



The CPU280

When 8 Bits Aren't Enough

By Tilmann Reh

The History

When Zilog first introduced the Z800 MPU in their 1983/84 data book, I was working with a homebrew Z80 system based on ECB-bus EuroCards (a well-established standard here in Europe these days). My system was running with the usual 4 MHz clock, but my CP/M-Plus BIOS already had some very nice advantages. For example, I had developed the AutoFormat system, a technique which supports a wide variety of disk formats automatically. The basic principle is the same as what MS-DOS uses: the disk contains a parameter block which holds all information necessary to process it. Another feature was the automatic installation of and adaptation to various peripherals. Since several some people around here had similar machines, none of them had to configure hardware parameters (as long as they didn't need very special things).

When I had a look at the 'preliminary' specifications of the Z800, I began thinking about some performance improvements for my system. So I waited for this chip to become available. I was still waiting in 1985 when Hitachi's HD 64180 suddenly appeared. This processor had much of what the Z800 was supposed to offer, but it was actually available! So I gave up waiting any longer and developed a 64180-based single board computer. It had the 64180 MPU (with both serial channels used), 32 KB EPROM and 512 KB DRAM, a 765-type floppy controller for up to four drives, and an ECB bus interface on a EuroCard. The clock frequency was 9.216 MHz, which made an effective Z80 speed of about 11 to 12 MHz. Since the ECB bus cannot handle these frequencies, the bus clock was only half the CPU clock (4.6 MHz), with timing signals stretched to meet standard Z80 peripheral requirements. My new 'CPU180' system did the work of three of the older boards: the CPU, the memory, and the floppy controller board.

The new single-board design brought the advantage of direct access to all basic I/O, so the BIOS could rely on these facilities and use them. The CP/M-Plus implementation I wrote for that system was more comfortable than the former Z80 multi-board version. For example, it introduced automatic disk exchange recognition. The high processor performance was amazing compared to the 4 MHz Z80.

Then came the IBM-PCs. At that time CP/M was systematically forced to death here in Europe. The special German computer syndrome became, "it has to be always the newest, biggest, and fastest machine," and the computer magazines followed that slogan. So, after a while I was the only one around here who was still active with CP/M. That was the reason why the CPU180 was built exactly once: I was working with the prototype board, and no one else seemed to be interested. (By the way, Wayne and Paul, if you had known that back in 1985/86, you wouldn't have had to design the YASBEC!)

At the end of 1988 Uwe Herczeg, another 'lone CP/M user' in Germany, placed an ad in a major computer magazine asking for other CP/M users to come forward. This brought the remaining activists together. But most of them were still using 4 or 6 MHz Z80 systems, often also based on ECB-bus EuroCards, and some were very interested in my CPU180 system. So we began thinking about a redesign of that board with state-of-the-art technology. One of those new friends suggested using the Z280 instead of the Z180. Z280? I ordered the data sheet and what did I find but the late incarnation of the old Z800 idea. Unbelievable but true, the Z280 was really available! So we decided to use the Z280, for this CPU has a far more powerful instruction set than the Z180.

The Hardware Design

Basics and Memory

As the hardware and software concepts of the CPU180 had proven themselves very well, all basic principles were carried over to the CPU280 design. As it was with the CPU180, the highest priority design rule was to get the absolute maximum performance out of the MPU with the absolute minimum circuit expense and parts cost. For the printed circuit board that meant using a standard double-sided PCB (no multi-layer) with normal wire thickness (0.25 mm) and no SMD parts at all. Just the good old sort of computer boards: reliable and easy to maintain.

In order to get the full power of the Z280, it must be run in the 16-bit Z-bus mode, with cache and burst-mode enabled, with no RAM wait states, and with the highest possible clock frequency (12.5 MHz). The on-board memory interface is 16 bits wide.

Main memory is built with dynamic RAM. There are eight sockets for 1 M-bit or 4 M-bit chips (with 4 bit data width). Chips must be used in groups of four to fit the 16-bit data bus. So the possible memory capacities are 512 KB

Tilmann Reh is an electronics engineer at the Center for Sensor Systems of the University of Siegen, Germany. In addition, he owns a small company which develops custom specific problem solutions, often using microcontrollers or microprocessors. He has been active with CP/M since 1983, changed to CP/M-Plus in 1985, and has developed a number of ECB-bus boards. Tilmann can be reached by regular mail at 'In der Grossenbach 46, W-5900 Siegen, Germany' or by e-mail (Internet/bitnet) at 'tilmann.reh@hrz.uni-siegen.dbp.de'.

and 1, 2, or 4 MB. This directly accessible memory should be enough for just about any circumstances. The standard configuration is with eight 1 M-bit chips (1 MB total). To save space and because of the availability of exactly pin-compatible 1 and 4 M-bit memory chips, ZIP-cased RAM is used.

Having had very good experience with synchronous DRAM timing (the CPU180 memory worked absolutely error-free for many years), I built a fully synchronous timing chain for the DRAMs again. This uses an HCT175 as a four-bit shift register and a GAL (gate array logic) containing the CAS decoding and selection logic. The DRAM interface also supports the processor's burst mode, using a GAL to sweep the lowest address bits during the burst (we can't use nibble-mode RAM's, as these are not available with 4-bit width!). The RAM timing is designed to meet all specifications of the Z280 at 12.5 MHz and RAMs with 100 ns access time. However, as the prices for 100 ns and 80 ns types are identical, we normally take the latter for safety.

Since the memory interface is far too fast for the ECB bus, external memory is not supported. Additional reasons for this decision were the data bus width (16 bit vs. 8) and the different timing and status signals (Z-bus vs. Z80). Support for external memory with burst-mode is absolutely impossible, as the ECB bus doesn't have strobing signals which could do that job. Interfacing the fast 16-bit internal data bus to the slow 8-bit ECB bus would require a large amount of circuitry, and that would violate our main design rule. So, if someone really needs more than 4 MB RAM, please connect an external I/O-accessed RAM-disk via ECB (DMA support is no problem).

The boot software is placed in two 27C256 or 27C512 EPROMs, so the boot capacity is 64 KB or 128 KB. For easy handling, ordinary 28-pin DIP sockets carry the EPROMs. As this memory usually is needed only during boot-up, burst-mode is not supported, and to accommodate slower memory chips up to three wait states may be added.

The Z280 is able to use different memory timings when accessing the lower or upper half of its 16 MB address space. The EPROM is decoded into the lower half (8 MB, what a waste!) and the RAM into the upper half. This way, it is possible to use the RAM with zero wait-states and burst mode while the EPROM uses up to three wait-states and doesn't support burst mode. Mapping any desired memory configuration to the CPU's logical 64k address space is done with the PMMU (Paged Memory Managing Unit) internal to the Z280.

I/O Basics and Bus Interface

In order to take advantage of widely available Z80 peripherals, the ECB bus clock and corresponding timing signals really should not exceed 6 MHz. As the CPU clock frequency is 12.288 MHz (for easy baud rate generation, it should be an integer multiple of 2.4576 MHz), it is convenient to use half the CPU clock frequency. This results in a bus clock of 6.144 MHz, which requires Z80B or Z80-6 components. With the internal wait-state generator of the Z280 programmed to insert four wait-states every I/O transaction (the maximum value), the duration of the I/O cycles exactly matches the divided clock. The clock divider is implemented by a single flip-flop (HCT74) and is reset at the beginning of each transfer to produce the correct phase relationship between bus clock and timing signals. Since the Z-bus control and timing signals cannot be used for the ECB bus, they are converted

into the appropriate Z80-type signals with a GAL.

Since the external peripherals use Z80-type vectored interrupts, the bus interface must be able to generate the correct interrupt acknowledge and RETI timings. Interrupt-acknowledge is treated as an I/O transaction and stretched by the Z280, but the RETI instruction consists of two memory cycles, which are too fast for the peripherals (besides the fact that the memory control signals do not appear on the ECB bus). Last but not least, the Z280 has a new interrupt mode (mode 3) which uses another return instruction (RETIL). So a slow RETI timing can be simulated using special accesses to on-board I/O locations, with a GAL generating the correct signals for the ECB bus. This way, vectored external interrupts are supported, although the clock speeds are very different.

The Z280 supports 24-bit I/O addresses as opposed to the 8/16-bit addresses of the Z80. The upper eight I/O address bits are accessed via the 'I/O Page Register' within the MPU. For full access to the 256 I/O addresses which are specified for the ECB bus, the bus interface is decoded to have an I/O page of its own. Another page is used for the on-board I/O, and two pages are reserved for the internal I/O registers of the Z280. Decoding of the I/O pages and addresses is done within GALs.

Internal and On-Board I/O

The Z280 comes with a serial interface (UART), three 16-bit counter/timer circuits, and four DMA channels on-chip. To avoid an external baud rate generator for the UART, one of the timers is used. The 12.288 MHz clock allows standard baud rates up to 38400 to be generated. Higher, but non-standard, baud rates are also possible.

The internal UART is completed with two handshake signals which are not supplied by the MPU. A second serial interface is provided by a Twenty-Pin-UART (TPUART) COM81C17 by SMC (it includes its own on-chip baud rate generator). Both interfaces are buffered and shifted to RS-232C levels with a 5V-only line driver and receiver, the LT1134.

The CPU280 contains a real-time-clock (RTC) with 50 bytes of nonvolatile RAM, the Dallas DS 1287. Since this part already contains the lithium battery, no external circuitry is required. The RTC is able to generate interrupts at a specified date and/or time (alarm) or periodically. The NVRAM is ideal for storing configuration parameters.

Floppy disk I/O is handled by a WD/SMC 37C65 floppy disk controller (FDC). This neat chip, cased in a 44-pin PLCC, contains the complete controller. Just connect the CPU bus to some pins and the disk drives to some other (and don't forget the quartz crystal), and everything is running. Floppy-related data transfer may be handled by one of the Z280's DMA channels.

Simple TTL chips (HCT367, HCT259) are used to implement one 6-bit input port and one 8-bit output port. One bit of each port is used for the handshake signals for the first RS-232C interface. The other outputs drive some FDC control lines and three LED's for status display. Some of the inputs are connected to jumpers so that they can be used for configuration purposes.

There are four 16V8 type GALs on the CPU280 board. They contain memory address and CPU state decoding, I/O decoding, RAM timing and CAS decoding, and some 'glue' logic. Since nearly all signals are processed with only one

logic stage, the standard 'slow' 25 ns GALs are fast enough.

The entire CPU280 circuit is designed using CMOS technology. The internal logic is made with the 74HCT series, which is fast enough for nearly every signal. The bus interface and one timing-critical function in the DRAM interface use 74ACT chips. All LSI also are CMOS, and the GALs should be taken from the 'Q' series (quarter-power). As a result, the complete CPU280 board—fully operating at maximum clock speed—draws only about 350 mA from a single 5V power supply. No other voltages are required. The 32 chips nearly exactly fill the board space of the EuroCard, with just enough space left to avoid a multilayer PCB. (Hey, Wayne and Paul, why did you need SMD for just 23 chips?)

As with every good single-boarder, you just need to connect a power supply, at least one disk drive of any kind and size, and a terminal to complete the CP/M-Plus workstation. However, by connecting the CPU280 to a standard ECB backplane, you are free to use nearly any available ECB I/O board.

The Software

The best hardware doesn't produce anything without the right software. With this in mind, I adapted my CPU180 BIOS to the CPU280, now using the powerful Z280 instruction set, of course. As with the circuit design, the basic principles and structures of the BIOS were taken directly from the CPU180. However, I completely redefined and enhanced the AutoFormat system and added a menu-driven hardware configuration program to the boot loader (this was impossible with the CPU180, as it had no NVRAM). Of course, some further improvements were made based on the experience of four years of CPU180 operation.

Normally, the complete system (boot loader, CCP, BDOS, and BIOS) is booted directly out of the EPROM. As a result, you can boot up your machine in two seconds without any noise or mechanical action. (For testing new system versions, of course, booting from disk is also possible by pressing a button during RAM-test.)

With all functional enhancements fully compliant with CP/M-Plus definitions, all CP/M-Plus and most CP/M-2.2 software can be run on the CPU280 without any problems. By the way, the operating system runs in the Z280's system mode, while all user programs run in user mode. This is done mainly to achieve easy bank switching (which the MMU does automatically in this case), but it also increases system security. Unfortunately, CP/M-Plus must have the BIOS entry vector and some data structures in common memory, so you cannot absolutely prevent user programs from damaging the system software. But since we don't have any better (and still compatible!) operating system, we have to live with that fact.

The Power

What is the real performance of a Z280 running with every booster switched on? The answer, unfortunately, is not as clear as you would probably like it to be. First, the Z280 is a pipelined CPU. So you really can't say how many clock cycles any instruction will take; it depends on the last few previous instructions. In addition, some instructions (jumps and calls, for example) flush the pipeline and thus are relatively 'slow'. Second, effective CPU speed depends on the 'hit ratio' of the cache controller. Small loops will run much faster than 'spaghetti code'. Third, the 16-bit arithmetic unit of the Z280 (opposed to the 4-bit one of the Z80) processes

indexing and math operations with greater speed gain than with other instruction types. So the more a program makes use of these instructions, the greater the effective speed.

Although it is impossible to specify exactly the power of the Z280, you can say that with normal 8080 or Z80 software it will have the power of a Z80 running at 16 to 20 MHz. Of course, using the new instructions (there are more than 600!) further increases the performance (while losing Z80 compatibility, of course). As the Z280 is used more and more, I hope we will soon see the first real Z280 programs or the first real Z280 operating system (which could get rid of the annoying 62 KB TPA limit of CP/M-Plus...).

The Development History

After the first version of the CPU280 ran stably in March, 1990, I made a redesign of the PCB layout with slight changes in parts of the circuit. In June, 1990, I ordered the first run of PCBs, and in November, 1990, I sent them along with all semiconductors and special parts to about 25 people here in Germany. A few other people around the world got a PCB without the part set. I had to wait until November because that was the time when the 12.5 MHz version of the Z280 became available, and we didn't want to take the slower version first and upgrade a few months later. As of early 1991, many of the machines were running very well, as far as I have been informed by the users. The CPU280 has proved itself to be very fast and very reliable. Our 'PD and ZCPR Man', Helmut Jungkunz, likes the machine for its flexible method of processing different disk formats; that's why he uses it for nearly all disk distribution he has to do. Of course, he likes the raw power, too.

