Armadillo 実践開発ガイド

～組み込み Linux の導入から製品化まで～

第 2 部

Version 2.0.0
2010/12/24

Armadillo 実践開発ガイド: 〜組み込み Linux の導入から製品化まで〜: 第 2 部

株式会社アットマークテクノ

060-0035 札幌市中央区北 5 条東 2 丁目 AFT ビル 6F
TEL 011-207-6550 FAX 011-207-6570

Version 2.0.0
2010/12/24

目次

図目次

表目次

1. はじめに

第 1 部では、Armadillo を使った組み込みシステムを構築する方法の全体像について説明を行いました。第 2 部では、システムの開発段階で役に立つ、より実践的な事柄について説明します。

まず、「2. 開発環境の整備と運用」で、開発環境 ATDE3 について第 1 部で説明しきれなかった詳細な設定方法や、パッケージ管理などの運用方法について説明します。また、ATDE3 を使わずに開発環境を構築する方法についても言及します。

「3. Linux システムの仕組みと運用、管理」では、Linux システム特有の用語や様々なコマンドについて、網羅的に紹介します。分からない用語やコマンドがあった場合のリファレンスとして活用してください。

続いて、「4. Armadillo 上に Debian GNU/Linux を構築する」で、Armadillo のルートファイルシステムを Debian GNU/Linux にする方法について説明します。Armadillo の標準のディストリビューションは Atmark Dist ですが、Debian GNU/Linux を使うと様々なアプリケーションプログラムを簡単にインストールできるなど、開発効率が高まります。そのため、開発段階では Debian GNU/Linux を使うことを推奨します。

プログラミングの最初の話題として、「5. シェルスクリプトプログラミング」を取り上げます。Linux システムに不慣れな方でも読めるように、基本的な文法から丁寧に説明をおこないます。

「6. C 言語による実践プログラミング」は、C 言語経験者を対象とした内容になっています。そのため、C 言語の文法に関する説明は行いません。Linux システムや開発環境に依存した独特な部分や、組み込みシステムでアプリケーションプログラムを開発する際に問題となることにフォーカスして説明します。

「7. 詳解 Atmark Dist」では、Atmark Dist の仕組みや動作について説明します。Debian GNU/Linux を使うと開発段階では便利ですが、組み込みシステムとしてみた場合、冗長な部分もあります。Atmark Dist を使うことで製品に最適化されたシステムを作ることができます。

最後に、「8. 組み込みシステム構築の定石」に Armadillo を使って組み込みシステムを構築する上での、組み込みまたは Armadillo 特有のノウハウについて、まとめました。この章に書いてあることは他の本には書いていませんが、とても重要な部分です。

1.1. 対象読者

本書が主な対象読者としているのは、Armadillo を使って組み込みシステムを開発したいと考えているソフトウェア開発者です。ソフトウェア開発者は、少なくとも C 言語での開発経験が必要です。Linux や Armadillo を使用した開発の経験が少ない場合や開発の全体像を把握していない場合は、第 1 部から読むことをお勧めします。

1.2. 表記方法

本書で使用している表記方法について説明します。

1.2.1. 使用するフォント

フォントは以下のものを使用します。

表 1.1 使用するフォント

フォント例	使用箇所
本文中のフォント	本文
等幅	コマンド入力例やソースコード
太字	ユーザーが入力する文字
斜体	状況によって置き換えられるもの
下線	キー入力

1.2.2. コマンド入力例の表記方法

1.2.2.1. Linux システムの場合

Linux システムでの端末からのコマンド入力例は、以下のように表記します。

```
[PC ~/]$ ls
```

図 1.1 コマンド入力表記例(Linux システム)

「[PC ~/]$」の部分をプロンプトと呼びます。プロンプトに続いてコマンドを入力してください。

「PC」の部分は、コマンドを実行する環境によって使い分けます。実行環境には、以下のものがあります。

表 1.2 コマンドの実行環境と対応する表記

表記	実行環境
PC	作業用 PC
ATDE	ATDE(Atmark Techno Development Environment[1])
armadillo	Armadillo(Atmark Dist で作成したユーザーランドの場合)
darmadillo	Armadillo(ユーザーランドが Debian GNU/Linux の場合)

[1]アットマークテクノ社製品用の開発環境

「~/」の部分は、カレントディレクトリのパスを表します。

「$」の部分は、コマンドを実行するユーザーの種類によって使い分けます。ユーザーの種類には、以下の二種類があります。

表 1.3 ユーザーの種類と対応する表記

表記	権限
#	特権ユーザー
$	一般ユーザー

1.2.2.2. 保守モードの場合

Armadillo を保守モードで起動した場合のコマンド入力例は以下のように表記します。

```
hermit> info
```

図 1.2 コマンド入力表記例(保守モード)

保守モードでは、プロンプトは「hermit>」となります。プロンプトに続いてコマンドを入力してください。

1.2.3. コラムの表記方法

本書では、随所にコラムを記載しています。コラムの内容によって、以下の表記を用います。

メモ

用語の説明や補足的な説明は、このアイコンで示します。

ヒント

知っていると便利な情報は、このアイコンで示します。

注意

ユーザーの注意が必要な情報は、このアイコンで示します。このアイコンが付いているコラムの内容に従わない場合、ハードウェアやシステムを破壊したり、以降の作業に支障をきたす場合があります。再度、ご確認ください。

注意: 本書の内容を実践する前に

ご使用になる製品のマニュアル(ハードウェアマニュアル、ソフトウェアマニュアル、その他関連資料)をよく読み、それらに記述されている注意事項に従って正しく安全にお使いください。

1.3. サンプルソースコード

本書で紹介するサンプルソースコードは、http://download.atmark-techno.com/armadillo-guide/source/ からダウンロードできます。サンプルソースコードは、MIT ライセンス[1]の下に公開します。

1.4. 困った時は

本書を読んでわからなかったり困ったことがあった際は、ぜひ Armadillo 開発者サイト[2]で情報を探してみてください。本書には記載しきれていない FAQ や Howto が掲載されています。

[1] http://opensource.org/licenses/mit-license.php
[2] http://armadillo.atmark-techno.com

Armadillo 開発者サイトでも知りたい情報が見つからない場合は、Armadillo シリーズに関する話題を扱う「Armadillo メーリングリスト」[3]で質問してみてください。多くのユーザーや開発者が参加しているので、知識のある人や同じ問題で困ったことがある人から情報を集めることができます。

メーリングリストに参加するための心構え

Armadillo メーリングリストには、現在までに数百人のユーザーが参加しています。メーリングリストに送られたメールは、メーリングリスト参加者すべてに送られます。それらのメールはアーカイブとして保存され、Web 上で誰でも閲覧可能な状態になり、過去に投稿されたメールを検索することもできます[4]。

メーリングリストには多くの人が参加していますので、そこにはマナーが存在します。一般的な対人関係と同様に、受け取り手に対して失礼にならないよう配慮はすべきです。とはいえ、技術的に簡単なものであるとか、ちょっとした疑問だからという理由で、投稿をためらう必要はありません。Armadillo に関係のある内容であれば、難しく考えることなく気軽にお使いください。

適切な質問をすると、適切なアドバイスが得られる可能性が高まります。その逆もまた然りです。メーリングリストに不慣れな方は、質問する前に「技術系メーリングリストで質問するときのパターン・ランゲージ」[5] あたりをご一読されることをお勧めします。

1.5. お問い合わせ先

本書に関するご意見やご質問は、Armadillo メーリングリスト[6] にご連絡ください。

何らかの事情でメーリングリストが使えない場合は、以下にご連絡ください。

```
株式会社アットマークテクノ
〒 060-0035 北海道札幌市中央区北 5 条東 2 丁目 AFT ビル 6F
電話         011-207-6550
FAX          011-207-6570
電子メール   sales@atmark-techno.com
```

1.6. 商標

Armadillo は、株式会社アットマークテクノの登録商標です。その他の記載の商品名および会社名は、各社・各団体の商標または登録商標です。™、®マークは省略しています。

1.7. ライセンス

[3]http://armadillo.atmark-techno.com/mailinglists
[4]投稿者のメールアドレスのみ、スパムメール対策として Web 上では秘匿されます。
[5]結城浩氏によるサイトより http://www.hyuki.com/writing/techask.html
[6]http://armadillo.atmark-techno.com/mailinglists

図 1.3 クリエイティブコモンズライセンス

1.8. 謝辞

Armadillo や ATDE で使用しているソフトウェアの多くは Free Software/Open Source Software で構成されています。Free Software/Open Source Software は世界中の多くの開発者や関係者の貢献によって成り立っています。この場を借りて感謝の意を表します。

2. 開発環境の整備と運用

Armadillo 用の標準開発環境を ATDE(Atmark Techno Development Environment)といいます。ATDE は仮想マシン上で動作する Linux デスクトップ環境(Debian GNU/Linux)で、開発に必要なソフトウェアー式がプリインストールされています。ATDE は VMware イメージとして提供されているため、作業用 PC の OS が Windows であるか Linux であるかを問わず、VMware Player を実行できる環境であれば、ATDE を実行することができます。

本書執筆時点での Armadillo-400 シリーズ用の標準開発環境は、ATDE3 となっています。ATDE3 のベースは Debian GNU/Linux 5.0 (コードネーム "lenny")です。ATDE3 のインストール及び基本的な設定方法については、第 1 部「開発環境の構築」を参照してください。

