Detecting Memory (x86)

From OSDev Wiki

One of the most vital pieces of information that an OS needs in order to initialize itself is a map of the available RAM on a machine. Fundamentally, the best way (and really the **only** way) an OS can get that information is by using the BIOS. There may be rare machines where you have no other choice but to try to detect memory yourself -- however, doing so is unwise in any other situation.

It is perfectly reasonable to say to yourself, "How does the BIOS detect RAM? I'll just do it that way." Unfortunately, the answer is disappointing:
Most BIOSes can't use any RAM until they detect the type of RAM installed, then detect the size of each memory module, then configure the chipset to use the detected RAM. All of this depends on chipset specific methods, and is usually documented in the datasheets for the memory controller (northbridge). The RAM is unusable for running programs during this process. The BIOS initially is running from ROM, so it can play the necessary games with the RAM chips. But it is completely impossible to do this from inside any other program.

It is also reasonable to wish to reclaim the memory from 0xA0000 to 0xFFFFF and make your RAM contiguous. Again the answer is disappointing:
Forget about it. It is likely that some of it is being used regularly by SMM or ACPI. Some of it you will probably need again, even after the machine is booted. Reclaiming pieces of it would require a significant amount of motherboard- or chipset-specific control. It *is* possible to write a "chipset driver" that may allow you to reclaim a little of it. However, it is almost certainly impossible to reclaim it all. The minuscule results will not be worth the effort, in general.

It might be important to note that all PCs require a memory hole just below 4GB for additional memory mapped hardware (including the actual BIOS ROM). So, for a machine with >3G of RAM, the motherboard / chipset / BIOS may map some of that RAM (that would overlap mapped hardware) above 4G -- use PAE or Long Mode to access it.

See Memory Map (x86) for a generic layout of memory.

Contents

Detecting Low Memory

"Low memory" is the available RAM below 1Mb, and usually below 640Kb. There are two BIOS functions to get the size of it.

INT 0x12: The INT 0x12 call will return AX = total number of Kb (or an error). The AX value measures from 0, up to the bottom of the EBDA (of course, you probably shouldn't use the first 0x500 bytes of the space either -- i.e. the IVT or BDA).

Usage:

```
    ; Nullify the A-register.
    xor ax, ax

    ; Switch to the BIOS (= request low memory size).
    int 0x12

    ; The carry flag is set, it failed.
    jc .Error

    ; Test the A-register with itself.
    test ax, ax

    ; The zero flag is set, it failed.
    jz .Error

    ; AX = amount of continuous memory in kB starting from 0.
```

Alternately, you can just use INT 0x15, EAX = 0xE820 (see below).

Detecting Upper Memory

BIOS Function: INT 0x15, EAX = 0xE820

By far the best way to detect the memory of a PC is by using the INT 0x15, EAX = 0xE820 command. This function is available on all PCs built since 2002, and on most existing PCs before then. It is the only BIOS function that can detect memory areas above 4G. It is meant to be the ultimate memory detection BIOS function.

In reality, this function returns an unsorted list that may contain unused entries and (in rare/dodgy cases) may return overlapping areas. Each list entry is stored in memory at ES:DI, and DI is **not** incremented for you. The format of an entry is 2 uint64_t's and a uint32_t in the 20 byte version, plus one additional uint32_t in the 24 byte ACPI 3.0 version (but nobody has ever seen a 24 byte one). It is probably best to always store the list entries as 24 byte quantities -- to preserve uint64_t alignments, if nothing else. (Make sure to set that last uint64_t to 1 before each call, to make your map compatible with ACPI).

- First uint64_t = Base address
- Second uint64_t = Length of "region" (if this value is 0, ignore the entry)
- Next uint32_t = Region "type"
  - Type 1: Usable (normal) RAM
  - Type 2: Reserved - unusable
  - Type 3: ACPI reclaimable memory
  - Type 4: ACPI NVS memory
  - Type 5: Area containing bad memory
- Next uint32_t = ACPI 3.0 Extended Attributes bitfield (if 24 bytes are returned, instead of 20)
  - Bit 0 of the Extended Attributes indicates if the entire entry should be ignored (if the bit is clear). This is going to be a huge compatibility problem because most current OSs won't read this bit and won't ignore the entry.
  - Bit 1 of the Extended Attributes indicates if the entry is non-volatile (if the bit is set) or not. The standard states that "Memory reported as non-volatile may require characterization to determine its suitability for use as conventional RAM."
  - The remaining 30 bits of the Extended Attributes are currently undefined.

