東京エリアデビアン勉強会

Debian勉強会幹事 上川純一

2009 年 3 月 21 日

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。

2009 年の計画は仮です。

1. 新年の企画 (アンサンブル荻窪開催)
2. OSC Tokyo
3. VAIO P インストール記録、カーネル読書会 ディストリビューション大集合 (小林さん)(東京大学?)
4. Git Handson (岩松)(あんさんぶる荻窪?)
5. 家 Debian サーバ vs 職場のネットワーク (千代田区都立図書館?[*1])
6. Asterisk (東京大学?)
7. スペインにて開催
8. Debconf 報告会
9. OSC Fall?
10. udev + HAL(岩松さん)
11. 3D graphics 開発（藤沢さん）
12. Debian サーバ +VMware + 各種 OS、他の仮想化ツール (vserver etc.)、忘年会

会場候補としては下記があります:

- 大学
- 恵比寿 SGI ホール
- Google オフィス
- 公民館 (あんさんぶる荻窪等)
- 都立会議室 (無線 LAN)
- 健保の施設

[*1] http://www.library.chiyoda.tokyo.jp/

デビアン勉強会

目次

1	Introduction	1
2	**最近の** Debian **関連のミーティング報告**	3
3	Debian Trivia Quiz	4
4	**研究室のソフトウェアを** Debian **パッケージにしてみる**	5
5	Debian **における** Common Lisp **プログラミング環境**	16
6	advi **をデバッグしてみた**	23
7	Debian on chumby **の作り方**	27

2　最近の Debian 関連のミーティング報告

上川 純一

2.1　東京エリア Debian 勉強会 48 回目報告

2009 年 1 月 17 日土曜日に東京エリア Debian 勉強会の第 48 回を開催しました。

今回の参加者は id774 さん、あけどさん、前田耕平さん、小室文さん、山本浩之さん、岩松信洋さん、やまだたくまさん、じつかたさん、キタハラさん、小林儀匡さん、藤沢理聡さん、上川の 12 名でした。

クイズについては、今回は小林さんが当初不在だったのとクイズを用意していなかったのでキャンセルです。何か別の企画を期待したいところです。

最初に最近のイベントの紹介をしました。前回の勉強会のおさらいと、Debian JP で行った IAX 会議について紹介しました。IAX での音声会議に興味をそそられた人もいたようです。

`http://pspunch.com/pd/article/asterisk_meetme_ja.html` にて今回利用した設定が紹介されています。

まず最初に事前課題を紹介しました。

2009 年にどういう内容を実施するのかについて議論しました。ブレインストーミングをしてそのあと 2009 年の毎月の計画をたてました。今年は無事にできるかな?

最後に冬休みの宿題を発表しあって終了。

上川は Git format-patch を利用したワークフローでいかにコンフリクトを発生させないかを紹介しました。前田さんは MacBook に Lenny をインストールしなおしたときにはまった内容。小室さんは、Google Ajax API の紹介。id774 さんは Aspire One にインストールしたときの話。山本浩之さんは 2ch ビューアーパッケージについて。岩松さんは Linux Kernel の .config 自動生成ツールについての紹介でした。

2.2　東京エリア Debian 勉強会 49 回目報告

第 49 回東京エリア Debian 勉強会は OSC2009 Tokyo/Spring @日本電子専門学校に参加してきました。

セミナーとしてはハンズオンを二枠行い、CDBS を用いた Debian パッケージ作成を伝授しました。参加者は全員で 30 名強、当日飛び入りが 4 名いました。ハンズオンをやってみると、参加者のスキルレベルが予想以上にまちまちで、慣れない人だとスペルミスだけでもつまづいてしまい、前準備だけでも 30 分程かかってしまうことなどが分かって来ました。時間内に最後まで課題を終えられた人は半分ぐらいでした。終わらなかった残りの人には、宿題としておきました。しかし、ほとんどの人が時間内に課題をクリアできなかった、昨年のハンズオンよりは一歩前進です。

ブースでは Debian on Chumby や Debian on EeePC の展示や、リリースされたばかりの Lenny で作った Live DVD の配布などを行いました。会場の都合でブースのスペースがとても狭く、初めての人には近寄り難かったかもしれませんが、それでも用意していた DVD、40 枚程がほぼ無くなってしまうほど好評でした。

3 Debian Trivia Quiz

岩松 信洋

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.deban.org` に投稿された内容と Debian Project News からです。

問題 1. 2 月 14 日にリリースされたのは
☐ A etch
☐ B sarge
☐ C lenny

問題 2. 3 月 12 日にリリースされた Debian Policy のバージョンは
☐ A 3.8.0
☐ B 3.8.1
☐ C 3.8.2

問題 3. Debian Data Export は何か
☐ A Debian パッケージに関するデータを容易に取り寄せるためのインターフェイス
☐ B 全世界にある、あらゆるデータを Debian パッケージ化するプロジェクト
☐ C Debian のあらゆるデータを Google にすべて喰わせてみましたというプロジェクト

問題 4. Olivier Berger が ML に流した bts-link に関する情報は
☐ A bts-link のソース管理リポジトリが壊れました
☐ B bts-link でリンクできる BTS を増やしました
☐ C bts-link サービスは終了です

問題 5. Debian FTP master が変わりました。誰が新しく入ったでしょうか。
☐ A Kei Hibino
☐ B Ryan Murray
☐ C Mark Hymers

問題 6. 3 月 16 日に Joerg Jaspert が ML に流したアナウンスは
☐ A Debian の新しいロゴが決まりました
☐ B パッケージのセクションが追加されます
☐ C いくつかのディストリビューションと吸収合併します

問題 7. 3 月 21 日にリリースされた pbuilder のバージョンは?
☐ A 0.187
☐ B 1.298
☐ C 3.1

問題 8. Debian.org DPL に立候補していないのは
☐ A Stefano Zacchiroli
☐ B Steve McIntyre
☐ C Nobuhiro Iwamatsu

問題 9. Debian JP 選挙の立候補しめきりは?
☐ A 3 月 20 日
☐ B 3 月 22 日
☐ C 3 月 24 日

4 研究室のソフトウェアを Debian パッケージにしてみる

藤澤 徹

4.1 対象とするソフトウェア

今回パッケージにしてみるソフトウェアは東京大学 近山・田浦研究室で開発されているグリッド用の MPI ライブラリである MC-MPI, および MC-MPI を動作させるのに必要となる, 同じく近山・田浦研究室で開発されている並列分散シェルの GXP の 2 つです.

4.2 パッケージにするに際して

ソフトウェアをパッケージにするには, そのソフトウェアの構成と特徴をよく理解する必要があります. 以下にそれぞれのソフトウェアの特徴のうち, パッケージを作る際に関係のありそうな事を並べます.

4.2.1 MC-MPI

MC-MPI は主に C 言語で記述されたコンパイラとライブラリによって構成されています. また, 一部に Fortran のコードも含まれており, Fortran インターフェースも持っていますが, これはオプションによりオフにする事もできます.

Autotools を使用しており, `./configure && make && make install` という見慣れたコマンドによってビルド, インストールする事ができます.

4.2.2 GXP

GXP は Python で記述されたコマンドと, そのコマンドが必要とする Python モジュールによって構成されています.

インストールする, という作業は想定されておらず, ダウンロードして展開して出てきたディレクトリをどこかに置き, そこにパスを通して使うように作られています.

4.3 使用するツール

Debian のパッケージを作成する方法にはいくつかあるらしいのですが, 今回は debhelper を使用してみます.

4.4 とりあえず動かしてみる

とりあえずは自分の環境で動く事を確かめるため, パッケージとは関係なく使ってみます.
MC-MPI は GXP を必要とするので, まず GXP から試してみましょう.

4.4.1 GXP を使ってみる

以下のサイトから執筆時点で最新版である gxp-3.05.tar.bz2 をダウンロードし, 展開します.

http://www.logos.t.u-tokyo.ac.jp/gxp/

```
$ mkdir gxp
$ cd gxp
$ # なんとかして gxp-3.05.tar.bz2 を持ってくる
$ tar jxvf gxp-3.05.tar.bz2
$ cd gxp-3.05
```

さて, GXP はインストール不要なので, このままでも動きます. GXP の制御は全て gxpc コマンドで行います.

```
$ ./gxpc
gxpc: no daemon found, create one
/tmp/gxp-***-default/gxpsession-***-***-2009-03-20-06-46-07-15360-92311801
```

問題なく動いているようです.

