digital whisper Technologic Papers

משחקים ונהנים עם PCI

מאת מתן קוטיק

הקדמה

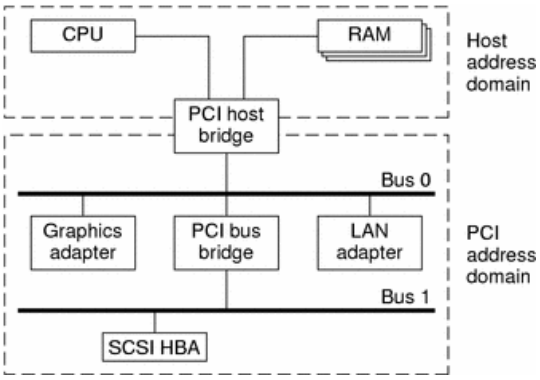
אני רחוק מאוד מלהיות איש חומרה ולכן המאמר יסקור את העולם העשיר של רכיבי ה-PCI מנקודת המבט של התוכנה שנהנית מהעושר שיש לעולם זה להציע. המטרות העיקריות של מאמר זה הן:

- לספק לקורא את הכלים לכתוב דרייבר (driver) לרכיב חומרה גם ללא ספריות מערכת הפעלה.
- לסקור את האתגרים שעולים מהיכולות של הרכיבים, להסביר כיצד ה-IOMMU פותר המון מהם וכיצד הוא פועל.

במהלך המאמר אציג קטעי קוד מתוך תשתית קוד אישית שלי שלצערי לא אוכל להעלות אותה באופן מלא. על מנת שתוכלו לתרגל אני ממליץ למצוא כלי/ספרייה שמאפשרת לכם גישה לזיכרון הפיזי לפני/אחרי עליית מערכת ההפעלה.

PCI

נתחיל בכך שרכיב החומרה הכי משמעותי במחשב הוא המעבד. המעבד הוא זה שמאפשר לנו להריץ תוכנות ולוגיקות מורכבות, אבל, ישנם עוד רכיבי חומרה שעוזרים לנו ביום יום. ביניהם אפשר למצוא כרטיסי רשת, מסך, קול ואפילו עכבר ומקלדת. כל אותם רכיבים מתחברים ב-BUS כלשהו למעבד. בדרך כלל אפשר לדמות את זה לסוג של רשת שכוללת את המעבד ורכיבי החומרה והם "מדברים" ביניהם בפרוטוקולים מוגדרים. בעוד שחלק מהרכיבים שתיארתי, לדוגמא: עכבר ומקלדת, יתחברו בחיבור מסוג USB. אחרים, יתחברו בחיבור בשם PCI (קיצור של *Peripheral Component Interconnect*).



[מקור: https://docs.oracle.com]

בואו נסתכל על הפלט של הפקודה lspci:

```
a@a:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03)
```

ניתן לשים לב בבירור שלצד כל רכיב ברשימה (מצד שמאל) יש כתובת שבנויה מ-3 מספרים. המבנה הוא:

```
Bus : Slot : Function
```

המבנה אשר מתואר כאן הוא היררכי. לכל Bus יש מספר Slots שאליהם מתחברים רכיבים ורכיב יכול ליחצן מספר functions ולתפקד כמו מספר רכיבים שונים, לדוגמא, כרטיס רשת אחד שמציג את עצמו למערכת ההפעלה כמו מספר כרטיסי רשת שונים. בגלל שיש לי חולשה לא מוסברת לכרטיסי רשת (NICs) אשתמש בהם כדוגמא בהמשך המאמר.

לאותם רכיבים שמתחברים ב-PCI יש מספר יכולות ביניהן:

- IRQ (Interrupt Request) - המעבד רשאי לדרוש מרכיבי החומרה לבצע פעולות מסוימות ואז לדגום אותן עד שהם סיימו לבצע את הפעולה (Poll Mode), צורת עבודה זו בזבזנית בכוח עיבוד ולכן יש לרכיבי חומרה דרך ליזום בקשה שתקבל מענה על ידי המעבד ומערכת ההפעלה והיא Interrupt. כך, ניתן לייצר תצורת עבודה אסינכרונית יעילה. במהלך המאמר לא אתייחס לעולם של ה-Interrupts אלא אתמקד ב-DMA למרות שרוב המאמר רלוונטי לשניהם.
- DMA (Direct Memory Access) - במקום שכתיבה/קריאה לזיכרון (RAM) תעשה על ידי המעבד כמתווך, מה שעלול לבזבז המון כוח עיבוד, לרכיבים השונים יש גישה ישירה לזיכרון הפיזי ובכך הם יכולים לבצע את עבודתם ללא התערבות המעבד. לדוגמא חבילה שמגיעה מהרשת תיכתב ישירות לזיכרון על ידי הכרטיס רשת.

PCI Configuration Space

ניתן להשפיע על רכיבי ה-PCI השונים בעזרת קונפיגורציה שהם מיחצנים. כל רכיב PCI (הערה: חוץ מה-Host Bridge) מחויב ליחצן 256 בתים של אזור קונפיגורציה. הגישה אל אותו אזור תעשה על ידי שני פורטי ה-I/O הבאים:

0xCF8 - פורט זה נקרא גם CONFIG_ADDRESS. הוא משמש לציין לאיזה קונפיגורציה של רכיב PCI אנחנו מעוניינים לגשת ובאיזה offset. לכן, הערך המספרי שנכתוב לפורט יהיה בהתאם:

Bit 31	Bits 30-24	Bits 23-16	Bits 15-11	Bits 10-8	Bits 7-0
Enable Bit	Reserved	Bus Number	Device Number	Function Number	Register Offset[1]

- הביטים 0-7 הראשונים יציינו את ה-offset, חשוב לשים לב כי הם מחויבים להצביע לכתובת שהיא בכפולות של 4-בתים מתוך הקונפיגורציה ולכן 2 הביטים התחתונים חייבים להיות אפסים.
- הביטים 8-23 יציינו את המיקום של רכיב ה-PCI שאליו אנחנו מעוניינים לגשת (לפי ההיררכיה).
- הביט ה-31 צריך להיות דלוק על מנת לאפשר גישה ישירה ל-CONFIG_DATA.

0xCFC - פורט זה נקרא גם CONFIG_DATA. לאחר שקבענו CONFIG_ADDRESS את הכתובת והרכיב אליו נרצה לגשת, כעת ניתן לגשת אליו דרך ה-CONFIG_DATA. פקודות in/out לפורט יבצעו קריאה/כתיבה בהתאם. כעת, בואו נתבונן ב-2 השדות הראשונים של אזור הקונפיגורציה:

31 … 16		15 … 0		
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base Address Registers				10h 14h 18h 1Ch 20h 24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

Figure 6-1: Type 00h Configuration Space Header

[מקור: http://www-subatech.in2p3.fr/~electro/projets/alice/dimuon/trigger/jtag/data/PCI/pci_conf_space.pdf]

- **Vendor ID** - 2 בתים שמציינים את ה-ID של היצרן. רכיבי Intel לדוגמא יכילו את הערך 0x8086. הערך 0xFFFF מציין שלא קיים רכיב בכתובת שצוינה.
- **Device ID** - 2 בתים שמציינים מזהה ייחודי של הרכיב שניתנו על ידי היצרן.

ולכן, בתהליך העלייה של מערכת ההפעלה יקרה התהליך הבא:

1. המערכת תסרוק את כל ה-slots האפשריים בכל bus.
2. תקרא את ה-DWORD הראשון של כל רכיב (VendorID).
3. במידה והערך הוא לא 0xFFFF:
   a. יש לבדוק את ה-DeviceID ועל פי זה לבחור את הדרייבר המתאים לרכיב, ניתן לראות מזהים של רכיבים באתר הבא.

ומה זה אותו דרייבר אתם שואלים? זה פשוט קוד שיודע לגשת ולטפל בשאר המבנה שהוצג ולאחר מכן בשאר אזורי הזיכרון של הרכיב שגם אותם ניתן להסיק מהמבנה. אותם שינויי קונפיגורציה יעשו על פי הוראות היצרן של אותו רכיב.

האם אותו דרייבר חייב לשבת במערכת ההפעלה? מסתבר שלא. ב-open source בשם DPDK שמשמש לבצע פעולות תקשורת מהירות ככה שכל הריצה היא מה-user space קיימים המון דרייברים לכרטיסי רשת שונים. הקסם הזה נעשה על ידי דרייבר ראשוני שכל מהותו היא לתת ל-user space את הרשאות הגישה הנחוצות לזיכרון של רכיבי ה-PCI ולאחר מכן כל הלוגיקה נעשית מה-user space בלבד ללא התערבות מערכת ההפעלה כלל.

וכמו שנותנים גישה, ניתן גם לקחת גישה. צירפתי לכם דוגמת קוד מתוך hypervisor קטן אישי שלי. לא ארחיב כאן מה זה hypervisor, גם על זה יש מאמר מעולה מהאתר.

אסביר בקצרה שקיימת תמיכה במעבד להוסיף שכבה של קוד כך שכאשר פעולות מסוימות יקרו, פעולת המעבד תיפסק והשכבה שהזכרנו תקבל זמן ריצה לטפל בפעולה שהתרחשה. ניתן להכיל זאת גם על פקודות I/O. הקוד שאציג לכם כאן הוא באמת proxy לפקודות ה-I/O שמבצעת מערכת ההפעלה:

```
__attribute__((always_inline))
VOID INLINE __out(IN DWORD port, IN DWORD data)
{
    asm volatile("out %1, %0" :: "d" (port), "a" (data));
}

__attribute__((always_inline))
VOID INLINE __in(DWORD port, DWORD_PTR data)
{
    asm volatile("in %1, %0" : "=a" (*data): "d" (port));
}

if (!is_in)
{
    __out(port, data->guestRegisters.rax);
    if (port == 0xCF8)
    {
        data->currentCPU->currentConfigAddress = config_address;
    }
}
else
{
    DWORD in_data;
    __in(port, &in_data);
    if (port == 0xCFC)
    {
        if(data->currentCPU->currentConfigAddress.slotNumber == 0x03)
        {
            in_data = 0xffff;
        }
    }
    data->guestRegisters.rax = in_data;
}
```

כאן המימוש שלי היה כך:

1. כאשר יש פקודת OUT לפורט 0xCF8 בצע את הפקודה ושמור את הערך שנכתב על פי המבנה שתיארתי קודם.
2. כאשר יש פקודת IN תבדוק האם היא מיועדת לרכיב PCI ב-slot מספר 3.
3. אם כן, תחליף את התוצאה שאמורה לחזור באוגר RAX ב-0xFFFF.

ובכך, מערכת ההפעלה לא מצליחה לזהות שקיים רכיב PCI בחיבור למרות שהוא נוכח ולכן כאשר נבצע lspci נקבל:

```
Last login: Fri Feb 25 14:11:42 UTC 2022 on tty1
a@a:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
```

וזאת אותה הרשימה ממקודם ללא כרטיס הרשת שהיה קיים שם קודם ב-slot 3. אולי לא התרשמתם במיוחד ממופע הקסמים שהוצג פה אבל המשמעות היא שניתן לחסום/לתת גישה אל רכיבי PCI גם ברמת התוכנה.

PCI Express

בשנת 2004 הוצג הדור החדש של חיבור ה-PCI שהוא PCI Express. השינויים באו לידי ביטוי במבנה של ה-bus, בקצבים וכנראה בעוד כמה פרמטרים (שוב, אני לא איש של חומרה).

הגישה לרכיב מנקודת המבט של התוכנה גם היא השתנתה. ארגיע אתכם קודם כל ואומר שלמרות שהשתנה הגישה לרכיבים עדיין התקן שומר על Backwards compatibility ולכן השיטה שהצגתי לכם קודם לכן עם הפורטים 0xCFC ו-0xCF8 עדיין תעבוד.

השינוי המרכזי הוא שכעת לא נעשה שימוש בפורטי I/O אלא במרחב זיכרון שניתן לכתוב ולקרוא ממנו (MMIO). המשמעות היא שכתיבה וקריאה לאזור הזיכרון שמוגדר תשפיע על ההתנהגות של הרכיב באופן ישיר - ניתן לחשוב על זה כמו על אוגר חומרה שממוקם ב-RAM ולא במעבד עצמו. בנוסף, מרחב הזיכרון גדל מ-256 בתים ל-4096 בתים. וכעת אציג לכם כיצד מוצאים את אותו מרחב זיכרון של הרכיב.

ACPI ("Advanced Configuration and Power Interface") כולל בתוכו אוסף של טבלאות שנטענות בשלב מוקדם בתהליך העלייה לזיכרון, עוד לפני מערכת ההפעלה. אותן טבלאות מכילות מידע על החומרות שקיימות במערכת. לא ארחיב כיצד למצוא את הטבלאות בזיכרון אבל המידע נמצא כאן.

הטבלה שאנחנו מחפשים מכילה את ה-Signature "MCFG" והמבנה שלה הוא:

Offset	Length	Description
0	4	Table Signature ("MCFG")
4	4	Length of table (in bytes)
8	1	Revision (1)
9	1	Checksum (sum of all bytes in table & 0xFF = 0)
10	6	OEM ID (same meaning as other ACPI tables)
16	8	OEM table ID (manufacturer model ID)
24	4	OEM Revision (same meaning as other ACPI tables)
28	4	Creator ID (same meaning as other ACPI tables)
32	4	Creator Revision (same meaning as other ACPI tables)
36	8	Reserved
44 + (16 * n)	16	Configuration space base address allocation structures. Each structure uses the following format:

ACPI Header

Offset	Length	Description
0	8	Base address of enhanced configuration mechanism
8	2	PCI Segment Group Number
10	1	Start PCI bus number decoded by this host bridge
11	1	End PCI bus number decoded by this host bridge
12	4	Reserved

[מקור: https://wiki.osdev.org/PCI]

- תחילת המבנה (עד 32 offset כולל) הוא ACPI Header כללי.
- לאחר מכן יש 8 בתים שהם "Reserved"
- אחרי כן מתחיל מערך שאת גודלו ניתן להסיק מאורך הטבלה, לרוב המערך יכיל רשומה אחת.

המערך מורכב מהשדות הבאים:

- Base Address - הכתובת של תחילת אזור הקונפיגורציה.
- PCI Segment Group Number - PCI-E הגדיל את ההיררכיה של החיבור והוסיף מעבר ל-BUS הגדרה בשם Domain (segment), רמה נוספת שמאפשרת יותר חיבורים. לרוב יהיה רק Domain אחד שערכו 0.
- Start PCI bus - לאיזה Bus ראשוני מתייחס המקטע (לרוב 0).
- End PCI bus - עד איזה bus מתייחס המקטע (לרוב 0xFF).
- 4 בתים "reserved"

ה-Base Address מציין את ההתחלה של אזור הקונפיגורציות, על מנת לאתר קונפיגורציה של רכיב ספציפי ניעזר בחישוב הבא:

```
Base_address + ((Bus - start_bus) << 20 | Slot << 15 | Function << 12)
```

כעת שיש לנו את כל המידע למצוא את מרחב הזיכרון אחשוף לכם קצת מ-"גן המשחקים" שאני משתמש בו.

סביבת בדיקות

הדרך שאני משתמש בה לסמלץ סוגי סביבות היא בעזרת VM מעל QEMU. מסתבר שבברירת המחדל של QEMU נעשה שימוש ב-PCI BUS ולא ב-PCI Express. במידה וכן רוצים מכונה וירטואלית שתומכת בPCI-E יש להוסיף את הדגל "-machine q35". ניתן לקרוא עוד על הפיצ'ר המעולה הזה של QEMU שאשתמש בו כאן.

כעת ננסה להגיע בעצמנו לבתים הראשונים של הכרטיס רשת, המבנים אותם תיארתי הם:

```
typedef struct {
    UINT32      Signature;
    UINT32      Length;
    BYTE        Revision;
    BYTE        Checksum;
    BYTE        OemId[6];
    UINT64      OemTableId;
    UINT32      OemRevision;
    UINT32      CreatorId;
    UINT32      CreatorRevision;
} __attribute__ ((packed)) ACPI_DESCRIPTION_HEADER;
typedef struct {
    UINT64      base_address;
    WORD        pci_segment_group_number;
    BYTE        start_pci_bus;
    BYTE        end_pci_bus;
    UINT32      reserved;
} __attribute__ ((packed)) BASE_ADDRESS_STRUCTURE;

typedef struct {
    ACPI_DESCRIPTION_HEADER        header;
    UINT64                         reserved;
    BASE_ADDRESS_STRUCTURE         base[N];
} __attribute__ ((packed)) ACPI_MCFG_HEADER;
```

לאחר שנמצא את המצביע לטבלת ה-"MCFG" נוכל להדפיס את הפרמטרים השונים שלה ואת ההתחלה של אזור הקונפיגורציה של הכרטיס רשת שנמצא כעת ב-slot מספר 2:

```
ACPI_MCFG_HEADER* mcfg = (ACPI_MCFG_HEADER*)mcfgTable;
UINT64 address = mcfg->base[0].base_address + ((0 << 20) | (2 << 15) | (0 << 12)); // 00:02.0

Print("*****************\n");
Print ("PCI domains %d\n", (mcfg->header.Length -
sizeof(ACPI_DESCRIPTION_HEADER))
                                    / sizeof(BASE_ADDRESS_STRUCTURE));
Print ("PCI Base Address %8\n", mcfg->base[0].base_address);
Print ("PCI Start Bus Number %1\n", mcfg->base[0].start_pci_bus);
Print ("PCI End Bus Number %1\n", mcfg->base[0].end_pci_bus);
Print ("Slot 2 Address: %8\n", address);
Print ("First Bytes: %8\n", *(UINT32*)address);
Print("*****************\n");
```

וה-output של הריצה הוא:

```
*****************
PCI domains 1
PCI Base Address 00000000B0000000
PCI Start Bus Number 00
PCI End Bus Number FF
Slot 2 Address: 00000000B0010000
First Bytes: 0000000010D38086
*****************
```

ניתן לשים לב שקיבלנו את ה-DeviceID וה-VendorID של הכרטיס רשת (מעל הצבעים האדום והצהוב).

במידה ורוצים לבצע השוואה לתמונת הזיכרון של המכונה ניתן להריץ את הפקודה:

```
sudo cat /proc/iomem
```

ונקבל:

```
b0000000-bfffffff : PCI MMCONFIG 0000 [bus 00-ff]
  b0000000-bfffffff : Reserved
```

ובבירור ניתן להבחין שזה אזור הזיכרון שמצאנו בעצמנו (PCI Base Address).

שימו לב: שאזור הזיכרון מסומן בתור "Reserved" והמשמעות היא שבעלייה של מערכת ההפעלה היא לא תשתמש בזיכרון הזה למבנים שלה. גישה לאזור זיכרון זה צריכה להיעשות בצורה חכמה מכיוון שהוא משפיע ישירות על רכיבי החומרה של המערכת.

דרייבר ב-Bash

בחלקים הקודמים הצגתי לכם את הידע הדרוש לכתיבה של דרייבר לחומרה. כפי שהזכרתי, עיקר העבודה היא לעבוד עם מרחב הקונפיגורציה שמיחצן הרכיב כלפי הזיכרון. כעת, נכתוב דרייבר בסיסי לרכיב שמחזיק ביכולות DMA. בעזרת הרכיב נקרא ונכתוב לזיכרון.

הרכיב שנכתוב לו דרייבר הוא רכיב מעניין בשם "edu". edu הוא רכיב חומרה וירטואלי ששייך ל-QEMU והוא משמש למטרות לימוד (נכתב באופן ייעודי עבור סטודנטים). ניתן להתבונן גם בקוד המקור שלו. בשביל שהמכונה הווירטואלית תכיל אותו יש להוסיף את הדגל:

```
-device edu
```

השלב הבא הוא לקרוא את המפרט של הרכיב על מנת להבין מה הוא מיחצן וכיצד ניתן לקנפג אותו. המידע כולו ממוקם כאן.

ראשית נתייחס להגדרות הראשונות:

```
PCI specs
---------
PCI ID: 1234:11e8
PCI Region 0:
   I/O memory, 1 MB in size. Users are supposed to communicate with the card
   through this memory.
```

מכאן ניתן לראות את ה-ID של המוצר שכמובן מורכב מהצמד האהוב: VendorID ו-DeviceID. לאחר מכן ניתן לשים לב שלרכיב יש מגה של אזור זיכרון שמשמש לקונפיגורציה שלו (MMIO). איפה ממוקם אזור הזיכרון הזה? ניתן לאתר אותו בעזרת ה-BAR (Base Address Register) הראשון מאזור הקונפיגורציה הכללי (BAR0):

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	Base address #0 (BAR0)			
0x5	0x14	Base address #1 (BAR1)			
0x6	0x18	Base address #2 (BAR2)			
0x7	0x1C	Base address #3 (BAR3)			
0x8	0x20	Base address #4 (BAR4)			
0x9	0x24	Base address #5 (BAR5)			

[מקור: https://wiki.osdev.org/PCI]

מכיוון שאני יודע שאתם כבר מסוגלים לעבוד עם אזור הקונפיגורציה ולחלץ ערכים, אגלה לכם שבלינוקס ניתן להריץ:

```
lspci -v
```

ובתשובה ניתן לראות שקיים אזור זיכרון בכתובת **0xfea00000**, ערך זה קיים גם ב-BAR0:

```
00:03.0 Unclassified device [00ff]: Device 1234:11e8 (rev 10)
        Subsystem: Red Hat, Inc. Device 1100
        Flags: bus master, fast devsel, latency 0, IRQ 11
        Memory at fea00000 (32-bit, non-prefetchable) [size=1M]
        Capabilities: <access denied>
```

כעת, נמשיך לקרוא את המפרט של הרכיב ונתמקד ב-API של ה-DMA, החלק הראשון הוא הכתובות שבהם נכניס את ההוראות לרכיב. כאשר נרצה לקנפג ערך בפשטות נכתוב לכתובת שלו מספר ברוחב 4 בתים. הגישה נראית כאילו הערך נכתב לזיכרון אבל בעצם הכתיבה "נתפסת" לפני שהיא מתרחשת וההוראה עוברת לרכיב (בגלל זה המרחק בין כתובת לכתובת לא באמת משמעותי):

```
0x80 (RW) : DMA source address
            Where to perform the DMA from.

0x88 (RW) : DMA destination address
            Where to perform the DMA to.

0x90 (RW) : DMA transfer count
            The size of the area to perform the DMA on.

0x98 (RW) : DMA command register, bitwise OR
            0x01 -- start transfer
            0x02 -- direction (0: from RAM to EDU, 1: from EDU to RAM)
            0x04 -- raise interrupt 0x100 after finishing the DMA
```

מנגנון ה-DMA של הרכיב עובד כך: הרכיב מחזיק בכתובת 0x40000 מערך. **האופציות היחידות הן לכתוב אל אותו מערך מהזיכרון או לקרוא ממנו ערך אל הזיכרון**. הדרייבר שאנחנו נכתוב יהיה ממומש ב-script של bash והוא יבצע את הפעולה הבאה:

1. הוא יגדיר לרכיב כתובת ממנה יש להעתיק 4 בתים.
2. לאחר מכן הוא יגדיר לרכיב כתובת לכתוב אליה את ה-4 בתים שהוא העתיק, הכתובת תמוקם 4 בתים אחרי הכתובת הראשונה.

התוצאה: שכפול של 4 בתים בזיכרון, זה הכל. ואיך נממש את זה ב-bash? נשתמש בכלי שנקרא busybox שמסוגל לבצע כתיבה לזיכרון הפיזי.

הקוד ייראה כך:

```
#!/bin/bash

ADDR=0xfea00000
DMA_ADDR=0x9fb00
sudo busybox devmem $((DMA_ADDR)) 32 0xffffffff

sudo busybox devmem $((ADDR + 0x80)) 32 $((DMA_ADDR))
sudo busybox devmem $((ADDR + 0x88)) 32 0x40000
sudo busybox devmem $((ADDR + 0x90)) 32 4
sudo busybox devmem $((ADDR + 0x98)) 32 1

sleep 2

sudo busybox devmem $((ADDR + 0x80)) 32 0x40000
sudo busybox devmem $((ADDR + 0x88)) 32 $((DMA_ADDR + 0x4))
sudo busybox devmem $((ADDR + 0x90)) 32 4
sudo busybox devmem $((ADDR + 0x98)) 32 3

sleep 2

sudo busybox devmem $((ADDR + 0x4))
```

הקוד בנוי מהשלבים הבאים:

1. בהתחלה נגדיר את הכתובת של אזור הקונפיגורציה והזיכרון אותו נרצה לשכפל. נכתוב גם ערך בן 4 בתים באותה כתובת.
2. לאחר מכן נעתיק את 4 הבתים למערך בזיכרון של הרכיב.
3. נפעיל sleep על מנת לוודא שהעברה הסתיימה. דרך יותר אלגנטית היא לבדוק שהרכיב כיבה את הדגל שהדלקנו בכתובת 0x98 שמסמל התחלה של העברה (כן, גם הרכיב עורך את מרחב הזיכרון)
4. נעתיק את 4 הבתים מהמערך של הרכיב לכתובת שנמצאת בהיסט של 4 (המשך הזיכרון שסיפקנו).
5. נבצע שוב sleep עד לסיום ההעברה.
6. לבסוף נדפיס את הערך בכתובת שכתבנו אליה לוודא שהתהליך הצליח.

איזה כיף! הצלחנו לכתוב דרייבר ב-bash. אכן, אבל, ישנו חלק שהחסרתי שהוא קריטי להצלחה של התהליך. קיים דגל בזיכרון שמאפשר למנוע ה-DMA של הרכיב גישה.

בתוך מרחב הקונפיגורציה (של רכיבי PCI באופן כללי) קיים שדה בשם "Command Register" שנראה כך:

Command Register

Here is the layout of the Command register:

Bits 11-15	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Interrupt Disable	Fast Back-to-Back Enable	SERR# Enable	Reserved	Parity Error Response	VGA Palette Snoop	Memory Write and Invalidate Enable	Special Cycles	Bus Master	Memory Space	I/O Space

[מקור: https://wiki.osdev.org/PCI]

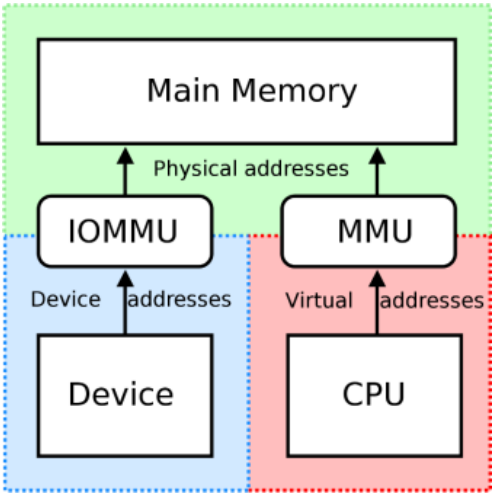
Bit2 בשדה מציין יכולת שנקראת "Bus Master", היכולת הזאת מאפשרת לרכיב לבצע DMA ולכן על מנת שהמנגנון יעבוד יש להדליק את הדגל הזה במרחב הקונפיגורציה בתחילת הפעולה של הדרייבר, ואיך עושים את זה? בעזרת כל הכלים שסיפקתי לכם במאמר עד כה אתם כבר מסוגלים לאתר את מרחב הקונפיגורציה של הרכיב ולערוך אותו בעצמכם.

IOMMU

אחד המנגנונים החשובים בעולם רכיבי ה-PCI הוא ה-Input-Output Memory Management Unit או בראשי תיבות IOMMU.

כפי שהזכרתי קודם לכן לרכיבים יש גישה ישירה לזיכרון הפיזי (DMA), יכולת זאת מעלה מספר אתגרים:

1. בעיה ברכיב מסוים או ב-driver שלו, או לחילופין תוקף שמנצל את היכולת של הרכיב יכולים להשפיע על זיכרון פיזי ולגרום לנזק חמור למערכת. דוגמא טובה היא תוקף שמתאם עם הכרטיס רשת כתובת שבה הוא מצפה לקבל את חבילות התקשורת, והכתובת מכילה דף שאין לתוקף הרשאות כתיבה אליו.
2. בעולם הווירטואליזציה המכונה הווירטואלית בדרך כלל לא תהיה מודעת למרחב הכתובות הפיזי אלא הכתובות שלה יומרו על ידי טבלה בשם "EPT" . מכיוון שהמכונה הוירטואלית פועלת על פי ה-EPT והרכיב מכיר אך ורק כתובות פיזיות אזי המערכת הפעלה במכונה לא מסוגלת לתאם עם הרכיב כתובת פיזית שתשמש את שניהם.



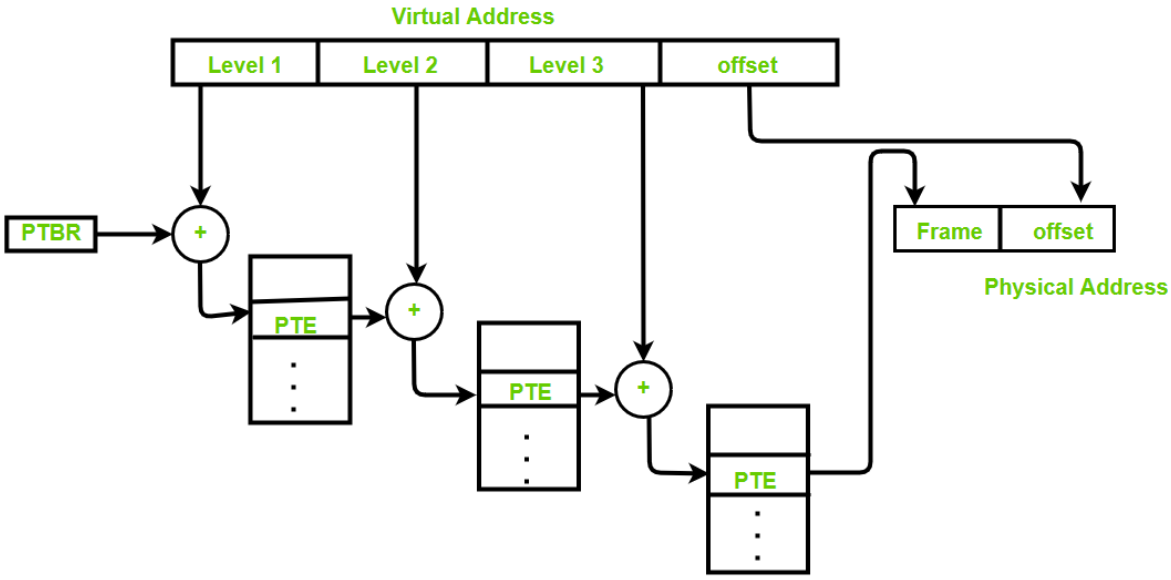
[מקור: https://www.wikipedia.com]

ולכן, רכיב ה-IOMMU נוצר על מנת להתמודד עם אתגרים אלו על ידי כך שהוא מונע מהרכיב גישה לכתובות פיזיות ומבצע תרגום בין כתובת שהרכיב ביקש לבין כתובת פיזית בדומה למנגנון ה-paging.

איך נראה המנגנון?

כמו שציינתי ה-IOMMU דומה למנגנון ה-paging.

Paging הוא מנגנון היררכי שמשמש להמרה בין כתובת וירטואלית לפיזית. המנגנון עובד בצורה הבאה:



3 Level paging system

[מקור: https://www.geeksforgeeks.org/multilevel-paging-in-operating-system]

בעצם הכתובת שלנו מורכבת מאוסף של offsets אל תוך טבלאות כאשר ה-offset הראשון הוא יותר גדול ומשמש כ-offset אל תוך דף פיזי. הטבלאות מצויות במבנה היררכי כך שבכל טבלה נמצא המצביע לטבלה הבאה. המצביע הראשון ממוקם בד"כ באוגר, ב-x86 המצביע שמור ב-CR3.

במקרה והגודל שמוגדר לדף הוא 4KB אז ה-offset הראשון יהיה בגודל של 12-bit (מגיע עד ל-4096 בתים) ואז ה-offset השני יהיה באורך 9-bit כי הוא ישמש כ-Index אל תוך טבלה שאורכה 512 (2 בחזקת 9).

בגלל שכל רשומה בטבלה תצביע להתחלה של page אחר בעצם ניתן למפות ככה 512*4096 בתים שזה 2MB. אם נוסיף עוד טבלה באורך 512 נוכל למפות 512*2MB שזה 1Gb, טבלה נוספת תיתן לנו 512Gb ובעזרת עוד טבלה נגיע כבר לכמה טרות של זיכרון.

בגלל זה לא נשתמש בכתובת שאורכה יותר מ-48-bit (offset ראשוני ו-4 טבלאות), פשוט כי אין צורך.

אציין שיכולים לקרות 2 מצבים:

- הראשון הוא לוותר על טבלה מימין ולהוסיף את הביטים שלה מתוך הכתובת ל-offset שמצביע אל תוך דף פיזי. ואז לדוגמא, 12 הביטים הראשונים יתחברו עם 9 הביטים שאחריהם ויתנו לנו להצביע אל תוך דף שגודלו 2MB. למרות שישנה פחות טבלה עדיין הכתובת כולה היא באורך שהייתה.

- השני הוא שמוגדר לנו מספר טבלאות מקסימלי. אם מוגדרים לנו לדוגמא עד 3 טבלאות המשמעות היא שהכתובת המקסימלית היא באורך 39-bit (12 + 3*9) ואז אנחנו יכולים למפות עד 512GB.

Table 5. Second-level Paging Structures

Paging Structure	Entry Name	Physical Address of Structure	Bits Selecting Entry	Page Mapping
Second-level PML4 table	SL-PML4E	Context-entry (or Extended-Context-entry)	47:39	N/A
Second-level Page-directory-pointer table	SL-PDPE	SL-PML4E[1]	38:30	1-GByte page (if Page Size (PS) field is Set)
Second-level Page directory	SL-PDE	SL-PDPE	29:21	2-MByte page (if Page-Size (PS) field is Set)
Second-level Page table	SL-PTE	SL-PDE	20:12	4-KByte page

1. For implementations supporting only 3-level paging structures for second-level translation, there is no SL-PML4E, and the physical address to locate the SL-PDPE is provided by the context-entry (or extended-context-entry).

[מקור: https://www.intel.com/]

ב-IOMMU הטבלאות יחסית זהות אך קיים מיפוי לכל רכיב ולכן זה נראה כך:

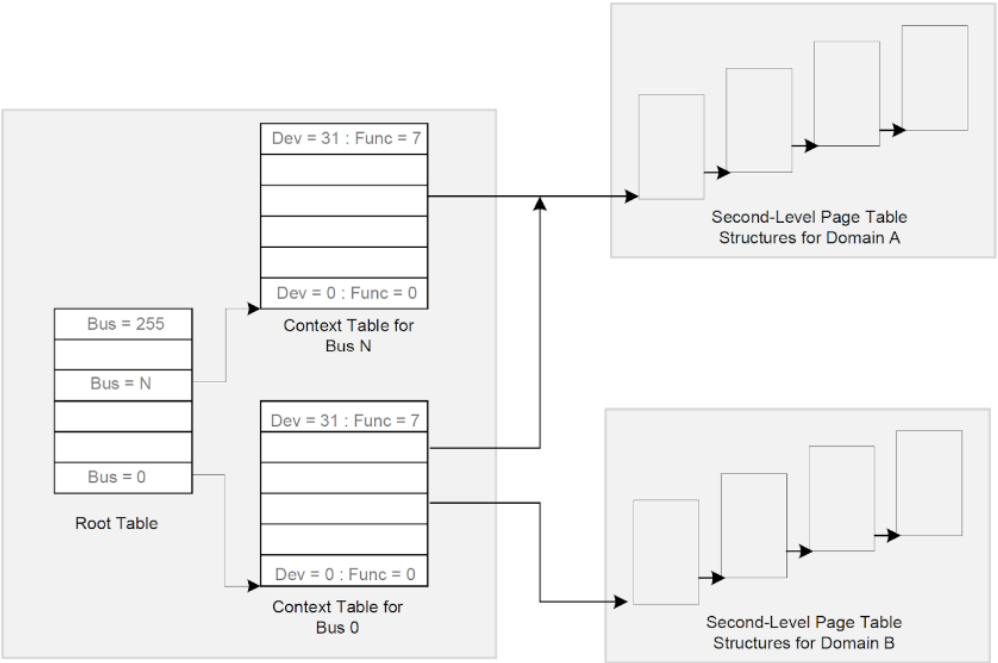


Figure 3-7. Device to Domain Mapping Structures in Legacy Mode

[מקור: https://www.intel.com/]

תחילה קיים מצביע ל-Root Table (בהמשך אראה כיצד מאתרים את המצביע). לאחר מכן יש מצביע בכל bus שמוביל לטבלה שמכילה רשומה עבור כל רכיב (צירוף של slot ו-func). מכל רשומה של רכיב ניתן להגיע אל הטבלאות המרה שלו.

חשוב לציין שיכולים להיות כמה קבוצות של busים כפי שהזכרתי קודם, ולכן יכול להיות מצביע ל- Root Table לכל Segment (Domain).

וכיצד מתממשקים עם המנגנון?

לפני שנצלול אל תוך החלק הטכני של המנגנון והצורה שבה הוא ממומש אתן שני דגשים חשובים:

ראשית, היישום של המנגנון משתנה בין סוגי מעבדים. אני אתייחס לטכנולוגיה של intel ששמה בישראל (ובתפוצות) הוא VT-d. לעיתים יש תפיסה שהמנגנון הוא חלק ממנגנון הווירטואליזציה והם חייבים לפעול יחדיו אבל לא כך הדבר, מדובר על מנגנון נפרד אשר יכול להתקיים ללא וירטואליזציה כלל ולשמש לצרכים מגוונים.

שנית סביבת הבדיקות בה אני משתמש היא עדיין QEMU, ולכן, אשתמש במנגנון שלהם ("vIOMMU") שאמור לדמות את המנגנון המקורי. הדגלים שהוספתי ל-QEMU הם:

```
-machine q35,accel=kvm,kernel-irqchip=split -device intel-
iommu,intremap=on
```

כעת אתחיל את החלק הפרקטי והמעניין שיסביר כיצד קורה הקסם שנקרא IOMMU בפועל. אתם תוהים כיצד ניתן למצוא את כל המידע בצורה הכי מפורטת ומסודרת, וכמובן שהתשובה היא בתיעוד של Intel.

ניתן להבחין שמדובר על כמה מאות עמודים לא פשוטים לקריאה ולכן סיכמתי לכם מימוש מסוים שמבוסס על הפרויקט HelloIommuPkg, בנוסף, אפנה אתכם לחלקים בתיעוד על ידי ציון של מספר הפרק בסוגריים.

וגם כאן הכל מתחיל מטבלת ACPI אחת בשם "DMAR". המבנה שלה מתחיל כשאר טבלאות ה-ACPI (8.1) אך בסופה קיים מערך של אובייקטים שמהם נוכל להבין היכן למקם את קונפיגורצית ה-IOMMU שלנו.

המערך בנוי מאובייקטים שונים שבתחילתם יש את המבנה הבא:

```
typedef struct {
    WORD    Type;
    WORD    Length;
} __attribute__ ((packed)) ACPI_DMAR_STRUCTURE_HEADER;
```

המבנה מאפשר לנו להבדיל ביניהם על פי Type ולדלג עליהם לפי Length. האובייקט שיעניין אותנו הוא אובייקט בשם DRHD (8.3) שבו מצוין היכן הקונפיגורציה ממוקמת (מדובר על MMIO).

הערכים שנעתיק מתוך מרחב הזיכרון שמפנה אותנו ה-DRHD הם:

- היכן ממוקם מרחב הזיכרון.
- ה-Capabilities שיש לקונפיגורציה (ארחיב בהמשך).

הקוד שמבצע את התהליך שתיארתי נראה כך:

```
UINT64 endOfDmar;
const ACPI_DMAR_STRUCTURE_HEADER* dmarHeader;
UINT64 discoveredUnitCount;

//
// Walk through the DMAR table, find all DMA-remapping hardware unit
// definition structures in it, and gather relevant information into DmarUnits.
//
discoveredUnitCount = 0;
endOfDmar = (UINT64)DmarTable + DmarTable->Header.Length;
dmarHeader = (const ACPI_DMAR_STRUCTURE_HEADER*)(DmarTable + 1);

while ((UINT64)dmarHeader < endOfDmar)
{
    if (dmarHeader->Type == ACPI_DMAR_TYPE_DRHD)
    {
        if (discoveredUnitCount < MaxDmarUnitCount)
        {
            const ACPI_DMAR_DRHD_HEADER* dmarUnit;

            dmarUnit = (const ACPI_DMAR_DRHD_HEADER*)dmarHeader;
            DmarUnits[discoveredUnitCount].RegisterBaseVa = dmarUnit->RegisterBaseAddress;
            DmarUnits[discoveredUnitCount].Capability.Uint64 =
                *(UINT64*)(DmarUnits[discoveredUnitCount].RegisterBaseVa + R_CAP_REG);
            DmarUnits[discoveredUnitCount].ExtendedCapability.Uint64 =
                *(UINT64*)(DmarUnits[discoveredUnitCount].RegisterBaseVa + R_ECAP_REG);
        }
        discoveredUnitCount++;
    }
    dmarHeader = (const ACPI_DMAR_STRUCTURE_HEADER*)
                        ((UINT64)dmarHeader + dmarHeader->Length);
}
```

לאחר שחילצנו את המידע הנחוץ לנו מהמבנה נוכל לבצע את הבדיקות הנחוצות לנו להמשך. בקוד שהתבססתי עליו הייתה תמיכה רק בכתובות באורך 48-bit. בגלל שה-vIOMMU תומך בכתובות באורך 39-bit ערכתי את הקוד בהתאם. איך גיליתי באיזה אורך של כתובת תומך המנגנון? בעזרת ה-capabilities שאספנו קודם. המבנה שלהם הוא זה:

```
typedef union {
  struct {
    BYTE          ND:3; // Number of domains supported
    BYTE          AFL:1; // Advanced Fault Logging
    BYTE          RWBF:1; // Required Write-Buffer Flushing
    BYTE          PLMR:1; // Protected Low-Memory Region
    BYTE          PHMR:1; // Protected High-Memory Region
    BYTE          CM:1; // Caching Mode

    BYTE          SAGAW:5; // Supported Adjusted Guest Address Widths
    BYTE          Rsvd_13:3;

    BYTE          MGAW:6; // Maximum Guest Address Width
    BYTE          ZLR:1; // Zero Length Read
    BYTE          Rsvd_23:1;

    WORD         FRO:10; // Fault-recording Register offset
    WORD         SLLPS:4; // Second Level Large Page Support
    WORD         Rsvd_38:1;
    WORD         PSI:1; // Page Selective Invalidation

    BYTE          NFR:8; // Number of Fault-recording Registers

    BYTE          MAMV:6; // Maximum Address Mask Value
    BYTE          DWD:1; // Write Draining
    BYTE          DRD:1; // Read Draining
```

```
    BYTE          FL1GP:1; // First Level 1-GByte Page Support
    BYTE          Rsvd_57:2;
    BYTE          PI:1; // Posted Interrupts Support
    BYTE          Rsvd_60:4;
  } Bits;
  UINT64      Uint64;
} __attribute__ ((packed)) VTD_CAP_REG;
```

והבדיקה שביצעתי היא לפי ה-flag שנקרא SAGAW:

Bits	Access	Default	Field	Description
12:8	RO	X	SAGAW: Supported Adjusted Guest Address Widths	This 5-bit field indicates the supported adjusted guest address widths (which in turn represents the levels of page-table walks for the 4KB base page size) supported by the hardware implementation. A value of 1 in any of these bits indicates the corresponding adjusted guest address width is supported. The adjusted guest address widths corresponding to various bit positions within this field are: • 0: Reserved • 1: 39-bit AGAW (3-level page-table) • 2: 48-bit AGAW (4-level page-table) • 3: Reserved • 4: Reserved Software must ensure that the adjusted guest address width used to set up the page tables is one of the supported guest address widths reported in this field.

```
if ((DmarUnits[i].Capability.Bits.SAGAW & (1 << 1)) == 0)
{
    Print("Unit %d SAGAW %d\n", i, DmarUnits[i].Capability.Bits.SAGAW);
    Print("Unit %d does not support 39-bit AGAW (3-level page-table) %d\n",
            i,
            DmarUnits[i].Capability.Uint64);
    return STATUS_FAILURE;
}
```

מכיוון שאני מעוניין להשתמש גם בדפים בגודל 2MB אבדוק גם את התמיכה בהם:

37:34	RO	X	SLLPS: Second Level Large Page Support	This field indicates the large page sizes supported by hardware. A value of 1 in any of these bits indicates the corresponding large-page size is supported. The large-page sizes corresponding to various bit positions within this field are: • 0: 21-bit offset to page frame (2MB) • 1: 30-bit offset to page frame (1GB) • 2: Reserved • 3: Reserved Hardware implementations supporting a specific large-page size must support all smaller large-page sizes. i.e., only valid values for this field are 0000b, 0001b, 0011b.

[מקור: https://www.intel.com/]

```
if ((DmarUnits[i].Capability.Bits.SLLPS & (1 << 0)) == 0)
{
    Print("Unit %d does not support 2MB second level large pages\n", i);
    return STATUS_FAILURE;
}
```

כעת נוכל להגדיר את הטבלאות. המבנים שנשתמש בהם הם:

```
typedef union {
  struct {
    UINT32  Present:1;
    UINT32  Reserved_1:11;
    UINT64  ContextTablePointer: 52;
    UINT64  Reserved_64;
  } Bits;
  struct {
    UINT64  Uint64Lo;
    UINT64  Uint64Hi;
  } Uint128;
} __attribute__ ((packed)) VTD_ROOT_ENTRY;

typedef union {
  struct {
    UINT32  Present:1;
    UINT32  FaultProcessingDisable:1;
    UINT32  TranslationType:2;
    UINT32  Reserved_4:8;
    UINT64  SecondLevelPageTranslationPointer: 52;
    UINT32  AddressWidth:3;
    UINT32  Ignored_67:4;
    UINT32  Reserved_71:1;
    UINT32  DomainIdentifier:16;
    UINT32  Reserved_88:8;
    UINT32  Reserved_96:32;
  } Bits;
  struct {
    UINT64  Uint64Lo;
    UINT64  Uint64Hi;
  } Uint128;
} __attribute__ ((packed)) VTD_CONTEXT_ENTRY;

typedef union {
  struct {
    UINT32  Read:1;
    UINT32  Write:1;
    UINT32  Execute:1;
    UINT32  ExtendedMemoryType:3;
    UINT32  IgnorePAT:1;
    UINT32  PageSize:1;
    UINT32  Ignored_8:3;
    UINT32  Snoop:1;
    UINT64  Address: 40;
    UINT32  Ignored_52:10;
    UINT32  TransientMapping:1;
    UINT32  Ignored_63:1;
  } Bits;
  UINT64    Uint64;
} __attribute__ ((packed)) VTD_SECOND_LEVEL_PAGING_ENTRY;

typedef struct _DMAR_TRANSLATIONS
{
    VTD_ROOT_ENTRY RootTable[256];

    //
    // The context table can be multiple but all root entries set up by this
    // project point to the same, single context table, hence this is not
```

```
    // ContextTable[256][256]. This table is made up of 256 entries.
    //
    VTD_CONTEXT_ENTRY ContextTable[256];
    VTD_SECOND_LEVEL_PAGING_ENTRY pdpe[512];
    VTD_SECOND_LEVEL_PAGING_ENTRY pde[512][512];
}__attribute__ ((packed)) DMAR_TRANSLATIONS;
```

שלושת המבנים הם לפי ההירכיה (RootTable->Context->Table). כעת אצרף את מקטע הקוד שאחראי לקנפג את המבנה של הטבלאות:

```
STATUS
FillTranslations(DMAR_TRANSLATIONS* Translations)
{

    VTD_ROOT_ENTRY defaultRootValue;
    VTD_CONTEXT_ENTRY defaultContextValue;
    VTD_SECOND_LEVEL_PAGING_ENTRY* pde;
    VTD_SECOND_LEVEL_PAGING_ENTRY* pd;
    UINT64 pml4Index;
    UINT64 destinationPa;

    ASSERT(((UINT64)Translations % PAGE_SIZE) == 0); // Check alignment to 4096

    defaultRootValue.Uint128.Uint64Hi = defaultRootValue.Uint128.Uint64Lo = 0;
    defaultRootValue.Bits.ContextTablePointer = (UINT64)Translations->ContextTable >> 12;
    defaultRootValue.Bits.Present = TRUE;
    for (UINT64 bus = 0; bus < 256; bus++)
    {
        Translations->RootTable[bus] = defaultRootValue;
    }

    defaultContextValue.Uint128.Uint64Hi = defaultContextValue.Uint128.Uint64Lo = 0;
    defaultContextValue.Bits.DomainIdentifier = 1;
    defaultContextValue.Bits.AddressWidth = 1; // 001b: 39-bit AGAW (3-level page table)
    defaultContextValue.Bits.SecondLevelPageTranslationPointer = (UINT64)Translations->pdpe >> 12;
    defaultContextValue.Bits.Present = TRUE;
    for (UINT64 i = 0; i < 256; i++)
    {
        Translations->ContextTable[i] = defaultContextValue;
    }
    destinationPa = 0;

    for (UINT64 pdptIndex = 0; pdptIndex < 512; pdptIndex++)
    {
        pde = Translations->pde[pdptIndex]; // Next-level Pointer
        Translations->pdpe[pdptIndex].Uint64 = (UINT64)pde;
        Translations->pdpe[pdptIndex].Bits.Read = TRUE;
        Translations->pdpe[pdptIndex].Bits.Write = TRUE;

        for (UINT64 pdIndex = 0; pdIndex < 512; pdIndex++)
        {
            pde[pdIndex].Uint64 = destinationPa;
            pde[pdIndex].Bits.Read = TRUE;
            pde[pdIndex].Bits.Write = TRUE;
            pde[pdIndex].Bits.PageSize = TRUE;

            if ((destinationPa <= 0x9fb00) && ((destinationPa + LARGE_PAGE_SIZE) >= 0x9fb00))
            {
                pde[pdIndex].Bits.Read = FALSE;
            }
            destinationPa += LARGE_PAGE_SIZE;  //2MB
        }
    }
    return STATUS_SUCCESS;
}
```

אסביר את השלבים שהקוד מיישם:

1. שינוי ה-RootTable כך שכל הרשומות שלו יצביעו אל אותה טבלה של ContextEntry. הסיבה היא שעל מנת לפשט נשתמש באותן טבלאות לכל הרכיבים במימוש זה.
2. יצירת ContextEntry אחד והשמה שלו לכל הטבלה. הרשומה תצביע לטבלה הראשונה בהיררכיה של הטבלאות מיפוי.
3. כעת יוצרים מיפוי 1-1, ז"א שכתובת וירטואלית תהיה שווה לכתובת הפיזית שמתאימה לה. בנוסף נשתמש בדפים בגודל 2MB. ניתן לשים לב שבמבנה של הדפים מצוינות גם ההרשאות הניתנות (כתיבה/קריאה). הוספתי שינוי קטן שהדף של הכתובת 0x9fb00 יהיה ללא הרשאות קריאה ל-DMA, הסיבה היא ההדגמה שאציג בהמשך.