But the CPU280 is not our only project here in Germany. In a future column I would like to introduce my next board, an IDE controller which connects standard (PC) AT-Bus harddisks to any ECB-bus based CP/M system. The board also contains an active termination of all bus signals, a Centronics printer interface, a four-LED power monitor, and two system control buttons for reset and NMI (which, with the CPU280, forces a warm system reboot). This board nicely matches the CPU280 and allows one to build a really high power CP/M workstation.

A friend of mine is developing an LCD terminal. Together with a low-power Z180 single-boarder (based on my CPU180) it makes a powerful CP/M-Plus laptop! My newest project, which is coming along very well, is a high-performance CRT terminal for text and graphics at an unbeatable price. Wait and see! Needless to say, all three of these boards are EuroCard size. My friends and I look forward to describing these projects in future issues of *TCJ*.

How to Get a CPU280

Some of you may be wondering how you can get a CPU280 of your own. Well, overseas shipment costs would be too high for sending complete kits as we did here in Germany. I think the easiest way to make the CPU280 available in the USA (and elsewhere outside Germany) is for a local dealer to provide my PCB together with locally obtained components. For the US market, who could do that job better than Sage Microsystems East? So, if you are interested in the board, the semiconductors, or both, please contact Jay Sage. Kits will include a disk with the complete system software, a hardware manual describing the circuit details, and a brief software description (I still haven't found the time to complete a real software manual).●

TCJ Center Fold CPU280

Tilman Reh

Special Feature

All Users

Zilog Z280 CPU

Now it's been exactly four years since I described the CPU280 in TCJ issue #53. In that early article, I described the circuit principles and also some details, but we didn't include a schematic of the board (though in TCJ #54, a photograph of the CPU280 board was printed (p.6), which was taken at our club meeting in Germany). Now Dave, our new editor, has suggested to include these drawings as the Centerfold of this TCJ issue, to give you an example of current 8-bit computer technology.

So you should now see the circuit drawing of the CPU280, along with the PCBs part layout and some accompanying information like bus connector pinout and GAL contents, on these Centerfold pages.

To give our new readers a first impression: the CPU280 is a powerful single-board computer using the Z280 processor, which is an enhanced Z80 core with extended instruction set (about twice as much instructions and addressing modes) and many on-chip peripherals. The CPU280 is built on a standard EuroCard (100x160 mm, or about 4x6.2 inches) and connects to the ECB bus (a european standard 8-bit bus using VME-type connectors).

As mentioned in that early description, the circuit design of the CPU280 is really straightforward, though it might not seem so at the first glance. The drawing might appear rather complex because it is all combined on a single sheet, while the functional circuit blocks are much simpler. We might divide the circuit into five parts:

1. CPU basics
2. Memory interface
3. Onboard I/O
4. ECB bus interface
5. Glue logic & GALs

1. CPU basics

On the CPU280, the Z280 processor is used in 16-bit mode. This means we'll have a 16-bit multiplexed address/data bus, and "Z-bus" type control signals (as used by the Z8000). So we definitely have to latch the 16 lower address bits on each memory or I/O access (IC2, IC3) to get a static 24-bit address, and we also have to decode the Z-bus lines (especially its status lines ST0..3) to get appropriate access signals. Upon reset, some basic operating parameters of the Z280 processor can be set up by asserting /WAIT during the rising edge of /RESET and placing the configuration data

on the lower data bus. After /WAIT is released, the processor starts operation. This configuration is done by the circuit around IC5, with /WAIT delayed by R1/C5, while RN2, RN6, and R4..7 supply the configuration data to the processor (depending on the jumpers J1..4). Note that the reset signal to the Z280 itself is fed through a GAL to satisfy the CPUs rise time needs. The Z280 chip also contains a serial interface (UART) and a four-channel DMA controller, which are used within the CPU280.

2. Memory interface

All memory accessible by the Z280 is located onboard. Since the Z280 supports different timing for two memory areas, EPROM memory and RAM memory each use a different area. There are two EPROMs (IC9..10) to hold the 16 bits of data, while four 4-bit RAMs are connected "in parallel" for 16 bit. Interfacing to the EPROMs is as simple as can ever be, however the RAM circuit is somewhat more complex. It uses a fully synchronous timing chain (IC30, IC4) and some glue logic contained in GALs (IC21..22) to provide the necessary signals for the dynamic RAMs (IC11..18) and their address multiplexers (IC19..20). (The principle of the timing chain is that a logic '1' is shuffled through the chain with each clock pulse during memory accesses.) To support the Z280s burst mode (used to fill up the internal cache memory), the GALs also contain logic to increment the lower address bits and generate separate /CAS pulses for each in-burst access.

3. Onboard I/O

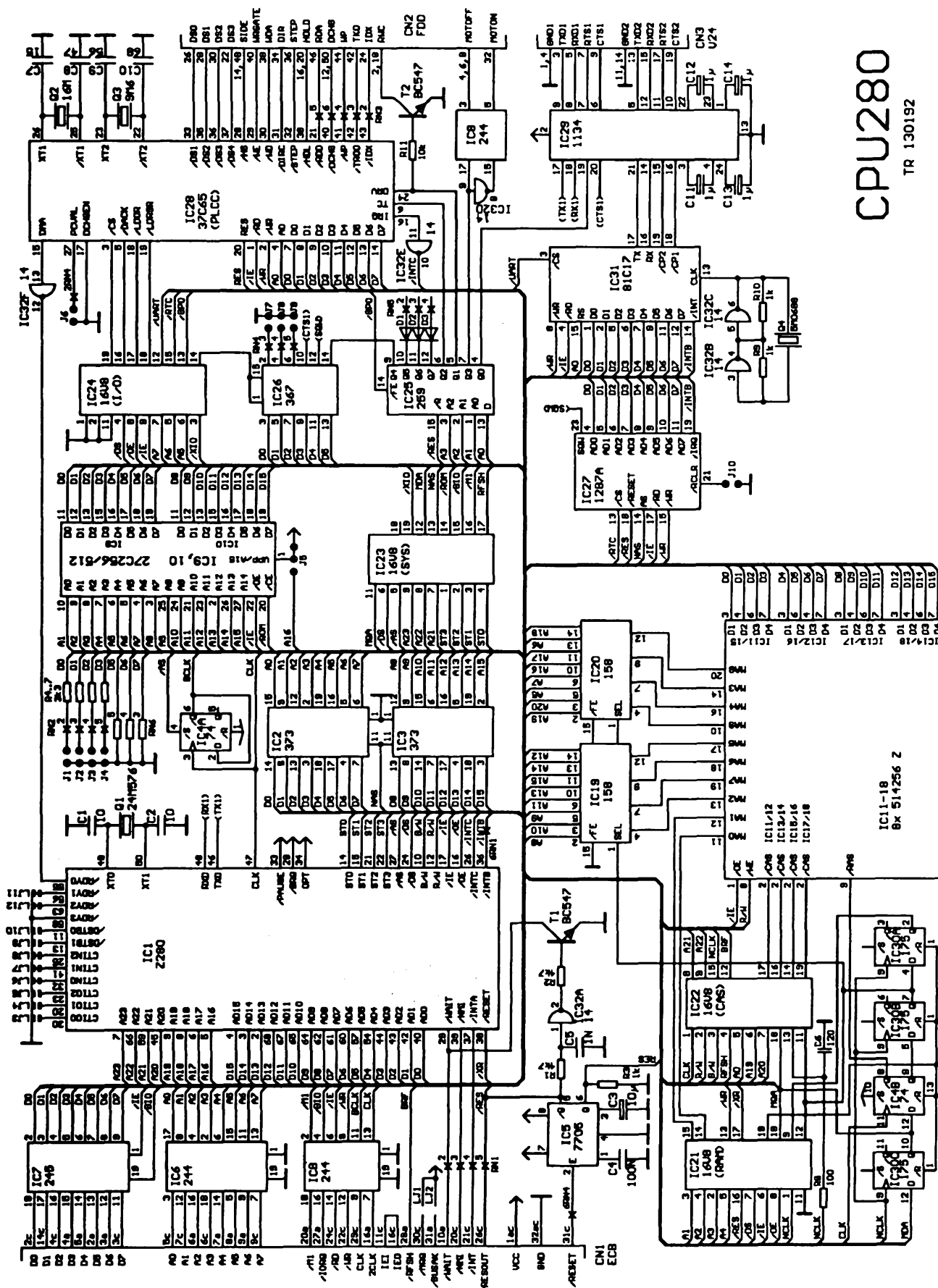
The I/O circuit is rather simple again. All decoding is done in a GAL, which provides differently sized address areas for each select signal. The on-board I/O consists of the floppy controller (IC28), a second serial interface (IC31), and a real-time clock with non-volatile RAM (IC27). There further are two general purpose ports for bitwise input and output, GPI resp. GPO (IC26, IC25). The floppy controller uses one channel of the Z280s DMA controller to transfer data to/from memory.

4. ECB bus interface

The interface to the ECB bus supports I/O transactions only. Due to the speed limits of the bus, I/O accesses are stretched to meet the standard 6 MHz Z80 timing, while appropriate Z80-type control signals are generated. The processor clock is also divided by two for the bus, and the

CPU280

TR 130192



ECB-bus pinout:

Physical connector: VME-type 64-pin, rows a+c used.

Row a	Pin No.	Row c
+5V	1	+5V
D5	2	D0
D6	3	D7
D3	4	D2
D4	5	A0
A2	6	A3
A4	7	A1
A5	8	A8
A6	9	A7
/WAIT	10 *	D8 (A16)
/BUSREQ	11	IEI
BAI (A18) *	12 *	D9 (A17/A19)
+12V	13 *	D10 (A18/A20/-12V)
D11 (A19/-12V) *	14	D1
-5V	15	-12V
2xCLK	16	IE0
BA0 (A17) *	17	A11
A14	18	A10
+12V	19 *	D13 (A16)
/M1	20	/NMI
D14 (A22) *	21	/INT
D15 (A23/RDY) *	22	/WR
(BAI) *	23 *	D12 (A21)
+UBAT	24	/RD
NxCLK (BA0) *	25	/HALT
(RDY) *	26	/RESOUT
/IORQ	27	A12
/RFSH	28	A15
A13	29	CLK
A9	30	/MRQ
/BUSAK	31	/RESIN
GND	32	GND

The 13 pins marked with an asterisk have different meaning, depending on the maker of the boards. The original

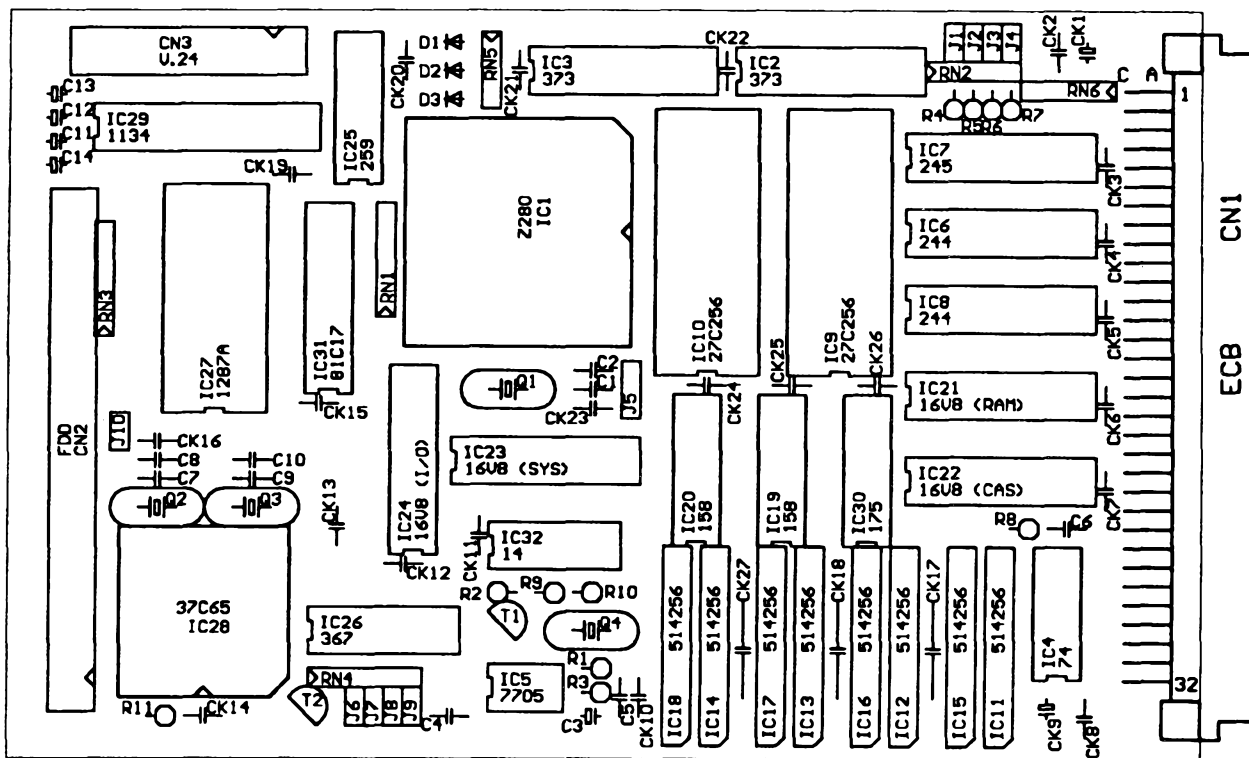
meaning defined by Kontron (the company which introduced the ECB-bus) is always printed first. Additions or redefinitions by other makers are printed in parantheses. The differences concern three signal types: higher order address/data lines, DMA priority chaining, and -12V power supply. The latter easily produces grey smoke if boards of different makers are combined!

Note that Kontron defined the bus with 16 address and 16 data lines. When more memory address space was needed for Z80-based computers, people simply added address lines from A16 up, using some of the "free" bus lines. Unfortunately, each maker took different ones. Maybe this was part of the (common) philosophy to prevent buyers from purchasing anything from another maker...

Luckily, most of the more simple boards (or those which need the bus only for simple tasks, like the CPU280) use only the common subset of the bus, so there are many cases where no problems arise.

Just for information, the 16-bit variant for which Kontron reserved those high-order data lines was rarely seen. I don't think it ever had a significant meaning in the ECB-bus market.

The meaning of the bus signals exactly follows the Z80 processor signals, since this is where this bus is derived from. So, for timing and meaning of the signals, please refer to the Z80 CPU data sheet. Besides the power supply, additional signal lines are just the priority chains for interrupts (IEI/IE0) and DMA (BAI/BA0/RDY).



divider flipflop is synchronized for every I/O access to exactly reproduce the Z80s phase relationship between control signals and clock. To provide support for external interrupts from the bus, some extras were added (/M1 to simulate interrupt acknowledge, special I/O decoding to simulate RETI instruction fetch for the bus). External interrupts might use external vectors (Z80 peripherals) or an internal (fixed) vector. The non-maskable interrupt also is connected to the bus. Note that there also is a refresh signal (BRF) generated for the bus, although there is no external memory support. This signal is to support I/O-based RAM-disks built with dynamic RAM.

5. Glue logic & GALs

Looking at the four GALs containing all the "glue logic", I must admit that it took some time to design this part of the circuit. When designing a complex circuit like the CPU280, I first make drafts of all the main functions I need, and don't care much about the glue logic which is necessary to make them work. After all functional blocks are designed, I collect all the glue logic and try to combine it in a senseful manner. This results in some standard TTL parts (like flipflops, multiplexers, inverters, buffers etc.) and some remaining "glue" which I then try to fit in as few GALs as possible. I could also have used a larger PLD, but I still prefer the smaller GALs for some reasons. So after I knew the functions which should be programmed into GALs, I started puzzling inputs and outputs to need a minimum number of GALs. This surely is more work than is obvious when seeing the final result! Though much more versatile than PALs, GALs are restricted in what you can do, mostly because there are only eight outputs per chip. Some times I thought I had found the optimum combination just to realize that one of the GALs would need nine output lines for it (of course two outputs were free at another GAL then, or even some inputs at the particular one!). One of the main rules when optimizing GAL circuits is to connect any given signal to a minimum number of GALs. For this, you sometimes have to "cascade" two or more GALs, like I did with the I/O decoder on the CPU280: the system decoder GAL (IC23) decodes *that* there is an on-board I/O access, while the I/O decoder GAL (IC24) uses this signal to further decode *which* I/O device was selected. (When cascading GALs, however, you must keep an eye on the overall timing.) During GAL optimization, sometimes you also modify the functional block circuits to make more efficient use of the glue logic. At that point, you have to enter the loop again and restart collecting and optimizing all "glue" needed. I did this with the CPU280 several times, before the circuit made perfect use of all TTL parts and GALs, minimizing total chip count. But the result can be seen, at least in my opinion: With four cheap standard GALs, the total parts costs are much lower, the PCB layout much easier, and support is much better, than with a single large PLD!

Tilman Reh, Dec 1995

Ed: Just after Tilman sent me this, Zilog announced that they would stop making the Z280 in early 1996.

CPU280 GAL EQUATIONS

```
TITLE      CPU280 RAM-TIMING AND NIBBLEMODE IC21
AUTHOR     TILMANN REH
COMPANY    REHDESIGN
DATE       23.07.1990

CHIP Z280RAM PALCE16V8

NCLK A3 A1 A2 A4 IE DS OE MOD GND
OQE MUX WR MA0 MA1 RES CPURES MQA FFR VCC

EQUATIONS

/WR = /OE * /DS
MA0 := /MQA * A3 + MQA * /MUX * A1 + MUX * /MA0
MA1 := /MQA * A4 + MQA * /MUX * A2 + MUX * MA0 + MUX * MA1
/FFR = IE * DS * MOD
CPURES = RES
```

```
TITLE      CPU280 CAS-DECODER IC22
AUTHOR     TILMANN REH
COMPANY    REHDESIGN
DATE       20.09.1992

CHIP Z280CAS PALCE16V8

CLK RW BW RFSH A0 A19 A20 A21 A22 GND
MUX BRF NCK CAS1L NCLK CAS0H CAS0L MQA CAS1H VCC

EQUATIONS

/BRF = RFSH * MQA
NCLK = /CLK

/CAS0L = MUX * /RFSH * /A22 * /A21 * /A20 * /A19 * BW * A0
+ MUX * /RFSH * /A22 * /A21 * /A20 * /A19 * /BW *
( CLK + NCK + /RW ) + RFSH * /MUX * MQA

/CAS0H = MUX * /RFSH * /A22 * /A21 * /A20 * /A19 * BW * /A0
+ MUX * /RFSH * /A22 * /A21 * /A20 * /A19 * /BW *
( CLK + NCK + /RW ) + RFSH * /MUX * MQA

/CAS1L = MUX * /RFSH * /A22 * /A21 * /A20 * A19 * BW * A0
+ MUX * /RFSH * /A22 * /A21 * /A20 * A19 * /BW *
( CLK + NCK + /RW ) + RFSH * /MUX * MQA

/CAS1H = MUX * /RFSH * /A22 * /A21 * /A20 * A19 * BW * /A0
+ MUX * /RFSH * /A22 * /A21 * /A20 * A19 * /BW *
( CLK + NCK + /RW ) + RFSH * /MUX * MQA
```

```
TITLE      CPU280 SYSTEM-SIGNALS IC23
AUTHOR     TILMANN REH
COMPANY    REHDESIGN
DATE       02.12.1990

CHIP Z280SYS PALCE16V8

ST3 ST2 ST1 ST0 AS DS A21 A22 A23 GND
MQA MDA NAS ROM BIO M1 RFSH NNAS XIO VCC

EQUATIONS

NAS = /AS
NNAS = NAS
/ROM = ST3 * /A23 * /DS
MDA = ST3 * A23 * /A22 * ( /AS + NAS + NNAS )
+ /ST3 * /ST2 * /ST1 * ST0 * ( /AS + NAS + NNAS )
+ MQA
RFSH = /ST3 * /ST2 * /ST1 * ST0
/BIO = /ST3 * /ST2 * ST1 * /ST0 * /A23 * /A22 * /A21 * /DS
+ /ST3 * ST2 * /ST1 * /ST0
/XIO = /ST3 * /ST2 * ST1 * /ST0 * /A23 * /A22 * A21
+ /ST3 * /ST2 * ST1 * /ST0 * /A23 * A22 * /A21
/M1 = /ST3 * ST2 * /ST1 * /ST0
+ /ST3 * /ST2 * ST1 * /ST0 * /A23 * /A22 * A21
```

```
TITLE      CPU280 IO-ADDRESS-DECODER IC24
AUTHOR     TILMANN REH
COMPANY    REHDESIGN
DATE       24.03.1992

CHIP Z280IO PALCE16V8

NG NC XIO DS A6 A5 A7 OE IE GND
NC UART GPO GPI RTC DACK LDOR LDRSR FDC VCC

EQUATIONS

/RTC = /XIO * /A7 * /A6
/FDC = /XIO * /A7 * A6 * /A5
/DACK = /XIO * /A7 * A6 * A5
/LDOR = /XIO * A7 * /A6 * /A5
/LDRSR = /XIO * A7 * /A6 * A5
/UART = /XIO * A7 * A6 * /A5
/GPI = /XIO * A7 * A6 * A5 * /IE * /DS
/GPO = /XIO * A7 * A6 * A5 * /OE * /DS
```

Connecting IDE Drives to 8-Bit Systems

By Tilmann Reh

Interfacing Systems

Intermediate Skills

8 - Bit Construction

I recently went to a IDE drive on my own system. Since then I have been wondering about their technical side. Although Tilmann is interested mostly in their 8-Bit data usage, the information presented here will help with any system. So Tilmann can I hook this to my NOVI system? Read on and see for yourself.
BDK

Most of us know about the features of a hard disk compared to floppy disk operation. You get much higher storage capacity while dropping access times down to a few milliseconds. Before installing a hard disk, I was used to copying all files for the current project to the RAM disk of my CPU280 and then copying all changed files back to floppy when I was finished working for the day. Although this is much better than working with floppy disks only, it is not comparable to using a hard disk. With a hard disk, you just work on your projects, which can now use files that would not fit on a floppy or RAM disk, and you don't have to copy files around the drives. In addition, the access times are almost as fast as those of a RAM disk. Last but not least, you are freed from changing floppies like a D.J., since all files are accessible without mechanical action on your part.

The Technology Decision

When thinking about connecting a hard disk to a given computer system, the main decision that must be made is which interface technology should be used. The old ST-412/506 interface is not up to date (I think it is impossible to buy new, small drives with that interface), and the hardware expense for this type of controller is great. Additionally, these con-

trollers have some critical analog circuits which have to be adjusted very carefully.

The next technological step was the ESDI interface, which is quite similar to ST-412/506, except that data is transmitted in parallel and with higher (but still fixed) data rates. The ESDI controllers are more complex and the drives more expensive than the ST-506 components. So ESDI is not interesting for our use.

The SCSI interface is a universal peripheral interface that is often used for hard disk connection. Some machines (Apple Macintosh, NeXT) even support no other interface for this purpose. SCSI is a very powerful and good interface, and SCSI drives are well established and available at acceptable prices. For the host interface (that is the controller) there are some chips available that do the complete bus protocol work in hardware and software and deliver raw data for the host processor.

Another alternative is the IDE interface (also known as the AT Bus interface). This interface is used in almost all IBM clones these days. Like the SCSI drives, the IDE drives contain the complete hard disk controller on the drive (this is where the name comes from: IDE means "Integrated Drive Electronics"). But these drives are connected to the standard PC-AT bus system and are accessed just as with the normal PC-AT hard disk controller. So, in effect, you don't need a controller any more but just some interface electronics that simulates an AT bus. The IDE drives are slightly cheaper than similar SCSI drives, and since the interface is simpler, I chose this one for my project.

However, it must be said that this decision was made with only the hard disk connection in mind. If someone wants to connect more peripheral devices (i.e., scanners, CD ROM, etc.) or more than two hard disks (which is the IDE limit), the SCSI interface is preferable (one interface for all devices).

The Interface Circuit

The greatest problem when interfacing an IDE disk to an 8-bit computer system is the differing bus width. While the control registers of the IDE drive are still eight bits wide, the data transfer is done word-wise (16 bits each transfer). So we need an interface which maps the 16-bit data register to the 8-bit bus.

There are two basic approaches to doing that. The first (and easiest) one is to map the two halves of the 16-bit data into two different I/O addresses. The strobe for the IDE drive would have to be generated with the read signal for the lower half and with the write signal for the upper half. The control registers would then be accessed by always reading the LSB and writing the MSB. However, although only simple hardware is required, this method has a very great disadvantage: since two different I/O addresses must be accessed alternately, you cannot use string instructions (such as the Z80 INIR/OTIR) or a DMA controller for data transfer. Thus, the transfer rate would be relatively slow and the programming not very elegant.

The other approach is to map the 16-bit data onto two consecutive accesses to the same I/O address. This way, some circuit expense is necessary to switch the right data halves to the data bus and to

handle the 8-bit accesses to the control registers. However, with this slightly more complex hardware, we get great software advantages. With this technique, using string instructions or DMA is the normal way to transfer the data to or from the disk drive.

Of course, we need some circuitry to remember which half is to be processed next, in order to select the correct data path and generate the correct strobes for the drive. As we have to distinguish only two cases, one flip-flop is enough. Since I planned to use a GAL (generic array logic) for address decoding and bus interface anyhow, I used one of the GAL macrocells to make the flip-flop. The clock pulse for the flip-flop is generated each time any register is accessed, and the data is set to zero when other than the data register is selected. This way, accesses to any control register also reset the state flip-flop, thus ensuring proper conditions when the data transfer is started.

The rest of the interface circuit is self-explanatory: one half of the 16-bit data is always processed directly, while the other half is stored in a latch or register. For the control registers, the latch becomes transparent. The strobes and select signals for latches, buffers, and drive are generated with the GAL mentioned above.

Slack Space On Board

The IDE interface itself would easily fit twice (or even three or more times) on a standard EuroCard-sized PCB. To avoid wasting board space, I filled the free space with some useful circuits which would serve the CPU280 very especially well but also make sense with other systems. The first additional circuit is an active termination for the complete ECB Bus, which is absolutely necessary with bus clocks of 4 MHz or more and a backplane of some length. With still more space left, I added two control buttons for hardware reset and NMI (non-maskable interrupt) generation and four LEDs as a power-control monitor. Last but not least, I finished the design with a Centronics-type parallel printer interface. The decoding signals for this interface

are generated using components that were already there for the IDE interface, so this involved almost no additional circuit expense. Of course, if someone needs only the IDE interface, they can just leave the rest of the board unused!

How To Get One

If you are interested in the interface, I think it will again be the best to contact Jay Sage for the availability of PCBs, programmed GALs, driver software, etc.