この章では、第 1 部で触れなかった ATDE3 の設定方法や運用方法について説明をおこないます。また、ATDE3 と同じ環境を別の Linux PC に構築する方法についても説明します。

2.1. ATDE3 で USB メモリを使用する

この章では、作業用 PC に接続した USB メモリを ATDE3 で使用する方法について説明します。シリアルポートや共有フォルダを使う方法は、第 1 部を参照してください。

2.1.1. ホスト OS が Windows の場合

本章では、作業用 PC の OS が Windows の場合について説明します。作業用 PC の OS が Linux の場合は、「2.1.2. ホスト OS が Linux の場合」を参照してください。

まず、USB メモリを作業用 PC に接続してください。USB メモリが作業用 PC で認識されると、VMware Player の「仮想マシン」-「取り外し可能デバイス」メニューに「デバイス名[1]」が追加されます。「接続(ホストから切断)」と表示されている場合は、メニューを選択してください[2]。そうすると、VMware Player 側(ATDE3)で USB メモリを使用できるようになります。

[1] USB メモリの名前が表示されます。
[2] 「切断(ホストに接続)」と表示されている場合は、すでに USB メモリが VMware Player に接続されていますので、この手順は不要です。

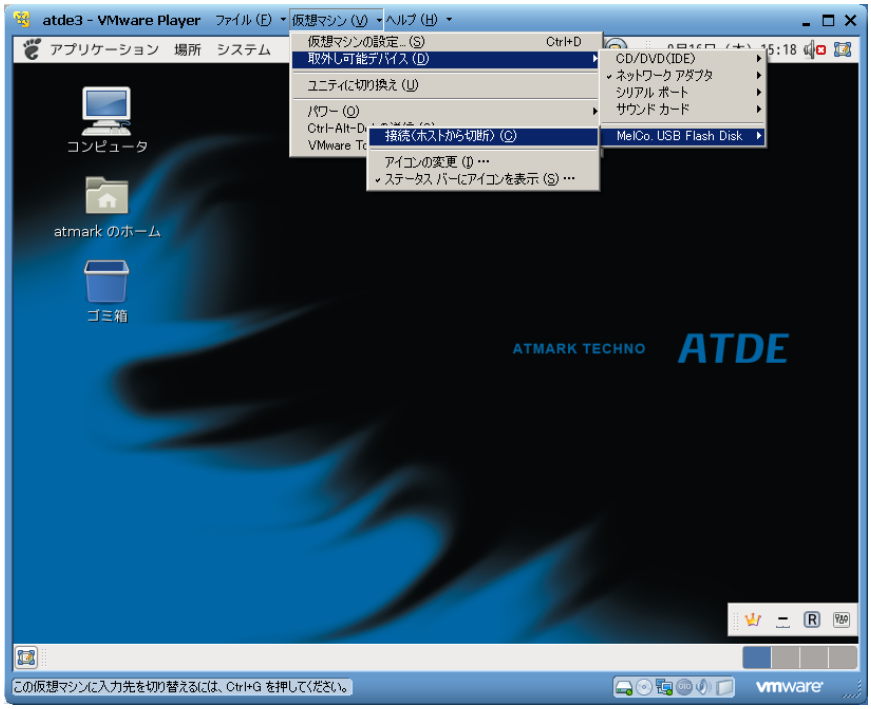


図 2.1 USB メモリの接続

しばらくして ATDE3 が USB メモリを認識すると、USB メモリは自動で/media/disk ディレクトリにマウントされます。また、以下のようにデスクトップに USB メモリのアイコンが作成され、Nautilus(ファイルマネージャ[3])が起動します。

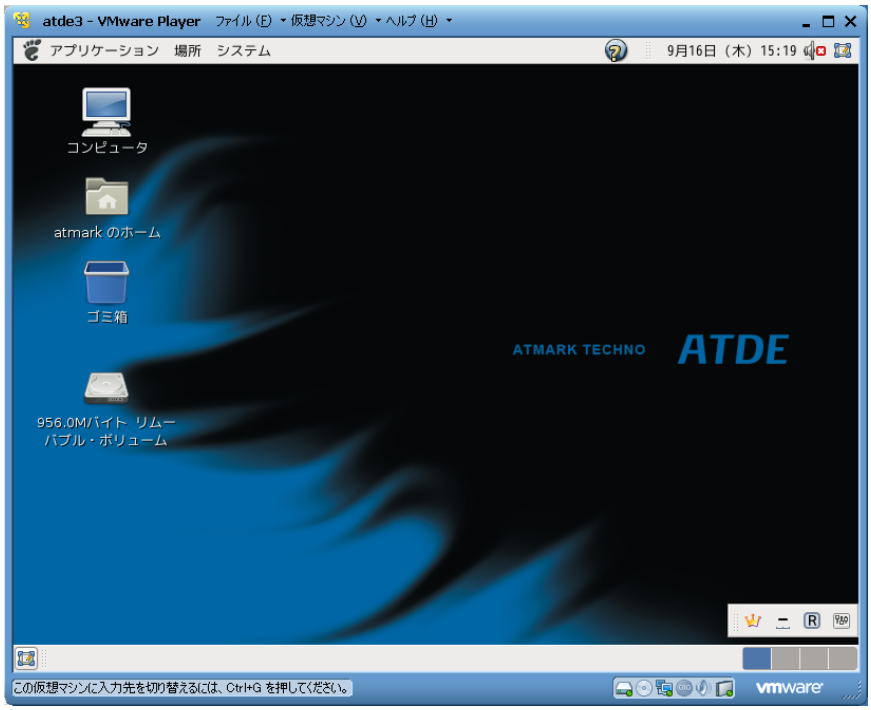


図 2.2 USB メモリのアイコン

[3]ファイルシステムを扱うためのユーザーインターフェースを提供するソフトウェア。ファイラーと呼ばれることもあります。Windows ではエクスプローラなどがあります。

USB メモリを切断する

VMware Player 側で USB メモリを使用している間は、作業用 PC では使用できません。作業用 PC で USB メモリを使用する場合は、一度 VMware Player から USB メモリを切断する必要があります。

VMware Player から USB メモリを切断するには VMware Player の「仮想マシン」-「取り外し可能デバイス」-「デバイス名」-「切断(ホストに接続)」メニューを選択してください。

2.1.2. ホスト OS が Linux の場合

本章では、作業用 PC の OS が Linux の場合について説明します。作業用 PC の OS が Windows の場合は、「2.1.1. ホスト OS が Windows の場合」 を参照してください。

まず、USB メモリを作業用 PC に接続してください。USB メモリが作業用 PC で認識されると、VMware Player の「VM」-「Removable Devices」メニューに「デバイス名[4]」が追加されます。「Connect」と表示されている場合は、メニューを選択してください[5]。そうすると、VMware Player 側(ATDE3)で USB メモリを使用できるようになります。

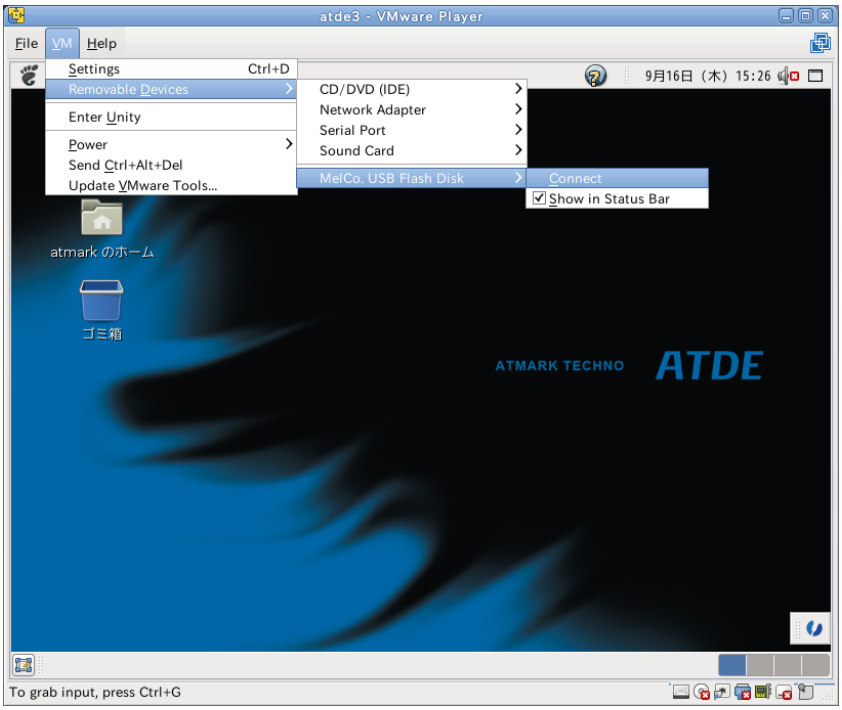


図 2.3 USB メモリの接続

[4]USB メモリの名前が表示されます。
[5]「Disconnect」と表示されている場合は、すでに USB メモリが VMware Player に接続されていますので、この手順は不要です。