4.4.2 MC-MPI を使ってみる

以下のサイトから執筆時点で最新版である mcmpi-0.21.0.tar.gz をダウンロードし, 展開します.

http://www.logos.ic.i.u-tokyo.ac.jp/~h_saito/mcmpi/

```
$ mkdir mcmpi
$ cd mcmpi
$ # なんとかして mcmpi-0.21.0.tar.gz を持ってくる
$ tar zxvf mcmpi-0.21.0.tar.gz
$ cd mcmpi-0.21.0
```

前述のとおり, Autotools を使用しているので以下の見慣れたコマンドを打ちます.

```
$ ./configure
$ make
$ sudo make install
```

サンプルプログラムが付属しているので, これを mpicxx でコンパイルしてみます.

```
$ cd app
$ mpicxx -o hello ./hello.cpp
printf: 28: %q: invalid directive
printf: 28: %q: invalid directive
printf: 28: %q: invalid directive
[ g++ -I/usr/local/include    /usr/local/lib/libmpigxp.a -lresolv -lpthread -lnsl -lm  ]
/usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib/crt1.o: In function ‘_start’:
(.text+0x18): undefined reference to ‘main’
collect2: ld はステータス 1 で終了しました
```

おや, 動かないようです. 調べてみたところどうやら %q は使えない事もあるようです. これは文字列を必要ならばクォートするという指定子なので (おそらく), サクッと \"%s\" に置き換えてしまいます. util 以下の mpicc.in, mpicxx.in, mpif77.in に修正を加え, ビルドし直します.

```
$ cd ..
$ make distclean
$ ./configure
$ make
$ sudo make install
```

さて改めてコンパイルします.

```
$ cd app
$ mpicxx -o hello ./hello.cpp
[ g++ -I/usr/local/include "-o" "hello" "./hello.cpp" /usr/local/lib/libmpigxp.a -lresolv -lpthread -lnsl -lm  ]
```

今度は上手くいったようです．では実行してみましょう．まず GXP で `localhost` のみのクラスタを指定し，カレントディレクトリに移動します．そこで `mpirun` によって実行を開始します．

```
$ gxpc
$ gxpc use ssh localhost
$ gxpc explore localhost
$ gxpc cd `pwd`
$ mpirun -np 1 ./hello
/usr/local/bin/mpirun: 157: Bad substitution
```

おや，またしても失敗です．該当行を見ても

```
done
```

と，よく分からない感じですが，よく調べてみると

```
        if [ "$CONF_OPT" == "" -o "${CONF_OPT:0:1}" == "#" ] ; then
```

の行で落ちています．

`${CONF_OPT:0:1}` という書き方は Bash では動きますが POSIX Shell では動かないようなので，とりあえず Bash で動かすように変更してしまいます．

```
$ cd ..
$ make distclean
$ ./configure
$ make
$ sudo make install
```

では改めて実行しましょう．

```
$ cd app
$ mpirun -np 1 ./hello
INFO: _exchange_end_points: 42498
INFO: _measure_latencies: 23884
INFO: _create_bounding_graph: 26520
INFO: _create_routing_table: 57663
INFO: _create_spanning_tree: 20144
INFO: Env_Init: 172304
Hello 0/1
```

今度こそ見事動きました．

4.5 MC-MPI のパッケージ作成

まずは MC-MPI のパッケージを作成します．

4.5.1 準備

とりあえずさっきとは別のディレクトリで作業をしたほうが良さそうです．

```
$ mkdir deb_mcmpi
$ cd deb_mcmpi
$ # なんとかして mcmpi-0.21.0.tar.gz を持ってくる
$ tar zxvf mcmpi-0.21.0.tar.gz
$ cd mcmpi-0.21.0
```

ここでさっきのバグの修正を加えておきます．

4.5.2 さらに修正

MC-MPI では自身のバージョンを `/usr/etc/VERSION` に記述する事になっているのですが，これはビミョーなので `/usr/share/mcmpi/VERSION` あたりに変更しておきます．変更するファイルは以下のとおりです．

- etc/Makefile.in
- util/mpicc.in
- util/mpicxx.in
- util/mpif77.in
- util/mpirun.in

4.5.3 パッケージ情報記述用のファイルの作成

パッケージを作成する場合にはソフトウェア本体はもちろんの事, インストール方法や依存関係等, そのパッケージの情報を記述したファイルが必要になります.

dh_make コマンドを使用するとこれらを記述するためのファイルの雛形を作成してくれます. この際, DEBFULLNAME, DEBEMAIL 変数により作者情報を指定できます. この名前, メールアドレスが GPG の鍵の情報と一致しないと作成したパッケージにサインできないので困ります.

```
$ export DEBFULLNAME="Tooru Fujisawa"
$ export DEBEMAIL="arai_a@mac.com"
$ dh_make -e arai_a@mac.com -f ../mcmpi-0.21.0.tar.gz
Type of package: single binary, multiple binary, library, kernel module or cdbs?
 [s/m/l/k/b]
> s

Maintainer name : Tooru Fujisawa
Email-Address   : arai_a@mac.com
Date            : Fri, 20 Mar 2009 06:55:00 +0900
Package Name    : mcmpi
Version         : 0.21.0
License         : blank
Using dpatch    : no
Type of Package : Multi-Binary
Hit <enter> to confirm:
> [ENTER]
```

途中で Type of package と聞かれます. MC-MPI はライブラリも含みますが, メインはコンパイラ等なのでとりあえず single binary を選んでおけばよさそうです.

4.5.4 パッケージ情報記述用のファイルの修正

さて, さきほどの dh_make によって, debian というディレクトリが作成され, この中にいろいろなファイルが並んでいます.

この中で重要なのは次のものです.

- changelog
- copyright
- dirs
- control
- rule

順番に修正していきましょう.

changelog

これはパッケージの更新履歴です. ソフトウェア本体の更新履歴とは別モノです. とりあえず雛形に沿って以下のようにしておきます.

```
mcmpi (0.21.0-1) unstable; urgency=low

  * Initial release

 -- Tooru Fujisawa <arai_a@mac.com>  Fri, 20 Mar 2009 06:55:00 +0900
```

copyright

これはソフトウェアの著作権情報を記述するファイルです．雛形に沿ってダウンロード元, 元々の作者, ライセンス等を記述します．

```
This package was debianized by Tooru Fujisawa <arai_a@mac.com> on
Fri, 20 Mar 2009 06:55:00 +0900.

It was downloaded from http://www.logos.ic.i.u-tokyo.ac.jp/~h_saito/mcmpi/

Upstream Author(s):
    Hideo Saito <h_saito@logos.ic.i.u-tokyo.ac.jp>

Copyright:
    (c) 2007 Hideo Saito. All Rights Reserved.

License:
    GPL Version 2

The Debian packaging is (C) 2009, Tooru Fujisawa <arai_a@mac.com> and
is licensed under the GPL, see '/usr/share/common-licenses/GPL'.
```

dirs

これはパッケージ作成時に自動的に作成してほしいディレクトリを記述するファイルです．デフォルトでは usr/bin と usr/sbin が入っていますが, usr/sbin は要らないので削除してしまいます．また, VERSION を usr/share/mcmpi にコピーするようにしたのでこれを追加します．

```
usr/bin
usr/share/mcmpi
```

control

これはパッケージの依存関係や説明を記述するファイルです．

雛形通りに進むと, まず Homepage にダウンロード元を書き, Depends に次に作成する gxp を追加しておきます．また, バグ修正の際に Bash を使用する事にしたので bash も追加します．最後に説明を短いものと長いものと書いておきます．

```
Source: mcmpi
Subsection: unknown
Priority: extra
Maintainer: Tooru Fujisawa <arai_a@mac.com>
Build-Depends: debhelper (>= 7), autotools-dev
Standards-Version: 3.7.3
Homepage: http://www.logos.ic.i.u-tokyo.ac.jp/~h_saito/mcmpi/

Package: mcmpi
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends} bash gxp
Description: Grid-enabled implementation of MPI
  MC-MPI is a Grid-enabled implementation of MPI, developed by Hideo
  Saito at the University of Tokyo.  Its main features include the
  following:
  - [Firewall and NAT traversal]: MC-MPI constructs an overlay
    network, allowing nodes behind firewalls and nodes without global
    IP addresses to participate in computations.  There is no need to
    perform maual configuration; MC-MPI automatically probes
    connectivity, selects which connections to establish, and performs
    routing.
  - [Locality-aware connection management]: Establishing too many
    connections, especially wide-area connections, results in many
    problems, including but not limited to the follwing: exhaustion of
    system resources (e.g., file descriptors, memory), high message
    reception overhead, and congestion between clusters during
    all-to-all communication.  Therefore, MC-MPI limits the number of
    connections that are established.  If we assume, for simplicity,
    that n processes are distributed equally among c clusters, then at
    most O(log n) connections are established by each process and at
    most O(n log c) connections are established between clusters.  As
    MC-MPI uses a lazy connect strategy, fewer connections are
    established for applications in which few process pairs
    communicate.  The maximum number of connections allowed can be
    controlled by passing the -beta option to mpirun (see Subsection 3).
  - [Locality-aware rank assignment]: Temporarily disabled in this
    version.
```

rule

これはビルドやパッケージの方法を記述する Makefile です．MC-MPI の場合には Autotools を使っているので得に変える所は無いハズです．(実はありますがそれは後述...)

4.5.5 ソースパッケージの作成

準備が出来たら debuild コマンドでパッケージを作成します．ソースパッケージとバイナリパッケージの両方を作ってみます．

まずはソースパッケージです．

```
$ debuild -S
```

小人さんが頑張ってくれた後, サインをするためのパスフレーズを聞いてくるので 2 回くらい入力します．

するとソースパッケージの完成です．上のディレクトリに色々出来ています．

4.5.6 バイナリパッケージの作成

さて, 続いてバイナリパッケージです．

```
$ debuild
```

configure, make なんかが走っている様子が流れていきます．

```
fortran/.libs/libfortran.a(farg.o): In function ‘mpigxp_getarg’:
*/deb_mcmpi/mcmpi-0.21.0/src/fortran/farg.f:9: undefined reference to ‘_gfortran_getarg_i4’
fortran/.libs/libfortran.a(farg.o): In function ‘mpigxp_iargc’:
*/deb_mcmpi/mcmpi-0.21.0/src/fortran/farg.f:2: undefined  reference to ‘_gfortran_iargc’
fortran/.libs/libfortran.a(initf.o): In function ‘mpi_init__’:
*/deb_mcmpi/mcmpi-0.21.0/src/fortran/initf.c:16: undefined reference to ‘mpigxp_iargc__’
*/deb_mcmpi/mcmpi-0.21.0/src/fortran/initf.c:20: undefined reference to ‘mpigxp_getarg__’
collect2: ld returned 1 exit status
```

超怒られました．しかも私の知らない Fortran のコードです．調べてみたところ, この関数は処理系によっては勝手に作られるものだそうで, gFortran は作らないようです．解決する方法が分からないので, ここはいさぎよく Fortran インターフェースを無効にして作り直しましょう．

debian/rule ファイルの中で, ./configure してる行に --disable-f77 を追加します

```
	./configure $(CROSS) --prefix=/usr --mandir=\$${prefix}/share/man --infodir=\$${prefix}/share/info \
	CFLAGS="$(CFLAGS)" LDFLAGS="-Wl,-z,defs" --disable-f77
```

改めてビルドします．

```
$ debuild
...
/usr/bin/install -c -d /usr/etc
/usr/bin/install: ‘/usr/etc’ の属性を変更できません: No such file or directory
```

またまた怒られました．debuild は debian/mcmpi/ 以下にソフトウェアをインストールしてパッケージにするハズなのに, 外にインストールしようとしています．これは rule ファイルからディレクトリを DESTDIR として渡しているのに, Makefile の方が対応していないためです．

etc/Makefile.in, util/Makefile.in, の中で prefix に DESTDIR を追加します．

```
prefix = $(DESTDIR)@prefix@
```

```
$ debuild
```

今度は成功しました．

上のディレクトリに mcmpi_0.21.0-1_i386.deb が出来ています．

4.5.7 インストールテスト

依存関係が正しいかどうか, インストールしてみましょう.

```
$ cd ..
$ sudo dpkg -i mcmpi_0.21.0-1_i386.deb
未選択パッケージ mcmpi を選択しています。
(データベースを読み込んでいます ... 現在 219582 個のファイルとディレクトリがインストールされています。)
(mcmpi_0.21.0-1_i386.deb から) mcmpi を展開しています...
dpkg: 依存関係の問題により mcmpi の設定ができません:
 mcmpi は以下に依存 (depends) します: gxp ... しかし:
  パッケージ gxp はまだインストールされていません。
dpkg: mcmpi の処理中にエラーが発生しました (--install):
 依存関係の問題 - 設定を見送ります
以下のパッケージの処理中にエラーが発生しました:
 mcmpi
```

gxp が無いと言われました. 予定通り作成できているようです. とりあえず削除しておきます.

```
$ sudo apt-get remove --purge mcmpi
```

4.6 GXP のパッケージ作成

さて, 無いと言われた GXP パッケージの方を作ります.

4.6.1 準備

こちらもさっきとは別のディレクトリで作業をします. ただし今回, 作業を開始した時点では 3.03 が最新バージョンだったので, バージョンアップのテストも兼ねて 3.03 のパッケージをまず作ります.

```
$ mkdir deb_gxp
$ cd deb_gxp
$ # なんとかして gxp-3.03.tar.bz2 を持ってくる
$ tar jxvf gxp-3.03.tar.bz2
$ cd gxp-3.03
```

4.6.2 パッケージ情報記述用のファイルの作成

ほぼ同様です.

```
$ export DEBFULLNAME="Tooru Fujisawa"
$ export DEBEMAIL="arai_a@mac.com"
$ dh_make -e arai_a@mac.com -f ../gxp-3.03.tar.bz2
Type of package: single binary, multiple binary, library, kernel module or cdbs?
 [s/m/l/k/b]
> s

Maintainer name : Tooru Fujisawa
Email-Address   : arai_a@mac.com
Date            : Fri, 20 Mar 2009 07:15:35 +0900
Package Name    : gxp
Version         : 3.03
License         : blank
Using dpatch    : no
Type of Package : Single
Hit <enter> to confirm:
> [ENTER]
```

GXP は外から見ればコマンド 1 つなので, これも `single binary` でよさそうです.

4.6.3 パッケージ情報記述用のファイルの修正

さて, これらを記述する前に考えなければならない事があります. GXP はインストールを想定されていないため, パッケージにした場合にどこに置いてどのように使うかを決めなければいけません.

今回は `/usr/share/gxp` 以下にファイルをコピーし, `/usr/bin/gxpc` を `/usr/share/gxp/gxpc` にリンクする事にします.

changelog

特に変わった事はしません.

```
gxp (3.03-1) unstable; urgency=low

  * Initial release

 -- Tooru Fujisawa <arai_a@mac.com>  Fri, 20 Mar 2009 07:15:35 +0900
```

copyright

こちらも特に変わった事はしません.

```
This package was debianized by Tooru Fujisawa <arai_a@mac.com> on
Fri, 20 Mar 2009 07:15:35 +0900.

It was downloaded from http://www.logos.t.u-tokyo.ac.jp/gxp/

Upstream Author(s):
    Dun Nan
    Kenjiro Taura
    Yoshikazu Kamoshida

Copyright:
    (c) 2008 by Kenjiro Taura. All rights reserved.
    (c) 2007 by Kenjiro Taura. All rights reserved.
    (c) 2006 by Kenjiro Taura. All rights reserved.
    (c) 2005 by Kenjiro Taura. All rights reserved.

License:
    GPL Version 2

The Debian packaging is (C) 2009, Tooru Fujisawa <arai_a@mac.com> and
is licensed under the GPL, see '/usr/share/common-licenses/GPL'.
```

dirs

さて, GXP はインストーラを持っていないので, インストールのコードは直接 `rule` に書く事になります. できるだけ作業を減らしたいので, 必要なディレクトリはこちらに全部書いてしまいましょう.

```
usr/bin
usr/share
usr/share/gxp
```

control

GXP は Python のモジュールを内部に持っているので, これらをインストール先でバイトコードにコンパイルしてあげる作業をしてあげなければいけません. この作業を勝手にしてくれるのが `dh_pycentral` です. このために, 2 箇所に `XS-Python-Version: all` を追加します.

また, GXP は環境にアーキテクチャに依存しないので, `Architecture` を `all` にします. ただし, 実行に Python が必要になるので `Depends` に `python` を, また前述の作業のために `dh_pycentral` が必要になるので, `Depends`, `Build-Depends` に `python-central` を追加します.

```
Source: gxp
Subsection: unknown
Priority: extra
Maintainer: Tooru Fujisawa <arai_a@mac.com>
Build-Depends: debhelper (>= 7)
Standards-Version: 3.7.3
XS-Python-Version: all
Homepage: http://www.logos.t.u-tokyo.ac.jp/gxp/

Package: gxp
Architecture: all
Depends: ${shlibs:Depends}, ${misc:Depends}, python
XS-Python-Version: all
Description: parallel/distributed shell
 GXP is a parallel/distributed shell, plus a parallel task execution engine
 that runs your Makefile in parallel on distributed machines.
 Very easy to install
 (no need to compile. install it on YOUR machine and use it on ALL machines).
```

rule

まず, `Makefile` が無いので `$(MAKE)` と入った行は全てコメントアウトします.

そして, `$(MAKE) DESTDIR=$(CURDIR)/debian/gxp install` の行の後に次のようなインストールのコマンドを追加します. 展開して出てきたもの全部, としたいのですが, `debian` ディレクトリがあるのでファイルを並べます.

```
	cp -r ChangeLog License README doc ex expectd.py gxpbin gxpc gxpc.py gxpd.py gxpm.py ifconfig.py inst_local.py \
	inst_remote.py inst_remote_stub.py ioman.py misc mkrelease opt.py $(CURDIR)/debian/gxp/usr/share/gxp
	ln -s $(CURDIR)/debian/gxp/usr/share/gxp/gxpc $(CURDIR)/debian/gxp/usr/bin/gxpc
```

また, `dh_pycentral` を動かすために, `dh_python` の次の行あたりに `dh_pycentral` と書いておきます.

```
#	dh_python
	dh_pycentral
```

4.6.4 ソースパッケージの作成

まずはソースパッケージを作ります.

```
$ debuild -S
```

何事もなく完了します.

4.6.5 バイナリパッケージの作成

さて, 続いてバイナリパッケージです.

```
$ debuild
...
E: gxp: missing-dep-for-interpreter expect => expect (./usr/share/gxp/gxpbin/ssh_passwd)
E: gxp: missing-dep-for-interpreter expect => expect (./usr/share/gxp/gxpbin/su_cmd)
...
```

途中でエラーが出ています. これは `ssh_passwd`, `su_cmd` が `/usr/bin/expect` を使用しているのに, `Depends` に入っていないという事なので, `debian/control` を更新します.

```
Depends: ${shlibs:Depends}, ${misc:Depends}, python, python-central, expect
```

ではビルドし直します.

```
$ debuild
```

今度は何事もなく完了し, 上のディレクトリに `gxp_3.03-1_all.deb` ができています.

4.6.6 バージョンアップ

さて, 作業をしている間に GXP の新しいバージョン 3.05 がリリースされたのでこれをパッケージに反映します. debhelper には新しいバージョンのチェック, 更新を自動化するための機構があります. `debian/watch` ファイルに

以下のように記述します. (というか, SourceForge なモノは例にあるので拡張子だけ変えます)

```
version=3
http://sf.net/gxp/gxp-(.*)\.tar\.bz2
```

2 行目が最新バージョンの URL のパターンです. ここから自動的に最新バージョンを探して落としてくれます.

```
$ uscan -verbose
-- Scanning for watchfiles in .
-- Found watchfile in ./debian
-- In debian/watch, processing watchfile line:
   http://sf.net/gxp/gxp-(.*)\.tar\.bz2
-- Found the following matching hrefs:
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.02.tar.bz2
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.02.tar.bz2
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.03.tar.bz2
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.03.tar.bz2
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.05.tar.bz2
     /sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.05.tar.bz2
Newest version on remote site is 3.05, local version is 3.03
 => Newer version available from
    http://www.mirrorservice.org/sites/download.sourceforge.net/pub/sourceforge/g/gx/gxp/gxp-3.05.tar.bz2
-- Downloading updated package gxp-3.05.tar.bz2
-- Successfully downloaded updated package gxp-3.05.tar.bz2
    and symlinked gxp_3.05.orig.tar.bz2 to it
-- Scan finished
```

と, いう感じで 3.05 がダウンロードされました. これを展開し, 簡単に現在と同じような構成にする事ができます.

```
$ uupdate ../gxp-3.05.tar.bz2
New Release will be 3.05-0ubuntu1.
-- Untarring the new sourcecode archive ../gxp-3.05.tar.bz2
Success!  The diffs from version 3.03-1 worked fine.
Remember: Your current directory is the OLD sourcearchive!
Do a "cd ../gxp-3.05" to see the new package
$ cd ../gxp-3.05/
```

はて, 何かバージョンがおかしな事になっています. これは `debian/changelog` にのみ影響するので, これを書き換えておきます.

```
gxp (3.05-1) unstable; urgency=low

  * New upstream release

 -- Tooru Fujisawa <arai_a@mac.com>  Fri, 20 Mar 2009 07:32:35 +0900

gxp (3.03-1) unstable; urgency=low

  * Initial release

 -- Tooru Fujisawa <arai_a@mac.com>  Fri, 20 Mar 2009 07:15:35 +0900
```

あとはビルドし直せば完了です.

```
$ debuild -S
$ debuild
...
E: gxp: ruby-script-but-no-ruby-dep ./usr/share/gxp/gxpbin/tmsub.rb
...
```

さて, 今度は Ruby が必要になったようなので, これを `debian/control` に追加します.

```
Depends: ${shlibs:Depends}, ${misc:Depends}, python, python-central, expect, ruby1.8
```

ではビルドし直します.

```
$ debuild
```

4.6.7 インストールテスト

GXP の方は特別な依存関係は無いのでインストールできるハズです.

```
$ cd ..
$ sudo dpkg -i gxp_3.05-1_all.deb
...
 gxp は以下に依存 (depends) します: expect ... しかし:
  パッケージ expect はまだインストールされていません。
...
```

と思ったら expect が入ってませんでした.

```
$ apt-get remove --purge gxp
$ sudo apt-get install expect
$ sudo dpkg -i gxp_3.05-1_all.deb
未選択パッケージ gxp を選択しています。
(データベースを読み込んでいます ... 現在 219600 個のファイルとディレクトリがインストールされています。)
(gxp_3.05-1_all.deb から) gxp を展開しています...
gxp (3.05-1) を設定しています ...
```

続いて MC-MPI もインストールします.

```
$ cd ../deb_mcmpi
$ sudo dpkg -i mcmpi_0.21.0-1_i386.deb
未選択パッケージ mcmpi を選択しています。
(データベースを読み込んでいます ... 現在 219699 個のファイルとディレクトリがインストールされています。)
(mcmpi_0.21.0-1_i386.deb から) mcmpi を展開しています...
mcmpi (0.21.0-1) を設定しています ...
```

今度は正しくインストールできました.

5 DebianにおけるCommon Lisp プログラミング環境

日比野 啓

5.1 Common Lisp はどんな言語か

Debian が Common Lisp のために用意している仕組みが何を目的としているのか、なぜそのような仕組みが用意されているのかを理解するために、まずは Common Lisp がどんな言語でライブラリをどのように構築しているのかを説明します。

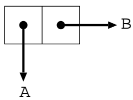
5.1.1 Lisp のプログラムの構文

Lisp のプログラムは、S 式と呼ばれる、トークンと入れ子の括弧の列で表現されます。表記と構造を対応づけて説明するために、まずはドット . を使った表記から説明します。

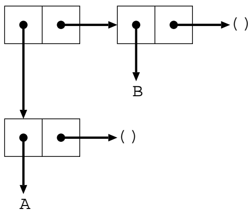
```
S-exp  : () | token | (S-exp . S-exp)
```

S 式は 空の括弧 () かトークンか S 式のペア (ドットをはさんで括弧でくくったもの) ということになります。S 式のペアの構造を考えるには以下のようなポインタの組を考えると構造的な理解がしやすいです。

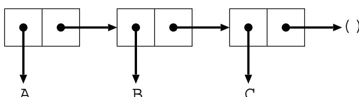
```
(A . B)
```




```
((A . ()) . (B . ()))
```

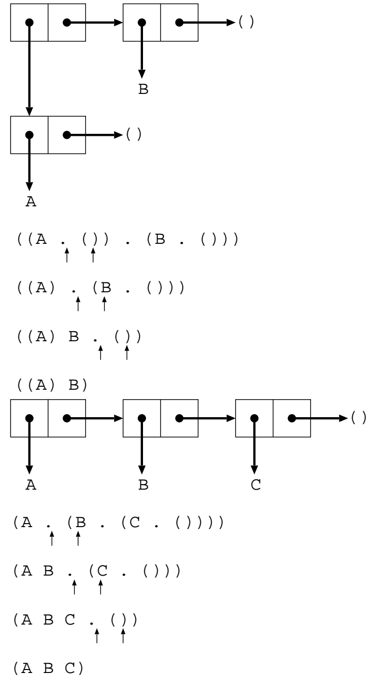



```
(A . (B . (C . ())))
```



このポインタのペアのことを Lisp では cons セルと呼びます。左側のポインタは car、右側のポインタは cdr と呼びます。

cdr が cons セルあるいは空の括弧 () を指している場合は . と cdr 内の括弧 ( ) を省略できます



このように cdr で連なる連結リストを簡潔に表現することができます。空の括弧 () は要素が一つもない空のリストということです。Common Lisp では空リスト () は nil と書くこともできます。この . の省略まで含んだのが一般的に S 式と呼ばれている表記です。

Lisp の処理系はこのリスト構造で表現されるプログラムを処理することで実現できます。LISP は LISt Processing language の略だというわけです。LISP のプログラムがリスト構造と等価であるということが今回の話の重要なポイントなので注意しておいてください。

5.1.2 Common Lisp のプログラム

では実際に Lisp のプログラムを見ていきましょう。ここから先は実際に対話環境でプログラムを試しながら説明していきます。`CL-USER>`というのが対話環境のプロンプトです。

■関数の呼び出し ではもともと定義されている関数を呼び出してみます。足し算を行なう関数 + の例です。

```
CL-USER> (+ 1 2 3)

6
```

リストの最初の要素が関数の名前で、残りの要素が引数です。かけ算の関数*も使ってみましょう。

```
CL-USER> (* (+ 1 2) 3)

9
```

引数の計算を行なったのちに関数の呼び出しが行なわれます。この引数の計算のことを引数を 評価する と言います。

■関数の定義 自分でも関数の定義を行なってみましょう。

```
CL-USER> (defun my-plus (x y)
 (+ x y))
MY-PLUS
CL-USER> (my-plus (* 2 3) 2)
8
```

2 つの引数を足し算する関数が定義できました。

```
(defun <関数名> (<引数>*) [<省略可能なドキュメント文字列>] <本体の式>*)
```

最後の body form の結果が関数の返り値になります。

■特殊オペレーター - special operator　Common Lisp の構文はほぼ S 式しかありません。では条件分岐やループといったプログラムの制御構造はどうやって実現しているのでしょうか。

たとえば条件分岐を行なうために `if` という特殊オペレーターがあります。`t` は真を示したいときに慣習的に使用する値です。

```
(if <式> <条件が nil ではない> [<条件が nil>])
```

```
CL-USER> (if t (print "then") (print "else"))

"then"
"then"
CL-USER> (if nil (print "then") (print "else"))

"else"
"else"
```

Common Lisp では `nil` が偽でそれ以外は真です。`if` は関数で表現することはできません。もし関数であったとすると、

```
CL-USER> (defun my-if (p then else) (if p then else))
MY-IF
CL-USER> (my-if T (print "then") (print "else"))

"then"
"else"
"then"
```

というように、`then` の部分も `else` の部分も関数 `my-if` の引数ですから、両方とも評価された後に my-if が呼び出されてしまうのです。

長くなりそうなので詳しくは述べませんが、ループを実現できる機能としては C の goto のような動きをする go という特殊オペレーターがあります。

■マクロの呼び出し　Lisp では S 式で表現できる構文を自分でも定義することができます。それが Lisp のマクロです。

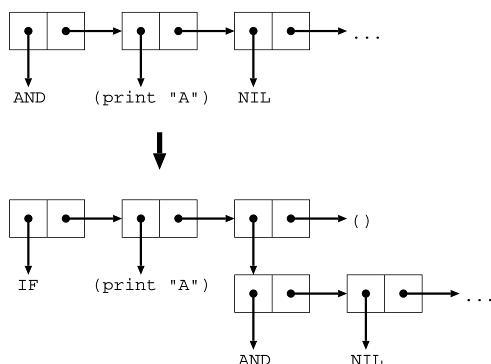
まずはもともと定義されているマクロ and を使ってみます。

```
CL-USER> (and (print "A") (print "B") (print "C"))

"A"
"B"
"C"
"C"
CL-USER> (and (print "A") nil (print "C"))

"A"
NIL
```

`and` マクロは引数の式の評価が真であるかぎりは残りを評価し、偽 (`nil`) より後は評価しません。最後に評価した値が結果になります。引数が無い場合は `t` が結果になります。マクロは S 式を S 式に変換する機能だと考えるとわかりやすいです。これはあるリスト構造を別のリスト構造に変換するということでもあります。関数 `macroexpand-1` を使うと `and` マクロでどのような変換が行なわれたのかを見ることもできます。

```
CL-USER> (macroexpand-1 '(and (print "A") nil (print "C")))
(IF (PRINT "A") (AND NIL (PRINT "C")) NIL)
T
```

引数の一つ目を条件とする if の式になりました。評価はマクロが全て変換された後に行なわれます。リストだと思って見てみれば以下のような変形です。



マクロの動きを理解するにはS式とリストを対応づけて見ていくのがコツです。

■マクロの定義　自分でも and マクロのようなもの定義してみます。

```
CL-USER> (defmacro my-and (&rest forms)
  (if forms
      (list 'if (car forms) (cons 'my-and (cdr forms)))
      t))
MY-AND
CL-USER> (my-and (print "A") (print "B") (print "C"))

"A"
"B"
"C"
T
CL-USER> (my-and (print "A") NIL (print "C"))

"A"
NIL
CL-USER> (macroexpand-1 '(my-and (print "A") nil (print "C")))
(IF (PRINT "A") (MY-AND NIL (PRINT "C")))
T
```

car は cons セルから car を返す関数、list は複数の引数をリストにして返す関数、cons は 2 つの引数を car cdr の順に指す cons セルを作る関数です。&rest は可変引数をリストで受けとるためのパラメータの指定です。

このように、マクロはリスト構造を変換するようなプログラムを書いて定義を行ないます。マクロの定義の中身はマクロの展開のときに評価が行なわれるということが、ここでの重要なポイントです。

似たような動きをしているようですが、オリジナル and とは少し違っています。最後に評価されたものが結果にはなっていないようです。ここでは定義を単純にするために少し動きを変えてみました。

```
(defmacro <マクロの名前> (<引数>*) [<省略可能なドキュメント文字列>] <本体の式>*)
```

5.2　Common Lisp のライブラリとマクロ

Common Lisp にも他の実用的な言語と同様に数多くのライブラリがあります。他の言語とは異なるかもしれない事情はライブラリ内のマクロの存在です。マクロの展開がすべて終わった後でないとプログラムをコンパイルし、実行することができないからです。

たとえば、あるライブラリ A は別のライブラリ B のマクロを使用しているかもしれません。すると、A のコードは B のマクロ定義をすべて展開した後でないとコンパイルすることができません。さらに、対話環境で開発を行なうことを考えたとき、利用することにしているライブラリをコンパイルが済んだ状態でロードしておきたいと思うかもしれません。そのときにはライブラリをロードした後にマクロ展開を全て行ない、その後にコンパイルする必要があるのです。数多くのライブラリを使用することにしていたら対話環境を利用できる状態にするまでに多くの時間がかかってしまいます。

このような状況を解決するために、Lisp の処理系では、マクロの展開とコンパイルが済んだ状態のイメージをダンプして保存しておいて再利用するのが一般的です。

5.2.1 ASDF - Another System Definition Facility

Common Lisp ライブラリのコンパイルを支援するためのライブラリとして ASDF があります。Makefile のようなもので、コンパイルに必要な情報を記述しておくことができます。ASDF ではライブラリモジュールのことを system と呼んでいて、system ごとに名前 (system name) を付けることになっています。同じモジュール内でファイル間に依存関係がある場合はそれを記述します。コンパイルに必要な別の system がある場合はその system name も記述します。Debian では cl-asdf パッケージです。以下は SBCL のドキュメントにあった ASDF のシステム定義の例です。

```
(defpackage hello-lisp-system
  (:use :common-lisp :asdf))

(in-package :hello-lisp-system)

(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2"
    :author "Joe User <joe@example.com>"
    :licence "Public Domain"
    :components ((:file "packages")
                 (:file "macros" :depends-on ("packages"))
                 (:file "hello" :depends-on ("macros"))))
```

`hello-lisp` という名前のシステム定義を行なっています。

5.2.2 Common Lisp Controller

Common Lisp の処理系のダンプイメージを作りなおしてくれるツールです。処理系のパッケージを追加するとダンプイメージを作ってくれます。ライブラリをインストールすると各々の処理系ごとにダンプイメージを作り直してくれます。ライブラリが ASDF に対応していることが条件です。Debian では common-lisp-controller パッケージです。

5.2.3 dh-lisp

Common Lisp の処理系やライブラリの Debian パッケージ作成時に common-lisp-controller に対応させるための支援をしてくれるツールです。

パッケージのビルドの過程で dh_lisp コマンド呼び出すようにすると、パッケージ内の ASDF の定義を書いたファイル (.asd) を検索して、common-lisp-controller を呼び出すフックをメンテナスクリプトに追加してくれます。common-lisp-controller がメンテナスクリプトから呼びだされたときには asd ファイルの名前を見てダンプイメージ作り直しの対象かどうか調べてからダンプが行なわれます。`/etc/common-lisp/images/<implementation>`に asd ファイルの名前を書いておくと作り直しの対象になります。

現状だと例えばパッケージインストール用には以下のようなフックが追加されます。

```
if [ "$1" = "configure" ] &&
   which register-common-lisp-source > /dev/null; then
   register-common-lisp-source "#SYSTEMDIR#"
fi
```

`#SYSTEMDIR#`が asd ファイルの名前に置き換わります。

Common Lisp の処理系のパッケージを作成する場合には、ダンプイメージ出力のスクリプトを用意して、dh_lisp の引数に与える名前に合わせた名前を付けてやれば、やはり common-lisp-controller を呼び出すフックをメンテナスクリプトに追加してくれます。

現状だと例えばパッケージインストール用には以下のようなフックが追加されます。

```
case "$1" in
   configure)
           if [ -x /usr/lib/common-lisp/bin/"#IMPLEMENTATION#.sh" ] &&
               which register-common-lisp-implementation > /dev/null; then
               register-common-lisp-implementation "#IMPLEMENTATION#"
           fi
           ;;
   abort-upgrade|abort-remove|abort-deconfigure)
           if which register-common-lisp-implementation > /dev/null; then
               unregister-common-lisp-implementation "#IMPLEMENTATION#"
           fi
           ;;
esac
```

`#IMPLEMENTATION#`が dh_lisp の引数に与える名前に置き換わります。

5.3 Emacs での開発環境

最後に今回の話で使用している Emacs 上の対話環境の紹介をしておきます。

5.3.1 SLIME - Superior Lisp Interaction Mode for Emacs

SLIME は Emacs 用の Lisp 開発環境です。Debian では slime というパッケージに入っています。Emacs 側の Elisp で書かれたクライアントと Lisp の処理系側で書かれたサーバーが通信しながら対話的な開発環境を実現しています。Lisp の処理系側で書かれたサーバーの実装は swank と呼ばれています。swank を実装すれば他の Lisp の処理系でも slime を使うことができるらしいです。

Emacs からの利用方法ですが、たとえば処理系に SBCL を使用する場合は.emacs には以下のように書いておけばよいでしょう。

```
(setq slime-auto-connect 'ask)
(setq inferior-lisp-program "sbcl")
```

Emacs のキーバインドで、対話環境で試験的に実行してみるときに良く使いそうなものを挙げておきます。

C-c C-z run-lisp　Lisp 処理系との対話用バッファへスイッチ

C-c C-c slime-compile-defun　カーソル位置の関数を対話用バッファの環境でコンパイル

C-c C-k slime-compile-and-load-file　編集中のプログラムのバッファのファイルを対話用バッファの環境でコンパイルしてロードする

C-c C-l slime-load-file　対話用バッファの環境で Lisp プログラムのファイルをロードする

5.3.2 Hyperspec

ANSI Common Lisp の仕様のオンラインドキュメントを SLIME から読むことができます。Debian では hyperspec というパッケージがインストーラーのパッケージになっています。

たとえば w3m-el パッケージを入れておいた状態で .emacs で

```
(set-default 'browse-url-browser-function 'w3m-browse-url)
```

などとやっておくと emacs バッファ内で関数やマクロのヘルプを読むことができます。次のキーバインドでドキュメント内の検索を行なうことができます。

C-c C-d h slime-hyperspec-lookup　カーソル位置のワードで Hyperspec のドキュメント内を検索

5.4 参考文献

- 実践 Common Lisp
  - ISBN : 978-4274067211

– 著者：Peter Seibel
– 出版社：オーム社

6 advi をデバッグしてみた

日比野 啓

2008 年 11 月の LaTeX を使ったハンズオンで、wizzytex-mode から使われている advi がときどき固まってしまう問題について調べてみました。

6.1 advi がまる?

advi は一見普通の DVI viewer なのですが、なぜか OCaml という変わった言語で実装されています。今回は advi から呼ばれる ghostscript が止まっているらしい、ということまで分かっている状態から調べ始めました。

6.2 とりあえずアタリをつける

とりあえず、問題が起きているソースを取ってきて展開してみます。

```
% apt-get  source advi
...
dpkg-source: extracting advi in advi-1.6.0
dpkg-source: info: unpacking advi_1.6.0.orig.tar.gz
dpkg-source: info: applying advi_1.6.0-13.diff.gz
% cd advi-1.6.0
% ls *.ml
addons.ml     drawimage.ml  font.ml            gs.ml            main.ml     search.ml      transimpl.ml
ageometry.ml  driver.ml     global_options.ml  gterm.ml         misc.ml     shot.ml        ttfont.ml
...
```

*.ml というのが OCaml のソースファイルです。なんか、gs.ml とかいうそのものズバリっぽいものが見えます。gs.ml の中をまず gs で検索していってみると、

```
...
  let command = Config.gs_path in
  let command_args =
    [|
      command;
      "-dNOPLATFONTS"; "-dNOPAUSE";
      "-sDEVICE=" ^ (if !antialias then x11alpha else x11);
      "-q";
      "-dSAFER";
      "-";
    |] in

  let _ = debugs command;
...
```

おお、それっぽい。あと、デバッグ用っぽい機能 - debugs を発見。さらにこんどは command で探していくと、

```
...
  let lpd_in, lpd_out = Unix.pipe () in
...
  let leftout = Unix.out_channel_of_descr lpd_out in
...
  let pid =
    Unix.create_process command command_args lpd_in rpd_out
      (* Unix.stdout *) Unix.stderr
...
    method line l =
      try
        showps l;
        output_string leftout l;
        output_char leftout '\n';
...
```

どうやら gs にパイプで PS を書きこんでいるようです。showps とかいうので PS の中身を見ることができるんじゃないかなーとか。

6.3 まじめに調べてみたんですが...

もう一度、こんどは gs.ml の最初の方からデバッグ用の機能だけ見ていきます。

```
...
let debugs = Misc.debug_endline;;
...
let showps_ref = ref false;;
let showps s =
  if !showps_ref then (print_endline  (Printf.sprintf "%s" s));;
...
Options.add
  "--showps" (Arg.Set showps_ref)
  "  ask advi to print to stdout a copy\
  \n\t of the PostScript program sent to gs.";;
...
```

`Misc.` というのは Misc という別のモジュールへの参照です。ここでは単に misc.ml の中を見ればよさそうです。`showps_ref` は書き換え可能なフラグのようです。と思ったらすぐ下にコマンドライン引数からフラグをセットできるようになっているようです。misc.ml の中も見てみると、

```
...
(* Debugging. *)
let forward_debug_endline =
  ref (function (_ : string) -> failwith "undefined forward debug_endline");;

let debug_endline s = (!forward_debug_endline s : unit);;

let set_forward_debug_endline f = forward_debug_endline := f;;
...
```

さらに `set_forward_debug_endline` で grep すると、`global_options.ml` が引っかかるので、その中も見てみると

```
...
(* To print debugging messages. *)
let debug_endline = Options.debug "--debug" " General debug";;

(* Setting the forward in Misc. *)
Misc.set_forward_debug_endline debug_endline;;
...
```

結局、どっちもコマンドラインから設定できるようですね。さっそく試してみると、

```
% platex debianmeetingresume200812-presentation.tex
...
% advi debianmeetingresume200812-presentation.dvi
...
/usr/bin/gs
-dNOPLATFONTS
-dNOPAUSE
-sDEVICE=x11
-q
-dDELAYSAFER
-
...
%!PS-Adobe-2.0
%%Creator: Active-DVI
%!
[1 0 0 -1 0 0] concat
(/usr/share/texmf-texlive/dvips/base/texc.pro) run
(/usr/share/texmf-texlive/dvips/base/special.pro) run
...
%% Newpage

grestore
0 0 moveto
TeXDict begin 12769384 12769384 div dup /Resolution X /VResolution X end
TeXDict begin /DVImag 194.845342 def end
gsave
flushpage (...
) print flush
```

たしかに gs のコマンドラインらしきものと、それから書きこんだ PS の内容らしいものが見えてます。PS で目印となる文字列を出力される命令 `flushpage (...) print flush` を gs に書きこんで、その出力を待っているようなのですが、戻ってきていないようです。

gs が止まってしまう場合とそうでない場合も比べてみたのですが、止まってしまう場合の PS の最小セットを割り出すのが難しく、よくわかりませんでした。

6.4 別の回避策?

なにか別の方法で止まってしまうのを回避できないか、と gs の出力を待っている部分も見てみます。

```
...
let rec select fd_in fd_out fd_exn timeout =
  (* dirty hack: Graphics uses itimer internally! *)
  let start = Unix.gettimeofday () in
  try
    Unix.select fd_in fd_out fd_exn timeout
  with
    Unix.Unix_error (Unix.EINTR, _, _) as exn ->
      let now = Unix.gettimeofday () in
      let remaining = start +. timeout -. now in
      if remaining > 0.0 then select fd_in fd_out fd_exn timeout else [], [], []
...
      match select [ rpd_in ] [] [] 1.0 with
      | [], _, _ ->
          begin match Unix.waitpid [ Unix.WNOHANG ] pid with
          | x, Unix.WEXITED y when x > 0 ->
              raise (Killed "gs exited")
          | 0, _ ->
              raise (Killed "gs alive but not responding")
          | _, _ ->
              raise (Killed "gs in strange state")
          end
...
```

gs の出力を select で待っているようです。タイムアウトも仕込んであるようです。なぜうまくいっていないのでしょう。

ここでは前半で定義されている select に注目です。せっかくタイムアウトの残り時間を計算しているのに、渡しているのはもとの値です。どうりでいつまでたってもタイムアウトしないわけです。

```
...
if remaining > 0.0 then select fd_in fd_out fd_exn timeout else [], [], []
...
```

これを

```
...
if remaining > 0.0 then select fd_in fd_out fd_exn remaining else [], [], []
...
```

と直すと、gs を待ってもタイムアウトするようになります。gs が固まる原因を取り除くような根本的な解決はでき

ませんでしたが、とりあえずは advi が止まらないようにはなりそうです。

7 Debian on chumbyの作り方

まえだこうへい

OSC 2009 Tokyo/Spring での東京エリア Debian 勉強会のブースで、Debian on chumby を行いました。今回はその環境の作り方についてまとめました。

7.1 概要

今回、実は chumby の上でネイティブに Debian を動かしたわけではありません。USB メモリにインストールした Debian に chroot して擬似的に動かしているように見せかけました。ネイティブに動かすとなるとブートローダをいじる必要がありますが、今回は chumby 自体はほとんど変更せずに済む方法をとりました。

7.1.1 chumby の仕様

chumby は、インターネットに接続可能な無線 LAN 環境が必要で、接続できなければアナログ時計の widget の表示しかできません。また、書き込み可能なメモリ領域はフラッシュメモリも 64MB のうち、わずかです[*2]。当然、Debian をローカルにインストールすることはできないので、USB メモリを外部ストレージとして使います。

もう一つの制約は、chumby は、ext2 などを使えません。USB メモリを使う場合は vfat のみです。しかし vfat では Linux をインストールできません[*3]。そこで、大きく 3 つやることがあります。

1. chumby のカーネルリビルド
2. USB メモリへの Debian インストール
3. USB メモリの Debian への chroot 設定

7.2 環境構築

7.2.1 前提条件

環境構築時に最低限必要なもの

- chumby
- USB メモリ
- Debian 環境構築用の PC
- ネットワーク環境

7.2.2 chumby のカーネルリビルド

前述のとおり、chumby の kernel は ext2 を使えません。今回、USB メモリには ext2 フォーマットで Debian をインストールするので、chumby 自体も ext2 を読み込めるようにします。まず、下記リンク先から chumby のカーネル構築用のツールキットを入手します。手順はリンク先に従います。

- GNU Toolchain[*4]
- GCC Toolchain[*5]

これらのツールキットは、/usr 以下に展開されるので、kvm/qemu などの仮想 OS 環境に、環境を構築すると良いでしょう。

[*2] jffs2 ファイルシステムで/psp としてマウントされています。
[*3] 原因は symlink を作成できないこと、適切なパーミッションを設定できないこと、など。
[*4] `http://wiki.chumby.com/mediawiki/index.php/GNU_Toolchain`
[*5] `http://wiki.chumby.com/mediawiki/index.php/GCC_Toolchain`

```
$ cd /
$ sudo tar zxf ~/arm-linux-v4.1.2b.tar.gz
$ sudo tar zxf ~/Gcc-3.3.2-glibc-2.3.2.tar.gz
$ sudo mkdir -p /opt/Embedix/usr/local/arm-linux
$ sudo ln -s /usr \
 /opt/Embedix/usr/local/arm-linux/gcc-3.3.2-glibc-2.3.2
$ sudo vi /usr/bin/arm-linux-make
$ sudo chmod +x /usr/bin/arm-linux-make
```

/usr/bin/arm-linux/make には以下のように記述します。

```
#!/bin/sh
echo make ARCH=arm CROSS=arm-linux- CC=arm-linux-gcc \
 AR=arm-linux-ar NM=arm-linux-nm RANLIB=arm-linux-ranlib \
 CXX=arm-linux-g++ AS=arm-linux-as LD=arm-linux-ld \
 STRIP=arm-linux-strip BUILDCC=gcc BUILD_CC=gcc \
 CC_FOR_BUILD=gcc ‘‘$@’’
exec make ARCH=arm CROSS=arm-linux- CC=arm-linux-gcc \
 AR=arm-linux-ar NM=arm-linux-nm RANLIB=arm-linux-ranlib \
 CXX=arm-linux-g++ AS=arm-linux-as LD=arm-linux-ld \
 STRIP=arm-linux-strip BUILDCC=gcc
 BUILD_CC=gcc CC_FOR_BUILD=gcc ‘‘$@’’
```

私の chumby のファームウェアは 1.6[*6]なので、Wiki の firmware 1.6 の手順を実施します。

- Hacking Linux for chumby - ChumbyWiki[*7]

また、カーネルビルド用の環境には次の Debian パッケージは最低限入れておく必要があります。

- make
- gcc
- libncurses5-dev
- libncursesw5-dev
- zip

また、chumby 用のカーネルソースコードと、chumby へ新しいカーネルをインストールするためにアライメントする Perl スクリプトをそれぞれダウンロードしておきます。

- linux-2.6.16-chumby-1.6.0.tar.gz[*8]
- align.pl[*9]

カーネルソースを展開し、make menuconfig で ext2 を組み込みます[*10]。

```
$ cd
$ mkdir kernel
$ cd kernel
$ cp ~/{align.pl,linux-2.6.16-chumby-1.6.0.tar.gz} ./
$ tar zxf linux-2.6.16-chumby-1.6.0.tar.gz
$ cd linux-2.6.16-chumby-1.6.0
$ ARCH=arm BOARD=mx21ads CROSS_COMPILE=arm-linux- \
 make menuconfig
$ ARCH=arm BOARD=mx21ads CROSS_COMPILE=arm-linux- make
$ perl ../align.pl arch/arm/boot/zImage
$ zip k1.bin.zip arch/arm/boot/zImage
```

これで、kernel/linux-2.6.16-chumby-1.6.0/ディレクトリ直下に、k1.bin.zip が生成されます。これを USB メモリの vfat 領域にコピーします。

```
$ sudo mount -t vfat /dev/sda1 /media/usb
$ sudo mkdir /media/usb/update2
$ sudo cp -i k1.bin.zip /media/usb/update2/
$ sudo umount /media/usb
```

chumby を special option mode で起動し、kernel をアップデートします。

1. chumby の電源を OFF にした状態で USB メモリを挿します。
2. タッチスクリーンを押したまま、電源を入れる。途中で押したままにすると special option mode になるよ、と表示されるのでそのまま押しつづけます。
3. special option mode のメニュー画面で”install updates” をクリックします。
4. “Install from USB flash drive” をクリックすると、kernel がアップデートされ、自動的に再起動されます。

7.2.3 USB メモリへの Debian インストール

次に、USB メモリに Debian をインストールしますが、USB メモリには chumby 自体の設定を行うためのファイルを置く vfat 領域も必要なので、fdisk コマンドで/dev/sda1 を vfat、/dev/sda2 を Linux 用領域を作り、mkfs コマンドでファイルシステムを作成しておきます。

```
$ sudo fdisk /dev/sda
$ sudo mkfs.vfat /dev/sda1
$ sudo mke2fs    /dev/sda2
```

この sda2 の方に、Debian をインストールします。chumby は、EABI(armel) ではなく、OABI(arm) であるため、arm 版のインストーラを用意する必要があります。しかし、Qemu では armel のサブアーキテクチャversatile しかサポートしていないため、arm 版の kernel イメージを起動させることしかできません。なので、今回は、同じ OABI のアットマークテクノ社の armadillo-9 用に公開されている Debian Etch イメージを利用しました。下記リンク先から、5 つの tar ボールを全てダウンロードします。

[*6] 確認方法は、ssh で chumby にログイン後、chumby_version -f を実行します。
[*7] http://wiki.chumby.com/mediawiki/index.php/Hacking_Linux_for_chumby
[*8] http://files.chumby.com/source/ironforge/build733/linux-2.6.16-chumby-1.6.0.tar.gz
[*9] http://files.chumby.com/source/ironforge/build396/align.pl
[*10] モジュールにしても構いませんが、その場合は手動でカーネルモジュールをロードする必要があるので面倒です。

- debian directory - Armadillo 開発者サイト[11]

ext2 領域をマウントし、tar ボールを展開します。

```
$ sudo mount /dev/sda2 /mnt
$ cd /mnt
$ tar zxf ~/debian-etch-a9-1.tgz
$ tar zxf ~/debian-etch-a9-2.tgz
$ tar zxf ~/debian-etch-a9-3.tgz
$ tar zxf ~/debian-etch-a9-4.tgz
$ tar zxf ~/debian-etch-a9-5.tgz
```

chumby の電源を落とし、この USB メモリを挿して電源を入れると、自動的に ext2 領域もマウントされ、この下の領域のバイナリも正常に実行できます。

7.2.4 USB 環境への chroot 準備

USB メモリの Debian 環境に chroot し、そしてその環境下で Etch から Lenny にバージョンアップさせます。まず、ssh でログインし、/proc、/dev、devpts をバインドさせます。

```
chumby:~# mount -o bind /proc /mnt/usb2/proc
chumby:~# mount -o bind /dev  /mnt/usb2/dev
chumby:~# mount -t devpts devpts /mnt/usb2/dev/pts/
chumby:~# chroot /mnt/usb2
chumby:/1 df
Filesystem    1K-blocks      Used Available Use% Mounted on
/dev/hda1       1373548    181321   1118946  14% /
tmpfs           1373548    181321   1118946  14% /lib/init/rw
sysfs           1373548    181321   1118946  14% /sys
udev            1373548    181321   1118946  14% /dev
tmpfs           1373548    181321   1118946  14% /dev/shm
devpts          1373548    181321   1118946  14% /dev/pts
```

apt line を etch から lenny に書き換え、バージョンアップを行うと、問題なくアップグレードできるはずです。

次に、chroot の Debian で、ssh を自動起動させるため、次の設定を行います。22/tcp は chumby 自体の sshd が使うので別のポートを割り当てる方が良いでしょう。

/mnt/usb2/etc/ssh/sshd_config (一部抜粋)

```
Port 2222
(snip)
PermitRootLogin no
StrictModes yes
RSAAuthentication yes
PubkeyAuthentication yes
(snip)
PermitEmptyPasswords no
ChallengeResponseAuthentication no
PasswordAuthentication no
(snip)
```