Basic Usage:
For the first call to the function, point ES:DI at the destination buffer for the list. Clear EBX. Set EDX to the magic number 0x534D4150. Set EAX to 0xE820 (note that the upper 16-bits of EAX should be set to 0). Set ECX to 24. Do an INT 0x15.

If the first call to the function is successful, EAX will be set to 0x534D4150, and the Carry flag will be clear. EBX will be set to some non-zero value, which must be preserved for the next call to the function. CL will contain the number of bytes actually stored at ES:DI (probably 20).

For the subsequent calls to the function: increment DI by your list entry size, reset EAX to 0xE820, and ECX to 24. When you reach the end of the list, EBX may reset to 0. If you call the function again with EBX = 0, the list will start over. If EBX does not reset to 0, the function will return with Carry set when you try to access the entry after the last valid entry.

(See the code examples below for a detailed ASM example, implementing the algorithm.)

Notes:

- After getting the list, it may be desirable to: sort the list, combine adjacent ranges of the same type, change any overlapping areas to the most restrictive type, and change any unrecognised "type" values to type 2.
- Type 3 "ACPI reclaimable" memory regions may be used like (and combined with) normal "available RAM" areas as long as you're finished using the ACPI tables that are stored there (i.e. it can be "reclaimed").
- Types 2, 4, 5 (reserved, ACPI non-volatile, bad) mark areas that should be avoided when you are

allocating physical memory.
- Treat unlisted regions as Type 2 -- reserved.
- Your code must be able to handle areas that don't start or end on any sort of "page boundary".

Typical Output by a call to INT 15h, EAX=E820 in Bochs:

```
 Base Address       | Length             | Type
 0x0000000000000000 | 0x000000000009FC00 | Free Memory (1)
 0x000000000009FC00 | 0x0000000000000400 | Reserved Memory (2)
 0x00000000000E8000 | 0x0000000000018000 | Reserved Memory (2)
 0x0000000000100000 | 0x0000000001F00000 | Free Memory (1)
 0x00000000FFFC0000 | 0x0000000000040000 | Reserved Memory (2)
```

Other Methods

PnP

It is possible to get a fairly good memory map using Plug 'n Play (PnP) calls. {Need description and code.}

SMBios

SMBIOS is designed to allow "administrators" to assess hardware upgrade options or maintain a catalogue of what hardware a company current has in use (ie. it provides information for use by humans, rather than for use by software). It may not give reliable results on many computers -- see: [1] (http://www.pcpitstop.com/faq/smbios.asp) .

SMBIOS will try to tell you the number of memory sticks installed and their size in MB. SMBIOS can be called from protected mode. However, some manufacturers don't make their systems fully compliant. e.g. HP Itanium goes by the DIG64 specification so their SMBIOS doesn't return all the required device types.

Detecting memory with these functions completely ignores the concept of memory holes / memory mapped devices / reserved areas.

BIOS Function: INT 0x15, AX = 0xE881

Note: This function is identical to the E801 function, except that is uses extended registers (EAX/EBX/ECX/EDX). It only reports contiguous (usable) RAM. It cannot detect any more memory than the E801 function, but there is a chance that this function may succeed on BIOSes where E801 fails.

BIOS Function: INT 0x15, AX = 0xE801

This function has been around since about 1994, so all systems from after then up to now should have this function. It is built to handle the 15M memory hole, but stops at the next hole / memory mapped device / reserved area above that. That is, it is only designed to handle contiguous memory above 16M.

Typical Output:
AX = CX = extended memory between 1M and 16M, in K (max 3C00h = 15MB)

BX = DX = extended memory above 16M, in 64K blocks

There are some BIOSes that always return with AX = BX = 0. Use the CX/DX pair in that case. Some other BIOSes will return CX = DX = 0. Linux initializes the CX/DX pair to 0 before the INT opcode, and then uses CX/DX, unless they are still 0 (when it will use AX/BX). In any case, it is best to do a sanity check on the values in the registers that you use before you trust the results. (GRUB just trusts AX/BX -- this is not good.)