しばらくして ATDE3 が USB メモリを認識すると、USB メモリは自動で/media/disk ディレクトリにマウントされます。また、以下のようにデスクトップに USB メモリのアイコンが作成され、Nautilus(ファイルマネージャ[6])が起動します。

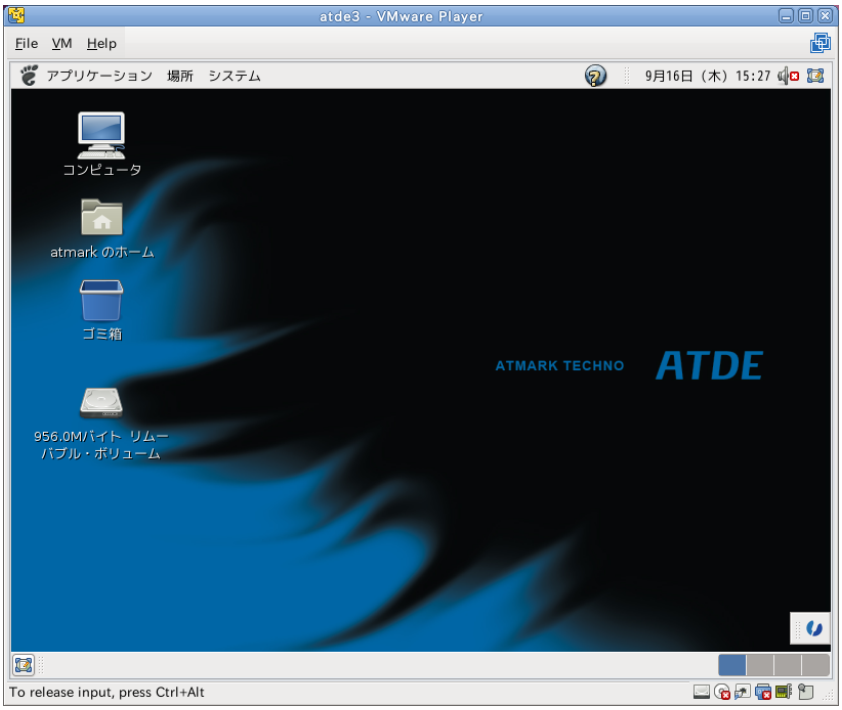


図 2.4 USB メモリのアイコン

USB メモリを切断する

VMware Player 側で USB メモリを使用している間は、作業用 PC では使用できません。作業用 PC で USB メモリを使用する場合は、一度 VMware Player から USB メモリを切断する必要があります。

VMware Player から USB メモリを切断するには VMware Player の「VM」-「Removable Devices」-「デバイス名」-「Disconnect」メニューを選択してください。

2.2. ATDE3 にソフトウェアを追加する

Debian GNU/Linux ではアプリケーションプログラムやライブラリなどのソフトウェアの管理は Debian パッケージと呼ばれるパッケージ単位で行います。この章では、Debian パッケージを検索し、それを ATDE3 へインストールする方法について説明します。

2.2.1. ソフトウェアが含まれるパッケージを検索する

まず、インストールしたいアプリケーションプログラムやライブラリが含まれるパッケージのパッケージ名を知る必要があります。パッケージ名を調べる一般的な方法には、以下のものがあります。

[6]ファイルシステムを扱うためのユーザーインターフェースを提供するソフトウェア。ファイラーと呼ばれることもあります。Windows ではエクスプローラなどがあります。

- Debian のパッケージページ[7]から検索する。
- **apt-cache search** を使用しパッケージを検索する。
- すでにインストールされているファイルからパッケージ名を検索する。

本章ではこれらの方法について、説明します。

2.2.1.1. Debian パッケージページで検索する

debian.org(Debian GNU/Linux の公式サイト)のパッケージページで Debian パッケージの検索をすることができます。Debian のパッケージページでのパッケージの検索方法としては以下の方法があります。

- パッケージ名
- パッケージ説明文
- ソースパッケージ名
- パッケージに含まれるファイルのパス

それぞれの検索方法でパッケージを選択し、ダウンロードすることができます。

Debian GNU/Linux のバージョンとアーキテクチャ

Debian GNU/Linux には安定版(stable)、テスト版(testing)、不安定版(unstable)、旧安定版(oldstable)というリリースが存在します。それぞれ用途・更新頻度・サポート体制が違います。

ATDE3 では、2010 年 9 月現在の安定版(stable)である Debian GNU/Linux 5.0(コードネーム "lenny")をベースに開発環境を構築しています。

また、Debian では各種アーキテクチャ用のパッケージが提供されています。

Intel x86 系 CPU 用のアーキテクチャ名は i386 です。ATDE3 では、このアーキテクチャを使用します。

ARM CPU 用には、arm と armel の二つのアーキテクチャがあります。それぞれ ABI が異なります。arm が OABI 用、armel が EABI 用のアーキテクチャです。Armadillo-400 シリーズは EABI ですので、armel アーキテクチャ用のパッケージを使用します。

2.2.1.2. apt-cache コマンドで検索する

apt-cache コマンドの引数に search を付けて実行することでパッケージの検索ができます。パッケージ名やパッケージの説明に対して検索します。パッケージ名やパッケージの説明に対して検索できます。

例えば「coreutils」という **ls** コマンドや **cp** コマンドなどの基本的なコマンドが含まれるパッケージを検索する場合は以下のコマンドを実行します。

[7]http://www.debian.org/distrib/packages

```
[ATDE ~]$ apt-cache search coreutils
realpath - Return the canonicalized absolute pathname
python-sepolgen - A Python module used in SELinux policy generation
policycoreutils - SELinux コアポリシーユーティリティ
coreutils - GNU コアユーティリティ
bsdmainutils - FreeBSD 由来のユーティリティ集
```

図 2.5 apt-cache search によるパッケージの検索

一番左に表示されているものが、パッケージ名です。

apt-cache コマンドには他にもいろいろな機能が備わっています。詳しくは **man apt-cache** を参照してください。

2.2.1.3. すでにインストールされているパッケージから逆引きする

ATDE3 にはインストールされていないソフトウェアが、Debian GNU/Linux 5.0 などがインストールされている作業用 PC には入っている場合があります。この章では、ファイル名やコマンド名は分かるが、パッケージ名が分からないときに便利なコマンドを紹介します。

インストールされているファイル名や、コマンド名からパッケージを調べるには **dpkg** コマンドに-S オプションを付けて使用します。ここでは例として **ls** コマンドがどのパッケージに含まれているかを調べます。

```
[PC ~]$ which ls
/bin/ls
[PC ~]$ dpkg -S /bin/ls
coreutils: /bin/ls
```

図 2.6 ls コマンドが含まれているパッケージを調べる

which コマンドを使って **ls** コマンドのパスを調べた結果を、**dpkg** コマンドの引数として指定しています。**dpkg -S** はファイルのパスで検索してくれます。ファイル名ではなく、パスを使用したほうが検索結果が少なく便利です。

dpkg -S /bin/ls を実行した結果から、**ls** コマンドは coreutils パッケージに含まれていることが分かります。

2.2.2. パッケージをインストールする

インストールしたいパッケージのパッケージ名が分かれば、**apt-get install** でパッケージをインストールできます。**apt-get install** は、指定されたパッケージ名のパッケージをダウンロードし、インストールまでを行います。

例として、FTP クライアントの lftp パッケージをインストールする場合、「図 2.7. apt-get install によるパッケージのインストール」のようになります[8]。

[8]第 1 部の手順をすべて実行している場合はすでに lftp がインストールされています。

```
[ATDE ~]$ sudo apt-get install lftp
[sudo] password for atmark:
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています
状態情報を読み取っています... 完了
以下のパッケージが新たにインストールされます:
  lftp
アップグレード: 0 個、新規インストール: 1 個、削除: 0 個、保留: 0 個。
587kB 中 0B のアーカイブを取得する必要があります。
この操作後に追加で 1540kB のディスク容量が消費されます。
取得:1 http://ftp.jp.debian.org lenny/main lftp 3.7.3-1+lenny1 [587kB]
587kB を 0s で取得しました (789kB/s)
未選択パッケージ lftp 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 92123 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。)
(.../lftp_3.7.3-1+lenny1_i386.deb から) lftp を展開しています...
man-db のトリガを処理しています ...
lftp (3.7.3-1+lenny1) を設定しています ...
```

図 2.7 apt-get install によるパッケージのインストール

apt-get コマンドにはいろいろな機能が備わっています。詳しくは **man apt-get** を参照してください。

2.3. ATDE3 にクロス開発用ライブラリをインストールする

Atmark Dist に含まれないアプリケーションやライブラリをビルドする際に、付属 DVD やダウンロードサイトには用意されていないライブラリパッケージが必要になることがあります。ここでは、クロス開発用ライブラリパッケージの作成方法およびそのインストール方法を紹介します。

2.3.1. dpkg-cross コマンドを使用する

まず、作成したいクロス開発用パッケージの元となるライブラリパッケージを取得します。取得するパッケージは、アーキテクチャをターゲットに、Debian ディストリビューションのバージョンを開発環境に合わせる必要があります。Armadillo-400 シリーズでは、アーキテクチャは armel、Debian ディストリビューションのバージョンは lenny (2010 年 9 月現在の安定版)になります。