Tillmann Reh is an electronic engineer at the University of Siegen, Germany. He also owns a small company that develops custom solutions using embedded controllers or microcomputers. Tillmann has been active with CP/M since 1983 and developed a number of ECB-bus boards. He can be reached by regular mail at 'In der Grossenbach 46, W-5900 Siegen, Germany' or by E-mail (international/bitnet) at 'tilmann.reh@hrz.uni-siegen.dbp.de'.

```
TITLE          IDE/CENTRONICS INTERFACE GAL IC1
AUTHOR         TILMANN REH
COMPANY        REHDESIGN
DATE           22.03.1992
```

```
; Accesses to the hard disk are always with LH = high. So this signal
; has complementary meanings when reading resp. writing. To access the
; drive with the first data read, the LH flipflop has to be set before
; the real data transfer begins (one dummy-read of the data register).
```

```
CHIP IDE PALCE20V8
```

```
CK A7 A4 A5 IORQ A6 WR A0 A1 A2 A3 GND
OE M1 CLK LH CS0 RD16 SEL WRLO WR16 RDHI RD VCC
```

```
; Base address (BASE) to be changed only here! The lower nibble of the
; addresses are partly fixed by the hardware design.
```

```
STRING BASE '(A7 * /A6 * /A5 * /A4)' ; Base Address 80h
```

```
STRING PARSEL '(BASE * /A3 * A2 * /A1 * A0)' ; Centronics Adr. x5
STRING CS1ADR '(BASE * /A3 * A2 * A1)' ; CS1 Adr. x6..x7
STRING DATADR '(BASE * A3 * /A2 * /A1 * /A0)' ; CS0/Data Adr. x8
STRING TFRADR '(BASE * A3 * (A2 + A1 + A0))' ; CS0/Task Adr. x9..xF
STRING IDEADR '(CS1ADR + DATADR + TFRADR)' ; all IDE-Addresses
```

EQUATIONS

```
/CS0 = TFRADR ; Task File Access
+ DATADR * LH ; Data Write MSB / Read LSB
```

```
/SEL = (PARSEL + IDEADR) * /IORQ * M1 ; Board Access
```

```
/CLK = (DATADR + TFRADR) * /IORQ ; LH-Clock: Data & Task File
```

```
LH := /LH * /TFRADR ; FlipFlop: LSB/MSB Toggle
; Reset if Task File Access
```

```
WRLO = DATADR * /IORQ * /WR * /LH ; write Data LSB in latch
+ (TFRADR + CS1ADR) * /IORQ * /WR ; transparent for all others
```

```
/WR16 = IDEADR * /IORQ * /WR ; MSB and latched LSB to IDE
```

```
/RD16 = DATADR * /IORQ * /RD * LH ; read Data LSB, latch MSB
+ (TFRADR + CS1ADR) * /IORQ * /RD ; all others transparent
```

```
/RDHI = DATADR * /IORQ * /RD * /LH ; read Data MSB from latch
```

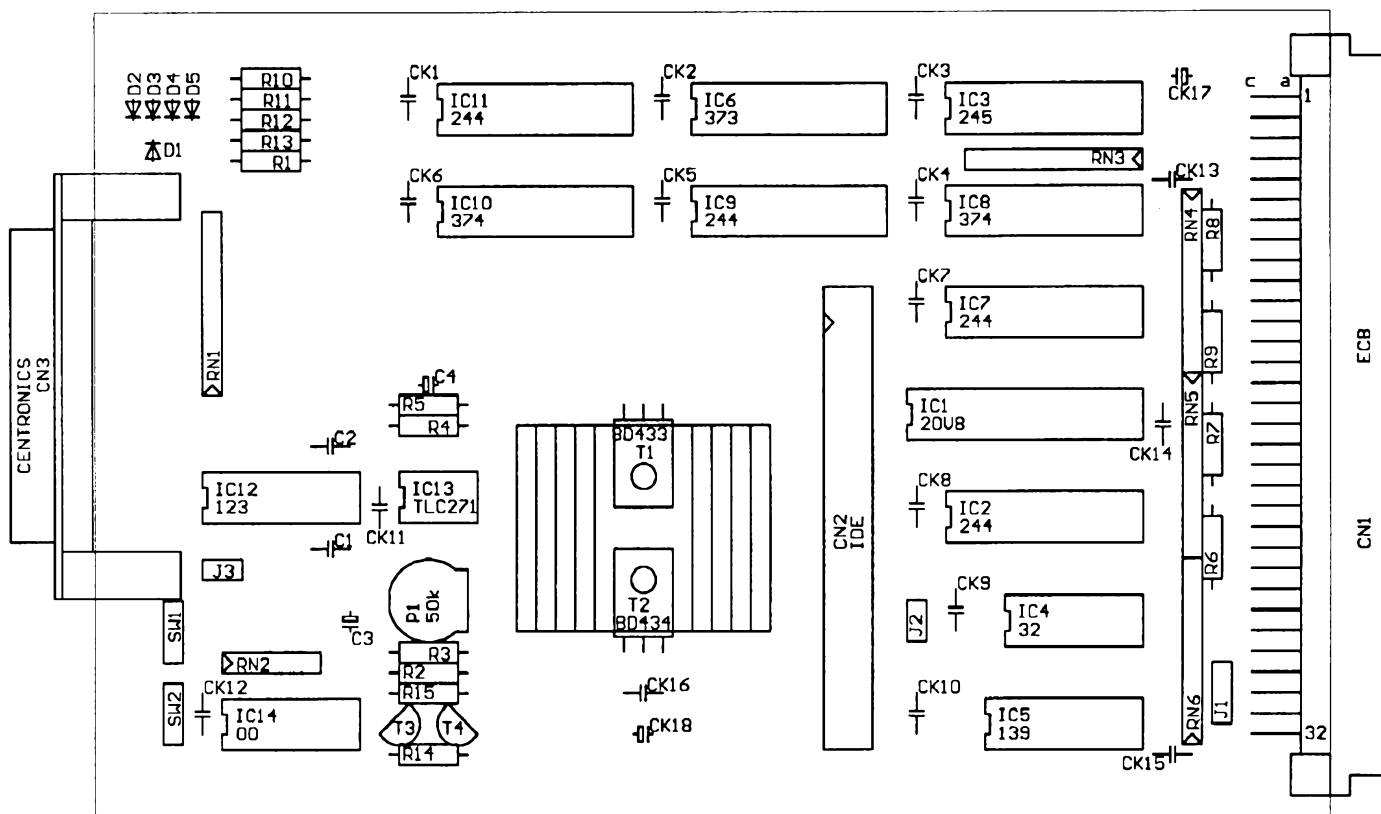
220392 TR



```

R03 -> D0..D7
R04 -> A0..A7
R05 -> A8..A15
R06 -> M0Q, I0RQ, R0D, W0R, M1, RFSH, CLK, ZCLK
R6..R9 -> INT, NMI, WAIT, BUSRQ

```



IDE

CONNECTING IDE DRIVES

by Tilmann Reh

Special Feature

Intermediate Users

Part 2: IDE Basics

Now it has been already one year since I described my 8-bit ECB-bus-based IDE interface here in TCJ. The delay in continuing with my description was caused by difficulties with the communication path between me and Bill Kibler. Since then, some questions have come up which were not covered by that article. So here is the missing information, I hope.

Remembering the Basics

Let us first have a short look at the drives we want to use. When discussing the different hard disk interfaces in my last article, I already pointed out that IDE drives of the AT-type, thus often called AT-bus drives (Bill Kibler calls them ATA drives, but this abbreviation is not the usual one, at least in Europe), are the ones with the best price/performance ratio one can get. This is even more the case now. So IDE drives still are the very first choice if you are looking for a good and cheap hard disk for your computer.

But what's special with those drives? I already mentioned that the IDE drive contains the complete hard disk controller. It is accessed with a system-bus interface compatible with the PC/AT (ISA) bus and offers control and data registers still compatible with the very first PC/AT hard disk controller (based on the WD 1010 controller chip). But even if those specifications come from something I don't like at all, why not use the low-price components for real computing (i.e., with a CPU280)?

Bringing the hard disk controller into the drive electronics offers some advantages. One of the main features is that you don't have a serial data stream with fixed bit rate between controller and drive. Thus, there's no need for conditioning the signals for the interface, and you can use any bit rate. As a result, the hard disk performance is limited by the drive technology, not by the interface's bit rate. This is one reason why today's drives are so much faster than the older ones. And technologies like Seagate's ZBR (Zone Bit Recording) are possible with hardware-independent interfaces.

There is another main feature of bringing the controller into the disk drive. Today's drives have very extensive checks for data security. They store error correction codes (ECC) together with the sector data and automatically correct single-bit errors, so the sector need not be re-read in those cases. Additionally, if a sector is found to be too unreliable, it is internally marked

as bad and the data is mapped to a spare sector (usually there is one spare sector per track). All this is absolutely transparent to the user. So you now know the reason why today's intelligent drives don't have "defect lists" any more.

Since the PC's have such bad software (and hardware, too), there is another thing the integrated controller can do: translate virtual addressing information into physical. That means that the IDE drive is able to emulate another drive with different parameters (cylinder count, number of heads, and sectors per track). For the PC this is necessary because many PCs don't support drives with other than the historical 17 sectors per track, and many do not support free configuration of the drive parameters (only selection from a table is allowed). Also, some PCs mask off some bits of the cylinder number, since the first controller only had a 10-bit cylinder register -- so nearly every IDE drive still supports an emulation mode with less than 1024 cylinders and 17 sectors per track.

The IDE Interface

As mentioned above, the IDE interface is almost completely identical with a subset of the PC/AT expansion bus, so the drive can be connected (almost) directly to that. The only things required externally are two select signals (I/O address decoding). This gives us some information about how the interface works. In a PC the drive is accessed directly by the CPU via I/O accesses to registers internal to the drive. The disk data is transferred via the 16-bit data bus, but for compatibility to the older systems (again!) only 8 bits are used for command and status information. Besides the data bus, there are the standard Intel-type data strobe signals (/IORD and /IOWR), a few address lines, and some special signals. The connector is a 40-pin header, not to be confused with the XT-type IDE interface connector, which is also a 40-pin header but needs somewhat different hardware and totally different software!

The IDE interface allows connection of two drives with one cable. The second drive (slave) is then chained to the first one (master). However, I heard about problems when trying to connect different drives from different manufacturers. And the capacities of today's drive are so high that a single drive will

always be enough for an 8-bit personal computer system! So, I never tried this option.

To understand the interface in detail, let's have a closer look at the IDE interface connector and its signals:

1	/RES	2	GND
3	D7	4	D8
5	D6	6	D9
7	D5	8	D10
9	D4	10	D11
11	D3	12	D12
13	D2	14	D13
15	D1	16	D14
17	D0	18	D15
19	GND	20	No Pin
21	/IOCHRDY	22	GND
23	/IOWR	24	GND
25	/IORD	26	GND
27	/IOCHRDY	28	ALE
29	No Connection	30	GND
31	IRQ	32	/IO16
33	A1	34	/PDIAG
35	A0	36	A2
37	/CS0	38	/CS1
39	/ACT	40	GND

The signals of the IDE interface can be collected in several groups: The general control signals are /RES (Reset) and /PDIAG (Passed Diagnostics). The data bus consists of 16 data lines (D0..D15). The access control lines are three address lines (A0..A2), the select signals /CS0 and /CS1 (Chip Select 0/1), and the strobe signals /IORD and /IOWR (and eventually ALE, the address strobe). The remaining signals (IOCHRDY, IRQ, /ACT, /IO16) are status signals.

Now Let's Go Into Details.

The reset signal normally is active-low. However, I heard about drives with an active-high reset signal, but I never saw one (or read such specifications). The /PDIAG pin carries a bidirectional signal used for chaining two IDE drives (master/slave). It normally can also be left open.

The data bus carries the 16-bit data words to and from the host. However, when accessing the control and status registers of the IDE drives, only data bits 0 through 7 are used (8-bit transfer). The data bus lines are tri-state lines that may be connected directly to the host's data bus. However, to meet the host bus specs and to avoid noise problems caused by the interface cable, a bus driver IC should be used to decouple the IDE bus and the host bus.

The drive is accessed using the selection signals /CS0 and /CS1. This also has historical (compatibility) reasons. Together with the three address lines, there could be two-times-eight addresses being occupied by an IDE drive. However, while the main register set really has eight registers and is accessed with

/CS0 active, the other set (with /CS1) has only two valid addresses. We will have a deeper look at all the registers later. The data transfer is always strobed by the timing signals /IORD and /IOWR, for reading and writing, respectively. The address strobe (ALE) is often unused in the drive; it should be pulled high for static address lines (non-multiplexed busses).

The status signals are not absolutely needed to use IDE drives. Some of these signals are not commonly delivered at all (for example, /IOCHRDY (I/O Channel Ready), which is a WAIT signal for the host when the drive is much slower than the host processor in terms of interface access times). The /IO16 line informs the host of 16-bit transfers. Since we already know that data transfers are always 16-bit and everything else is always 8-bit, this is redundant (however, needed in the PC/ATs for their ISA bus). Line /ACT (Active) is an output which can be used for driving a drive-busy LED. Line IRQ is an interrupt request line that goes active on some internal events (if enabled by software).

Most IDE drives contain some jumpers that allow some options to be selected. This normally includes at least master/slave selection. Sometimes the /ACT signal may also be jumpered as an output signalling the presence of a second (slave) drive. The default state of the jumpers normally need not be changed (single drive, no special situation).

All interface lines carry CMOS-TTL-compatible signal levels. However, some signals (IRQ, /PDIAG, /IO16, /ACT) are able to drive higher currents. Those details should be looked up in each drive's specifications (for example, the /ACT output sinks 20 mA on my Conner drive, more than enough for an LED).

Accessing the drive is done with the following sequence of operations: First, the address lines and the chip selects must be set according to the desired register address. After some time (a minimum of 25 ns), /IORD or /IOWR is activated. This causes the data to appear on the data lines (when reading) or to be written to the drive (with the trailing edge of /IOWR, but there are setup and hold times to take care of). After a minimum of 80 ns, the strobe signal has to be removed. There are some more timing requirements, but these are the main ones.

The above timing details might differ from drive to drive. Always keep in mind that the IDE definition follows the PC/AT system expansion bus and that official standards were not specified until two years ago, when the IEEE finally defined some specifications (which many PC manufacturers are not following).

Unfortunately, I found that the drives do not match their own specs in every detail. For example, I found that the address lines of my Conner drive (a CP-3044 with 42 MB) must be kept stable for much more than the specified setup time. In addition, the drive is very sensitive to spike noise on the address lines, even if the noise appears long before an access is initiated. I

spent a great deal of time struggling with such unlucky details (fixing other people's bugs).

IDE Interface Registers

Now that we've covered the interface signals and their meaning and usage, let's look at the registers of the interface. We saw that there are eight addresses being accessed through /CS0 and two addresses through /CS1. The following is a list of all the internal registers of an IDE drive:

/CS0	/CS1	A2	A1	A0	Addr.	Read Function	Write Function
0	1	0	0	0	1F0	Data Register	Data Register
0	1	0	0	1	1F1	Error Register	(Write Precomp Reg.)
0	1	0	1	0	1F2	Sector Count	Sector Count
0	1	0	1	1	1F3	Sector Number	Sector Number
0	1	1	0	0	1F4	Cylinder Low	Cylinder Low
0	1	1	0	1	1F5	Cylinder High	Cylinder High
0	1	1	1	0	1F6	SDH Register	SDH Register
0	1	1	1	1	1F7	Status Register	Command Register
1	0	1	1	0	3F6	Alternate Status	Digital Output
1	0	1	1	1	3F7	Drive Address	Not Used

The above addresses are those used in the PC/AT. Of course they are dependent on the decoding of the chip-select signals. The registers accessed via /CS1 might differ depending on the manufacturer of the drive. As far as I know, they don't always follow the compatibility principle with the first hard disk controller of the PC/AT.

The registers being accessed with /CS0 are also called the "Task File", so sometimes the IDE is also referenced to as "Task File Interface".

The error register can only be read. It contains valid information only if the error bit in the status register is set. Only five of the eight bits are used. They have the following meaning:

Bit 7: Bad block. This bit is set when the requested sector's ID contained a bad block mark (can be set when formatting the disk).

Bit 6: Uncorrectable data error. Set when the sector data can't be recreated (even with ECC).

Bit 4: Requested sector ID not found (wrong sector number).

Bit 2: Command was aborted due to drive status error or invalid command.

Bit 1: Track 0 has not been found when recalibrating.