Linux Usage:

```
        XOR CX, CX
        XOR DX, DX
        MOV AX, 0xE801
        INT 0x15                ; request upper memory size
        JC SHORT .ERR
        CMP AH, 0x86            ; unsupported function
        JE SHORT .ERR
        CMP AH, 0x80            ; invalid command
        JE SHORT .ERR
        JCXZ .USEAX             ; was the CX result invalid?

        MOV AX, CX
        MOV BX, DX
.USEAX:
        ; AX = number of contiguous Kb, 1M to 16M
        ; BX = contiguous 64Kb pages above 16M
```

BIOS Function: INT 0x15, AX = 0xDA88

This function returns the number of contiguous KiB of usable RAM starting at 0x00100000 in Cl:BX in KiB. This is very similar to "INT 0x15, AX=0x8A" - if this function says there's 14 MiB of RAM at 0x00100000 then you can't assume there isn't more RAM at 0x01000000, so you'd probe for any extra memory starting at 0x01000000.

If this function isn't supported it'll return "carry = set".

BIOS Function: INT 0x15, AH = 0x88

Note: This function may limit itself to reporting 15M (for legacy reasons) even if BIOS detects more memory than that. It may also report up to 64M. It only reports contiguous (usable) RAM.

Usage:

```
        MOV AH, 0x88
```

```
        INT 0x15                ; request upper memory size
        JC SHORT .ERR
        TEST AX, AX             ; size = 0 is an error
        JE SHORT .ERR
        CMP AH, 0x86            ; unsupported function
        JE SHORT .ERR
        CMP AH, 0x80            ; invalid command
        JE SHORT .ERR
        ; AX = number of contiguous Kb above 1M
```

BIOS Function: INT 0x15, AH = 0x8A

This function returns the extended memory size in DX:AX in KiB, or to be more specific, it returns the number of contiguous KiB of usable RAM starting at 0x00100000. This is also where it starts getting tricky...

If the ISA memory hole is present (which is a 1 MiB hole from 0x00F00000 to 0x00FFFFFF used by ISA devices for memory mapped I/O - e.g. an ISA video card's linear frame buffer) then this function might not report all usable RAM. For example, it might report RAM from 0x00100000 to 0x00F00000 and wouldn't be able to report any RAM above 0x01000000 (if present).

Basically, if this function says there's 14 MiB of RAM at 0x00100000 then you can't assume there isn't more RAM at 0x01000000. In this case, it's likely that none of the other methods will be able to tell you more, so you'd probe for any extra memory starting at 0x01000000.

If this function isn't supported it'll return "carry = set".

BIOS Function: INT 0x15, AH = 0xC7

Although not widely supported, this function defined by IBM, provides a nice memory map (although not as nice as 0xE820). DS:SI points to the following memory map table:

```
Size   Offset  Description

2      00h     Number of significant bytes of returned data (excluding this uint16_t)
4      02h     Amount of local memory between 1-16MB, in 1KB blocks
4      06h     Amount of local memory between 16MB and 4GB, in 1KB blocks
4      0Ah     Amount of system memory between 1-16MB, in 1KB blocks
4      0Eh     Amount of system memory between 16MB and 4GB, in 1KB blocks
4      12h     Amount of cacheable memory between 1-16MB, in 1KB blocks
4      16h     Amount of cacheable memory between 16MB and 4GB, in 1KB blocks
4      1Ah     Number of 1KB blocks before start of nonsystem memory between 1-16MB
4      1Eh     Number of 1KB blocks before start of nonsystem memory between 16MB and 4GB
2      22h     Starting segment of largest block of free memory in 0C000h and 0D000h segments
2      24h     Amount of free memory in the block defined by offset 22h
```

The minimum number which can be returned by the first uint16_t is 66 bytes. Here's how the memory types are defined:

- Local Memory on the system board or memory that is not accessible from the channel. It can be system or nonsystem memory.
- Channel Memory on adapters. It can be system or nonsystem memory.

- System Memory that is managed and allocated by the primary operating system. This memory is cached if the cache is enabled.
- Nonsystem Memory that is not managed or allocated by the primary operating system. This memory includes memory-mapped I/O devices; memory that is on an adapter and can be directly modified by the adapter; and memory that can be relocated within its address space, such as bank-switched and expanded-memory-specifications (EMS) memory. This memory is not cached.

CMOS

The CMOS memory size information may ignore the standard memory hole at 15M. If you use the CMOS size, you may want to simply assume that this memory hole exists. Of course, it also has no information about any other reserved regions.