例えば、libjpeg62 の場合、ダウンロードするファイルは「libjpeg62_*[version]*_armel.deb」になります。Debian パッケージは、Debian のパッケージページから検索して取得することができます。

取得したライブラリパッケージをクロス開発用に変換するには、**dpkg-cross** コマンドを使用します。

```
[ATDE ~]$ dpkg-cross --build --arch armel libjpeg62_[version]_armel.deb
[ATDE ~]$ ls
libjpeg62-armel-cross_[version]_all.deb libjpeg62_[version]_armel.deb
```

--build オプションはクロス開発用に変換された Debian パッケージを作成することを意味します。--arch オプションにはアーキテクチャを指定します。最後の引数には変換したいライブラリパッケージを指定します。

dpkg-cross はクロス開発用のライブラリパッケージパッケージの「libjpeg62-armel-cross_*[version]*_all.deb」を作成します。ファイル名の「armel-cross」は armel クロス開発用を意味し、「_all」はすべてのアーキテクチャの開発用 PC にインストールすることができることを意味します。

次に、**dpkg** コマンドを-i オプションを付けて実行し、クロス開発用に変換したパッケージをインストールします。

```
[ATDE ~]$ sudo dpkg -i libjpeg62-armel-cross_[version]_all.deb
```

dpkg-cross --build --arch armel で作成したクロス開発用ライブラリパッケージをインストールすると、ファイルは/usr/arm-linux-gnueabi ディレクトリ以下にインストールされます。ライブラリファイルは/usr/arm-linux-gnueabi/lib ディレクトリに、ヘッダファイルは/usr/arm-linux-gnueabi/include ディレクトリに置かれます。

パッケージ名の最後が-dev のパッケージ

パッケージ名の最後が-dev になっているパッケージは開発用のものです。開発用パッケージにはヘッダーファイルなどが入っています。例えば libjpeg62 パッケージの開発用パッケージ名は、libjpeg62-dev になります。libjpeg62 を使用するソースコードをコンパイルする場合は、libjpeg62-dev パッケージも必要になります。

クロス開発を行う場合はこのようなケースが多いので、新しいクロス開発用パッケージをインストールする場合、開発用パッケージもインストールしておくのが良いでしょう。

アプリケーションの場合

アプリケーションのパッケージには、ライブラリファイルもヘッダーファイルも含まれていないので、**dpkg-cross** コマンドでクロス開発用パッケージに変換することができません[9]。

ARM 用の Debian パッケージに含まれる、アプリケーションの実行ファイルを Armadillo で動作させたい場合、まず、**dpkg -x** を実行して、パッケージに入っているファイルを取り出します。該当のファイルを探し、Armadillo のユーザーランドに配置するだけでアプリケーションを使用することができます。

このようにして配置したアプリケーションを使用する場合に、ライブラリや設定ファイルが必要な場合もあります。うまく動作しなかった場合は、パッケージの依存関係や、設定ファイルの有無を調べてください。

また、「4. Armadillo 上に Debian GNU/Linux を構築する」で紹介する手順を行ない、Armadillo 上に Debian GNU/Linux を構築することで Armadillo 上でも **apt-get** コマンドが使用できるようになります。

[9] --convert-anyway オプションを使用して、中身が空のダミーパッケージを作成することはできます。

2.3.2. apt-cross コマンドを使用する

apt-cross コマンドを使用すると、「2.3.1. dpkg-cross コマンドを使用する」で紹介した一連の作業を 1 つのコマンドで行うことができます。

```
[ATDE ~]$ apt-cross --arch armel --suite lenny --install libjpeg62
```

図 2.8 apt-cross コマンド

--arch オプションにはアーキテクチャを指定します。--suite オプションには Debian ディストリビューションのバージョンを指定します。--install オプションは取得/変換したパッケージをインストールすることを意味します。最後の引数には、パッケージ名を指定します。

2.4. ATDE3 と同じ環境を構築する

ATDE3 は仮想マシン上で動作するため、メンテナンスが簡単などのメリットがありますが、その分動作が遅いというデメリットもあります。Debian GNU/Linux 5.0 がインストールされた PC であれば、ATDE3 と同様の開発用環境を構築することもできます。ここでは、その手順について説明します。

まず、/etc/apt/sources.list.d/atmark.source.list に以下の内容を追加してください。

```
deb http://download.atmark-techno.com/debian lenny/
```

続いて、以下のコマンドを実行し、パッケージの情報の更新をしてください。

```
[PC ~]$ sudo apt-get update
```

最後に、a440-development-environment パッケージをインストールしてください。そうすると、Armadillo の開発に必要なパッケージが全てインストールされます。

```
[PC ~]$ sudo apt-get install a440-development-environment
```

3. Linux システムの仕組みと運用、管理

Linux システムを組み込みで使うには、通常の Linux システムと同様に、その運用、管理方法についても知っておく必要があります。本章では、Linux システムの基本的な運用、管理方法について、その背景となる仕組みも交えながら説明していきます。

3.1. コマンドの使い方を調べる

Linux システムには、便利なコマンドが数多く用意されています。ここでは、コマンドの使い方を調べる方法について紹介します。

使い方が分からないコマンドに遭遇したら、まず、コマンド自身のヘルプを見てみましょう。コマンドに--help オプションをつけて実行すると、多くの場合ヘルプを表示してくれます。

```
[ATDE ~]$ cat --help
使用法: cat [オプション] [ファイル]...
Concatenate FILE(s), or standard input, to standard output.

  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines
  -e                       equivalent to -vE
(後略)
```

図 3.1 --help オプションでコマンドの使用方法を調べる

コマンドが--help オプションをサポートしていない場合でも、使用方法(usage)を表示してくれることが多いでしょう。

```
[ATDE ~]$ cd --help
bash: cd: --: invalid option
cd: usage: cd [-L|-P] [dir]
```

図 3.2 コマンドの使用方法を調べる(--help オプションをサポートしていない場合)

より詳しく調べたい場合、オンラインマニュアル(man ページ)が参考になります。オンラインマニュアルを表示するコマンドは、**man** です。**man** コマンドに引数としてコマンド名を渡すと、指定したコマンドのマニュアルを表示します。

例えば、**cat** コマンドの使い方を調べるには、以下のようにします。

```
[ATDE ~]$ man cat
```

図 3.3 cat コマンドの使用方法を調べる

man コマンドは、man ページの表示はページャーと呼ばれるテキスト閲覧用の別のプログラムに任せます。ATDE3 では、ページャーとして **less** を使用します。**less** は、**vi** と同じような操作方法ができる

ページャーです。閲覧を終了するには、q キーを押してください。**less** の使用方法を調べるには、**man less** を実行してください。

man ページは、説明している内容によっていくつかのセクションに分かれています。セクションには以下のものがあります[1]。

1. 実行プログラムまたはシェルのコマンド
2. システムコール (カーネルが提供する関数)
3. ライブラリコール (システムライブラリに含まれる関数)
4. スペシャルファイル (通常 /dev に置かれている)
5. ファイルのフォーマットとその約束事。例えば /etc/passwd など
6. ゲーム
7. マクロのパッケージとその約束事。例えば man(7), groff(7) など
8. システム管理用のコマンド (通常は root 専用)
9. カーネルルーチン [非標準]

同じ名前で、複数のセクションに説明がある man ページもあります。例えば、write という名前のページは、**write** コマンドと write システムコールの二つあります。このような場合、**man write** コマンドを実行すると、write に対応するページを検索し、もっとも適切と判断したページのみを表示します。他のセクションにあるページを見たい場合は、セクション番号を指定して **man** コマンドを実行します。write システムコールの場合は、**man 2 write** となります。

man コマンドには、内容を検索する機能など様々な機能があります。例えば、**man -k write** とすると、man ページの要約文とページ名から、write にマッチするページの一覧をすべて表示します。**man** コマンドの詳細な使い方を調べるには、**man man** を実行してください。

英語版 man ページ

ATDE3 では、標準設定では言語設定が日本語になっています。そのため、日本語訳のある man ページは日本語で表示されます。

しかし、翻訳がいまいちな場合など、英語で記述されたページを見たい場合もあるでしょう。そのような場合、LANG=C man man のように、言語を一時的に設定して **man** コマンドを実行することで、英語版の man ページを見ることができます。

3.2. ユーザー管理

Linux システムは、マルチユーザー、マルチタスクなシステムです。一つのシステムに同時に複数のユーザーがログインすることもできますし、それぞれのユーザーが複数のタスク(プロセス)を実行することもできます。そのため、ユーザーの管理をおこなうことは、Linux システム管理の基本といえるでしょう。

[1] **man** コマンドの man ページより引用。

ユーザーには、それぞれ一意なユーザー名とユーザー ID が割り当てられます。これらの情報は、/etc ディレクトリにある passwd ファイルに記述されています。passwd ファイルには「:」区切りで、ユーザー名、パスワード、ユーザー ID、グループ ID、コメント、ホームディレクトリ、使用するシェルが順番に記述されています。passwd ファイルについての詳細は、**man 5 passwd** で参照することができます。

有効なユーザー名の一覧を表示するには、以下のようにします。

```
[ATDE ~]$ cut -d ':' -f 1 /etc/passwd
root
daemon
(中略)
atmark
messagebus
gdm
haldaemon
avahi
```