次に、chumby 側の設定。USB メモリに配置した Widget をロードさせる手順の応用で、6.2.3 で作成した vfat 領域の直下に、以下の内容で debugchumby というファイル名でスクリプトを作成します。

```
#!/bin/bash

mount -o bind /proc /mnt/usb2/proc
mount -o bind /dev  /mnt/usb2/dev
mount -t devpts devpts /mnt/usb2/dev/pts/
chmod 666 /mnt/usb2/dev/null
chroot /mnt/usb2 /bin/hostname chumby
chroot /mnt/usb2 /usr/sbin/sshd
```

これで、次回以降、自動的に chroot 環境の Debian の sshd が 2222/tcp で起動するようになります。

7.3 OSC 会場での展示準備

6.1 でも書きましたが、Chumny はインターネットに繋がらないと単なる時計です。理由は、起動時に chumby.com から controlpanel.swf という管理コンソールの flash ファイルや、その他自分で設定している Widget をダウンロードしてくるためです。そこで、簡単な Widget を作成し、画面上はそれを表示しつつ、スタンドアロン環境でも有線 LAN を介して SSH でログインできるようにします。

7.3.1 前提条件

環境構築に加えて必要なもの

- mtasc パッケージ
- USB-Ethernet 変換アダプタ
- クロスケーブル
- 展示用の PC

7.3.2 Widget 作成

chumby の Widget は Flash です。Debian を使っているので Widget はもちろんテキストエディタで ActionScript を書いて、フリーソフトウェアでコンパイルします。今回の展示で表示させていた Widgit のソースコードは以下のとおりです。

[11] http://armadillo.atmark-techno.com/filebrowser/armadillo-9/debian

```
class DisplayDebian {
  public static function main(mc:MovieClip):Void
  {
    var app = new DisplayDebian(mc);
  }

  public function DisplayDebian(mc:MovieClip)
  {
    mc.createEmptyMovieClip(``image'',
    mc.getNextHighestDepth());
    var image:MovieClip = mc.image;
    var imageArr:Array = [ ``./openlogo.png'' ];
    image._xscale= 100;
    image._yscale= 100;
    image._x= 69;
    image._y= 0;
    image.loadMovie(imageArr[0]);

    var textField:TextField = mc.createTextField('textField',
    mc.getNextHighestDepth(), 15, 10, 320, 240);
    var fmt:TextFormat = new TextFormat('', 24, 0x000000);
    textField.text = ' 東京エリア Debian 勉強会\n\n\n' +
    ' 次回は 3/21, 東大で開催予定';
    textField.setTextFormat(fmt);
  }
}
```

これを Hoge.as として保存し、同じディレクトリに openlogo.png[*12]を配置します。

そして以下のワンライナー (ascompile.sh) の引数として渡し、コンパイルします。

```
$ ./ascompile.sh Hoge.as
```

ワンライナー ascompile.sh は以下のように記述します。

```
#!/bin/bash
test -z $1 && exit 1
mtasc -swf `basename $1 .as`.swf -main $1 -header \
 320:240:12 -version 8
```

コンパイルすると、Hoge.swf という flash ファイルができます。この Hoge.swf を Widget として読み込むために、profile.xml という名前で設定します。

```
<?xml version=''1.0'' encoding=''utf-8'' ?>
<profile>
 <widget_instances>
  <widget_instance id=''1''>
   <widget>
    <name>Debian Logo</name>
    <description>Debian GNU/Linux Logo</description>
    <version>1.0</version>
    <mode time=''30'' mode=''timeout'' />
    <access sendable=''false'' deletable=''false''
     access=''private'' virtualable=''false'' />
    <user username=''Kouhei Maeda'' />
    <thumbnail href=''file:///mnt/usb/openlogo.png''
     contenttype=''image/png'' />
    <movie href=''file:///mnt/usb/Hoge.swf''
     contenttype=''application/x-shockwave-flash'' />
   </widget>
   <access access=''private'' />
   <mode time=''30'' mode=''timeout'' />
   <widget_parameters>
    <widget_parameter>
     <name>auther1</name>
     <value>Kouhei</value>
    </widget_parameter>
    <widget_paramter>
     <name>auther2</name>
     <value>Maeda</value>
    </widget_parameter>
   </widget_parameters>
  </widget_instance>
 </widget_instances>
</profile>
```

この profile.xml および、Hoge.swf と openlogo.png を USB メモリの vfat 領域直下にコピーします。これで、起動時に USB メモリからこの Widget が読み込まれるようになります。

7.3.3 スタンドアロンでの起動設定

次に、スタンドアロンで先ほどの Widget が読み込まれ、ssh でログインできるようにします。基本的にはフォーラム[*13]の内容に従って行えば問題ありません。まず、chumby-offline.zip[*14] をダウンロードします。展開すると offline-howto.txt というドキュメントがあるので、これに従い設定します。profile.xml は既に 6.3.2 で作成していますので、これを利用してください。USB メモリの vfat 領域の直下に以下のファイルをコピーします。

- chumby-offline.zip を展開してできる offline ディレクトリ以下にある www/ディレクトリ[*15]
- chumby の/usr/widgets/controlpanel.swf

USB メモリの vfat 領域の直下の debugchumby を次のように書き換えます。

[*12] Debian.org のサイトのロゴ http://www.debian.org/logos/openlogo.xcf.gz を利用。そのままでは画像サイズが合わないため、gimp で高さ 240 ピクセルに収まるようにリサイズしています。
[*13] http://forum.chumby.com/viewtopic.php?pid=12258
[*14] http://stud3.tuwien.ac.at/~e9825447/chumby-offline.zip
[*15] offline-howto.txt に従い、設定済みであること。

```
#!/bin/bash

killall httpd
/usr/sbin/httpd -h /mnt/usb/www
cp /mnt/usb/www/hosts.offline /psp/hosts
#cp /mnt/usb/www/hosts.online /psp/hosts

mount -o bind /proc /mnt/usb2/proc
mount -o bind /dev  /mnt/usb2/dev
mount -t devpts devpts /mnt/usb2/dev/pts/
chmod 666 /mnt/usb2/dev/null
chroot /mnt/usb2 /bin/hostname chumby
```

なお、オフラインモードからオフラインモードに戻す場合は、killall から 3 行をコメントアウトし、hosts.online をコピーする行を有効にしてください。

そして最後に、USB-Ethernet アダプタを使い、有線 LAN 経由でアクセスできるように設定します。今までと同じく、vfat 領域に userhook1 というファイルを作成します。内容は以下のとおりです。

```
#!/bin/sh

USE_DHCP=0
IPADDR=192.168.3.10
NETMASK=255.255.255.0
GATEWAY=192.168.3.1

/sbin/insmod /drivers/usbnet.ko
/sbin/insmod /drivers/pegasus.ko

ifconfig rausb0 127.0.0.1

if [ $USE_DHCP == 1 ]\daggerhen
   udhcpc -t 5 -n -p /var/run/udhcpc.eth0.pid -i eth0
else
   /sbin/ifconfig eth0 $IPADDR netmask $NETMASK
   /sbin/route add default gw $GATEWAY eth0
fi
```

ちなみに、今回は BUFFALO の LUA2-TX を使っています。*16

以上で、まったくインターネットを使えない環境でも、有線 LAN 経由で SSH 接続かつ、液晶モニタ上に USB に入れた Widget を稼働させることができるようになります。

7.4 残作業

以下が残っていますが、気が向いたらやるかもしれないし、やらないかもしれません。

- debootstrap で Lenny arm 版を作ってみて chroot 環境として使えるか？
- chroot 環境からフレームバッファを乗っ取る。
- ブートローダをハックして、直接 USB ブートさせる。

7.5 おまけ：今月のヨメ八苦

2 月はこの OSC 準備と別件で、ヨメよりも早く帰ってきても晩飯を作らなかった（作れなかった）のと、毎週水曜に Hack Meeting に参加するようになったため、めっちゃ不機嫌になりました。コワ。方々のアドバイスを頂き、3 月現在は "すい〜つ" でなだめています。

7.6 参考文献

- chumby で遊ぼう！
  6.3.2 の Hoge.as は本書の p.48-49 の Main.as を、profile.xml は p.53 の USB メモリ用 profile.xml の例を、6.3.3 の userhook1 は p.156 の userhook1 を引用しました。（一部パラメータを変更）
  - ISBN：978-4-7973-5039-5
  - 著者：米田 聡
  - 発行所： ソフトバンククリエイティブ株式会社
- Using Chumby offline - Chumbysphere Forum
  - `http://forum.chumby.com/viewtopic.php?pid=12258`
- Hacking Linux for chumby - ChumbyWiki
  - `http://wiki.chumby.com/mediawiki/index.php/Hacking_Linux_for_chumby`

*16 これのデバイスドライバは pegasus.ko です。

Debian 勉強会資料

2009 年 3 月 21 日　初版第 1 刷発行

東京エリア Debian 勉強会 （編集・印刷・発行）