CMOS

"CMOS" is a tiny bit of very low power static memory that lives on the same chip as the Real-Time Clock. It is fairly convenient to actually think of the RTC as being "part" of CMOS. CMOS (and the Real-Time Clock) can only be accessed through IO Ports 0x70 and 0x71. The function of the CMOS memory is to store 50 (or 114) bytes of "Setup" information for the BIOS while the computer is turned off -- because there is a separate battery that keeps the Clock and the CMOS information active.

CMOS values are accessed a byte at a time, and each byte is individually addressable. Each CMOS address is traditionally called a "register". The first 14 CMOS registers access and control the Real-Time Clock. In fact, the only truly useful registers remaining in CMOS are the Real-Time Clock registers, and register 0x10. All other registers in CMOS are almost entirely obsolete (or are not standardized), and are therefore useless.

Contents [hide]

1 Non-Maskable Interrupts
2 CMOS Registers
  2.1 Accessing CMOS Registers
    2.1.1 Checksums
  2.2 Register 0x10
  2.3 Memory Size Registers
  2.4 Hard Disk Registers
3 The Real-Time Clock
  3.1 Getting Current Date and Time from RTC
  3.2 Century Register
    3.2.1 Century Register As Time and Date Sanity Check
  3.3 Weekday Register
  3.4 RTC Update In Progress
  3.5 Format of Bytes
  3.6 Interpreting RTC Values
4 Examples
  4.1 Reading from the CMOS
  4.2 Writing to the CMOS
  4.3 Reading All RTC Time and Date Registers
5 See Also

Non-Maskable Interrupts

For frugality in the olden days, many functions were merged together on chips where there was "room" -- even if they did not belong together. An example is putting the A20 address enable on the PS2 keyboard controller. In the same way, the "NMI disable" control was put together with the CMOS controller and the Real-Time Clock.

NMI is meant to communicate a "panic" status from the hardware to the CPU in a way that the CPU cannot ignore. It is typically used to signal memory errors. For more information about NMI, see the NMI article.

Whenever you send a byte to IO port 0x70, the high order bit tells the hardware whether to disable NMIs from reaching the CPU. If the bit is on, NMI is disabled (until the next time you send a byte to Port 0x70). The low order 7 bits of any byte sent to Port 0x70 are used to address CMOS registers.

CMOS Registers

The most updated CMOS register map, showing all of the conflicting register definitions between the various BIOSes, is in RBIL in the file called CMOS.LST.

Accessing CMOS Registers

Accessing CMOS is extremely simple, but you always need to take into account how you want to handle NMI. You "select" a CMOS register (for reading or writing) by sending the register number to IO Port 0x70. Since the 0x80 bit of Port 0x70 controls NMI, you always end up setting that, too. So your CMOS controller always needs to know whether your OS wants NMI to be enabled or not. Selecting a CMOS register is done as follows:

- outb (0x70, (NMI_disable_bit << 7) | (selected CMOS register number));

Once a register is selected, you either read the value of that register on Port 0x71 (with inb or an equivalent function), or you write a new value to that register -- also on Port 0x71 (with outb, for example):

- val_8bit = inb (0x71);

Note1: Reading or writing Port 0x71 seems to default the "selected register" back to 0xD. So you need to **reselect** the register every single time you want to access a CMOS register.

Note2: It is probably a good idea to have a reasonable delay after selecting a CMOS register on Port 0x70, before reading/writing the value on Port 0x71.

Checksums

The proper functioning of the BIOS during bootup depends on the values in CMOS. So the values are protected against random changes with checksums. It is very unwise ever to write a value into any of the CMOS registers (except for the RTC) -- because when you change a value you also have to go fix a BIOS-specific checksum in a different register -- or else the next boot will crash with an "invalid checksum" error. And since the checksum is located at a proprietary BIOS-specific register number, good luck finding it.

Register 0x10

This register contains the only CMOS value that an OS might ever find to be useful. It describes the "type" of each of the two floppy drives that may be attached to the system. The high nibble describes the "master" floppy drive on the primary bus, and the low nibble has an identical description for the "slave" floppy drive.

Value of each 4 bit nibble, and associated floppy drive type:

```
Value   Drive Type
 00h    no drive
 01h    360 KB 5.25 Drive
 02h    1.2 MB 5.25 Drive
 03h    720 KB 3.5 Drive
 04h    1.44 MB 3.5 Drive
 05h    2.88 MB 3.5 drive
```

Bits 0 to 3 = slave floppy type, bits 4 to 7 = master floppy type

Memory Size Registers

There are several CMOS registers that are standardized, and that seem to report useful information about the total memory on the system. However, each of them is lacking vital information that your OS will need. It is always better to use a BIOS function call to get information about memory than to use the information in CMOS. See Detecting Memory (x86).