Usage:

```
    unsigned short total;
    unsigned char lowmem, highmem;

    outportb(0x70, 0x30);
    lowmem = inportb(0x71);
    outportb(0x70, 0x31);
    highmem = inportb(0x71);

    total = lowmem | highmem << 8;
    return total;
```

E820h

There are a few other BIOS functions that claim to give you memory information. However, they are so widely unsupported that it is impossible to even find machines to test the code on. **All** current machines support E820 (above). If some user should happen to dig up such a dinosaur of a machine that its BIOS does not support any standard memory detection function -- they will not complain that your modern OS fails to support that machine. Just give an error message.

Manual Probing

WE DISCOURAGE YOU FROM DIRECTLY PROBING MEMORY

Use BIOS to get a memory map, or use GRUB (which calls BIOS for you).

When perfectly implemented, directly probing memory may allow you to detect extended memory even on systems where the BIOS fails to provide the appropriate support (or without even worrying about whether your BIOS can do it or not). The algorithm may or may not take into account potential holes in system memory or previously detected memory mapped devices, such as frame buffering SVGA cards, etc.

However, the BIOS knows things you ignore about your motherboard and PCI devices. Probing memory-mapped PCI devices may have *unpredictable results* and may theoretically *damage your system*, so once again we **discourage** its use.

Note: You will never get an error from trying to read/write memory that does not exist -- this is important to understand: you will not get valid results, but you won't get an error, either.

Theoretical obstacles to probing

- There can be a memory mapped device from 15 MB to 16 MB (typically "VESA local bus" video cards, or older ISA cards that aren't limited to just video).
- There can also be an (extremely rare) "memory hole" at 0x00080000 for some sort of compatibility with ancient cards.
- On modern systems there can also be faulty RAM that INT 0x15, eax=0xE820 says not to use, that can be anywhere (except in the first 1 MB, probably).
- There can also be large arbitrary memory holes. (e.g. a NUMA system with RAM up to 0x1FFFFFFF, a hole from 0x20000000 to 0x3FFFFFFF, then more RAM from 0x40000000 to 0x5FFFFFFF.)
- It's possible for physical addresses to be truncated in various ways. For example, older chipsets often have less address lines than supported by the processor and typically on motherboard using such chipsets, the extra address lines are simply not connected. For example, Intel desktop chipsets prior to the 955X only have 32 address lines, even though they are usually used with a processor that supports at least PAE. The extra address lines (A32-A35) are simply not connected on the motherboard, and if the processor attempts to access memory using physical addresses exceeding 32-bit, the physical address is truncated by virtue of the extra address lines not being connected on the motherboard.
- There are memory mapped devices (PCI video cards, HPET, PCI express configuration space, APICs, etc) with addresses that must be avoided.
- There are also (typically older) motherboards where you can write a value to "nothing" and read the same value back due to bus capacitance; there are motherboards where you write a value to cache and read the same value back from cache even though no RAM is at that address.
- There are (older, mostly 80386) motherboards that remap the RAM underneath the option ROMs and BIOS to the end of RAM. (e.g. with 4 MB of RAM installed you get RAM from 0x00000000 to 0x000A0000 and more RAM from 0x00100000 to 0x00460000, which causes problems if you test each MB of RAM because you get the wrong answer -- either under-counting RAM up to 0x00400000, or over-counting RAM up to 0x00500000).
- There can be important data structures left in RAM by the BIOS (e.g. the ACPI tables) that you'd trash.
- If you write the code properly (ie. to avoid as many of the problems as you can), then it is *insanely* slow.
- Lastly, testing for RAM (if it actually works) will only tell you where RAM is - it doesn't give you a complete map of the physical address space. You won't know where you can safely put memory mapped PCI devices because you won't know which areas are reserved for chipset things (e.g. SMM, ROMs), etc.

In contrast to this, using the BIOS functions isn't too hard, is much more reliable, gives complete information, and is extremely fast in comparison.

Memory Map Via GRUB

GRUB, or any bootloader implementing The Multiboot Specification (http://www.gnu.org/software/grub/manual/multiboot/multiboot.html) provides a convenient way of detecting the amount of RAM your machine has. Rather than re-invent the wheel, you can ride on the hard work that others have done by utilizing the multiboot_info structure. When GRUB runs, it loads this structure into memory and leaves the address of this structure in the EBX register. You may also view this structure at the GRUB command-line with the GRUB command `displaymem`.

The methods it uses are:

- Try BIOS Int 0x15, eax = 0xE820
- If that didn't work, try BIOS Int 0x15, ax = 0xE801 and BIOS Int 0x12
- If that didn't work, try BIOS Int 0x15, ah = 0x88 and BIOS Int 0x12

However, it does not take into account any bugs that are known to effect some BIOSs (see entries in RBIL). It does not check "E801" and/or "88" returning with carry set.

To utilize the information that GRUB passes to you, first include the file multiboot.h (http://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot.h.html) in your kernel's main file. Then, make sure that when you load your _main function from your assembly loader, you push EBX onto the stack. The Bare bones tutorials have already done this for you.

The key for memory detection lies in the multiboot_info struct. To get access to it, you've already pushed it onto the stack...define your start function as such:

```
_main (multiboot_info_t* mbd, unsigned int magic) {...}
```

To determine the contiguous memory size, you may simply check `mbd->flags` to verify bit 0 is set, and then you can safely refer to `mbd->mem_lower` for conventional memory (e.g. physical addresses ranging between 0 and 640KB) and `mbd->mem_upper` for high memory (e.g. from 1MB). Both are given in kibibytes, i.e. blocks of 1024 bytes each.

To get a complete memory map, check bit 6 of `mbd->flags` and use `mbd->mmap_addr` to access the BIOS-provided memory map. Quoting specifications (http://www.gnu.org/software/grub/manual/multiboot/html_node/Boot-information-format.html#Boot%20information%20format) ,

> If bit 6 in the flags uint16_t is set, then the mmap_* fields are valid, and indicate the address and length of a buffer containing a memory map of the machine provided by the BIOS. mmap_addr is the address, and mmap_length is the total size of the buffer. The buffer consists of one or more of the following size/structure pairs (size is really used for skipping to the next pair):

-4	size
0	base_addr_low
4	base_addr_high
8	length_low
12	length_high
16	type

- "size" is the size of the associated structure in bytes, which can be greater than the minimum of 20 bytes. base_addr_low is the lower 32 bits of the starting address, and base_addr_high is the upper 32 bits, for a total of a 64-bit starting address. length_low is the lower 32 bits of the size of the

memory region in bytes, and length_high is the upper 32 bits, for a total of a 64-bit length. type is the variety of address range represented, where a value of 1 indicates available RAM, and all other values currently indicated a reserved area.

- GRUB simply uses INT 15h, EAX=E820 to get the detailed memory map, and does not verify the "sanity" of the map. It also will not sort the entries, retrieve any available ACPI 3.0 extended uint32_t (with the "ignore this entry" bit), or clean up the table in any other way.
- One of the problems you have to deal with is that grub can theoretically place its multiboot information, and all the tables it references (elf sections, mmap and modules) anywhere in memory, according to the multiboot specification. In reality, in current grub legacy, they are allocated as parts of the grub program itself, below 1MB, but that is not guaranteed to remain the same. For that reason, you should try to protect these tables before you start using a certain bit of memory. (You might scan the tables to make sure their addresses are all below 1M.)
- Another problem is that the "type" field is defined as "1 = usable RAM" and "anything else is unusable". Despite what the multi-boot specification says, lots of people assume that the type field is taken directly from INT 15h, EAX=E820 (and in older versions of GRUB it is). However GRUB2 supports booting from UEFI/EFI (and other sources) and code that assumes the type field is taken directly from INT 15h, EAX=E820 will become broken. This means that (until a new multi-boot specification is released) you shouldn't make assumptions about the type, and can't do things like reclaiming the "ACPI reclaimable" areas or supporting S4/hibernate states (as an OS needs to save/restore areas marked as "ACPI NVS" to do that). Fortunately a new version of the multi-boot specification should be released soon which hopefully fixes this problem (but unfortunately, you won't be able to tell if your OS was started from "GRUB-legacy" or "GRUB2", unless it adopts the new multi-boot header and becomes incompatible with GRUB-legacy).

Memory Detection in Emulators

When you tell an emulator how much memory you want emulated, the concept is a little "fuzzy" because of the emulated missing bits of RAM below 1M. If you tell an emulator to emulate 32M, does that mean your address space definitely goes from 0 to 32M -1, with missing bits? Not necessarily. The emulator might assume that you mean 32M of *contiguous* memory above 1M, so it might end at 33M -1. Or it might assume that you mean 32M of total *usable* RAM, going from 0 to 32M + 384K -1. So don't be surprised if you see a "detected memory size" that does not exactly match your expectations.