לבסוף, מה שנשאר הוא רק להפעיל את המנגנון:

```
#define R_RTADDR_REG      0x20
#define R_GCMD_REG        0x18

#define B_GMCD_REG_SRTP   (1 << 30)
#define B_GMCD_REG_TE     (1 << 31)
#define B_GSTS_REG_TE     (1 << 31)
```

```
typedef union _VTD_ROOT_TABLE_ADDRESS_REGISTER
{
    struct
    {
        UINT64 Reserved_1 : 10;              // [9:0]
        UINT64 TranslationTableMode : 2;     // [11:10]
        UINT64 RootTable : 52;               // [63:12]
    } Bits;
    UINT64 AsUInt64;
} VTD_ROOT_TABLE_ADDRESS_REGISTER;
```

```
VTD_ROOT_TABLE_ADDRESS_REGISTER rootTableAddressReg;

rootTableAddressReg.AsUInt64 = 0;
rootTableAddressReg.Bits.RootTable = (UINT64)Translations->RootTable >> 12;

*(UINT64*)(dmarUnits[0].RegisterBaseVa + R_RTADDR_REG) = rootTableAddressReg.AsUInt64;
*(UINT32*)(dmarUnits[0].RegisterBaseVa + R_GCMD_REG) = B_GMCD_REG_SRTP;

for (; (*(UINT32*)(dmarUnits[0].RegisterBaseVa + R_GSTS_REG) & B_GSTS_REG_RTPS) == 0;)
{
}
```

תחילה נבצע השמה של הטבלאות שהכנו אל תוך האוגר (שוב, אוגר בזיכרון) שאחראי לציין היכן הטבלת תרגום. לאחר מכן נמתין עד שהפעולה הושלמה ועודכנה באוגר שאחראי על ה-Status.

המימוש של הפעולה מבוסס על התיעוד הבא:

Bits	Access	Default	Field	Description
30	WO	0	SRTP: Set Root Table Pointer	Software sets this field to set/update the root-table pointer used by hardware. The root-table pointer is specified through the Root Table Address (RTADDR_REG) register. Hardware reports the status of the 'Set Root Table Pointer' operation through the RTPS field in the Global Status register. The 'Set Root Table Pointer' operation must be performed before enabling or re-enabling (after disabling) DMA remapping through the TE field. After a 'Set Root Table Pointer' operation, software must globally invalidate the context-cache and then globally invalidate the IOTLB. This is required to ensure hardware uses only the remapping structures referenced by the new root-table pointer, and not stale cached entries. While DMA remapping is active, software may update the root table pointer through this field. However, to ensure valid in-flight DMA requests are deterministically remapped, software must ensure that the structures referenced by the new root table pointer are programmed to provide the same remapping results as the structures referenced by the previous root-table pointer. Clearing this bit has no effect. The value returned on a read of this field is undefined.

[מקור: https://www.intel.com/]

לאחר מכן יש לרענן את ה-cache, הקוד שמבצע זאת ממוקם בפרויקט שצירפתי, תוכלו להסתכל שם.

ומולכם הקוד שמפעיל את המנגנון כולו:

```
*(UINT32*)(dmarUnits[0].RegisterBaseVa + R_GCMD_REG) = B_GMCD_REG_TE;
for (; (*(UINT32*)(dmarUnits[0].RegisterBaseVa + R_GSTS_REG) & B_GSTS_REG_TE) ==
0;)
{
}
```

מעולה! הפעלנו את המנגנון עבור ה-DRHD הראשון. בשביל להתייחס אל כל ה-Segments יש לעשות את אותה פעולה עבור כל היחידות.

הדגמה

ועכשיו ננסה לראות את המנגנון בפעולה. נשתמש בדרייבר שכתבנו קודם לכן ל-EDU. אזכיר 2 נקודות חשובות:

- הדרייבר משפיע על הרכיב ככה שיקרא ויכתוב לכתובת הפיזית 0x9fb00.
- בקונפיגורצית ה-IOMMU שיישמנו חסמנו לכל הרכיבים במערכת הרשאות קריאה לדף שמכיל את הכתובת 0x9fb00.

וכעת, כאשר נריץ את הדרייבר נקבל מ-QEMU את הפלט הבא:

```
qemu-system-x86_64: vtd_iova_to_slpte: detected slpte permission error (iova=0x9fb00, level=0x2, slpte=0x82, write=0)
qemu-system-x86_64: Interrupt Mask set, irq is not generated
qemu-system-x86_64: vtd_iommu_translate: detected translation failure (dev=00:03:00, iova=0x9fb00)
```

ניתן בבירור לשים לב להודעה שמציינת "translation failure". היא מתארת שהרכיב 00:03:00 (הערך של ה-edu) ניסה לבצע פעולת קריאה (write=0) בכתובת 0x9fb00 והפעולה נכשלה.

כמובן שאם ניתן לרכיב הרשאות קריאה לדף, הפעולה תסתיים בהצלחה!

סיכום

במהלך המאמר הצגתי לכם את טכנולוגיית ה-PCI וכיצד מערכת ההפעלה מתממשקת איתה (עד לרמת הברזלים). חשוב לי להדגיש את החשיבות של טכנולוגיה זאת, לדוגמא במידה ואין קונפיגורציית IOMMU רכיבים יכולים לכתוב/לקרוא ישירות מהזיכרון ובכך לסכן תשתיות וירטואליזציה. מעבר לכך, היכולת לפנות ישירות לרכיבי חומרה יכולה לסייע בפרוייקטי חומרה ואבטחה רבים ומגוונים.

קצת עליי

שמי מתן קוטיק, אני בן 22 כיום ראש צוות מחקר בתחום הסייבר. את הנושא שתיארתי פה הכרתי לאחר מחקר שעשיתי בתחום הוירטואליזציה שבמהלכו נחשפתי לטכנולוגיה הכל כך חשובה של ה-IOMMU שמשום מה היא פחות מתועדת ומוסברת מטכנולוגיות אחרות. ולכן, הכנתי את המאמר כשירות לציבור לחסוך לחוקרים ומפתחים דוברי עברית המון זמן (והמון ייאוש), ולתת להם את הכלים להיכנס לעולם מסויים שלפעמים פחות נעים להעמיק בו.

במידה ויש שאלות מסויימות או רעיונות לפרוייקטים אשר מבוססים על המאמר אשמח שתפנו אליי למייל האישי שלי: matank001@gmail.com

מקורות

- https://docs.oracle.com/cd/E19683-01/806-5222/hwovr-22/
- https://wiki.qemu.org/Features/Q35
- https://github.com/qemu/qemu/blob/master/docs/specs/edu.txt
- https://github.com/tandasat/HelloIommuPkg
- https://lettieri.iet.unipi.it/virtualization/vt-directed-io-spec.pdf
- https://projectacrn.github.io/2.1/developer-guides/hld/hv-dev-passthrough.html
- https://www.geeksforgeeks.org/multilevel-paging-in-operating-system/
- https://wiki.osdev.org
- https://docs.kernel.org/PCI/pci.html
- https://standa-note.blogspot.com/2020/05/introductory-study-of-iommu-vt-d-and.html
- https://www.digitalwhisper.co.il/files/Zines/0x7C/DW124-1-NativeHyperVisoer.pdf