(Register 0x16 (high byte) | Register 0x15 (low byte)) << 10 = 640K = size of low memory (without taking the EBDA into account.

(Register 0x18 (high byte) | Register 0x17 (low byte)) << 10 = total memory between 1M and 16M, or maybe 65M ... usually. But this number is extra untrustworthy when the system has more than 64M, it ignores "memory holes", it ignores memory mapped hardware, and it ignores memory reserved for important ACPI system tables.

(Register 0x31 (high byte) | Register 0x30 (low byte)) << 16 = total memory between 16M and 4G ... usually. But this number is untrustworthy when the system has more than 4G, it ignores "memory holes", it ignores memory mapped hardware, and it ignores memory reserved for important ACPI system tables.

Hard Disk Registers

There are many CMOS registers in various locations, used by various ancient BIOSes, to store a "hard disk type" or other hard disk information. Any such information is strictly for use on obsolete CHS-based disk drives. Better information can always be obtained via BIOS function INT13h AH=8, or by sending an ATA 'Identify' command to the disk in ATA PIO Mode.

The Real-Time Clock

The RTC keeps track of the date and time, even when the computer's power is off. The only other way that a computer used to be able to do this was to ask a human for the date/time on every bootup. Now, if the computer has an internet connection, an OS has another (arguably better) way

to get the same information.

The RTC also can generate clock ticks on IRQ8 (similarly to what the PIT does on IRQ0). The highest feasible clock frequency is 8KHz. Using the RTC clock this way may actually generate more stable clock pulses than the PIT can generate. It also frees up the PIT for timing events that really need near-microsecond accuracy. Additionally, the RTC can generate an IRQ8 at a particular time of day. See the RTC article for more detailed information about using RTC interrupts.

Getting Current Date and Time from RTC

To get each of the following date/time values from the RTC, you should first ensure that you won't be effected by an update (see below). Then select the associated "CMOS register" in the usual way, and read the value from Port 0x71.

```
Register  Contents
 0x00      Seconds
 0x02      Minutes
 0x04      Hours
 0x06      Weekday
 0x07      Day of Month
 0x08      Month
 0x09      Year
 0x32      Century (maybe)
 0x0A      Status Register A
 0x0B      Status Register B
```

Century Register

Originally the RTC didn't have a century register at all. In the 1990s (as the year 2000 got closer) hardware manufacturers started realising that this might become a problem; so they starting adding century registers to their RTC. Unfortunately, because there was no official standard to follow, different manufacturers used different registers.

This meant that software didn't know if there was a century register, and (if there is) which register it might be. To fix that problem the ACPI specification included a "RTC century register" field at offset 108 in its "Fixed ACPI Description Table". If this field contains zero then the RTC doesn't have a century register, and if the field is non-zero then it contains the number of the RTC register to use for century.

If there is no century register then software can guess. For example, a piece of software written in 1990 can use the (2 digit) year register to determine the most likely century - if the RTC year register is higher than or equal to 90 then the year is probably be "19YY" and if the RTC year register is less than 90 than the year must be "20YY". In this way, software can correctly determine the century for up to 99 years after the software is written.

Century Register As Time and Date Sanity Check

If the CMOS/RTC has a century register, your software was released 2014, and the CMOS/RTC says the century and year are 2008; then obviously the CMOS/RTC must be wrong.

Similarly, people tend to update their OS occasionally. If the CMOS/RTC has a century register, your software was released 2014, and the CMOS/RTC says the century and year are 2154; then it's unlikely that the OS hasn't been updated for 140 years, and far more likely that the CMOS/RTC is wrong.

Essentially; the method (described above) for guessing the century when there is no century register is much more reliable than the CMOS/RTC century register (if it exists). This means that the century register (if/when present) can be used in reverse, as a way to check if the CMOS/RTC time and date are sane (or if the CMOS/RTC has a flat battery or something).

Basically, you'd guess the century (based on the software's release date and RTC's year), then check if the CMOS/RTC century is the same as your guess and if it's not then assume all CMOS/RTC time and date fields are invalid.

Weekday Register

The RTC chip is able to keep track of the current day of the week. All it does is increment its "Weekday" register at midnight and reset it to zero if it reaches 7. Unfortunately there is no guarantee that this register was ever set correctly by anything (including when the user changes the time and date using the BIOS configuration screen). It is entirely unreliable and should not be used.

The correct way to determine the current day of the week is to calculate it from the date (see the article on Wikipedia for details of this calculation).

RTC Update In Progress

When the chip updates the time and date (once per second) it increases "seconds" and checks if it rolled over. If "seconds" did roll over it increases "minutes" and checks if that rolled over. This can continue through all the time and date registers (e.g. all the way up to "if year rolled over, increase century"). However, the RTC circuitry is typically relatively slow. This means that it's entirely possible to read the time and date while an update is in progress and get dodgy/inconsistent values (for example, at 9:00 o'clock you might read 8:59, or 8:60, or 8:00, or 9:00).

To help guard against this problem the RTC has an "Update in progress" flag (bit 7 of Status Register A). To read the time and date properly you have to wait until the "Update in progress" flag goes from "set" to "clear". This is not the same as checking that the "Update in progress" flag is clear.

For example, if code does "*while(update_in_progress_flag != clear)*" and then starts reading all the time and date registers, then the update could begin immediately after the "Update in progress" flag was checked and the code could still read dodgy/inconsistent values. To avoid this, code should wait until the flag becomes set and then wait until the flag becomes clear. That way there's almost 1 second of time to read all of the registers correctly.

Unfortunately, doing it correctly (waiting until the "Update in progress" flag becomes set and then waiting until it becomes clear) is very slow - it may take an entire second of waiting/polling before you can read the registers. There are 2 alternatives.

The first alternative is to rely on the "update interrupt". When the RTC finishes an update it generates an "update interrupt" (if it's enabled), and the IRQ handler can safely read the time and date registers without worrying about the update at all (and without checking the "Update in progress" flag); as long as the IRQ handler doesn't take almost a full second to do it. In this case you're not wasting up to 1 second of CPU time waiting/polling, but it may still take a full second before the time and date has been read. Despite this it can be a useful technique during OS boot - e.g. setup the "update interrupt" and its IRQ handler as early as you can and then do other things (e.g. loading files from disk), in the hope

that the IRQ occurs before you need the time and date.

The second alternative is to be prepared for dodgy/inconsistent values and cope with them if they occur. To do this, make sure the "Update in progress" flag is clear (e.g. "*while(update_in_progress_flag != clear)*") then read all the time and date registers; then make sure the "Update in progress" flag is clear again (e.g. "*while(update_in_progress_flag != clear)*") and read all the time and date registers again. If the values that were read the first time are the same as the value that were read the second time then the values must be correct. If any of the values are different you need to do it again, and keep doing it again until the newest values are the same as the previous values.

Format of Bytes

There are 4 formats possible for the date/time RTC bytes:

- Binary or BCD Mode
- Hours in 12 hour format or 24 hour format

The format is controlled by Status Register B. On some CMOS/RTC chips, the format bits in Status Reg B **cannot be changed**. So your code needs to be able to handle all four possibilities, and it should not try to modify Status Register B's settings. So you always need to read Status Register B first, to find out what format your date/time bytes will arrive in.

- Status Register B, Bit 1 (value = 2): Enables 24 hour format if set
- Status Register B, Bit 2 (value = 4): Enables Binary mode if set

Binary mode is exactly what you would expect the value to be. If the time is 1:59:48 AM, then the value of hours would be 1, minutes would be 59 = 0x3b, and seconds would be 48 = 0x30.

In BCD mode, each of the two hex nibbles of the byte is modified to "display" a **decimal** number. So 1:59:48 has hours = 1, minutes = 0x59 = 89, seconds = 0x48 = 72. To convert BCD back into a "good" binary value, use: binary = ((bcd / 16) * 10) + (bcd & 0xf) *[Optimised: binary = ((bcd & 0xF0) >> 1) + ((bcd & 0xF0) >> 3) + (bcd & 0xf)]*.

24 hour time is exactly what you would expect. Hour 0 is midnight to 1am, hour 23 is 11pm.

12 hour time is annoying to convert back to 24 hour time. If the hour is pm, then the 0x80 bit is set on the hour byte. So you need to mask that off. (This is true for **both** binary and BCD modes.) Then, midnight is 12, 1am is 1, etc. Note that carefully: midnight is not 0 -- it is 12 -- this needs to be handled as a special case in the calculation from 12 hour format to 24 hour format (by setting 12 back to 0)!

For the weekday format: Sunday = 1, Saturday = 7.

Interpreting RTC Values

On the surface, these values from the RTC seem extremely obvious. The main difficulty comes in deciding which timezone the values represent. The two possibilities are usually UTC, or the system's timezone, including Daylight Savings. See the Time And Date article for a much more complete discussion of how to handle this issue.

Examples

Reading from the CMOS

```
ReadFromCMOS (unsigned char array [])
{
   unsigned char tvalue, index;

   for(index = 0; index < 128; index++)
   {
      _asm
      {
         cli             /* Disable interrupts*/
         mov al, index   /* Move index address*/
         /* since the 0x80 bit of al is not set, NMI is active */
         out 0x70,al     /* Copy address to CMOS register*/
         /* some kind of real delay here is probably best */
         in al,0x71      /* Fetch 1 byte to al*/
         sti             /* Enable interrupts*/
         mov tvalue,al
       }

       array[index] = tvalue;
   }
}
```

Writing to the CMOS

```
WriteTOCMOS(unsigned char array[])
{
   unsigned char index;

   for(index = 0; index < 128; index++)
   {
      unsigned char tvalue = array[index];
      _asm
      {
         cli             /* Clear interrupts*/
         mov al,index    /* move index address*/
         out 0x70,al     /* copy address to CMOS register*/
         /* some kind of real delay here is probably best */
         mov al,tvalue   /* move value to al*/
         out 0x71,al     /* write 1 byte to CMOS*/
         sti             /* Enable interrupts*/
      }
   }
```

```
}
```

Reading All RTC Time and Date Registers

```
#define CURRENT_YEAR        2014                            // Change this each year!

int century_register = 0x00;                                // Set by ACPI table parsing code if possible

unsigned char second;
unsigned char minute;
unsigned char hour;
unsigned char day;
unsigned char month;
unsigned int year;

void out_byte(int port, int value);
int in_byte(int port);

enum {
      cmos_address = 0x70,
      cmos_data    = 0x71
};

int get_update_in_progress_flag() {
      out_byte(cmos_address, 0x0A);
      return (in_byte(cmos_data) & 0x80);
}

unsigned char get_RTC_register(int reg) {
      out_byte(cmos_address, reg);
      return in_byte(cmos_data);
}

void read_rtc() {
      unsigned char century;
      unsigned char last_second;
      unsigned char last_minute;
      unsigned char last_hour;
      unsigned char last_day;
      unsigned char last_month;
      unsigned char last_year;
      unsigned char last_century;
      unsigned char registerB;

      // Note: This uses the "read registers until you get the same values twice in a row" technique
```

```
      //       to avoid getting dodgy/inconsistent values due to RTC updates

      while (get_update_in_progress_flag());                // Make sure an update isn't in progress
      second = get_RTC_register(0x00);
      minute = get_RTC_register(0x02);
      hour = get_RTC_register(0x04);
      day = get_RTC_register(0x07);
      month = get_RTC_register(0x08);
      year = get_RTC_register(0x09);
      if(century_register != 0) {
            century = get_RTC_register(century_register);
      }

      do {
            last_second = second;
            last_minute = minute;
            last_hour = hour;
            last_day = day;
            last_month = month;
            last_year = year;
            last_century = century;

            while (get_update_in_progress_flag());           // Make sure an update isn't in progress
            second = get_RTC_register(0x00);
            minute = get_RTC_register(0x02);
            hour = get_RTC_register(0x04);
            day = get_RTC_register(0x07);
            month = get_RTC_register(0x08);
            year = get_RTC_register(0x09);
            if(century_register != 0) {
                  century = get_RTC_register(century_register);
            }
      } while( (last_second != second) || (last_minute != minute) || (last_hour != hour) ||
               (last_day != day) || (last_month != month) || (last_year != year) ||
               (last_century != century) );

      registerB = get_RTC_register(0x0B);

      // Convert BCD to binary values if necessary

      if (!(registerB & 0x04)) {
            second = (second & 0x0F) + ((second / 16) * 10);
            minute = (minute & 0x0F) + ((minute / 16) * 10);
            hour = ( (hour & 0x0F) + (((hour & 0x70) / 16) * 10) ) | (hour & 0x80);
            day = (day & 0x0F) + ((day / 16) * 10);
            month = (month & 0x0F) + ((month / 16) * 10);
            year = (year & 0x0F) + ((year / 16) * 10);
            if(century_register != 0) {
```

```
                century = (century & 0x0F) + ((century / 16) * 10);
            }
        }

        // Convert 12 hour clock to 24 hour clock if necessary

        if (!(registerB & 0x02) && (hour & 0x80)) {
            hour = ((hour & 0x7F) + 12) % 24;
        }

        // Calculate the full (4-digit) year

        if(century_register != 0) {
            year += century * 100;
        } else {
            year += (CURRENT_YEAR / 100) * 100;
            if(year < CURRENT_YEAR) year += 100;
        }
}
```

See Also

- Old CMOS Map
- Better CMOS Map

Categories: X86 | Time