# .Deb

銀河系唯一のDebian専門誌

2013 年 7 月 20 日

**特集 1：** Debian armmp / Raspberry Pi

**特集 2：** 月刊Debhelper: dh_strip

# 1 Introduction

上川 純一

今月の Debian 勉強会へようこそ。これから Debian の世界にあしを踏み入れるという方も、すでにどっぷりとつかっているという方も、月に一回 Debian について語りませんか?

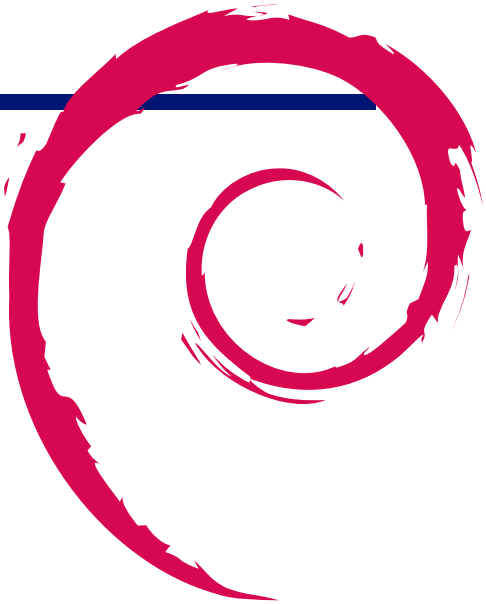
Debian 勉強会の目的は下記です。

- Debian Developer (開発者) の育成。
- 日本語での「開発に関する情報」を整理してまとめ、アップデートする。
- 場の提供。
  - 普段ばらばらな場所にいる人々が face-to-face で出会える場を提供する。
  - Debian のためになることを語る場を提供する。
  - Debian について語る場を提供する。

Debian の勉強会ということで究極的には参加者全員が Debian Package をがりがりと作るスーパーハッカーになった姿を妄想しています。情報の共有・活用を通して Debian の今後の能動的な展開への土台として、「場」としての空間を提供するのが目的です。

# デビアン勉強会

## 目次



# 2 事前課題

上川 純一

今回の事前課題は以下です:

1. お使いのマシンに ARM はありますか? もしあるのでしたら、どのように使っているか教えてください。
2. Debian の ARM に期待していること、お願いしたいことがあれば教えてください。

この課題に対して提出いただいた内容は以下です。

## 2.1 koedoyoshida

1. お使いのマシンに ARM はありますか? もしあるのでしたら、どのように使っているか教えてください。
   Raspberry Pi を使ってます。とりあえずリモートアクセス用がメインですが、 I2C とかを使って外部 IO をやってみたいと思っています。
2. Debian の ARM に期待していること、お願いしたいことがあれば教えてください。
   省電力な環境を生かして ReadOnlyboot(でも定期的にセキュリティ update はしたい) で放置できる環境とかが作れるとよいですね。

## 2.2 吉野

1. 地図マシンです
2. なし

## 2.3 dictoss

1. CuBox を持っている。 eSATA 端子があるのでファイルサーバのバックアップをするマシンとして使おうとしたが、 USB ポートから電源供給量が不足のため、 HDD が動かず使えていない。そのため単なる armel バイナリのお試しマシンになっている。
2. 新 Nexus7 で Debian が動いてほしい。

## 2.4 mtoshi.g

1. お使いのマシンに ARM はありますか?
   ないっす
2. Debian の ARM に期待していること、お願いしたいことがあれば教えてください。
   今のところないっす

## 2.5 野島 貴英

1. 昔売っていた B&N の NookColor を ARM 機材実験ボードとして未だに使ってます。この電子書籍端末は 2012/4 の東京エリア debian 勉強会,2012/6 の大統一 debian 勉強会で喋ったとおり、特定ブートフォーマットの SD カードを用意して SD カードスロットへ入れちゃうと、うっかりそのままブートしちゃうという隠れ (?) 機能が大変便利です。おまけに連続 8 回起動失敗するとリカバリーが始まるという非文鎮化機能まで搭載しています。そのままでも pdf リーダ、ポータブル mp4 ビデオ/mp3 鑑賞機材として、また、 debian ARM の native ブートの可能性を秘めた hack 機材としても楽しくお使いいただけます。
2. NookColor 用の debian ネイティブブートが可能なイメージ( というかインストーラ) が欲しいといってみるテスト。

## 2.6 岩松 信洋

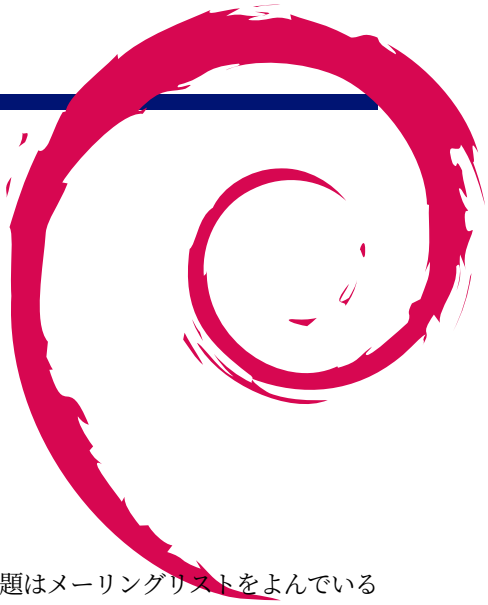
1. お使いのマシンに ARM はありますか? もしあるのでしたら、どのように使っているか教えてください。
   たくさんある。開発用とかビルドマシンとして使っています。 Raspberry Pi は家のゲートウェイマシンとして動いています。
2. Debian の ARM に期待していること、お願いしたいことがあれば教えてください。
   マルチメディア系が弱いので整備して欲しい。

## 2.7 上川純一

なし

## 2.8 まえだこうへい

1. Armadillo J で自宅内の DHCP サーバとして使っています (not Debian)。 OpenBlockS AX3 を、昨年夏ごろにカッとなって作った iori というツール (最近話題の docker みたいなの) の開発環境として使っています。
2. 今は特にないですが、今後 ARM サーバの製品版が出てきた時に d-i でインストールできることでしょうか。



# 3 Debian Trivia Quiz

上川純一

ところで、みなさん Debian 関連の話題においついていますか? Debian 関連の話題はメーリングリストをよんでいると追跡できます。ただよんでいるだけではりあいがないので、理解度のテストをします。特に一人だけでは意味がわからないところもあるかも知れません。みんなで一緒に読んでみましょう。

今回の出題範囲は `debian-devel-announce@lists.debian.org` や `debian-devel@lists.debian.org` に投稿された内容と Debian Project News からです。

問題 1. Stefano Zacchiroli さんが新しく作成したサービスは?

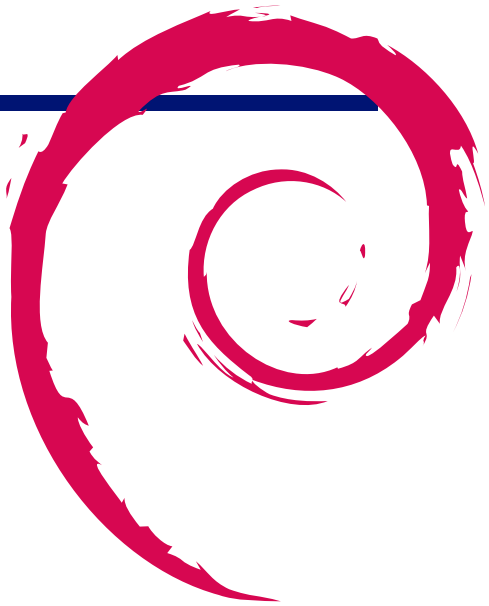
☐ A chiebukuro.debian.net
☐ B 2ch.debian.net
☐ C sources.debian.net

問題 2. 新しく FTP master チームに入った人は?

☐ A Gergely Nagy
☐ B Kouhei Maeda
☐ C Joerg Jaspert

問題 3. Debian GNU/Hurd がリリースされましたが、バージョンはいくつでしょう。

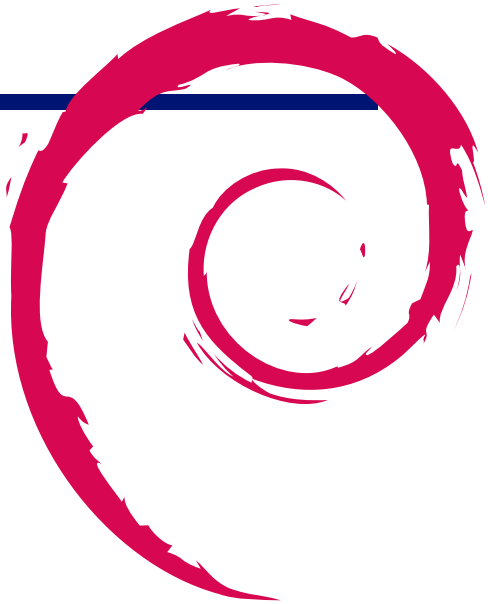
☐ A 2013
☐ B 3.141592
☐ C 7.0



# 4 最近の Debian 関連のミーティング報告

上川純一

## 4.1 東京エリア Debian 勉強会 100 回目報告



# 5 Debian linux kernel / armmp フレーバ

岩松 信洋

## 5.1 はじめに

Debian linux kernel の 3.9 から armhf アーキテクチャに armmp (ARM Multi Platform) フレーバが入りました。本稿では armmp の仕組みと対応方法について説明します。

## 5.2 arm multi platform とその仕組み

Linux kernel バージョン 3.8 ぐらいから arm multi platform（ 以下、 ARMMP） サポートが入りました。これは一つの linux kernel バイナリで複数の ARM Soc、ターゲットボードをサポートする仕組みです。今のところ、 mvebu（ Marvell SoCs）を筆頭に Versatile Express、 OMAP などがサポートされています。バージョン 3.11 ではほとんどの ARM SoC がサポートされる予定になっています。

複数 SoC やターゲットボードへの対応方法ですが、これはカーネル起動時に Device Tree (以下、 DT）と呼ばれるハードウェア情報を渡すことによって、カーネルを各デバイス向けに初期化し動作するようになります。

### 5.2.1 ARMMP カーネルの起動方法

通常 Linux カーネルを起動する場合、 zImage だけで起動しますが、 ARMMP や最近の ARM 向けカーネルは DT が必要になっています。 DT を絡めたカーネル起動方法は 3 つほどあります。起動方法別に Linux 上での操作( linux$が linux 上での操作）と U-Boot [*1] 上での操作( u-boot$は U-boot 上での操作）を以下に紹介します。

1. zImage に dtb (DT blob。 DT のバイナリ) を付加する。そしてその zImage で起動する。
   まずカーネルの CONFIG_ARM_APPENDED_DTB が必要です。有効になっていない場合は有効にしましょう。
   そしてビルドされた zImage と dtb を結合します。

   ```
   linux$ cat arch/arm/boot/zImage arch/arm/boot/dts/armada-xp-openblocks-ax3-4.dtb  > arch/arm/boot/zImage+dtb
   ```

   上で作成した zImage+dtb をターゲット機器上にコピーします。以下の例は tftp を使って 0x2000000 にコピーしています。 0x2000000 はターゲットによって異なるので環境に合わせて変更してください。そして go コマンドにロードしたアドレスを指定して実行し、カーネルを起動させています。

   ```
   u-boot$ tftpboot 2000000 zImage+dtb
   u-boot$ go 2000000
   ```

2. 1. で作成した zImage を uImage に変換して起動する。
   以下は 結合したイメージを uImage 形式に変換しています。最初の `make uImage` では zImage を uImage に変

[*1] U-Boot は ARM 組込機器でよく利用されるブートローダ

換していますが、その後 zImage と dtb を結合( zImage+dtb）し、 arch/arm/boot/.uImage.cmd を使って結合したイメージを uImage （ uImage+dtb）に変換しています。*2

```
linux$ make uImage
linux$ cat arch/arm/boot/zImage arch/arm/boot/dts/armada-xp-openblocks-ax3-4.dtb  > arch/arm/boot/zImage+dtb
linux$ ‘cut -f 3- -d ’ ’ < arch/arm/boot/.uImage.cmd | sed -e ’s/zImage/zImage+dtb/g’ -e ’s/uImage/uImage+dtb/g’‘
```

起動に利用するフォーマットが uImage 形式の場合、 go コマンドではなく bootm コマンドのを使います。

```
u-boot$ tftpboot 2000000 uImage+dtb
u-boot$ bootm
```

これらの方法は ARMMP ではない SoC 単体で DT を使う場合のみ有効です。カーネルがロードされるアドレスなどは SoC やボード毎に異なるのですが、 SoC 単体ではカーネルのコンフィグ情報としてこれらを持っているので上記の説明で uImage が作成できます。しかし ARMMP の場合は SoC 毎にこれらの値が異なるのでターゲット SoC 毎にこれらの値を変更する必要があります。 ARMMP で実行すると以下のようなエラーになるでしょう。

```
linux$  ‘cut -f 3- -d ’ ’ < arch/arm/boot/.uImage.cmd | sed -e ’s/zImage/zImage+dtb/g’ -e ’s/uImage/uImage+dtb/g’‘
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: ‘include/generated/mach-types.h’ is up to date.
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/zImage is ready
multiple (or no) load addresses:
This is incompatible with uImages
Specify LOADADDR on the commandline to build an uImage
make[1]: *** [arch/arm/boot/uImage] Error 1
make: *** [uImage] Error 2
```

ロードアドレスなどの情報は mkimage *3で指定できます。*4
-a でロードアドレス、-e でエントリポイントアドレスを指定します。

```
linux$ cat zImage foo.dtb > zImage+dtb
linux$ mkimage -A arm -O linux -T kernel -C none -a 0x2000000 -e 0x2000040 -n ’Linux-marvell’ -d arch/arm/boot/zImage+dtb arch/arm/boot/
```

*5

3. uImage と dtb blob 別に読み込んで起動する。
   ARMMP は 一つのバイナリで複数の ARM SoC、ボードをサポートすることが目的なので、上記の方法ではサポートできません。よって、 uImage と dtb blob を分けて起動させるのが理想です。 U-Boot の場合は カーネルを起動するコマンド bootm を使います。

```
linux$ mkimage -A arm -O linux -T kernel -C none -a 0x2000000 -e 0x2000040 -n ’Linux-marvell’ -d arch/arm/boot/zImage arch/arm/boot/uIma
```

```
u-boot$ tftpboot 2000000 uImage
u-boot$ tftpboot 3000000 dtb
u-boot$ bootm 2000000 - 3000000
```

bootm コマンドの 第 1 引数は uImage がロードされているアドレス、第 2 引数は uInitrd （ initrd イメージの uImage）がロードされているアドレス、第 3 引数には dtb がロードされているアドレスを指定します。-は 指定なしを意味します。
ちなみに dtb の格納されてるアドレスはブートローダによってカーネル起動時の r2 レジスタに設定され、カーネルが指定されているアドレスからデータを読み込み起動する用になっています。これは Linux カーネルの Documentation/arm/Booting に書かれています。

*2 後述する mkimage を直接呼んでもよい
*3 U-Boot 用のイメージを作成するツール。
*4 arch/arm/boot/.uImage.cmd 内では mkimage を呼んでいる
*5 カーネル開発者は./scripts/mkuboot.sh を使う事が多いかも。

### 5.2.2 カーネルモジュールの対応

カーネルモジュールも DT によって設定できます。必要なカーネルドライバを組み込みにしてカーネルを起動させるのもよいのですが、 ARMMP の場合カーネルが肥大化しますので、 SoC のコア部分は最低限の機能は組み込みに設定し、各デバイス用ドライバはモジュールにして initrd などからロードするようにするのがよいでしょう。 Debian の場合はサポートボード毎に zImage を用意せず、上記の方法で対応しています。

## 5.3 Debian でのサポート

Debian の armhf アーキテクチャの armmp フレーバ は上記で説明した機能を持ったカーネルをサポートするためのものです。今回 Plat'Home さん[*6] の Openblocks AX3 を Debian でサポートするために実装しました。 AX3 は `mvebu` という Marvell 製 SoC をまとめている SoC アーキテクチャとしてサポートされているのですが、これは ARMMP サポートのみで実装されているので、 Debian 側でも ARMMP サポートする必要がありました。また、他の今後 ARM カーネルは ARMMP に移行することが決定していたので、誰かやる必要があったというのも理由の一つです。

## 5.4 Debian でのカーネル提供と利用方法

Debian はカーネルイメージを uImage で提供していません。 uImage で提供した場合、カーネルがロードされるアドレスが固定値になってしまうので、複数のデバイスをサポートできません。 Debian では カーネルを zImage(vmlinuz) として提供しています。

しかしこのままでは U-Boot を使ったデバイスなどで使う場合に手間がかかるので、 Debian では flash-kernel[*7] パッケージでサポートしています。

flash-kernel は設定されたデータをもとにカーネルや initrd を U-Boot などで扱える形式に変換し、フラッシュメモリに書き込む機能等を提供します。

データの形式は以下のようになります。このデータは`/usr/share/flash-kernel/db/all.db` に記述されます。

```
Machine: Marvell Armada 370/XP (Device Tree)
Kernel-Flavors: armmp
DTB-Id: armada-xp-openblocks-ax3-4.dtb
DTB-Append: yes
U-Boot-Kernel-Address: 0x2000000
U-Boot-Initrd-Address: 0x0
Boot-Device: /dev/sda1
Boot-Kernel-Path: /boot/uImage
Boot-Initrd-Path: /boot/uInitrd
Required-Packages: u-boot-tools
Bootloader-Sets-root: no
```

flash-kernel を実行すると設定されているデータと起動しているカーネル情報( /proc/cpuinfo、または/proc/device-tree/model）を元にカーネルと inittd を変換し、インストールします。

```
$ cat /proc/cpuinfo | tail -3
Hardware        : Marvell Armada 370/XP (Device Tree)
Revision        : 0000
Serial          : 0000000000000000

$ flash-kernel
flash-kernel: installing version 3.10-1-armmp
Generating kernel u-boot image... done.
Installing new uImage.
Generating initramfs u-boot image... done.
Installing new uInitrd.
Installing new dtb.

$ ls /boot
System.map-3.10-1-armmp  initrd.img-3.10-1-armmp  uInitrd
config-3.10-1-armmp      uImage                    vmlinuz-3.10-1-armmp
```

flash-kernel によって作成されたイメージを使った起動方法はボード毎に異なります。 OpenBlocks AX3 の場合は以下

[*6] http://www.plathome.co.jp/

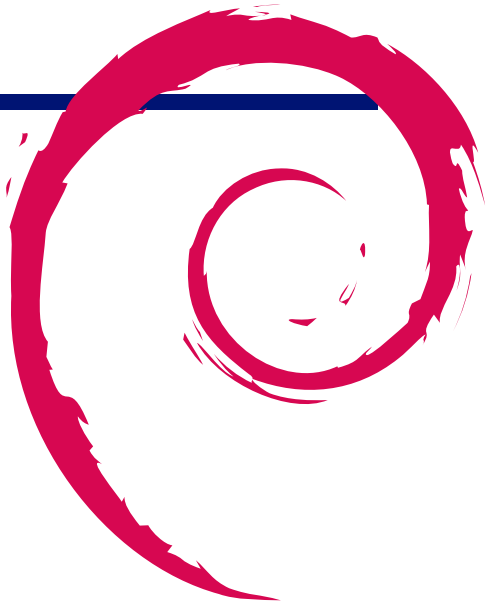
[*7] http://packages.qa.debian.org/f/flash-kernel.html

の方法で起動できるはずです。

```
$ ide reset
$ ext2load ide 0 2000000 /boot/uImage
$ ext2load ide 0 3000000 /boot/uInitrd
$ bootm 2000000 3000000
```

## 5.5 終わりに

ARMMP の仕組みと、 Debian の対応方法について説明しました。 flash-kernel のデータはぷらっとホームさんと相談して入れようと思っているのでまだコミットされていません。数日後にはパッチが BTS に上がるでしょう。 OpenBlocks A6 もサポートします。また、エントリーポイントを設定する項目と DT の model を指定する項目がないのでこれも対応する必要があります。パッチはつくってあるので後日 BTS します。あと、残作業としてはカーネル 3.10 では mvebu が動作しない( 他でも同じかも）のでパッチを当てる必要があります( コミット: `faefd550c45d8d314e8f260f21565320355c947f`）。これも BTS します。ということでまだまだやることはたくさんあります。



# 6 月刊 Debhelper dh_strip

野島 貴英

## 6.1 今月のコマンド

今月は dh_strip を取り上げます。紙面の都合で dh_strip の 1 コマンドの紹介に今回はとどめておきます。

## 6.2 dh_strip

dh_strip コマンドは、“debian/tmp/” 等のビルドディレクトリ以下を再帰的に探し回り、ビルドしたばかりの実行ファイルのバイナリ、共有ライブラリのバイナリ、静的ライブラリのバイナリを見付けると、片っ端からバイナリに含まれているデバッグシンボルを取り除いてくれます。

また、別途コマンドラインオプションを指定すると、取り除いたデバッグシンボルを別ファイルに取り置きます。このため、dh_strip コマンドは*-dbg パッケージ作成の時に重要な役割を持ちます。

### 6.2.1 すでに発表済みの件

実は、dh_strip の*-dbg パッケージを作成する為の動作については、大統一 Debian 2012 のセッションである「debug.debian.net」[1] のプレゼン資料に詳しいので、こちらを参照してください。

また、*-dbg パッケージを使って実際にデバッグをしてみる件については、大統一 Debian 2013 のセッションである「gdb+python 拡張を使ったデバッグ手法」[2] に紹介していますので、こちらも参照ください。

ここでは、これらの発表で扱わなかった件を重点的に記載します。

### 6.2.2 debug/*.so,lib*_g.a ファイルの扱い

dh_strip コマンドは、元々デバッグ用途として用意されている debug/*.so ファイルや、lib*_g.a ファイルについては、何も処理を行いません。

### 6.2.3 コマンドラインオプションと動作

dh_strip コマンドは表 1 のコマンドラインオプションを解釈します。

### 6.2.4 COMPATIBILITY LEVEL での動作の違い

dh_strip はソースパッケージに含まれる debian/compat で指定される COMPATIBILITY LEVEL により、振る舞いが変わります。

–dbg-package オプション指定時の振る舞いの違いは表 2 に、また、-k/–dbg-package オプション指定時に生成されるデバッグシンボルファイルの違いについて表 3 に、記載します。

| オプション名 | 動作 |
|---|---|
| man debhelper 記載のオプション | こちらは全 debhelper に共通のオプションです。動作は man debhelper に記載されていたり、以前の月刊 debhelper で説明されているとおりとなりますので、ここでは割愛します。 |
| -Xitem, –exclude=item | item で指定されるファイル名を持つバイナリを処理の対象から外します。複数のファイルを指定したい場合は、繰り返し指定すると指定できます。例: dh_strip -Xfile1 -Xfile2 |
| –dbg-package=package | *-dbg パッケージを作る際に、デバッグシンボルを格納するパッケージ名を指定する為に利用します。通常、*-dbg パッケージを作る場合は、こちらのオプションを利用する事となります。なお、ソースパッケージにある debian/compat で指定される COMPATIBILITY LEVEL で振る舞いが異なります( 後述) 。 |
| -k | 分離したデバッグシンボルをバイナリパッケージに含める場合にこちらのオプションを利用します。この場合、出来上がったパッケージにはバイナリの他に、分離したデバッグシンボルも/usr/lib/debug/以下に含まれる形で収録されます。–dbg-package が同時に指定されると、–dbg-package の指定の方が優先されます。 |

表 1　dh_strip コマンドラインオプション

| COMPATIBILITY LEVEL | –dbg-package 指定時の動作 |
|---|---|
| 4 以下 | “dh_strip –dbg-package=xxxx –dbg-package=yyyy” のように–dbg-package を複数指定する事が出来ます。ここで、こちらで指定した名前 (xxxx や、 yyyy) に合致するパッケージを処理する際、” パッケージ名-dbg” というパッケージ名を*-dbg パッケージのパッケージ名として自動的に利用します。また、ソースパッケージに含まれる debian/control には、これら*-dbg パッケージの為の記述は不要となります。 |
| 5 以上 | –dbg-package は 1 つ指定するのが原則となります。もし複数指定した場合は、最初に指定された内容のみ利用されます。また、–dbg-package=xxxxx とすると、デバッグシンボルを格納するパッケージ名として、 xxxxx がそのまま使われます。例えば、 libfoo パッケージと、 foo パッケージを構築し、これらパッケージに含まれているバイナリのデバッグシンボルを foo-dbg パッケージに全部入れるには、–dbg-package=foo-dbg と指定します。また、ソースパッケージに含まれる debian/control には、生成予定の*-dbg パッケージの為の記述は必須となり、万一記載されていない場合はエラーとして扱われ、この場合は dh_strip はエラーを表示して終了します。 |

表 2　COMPATIBILITY LEVEL と–dbg-package オプションの振る舞いの違い

### 6.2.5　デバッグシンボル

dh_strip で取り分けられるデバッグシンボルは、 elf バイナリベースのシステム[*8]であれば、 DWARF[3] をベースとした形式のファイルとなります。

ここで DWARF は、コンパイラを用いて CPU の命令に直接変換するようなコンパイル言語( C/C++ 等) を用いて作成したバイナリを対象として、ソースレベルのデバッグを行う場合に必要な/便利な情報を格納する為に作られた、よくできたフォーマットです。元々は System V 用のデバッガである sdb の為に Bell 研で開発されたフォーマットがベースとのことです[*9]。 DWARF は、”Debugging With Attributed Record Formats” の略とのことです。

図 1 にバイナリと、デバッグシンボルの中身を簡単に図示します。

### 6.2.6　デバッグシンボルファイルの情報を覗いてみる

実はバイナリや、デバッグシンボルの中身を見る方法は知っておくと、いろいろデバッグ等で役立つ場面が多いです。以下に簡単に参照する方法を載せます。

[*8] お馴染みの i386/amd64 用 linux の場合等

[*9] Bell 研とか、 System V とか、今ではもう完全にオッサンホイホイな用語となってしまいました...

| COMPATIBILITY LEVEL | -k/–dbg-package 指定時のデバッグシンボルファイルの違い |
|---|---|
| 8 以下 | “/usr/lib/debug/+ バイナリのインストール先パス/+ バイナリファイル名” に格納されるようにデバッグシンボルファイルが生成されます。また、デバッグに関する情報はコンパイラが生成したままの形で保存されるため、デバッグシンボルファイルのサイズは大きくなりがちです。 |
| 9 以上 | バイナリが BuildID[2] を含んでいる場合は、 “/usr/lib/debug/.build-id/BuildID の上位 2 桁/+BuildID の残りの桁 +.debug” に格納されるようにデバッグシンボルファイルが生成されます。 BuildID を含んでいなければ COMPATIBILITY LEVEL 8 以下と同様のファイル名となります。また、 BuildID の有無にかかわらず、デバッグに関する情報は zlib により圧縮されて格納されます。そのため、デバッグシンボルファイルのサイズはできるだけ小さくなるようになっています。 |

表 3　COMPATIBILITY LEVEL と-k/–dbg-package オプションの振る舞いの違い

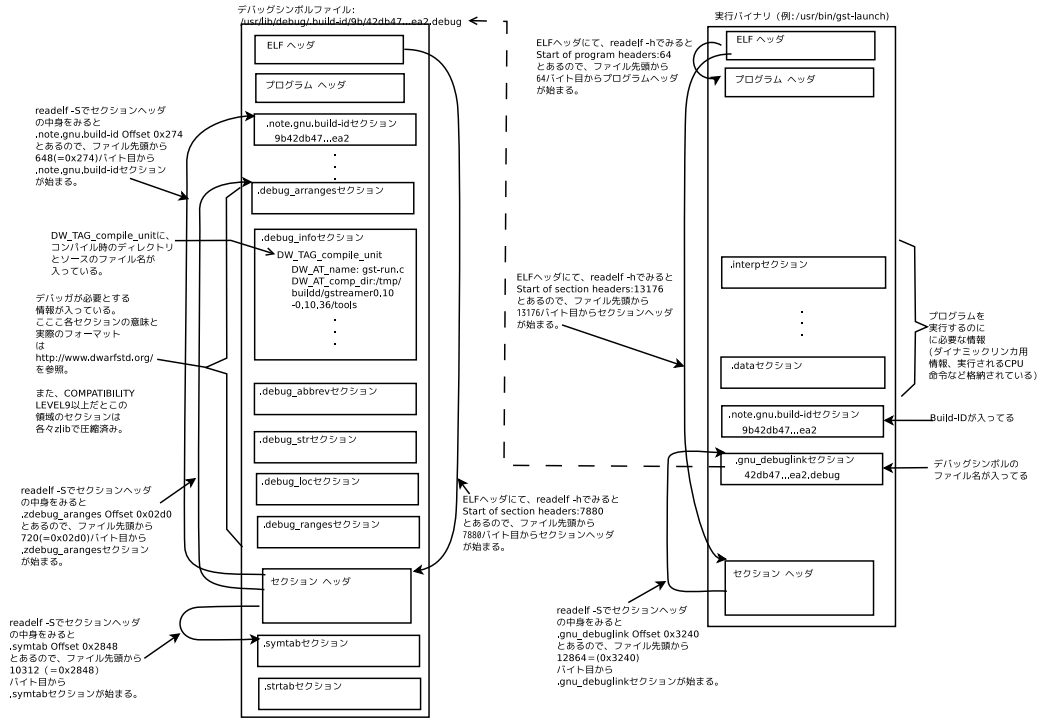



図 1　バイナリと、デバッグシンボルの中身

まず、バイナリにはどんなヘッダ情報とセクションが入っているかのサマリを掌握してみます。

```
$ sudo aptitude install binutils
$ readelf -e /usr/bin/gst-launch
ELF ヘッダ
   マジック:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
   ... 中略...
セクションヘッダ:
  [番] 名前              タイプ            アドレス          オフセット
       サイズ            EntSize           フラグ Link  情報  整列
  [ 0]                   NULL              0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS          0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
... 中略...
  [27] .gnu_debuglink    PROGBITS          0000000000000000  00003240
       0000000000000034  0000000000000000           0     0     1
  [28] .shstrtab         STRTAB            0000000000000000  00003274
       00000000000000fe  0000000000000000           0     0     1
... 中略...
```

次にどんなデバッグシンボルファイルが使われる予定かを見てみます。

```
$ readelf -x .gnu_debuglink /usr/bin/gst-launch
セクション '.gnu_debuglink' の 十六進数ダンプ:
  0x00000000 34326462 34373631 31386162 32623831 42db476118ab2b81
  0x00000010 30343161 63643830 65343565 38396534 041acd80e45e89e4
  0x00000020 32383965 61322e64 65627567 00000000 289ea2.debug....
  0x00000030 dbf8060d                             ....
$
```

デバッグシンボルのファイル名の形式から、 COMPATIBILITY LEVEL 9 のソースパッケージで構築されたデバッグシンボルファイルである事がわかります。このため、 BuildID から、/usr/lib/debug/.build-id/9b/42db476118ab2b81041acd80e45e89e4289ea2.debug がデバッグシンボルのファイルとなります。

次にデバッグシンボルファイルから、ビルド時のディレクトリ位置を求めてみます。

```
$ aptitutde install libgstreamer0.10-0-dbg
$ readelf -wi /usr/lib/debug/.build-id/9b/42db476118ab2b81041acd80e45e89e4289ea2.debug | lv
.debug_info セクションの内容:

  コンパイル単位 @ オフセット 0x0:
   長さ:        0x1b3b (32-bit)
   バージョン:    2
   Abbrev Offset: 0x0
   ポインタサイズ:8
 <0><b>: 省略番号: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (間接文字列、オフセット: 0x442): GNU C 4.7.2

    <10>   DW_AT_language    : 1        (ANSI C)
    <11>   DW_AT_name        : (間接文字列、オフセット: 0x57a): gst-run.c

    <15>   DW_AT_comp_dir    : (間接文字列、オフセット: 0x36b): /tmp/buildd/gstr
eamer0.10-0.10.36/tools
  ... 中略...
```

.debug_info セクションに定義されている、 DW_TAG_compile_unit の結果から、ディレクトリ/tmp/buildd/gstreamer0.10-0.10.36/tools に配置されている gst-run.c がコンパイルされている事がわかります。

では、この結果を元に、 gdb の set substitute-path を使って gdb でソースの閲覧をやってみます[*10]。

```
$ pwd
/home/yours/src/gstreamer/
$ apt-get source gstreamer0.10-tools/sid
$ gdb --args /usr/bin/gst-launch
... 中略...
Reading symbols from /usr/bin/gst-launch...Reading symbols from
/usr/lib/debug/.build-id/9b/42db476118ab2b81041acd80e45e89e
4289ea2.debug...done.
(gdb) set substitute-path /tmp/buildd/ /home/yours/src/gstreamer/
(gdb) b main
(gdb) run
(gdb) l
313         return candidates;
314       }
315
316       int
317       main (int argc, char **argv)
318       {
319         GHashTable *candidates;
320         gchar *dir;
```

無事ソース閲覧ができています。

## 6.3 おわりに

今回は、 dh_strip コマンドをネタに、デバッグシンボルファイルの中を覗き、その結果を元に最後は gdb でソース閲覧ができるようになるまでをやってみました。

また、 dh_strip の周辺技術を見てみると、ソースコードのデバッグができるようになる為に、とても多くの人の成果物を活用していることが実感できました。

[*10] なお、本方法は、 2013 年大統一 Debian 勉強会 [2] の当時では、自分がまだ見つけていなかった方法でして... これで gdb の dir コマンドを使わなくて済み、今後は*-dbg パッケージを導入するだけで便利にソースの閲覧できるようになります。

# 参考文献


[1] 2012 年大統一 Debian 勉強会「 debug.debian.net」, http://gum.debian.or.jp/2012/

[2] 2013 年大統一 Debian 勉強会「 gdb+python 拡張を使ったデバッグ手法」, http://gum.debian.or.jp/2013/slide_data_list

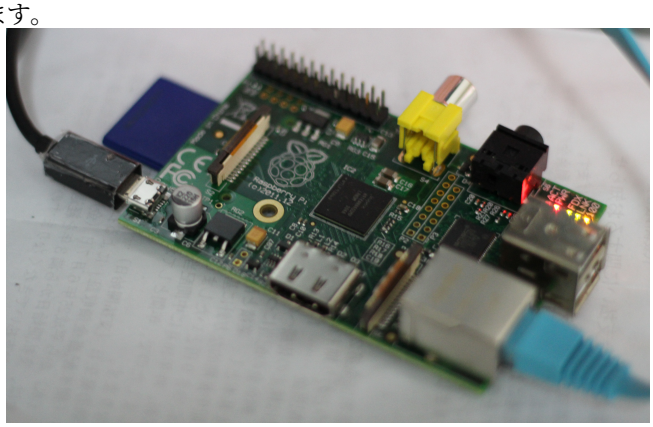
[3] The DWARF Debugging Standard, http://www.dwarfstd.org/Home.php

# 7 raspberry pi

上川 純一

## 7.1 はじめに

raspberry pi は市販されている安価な ARM のボードです。日本国内であれば RS Components から直販されています。



Debian が動くデバイスという観点でみると、 CPU として ARM1176JZF-S が搭載されているデバイスです。ARM11 (ARMv6) で、ハードウェアで浮動少数点演算ができる VFP が搭載されており、 ARMv7 以降の機能である NEON には対応していません。

HDMI 出力、イーサネット、 USB ホストなどがあるため、モニタとキーボードを接続すると普通のパソコンのように利用できるようになっています。電源は micro USB 端子で、 1A 以上が供給できるようになっている最近のスマートホン用の電源であれば流用できます。

## 7.2 Raspbian

Raspberry pi 用の Linux ディストリビューションの一つが Raspbian です。今回はそれを採用します。

### 7.2.1 Raspbian でなにがうれしいのか?

Debian そのものではなく Raspbian を利用する理由はなんでしょうか。インストール画面がカスタマイズされていますが、それ以外に何のメリットがあるのでしょうか。 Debian GNU/Linux の wheezy 時点の安定版でサポートされているアーキテクチャは 2 つあり、 armel と armhf です。

raspberry pi は armel で動作します。しかし、 armel は armv4 互換で、 armv6 の機能を生かしてくれません。armv6 では VFPv2 (ARM の浮動小数点ユニット) や SIMD ( 整数演算がストリーミング計算できる) 命令ができることになっているので、それを利用しない手はないです。一方ハードウェア浮動小数点命令を活用する設定になっている armhf は一方で armv7 以上でのみ動くので、 armv6 では動いてくれません。

そこで登場するのが Raspbian です。これは armv6, VFP 対応で Debian パッケージをコンパイルしなおしたディストリビューションです。

アーキテクチャ名は armhf となっており、 Debian の armhf アーキテクチャのパッケージをそのままインストールすることができますが、実行してもサポートされていない命令を実行した時点でエラーを出力するようです。

表 4　各 Debian arm アーキテクチャの違い

| | dpkg アーキテクチャ表示 | 浮動小数点演算 ABI | 命令セットアーキテクチャ |
|---|---|---|---|
| Debian armel | armel | soft | armv4 |
| Debian armhf | armhf | hard | armv7 |
| raspbian | armhf | hard | armv6 |

### 7.2.2　インストール

rapbian はインストール済の SD カードを入手するという事も可能ですが、そうでない場合は本体だけでインストールが完了する手順というのはおそらくないので、別途パソコンを用意してください。 SD カードを用意して、配布されている SD カードの起動イメージを dd で書きこみ、本体に挿入して起動するだけでよいです。 [1, 2]

HDMI でモニターに接続して、 USB でキーボードを接続して、イーサネット[*11]も結線しておきます。電源となる USB ケーブルを接続すると LED が点灯して起動します。しばらく起動メッセージが表示され、完了するとメニューが表示されます。

SD カードのイメージを書き込むとなるとパーティションのサイズが気になるかもしれませんが、ファイルシステムを SD カード全体の大きさへ拡張するなどの操作がメニューにあります。

locale の設定とかはあとで適当にやりましょう。

```
$ sudo dpkg-reconfigure locales
```

### 7.2.3　便利な小技

raspberry pi 自体に HDMI 出力や USB などがついているため、メイン開発機として利用することも可能ではありますが、通常は別のマシンなどがあって作業することになると思います。そのときに便利なのが avahi-daemon と sshfs です。

avahi-daemon をいれておけば mdns 対応のクライアントからなら “raspberrypi.local” というホスト名で接続できるようになります。

ssh の設定はするだろうから、 sshfs でファイルシステムをマウントしておけばファイルの共有が楽にできます。 sshfs は ssh の接続さえできればファイルシステムをマウントできるので便利です。個人的には最近はファイル共有は sshfs か git を使っています。

```
sshfs corei7.local:path/to/work ./mnt/
```

### 7.2.4　Raspbian での呼び出し規約

Raspbian（ armhf）では armel と違い、浮動小数点関連の関数の呼び出し規約自体も変わり、使えるレジスタとして r0-r31 だけでなく d0-d31 も利用できるようになります。