パスワード保存ファイル

パスワードは、暗号化されてファイルに保存されます。しかし、暗号文は時間をかければ解読可能ですので、すべてのユーザーから見える passwd ファイルに暗号化したパスワードを書いておくことは望ましくありません。

そのため、暗号化したパスワードは/etc ディレクトリにある shadow ファイルに保存されています。shadow ファイルは特権ユーザーしか見ることができないようにパーミッションが設定されています。パーミッションについては、「3.3.4. ファイルの所有権とパーミッション」で説明します。

3.2.1. 特権ユーザーと一般ユーザー

Linux システムでは、ユーザーを一般ユーザーと特権ユーザーの 2 種類に大別します。

特権ユーザーは、システム管理などをおこなうために用意されているユーザーで、一つのシステムに必ず一つ存在します。Windows での Administrator のような役割です。通常、特権ユーザーには root という名前を付けることが多いので、root ユーザーと呼ばれることもあります。

一般ユーザーは、特権ユーザー以外のユーザーのことで、ユーザーごとに行える作業(権限)に制限があります。Linux システムでは、ユーザーごとに必要最小限の権限を与えることで、システム全体のセキュリティを保ちます。

ATDE3 では、特権ユーザーとして root ユーザー、一般ユーザーとして atmark ユーザーが用意されています。

特権ユーザーはすべての権限を持っているため、誤った作業をおこなうと、ファイルをすべて消してしまうなど、システムを復旧不可能な状態に破壊してしまう可能性があります。そのため、通常の作業は一般ユーザーでおこないます。しかし、ソフトウェアのインストールなどでは、特権ユーザーの権限が必要になります。そのような場合は、一時的にユーザーを切り替えて作業をおこないます。

一般ユーザーから特権ユーザにユーザーを切り替えるには、以下のように **su** コマンドを使用します。特権ユーザーでの作業が終了したら、**exit** コマンドを実行し、元のユーザーに戻ることを忘れないでください。

```
[ATDE ~]$ su
パスワード: root
[ATDE ~]# 特権ユーザーでの作業
[ATDE ~]# exit
[ATDE ~]$
```

図 3.4 ユーザーの切り替え(su コマンド)

sudo コマンドを使うと、そのコマンドだけ別のユーザーとして実行することができます。どのユーザーが、どのユーザーとして、どのコマンドを実行できるかは、/etc ディレクトリにある sudoers ファイルに記述します。

ATDE3 では armark ユーザーが root ユーザーとしてすべてのコマンドを実行できるように設定してあります。以下のように **sudo** コマンドを実行すると、特権ユーザー権限で *somecommand* を実行します。なお、**sudo** コマンド実行時に入力するパスワードは、**sudo** コマンドを実行したユーザー自身(この場合では armark ユーザー)のパスワードです。

```
[ATDE ~]$ sudo somecommand
パスワード: atmark
```

図 3.5 一時的なユーザーの切り替え(sudo コマンド)

3.2.2. ユーザーの追加と削除

システムに新しくユーザーを追加するには、**useradd** コマンドを使用します。

newuser というユーザー名のユーザーを追加するには、以下のようにします。ユーザーの追加には、特権ユーザーの権限が必要なので、**sudo** コマンドを介して **useradd** コマンドを実行します。-m オプションは、ホームディレクトリを作成するオプションです。

```
[ATDE ~]$ sudo useradd -m newuser
```

図 3.6 ユーザーの追加(useradd コマンド)

作成したばかりのユーザーは、パスワードが設定されていないため、そのユーザーでログインすることができません。パスワードの設定、変更をおこなうには、**passwd** コマンドを使用します。

以下の例では、newuser のパスワードを password に設定しています。

```
[ATDE ~]$ sudo passwd newuser
新しい UNIX パスワードを入力してください: password
新しい UNIX パスワードを再入力してください: password
passwd: パスワードは正しく更新されました
```

図 3.7 パスワードの設定(passwd コマンド)

ユーザーを削除するには、**userdel** コマンドを使用します。-r オプションを付けることで、ユーザーのホームディレクトリも削除することができます。

```
[ATDE ~]$ sudo userdel -r newuser
```

図 3.8 ユーザーの削除(userdel コマンド)

ログインできないユーザー

shadow ファイルの二番目のフィールドには、暗号化したパスワードが記述されています。ここに、*や!と書くと、無効なパスワードになり、そのユーザーでログインすることができなくなります。

ATDE の/etc/shadow を見てみると、有効なパスワードが設定されていて、ログインできるのは root ユーザーと armark ユーザーだけです。その他のユーザーは、サーバーやシステム管理用のユーザーとして用意されています。例えば、www-data は Web サーバー用のユーザーです。

su コマンドを使用しても、ログインできないユーザーには切り替えることができません。しかし、**sudo** コマンドの-u オプションを指定すると、コマンドを実行するユーザーを指定することができます。この機能を使用すると、ログインできないユーザーとしてコマンドを実行することもできます。

```
[ATDE ~]$ sudo -u www-data whoami
www-data
```

図 3.9 ログインできないユーザーとしてコマンドを実行する(sudo コマンド)

3.2.3. ユーザーとグループ

Linux システムでは、ユーザーの集まりをグループという単位で管理することができます。ユーザーとグループという仕組みを使用して、システム(特にファイル)の管理を行う方法は、「3.3.4. ファイルの所有権とパーミッション」で詳しく説明します。

新しくグループを作成するコマンドは、**groupadd** コマンドです。グループを削除には、**groupdel** コマンドを使用します。

```
[ATDE ~]$ sudo groupadd newgroup
```

図 3.10 グループを追加する(groupadd コマンド)

ユーザーは、少なくとも一つのグループに属します。ユーザーが属する一つ目のグループを、ログイン時初期グループといいます。**useradd** コマンドを使用して新しくユーザーを追加すると、ユーザー名と同名のグループ名が新規に作成され、ログイン時初期グループに設定されます。また、ユーザーが属しているログイン時初期グループ以外のグループを、補助グループといいます。

ユーザーが所属するグループを変更するには、**usermod** コマンドを使用します。**usermod** コマンドの-g オプションを使うと、ログイン時初期グループを変更することができます。また、-G オプションを使うと、補助グループを変更することができます。

```
[ATDE ~]$ sudo usermod -G newgroup newuser
```

図 3.11 ユーザーが所属するグループを変更する(usermod コマンド)

3.3. ファイルの操作

Linux システムを含む、UNIX システムでは「すべてのものはファイルである(Everything is a file)」という考え方があります。Linux システムでは、ディスク上のデータも、動作中のプロセスも、ハードウェアであるデバイスさえも、ファイルとして表現します。基本的なファイルの操作は、すべてのファイルで共通です。テキストファイルの内容を読むのも、プロセスの状態を調べるのも、デバイスからデータを読み出すのも、基本的には同じ操作でできます。

本章では、様々なファイルに対する操作方法について説明していきます。

3.3.1. ファイルの種類

Linux システムで扱えるファイルには、以下のような種類があります。

1. 通常ファイル

   いわゆる普通のファイルです。大抵の場合、ハードディスクドライブなどのストレージに記録され、テキストファイル、バイナリファイル、実行ファイル、データファイルなどとして読み書きできます。

2. ディレクトリ

   他のファイルやディレクトリを格納することができるファイルを、ディレクトリといいます。Windows でのフォルダと同様の概念です。

3. デバイスファイル

   Linux カーネル内のデバイスドライバ[2]とのインターフェースとなるファイルです。スペシャルファイルやデバイスノードという場合もあります。

   スペシャルファイルには、キャラクタデバイスファイルとブロックデバイスファイルの 2 種類があります。キャラクタデバイスファイルへの入出力は、ストリームとして扱われ、一度書き込んだ内容は取り消せず、同じ内容を 2 回読み出すこともできません。対して、ブロックデバイスはランダムアクセス(任意の位置への読み書き)が可能なので、同じ位置に何度も読み書きすることができます。

   シリアルインターフェースや、マウス、キーボードなどはキャラクタデバイスファイルで、ハードディスクなどのストレージやメモリはブロックデバイスファイルとして扱われます。

   デバイスファイルは、必ずしも物理的なデバイスと結びついているわけではありません。そのようなデバイスファイルを、疑似デバイスファイルといいます。読み込むと常に 0 を返す/dev/zero、ある程度ランダムな値を返す/dev/urandom、書き込んだ内容を捨てる/dev/null などがあります。

[2]物理的なデバイスを操作するためのプログラム

4. シンボリックリンク

   ファイル名とファイルの実体との関係をリンクといいます。シンボリックリンクは、ファイルのパス名によって別のファイルを参照するリンクです。

   シンボリックに対して、ハードリンクというリンクもあります。これについては、「3.3.2. ファイルの属性情報(inode)」で説明します。

5. その他

   その他のファイルの種類として、FIFO(名前付きパイプ)と UNIX ドメインソケットがあります。これらは、IPC(InterProcess Communication、プロセス間通信)に使われます。

3.3.2. ファイルの属性情報(inode)

すべてのファイルはファイルの内容とは別に、ファイルの属性を表すメタデータを持っています。このメタデータのことを、inode といいます。inode には、以下の情報が格納されています。

表 3.1 inode が持つ情報

情報	説明
種類	「3.3.1. ファイルの種類」で挙げたファイル種類のどれであるか
所有者情報	ファイルを所有するユーザー(所有者)及び所有するグループ(所有グループ)の ID
パーミッション	所有者、所有グループに所属するユーザー、それら以外のユーザーに対する読み出し、書き込み、実行許可情報
ハードリンク数	ファイルに対するハードリンクの数
サイズ	ファイルのサイズ
時刻情報	最終アクセス時刻、最終修正時刻、最終属性状態変更時刻[1]

[1]Linux システムでは、ファイルの作成日時は保存されません。

inode にはファイル名が含まれていないことに注目してください。Linux システムでは、ファイル名を保持しているのはディレクトリです。inode には inode 番号と呼ばれる一意な数値が割り振られており、ディレクトリはファイル名と inode 番号の対応のリストを保持します。この、ファイル名と inode との対応をハードリンクといいます。一つの inode に対し、複数のファイル名を付ける、即ち、複数のハードリンクを張ることも可能です。

このため、Linux システムではファイルを削除することをアンリンク(unlink)といいます。複数のハードリンクがある場合、アンリンクはディレクトリからリンクを削除するだけです。ハードリンク数が 0 になった時に、実際にファイルの内容が削除されます。

inode が持つ情報は、**ls** コマンドに-l オプションをつけて実行することで確認することができます。

```
[ATDE ~]$ ls -l /bin/cat
-rwxr-xr-x 1 root root 26860 2008-04-04 23:22 /bin/cat
```

図 3.12 inode が持つ情報を確認する(ls -l コマンド)

最初の一文字は、ファイルの種類を表します。ファイルの種類によって、以下の表記になります。

表 3.2 ファイル種類の表記

表記	ファイル種類
-	通常ファイル
d	ディレクトリ
c	キャラクタデバイスファイル
b	ブロックデバイスファイル
l	シンボリックリンク

「rwxr-xr-x」の部分は、ファイルのパーミッションを表します。パーミッションについては、「3.3.4. ファイルの所有権とパーミッション」で説明します。

続く「1」は、ハードリンクの数を表します。

「root root」は、それぞれ、所有者のユーザー名、所有グループのグループ名を表します。これらは、パーミッションの設定と密接に関わっています。

「26860」はバイト単位のファイルサイズです。

「2008-04-04 23:22」は、ファイルの最終修正時刻を表します。

3.3.3. ファイルシステムとパス

Linux システムでは、ファイル同士の位置関係は階層的な木構造として表現されます。ファイルシステムとは、ファイルの木構造をある形式に従って構成したものです。

木構造の最上位に位置するディレクトリをルートディレクトリといいます。全てのファイルはルートディレクトリから辿ることができます。

また、Linux システムのファイルシステムでは、木構造の任意の位置にファイルシステムを追加または削除することができます。この操作をそれぞれ、ファイルシステムをマウントする、アンマウントするといいます。

システムに最初にマウントされるファイルシステムをルートファイルシステムといいます。ルートファイルシステムは、システム起動時にルートディレクトリにマウントされます。

木構造の中のファイルの位置は、パスで表します。パスはあるディレクトリからファイルに到達するまでの間にあるディレクトリ名の間にスラッシュ(/)を挟んだものです。ルートディレクトリは/一文字で表します。パスの記述方法には二種類あり、/から始まるルートディレクトリからの位置を表したパスを絶対パスといい、ルートディレクトリ以外からの位置を表したパスを相対パスといいます。パスには、/以外にもいくつか特殊な意味を持つ文字があります。. は現在のディレクトリを、.. は一つ上のディレクトリを意味します。また、多くのシェルでは~は、ホームディレクトリを意味します。

ファイルシステムには、構造を構成する形式が異なるいくつかの種類があります。Linux で一般的に使用されるファイルシステムには、ext2 ファイルシステム、ext3 ファイルシステムがあります。Armadillo のルートファイルシステムは、RAM ディスク[3]上に構成された ext2 ファイルシステムです。ext3 ファイルシステムは、ext2 ファイルシステムにジャーナリング機能[4]を追加したもので、耐障害性に優れます。また、Windows で使用される VFAT(FAT32)ファイルシステムも扱うことができます。

これらのファイルシステムは物理的なデバイスと結びついているものですが、メモリ上にしか存在しないファイルシステムもあります。その一つである仮想ファイルシステムには、カーネルの内部情報を

[3]RAM の一部を仮想的なストレージとして使用する仕組み。
[4]定期的にファイルシステムの状態を保存しておく機能。不意なシステムシャットダウン時にもファイルシステムの破損を防止し、記録された状態に復旧することが可能となる。

参照又は設定できる procfs や sysfs などがあります。また、疑似ファイルシステム(pseudo filesystem)は、RAM 上に直接ファイルシステムを構成します。擬似ファイルシステムには、tmpfs や ramfs があります。

さらに、ネットワークファイルシステム(NFS)を使用すると、ネットワーク越しのコンピューターに存在するデータを扱うことができます。

3.3.4. ファイルの所有権とパーミッション

前章で説明したように、Linux システムではすべてのファイルに、そのファイルを所有するユーザー(所有者)とグループ(所有グループ)が設定されています。そして、所有者、所有グループに所属するユーザー、それ以外のユーザーに対して、許可する操作を決めることができます。これを、パーミッションといいます。パーミッションによって、それぞれのユーザーに対して、読み出し、書き込み、実行の可否を設定できます。

ディレクトリのパーミッション

ディレクトリもファイルの一種ですので、パーミッションを設定できます。

ディレクトリの読み出し許可がある場合、ディレクトリ内のファイルのリストを取得することができます。

書き込み許可がある場合、ディレクトリの中にファイルを作成したり、ファイルを削除できます。

実行許可がある場合、ディレクトリに中のファイルにアクセスできます。反対にいうと、実行許可がない場合、ディレクトリ内のファイルに対して、それらのファイルのパーミッションに関わらず、読み、書き、実行のすべてを行うことができませんので、注意してください。

「図 3.12. inode が持つ情報を確認する(ls -l コマンド)」の例では、パーミッションは「rwxr-xr-x」と表示されていました。それぞれのユーザーに対するパーミッションは、三文字ずつで表現されます。順番に、所有者、所有グループに所属するユーザー、その他のユーザーに対するパーミッションを意味します。r が読み出し許可、w が書き込み許可、x が実行許可を意味します。「rwxr-xr-x」の場合、所有者に対しては「rwx」即ち、読み、書き、実行すべてを許可します。所有グループに所属するユーザーとその他のユーザーに対しては、「r-x」即ち読みと実行だけ許可します。

パーミッションは、rwx といったアルファベットでの表記の他に、8 進数で表記する場合もあります。r=4、w=2、x=1 として[5]、その合計で表現します。8 進数表記でのパーミッションは、rwx は 7、rw-は 6、r-x は 5、r--は 4、---は 0 となります。そのため、「rwxr-xr-x」を 8 進数 3 桁で表記すると、「755」となります。

パーミッションを変更するには、**chmod** コマンドを使用します。すべてのユーザーに対して実行を許可する(つまり、実行権限を与える)場合、+x オプションを使用します。

```
[ATDE ~]$ chmod +x ファイル名
```

図 3.13 ファイルのパーミッションを変更する(chmod コマンド)

[5]rwx を 2 進数の各桁に見立てています。

ファイルを新規に作成した場合のパーミッションは、ファイルの種類と umask と呼ばれる値によって決まります。標準のパーミッションは、ディレクトリの場合 rwxrwxrwx(777)、その他のファイルは rw-rw-rw-(666)です。この値から umask を差し引いた値が、ファイルを新規作成した場合のパーミッションとなります。

umask は、一般的には 022 となっています。そのため、ディレクトリを新規作成した場合のパーミッションは rwxr-xr-x(755)、通常ファイルの場合は、rw-r—r--(644)となります。

```
[ATDE ~]$ umask
0022
[ATDE ~]$ touch file
[ATDE ~]$ mkdir dir
[ATDE ~]$ ls -l
合計 0
drwxr-xr-x 2 atmark atmark 48 2010-09-28 18:56 dir
-rw-r--r-- 1 atmark atmark  0 2010-09-28 18:56 file
```

図 3.14 umask と新規作成時のファイルパーミッション

touch は、引数に指定したファイルの最終修正時刻を更新するコマンドです。存在しないファイルを指定した場合、空のファイルを作成します。

ところで、**umask** コマンドを実行した際、4 桁で表示されています。この、4 桁目の値は、特別なパーミッションを意味します。特別なパーミッションには、SUID ビット(set-uid bit)、SGID ビット(set-gidbit)、スティッキービット(sticky bit) の 3 種類があります。

8 進数で表すと、SUID ビットがセットされている場合 4、SGID ビットは 2、スティッキービットは 1 となります。この値は、**chmod** コマンドや **umask** コマンドでパーミッションを 8 進数で指定する場合の4桁目として使用できます。

