うぶんちゅ！ まがじん ざっぱ～～ん♪

体験版

vol.3

うぶんちゅ！　まがじん ざっぱ〜ん♪
vol.3 体験版

うぶんちゅ！　まがじん ざっぱ〜ん♪ の仲間たち　著

2015-07-25 版　　うぶんちゅ！　まがじん ざっぱ〜ん♪ 編集部 発行

Chap. 1

OpenNebula で PCI passthrough

おおたあきひこ

青羽家は OpenNebula で家庭内クラウドを構築し、ここなちゃんとママの二人で計算機リソースをやりくりしています。

ここなちゃんは 14 歳の誕生日に以前から欲しかった Mellanox の ConnectX-3 InfiniBand HCA を買ってもらいました。でも彼女の仮想マシンはホストしている物理マシンから論理的に分離されているため、InfiniBand HCA を物理マシンに挿しても仮想マシンからはアクセスできません。

ママもせっかくのプレゼントが台無しで大弱りです。

1.1 OpenNebula ノススメ

OpenNebula はシンプルで安定したクラウド管理ツールです。簡単な説明や Ubuntu でのインストール方法、セットアップ方法については、gihyo.jp の Ubuntu Weekly Recipe 第 345 回と 346 回に書かせていただいたので、そちらをご参照ください。

- http://gihyo.jp/admin/serial/01/ubuntu-recipe/0345
- http://gihyo.jp/admin/serial/01/ubuntu-recipe/0346

OpenNebula の特徴の一つにカスタマイズの容易さがあります。公式には提供されていない機能でも、なんとなく実装できたりできなかったりします。

今回のざっぱ〜んはハードウェアがテーマということなので、Ubuntu & OpenNebula 環境で物理マシン上のハードウェアデバイスを仮想マシンに提供する方法に取り組んでみたいと思います。

1.2 前置き

- OpenNebula 環境での、OpenNebula デーモンが稼働するマシンをフロントエンドと呼称します。
- OpenNebula 環境での、仮想マシンが稼働するマシンをホストノードと呼称します。
- フロントエンド、ホストノードは、いずれも Ubuntu 14.04.2 で構築しています。
- 使用する OpenNebula のバージョンは 4.12.1 です。

- OpenNebula から作成する仮想マシンは、OpenNebula Marketplace の Ubuntu 14.04 KVM イメージを使用しています[*1]。
- ハイパーバイザは KVM を使用します。
- OpenNebula が一通りセットアップされ、各種リソースが登録されているものとします。
- ホストノードは CPU、チップセット、BIOS/UEFI が I/O 仮想化支援に対応しているものとします。また、kernel やドライバモジュールのオプションが適切に設定されているものとします。

1.3 PCI passthrough と SR-IOV

最近の CPU には I/O 仮想化支援機能を有するものがあります。x86_64 系では Intel の VT-d、AMD の AMD-Vi（以前は AMD IOMMU と呼ばれていました）がその機能です。これらの I/O 仮想化支援機能を利用して、ハイパーバイザが物理マシン上の PCI デバイスを仮想マシンに提供する技術が、PCI passthrough と SR-IOV です。

PCI passthrough

物理マシンの PCI デバイスを仮想マシン 1 台に排他的に割り当てます。
物理マシン上で複数の仮想マシンが稼働している場合、ある PCI デバイスを割り当てられるのはどれか 1 台の仮想マシンに限られます。

SR-IOV

PCI passthrough の上位種のような機能で、物理マシンの PCI デバイスを複数の仮想マシンに割り当てることができます。
ただし、対象のデバイスがハードウェア/ファームウェア的に SR-IOV に対応している必要があります。

ここなちゃんが誕生日に買ってもらった Mellanox ConnextX-3 InfiniBand HCA は SR-IOV に対応したデバイスです。ですが、筆者は残念ながら、このカードはおろか SR-IOV に対応したデバイスを一枚も持ち合わせていません。しかたがないので、どこのご家庭にもあるごく一般的な PCI デバイスであるところの、アースソフト PT3[*2]を PCI passthrough してみようと思います。

SR-IOV も PCI passthrough も設定方法はだいたい同じなので、ここなちゃんもこの原稿を読んで OpenNebula 環境で InfiniBand HCA を SR-IOV できるだろうと信じています。

1.4 libvirt から PCI passthrough

OpenNebula は libvirt を介して仮想マシンを操作します。そこでまずは libvirt だけで PT3 を PCI passthrough してみます。

まず、PCI passthrough したいデバイスのバス番号、スロット番号、ファンクション番号を lspci コマンドで調べます。PT3 は lspci コマンドでは"Multimedia controller: Altera Corporation

[*1] OpenNebula Marketplace: http://marketplace.opennebula.systems/appliance
[*2] PT1、PT2 でも大丈夫だと思います。筆者の環境では PT2 が現役ですが、製造が終了している点と、PCI スロットを備えたマザーボードが手に入りにくくなっている点を鑑みて、PT3 を例に取り上げました。

Device..."という名称で表示されます。

```
$ lspci| grep -i altera
06:00.0 Multimedia controller: Altera Corporation Device 4c15 (rev 01)
```

行頭の 06:00.0 がそれぞれ"バス番号":"スロット番号"."ファンクション番号"です。これを libvirt の Domail XML フォーマット[*3]の PCI デバイスアサイン設定に当てはめて記述します。

リスト 1.1: libvirt Domain XML の PCI デバイスアサイン設定

```
<hostdev mode='subsystem' type='pci' managed='yes'>
  <source>
    <address domain='0x0000' bus='0x06' slot='0x00' function='0x0'/>
  </source>
</hostdev>
```

が、この記述の Domain XML ファイルで virsh create すると、筆者のホストノードではエラーとなってしまいました。

hostdev の managed 属性を'yes' と指定した場合、libvirt はゲスト OS(仮想マシン) を起動する際に指定された PCI デバイスを自動的にホストからデタッチし、終了する際にはホストに再アタッチすることになっています。が、筆者の環境ではこれが正常に動作せず、PT3 のデタッチに失敗して仮想マシンが起動できませんでした。

不思議なことに、virsh nodedev-detach コマンドから手動でデタッチする場合には、PT3 のデタッチが成功しました。また、PT3 を手動でデタッチした後、hostdev managed='no' と明示的に指定[*4]した Domain XML ファイルで virsh create すると、仮想マシンが起動できました。

筆者の環境だけの挙動なのか、どの環境でも再現する挙動なのかは分かりません。環境によっては managed='yes' でも問題なく動作するのかもしれません。

1.4.1 デバイスの手動デタッチ

managed='yes' 指定での自動アタッチ/デタッチが正常に動作する場合は、この節は読み飛ばしてください。

virsh nodedev-detach コマンドにホストノードのデバイスを指定して実行することで、テバイスをホストノードからデタッチできます。デタッチされたデバイスは仮想マシンにパススルー可能な状態となります。

[*3] libvirt Domain XML format: http://libvirt.org/formatdomain.html#elementsHostDev

[*4] libvirt Domain XML format のドキュメントには managed 属性を省略した際の挙動は'no' とありますが、Ubuntu 14.04 でインストールされる libvirt 1.2.2 では'yes' の挙動に変わっているようです。そのため、libvirt による自動アタッチ/デタッチを使用したくない場合、managed='no' を明示的に指定する必要がありました。

Chap. 2

Let's note CF-RZ4 に Ubuntu をインストールしてみる話

Rakugou

どうも、お初にお目にかかります。Rakugou です。最近、迷いに迷った挙句、Let's note CF-RZ4 を購入しました。インストールから、タブレットモードで利用するための初期設定における悪戦苦闘の一部始終です。

2.1 我が家に Let's note がやってきた話

我が家に 2 台目の ultrabook　Let's note CF-RZ4 が到着しました。Ubuntu をインストールする前提で購入したので、Office もない、SSD の容量も 128GB な一番廉価なモデルを選択しました。初期価格の 17 万円を超える金額では、筆者にとってはとてもとても手に届くシロモノではなかったですが、購入を考えていた当初の金額からは破格の税込み 11 万 8000 円で購入出来ました。ちなみに、モデルチェンジされたそうで、すでに新しいモデルに入れかわっておりました。

とりあえず使ってみましたが、軽いです。745g という軽量ボディは外に持ち出すのに全く困らないですし、バッテリ容量も JEITA2.0 基準で約 10 時間と申し分ないです。ひざ上で使うには、10 インチという小型のサイズだと筆者の細くてか弱い太ももに収まりますし、キー配列も Fn キーと Ctrl キーを間違えてしまう[*1]ことを除いては打ちやすくていい感じですね。SD カードスロットもついてるので、出先で撮影した写真の管理もできますし、USB3.0 のポートが 3 口あり、VGA 端子や HDMI 端子もついているので、ご家庭でメイン端末として使用することができるくらいの拡張性もあります。Windows 8.1 の操作性にようやく慣れてきて、普通に使ってても文句の一つも出てこなくなってきた CF-RZ4 ですが、Ubuntu をインストールしてみたいと思います[*2]。

[*1] どうやら BIOS 画面で Ctrl キーと Fn キーの順序が替えることができるようです。筆者は慣れだと思ってデフォルトのまま設定を変更していませんが。

[*2] 読者の方々にとっては、既に重々承知の上であると思いますが、パソコンを解体したり、SSD を換装したり Windows とは違う OS をインストールしたりすると、保証の対象外になってしまいますので、くれぐれも実行する際は自己責任でお願いいたします。

2.2 Ubuntu インストールの話

Ubuntu をインストールしてみるにあたり、あまりにも Ubuntu がうまく動かなさすぎて、Windows を消してしまったがゆえにもう引き返せないという悲劇があったら困るので、Windows 環境は残しておきたい。ということで、Transcend の 128GB の M.2 SSD MTS800 を購入し、新しい SSD に Ubuntu をインストールすることにしました。M.2 な SSD を購入するときには注意が必要で、サイズに種類があるようです。SATA な SSD ではインチでサイズ表記がなされていたりしてわかりやすいですが、M.2 な SSD はそうでもないようです。購入段階できちんとサイズを調べてからでないと、「買ったのはいいが SSD が入らない」という可能性があるので注意する必要があります。少なくとも筆者が中を開けて調べた限りでは、もともと入っているのは SAMSUNG 製の MZ-NTE1280 という短辺 22mm、長辺 80mm の SSD でしたし、上述した SSD は問題なく入っております。

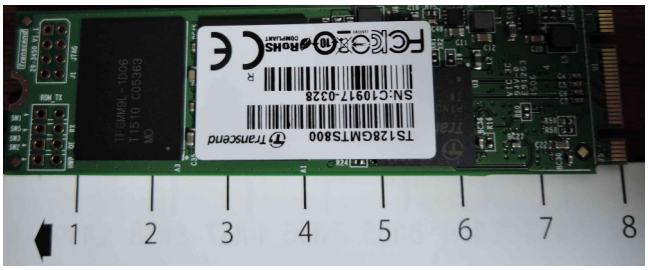

図: SSD のサイズ感。長さは某家電量販店の紙製測定器で計測しています。

2.2.1 Windows 8.1 の設定の話

まず高速スタート状態になっている Windows から、BIOS を立ち上げるための操作です。こちらは Windows 8.1 の画面で、右端をフリックして「設定」→「保守と管理」→「スタートアップ設定」の順にクリックしていきます。そうすると、再起動されますので、再起動された場合に、「トラブルシューティング」→「詳細のオプション」→「UEFI ファームウェアの設定」を選択します。選択するとさらに再起動され、BIOS 画面が開きます。さらに、Live USB イメージを接続している場合、起動の優先順位を USB デバイスを一番上にして、最初に Live USB イメージが起動するようにしましょう。スタートアップ設定は保存されないので、ここまでの作業を下見して、SSD を取り替えたりして実際のインストール作業を後回しにする場合は、再度上記の動作を行ってください。以後 Live USB イメージから Ubuntu をインストールしますが、現状の Windows 環境を壊してもいい、あるいはデュアルブートするという人は換装しないで、壊すと困るという方は、上述したような SSD を購入し、換装します。

2.2.2 CF-RZ4 開封の儀

換装方法は、バッテリを外し、裏面のネジを全て外し、カバーを開けます。すると、昆布のようなカバーで覆われていて、それをめくると M.2 SSD が見えます。その M.2 SSD の下部についているネジを外し、そこに SSD を取り付けます。おそらく、昆布のようなそれが取れると（というか解体した時点でそうですが）保証が効かなくなりますので、慎重に作業してください。粘着テープみたいなので貼られているので、剥がれやすくなっております。その後に元の通りネジを戻せば完了です。AC アダプタが刺さっているディスプレイ側の部分の 2 つのネジだけサイズが違うので、外したネジを整理する際はお気をつけ下さい。

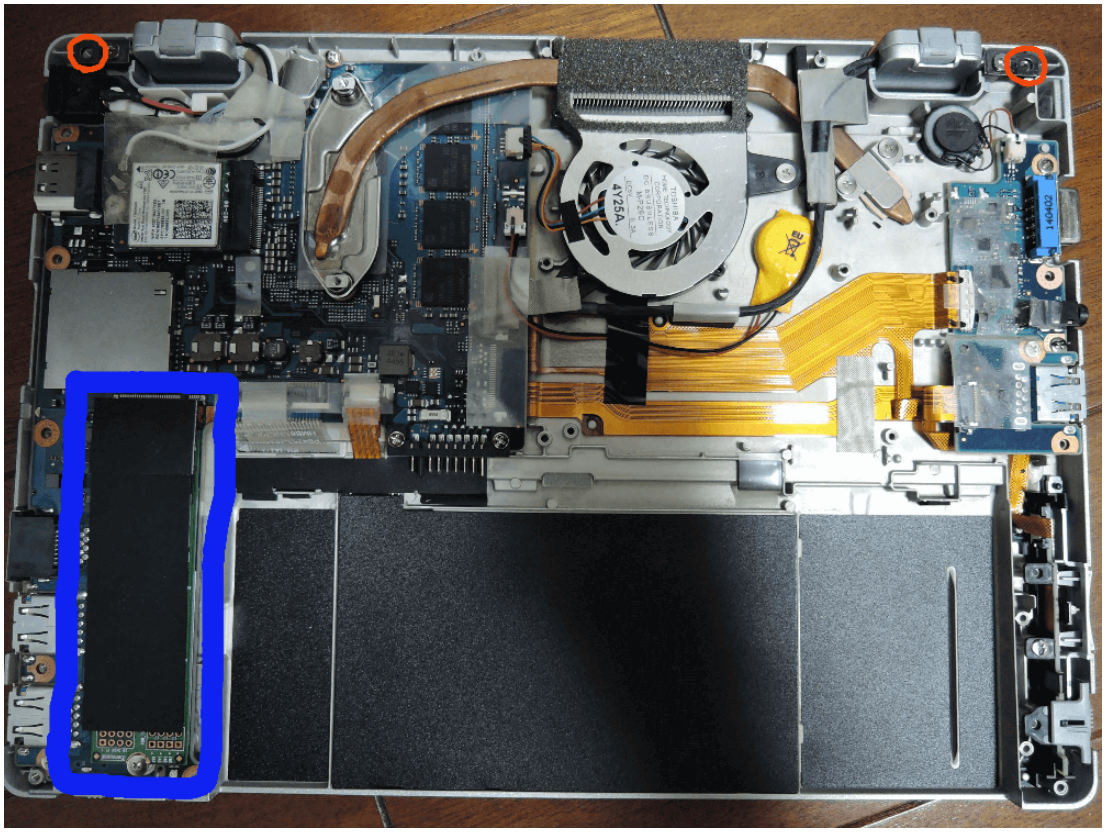

図: Let's note のあられもない姿。赤丸がネジのサイズの違う箇所で、青丸が昆布の部分、もとい、SSD のある場所。

2.2.3 Ubuntu をインストールする話

BIOS 画面から、事前に作成している Live USB イメージの起動を優先にして、起動します。そして、選択画面が現れるので、「Try Ubuntu without installing」を選択します。そうすると Ubuntu デスクトップが現れるかと思います。あとは、音声と動画、Bluetooth 等、必要なデバイスがちゃんと動作するかを確認してください。なお、明るさについてはうまく動作しないはずなので、かな

Chap. 3

21 世紀の Device Tree

柴田充也

Raspberry Pi は、今日最も広く売られて遊ばれている ARM ボードのひとつです。でもそのボードが長期にわたり、どう推移してきたかについて、本当にわかっているのは何でしょう？　特定のモデルのメモリ容量はどれくらいでしょうか？　I2C バスの数は同じでしょうか？　そもそも、カーネルは同じものを利用できるのでしょうか？

本章で答えようとするのはそのようなボードの「バリエーション」問題です。簡単に言うと、カーネルの「Device Tree」について解説します[*1]。

3.1　二つの世界

「**Device Tree (DT)**」はハードウェアの情報を記述するデータ構造です。たとえば CPU のコアの数、メモリの大きさや個々の周辺機器で使用する GPIO、割り込みコントローラーといったハードウェアの状態や、デバイスがバスにどのように接続されているかなどのハードウェアに近い設定はもちろんのこと、デバイスドライバーの中で使用するデータやさらにはカーネルの起動オプションなど、カーネルが起動する前に決まっているべきあらゆる情報を Device Tree の中に記述し、ブートローダーからカーネルへと渡すことができます。

ARM ボードのような組み込みデバイスは、用途に応じて微妙に周辺機器の組み合わせが異なる多様なモデルを持っている場合があります。そのモデルの違いをカーネルやデバイスドライバーで吸収するとなると、新モデルが出る度にカーネルが肥大化していくことになります。カーネルコンフィグでモデルごとに切り分けるとしても、今度はモデルごとに異なるカーネルをビルドしなくてはならないという問題が出てくるのです。

カーネルコードの複雑化はメンテナンスの困難さにもつながります。そこで繋がっている周辺機器やその設定、デバイスドライバーに渡すべきデータといったデバイスによって異なる情報を、カーネルコードの中から分離する仕組みが採用されました。これが Device Tree です。ちなみに Device Tree そのものは Power アーキテクチャなどで利用される Open Firmware で作成された仕組みであ

[*1] 本章では明示的な言及が無い限り、本家 Linux カーネルの 4.0.6 のコードを参照しています。Raspberry Pi に関するコードは、GitHub の Raspberry Pi プロジェクトにある同じバージョンのカーネルを参照しています。あとピケティさん、ごめんなさい。

り、ARM 固有のものというわけではありません。

そんな Device Tree には似たような意味を持ついくつかの用語がありますので、まとめておきます。

Device Tree Source (DTS)
: Device Tree のソースコードとも言うべきファイルです。カーネルソースにはこのファイルが存在し、デバイスごとに新しい DTS ファイルを作成して対応します。拡張子は.dts が使われます。また複数のボードで共通の部分については、「.dtsi」という拡張子のついたファイルに分離することがあります（詳しいことは「3.2.2 ツリーの分離とファイルのインクルード」を参照してください）。

Device Tree Compiler (DTC)
: テキスト形式の DTS ファイルをバイナリ形式の DTB ファイルにコンパイルするプログラムです。Linux カーネルは自前で dtc コマンドを持っていて、カーネルビルド時に指定した複数の DTB ファイルを生成するようになっています。Ubuntu リポジトリには device-tree-compiler という名前の dtc コマンドを提供するパッケージも存在します。

Device Tree Blob もしくは Binary (DTB)
: バイナリ形式の Device Tree ファイルです。ブートローダーやカーネルはこのバイナリ形式の DTB ファイルを解釈します。新しい U-Boot であれば、メモリ上の DTB を直接変更できる fdt コマンドが存在します。

Flattened Device Tree (FDT)
: ブートローダーやカーネルで扱う時の DTB のファイル形式です。DTB のデータ構造は DTS のツリー構造をより「平ら」にした形になっているためにこう呼ばれます。DTB と FDT はほぼ同じ意味で使われています。

Open Firmware (OF)
: Power アーキテクチャや SPARC アーキテクチャで採用されているファームウェアの規格です。Device Tree はもともと、Open Firmware とクライアントプログラム（ブートローダーやカーネル）との間で情報をやりとりするために開発されました。このため、カーネルコードでは Device Tree 対応コードは「CONFIG_OF」で囲みますし、DTB をパースするライブラリ関数は drivers/of/以下に「of_FOO()」といった名前でまとまっています。

本節ではまず Device Tree に対応していなかった頃の ARM ボードの初期化の流れを概観し、そして Device Tree 対応がどのように世界を変えたかを見てみることにしましょう。

3.1.1 Device Tree なき初期化

Device Tree がなかった頃、特定の ARM ボードをサポートするために、arch/arm/の下に「ボードファイル」と呼ばれるボードごとのコードを用意していました。

たとえば Ubuntu をサポートした ARM デバイスとして、「NetWalker」というシャープ製のデバイスが日本の家電量販店に並んでいたことがありました。この Linux カーネルのソースコードには、以下のような 3 種類のボードファイルが存在します[*2]。

```
$ grep -rI MACHINE_START arch/arm/mach-mx51/
.../mx51_erdos.c:MACHINE_START(MX51_BABBAGE, "SHARP PC-Z1")
.../mx51_babbage.c:MACHINE_START(MX51_BABBAGE, "Freescale MX51 Babbage Board")
.../mx51_3stack.c:MACHINE_START(MX51_3DS, "Freescale MX51 3-Stack Board")
```

NetWalker は Freescale 製の SoC である i.MX51 を採用していました。MX51_BABBAGE が i.MX51 EVK（Evaluation Kit）のコードで、MX51_3DS が i.MX51 PDK（Product Development Kit）のコードです。これを見ると NetWalker である PC-Z1 は MX51_BABBAGE のコードをベースに開発されていたことがわかりますね。

MACHINE_START は machine_desc 構造体を作るためのマクロです。このうち第一引数は「マシンタイプ番号」になります。Device Tree を使わない場合、このマシンタイプ番号は、ブートローダー（NetWalker の場合は RedBoot）がカーネルを起動する際に指定する第二引数[*3] として渡されます。カーネルはそのマシンタイプ番号にマッチする machine_desc 構造体を利用して、ボードの初期化を行うのです。現在有効なマシンタイプ番号は arch/arm/tools/mach-types にリストアップされています。

mx51_erdos.c の中では MX51_BABBAGE 用の周辺機器の情報や初期化コードを記述します。ちなみに異なるマシンタイプ番号であっても処理が似通っている場合は、同じボードファイルの中に複数の machine_desc 構造体を記述することもあります（例：arch/arm/mach-omap2/board-n8x0.c）。

SoC に接続している動的には検出されない周辺機器ごとに「platform_device 構造体」を設定して、「platform_device_register() 関数」で静的にデバイス情報を登録していくのが、初期化コードの役割です。platform_device 構造体には、デバイス名や番号、I/O メモリアドレスなどのリソース情報、ドライバーで利用するデータなどを設定します。PC-Z1 の場合は、フレームバッファや LCD、SDHCI などの platform_device 構造体を設定していました。

リスト 3.1: LCD の設定部分

```
static struct platform_device mxc_lcd_device = {
        .name = "lcd_sharp",
        .id   = 0,
        .dev  = {
                .release           = mxc_nop_release,
                .coherent_dma_mask = 0xFFFFFFFF,
                .platform_data     = &lcd_data,
        },
};

static void __init mxc_init_lcd(void)
{
```

[*2] BeagleBone Black などもっと適切な例があるのではという意見はもっともですが、せっかく日本の Ubuntu の同人誌なので NetWalker のコードを持ち出してみました。ちなみにこの Linux カーネルのベースバージョンは 2.6.28 です。NetWalker のコードが本家のカーネルに取り込まれることはありませんでしたが、ベースとなった i.MX51 シリーズのコードはより新しい Linux カーネルに取り込まれています。

[*3] ここで「第二引数」と言っているのは、ブートローダーからカーネルのエントリーポイント（stext）にジャンプする際の r1 レジスタです。

```
        if (!enable_vga) {
                platform_device_register(&mxc_lcd_device);
        }
}
```

platform_device_register() は、デバイス情報を登録する時に該当するドライバーがロード済みかを確認します。ロードしていない場合は、どこかのタイミングで個々のデバイスドライバーがロードされた時に、今度は platform_driver 構造体以下の .name に一致する platform_device が登録済みかどうか、つまり対応する周辺機器が存在するかどうかをを確認します。ロード済みだった場合、もしくは存在したと判断された場合に呼び出されるのがドライバー固有の probe() 関数です。

たとえばリスト 3.1 の.name が"lcd_sharp"に該当する LCD デバイスのドライバーは drivers/video/mxc/mxcfb_sharp_wsvga.c であり、リスト 3.2 がその platform_driver 構造体です。mxcfb_sharp_wsvga ドライバーがロードされると、その probe() 関数にリスト 3.1 の mxc_lcd_device が渡されるのです。

リスト 3.2: LCD のドライバー

```
static struct platform_driver lcd_driver = {
        .driver = {
                    .name = "lcd_sharp"},
        .probe = lcd_probe,
        .remove = __devexit_p(lcd_remove),
        .suspend = lcd_suspend,
        .resume = lcd_resume,
};
```

各デバイスドライバーはこの platform_device 構造体の情報を参考に、デバイスの初期化やデバイスファイルの作成といった操作を行います。

これが昔の Linux における、ARM ボードの初期化シーケンスでした。ここで抑えておくべきポイントは以下の 3 つです。

- ブートローダーから渡されるマシンタイプ番号
- 最初から静的に設定されている platform_device 構造体
- 初期化処理時に固定的に platform_device を登録する platform_device_register()

3.1.2 プラットフォームドライバーからデータドリブンへ

Device Tree を利用する場合、次のような違いがあります。

- 原則としてマシンタイプ番号は使用しない
- platform_device に該当するデータはすべて **Device Tree** に記述する
- 原則として platform_device_register() は使用しない

Chap. 4

ちょーなんさんちのノート PC 事情

長南 浩 (Ubuntu Japanese Team)

2015 年は Intel のタブレット向け低価格 CPU/Soc 攻勢の効果で Windows や Ubuntu Desktop が動きそうな小さいコンピュータが数多く登場した年となったが、オーソドックスなクラムシェル型のノート PC はあまり脚光を浴びず、不毛の時代となっていた。

とはいえ仕事や趣味でノート PC を使わないといけない事情があるときに、はたしてどんな選択肢があるのかというところを、筆者が経験した実例をもとに紹介する。

4.1 お出かけ用ノート PC が大ピンチ

まったくもって一大事である。

天板にステッカーを貼りまくり、メモリとストレージを増強して使っていた、筆者おでかけ用 PC の Gateway EC19C-N52C/B だったが、AC アダプタが壊れ (なんとか代替品を探した)、モニタは縦に線が入り表示が乱れ、バッテリは劣化して「ただのおもり」状態になってしまって、瀕死の状態になってしまった。CPU、メモリ、ストレージといったパーツが健在だけにちょっと悔しい気がする。

いくらノート PC がピンチでもイベントには PC を持っていかないといけないので、最近参加した OSC やオフライン・ミーティング兼リリースパーティでは魔改造 Lenovo G580 [*1] を持っていったのだが、なにげに 2.6kg の重量級。ノート PC だけではなく、私の腰や肩も大ピンチな状況になってしまっていた。

図 4.1　お出かけ用の EC19C の天板。我ながらステッカーのチョイスに節操がない。

外出先で PC を使う機会はそれほどないのだが、いざ持ち運べる外出用のノート PC がないとなると、それはそれで不便だ。そんなこともあり、持ち運び用のノート PC をなんとなく物色しはじめたのであったのだが...

4.2　ノート PC 不毛の時代

前回の「ざっぱ〜ん」では「少し待つとステキなハードウェアが出てきそう」という話題になったのだが、蓋をあけてみてそこにあったのはノート PC 不毛時代だった。

最近 Intel が攻勢をかけている低価格 CPU/SoC を搭載した製品は、どちらかというと拡張性の低い Chromebook が中心でスペック的にぐっとこないものが多く、モニタの解像度はのきなみ 1366x768(WXGA) をひきづっている残念な状況。

テレビでは 4K だ、8K だとか宣伝しているし、スマホやタブレットでもあんなに小さいのにフル HD は当たり前なのに、なんとなくノート PC だけ進化が止まっている感が否めない。

金にいとめをつけなければ Retina ディスプレイを搭載した MacBook や、フル HD 解像度のノート PC 製品があるといえばあるのだが、11 インチ級のコンパクトな製品で安価なものはなかなか存在しないし、Windows [*2] や MS Office なんて余分なものもついてくる。なによりノート PC1 台に 20 万円はちょっと厳しいし、ぶっちゃけちょっとケチってその分新しいお洋服を買いたい [*3] のが

[*1] 当時「3 万円台」でノート PC が買えるということで無邪気に買って CPU 交換までしてしまった魔改造ノート PC。改造の様子は blog 記事を参照してほしい。http://chonan.blog.pid0.org/2013/02/lenovo-g580.html

ホンネだ。

4.3 魅惑の中古 PC

そんなこんなでグダグダしつつ、漬物石のような魔改造 G580 を持ち運ぶ日々だったのだが、とある筋より「廃棄 PC をまとめたが興味があったら中身を消すことを前提に持って帰って良い」という非常にありがたい申し出を受け、見込みのありそうなところをチョイスして譲り受けることになった。[*4]

譲り受けたのは本命の Lenovo ThinkPad X1 12862HJ と、ある意味イロモノの NEC Lavie LL730/TG の 2 台。中古 PC の再生というとイマイチ手垢がついた話題かもしれないが、実際に中古 PC を発掘する際の参考にしてほしい。

4.4 Lenovo ThinkPad X1 12862HJ

まずは今回の本命の Lenovo ThinkPad X1。もらってきた時には、キーボードの Enter キーが壊れて外れていて、トラックポイントも劣化していたので、まずは交換用のキーボードの部品を手配した。Thinkpad ブランドはこのような保守部品もキチンと販売してくれるはずだったのだが、正規ルートでは手に入れることができず、修理に出そうにも修理不可という回答でお話にならない状況だった。少し検索してみると、中国系の業者が販売しているようなので、まずはキーボードを購入。[*5] さらにメモリとストレージも強化したいところなので、8GB のメモリユニットと 250GB の SSD ドライブも確保。

[*2] 実質無償でプレインストールされることが多い Windows 8.1 with Bing は Ubuntu ユーザにとっては価格的なインパクトがないのでむしろ歓迎だ。

[*3] 特にフリルとレースがたくさん実装されているものが個人的に好みだ。

[*4] 昨今は情報セキュリティの観点から PC を廃棄するのにも細心の注意が必要なので、同じような機会に恵まれた場合には細心の注意を払って取り扱う必要がある。

Chap. 5

かよちんとボクと、時々、録画

kazken3(@kazken3)

A「かよちんがすきだ...」
kazken3「そうか」
A「かよちんは清い...」
kazken3「そうか」
A「かよちんとずっとはなしていたい」
kazken3「そうか」
A「どうしたらいい？」
kazken3「... つくるんか...」
そして私は、かよちんを作成することにしたのであった...

5.1 かよちんを作ろう！

さて、かよちんを作るにあたってどういった機能が必要か考えてみた。

- おしゃべり
- リモート録画

まあ、おしゃべりは必須として、もともと Chinachu を運用していたので、リモートでの録画をやってみたかったところもあります[*1]。とりあえずはこんな感じで進めていきましょう。

5.2 なにでかよちんを作る？

では何でかよちんを作るか。今回はこの辺りでやってみることにしました。

- Raspberry Pi2
- Ubuntu 14.04
- Hubot
- Slack

[*1] おそらく自動録画を標榜する Chinachu の思想とは異なるとは思いますが...

今回のハードは Raspberry Pi2。今回やりたいことは Raspberry Pi2 でなくてもできますが、ARM 上でも Node.js は動くかな？ というところは興味深かったためと、そろそろちゃんと Raspberry Pi2 を使ってやろうという目論見です。[*2]

Raspberry Pi2 に Ubuntu を導入する流れは gihyo さんの Ubuntu Weekly Recipe あたりを確認すれば幸せになるでしょう。ただし、リモートの作業になる場合は openssh-server の導入をお忘れなく。

5.3 かーよちん、Node.js インストールしよーっ

まずは Hubot を動かすために Node.js を導入します。Node.js 自体は Ubuntu では apt でもインストールできますが、更新が早い Node.js の都合上、バージョンがどうしても古くなる。と、いうことで今回は Node.js のバージョンマネージャーである nvm を導入し、Node.js のバージョン管理を行うことにします。まずは下準備で以下のパッケージを導入します。

```
$ sudo apt-get install git build-essential curl
```

それでは、nvm をインストールしましょう。以下のコマンドを実行します。

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.25.4/install.sh | bash
```

エラーが出てなければ、nvm のインストールは成功しています。.bashrc にも nvm のロード処理が追加されていますので、source コマンドで再読込します。その後 nvm コマンドを実行してヘルプメッセージが出るのを確認して下さい。

```
$ source ~/.bashrc
$ nvm
```

nvm コマンドで **ls-remote** オプションをつけると、インストール可能な Node.js のバージョン[*3]の一覧を出してくれます。

```
$ nvm ls-remote
        :
       v0.12.7
```

[*2] BeagleBone Black もそろそろ動かしたいのですがね...
[*3] と、io.js のバージョン

この原稿時点では[*4]v0.12.7 が最新のようですね。では **nvm install** で v0.12.7 の Node.js をインストールしましょう。Raspberry pi2 上でソースからビルドを行うため、そこそこ時間がかかります。この辺りでコーヒーブレイクもよいでしょう。[*5]

```
$ nvm install v0.12.7
        :
```

Node.js のビルドとインストールが終わったら、**nvm use** バージョン名で利用する Node.js バージョンを固定し、**node -v** で Node.js がインストールできているかをバージョンで確認しましょう。また、nvm use で指定した Node.js はログアウトしてしまうと忘れてしまうため、**~/.bashrc** に登録しておくのがよいでしょう。

```
$ nvm use v0.12.7
Now using node v0.12.7 (npm v2.11.3)
$ node -v
v0.12.7
```

v0.12.7 と出れば、Node.js のインストールは成功です。次は Hubot ですね。

5.4　はなよ、Hubot をインストールするわよ

Hubot は GitHub が作成したロボット (bot) テンプレートで、Node.js 上動作する CoffeeScript を利用しています。CoffeeScript は実行時に JavaScript に変換し動作する言語です。

それでは Hubot をインストールしましょう。先ほどインストールした Node.js の npm コマンドを利用して yo と generator-hubot を導入します。

```
$ npm install -g yo generator-hubot
```

インストールが終わったら、Hubot をインストールするディレクトリを作成し、そのディレクトリに移動してから **yo hubot** コマンドを実行し、Hubot を作成します。

```
$ mkdir kayochin
$ cd kayochin
$ yo hubot
```

[*4] 2015/07/11 時点
[*5] 筆者はこの原稿を書きますね...

Chap. 6

磁気センサーを使った冷蔵庫監視システムの構築

水野源

全国 1 億 2 千万の中のほんのちょっぴりの、ざっぱーん読者のみなさんこんにちは! 会社の同僚が超音波距離センサーをごそごそいじっているのを見て興味を持ち、ついうっかり Arduino に手を出してしまった水野です[*1]。どうでもいいことですが、2015 年の 10 大がっかりには「ハルロック」の連載終了を入れておきたいと思っています。

6.1 はじめに

筆者は今までこういった電子工作、ハードウェア方面はまったくノータッチだったのですが、わからないなりに触ってみると面白いものですね。先日 Ubuntu Weekly Recipe で、AquesTalk Pi を使って合成音声をリアルタイムに再生させてみましたが、ソフトウェアが現実世界の何かに影響を与えるというのは、思った以上にワクワクすることを改めて実感しています。

先日、札幌市民にはおなじみ梅澤無線の店内をうろうろしていると、磁気センサーモジュールというものを見つけました。同僚に借りていじった超音波距離センサーは、出力端子が HIGH になっている時間で対象物までの距離を知ることができるというシロモノでした。こいつは名前からして、きっと磁気を検知すると出力端子が HIGH になるとか、そんな感じなんでしょう[*2]。それになにより安い (500 円くらい) ので、仮に失敗したとしても、松屋でネギ抜きと言い忘れて鉄皿チキングリルセットを頼んでしまったと思えば諦めがつくというものです[*3]。

(磁石をつけた) 物体の接近、離反といえば、身近な応用例としては、ドアの開閉センサーなどが簡単で実用的っぽいですね。それでは実際に、冷蔵庫のドアの開閉を監視、モニターするシステムを作ってみたいと思います。

[*1] ちなみにこの時点での筆者の知識は「この線どこに繋ぐの？ GND？ なにそれおいしいの？」レベルでした。

[*2] 本体にはマニュアルの類は付属しておらず、オンラインから PDF をダウンロードする仕様になっています。まるで豆腐でも売るが如く、ビニール袋にぽいっと詰められて売られている割り切りっぷりが素敵です。

[*3] いきなり弱気。

6.2　磁気センサー回路を作成する

使用したセンサーは、サンハヤトの磁気センサーモジュール、MM-DE21D です。こいつの接続端子は 2.54mm ピッチのスルーホールになっていますので、まずここに L 字の丸ピン IC ソケットをハンダづけして、ジャンパ線でブレッドボードに接続しやすいようにしました。

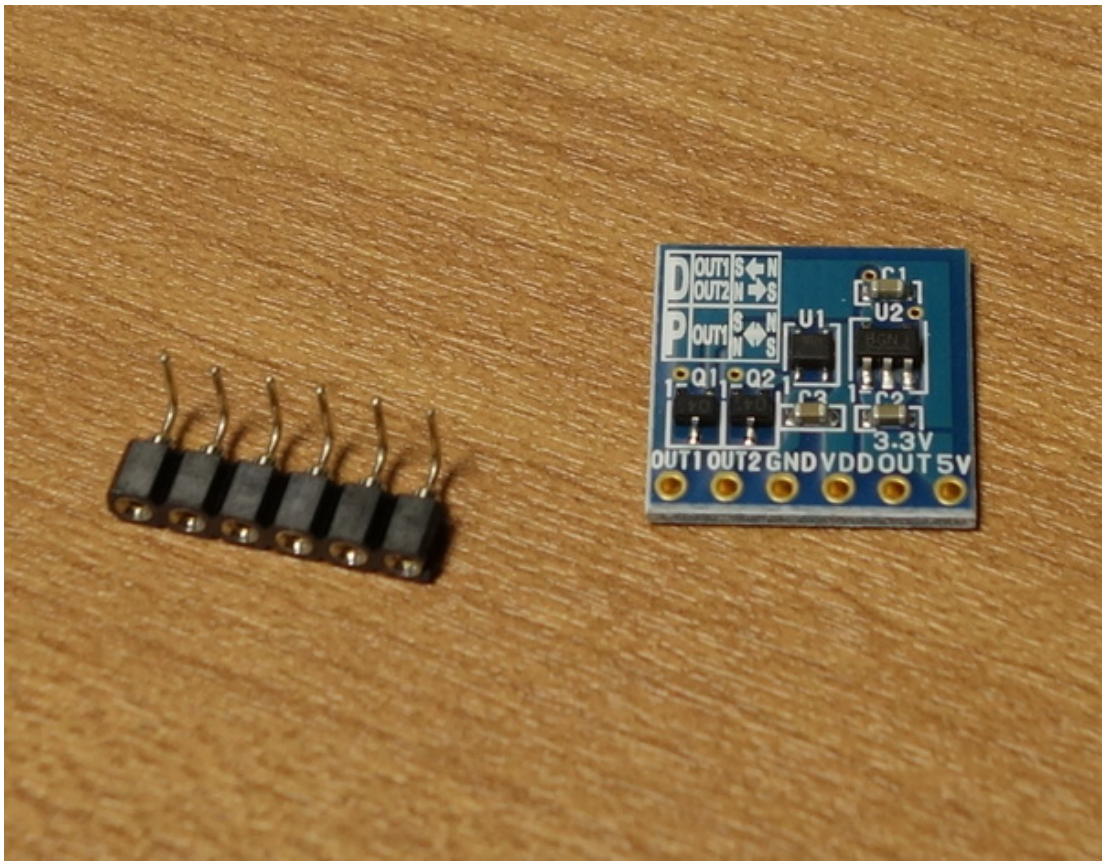

図 6.1　センサーとピンソケット

MM-DE21D(以下センサー) には、VDD と 5V のふたつの入力端子があります。電源電圧が 1.6〜3.6V の場合は VDD 端子に電源を直接接続すれば OK です。もしも電源電圧が 5V の場合は、5V 端子に電源を接続した後、センサーの 3.3V OUT 端子と VDD 端子を接続します。筆者は Arduino Leonardo を使っているのですが、こいつには 3.3V の電源ピンがありますので、これを VDD と直接接続します。そしてセンサーの GND 端子を Arduino の GND に繋いで回路を作ります。

出力端子は OUT1/OUT2 のふたつです。どちらも磁気を検知すると HIGH レベルになるという点では同じですが、磁気 (S/N) の向きによって、HIGH になる端子が変わる仕様です。つまり磁気の向きによって異なる動作をする回路を作ることができるわけですね。今回は OUT1/2 をそれぞれ、Arduino のデジタルピンの 8 番と 9 番に接続しました。また 5k Ωの抵抗を挟んで、3.3V とも

繋いでおきます[*4]。

デバッグ的な意味も兼ねて、センサーが反応していることが視覚的にわかるようにしたいと考えました。そこでデジタルピンの 10 番に青色 LED を繋ぎます。抵抗値を計算するのも面倒ですし、LED は非常に小さい電流でも点灯するので、カソード側には適当に 1k Ωの抵抗を挟んでおきます。あとはセンサーが反応している間、10 番ピンを HIGH にするようなプログラムを組めばよいわけです。L チカを卒業した人であればベイビー・サブミッションですね。ジャンパ線の長さという物理的制約に悩みつつ、ブレッドボードにプスプスと線や LED を挿せば回路は完成です。

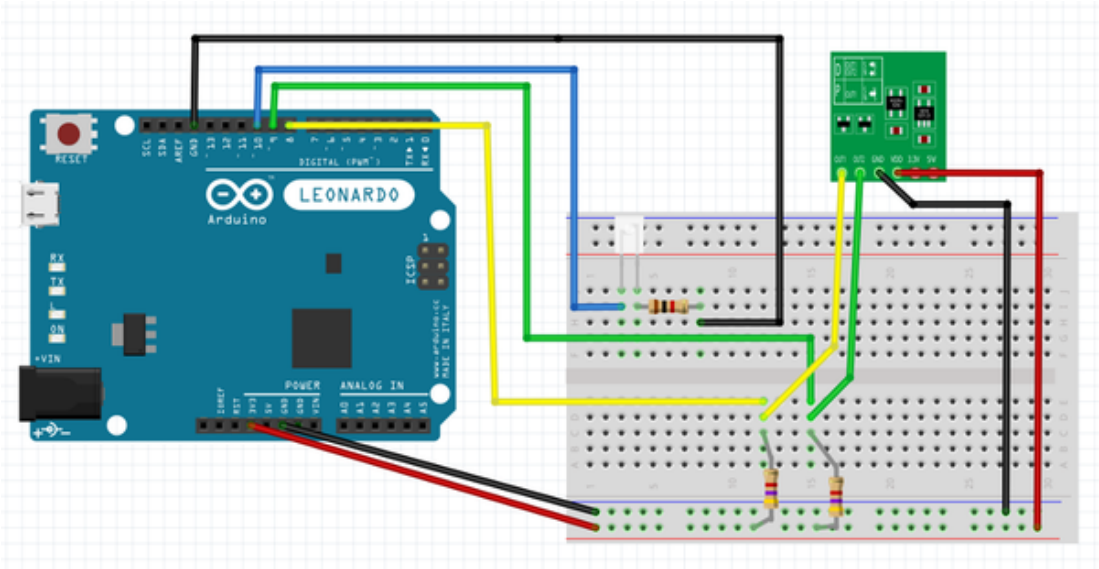

図 6.2　Arduino とセンサーの結線

[*4] なぜここに抵抗を挟んだ上で電源と繋がなければいけないのかはまったく理解できないのですが、マニュアルにそう書いてあるので従うことにしました。

Chap. 7

新し目の Mozc をビルドする

あわしろいくや

今回のテーマであるところの「ハードウェア」を決めたのは、言うまでもなく筆者なわけです。が、それを筆者自らぶっちぎるのは、いつものことなのか、それとも笑うところなのか。

7.1 動機

Mozc はバージョン 1.15.1857.102 以降 tar ボールでのリリースをやめています。これは確か Google Code でリリースファイルを置けなくなったからだと記憶していますが、その Google Code がサービスの終了を案内し、リリースファイルを置ける GitHub に移行してもなお tar ボールでのリリースは行っていません。もちろん執筆段階でのことであり、先のことはわかりませんが。

その代わりに、Google Code 時代は Wiki に、GitHub 時代になってからドキュメントでリリース履歴をお知らせするようになっています[*1]。

今回は、執筆段階での最新版であるところの 2.17.2095.102 をビルドしてみます。master はもっと新しくなってますけど、今回は対象としません[*2]。

7.2 事前知識

最も簡単に Mozc をビルドする方法は、Docker を使用することです。そのためのドキュメントも公開されています[*3]。

ただし確かにビルドは簡単なものの、ビルドしたものをどうすればいいのかはさっぱりわかりません。やっぱり普通にパッケージにしたいところです。もうひとつ問題があって、オフィシャルの方法だと fcitx-mozc も uim-mozc もビルドできません。まぁこれは Docker ファイルに手を入れればなんとかならないこともないですけど。

とはいえ、パッケージの作成に必要なフォルダーやファイルは Mozc のリポジトリには存在しないため、Debian/Ubuntu 用のソースパッケージを取得する必要があります。また、fcitx-mozc も

[*1] https://github.com/google/mozc/blob/master/doc/release_history.md

[*2] 執筆段階で master であるところの 2.17.2102.102 用の fcitx-mozc はリリースされているので、ビルドできるはずです。というか、ぶっちゃけ執筆完了後にリリースされました

[*3] https://github.com/google/mozc/blob/master/doc/build_mozc_in_docker.md

アップデートする必要があり、やはりこれも取得します。

ビルド環境の準備の仕方は特に書かないので、適宜用意してください。実機、仮想マシン、lxc、pbuilder、何でもいいと思います。今回は試しに lxc を使用してみましたが、さしたる問題もなくビルドできました。ただし、ビルドする Ubuntu のバージョンは 14.04 とします。

7.3 準備

まずは Ubuntu のリポジトリからビルドに必要なパッケージと Mozc のソースパッケージを取得します。次のコマンドを実行してください。

```
$ mkdir mozc
$ cd mozc
$ sudo apt-get install ubuntu-dev-tools git subversion ninja-build \
      clang libdistro-info-perl
$ sudo apt-get build-dep mozc
$ pull-lp-source -m http://jp.archive.ubuntu.com/ubuntu mozc vivid
```

続いて、fcitx-mozc の更新版と Mozc のソースコード取得に必要なツールをダウンロードし、パスを通します。

```
$ wget http://download.fcitx-im.org/fcitx-mozc/fcitx-mozc-2.16.2037.102.2.patch
$ git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
$ PATH=$PATH:$(pwd)/depot_tools
```

なお、fcitx-mozc のソースコードの最新版はダウンロードサイト[*4]を確認してください。もしビルド中に fcitx-mozc 由来によるビルドエラーが出た場合は、fcitx-mozc が最新の Mozc に追随していない可能性があるので、バグ報告[*5]してください。

ようやく今回ビルドする Mozc のソースコードを取得します。

```
$ mkdir mozc-git
$ cd mozc-git/
$ gclient config https://github.com/google/mozc.git --name=. --deps-file=src/DEPS
$ gclient sync --revision=321e0656b0f2e233ab1c164bd86c58568c9e92f2
```

gclient sync のリビジョンは、前述のリリース履歴に書かれています。

パッケージにするためにはどうしても tar ボールが必要なため、生成します。

[*4] http://download.fcitx-im.org/

[*5] とはいえどこにバグ報告すればいいんだろう……？ https://github.com/fcitx/mozc は Pull Request はできますけど、issue は書けないですね。

うぶんちゅ！　まがじん ざっぱ〜ん♪ **vol.3** 体験版

著　者　うぶんちゅ！　まがじん ざっぱ〜ん♪ の仲間たち
発行所　うぶんちゅ！　まがじん ざっぱ〜ん♪ 編集部

(C) 2015 Ubunchu! Magazine Zapppaaan