Code Examples

Getting a GRUB Memory Map

Declare the appropriate structure, get the pointer to the first instance, grab whatever address and length information you want, and finally skip to the next memory map instance by adding size+4 to the pointer, tacking on the 4 to account for GRUB treating base_addr_low as offset 0 in the structure. You must also use mmap_length to make sure you don't overshoot the entire buffer.

```
typedef struct multiboot_memory_map {
        unsigned int size;
        unsigned int base_addr_low,base_addr_high;
// You can also use: unsigned long long int base_addr; if supported.
        unsigned int length_low,length_high;
// You can also use: unsigned long long int length; if supported.
        unsigned int type;
```

```
} multiboot_memory_map_t;

int main(multiboot_info* mbt, unsigned int magic) {
        ...
        multiboot_memory_map_t* mmap = mbt->mmap_addr;
        while(mmap < mbt->mmap_addr + mbt->mmap_length) {
                ...
                mmap = (multiboot_memory_map_t*) ( (unsigned int)mmap + m
        }
        ...
}
```

Getting an E820 Memory Map

```
; use the INT 0x15, eax= 0xE820 BIOS function to get a memory map
; inputs: es:di -> destination buffer for 24 byte entries
; outputs: bp = entry count, trashes all registers except esi
do_e820:
        xor ebx, ebx            ; ebx must be 0 to start
        xor bp, bp              ; keep an entry count in bp
        mov edx, 0x0534D4150    ; Place "SMAP" into edx
        mov eax, 0xe820
        mov [es:di + 20], dword 1       ; force a valid ACPI 3.X entry
        mov ecx, 24             ; ask for 24 bytes
        int 0x15
        jc short .failed        ; carry set on first call means "unsuppor
        mov edx, 0x0534D4150    ; Some BIOSes apparently trash this regis
        cmp eax, edx            ; on success, eax must have been reset to
        jne short .failed
        test ebx, ebx           ; ebx = 0 implies list is only 1 entry lo
        je short .failed
        jmp short .jmpin
.e820lp:
        mov eax, 0xe820         ; eax, ecx get trashed on every int 0x15
        mov [es:di + 20], dword 1       ; force a valid ACPI 3.X entry
        mov ecx, 24             ; ask for 24 bytes again
        int 0x15
        jc short .e820f         ; carry set means "end of list already re
        mov edx, 0x0534D4150    ; repair potentially trashed register
.jmpin:
        jcxz .skipent           ; skip any 0 length entries
        cmp cl, 20              ; got a 24 byte ACPI 3.X response?
        jbe short .notext
        test byte [es:di + 20], 1       ; if so: is the "ignore this data
        je short .skipent
.notext:
```

```
        mov ecx, [es:di + 8]    ; get lower uint32_t of memory region ler
        or ecx, [es:di + 12]    ; "or" it with upper uint32_t to test for
        jz .skipent             ; if length uint64_t is 0, skip entry
        inc bp                  ; got a good entry: ++count, move to next
        add di, 24
.skipent:
        test ebx, ebx           ; if ebx resets to 0, list is complete
        jne short .e820lp
.e820f:
        mov [mmap_ent], bp      ; store the entry count
        clc                     ; there is "jc" on end of list to this pc
        ret
.failed:
        stc                     ; "function unsupported" error exit
        ret
```

Sample in C (assuming we are in a bootloader environment, real mode, DS and CS = 0000):

```
// running in real mode may require:
__asm__(".code16gcc\n");

// SMAP entry structure
#include <stdint.h>
typedef struct SMAP_entry {

        uint32_t BaseL; // base address uint64_t
        uint32_t BaseH;
        uint32_t LengthL; // length uint64_t
        uint32_t LengthH;
        uint32_t Type; // entry Type
        uint32_t ACPI; // extended

}__attribute__((packed)) SMAP_entry_t;

// load memory map to buffer - note: regparm(3) avoids stack issues with
int __attribute__((noinline)) __attribute__((regparm(3))) detectMemory(SM
{
        uint32_t contID = 0;
        int entries = 0, signature, bytes;
        do
        {
                __asm__ __volatile__ ("int  $0x15"
                                : "=a"(signature), "=c"(bytes), "=b"(cont
                                : "a"(0xE820), "b"(contID), "c"(24), "d"(
                if (signature != 0x534D4150)
                        return -1; // error
                if (bytes > 20 && (buffer->ACPI & 0x0001) == 0)
                {
                        // ignore this entry
```

```
                }
                else {
                        buffer++;
                        entries++;
                }
        }
        while (contID != 0 && entries < maxentries);
        return entries;
}

// in your main routine - memory map is stored in 0000:1000 - 0000:2FFF f
[...] {
        [...]
        SMAP_entry_t* smap = (SMAP_entry_t*) 0x1000;
        const int smap_size = 0x2000;

        int entry_count = detectMemory(smap, smap_size / sizeof(SMAP_entr

        if (entry_count == -1) {
                // error - halt system and/or show error message
                [...]
        } else {
                // process memory map
                [...]
        }
}
```