具体的なコンパイル例を見てみましょう。 C++ のコード例があった場合にこれがどうコンパイルされるかを見てみます。

[*11] 本体には物理的に RTC（ 時計）が搭載されていないため、起動時に現在時刻が設定されません。ネットワークにつながっている場合はネットワークから時間を取得することになるので通常は問題にはならないようですが、びっくりします。

表 5　1000000000 回 double の掛け算を行うループを実行するのにかかる時間( 秒) 、一回試行

| | armel | armhf | amd64(corei7) |
|---|---|---|---|
| -O1 | 4.5 | 4.6 | 0.35 |
| -O0 | 134 | 130 | 2.6 |

```
double a = 1.1;
double b = 2.3;
cout << a*b;
```

armel: mfloat-abi=soft の場合のアセンブリの出力例。 double が r レジスタで扱われていて、ライブラリコールが行われています。

```
30:        e50b4018        str        r4, [fp, #-24]
34:        e24b1014        sub        r1, fp, #20
38:        e8910003        ldm        r1, {r0, r1}
3c:        e24b301c        sub        r3, fp, #28
40:        e893000c        ldm        r3, {r2, r3}
44:        ebfffffe        bl        0 <__aeabi_dmul>
```

armhf: mfloat-abi=hard のアセンブリ出力例。 ABI が変わり、 d レジスタで関数呼び出しの値が渡されるようになり、 vmul で掛け算が行われています。

```
30:   e50b4018        str     r4, [fp, #-24]
34:   ed1b6b05        vldr    d6, [fp, #-20]  ; 0xffffffec
38:   ed1b7b07        vldr    d7, [fp, #-28]  ; 0xfffffff4
3c:   ee267b07        vmul.f64        d7, d6, d7
40:   e59f0028        ldr     r0, [pc, #40]   ; 70 <main+0x70>
44:   eeb00b47        vmov.f64        d0, d7
48:   ebffffe        bl      0 <_ZNSolsEd>
4c:   e3a03000        mov     r3, #0
```

### 7.2.5　hard float はどれくらい速度向上に貢献するのか?

気になったので掛け算の速度をベンチマークしてみたのですが、サブルーチンを呼び出すコードになっている armel と vmul.64 を直接呼ぶようになっている armhf の違いがよくわからんでした。もっと劇的に違うと思ったのに。

```
double mulbench(int iter, double a, double b) {
  double c;
  for (int i = 0; i < iter; ++i) {
    c = a * b;
  }
  return c;
}

int main(int argc, char** argv) {
  cout << mulbench(atoi(argv[1]), 1.2, 3.5) << endl;
  return 0;
}
```

## 7.3　raspberry pi にデフォルトで搭載されているセンサー

Raspberry pi は外部センサーを付けないと何もできない感じがしますが実は一部計測できるものもあります。電圧と CPU の温度がそれです。

温度を表示してみましょう。ミリ度 C で表示されるようです。今の温度は 55 度のようです。

```
$ cat /sys/class/thermal/thermal_zone0/temp
 55148
```

時系列で温度のグラフを作ってみました。

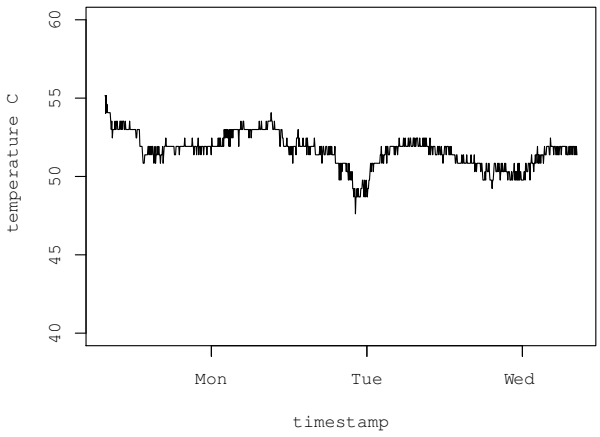



図 2　Raspberry pi CPU 温度 (℃) の時系列での変化

この温度センサーの値は 0.001 ℃単位 (m ℃?) で取得できますが、結果を見た限りでは実際に取得できる値は連続ではなく離散的で、 538 m ℃ 単位のようです。

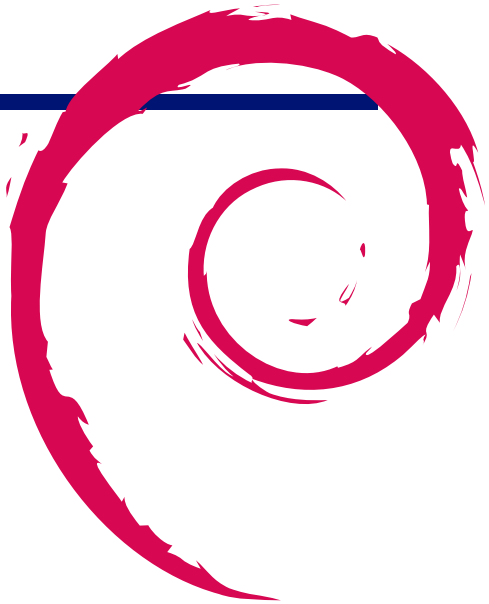
### 7.4　おわりに

Raspberry Pi を購入してみて二ヶ月くらい放置していたのですが、いいかげんたちあげて見ました。しかしまだ長期的に何をさせるのかは考え中。

# 参考文献

[1] raspberry pi のソフトウェアダウンロードサイト (raspbian へのリンクが貼ってある) `http://www.raspberrypi.org/downloads`

[2] raspbian のサイト `http://www.raspbian.org/`



# 8 索引

**Debian 勉強会資料**

2013 年 7 月 20 日　　初版第 1 刷発行

東京エリア Debian 勉強会（ 編集・ 印刷・ 発行）