The unused bits are always read as zero. However, I guess it's best not to rely on that!

The write precompensation register was previously used to set the starting cylinder for write precompensation (a slight shift of the serial data stream pulses to compensate for some mag-

netic effects on the disk surface). Since IDE drives handle all that internally, this function is not needed any more. Today, this register is often used as a parameter register for enabling or disabling look-ahead reading. We'll have a deeper look at that when talking about the various commands of IDE drives.

The sector-count register defines the number of sectors to be read or written with the next read/write command. A zero value causes 256 sectors to be processed, so the count varies from 1 to 256. This register is also used during drive initialization to specify the number of sectors per track (remember the emulation capability).

The sector-number register contains the starting sector number for any disk access. After a sector is processed, and after the command is completed, this register is updated. When an error occurs, this register contains the ID number of the erroneous sector. Normally, the sector numbers start with 1 and increase with each sector. However, by reformatting the disk, this order and the values may be changed.

The cylinder-low and cylinder-high registers contain the 10-bit cylinder number to be accessed. Since many drives have more than 1024 cylinders today, the cylinder-high register is often expanded to more than two bits. Like the sector-number register, these registers are updated after command completion and after errors. They are also used during drive initialization as the number-of-cylinders parameter.

The SDH register is a special register serving several functions. SDH is an abbreviation for "Sector size, Drive and Head". The bits of this register are arranged as follows:

Bit 7: Historical: Extension Bit. When zero, CRC data is appended to the sector's data fields. When set to one, no CRC data is appended. Since today's drives always use ECC error correction, this bit must always be set (no CRC).

Bit 6-5: Sector Size. Since today's drives always have 512-byte sectors (unchangeable by the user) because PCs are not able to support other sizes, these bits must always be 0-1.

Bit 4: Drive. This bit distinguishes between the two connected drives when using the master-slave chain. Single drives are always accessed with the drive bit set to zero.

Bit 3-0: Head number. These four bits contain the head number (that is, the disk surface number) for all following accesses. Similar to the cylinder and sector number, these bits are updated by the drive. The head number field is also used for drive initialization to specify the number of heads.

The read-only status register contains eight single-bit flags. It is updated at the completion of each command. If the busy bit is active, no other bits are valid. The index bit is valid independent of the applied command. The bit flags are:

Bit 7: Busy flag. When this flag is set, the task file registers

must not be accessed due to internal operations.

Bit 6: Drive ready. This bit is set when the drive is up to speed and ready to accept a command. When there is an error, this bit is not updated until the next read of the status register, so it can be used to determine the cause of the error.

Bit 5: Drive write fault. Similar to "drive ready", this bit is not updated after an error.

Bit 4: Drive seek complete. This bit is set when the actuator of the drive's head is on track. This bit also is updated similarly to "drive ready".

Bit 3: Data request. This bit indicates that the drive is ready for a data transfer.

Bit 2: Corrected data flag. Set when there was a correctable data error and the data has been corrected.

Bit 1: Index. This bit is active once per disk revolution. May be used to determine rotational speed.

Bit 0: Error flag. This bit is set whenever an error occurs. The other bits in the status register and the bits in the error register will then contain further information about the cause of the error.

The command register is used to pass commands to the drive. There are many commands, not always using all parameters in the task file. Command execution begins immediately after the command is written to this register. Since this article is already quite long, I will cover the commands, their parameters, and their usage in another article, probably in the next TCJ issue.

The alternate status register contains the same information as the status register in the task file. The only difference is that

reading this register does not imply interrupt acknowledge to reset a pending interrupt (as the main status register does).

The digital output register contains only two valid data bits. Bit 2 is the software reset bit, which causes a drive reset when being set, and bit 1 is the interrupt enable flag.

The drive-address register simply loops back the drive select bit and head select bits of the currently selected drive. This information normally is of no use for the programmer or user.

Last Words

Now that we had a look at the IDE interface, we also see the physical limits of this interface definition. With a fully expanded cylinder-high register, we are able to address up to 65536 cylinders, with up to 16 heads and up to 256 sectors per track. This results in a maximum addressable drive capacity of 128 gigabytes. I think this should be enough for microcomputing!! However, even if the PC/AT BIOS limitations are encountered, we could address 1024 cylinders with 16 heads and 64 sectors per tracks, giving 512 megabytes maximum capacity. This is also not bad, at least for small (8-bit) computer systems, where complete application software packages require only about 100 kilobytes of disk space.

Next time I would like to talk about the applicable commands of IDE drives and give examples of how to write software that accesses those drives. Perhaps I will also return to describing my IDE interface board for the 8-bit ECB bus in more detail. If you have questions or details about which you would like to read more, contact me at the following addresses:

Tilmann Reh

In der Grossenbach 46

D-57072 Siegen, Germany

e-Mail: tilmann.reh@hrz.uni-siegen.d400.de

List of Abbreviations:

AT	Advanced Technology	Class of PC's
BIOS	Basic I/O System	Hardware-dependent part of OS
CMOS	Complementary Metal-Oxid-Silicon	Semiconductor technology
CRC	Cyclic Redundancy Check	Error detection code, see also ECC
ECB	???	European standard 8-bit system bus
ECC	Error Correction Code	Additional data for security
IDE	Integrated Drive Electronics	Intelligent hard disk interface
IEEE	Institute of Electrical and Electronics Engineers	
I/O	Input/Output	(self-explanatory)
ISA	Industry Standard Architecture	PC/AT expansion bus
LED	Light Emmitting Diode	Optoelectronic component
OS	Operating System	Software which makes computers usable
PC	Personal Computer	Synonym for the worst computer architecture
TTL	Transistor-Transistor-Logic	Digital component standard (74xx series)
XT	eXtended Technology	Class of PCs, previous to AT
ZBR	Zone Bit Recording	Variable Density Recording Method

CONNECTING IDE DRIVES

by Tilmann Reh

Special Feature

Intermediate Users

Part 3: IDE Commands

In Part II (printed in the previous issue of *TCJ*) we covered the basics of the IDE interface in terms of history, concept, hardware, and register structure. This time we want to dig deeper into the software side of those drives.

Terminology

Using common terminology, I often simply refer to the "drive" when, in fact, I am thinking of the integrated controller of an IDE drive. However, when explicitly talking of an external controller like the WD1010, I always refer to the "controller".

Register Accessing

Let us first recall the Task File. It consists of the data register, a set of six parameter registers, and the command/status register. For those who don't have Part II lying nearby, here is a shortform:

Relative Address	Register	Abbr.
0	Data Register	D
1	Error Reg. / Write Precomp. Reg.	E / WP
2	Sector Count	SC
3	Sector Number	SN
4	Cylinder Low	C
5	Cylinder High	C
6	SDH (Sector Size, Drive, Head)	D,H
7	Status Reg. / Command Reg.	

Also remember that the data register is the only 16-bit register!

Every parameter register of the task file is freely accessible as long as there is no active command. Before loading the command register, all related parameter registers must contain the appropriate values. They may be loaded in any order. After the command register is loaded, the issued command is immediately started. The original WD1010 hard disk controller chip had a flag (bit 1 of the status register) which was set during execution. With IDE drives, the BUSY flag of the status register is simply set until the command execution is completed.

The WD1010 controller chip knew only 6 commands. However, some of the commands have option flags within them. To support additional features, today's drives have many more commands. The following is a list of common commands,

options, and needed parameters, with the WD1010 commands marked by an asterisk and the manufacturer-dependent expansions marked with a plus sign:

Command	Type	7 6 5 4 3 2 1 0	Hex	Parameters
Recalibrate	*	0 0 0 1 (Rate)	10-1F	D
Read Sector	*	0 0 1 0 0 M L T	20-27	SC,SN,C,D,H
Write Sector	*	0 0 1 1 0 M L T	30-37	SC,SN,C,D,H
Scan ID / Verify	*	0 1 0 0 0 0 0 T	40,41	D,(SC,SN,C,H)
Write Format	*	0 1 0 1 0 0 0 0	50	C,D,H,(SC,SN)
Seek	*	0 1 1 1 (Rate)	70-7F	C,D,(H)
Exec Diagnostics		1 0 0 1 0 0 0 0	90	D
Set Drive Parameters		1 0 0 1 0 0 0 1	91	SC,(C),D,H
Read Multiple	+	1 1 0 0 0 1 0 0	C4	SC,SN,C,D,H
Write Multiple	+	1 1 0 0 0 1 0 1	C5	SC,SN,C,D,H
Set Multiple	+	1 1 0 0 0 1 1 0	C6	SC,D
Power Commands	+	1 1 1 0 0 x x x	E0-E6	SC,D
Read Sector Buffer		1 1 1 0 0 1 0 0	E4	D
Write Sector Buffer		1 1 1 0 1 0 0 0	E8	D
Identify Drive		1 1 1 0 1 1 0 0	EC	D
Cache On/Off	+	1 1 1 0 1 1 1 1	EF	D,WP
Power Save	+	1 1 1 1 1 x x x	F8-FD	?

Parameters in parentheses are needed with some drives and ignored by others (depending on the manufacturer and age). Any required parameters must be valid before a command is started.

Although most of the commands are manufacturer-dependent, this usually does not raise problems. For normal operation of the drive, only the WD1010's and few of the really common commands are needed. Now let's have a look at the options.

In the Restore and Seek commands, there is a four-bit rate field. This was originally intended to specify the step rate for head movements, with a zero value meaning 35 us per step and all other values representing counts of 0.5 ms per step (so that the range was from 0.5 to 7.5 ms). The hard disk controller had a memory for each drive's step rate, so the same value would be used for implied seeks later. But very soon, even with later ST-506 controller boards, this step-rate field became obsolete (due to handshake mechanisms between controller and drive). With today's IDE drives, the four lower bits of those commands are generally ignored.

The Read and Write commands originally had an option flag (M) for multi-sector (m/s) transfers. Today this flag is nonexistent. For doing m/s transfers, the sector-count register is simply set to the desired number of sectors to be processed.

But today's drives have another flag which originally wasn't there: the "Long" flag (L). When it is set, the ECC (error-correction) data is transferred after the sector's data field. I assume this was meant for error correction when there are too many errors for the drive to correct automatically. However, there is a very odd characteristic: the ECC data is transferred in *bytes* through the data register. This is the only case where the data register doesn't transfer word data! By the way, the number of ECC bytes differs from drive to drive, but can be read out using the Identify Drive command.

All generations of hard disk controllers and IDE drives support the last option flag for Read/Write commands, the Retry Flag (T). Normally, the drive retries to read/write a sector after non-fatal errors. When this option flag is set, automatic retries are disabled. The Scan ID command also supports this option.

The Commands

I will now explain the commands and their parameters in detail. To do this, besides drawing on my own experience in IDE interfacing, I collected detailed information and specifications from three different, independent sources. However, there still might be some slightly different drives or controllers out there. Please inform me if you encounter problems or differences with your disk.

One general feature of both the WD1010 controller and modern IDE drives is implied seeks. That means you don't have to explicitly move the read/write (r/w) heads to the desired cylinder before starting to read from or write to the disk. When the command is issued and the actual cylinder number doesn't match that of the cylinder registers, an implied seek is performed, transparent to the user. This applies to every command where it may be needed (Read, Write, Verify, Format).

Recalibrate (1xh):

This command moves the r/w heads of the selected drive from anywhere to cylinder 0. The controller waits for the drive to complete the seek before the task file is updated and the busy flag is reset. Upon successful completion of the command, the error register and the cylinder registers are set to zero, while SC, SN, and SDH remain unchanged.

Read Sectors (2xh):

This is probably the mostly used command. It will read from 1 to 256 sectors of disk data as specified in the SC register (with an SC value of 0 meaning 256 sectors to be read). The starting sector is defined by SN, C, and H in the task file.

When the task file contains invalid parameters, an error occurs.

Otherwise, the r/w heads are moved to the requested cylinder if they are not already there (implied seek). Then the data of the starting sector is read into the sector buffer, and the DRQ (data request) bit in the status register is then set. This informs the host that the sector data can be read from the sector buffer. When this is completed, DRQ is reset and the drive is ready again.

For reading multiple sectors (SC>1), when the sector buffer is completely read and DRQ is reset, the busy flag is set again immediately, and the next sector's data is read into the sector buffer. When DRQ is set again, the next sector's data can be read from the buffer. This is repeated until the SC register value reaches zero (this register is decremented with every successfully read sector).

In any case, after successful completion of this command, the SC register contains a zero value, and the other registers in the task file are updated to contain the cylinder, head, and sector number of the last-read sector. If an error occurs, the task file will contain the parameters of the sector at which the error was detected.

For the Read Sectors command, two options are possible. The "M" option is valid only with the original WD1010 controller (thus with old AT's). When M was not set (0), the SC content was ignored, and exactly one sector was read. When M was set (1), the SC value was taken as the count of sectors to read. Today things are different. With modern IDE drives, if the M bit is set, an error occurs. So this bit must always be cleared!

The other option ("L") is valid only with modern IDE drives. "L" stands for long and means that the additional ECC data, which the drive automatically puts after each sector, is transferred after the net data of each sector. When this option is set, the drive also does not check these ECC bytes, so it won't detect or correct errors. This provides a way to read a sector's (remaining) data even if it is nonrecoverably damaged. When using this option, remember that the ECC data is transferred as bytes through the word-wide data register!

Write Sectors (3xh):

The Write Sectors command is very similar to the Read Sectors command. Of course the data flow direction is different... This command will write up to 256 sectors of data to the disk. All parameters and options are similar to those of the Read Sectors command. After writing the command to the command register, the drive sets the DRQ flag, informing the host that the data can be written into the sector buffer. When all data has been transferred, DRQ is reset, and the drive starts writing the data buffer contents to the disk. The busy flag is set as long as the drive is physically writing to disk. The SC register is decremented, and, if not zero thereafter, DRQ is set again for the next sector. When using this command with the "L" option, the drive will use the ECC bytes delivered by the host

Continued after Centerfold, page 33.

Continued from page 26.

and not generate any by itself. For the "M" option, the details described above apply.

Scan ID / Verify Sectors (4xh):

This is a very strange command. As far as I know, it is the only one that is totally incompatible between the old AT's hard disk controller and today's IDE drives. It would appear that this command was never used by common system implementation or application software...

For the WD1010 controller, this is the Scan ID command. It takes no parameters at all (except for the drive and head which originally had to be contained in a register external to the WD1010). When the command is started, the controller searches for the next ID field and reads the contents into the task file. This way the actual drive, head, cylinder, and sector size could be examined. The sector number was also transferred into the task file, so the sector numbering order could be figured out by repeating this command fast enough.

For the IDE drives, this is a completely different command: Verify Sectors. It is similar to the Read Sectors command except that no data is transferred to the host, and the "L" option is not supported. Thus, it needs all parameters in the task file. Up to 256 sectors of data will be read into the sector buffer, and their ECC bytes will be verified. The DRQ flag will never be set. The completion status of the command can be read from the status register.

It is interesting that both types of controller/drive support the retry option - so this is the only compatibility of this command.

Format Track (5xh):

Originally, this command was used to physically format an entire track of the hard disk, exactly as it's done when formatting floppy disks. The Format Track command is started similarly to the Write Sectors command: first the task file must be set up, then the command written to the command register. After that, the drive responds by setting the DRQ flag. The host must then write data into the sector buffer until the DRQ flag is reset. After that, the command is executed.

For the format command, the sector buffer must contain special data. As with the index field array when formatting a floppy disk, it must contain valid sector ID's for every physical sector of the track that will be formatted, beginning at the start of the buffer. Each sector ID in the buffer consists of two bytes. The unused remainder of the buffer is ignored by the format command, but must also be written for the DRQ signal to disappear.

The first byte of each sector ID is a flag byte. The WD1010 knew only two different values for this descriptor:
00h = good sector,

80h = bad sector.

Today's IDE drives offer two more descriptor values:

40h = assign sector to alternate location,

20h = unassign alternate location for this sector.

We'll look further at these values below.

The second byte of each ID is the sector-number byte. It contains the number by which the related sector is referenced later during normal r/w operation. The ID fields in the sector buffer are assigned to the physical sectors (created through formatting) in the order they are stored in the buffer. So it is possible to define an interleave factor by appropriate physical sector numbering. Here is an example:

```
Addr.  00 = 00 01 00 11 00 02 00 12
        08 = 00 03 00 13 00 04 00 14
        10 = 80 05 00 15 00 06 00 16
        etc.
```

Here we see the first 12 (of 32) ID words. The starting sector has number 1 (as usual). The interleave factor is two, since each sector appears two sectors after its logical predecessor. You can also see that sector number 5 (the 9th sector physically) is marked bad.

Due to surface errors on the hard disk, there are some positions where the media won't store magnetic information reliably enough (if at all). The defect list for a particular drive then shows the cylinder, head, and "BFI" (byte from index) value of the defect. People then had to calculate the bad-sector position and number from each of those BFI values. However, it is not commonly known that the relationship between the BFI value and the sector number depends not only on the sector size but also on the interleave factor and the starting sector number...

Again, things changed as the years went by... I already mentioned when introducing the features of modern IDE hard disks, that those drives don't have defect lists any more, due to the usage of internal spare sectors. For compatibility reasons, these drives still accept the Format Track command. However, most drives only simulate its execution - internally they don't really format any track. Modern drives are "hard-sectored" by the manufacturer, with the sector size unchangeable by the user. But by virtually formatting a track, one can assign new sector numbers (for example, starting with 0 instead of 1). However, the sector numbering order is often ignored. Because IDE drives commonly have built-in cache memories, the definition of an interleave factor would make no sense. So, the drive always uses the fixed sector ordering which gives maximum performance in combination with the cache.

To make things still more complicated, the Format Track command of IDE drives allows for the assignment of data sectors to the spare sectors and for the release of those assignments (look at the descriptor bytes above). All IDE drives have some spare sectors to which the data of defective sectors is automatically mapped. Normally, there is one spare sector per track, resulting in about 2-3% spare capacity. This is more than

enough. When a sector appears too unreliable during normal operation, the drive simply marks that sector as bad internally and moves the data to the nearest free spare sector. As long as not all spare sectors are assigned, the user won't notice anything. However, these assignments can also be done explicitly by use of the Format Track command. But it is strongly recommended not to do that! First, one will normally get no defect list for an individual IDE drive containing the BFI positions. Second, even if a sector which was assigned to one of the spares is marked good again, the related spare sector can not be used again! So with every unassignment of a spare sector, you loose that irretrievably.

So we come to this result: with standard (i.e., ST-506) drives and external controller (i.e., WD1010) it makes sense to format the drive in order to freshen the surface magnetism, to get a defined state (sector numbering and order), and to mark defect sectors as bad (so that the operating system can behave accordingly). With IDE drives, it's best to leave them just as they are coming from the factory!

Seek (7xh):

This command is used to move the r/w heads to a particular cylinder explicitly. For normal operation of the drive, it is usually not necessary, since all r/w commands perform implied seeks. However, this command can easily be used for benchmarks to determine the drive's seek times. With the WD1010 controller, the four lower bits of the command byte contain the step rate (described above). IDE drives simply ignore these four bits.

Execute Diagnostics (90h):

This command is common to all IDE drives but not available with the WD1010 controller. When issued, the drive performs an internal self-test. If the drive is a master drive, and a slave drive is connected to it, the master also waits a limited time for the slave to complete its self-test. During all this time, it is busy (the according flag in the status register is set). After finishing the test procedure, its results are placed in the error register. In this special case, the content of the error register has to be considered as a single byte value, not as several bit flags. There are the following error codes:

01h no error detected,

03h sector buffer error.

(These codes are supported by Conner drives. Maybe other manufacturers use more or different codes.)

If the slave drive diagnostics failed, the MSB of the error register is set, leading to values of 8xh. However, even with single drive configurations this bit sometimes is accidentally set. It may be ignored then.

Set Drive Parameters (91h):

An IDE-only command again. After power-up or reset, the drive can immediately be used in its default mode. However,

the drive's logical parameters can be changed by setting them with this command. This way, the drive can be set up to different modes in order to emulate the parameters of another common drive. The task file registers which are used with this command, and the way in which they are used, may differ. Some drives are really flexible and allow any parameters that result in no more than the drive's real capacity. Other drives (for example, my Conner CP-3044) support only two or three modes with fixed parameters. So for their selection, only part of the task file's registers are needed. Most, if not all, drives will accept this command with valid parameters in the SC, C, and H registers (even if not all the parameters are required), defining the number of sectors per track, cylinders, and heads.

Because of the differences, it is advisable to first collect detailed information about the supported emulation modes of a particular drive, before defining its operating parameters. Normally, it's best to operate a drive in its native mode (so the logical parameters equal the physical ones). However, there's another strange detail: there are drives which don't support the native mode! My Conner drive again serves as example: the drive has 1053x2x40 sectors (cylinders by heads by sectors) physically, but supports only a pseudo-native mode with 526x4x40 sectors, and an emulation mode with 981x5x17 sectors (which is for compatibility with older 40 MB drives). Additionally, depending on the internal software version, the drive defaults to the emulation mode or to the pseudo-native mode.

As a result, it is recommended that the operating parameters always be defined after power-up or reset. And to define them, you must have detailed information about the drive you want to use. There is a "Product Manual" for every drive type, describing all those details. Unfortunately, these manuals are hard to get. Most dealers are not willing to give them to their customers (and some even don't have them in stock). The other way is to try out some parameters, starting with the information delivered by the Identify Drive command.

The break - a sample program

I realize that I've already filled quite a few pages again. So I'll make a break here and continue the command descriptions in Part IV of the "Connecting IDE Drives" article series. Instead of continuing now, I'll show you a short program which reads the ID information of an IDE drive within a PC/AT. This sample program was written with Turbo Pascal 5.5 but may easily be used with any version above 4.0.

You can try out this program on your AT (if you have one with an IDE drive) and play with it until receiving the next issue of *TCJ* with Part IV of the article. That part will finish the command descriptions and will also contain some more programming examples and shortform tables as a programmer's overview of the IDE interface definition.

Abbreviation list:

BFI Byte From Index (position of surface defect)
 DRQ commonly used for Data Request (bit flag or signal line)
 IDE Integrated Drive Electronics (hard disk interface type)
 I/O Input/Output
 PC/AT Personal Computer/Advanced Technology (a class of computers)
 r/w read/write
 ST-506 older hard disk interface standard, used between separate controllers and MFM/RLL drives

```

program Get_IDE_ID;
(* Q&D 930903 Tilmann Reh *)
(* 930905 MSDOS *)
(* Reads the ID information of IDE drives and displays it. *)
(* Should run with every IDE/AT harddisk drive. *)
uses crt;
const SignOn = ^m^j^Read IDE ID Info V0.1 TR 930905^m^j;
(* I/O addresses and IDE commands: *)
  IDE_Data      = $1F0;
  IDE_Error     = $1F1;
  IDE_SecCnt    = $1F2;
  IDE_SecNum    = $1F3;
  IDE_CylLow    = $1F4;
  IDE_CylHigh   = $1F5;
  IDE_SDH       = $1F6;
  IDE_CmdStat   = $1F7;
  CMD_Identify  = $EC;
(* Data types and variables: *)
type WorkStr    = string[80];
     BufType    = array[0..255] of word;
     IDRecord   = record
       Config      : integer;
       NumCyls     : integer;
       NumCyls2    : integer;
       NumHeads    : integer;
       BytesPerTrk : integer;
       BytesPerSec : integer;
       SecsPerTrack : integer;
       d1,d2,d3    : integer;
       SerNo       : array [0..19] of char;
       CtrlType    : integer;
       BfrSize     : integer;
       ECCBytes    : integer;
       CtrlRev     : array [0..7] of char;
       CtrlModl    : array [0..39] of char;
       SecsPerInt  : integer;
       DblWordFlag : integer;
       WrProtect   : integer;
     end;
var   SecBuf    : BufType;
     IDR       : IDRecord absolute SecBuf;
     Secs      : real;
     i,j       : integer;

(* Convert byte/word values to hexadecimal strings: *)
function HexByte(x:byte):WorkStr;
const Nib : array[0..15] of char = '0123456789ABCDEF';
begin
  HexByte:=Nib[x shr 4]+Nib[x and 15];
end;
function HexWord(x:word):WorkStr;
begin
  HexWord:=HexByte(hi(x))+HexByte(lo(x));
end;
(* Swaps the bytes of each "word" in string for correct reading. *)
function SwapStr(s:WorkStr):WorkStr;
var   s1 : WorkStr;
     i : byte;
begin
  s1[0]:=s[0];

```

```

  for i:=0 to pred(length(s)) do s1[i+1]:=s[(i xor 1)+1];
  SwapStr:=s1+'<';
end;
(* Show error codes: status register and error register. *)
procedure Error(s:WorkStr);
begin
  writeln(' ',s); Status:=HexByte(port[IDE_CmdStat]);
  ' ',HexByte(port[IDE_Error]));
  halt; end;
(* Wait until drive is ready. *)
procedure WaitReady;
const TimeOut = 5000;
var   i : word;
begin
  i:=0;
  while (port[IDE_CmdStat]>128) and (i<TimeOut) do begin
    delay(1);
    inc(i);
  end;
  if i=TimeOut then Error('WaitReady TimeOut');
end;
(* Wait for data request (DRQ). *)
procedure WaitDRQ;
const TimeOut = 5000;
var   i : word;
begin
  i:=0;
  while (port[IDE_CmdStat] and 8=0) and (i<TimeOut) do begin
    delay(1);
    inc(i);
  end;
  if i=TimeOut then Error('WaitDRQ TimeOut');
end;
(* Send command to drive. *)
procedure IDEcommand(Cmd:byte);
begin
  WaitReady;
  port[IDE_CmdStat]:=Cmd;
  WaitReady;
end;
(* Read sector buffer of drive. *)
function ReadSecBuf(var Buf:BufType):boolean;
var   i : word;
begin
  WaitDRQ;
  for i:=0 to 255 do Buf[i]:=port[IDE_Data];
  ReadSecBuf:=port[IDE_CmdStat] and $89=0;
end;

(* MAIN: read drive's ID information. *)
begin
  writeln(SignOn);
  IDEcommand(CMD_Identify);
  if not ReadSecBuf(SecBuf) then Error('Read Identify');
  with IDR do begin
    writeln('ID constant      : ',Config,' (',HexWord(Config),')');
    writeln('fixed cylinders   : ',NumCyls);
    writeln('removable cylinders : ',NumCyls2);
    writeln('number of heads     : ',NumHeads);
    writeln('phys. bytes per track : ',BytesPerTrk);
    writeln('phys. bytes per sector : ',BytesPerSec);
    writeln('sectors per track    : ',SecsPerTrack);
    writeln('serial number       : ',SwapStr(SerNo));
    writeln('controller revision : ',SwapStr(CtrlRev));
    writeln('buffer size (sectors) : ',BfrSize);
    writeln('number of ECC bytes  : ',ECCBytes);
    writeln('controller model     : ',SwapStr(CtrlModl));
    Secs := int((NumCyls+NumCyls2) * NumHeads *
  SecsPerTrack);
    writeln('total sectors       : ',Secs:1:0);
    writeln('capacity (MBytes)   : ',Secs/2048:1:1);
  end;
end.

```

CONNECTING IDE DRIVES

by Tilmann Reh

Special Feature

Intermediate Users

Part 4: IDE Commands

In part II we covered the basics of the IDE interface in terms of history, concept, hardware, and register structure. In part III I started describing the various commands and parameters of IDE drives. This time I will finish that command description and offer some sample driver routines.

I must apologize!

Sorry for the badly formatted Pascal listing printed with part III in the previous issue of TCJ. Bill had to delete all the empty lines in order to compress it to a single page. Now I know that this doesn't make a program more readable or easier to understand, even if it's written in Pascal. We will try to do this better in the future.

Commands Continued...

We already covered most of the manufacturer-independent commands in the previous part. However, there are three commands not explained yet. Let's get started with the command which was already used in the sample program printed with the previous part -- so you'll now know what you really did there (in case you ran that program).

Identify Drive (ECh):

This command reads some detailed parameter information from the IDE drive. Again, it's invalid for the older (external) controllers. It is started by writing the command code into the command register, and then it executes like a Read Sectors command. The DRQ Flag will be set, declaring that data can be read. After having read a complete "sector" (256 words, 512 bytes) of data, the DRQ flag will be reset and the drive will be ready again. The data consists of the following fields:

Word Adr.	Byte Adr.	Type	Content
0	0	word	Configuration/ID word
1	2	word	Number of fixed cylinders
2	4	word	No. of removable cylinders
3	6	word	No. of heads
4	8	word	No. of unformatted bytes per physical track
5	10	word	No. of unformatted bytes per sector
6	12	word	No. of physical sectors per Track
7	14	word	No. of bytes in the inter-sector gaps

8	16	word	No. of bytes in the sync fields
9	18	word	0
10-19	20-39	20 char	Serial number
20	40	word	Controller type
21	42	word	Controller buffer size (in sectors)
22	44	word	No. of ECC bytes on "long" commands
23-26	46-53	8 char	Controller firmware revision
27-46	54-93	40 char	Model number
47	94	word	No. of sectors/interrupt (0 = no support)
48	96	word	Double word transfer flag (1 = capable)
49	98	word	Write protected
50-255	100-511	-	reserved (read as zero values)

Some of these fields have special meanings. The configuration/ID word consists of 16 single-bit flags. However, I don't know for sure if their meaning is really manufacturer-independent. The "controller type" word is encoded as a number representing a particular type.

Configuration/ID word bit flags:

15	Non-magnetic drive
14	Format speed tolerance gap required
13	Track offset option available
12	Data strobe offset option available
11	Rotational frequency tolerance > 0.5%
10	Data transfer rate > 10 MB/s
9	Data transfer rate > 5 MB/s, <= 10 MB/s
8	Data transfer rate <= 5 MB/s
7	Removable disk
6	Non-removable disk
5	Spindle motor can be switched off
4	Head switching time > 15 us
3	Not MFM encoded
2	Soft sector
1	Hard sector
0	reserved

Controller type word values:

0	Not specified
1	Single ported, single sector buffer
2	Dual ported, multiple sector buffer
3	= 2, with look-ahead read capabilities

The string-type data fields (character arrays) contain plain text information about the serial number, controller model, and firmware revision of the drive. Each word holds two characters,

which must be displayed with the high-byte character first in order to get readable results.

As far as I know, most IDE drives follow the data field description above. However, there still are many things which are manufacturer-dependent. Fortunately, these details are not critical. To give you some examples: The controller model field of Conner drives contains plain text with the complete drive description like

“Conner Peripherals 40 MB - CP3044”.

Seagate's IDE drives offer only a short cryptic ID string, which sometimes doesn't even contain the drive type.

A very interesting difference, even between drives of the same manufacturer, shows up with the “Number of cylinders/heads/sectors” fields. Some drives show their physical values there, independent of the active emulation mode (for example, my CP-3044 does so). Other drives always show the parameters of the active emulation, or those of the default emulation mode. Surprising especially with my drive is that the physical parameters can't be used for drive operation! As a result, the data delivered by this command must be considered carefully. However, it's normally possible to extract useful information by reading the drive's ID information for several different active emulation modes.

Read/Write Sector Buffer (E4h/E8h):

These are the last two common IDE commands. With these commands it's possible to read or write the drive's sector buffer directly. I haven't found any use for these yet, but probably there is (at least was) one. In my opinion, these commands are useless for normal operation.

Block Mode Commands (Read/Write/Set Multiple, C4..C6h):

By the use of these commands, one can access disk data in larger blocks than the physical sector size. Several sectors are grouped together and handled as a block of data. However, many drives don't support this mode. I don't have detailed information regarding the parameters. If a particular drive supports the block mode, the details will surely be printed in its user manual.

Power Commands (E0..E6h, except E4h):

The power commands are not supported by every IDE drive. However, if they are, they are normally compatible. The power commands are commonly used within portable computers (laptops, notebooks, handhelds, or whatever the names are). They allow for automatic or manual changing between normally four operation modes:

Read/Write Mode	(4.2 W)	complete drive circuitry operating
Idle Mode	(2.0 W)	motor running, r/w circuitry turned off while no command is active

Standby Mode	(0.5 W)	motor stopped, r/w circuitry turned off, interface active
Sleep Mode	(n/a)	everything stopped, exit only with reset

The power requirements mentioned in this table are those of my Conner 42-MB drive. While no r/w operation is in progress, the drive normally is in idle mode (also when being reset). Read/write mode is always automatically entered when a r/w command is issued; after completion of that command, the drive enters idle mode again.

When the drive is put into standby mode (manually or automatically, see below), the drive (motor, r/w circuit) is shut down while the host interface remains active. So when a command is issued which requires motor or r/w operation, the appropriate circuitry is automatically switched on again.

Once the sleep mode is entered, there is no way out except for reset by means of hardware or software. This is because even the drive's local processor and interface controller are stopped, so there is no way to communicate with the drive. (However, the task file can still be read.)

As mentioned above, there are six power commands:

Set Standby Mode (E0h), Set Idle Mode (E1h):

The drive will enter the desired mode immediately. There are no parameters required. If the drive already is in that mode, the command will have no effect.

Set Standby (E2h) or Idle (E3h) Mode with Auto-Power-Down:

These commands take a parameter in the sector count register. If that parameter is non-zero, the Auto-Power-Down (APD) feature is enabled (with a zero value, APD is disabled). When one of these commands is issued, the drive immediately enters the desired mode. If APD is enabled, the drive will automatically enter standby mode after being in idle mode without activities for a given period of time. This delay can be specified by means of the parameter for these two commands: the SC register must contain the delay time in counts of 5 seconds. The minimum delay of 60 seconds will be set if the SC register contents is smaller than 12. With a maximum value of 220, the maximum delay is about 18 minutes. These limits again apply to my particular drive; other drives may have other specifications.

Read Power Mode (E5h):

This command reads the actual mode. If the motor is spinning (meaning that the drive is in idle mode), the value FFh will be returned in the SC register. Else (when in standby mode or just spinning up) a zero value will be placed in the SC register.

Set Sleep Mode (E6h):

This command puts the drive into sleep mode immediately.

Every internal activity is terminated and all circuitry switched off.

There are some more power-related commands, having the command codes F8..FDh (except FCh). Their general meaning is similar to the power commands described above (E0..E5h), except that the time delays are specified more exactly (in counts of 0.1 seconds). However, I have not yet seen a drive which supported these commands, and I don't have detailed information about them.

Cache On/Off (EFh):

This is the last command which I will explain here. It is used for enabling or disabling the automatic read-ahead feature (read cache) of the drive. The write precompensation register (WP) is (mis-)used as a parameter register for this command (today, this is the only use of the WP register). If the WP register contains AAh, the feature is enabled; with 55h, it is disabled. Every other value will result in an aborted command error. After reset, the drive defaults to read-ahead feature enabled.

Whew -- this was a lot of stuff! (I hope it was not too hard.) However, now you should know about IDE commands in detail (if you didn't fall asleep while reading). Before we start practical work, here, for the programmers, are the short-form tables that I promised.

Table 1: Task File Registers (as printed in part II)

/CS0 /CS1 A2 A1 A0	Addr.	Read Function	Write Function
0 1 0 0 0	1F0	Data Register	Data Register
0 1 0 0 1	1F1	Error Register	(Write Precomp Reg.)
0 1 0 1 0	1F2	Sector Count	Sector Count
0 1 0 1 1	1F3	Sector Number	Sector Number
0 1 1 0 0	1F4	Cylinder Low	Cylinder Low
0 1 1 0 1	1F5	Cylinder High	Cylinder High
0 1 1 1 0	1F6	SDH Register	SDH Register
0 1 1 1 1	1F7	Status Register	Command Register
1 0 1 1 0	3F6	Alternate Status	Digital Output
1 0 1 1 1	3F7	Drive Address	Not Used

Table 2: Error Register

Bit	Flag	Meaning
7	BBK	Bad block mark detected
6	UNC	Uncorrectable data error
5	-	-
4	IDNF	Sector ID not found
3	-	-
2	ABRT	Command aborted (status error or invalid command)
1	TK0	Track 0 not found during recalibration
0	-	-

Table 3: SDH Register

Bit	Flag	Meaning
7	EXT	Extension Bit. Always 1.
6-5	SIZE	Sector Size. Always 01 (512 byte sectors).
4	DRV	Drive bit. Master/single drive = 0, slave = 1.
3-0	HEAD	Head field. Binary head number 0..15.

Table 4: Status Register, Alternate Status Register

Bit	Flag	Meaning
7	BSY	Drive busy. Task file cannot be accessed.
6	DRDY	Drive ready (up to speed and ready for command).
5	DWF	Drive write fault.
4	DSC	Drive seek complete (actuator on track).
3	DRQ	Data request (ready for data transfer).
2	CORR	Corrected data (bit is set when data has been recovered by use of ECC).
1	IDX	Index. Active once per disk revolution.
0	ERR	Error. See other bits and error register.

Table 5: Digital Output Register

Bit	Flag	Meaning
2	SRST	Software reset (active when set to 1).
1	/IEN	Interrupt enable (active when set to 0).

Table 6: Drive Address Register

Bit	Flag	Meaning
7	-	not driven (for PC floppy compatibility)
6	/WTG	Write gate (active when 0)
5-2	/HSx	Head select 3..0, one's complement of active head
1	/DS1	Drive 1 selected (active when 0)
0	/DS0	Drive 0 selected (active when 0)

Table 7: Commonly needed Commands with Parameters

Code	Command	Parameters
1x	Recalibrate	D
20	Read Sectors with retry	SC,SN,C,D,H
30	Write Sectors with retry	SC,SN,C,D,H
40	Verify Sectors with retry	SC,SN,C,D,H
50	Format Track	C,D,H
7x	Seek	C,D
90	Exec Diagnostics	D
91	Set Drive Parameters	SC,(C),D,H
Ex	Power Commands, see below	
E4	Read Sector Buffer	D
E8	Write Sector Buffer	D
EC	Identify Drive	D
EF	Cache On/Off	D,WP

Power Commands:

E0	Standby Mode	-
E1	Idle Mode	-
E2	Standby Mode with APD	SC
E3	Idle Mode with APD	SC
E5	Read Power Mode	(SC)
E6	Sleep Mode	-

Table 8: Error Conditions

When an error occurs, the error flag in the status register (ERR) is always set. For the different groups of commands, the following status/error flags are valid then:

Recalibrate	ABRT, TK0, DRDY, DWF, DSC
Read, Verify	BBK, UNC, IDNF, ABRT, DRDY, DWF, DSC, CORR
Read Long, Write, Write Long	BBK, IDNF, ABRT, DRDY, DWF, DSC
Format, Seek	IDNF, ABRT, DRDY, DWF, DSC
Diagnostics, Initialize, R/W Buffer, Identify, Set Cache	ABRT
Invalid command	ABRT

Table 9: Interrupt Conditions

The drive generates an interrupt (if enabled) under the following conditions:

Recalibrate	after successfully reaching track 0
Read	each time DRQ is set
Write	when DRQ is set, from second sector on (only when multiple sectors are written)
Verify	after completion for all sectors
Format Track	after completion
Seek, Initialize, Power Commands (except Sleep)	after command is issued/initiated
Set Sleep Mode	when drive is in sleep mode
Read Buffer, Identify	when data is ready for reading

Now let's come to the example routines for accessing an IDE drive. These examples are given as Turbo-Pascal (3.0) source (based on my IDE test program). They apply to the use of my IDE interface board (described in TCJ #56), so there always are 512 data *bytes* transferred instead of 256 data *words*.

In all examples, named constants are used for accessing the IDE registers at their particular I/O addresses. These named constants must be declared elsewhere. Their names are derived from the related IDE register names and IDE commands.

The examples are programmed in a very modular fashion so that they are easy to understand. For implementation in a system BIOS, for example, most of the subroutines will contain so little code that the complete read/write routines will normally be coded inline. In addition, a real implementation, unlike these examples, will have time-out functions in most loops. If someone is interested in the IDE driver of my CPU280 system BIOS, please contact me (however note, it's Z280 assembly language and commented in German).

1. General access: Wait for drive ready / wait for data request

In Pascal, two small procedures serve this purpose. In assembly language, I use two macros instead, because the subroutine calling overhead would be too much.

```
procedure Wait_Ready;
begin
repeat until port[IDE_CmdStat] <= 128;
end;
```

```
procedure Wait_DRQ;
begin
repeat until port[IDE_CmdStat] and 8 <> 0;
end;
```

2. General access: Command issue

```
procedure IDE_Command(Cmd: byte);
begin
Wait_Ready;
port[IDE_CmdStat] := Cmd;
end;
```

3. General access: Reading/Writing the sector buffer

In the Pascal implementation, the two functions return a Boolean value which is true if there were no errors during r/w of the buffer.

Both routines require the drive to be ready for data transfer!

```
function Read_SecBuf(var Buf: BufType): boolean;
var i : integer;
begin
Wait_DRQ;
i := port[IDE_Data]; (* specific to my IDE interface board *)
for i := 0 to 511 do Buf[i] := port[IDE_Data];
Read_SecBuf := port[IDE_CmdStat] and $89 = 0;
end;
```

```
function Write_SecBuf(var Buf: BufType): boolean;
var i : integer;
begin
Wait_DRQ;
for i := 0 to 511 do port[IDE_Data] := Buf[i];
Wait_Ready;
Write_SecBuf := port[IDE_CmdStat] and $89 = 0;
end;
```

4. General access: First access, initialization

```
procedure HD_Init(Cyls,Heads,Secs:integer);
begin
port[Dig_Out]:=6;
delay(10);      (* Drive Software Reset *)
port[Dig_Out]:=2;
Wait_Ready;
port[IDE_SecCnt]:=Secs;
port[IDE_CylLow]:=lo(Cyls);
port[IDE_CylHigh]:=hi(Cyls);
port[IDE_SDH]:=pred(Heads)+$A0;
IDE_Command(Cmd_Initialize);
end;
```

5. Data access: Single sector read

```
function HD_ReadSector(Cyl,Head,Sec:integer; var
Buf:BufType):boolean;
begin
Wait_Ready;
port[IDE_SecCnt]:=1;
port[IDE_SecNum]:=Sec;
port[IDE_CylLow]:=lo(Cyl);
port[IDE_CylHigh]:=hi(Cyl);
port[IDE_SDH]:=$A0+Head;
```

```
IDE_Command(Cmd_ReadSector);
HD_ReadSector:=Read_SecBuf(Buf);
end;
```

7. Data access: Single sector write

```
function HD_WriteSector(Cyl,Head,Sec:integer; var
Buf:BufType);
begin
Wait_Ready;
port[IDE_SecCnt]:=1;
port[IDE_SecNum]:=Sec;
port[IDE_CylLow]:=lo(Cyl);
port[IDE_CylHigh]:=hi(Cyl);
port[IDE_SDH]:=$A0+Head;
IDE_Command(Cmd_WriteSector);
HD_WriteSector:=Write_SecBuf(Buf);
end;
```

Now we have reached the end of the “behind IDE” article series. In another column I will describe my revised IDE interface board for the 8-bit ECB bus in somewhat more detail than in TCJ #56. This will include a TTL equivalent of the GAL contents, for those who are inexperienced in reading a Boolean equation design, or who want to build it up using discrete logic.

For a list of abbreviations, see parts II and III of this article.

CONNECTING IDE DRIVES

by Tilmann Reh

Special Feature

Intermediate Users

Part 5: GENERIC IDE

Generic Z80 IDE Interface

(plus a few words about the LittleNet interface)

Discussions about connecting IDE devices like hard disks or eventually CD-ROM have grown during the last months. Seemingly we triggered something with the articles about the interface basics and my interface board for the ECB-Bus. Another actual development was the single-chip IDE interface by Claude Palm (of Palmtech in Australia) who also offered an S-100 board draft containing his chip.

This in turn started a discussion between *TCJ* columnist "Dr. S-100" (Herb Johnson), Johnathan Taylor from England (with whom I discussed Wayne Sung's interface version for the QX-10 before), and me. Our main theme was how to reduce the total cost of an IDE interface so it will be of more general interest than the relatively expensive S-100 board.

I finally made a circuit and PCB draft for an interface which will directly connect to a Z80 processor. This has two great advantages: First, the board is very small, thus PCB prices will be much lower than for an S-100 board. Second, the direct connection to the Z80 processor socket opens the way to ALL Z80 computers, not only those based on the S-100 bus (or ECB-Bus, like my interface described earlier in *TCJ*). So we might get a larger volume, further reducing PCB costs. Of course, it also has a disadvantageous side: the mechanical construction is left to the end user.

However, it has to be made clear that this is only a draft yet. We are now trying to determine if there is enough interest in this generic interface to build a prototype and produce a run after it really works. But before we go further on this, here are some technical details:

The new interface (I call it GIDE, for Generic IDE) consists of a GAL chip (Generic Array Logic, a smaller programmable logic device) and a few TTLs. Its size is about 60 x 70 mm (2.4 x 2.8 in), including all connectors. It plugs directly into the Z80 socket via two pin strips, or onto a cable which plugs into the Z80 socket. The processor itself will normally be plugged in the appropriate socket on the interface board. The interface is I/O mapped, with user selectable base address (in increments of 10h). The IDE drive will be connected via flat cable. This cable and of course the hard disk drive will not be included with the interface.

Quoting Herb (from a message he placed in the nets): "It would be encouraging to all involved with this design to know who would buy it, what they would require for support, and how much they would pay for it. Clearly, this is not a very "commercial" venture: there are too few Z80 systems around, and few Z80 or CP/M vendors for "reselling" for a commercial effort.

But, to avoid losing money and to get a reasonable quantity produced, we need to set a fair and acceptable price. Too cheap a price will make it unreasonable to make; too large, unreasonable to buy. Please use your honest judgment and suggest what YOU would spend, and what you would expect."

Meanwhile, Herb and Johnathan started lists of interested people who responded to their messages in the nets. We also discussed how to handle further support like board testing (for those who would like a kit) and driver software (example routines, test programs etc.). But before we spend more time and money on this topic, we really should know if there is enough interest to justify all those expenses. So all you who are interested, please contact one of us (see addresses below) with the above requested information. We really would be glad to make this board! If there is interest, I could also offer to describe this new interface in another article here in *TCJ*.

For those who are already active in getting used and/or cheap hard disks, here is an advice: I strongly recommend Conner drives, since those drives have the lowest power consumption and also generate the least noise. Some other makers like Quantum and Sony are comparable in noise emissions. Seagate drives are horrible in both terms, and additionally have slightly different timing specifications which sometimes may lead to problems. So if you have the choice, try to get Conner drives.

Actual state of the LittleNet development:

Since my circuit draft for the isolated RS-485 interface converter was printed in *TCJ* #69 (in the column of Rick Rodman), we further discussed some technical details of this interface. Soon I will finish a PCB draft which probably will be single-sided, so the PCB will be easy to make in home cellars. The parts costs will also be very low. We will surely inform you about further results and also print the PCB artwork in *TCJ*. If any of you have some suggestions or questions, please contact Rick or me as soon as possible, so we can consider your thoughts in PCB layout and/or software details.

Contacts:

Tilmann Reh, Am Rueckelchen 5a, 57078 Siegen, Germany

InterNet: tilmann.reh@hrz.uni-siegen.d400.de

Fax (at work): +49 271 484520

Herbert R. Johnson, CN5256 #105, Princeton, NJ 08543, USA

InterNet: hjohnson@pluto.njcc.com

Voice/FAX +1 609 771 1503 (8am-11pm EDT)

Johnathan Taylor, UK

Internet: jet@centron.com, Fidonet: 2:2501/307.9

Rick Rodman, USA

InterNet: rickr@aib.com

Special Feature
Intermediate Users
IDE Part 6

GIDE

by Tilmann Reh

IDE again, plus words about the LittleNet interface

In *TCJ* issue #71 I already gave a brief description of my current "tiny-hardware" project, the Generic IDE interface board (GIDE). After building up a prototype and getting it running, I was told about some additional wishes of many (yet potential) users. And, as usual with hardware developments, it was necessary to modify the circuit due to a (probably common) oddity of the computer I used for testing the new interface (this was a Triumph-Alphatronic PC-8, a german Z80-based homecomputer running a ROM-BASIC and CP/M-2.2. Hopefully, I will report this development history and the current circuit in more details later - for now, another short description must be enough. As like most developers, I have just too little time for too much work...

Just to give you newcomers an impression about what we're discussing now: The GIDE board is a small (yes, small) PCB which plugs directly into any Z80 CPU socket and contains an I/O-mapped interface for connection to any IDE harddisk drive. The Z80 processor must be removed from the socket and plugged into the interface (daughter-) board. There should be an original-sized drawing of the PCB layout (parts locations) somewhere around this article.

For those that cannot remove the processor because it is directly soldered in, or if there's no sufficient space to plug in the GIDE board, another method of connection is provided. Just solder a DIP socket onto the processor (first case only), and connect it to the interface by a flat ribbon cable which is plugged into a header on the solder side of the IDE interface. However, care should be taken to make that cable as short as possible! And for some computers, it might be necessary to cut a PCB trace on the motherboard and connect a flying lead to the interface board.

Though I can't fully describe the current circuit now, let me say something about my design criterias. In the previous issue of *TCJ*, Claude Palm wrote about PLDs and mentioned that he prefers to put *all* logic circuitry into the PLD. I don't share this viewpoint. In my experience, it is most often better to combine PLDs and "discrete" logic. This has some advantages: First, the result normally is much cheaper (especially for small quantities); second, by using smaller PLDs you get cheaper development tools and are not dependent on any PLD manu-

facturer; third, you have more PCB layout flexibility since you can locate the small ICs at different locations. I agree that sometimes, especially for large-volume products, it can make sense to use a larger PLD which contains it all - but for this purpose the small ones are definitely better.

The GIDE design uses two standard GALs and two standard 74-series bidirectional registers (74 HCT 646). That's all. For those who want an additional real-time clock (RTC), a socket is provided for it, with a fully decoded chip-select signal. The RTC can be battery-backed by connecting a battery to a 3-pin header. The main design rule was to keep it simple and cheap. Hopefully you agree we met the goal!

One of the two GALs (20V8) does all the address decoding, and provides the necessary chip-select signals for the IDE drive and the RTC. The interface occupies 11 I/O-addresses out of a 16-bit address area. The base address (upper four address bits) can be selected by four jumpers. We also could have fixed the base address by GAL programming (it would have been a 16V8 instead of a 20V8 then). But we decided that the user-selectable base address is a real *must* for this interface - we can't provide differently programmed GALs all around the world!

The second GAL (16V8) contains the complete IDE-interface state machine. Its content is roughly comparable to that used in the IDE interface for the ECB bus (described in *TCJ* #56).

When I first introduced the GIDE project, I included the information that we needed some expressed requests before it makes sense to produce PCBs. There are rather high fixed costs for making PCBs, so a minimum number is required to keep costs reasonable low. From our requests at different networks and user groups, we have about 40 people who are interested in getting a board. Most of them mentioned reasonable prices

they would pay, so here we are: the board will surely be made, and I can offer it at the following prices:

bare PCB	DM 29	\$ 22
PCB & programmed GALs	DM 43	\$ 32
complete kit, w/o RTC	DM 69	\$ 51
complete kit, with RTC	DM 84	\$ 62
assembled unit, w/o RTC	DM 84	\$ 62
assembled unit, with RTC	DM 99	\$ 73

These prices are calculated on the base of DM, assuming today's exchange ratio of about DM 1.35 per US-\$ (which is the lowest ratio we ever had!). Probably Herb Johnson will import some bare boards into the USA, and get the other parts there - this will reduce import fees and exchange losses. Hopefully, Herb will be able to offer the same prices and still get some handling fee. And, if the number of orders grows, the price will get lower (remember PCB fixed costs). However, then I need to know this before I order the PCBs (this means: today when you read this!).

If you are interested, or want to place an order, send mail to Herb Johnson, Johnathan Taylor, or me (for addresses see below).

Now lets talk a bit about software aspects. The GIDE interface will physically allow access to the IDE drive, but some software is also needed to put the proper data into the drive's registers and to transfer disk data to and from the drive. I will offer my IDETEST software (which hopefully will be somewhat enhanced soon) which enables direct access to the harddisk drive and also contains some test routines for the drive (like linear and random read/write etc.). This will be good for some fundamental tests of the controller and the drive itself. When all IDETEST checks are passed without errors, the hardware can be assumed to be perfectly running.

You'll then need some software which brings IDE access to your operating system. For CP/M (which is most common for Z80 computers), this means related BIOS routines. I will provide sample routines which show how to initialize and access the IDE drive in terms of physical-sector BIOS routines. It will be up to you to implement them into your BIOS. Especially for the more common computer types, there surely will be someone which does this for the whole community. I (or Herb, or Johnathan) might perhaps serve as a "knowledge-base" of what implementations already are done.

Not all Z80-CP/M computer owners have the sources of their BIOS, or the system generation tools required to create a bootable system from the sources. Then the only reasonable way to support any additional disk device, is by using a loadable driver which relocates straight below the operating system and stays resident there (those PeeCee people call this "TSR" - terminate and stay resident). However, this principle has major drawbacks. The worst is that you can't use anything of the existing BIOS, so the complete deblocking routines and the sector buffers must be implemented again, significantly reduc-

ing TPA size. But I hope that someone will write such a generic IDE driver as CP/M-2.2 TSR - I can't provide this by myself since I am using CP/M-Plus only.

Eventually such a generic loadable driver will only be used to check out the new "harddisk feeling" and perhaps will motivate someone to disassemble the original BIOS routines. Once you have the BIOS sources, you are free - everything else is available in the public domain! For keeping track about what's done in this concern, please always inform one of us about current developments!

Finally, some new words about the LittleNet adaptor board:

I promised to do a PCB layout which contains the LittleNet adaptor (an isolated RS-232 to RS-485 converter) which was introduced as a draft in *TCJ* #69. After some discussions with Rick Rodman about the optimum connector types for both sides of the adaptor, we decided to use standard ITT-Cannon DE-9 connectors (one male, one female). The pinout of the RS-232 side will match a standard PC/AT serial connector, and the pinout of the RS-485 side allows for using twisted-pair ribbon cable with crimped connectors (though it's better to use round shielded twisted-pair cable and solder it to the connectors). All adaptors are supplied via the network cable, by YAWT (yet another wall transformer) delivering something between 8 and 20 volts DC.

The original-sized artwork and the placeplan of the single-sided PCB should be printed somewhere near this, along with the current schematics. If there is sufficient interest in PCBs, I could also make a run. If you are interested, contact Rick or me.

Contacts:

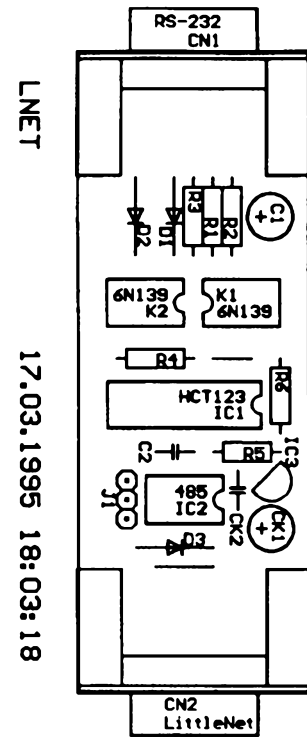
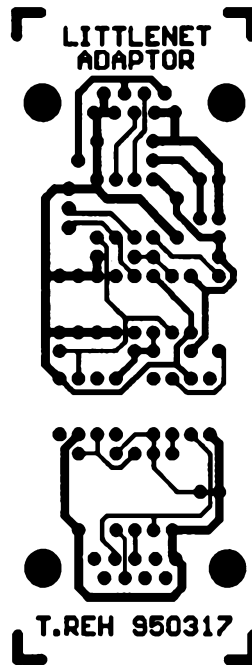
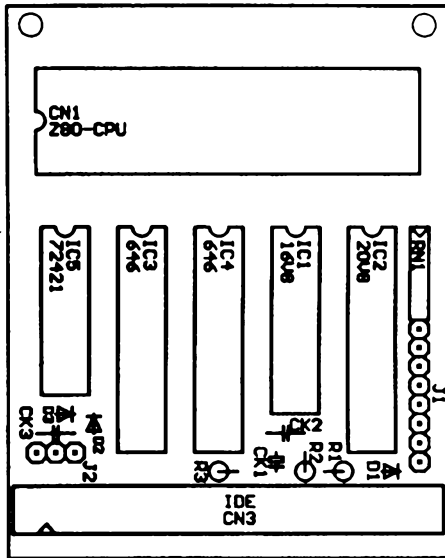
Tilman Reh, Am Rueckelchen 5a, 57078 Siegen, Germany
InterNet: tilmann.reh@hrz.uni-siegen.d400.de
Fax (at work): +49 271 484520

Herbert R. Johnson, CN5256 #105, Princeton, NJ 08543, USA
InterNet: hjohnson@pluto.njcc.com
Voice/FAX +1 609 771 1503 (8am-11pm EDT)

Johnathan Taylor, UK
InterNet: jet@centron.com
Fidonet: 2:2501/307.9

Rick Rodman, USA
InterNet: rickr@aib.com
BBS: +1 703 330 9049 (24h)

GIDE LAYOUT (actual size)



LittleNet isolated RS-485 converter

950421 Tilmann Reh