SUID ビットが設定されていて実行許可が与えられている場合、そのファイルを実行すると、実行したユーザーに関わらず、ファイル所有者として実行されます。この仕組みは、例えば **passwd** コマンドに使用されています。**passwd** コマンドがアクセスする/etc/shadow ファイルにはパスワードという重要な情報が情報が記述されているため、特権ユーザーである root ユーザーにだけ読み書きを許可しています。しかし、一般ユーザーが自分のパスワードを変更できないと不便です。そのため、**passwd** の実行ファイルには SUID ビットがセットされており、所有者は root になっています。すると、一般ユーザーで **passwd** を実行した場合でも root として実行されるため、/etc/shadow ファイルにアクセスすることができます。

```
[ATDE ~]$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1053 2010-02-23 19:11 /etc/shadow
[ATDE ~]$ ls -l $(which passwd)
-rwsr-xr-x 1 root root 39104 2009-12-06 13:35 /usr/bin/passwd
```

図 3.15 etc/shadow ファイルと passwd 実行ファイルのパーミッション

ls -l では、SUID ビットがセットされていて所有者の実行が許可されているファイルの場合、所有者の実行許可の表示が x ではなく s となります。

なお、**which** コマンドは、引数に指定したコマンドが実際に実行するファイルの絶対パスを表示するコマンドです。

SGID ビットは、SUID ビットと同様の仕組みです。実行したユーザーが所属するグループではなく、所有グループとして実行されます。**ls -l** では、SGID ビットがセットされていて所有グループに所属す

るユーザーに対する実行が許可されているファイルの場合、所有グループの実行許可の表示が x ではなく s となります。

スティッキービットは、ディレクトリとその他の実行ファイルに指定された時では挙動が異なります。

ディレクトリの場合、ディレクトリ内にあるファイルは、ファイル所有者とディレクトリ所有者のみが削除できます。この仕組みは、/tmp ディレクトリで使用されています。一時的に使用するファイルを置く/tmp ディレクトリには、すべてのユーザーに対して読み、書き、実行を許可していますが、ファイルを作成したユーザーか/tmp ディレクトリの所有者である root ユーザーしかファイルを削除することができません。**ls -l** では、スティッキービットがセットされていてその他のユーザーに対する実行が許可されている場合、その他のユーザーの実行許可の表示が x ではなく t となります。

```
[ATDE ~]$ ls -ld /tmp
drwxrwxrwt  17 root root 1456 2010-09-28 19:57 tmp
```

図 3.16 /tmp ディレクトリのパーミッション

実行可能ファイルに対してスティッキービットを設定した場合、そのコードをスワップ上に維持します。こちらの機能はあまり使用されていないようです。

3.3.5. デバイスファイルの管理

デバイスファイルは、メジャー番号とマイナー番号という二つの番号によって、対応するデバイスドライバを識別します。メジャー番号は、デバイスの種類を表します。デバイスの型(キャラクタ型かブロック型)とメジャー番号が同じデバイスファイルは、ほとんど場合、同じデバイスドライバを使用します。また、マイナー番号によって、同じデバイスの型とメジャー番号を持つデバイスのグループ内のデバイスを識別します。

デバイスファイルは、通常/dev ディレクトリ以下に配置されます。

ls -l でデバイスファイルを調べた場合、他のファイルとは異なった出力となります。

```
[ATDE ~]$ $ ls -l /dev/ttyS*
crw-rw---- 1 root dialout 4, 64 2010-09-18 18:58 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 2010-09-18 18:58 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 2010-09-18 18:58 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 2010-09-18 18:58 /dev/ttyS3
```

図 3.17 デバイスファイルの例

最初の一文字は、デバイスファイルの型を表します。「c」がキャラクタデバイスファイル、「b」がブロックデバイスファイルを意味します。所有者、所有グループの後に表示されている「4, 64」という番号が、それぞれメジャー番号とマイナー番号を表します。メジャー番号 4 は、シリアルポートに対応するデバイスファイルを意味します。メジャー番号とデバイスの対応は、Linux カーネルのドキュメントに記載されています。linux-2.6.26-at/Documentation/devices.txt を参照してください。

デバイスファイルを作成するには、**mknod** コマンドを使用します。例えば、5 個目のシリアルポートに対応するデバイスファイルを作成するには、以下のようにします。

```
[ATDE ~]$ sudo mknod /dev/ttyS4 c 4 68
```

図 3.18 デバイスファイルの作成(mknod コマンド)

通常は、上記のように **mknod** コマンドを使用してデバイスファイルを作成します。しかし、Linux システムはデバイスのホットプラグ[6]が可能です。システム動作中接続される可能性のあるデバイスに対するデバイスファイルをあらかじめすべて作っておくことは、現実的ではありません。そこで、デバイスが接続された時点でデバイスファイルを作成する udev という仕組みがあります。udev を使用すると、デバイスファイルの自動作成の他、デバイスが接続または切断された時点で任意のコマンドを実行する、といったことが可能になります。

3.4. プログラムとプロセス

プログラムの実行可能ファイルは、機械語の実行コードとデータから構成されます。多くの Linux システムでは、実行可能ファイルは ELF と呼ばれる形式で保存されています。

実行可能ファイルは、ローダーと呼ばれるプログラムによってメモリにロードされ、実行が開始されます。この際、ローダーはプログラムの再配置やメモリの初期化などをおこないます。

実行中のプログラムをプロセスといいます。

Linux システムは、マルチタスクなので複数のプロセスを同時に実行できます。しかし、CPU は通常 1 個[7]しかありません。そこで、プロセスには仮想的な CPU とメモリ空間が与えられます。カーネルは、各プロセスが CPU を使う時間とメモリ領域を管理します。カーネルは、プロセスが CPU を使ってよい時間が過ぎたら、そのプロセスの実行を停止し別のプロセスの実行を開始します。また、プロセスから見えている仮想的なメモリ空間と物理メモリとの橋渡しをおこないます。そのため、プロセスはあたかもシステム全体を占有しているように振る舞う事ができます。

プロセスが使用するメモリ空間は、いくつかの領域(セクション)に分かれおり、それぞれ用途が異なります。セクションには以下のものがあります。

1. テキストセクション: 機械語の実行や定数などを収めた、読み取り専用で実行可能な領域。
2. データセクション: グローバル変数やスタティック変数のうち初期値が設定されたのデータを収めた領域。
3. BSS セクション: グローバル変数やスタティック変数のうち初期値が設定されていないデータを収めた領域。0 で初期化される。
4. スタック: 関数呼び出し時に一時的なデータ用に使用される領域。
5. ヒープ: プロセスが要求する動的なメモリ用に使用される領域。

ps コマンドを使用すると、現在実行中のプロセスの一覧を見ることができます。

```
[ATDE ~]$ ps
  PID TTY          TIME CMD
 2488 pts/0    00:00:00 bash
18880 pts/0    00:00:00 ps
```

図 3.19 プロセス一覧の確認(ps コマンド)

[6]システム動作中にデバイスを追加すること。
[7]最近は、マルチコアな CPU も当たり前になってきましたが。

「CMD」の欄に表示されているものが、プロセスを実行したコマンドです。「PID」は、プロセス ID と呼ばれるプロセスを一意に識別する数値です。

プロセスは、プロセスの状態や所有するリソース(タイマー、ファイル、ハードウェア、ネットワーク接続、プロセス間通信で使用するもの)、そのプロセスを実行したプロセスの ID(親プロセス ID、PPID)などの情報を保持しています。

3.5. シグナル

シグナルとは、カーネル又はプロセスからプロセスに対して送られる非同期なメッセージです。通常、メモリアクセス保護違反(segmentation fault)や特殊なキー入力(例えば Ctrl+C)などのイベントが起こったことをプロセスに通知するために使用されます。

プロセスにシグナルが送られた場合、プロセスは以下の挙動のうちいずれかを行います。

1. 終了する
2. シグナルを無視する
3. プロセスのコア[8]をダンプ(表示)して終了する
4. 一時停止する
5. 停止中であれば再開する
6. シグナルを捕捉し、プロセス自身で指定したシグナルハンドラーで処理を行う

シグナルは、シグナル名かシグナル番号(整数値)で表されます。

代表的なシグナルを以下に示します。シグナルハンドラーを設定していない場合、標準の挙動をおこないます。捕捉できないシグナルは、必ず標準の挙動をおこないます。

表 3.3 代表的なシグナル

シグナル名	番号	捕捉可能か	標準の挙動	説明
SIGHUP	1	Yes	終了	制御端末のハングアップ、制御しているプロセスの死
SIGINT	2	Yes	終了	キーボードからの割り込み(Ctrl+C)
SIGQUIT	3	Yes	コアダンプ	キーボードからの割り込み
SIGABRT	5	Yes	コアダンプ	abort 関数による終了
SIGKILL	9	No	終了	強制的な終了
SIGSEGV	11	Yes	コアダンプ	不正なメモリ参照
SIGTERM	15	Yes	終了	終了シグナル
SIGUSR1	10,16,30	Yes	終了	ユーザー定義シグナル 1
SIGUSR2	12,17,31	Yes	終了	ユーザー定義シグナル 1
SIGSTOP	17,19,23	No	停止	プロセスの一時停止
SIGCONT	18,20,24	Yes	再開	プロセスの再開

kill コマンドで、プロセスに任意のシグナルを送ることができます。**kill** コマンドには、シグナルを送るプロセス名とシグナル名(SIG を除いたもの)を指定することができます。**kill** コマンドでシグナル名を指定しない場合、SIGTERM が送られます。

