols and Architecture,* Prentice Hall, Englewood Cliffs, NJ, 1988.

W. Ford and W. Top, *Assembly Language and Systems Programming for the M68000 Family,* D.C. Heath and Co., Lexington, MA, 1989.

A.S. Tannebaum, *Computer Networks,* Prentice Hall, Englewood Cliffs, NJ, 1981.

J.F. Wakerly, *Microcomputer Architecture and Programming, The 68000 Family,* John Wiley & Sons, New York, NY, 1992.

SOURCE

MC68xxx, ColdFire, PowerPC
Motorola
MCU Information Line
P.O. Box 13026
Austin, TX 7871 1-3026
(512) 328-2268
Fax: (512) 891-4465
freeware.aus.sps.mot.com

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful