by Alexander Pozhitkov (USA)

The NakedCPU (Part 1)

Hardware Experiments and a Roadmap for Navigating Documentation

The first part of this two-part series provides an overview of the NakedCPU. This platform is designed to provide full access to hardware and a CPU without any operating system restrictions while working in the protected mode.

Electronic projects involving microcontrollers are popular among design enthusiasts and professionals alike. Many interesting applications have been made with microcontrollers, and programming them can be a lot of fun. At the same time, the powerful central processing unit (CPU) found in the personal computer (PC)—which serves every electronics designer on a daily basis (including microcontroller programming)—is lacking such attention. Most experimentation with a PC is limited to developing high-level code software with the aid of numerous libraries and technologies hiding the hardware beneath layers and layers of code. Unlimited experimentation with PC hardware is rarely possible. However, you have to install drivers, enabling some access to hardware, because the operating system (OS) naturally does not permit us to do any low-level activities. The sad part is that such drivers are mysterious themselves. It is safe to say hardware programming was well known to many computer professionals and enthusiasts in the 1980s. Later, many people forgot about it, while the technology tremendously leapt ahead. In this article, I try to bridge the gap in time and revive interest in hardware programming based on state-of-the-art technologies and concepts. There is a Russian saying, "Everything new is actually well-forgotten old."

WHAT IS THE NakedCPU?

This article is a result of my interest in the Intel CPU, chipset, I/O controller, and other essential PC devices from the perspective of low-level hardware programming unobstructed by an OS and drivers. My motivation was to reach out to people with inquisitive minds who would appreciate the possibility to directly experiment with the CPU, chipset, and other hardware. Here I'll present the NakedCPU, which provides full access to hardware and a CPU without any restrictions imposed by the OS. Importantly, the processor isn't obscured by Linux, DOS, or Windows, and is operating in its most interesting and powerful mode—the protected mode. In this article, the users are referred to as inquirers, because the NakedCPU is made for researchers (i.e., devoted geeks) rather than regular users.

My aim is to provide you inquirers with some help navigating hardware documentation, which is confusing and otherwise difficult to find. I won't restate the documentation because many computer concepts and technologies quickly become obsolete. With this article, however, you'll find it easier to follow the newer technologies and documentation.

PROCESSORS & PLATFORMS

Think for a moment about one of the modern Intel CPU varieties: the Intel Core 2 Duo processor. This impressive

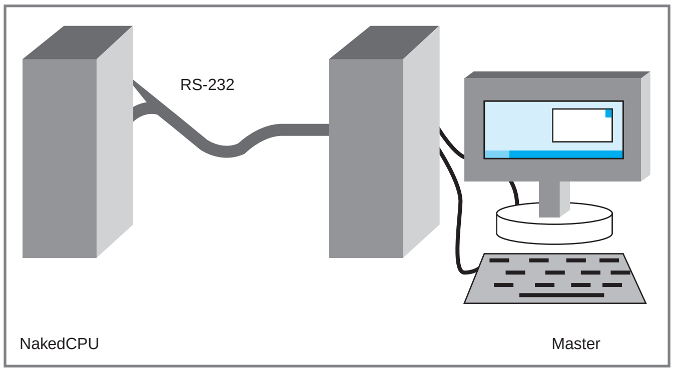


Figure 1—The NakedCPU and master computers are connected via a serial interface. The NakedCPU does not need a keyboard or a monitor.

Description	MBR Location
Determining the current address while the processor is still in real mode after power on. BIOS has loaded the MBR somewhere into the memory and transferred control to our code. The current address is necessary to locate physical address of the pseudo-descriptor, which is in turn defining a physical address and a limit for the Global Descriptor Table (GDT).	0x3E–0x4D
LGDT instruction (Load GDT register) is loading pseudodescriptor, which is pointing to GDT.	0x52
GDT and Interrupt Descriptor Table (IDT) are copied into a new memory location, beginning from linear address 0x0. GDT and IDT are defining memory segments for the processor to operate in protected mode.	0x57–0x64
The MBR contains a very tiny 32-bit protected mode "operating system" named the NakedOS.	0x80–0x186
The NakedOS is copied into a new memory location beginning with linear address 0x800.	0x65–0x71
Switching into protected mode is accomplished by adjusting the machine status word using a LMSW instruction.	0x72–0x78
Transfer control to the NakedOS.	0x7B
Set up 8259 interrupt controller.	0xF5–0x105
Transfer control to the inquirer's executable.	0x106

Table 1—Anatomy of the MBR

processor is capable of consuming up to 75 A of current![1] It is not a simple processor. Its documentation consists of five volumes with the total page count of approximately 3,800 pages.[2] The Intel CPU does not operate alone. It is interfaced to a chipset known as a graphics and memory controller hub (GMCH). The chipset on the other side is connected to an I/O controller hub (ICH). Interestingly, this arrangement is analogous to your nervous system with a brain, a brainstem, and a spinal cord. GMCH and ICH are processors themselves, containing hundreds of configuration and control registers. The documentation on GMCH and ICH spans more than 1,400 pages.[3, 4] It's no wonder OSes hide actual hardware under a thick blanket of intermediate code!

It is not easy to experiment with the Intel CPU given the complexity of surrounding hardware, such as the chipset, network controller, and so forth. Another difficulty is the fact that the documentation is filled with electrical engineering abbreviations and concepts. Also, pervasive layers of OS code interfere with truly free exploration.

The NakedCPU is an experimental platform exposing the hardware internals of a PC. Experimentation with NakedCPU requires two computers (see Figure 1). One is the master computer with Windows and Visual Studio software, whose job is interacting with us and the second computer. The second computer—the NakedCPU—is connected to the master via an RS-232 interface.

The NakedCPU computer is booted up with a small amount of start-up code (available on the *Circuit Cellar* FTP site), which enables it to communicate via RS-232 with the master. Upon start-up, the NakedCPU expects two separate packages of bytes: one is a stream of Intel CPU opcodes to be executed (i.e., the executable) and the other one is the data to be processed. The executable can modify any part of memory, chipset registers, and so forth, and even overwrite the start-up code. In other words, the freedom is yours.

STARTING THE NakedCPU

The NakedCPU won't run without some sort of a start-up code. At start-up, two tasks must be accomplished: switch the CPU into the Protected mode and begin listening on the serial port for two packets of bytes, executable and data. The easiest way to supply the NakedCPU with a start-up code is to prepare a bootable floppy disk with your own code. Certainly, one can also put this code onto a hard drive.

The start-up code (up to 512 bytes) was written in Assembly language and must be stored in sector 0—that is, the master boot record (MBR) of the disk. The Assembly language compilers and linkers, such as MASM, were not used to prepare the MBR due to various difficulties, although they may be suitable as well. There is, however, a binary editor, called HexIt, which among other things enables direct conversion of Assembly commands into binary code.[5] Using this editor, a binary file of the future MBR was created. The file's content is available on the *Circuit Cellar* FTP site. Refer to Table 1 and Table 2 for more details about the content.

A small utility, Firstsectwrite.exe (also available on the *Circuit Cellar* FTP site), was written to transfer this file into sector 0 of the disk. Although the code of this utility is quite simple, it deserves some attention. A Windows API call to `CreateFile(TEXT("\\\\.\\A:")...)` opens raw communication with a disk—a floppy drive A in this case—to enable writes into the sector 0. It is important to note that this call will be only be successful under the administrator account.

I used a Dell Optiplex 760 computer to conduct the experiments. It had a floppy drive attached via a USB. BIOS start-up options enabled me to boot up the computer from such a drive.

THE NakedOS

It may sound contradictive to the spirit of the article to be OS-free; however, the NakedCPU is booted up with a tiny (262 bytes long) 32-bit "operating system," the NakedOS, which enables the NakedCPU to communicate with the outside world via a serial port. In fact, I did not compromise my principle of truly free exploration, because the NakedOS

Structure	MBR Location
Pseudo-descriptor IDT	0x194
Pseudo-descriptor GDT	0x1BA
Null descriptor	0x1C0

Table 2—Critical data structures

Segment	Base	Size	Descriptors, type
Extended memory	0x100000	~128 Mb	0x28, data
Screen, character mode	0x0B8000	4 Kb	0x20, data
Target executable	0x93B	64 Kb	0x30, code 32 0x38, data
NakedOS	0x800	315 bytes	0x10, code 32
Stack	0x400	1024 bytes	0x18, stack 32
System data IDT: 0x3FF–0x200 GDT: 0x1FF–0	0x0	1,024 bytes	0x8, data
Interrupts	**Info**		
INT 0x20	Read a packet from serial port; destination ES:[EDI]; mandatory condition DS=ES. First 4 bytes of the packet indicate in bytes the length of the subsequent string. Upon return, ECX contains the number of bytes received.		
INT 0x21	Send to serial port a string of ECX bytes, located at DS:[ESI].		
IRQ0	Hardware interrupts base vector is 0x28.		

Table 3—Memory segments and interrupts defined by the NakedOS

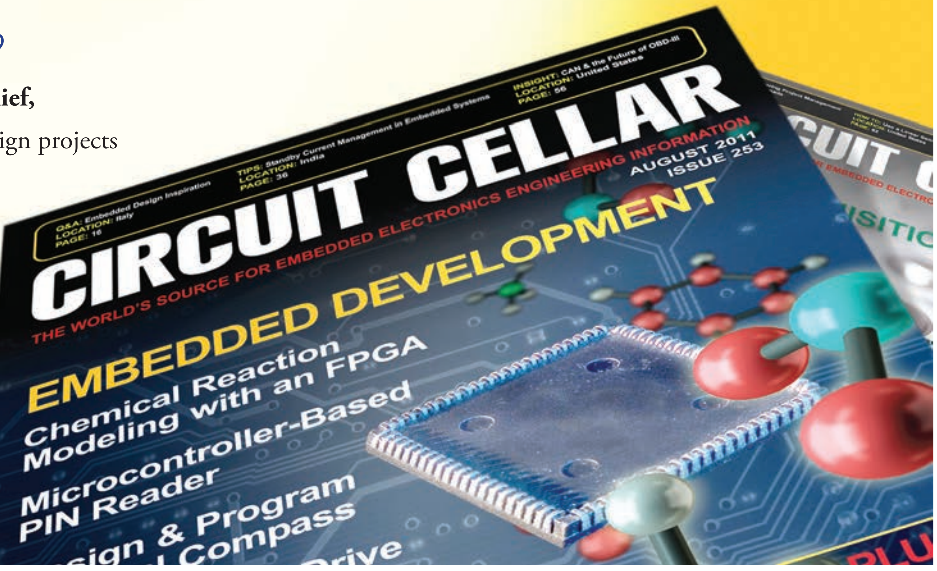
is absolutely transparent. Its code is available on the *Circuit Cellar* FTP site.

The NakedOS defines several memory segments, which are useful as an initial environment for the inquirer's executable (see Table 3). Intel documentation provides an explanation for protected mode memory segments, global descriptor table (GDT), and interrupt descriptor table (IDT).[2] In addition, the NakedOS defines two software interrupts and a base vector for hardware interrupts. Note that the IDT interrupts have nothing to do with DOS or BIOS; they are solely defined by the code.

Immediately after start-up, the NakedOS expects two transactions: one for the executable code and another for data. Each transaction is a stream of bytes sent via the

Byte index	0	1	2	3	0	1	2	…	N-1
Description	Length, N				Executable code or data				

Table 4—The format of NakedOS transactions. The first 4 bytes indicate the length of the subsequent byte stream.


CIRCUIT CELLAR
THE WORLD'S SOURCE FOR EMBEDDED ELECTRONICS ENGINEERING INFORMATION
AUGUST 2011 ISSUE 253
EMBEDDED DEVELOPMENT
Chemical Reaction Modeling with an FPGA
Microcontroller-Based PIN Reader

Listing 1—A fragment of a constructor for a class derived from NakedCPUcode

```
...
DWORD pe, ps;
__asm
{
       mov pe, offset end            // mandatory label end
       mov ps, offset start          // mandatory label start
       jmp end                       // master jumps over the code
       /////////////////////////
       start:     mov ax, 0x28       // loading
                  mov es, ax         // data and
                  mov ds, ax         // stack
                  mov ax, 0x18       // segment registers
                  mov ss, ax         // initializing
                  mov esp, 0x3fe     // stack pointer
                  //...
                  //here goes executable code
                  //...
                  _emit 0xEA  // this is
                  _emit 0x00  // a long jump to
                  _emit 0x00  // finish the executable
                  _emit 0x00  // and transfer control
                  _emit 0x00  // to NakedOS
                  _emit 0x10  // (if it is still
                  _emit 0x00  // in the memory)
       end:       nop
}
...
```

RS-232 (see Table 4). The first transaction is written into the memory segment "target executable," while the second transaction goes into the "extended memory" segment. After the second transaction, the NakedOS transfers control to the executable by a long jump: jmp 00030:000000000. From that moment, in principle, any memory occupied by the NakedOS can be overwritten by the activities of your executable.

The hardware interrupts are normally masked when the NakedOS is running; however, the 8259 interrupt controller is set up (refer to the *Circuit Cellar* FTP site) to handle them if you decide to unmask them. Detailed instructions on programming the interrupt controller are provided in the documentation for the ICH.[4]

THE NakedCPU EXPLORER

An important issue remains: sending an executable to the NakedCPU to conduct experiments. At the beginning of this article, I said two computers are involved. The master has a Microsoft Visual C++ project, the NakedCPU Explorer, which acts as a "shell" that enables the inspection and modification of chipset registers and memory. The code defines a class having a constructor, which provides __asm{} brackets for you to fill with executable code (see Listing 1).

"Immediately after start-up, the NakedOS expects two transactions: one for the executable code and another for data. Each transaction is a stream of bytes sent via the RS-232. The first transaction is written into the memory segment 'target executable,' while the second transaction goes into the 'extended memory' segment. After the second transaction, the NakedOS transfers control to the executable by a long jump: `jmp 00030:000000000`."

Since Visual C++ is running on the master PC with an Intel CPU, the compiler translates the Assembly code into appropriate opcodes, which are naturally suitable for the NakedCPU! Specifically, this class is derived from another class, `NakedCPUcode`, which performs preparatory work by extracting the opcodes produced from the code in the `__asm{}` brackets and making them available for sending over to the NakedCPU. Note that the NakedCPU only receives the code between `start` and `end` labels. It is important to understand that the master computer will not execute the code in the `__asm{}` brackets; it simply jumps over it. The strange keyword `_emit` enables the direct placement of opcodes by their hexadecimal values. For some reason, a long jump is not permitted when using the Visual Studio compiler.

The project also defines a class `SerialComm` and a function `SendNakedCPUdataRecvResponse` to send and receive data. It is worthwhile to examine the project's straightforward code to understand the details of communication with the NakedCPU. Besides serving as an example, the NakedCPU Explorer sends an executable to the NakedCPU, which permits the interactive examination and modification of various chipset and I/O controller registers. The NakedCPU Explorer offers eight commands: `write`, `write32`, `read`, `read32`, `pci`, `memread`, `memwrite`, and `quit`. The first four commands will ask for a port address (i.e., an address in the CPU I/O space). With these commands, the NakedCPU will write to and read from a GMCH or ICH register, 1 or 4 bytes. The fifth command will ask for Bus (decimal), Device (decimal), Function (decimal), and Register (hexadecimal) values. Their values will be packed into the port 0xCF8 to open a "window" into the PCI configuration space that's accessible via port 0xCFC. Details on addressing PCI devices are provided in the chipset documentation.[3] The `memread` and `memwrite` commands enable the reading and writing of double words from and to the memory, respectively.

A regular PC, ubiquitous in most homes, is filled with powerful and interesting hardware. Unfortunately, it tends to be difficult to experiment with PCs due to the lack of documentation and overly protective OSes. The first part of this article detailed at the path to the hard-to-find documentation. I also described the NakedCPU, which is my OS-free platform for experimenting with a PC's internals.

NEW EXPERIMENTS

The NakedCPU is controlled from another computer, the master, which provides you with an interface. In the next part of this series, I'll describe how to use the NakedCPU Explorer for experiments with the speaker, parallel port, and LAN adapter. In addition, I'll give you a peek at the BIOS—the power-on code in particular. Undoubtedly, the suggested experiments will be stepping stones to help you begin even more interesting research.

Author's note: The NakedCPU Explorer does not use any hidden "helper" drivers or libraries. The code is entirely transparent for the inquirer's perusal.

Dr. Alexander Pozhitkov (pozhit@uw.edu) has an MS degree in Chemistry and a PhD in Genetics from Albertus Magnus University in Cologne, Germany. He has been working for 12 years on interdisciplinary research involving molecular biology, physical chemistry, software, and electrical engineering. Currently, Dr. Pozhitkov is a researcher at the University of Washington, Seattle. His technical interests include hardware programming, vacuum tubes, and high-voltage electronics.

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2012/259.

REFERENCES

[1] Intel Corp., "Intel Core 2 Duo Processor E8000 and E7000 Series Datasheet," 2009, http://download.intel.com/design/processor/datashts/318732.pdf.

[2] ———, "Intel 64 and IA-32 Architectures Software Developer's Manuals," www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[3] ———, "Intel 4 Series Chipset Family, Datasheet" 2010, www.intel.com/Assets/PDF/datasheet/319970.pdf.

[4] ———, "Intel I/O Controller Hub 10 (ICH10) Family, Datasheet," 2008, www.intel.com/content/www/us/en/io/io-controller-hub-10-family-datasheet.html.

[5] M. Klasson, "HexIt – The Hex Editor," 2011, http://mklasson.com/hexit.php.

RESOURCE

E. Nisley, "Journey to the Protected Land," *Circuit Cellar* 48–65, 1994–1995.

SOURCES

Intel Core 2 Duo Processor
Intel Corp. | www.intel.com

Visual C++ Integrated development environment (IDE)
Microsoft Corp. | www.microsoft.com