Manual Probing in C

Notes:

- the interrupt disable and the cache invalidation keep memory consistent.
- the assembly language manual probing code that follows this example is better

```
 /*
  * void count_memory (void)
  *
  * probes memory above 1mb
  *
  * last mod : 05sep98 - stuart george
  *            08dec98 - ""     ""
  *            21feb99 - removed dummy calls
  *
  */
void count_memory(void)
{
        register ULONG *mem;
        ULONG   mem_count, a;
        USHORT  memkb;
```

```
        UCHAR   irq1, irq2;
        ULONG   cr0;

        /* save IRQ's */
        irq1=inb(0x21);
        irq2=inb(0xA1);

        /* kill all irq's */
        outb(0x21, 0xFF);
        outb(0xA1, 0xFF);

        mem_count=0;
        memkb=0;

        // store a copy of CR0
        __asm__ __volatile("movl %%cr0, %%eax":"=a"(cr0))::"eax");

        // invalidate the cache
        // write-back and invalidate the cache
        __asm__ __volatile__ ("wbinvd");

        // plug cr0 with just PE/CD/NW
        // cache disable(486+), no-writeback(486+), 32bit mode(386+)
        __asm__ __volatile__("movl %%eax, %%cr0", ::
                "a" (cr0 | 0x00000001 | 0x40000000 | 0x20000000) : "eax")

        do {
                memkb++;
                mem_count += 1024*1024;
                mem= (ULONG*) mem_count;

                a= *mem;
                *mem= 0x55AA55AA;

          // the empty asm calls tell gcc not to rely on what's in its re
          // as saved variables (this avoids GCC optimisations)
                asm(""::::"memory");
                if (*mem!=0x55AA55AA) mem_count=0;
                else {
                        *mem=0xAA55AA55;
                        asm(""::::"memory");
                        if(*mem!=0xAA55AA55)
                        mem_count=0;
                }

                asm(""::::"memory");
                *mem=a;

        } while (memkb<4096 && mem_count!=0);

        __asm__ __volatile__("movl %%eax, %%cr0", :: "a" (cr0) : "eax");
```

```
        mem_end = memkb<<20;
        mem = (ULONG*) 0x413;
        bse_end= (*mem & 0xFFFF) <<6;

        outb(0x21, irq1);
        outb(0xA1, irq2);
}
```

Manual Probing in ASM

This is the "least unsafe" algorithm for memory probing. It is "non-destructive" of the contents of memory, and is just generally better than the above C code.

Notes:

- **NEVER** use manual probing unless you absolutely **MUST**. This implies that the manual probing code is only used for dodgy old computers, and that the code for manual probing needs to be designed for dodgy old computers (assumptions that are "good" assumptions for modern computers, like "it's unlikely that an ISA video card is present", don't apply).
- minimize the amount of manual probing you do. For example, if the BIOS supports "Int 0x12" (they all do) then use it to avoid probing RAM below 1 MB. If "Int 0x15, AH=0x88" says there's 0xFFFF KB at 0x00100000 and you think there's more (because a 16-bit value can't tell you there's more if there is) then probe from the end of known RAM (not from 0x00100000).
- don't assume that writes to "non-RAM" won't be cached (flush the cache with WBINVD or CLFLUSH after testing to make sure you're testing the physical address and not the cache).
- don't assume that writes to "non-RAM" won't be retained due to bus capacitance (use a dummy write at a different address to avoid this, so you read back the dummy value and not the test value if there's no RAM at the address).
- don't write a set value to an address and read it back to test for RAM (for e.g. "mov [address],0x12345678; mov [dummy],0x0; wbinvd; cmp [address],0x12345678") because you might be unlucky and find a ROM that contains the same value you're using. Instead try to modify what's already there.
- test the last bytes of each block, not the first bytes of each block, and make sure that the size of each block is less than 16 KB. This is because some older motherboards relocate the RAM underneath the ROM area to the top of memory (e.g. a computer with 2 MB of RAM might have 128 KB of ROM from 0x000E0000 0x000FFFFF and RAM from 0x00100000 to 0x0020FFFF.
- don't make any assumptions about the "top of memory". Just because the last byte of RAM is at 0x0020FFFF doesn't mean that there's 2176 KB of RAM installed, and just because there's 2 MB of RAM installed doesn't mean that the last byte of RAM will be at 0x001FFFFF.
- it's better to assume that memory holes are present (and risk skipping some RAM) than to assume that memory holes aren't present (and risk crashing). This means assuming that the area from 0x00F00000 to 0x00FFFFFF can't be used and not probing this area at all (it's possible that some sort of ISA device is in this area, and that any write to this area can cause problems).

```
;Probe to see if there's RAM at a certain address
;
; note: "dummy" -> a known good memory address that has nothing importani
;
;Input
```

```
; edx   Maximum number of bytes to test
; esi   Starting address
;
;Output
; ecx   Number of bytes of RAM found
; esi   Address of RAM


probeRAM:
    push eax
    push ebx
    push edx
    push ebp
    mov ebp,esi             ;ebp = starting address
    add esi,0x00000FFF      ;round esi up to block boundary
    and esi, ~0x00000FFF    ;truncate to block boundary
    push esi                ;Save corrected starting address for later
    mov eax, esi            ;eax = corrected starting address
    sub eax, ebp            ;eax = bytes to skip from original starting a
    xor ecx,ecx             ;ecx = number of bytes of RAM found so far (r
    sub edx,eax             ;edx = number of bytes left to test
    jc .done                ;  all done if nothing left after rounding
    or esi,0x00000FFC       ;esi = address of last uint32_t in first bloc
    shr edx,12              ;edx = number of blocks to test (rounded dowr
    je .done                ; Is there anything left after rounding?

.testAddress:
    mov eax,[esi]           ;eax = original value
    mov ebx,eax             ;ebx = original value
    not eax                 ;eax = reversed value
    mov [esi],eax           ;Modify value at address
    mov [dummy],ebx         ;Do dummy write (that's guaranteed to be a di
    wbinvd                  ;Flush the cache
    mov ebp,[esi]           ;ebp = new value
    mov [esi],ebx           ;Restore the original value (even if it's not
    cmp ebp,eax             ;Was the value changed?
    jne .done               ; no, definitely not RAM -- exit to avoid dam
                            ; yes, assume we've found some RAM

    add ecx,0x00001000      ;ecx = new number of bytes of RAM found
    add esi,0x00001000      ;esi = new address to test
    dec edx                 ;edx = new number of blocks remaining
    jne .testAddress        ;more blocks remaining?
                            ;If not, we're done

.done:
    pop esi                 ;esi = corrected starting address (rounded up
    pop ebp
    pop edx
    pop ebx
    pop eax
```

```
    ret
```

Further Notes:

- Depending on how it's used some of the initial code could be skipped (e.g. if you know the starting address is always aligned on a 4 KB boundary).
- the WBINVD instruction seriously affects performance because it invalidates all data in all caches (except the TLB). It would be better to use CLFLUSH so you only invalidate the cache line that needs to be invalidated, but CLFLUSH isn't supported on older CPUs (and older computers is what this code is for). For older computers It shouldn't be too slow because the speed difference between cache and RAM wasn't so much and there's usually only a small amount of RAM (e.g. 64 MB or less). Modern computers have a lot more RAM to test and rely on caches a lot more. For example, an 80486 with 32 MB of RAM might take 1 second, but a Pentium 4 with 2 GB of RAM might take 30 seconds or more.
- Increasing the block size (e.g. testing every 16 KB instead of testing every 4 KB) will improve performance (and increase risk). 16 KB blocks is probably safe, and larger blocks sizes are not safe. Very large block sizes (e.g. testing every 1 MB) will probably work on modern computers (but you shouldn't need to probe at all on modern computers), and anything larger than 1 MB is guaranteed to give wrong results regularly.
- WBINVD isn't supported on 80386 and older computers. This means that for 80386 and older you can't flush the cache, but this shouldn't matter (for 80386 and older memory ran at the same speed as the CPU so there was no need for a cache). You will need to flush cache on later CPUs though. Having one routine that uses WBINVD and another routine that doesn't use WBINVD is probably better than doing an "if (CPU_is_80486_or_newer) { WBINVD }" in the middle of the loop.

See Also

Threads

- Grub's multiboot memory map (http://www.osdev.org/phpBB2/viewtopic.php?t=11391) , featuring real examples of GRUB/BIOS reported memory map.