[8] プロセスの状態を保存したもの。

3.6. プロセス間通信

それぞれのプロセスは、独立した仮想的なメモリ空間を与えられて動作するため、基本的に他のプロセスのデータにアクセスすることはできません。この特徴により、あるプロセスが誤って他のプロセスのデータを書き換えるということがないため、セキュリティやシステムの堅牢性を保つことができます。

しかし、場合によっては複数のプロセスが強調動作をしたり、情報のやりとりを行う必要があったりします。そのために、Linux カーネルはプロセス間通信(Inter Process Communication、IPC)の仕組みを提供しています。

IPC を行う方法には、以下のものがあります。

1. パイプ
2. 名前付きパイプ(FIFO)
3. メッセージキュー
4. 共有メモリ
5. セマフォ
6. インターネットソケット
7. 名前付きソケット(UNIX ドメインソケット)

3.7. 端末

端末(ターミナル)とは、ディスプレイと入力装置(キーボード)から構成され、コンピューターの使用者がホストコンピューターとのやりとりを行うために使用される装置です。ホストコンピューターと端末は、シリアル通信線や電話線、Ethernet などで接続されます。コンピューターが高価で、一つのコンピューターを複数人で共有していた時代は、一つのホストコンピューターに複数の端末を接続して使用していました。コンピューターが十分に安くなり、一人一台の「パーソナルな」コンピューター(PC)を持てるようになった現在では、端末専用装置を見かけることはほとんどありません。

今日では、端末専用装置の代わりに、PC 上で動作する端末エミュレーターを使用します。Armadillo と開発用 PC をシリアルケーブルで接続し、シリアル通信ソフトウェアで操作をおこなう場合、シリアル通信ソフトウェアを端末エミュレーターとして使用していることになります。大抵の端末エミュレーターでは、端末専用装置として事実上の標準であった VT100 の互換機能を持っています。

Linux システムでは、様々な装置を端末とみなして、互いに通信することができます。PC に接続されたモニターはコンソール端末、シリアルポートを介して接続されているコンピューターはシリアル端末、ネットワークを介して接続されているコンピューターは擬似端末とみなします。Linux システムでは、端末との通信を tty[9]という名前のついたデバイスファイル(tty デバイス)を介しておこないます。

3.7.1. シリアル端末

Linux システムでは、シリアルポートに対する読み書きは、シリアルポートに対応する tty デバイスへの読み書きとして扱います。つまり、シリアルポートの先にシリアル端末が繋がっているとみなしています。例えば、Windows では COM1 と表現される、PC の一番目のシリアルポートへの読み書きは、Linux システムでは通常、`/dev/ttyS0` に対しておこないます。

[9]tty は「TeleTYpe」の略です。テレタイプとは電動機械式のタイプライターのことで、初期の端末として使用されていたため、このような名前になっています。

```
[ATDE ~]$ echo hello > /dev/ttyS0 ❶
[ATDE ~]$ cat file > /dev/ttyS0 ❷
[ATDE ~]$ cat /dev/ttyS0 ❸
```

図 3.20 シリアルポートの読み書き

❶ 文字列「echo」をシリアルポートから送信します。

❷ *file* の内容をシリアルポートから送信します。

❸ シリアルポートに受信した内容を表示します。

シリアルポートの通信速度等の設定確認や変更は、**stty** コマンドで行うことができます。**stty** コマンドでは、-F オプションで設定の確認や変更をおこなう tty デバイスを指定します。詳細は、**man 1 stty** を参照してください。

```
[ATDE ~]$ stty -F /dev/ttyS0
speed 9600 baud; line = 0;
-brkint -imaxbel
```

図 3.21 シリアルポートの設定確認(stty コマンド)

Armadillo では、シリーズによってシリアルポート(シリアルインターフェース)に割り当てているデバイスファイル名が異なります。Armadillo-400 シリーズでは、/dev/ttymxc*#*(#は 10 進数値文字)となり、/dev/ttymxc1 がシリアルインターフェース 1 に割り当てられています。詳細は、「Armadillo-400 シリーズ ソフトウェアマニュアル」の「Linux カーネルデバイスドライバ仕様」の「UART」をご参照ください。

USB to シリアル変換ケーブルを使った場合、変換ケーブル内の IC によってデバイスファイル名が異なります。通常、/dev/ttyUSB*#*や/dev/ttyACM*#*になります。

3.7.2. コンソール端末

コンソール端末、あるいは単にコンソールとは、システム管理用の端末のことです。通常、ホストコンピューターに接続されたディスプレイとキーボードをコンソールとして使用します[10]。Linux システムでは、ホストコンピューターに接続されたディスプレイに表示できる、仮想的なコンソールを複数持つことができます。仮想コンソールには、/dev/tty*#*を介してアクセスします。/dev/tty0 は特別な意味を持ち、現在の仮想コンソールを意味します。

Debian GNU/Linux 5.0 では、Ctrl+Alt+F*#*を入力することで、仮想コンソールを切り替えることができます[11]。Ctrl+Alt+F1 で一番目の仮想コンソール(/dev/tty1)に切り替えます。Debian GNU/Linux 5.0 では、F1 から F6 までがテキストコンソールに割り当てられています。Ctrl+Alt+F7 で元の GUI 画面に戻ることができます。

端末が使用している tty デバイスは、**tty** コマンドで調べることができます。Ctrl+Alt+F1 で仮想コンソールを切り替え、ログインしてから **tty** コマンドを実行すると、以下のように表示されます。

[10]シリアルケーブルで接続された端末をコンソールとして使用することもあります。この場合、シリアルコンソールと呼びます。
[11]ATDE は VMware Player 上で動作しているため、ホットキーをそのまま入力することができません。Ctrl+Alt+スペースを入力したあと、Ctrl+Alt を押したままで、Ctrl+Alt+F*#*を入力してください。

```
[ATDE ~]$ tty
/dev/tty1
```

図 3.22 端末が使用している tty デバイスの確認(tty コマンド)

/dev/console は、システムメッセージを表示するコンソール(システムコンソール)用のデバイスファイルです。カーネルメッセージは/dev/console に送信されます。

どの端末をシステムコンソールとして使用するかは、カーネルの起動時に渡すカーネルパラメーターで指定できます。Armadillo-400 シリーズでは、標準状態のカーネルパラメーターとして「console=ttymxc1,115200」を渡しているので、シリアルインターフェース 1 がシステムコンソールとして使用されます。ATDE3 では「root=/dev/mapper/atde3-root ro quiet acpi=off」を渡しており、システムコンソールを明示的に指定していません。この場合、カーネルは最初に/dev/tty#を調べ、次にシリアルデバイスを順番に調べます。そして、最初に使用可能であったものをシステムコンソールとして使用します。

カーネルパラーメーターは、/proc/cmdline ファイルで調べることができます。

```
[ATDE ~]$ cat /proc/cmdline
root=/dev/mapper/atde3-root ro quiet acpi=off
```

図 3.23 カーネルパラメーターの確認(proc/cmdline ファイル)

3.7.3. 擬似端末

擬似端末は、シリアル端末やコンソール端末のように、必ずしも物理的に接続されているとは限らない端末との通信に使用されます。ATDE3 の[アプリケーション]-[アクセサリ]-[端末]メニューで起動できる端末エミュレーター(Gnome 端末)で **tty** コマンドを実行すると、/dev/pts/0 のように表示されます。

```
[ATDE ~]$ tty
/dev/pts/0
```

図 3.24 Gnome 端末での tty デバイス

擬似端末は、マスターとスレーブがセットになって使用されます。スレーブへ書き込んだデータは、マスターから読み出すことができ、また、マスターへ書き込んだデータはスレーブから読み出すことができます。/dev/pts/#はスレーブで、マスターは常に/dev/ptmx です。マスターは/dev/ptmx 一つだけですが、プロセスが/dev/ptmx をオープンすると、都度/dev/pts ディレクトリに対応するスレーブ用のデバイスファイルが作成され、以後そのスレーブとやりとりを行うことができるようになっています。このような命名規則を UNIX98 pty naming といいます。詳細は、**man 4 pts** を参照してください。

UNIX98 pty naming を採用せずに、スレーブに ttyp*M*(*N* は 16 進数値文字)、マスターに ptyp*N* という名称を使用する場合もあります。例えば、作業用 PC から標準イメージで動作する Armadillo-440 に telnet で接続する[12]と、**tty** コマンドで確認できる tty デバイスは/dev/ttyp0 になります。telnet クライアントが/dev/ttyp0 へ書き込んだデータは、telnet サーバーが/dev/ptyp0 から読み出します。

[12]この場合、telnet クライアントが端末エミュレーターです。

```
[ATDE ~]$ telnet Armadillo の IP アドレス
Trying Armadillo の IP アドレス...
Connected to Armadillo の IP アドレス.
Escape character is '^]'.

atmark-dist v1.26.1 (AtmarkTechno/Armadillo-440)
Linux 2.6.26-at10 [armv5tejl arch]

armadillo440-0 login: guest
[armadillo ~]$ tty
/dev/ttyp0
```

図 3.25 Armadillo に telnet で接続した場合の tty デバイス

3.8. 時間の管理

Linux システムでは時間の管理は二つの時計、システムクロックとハードウェアクロックでおこなっています。

システムクロックは Linux カーネルが管理している時計で、タイマー割り込みによって駆動されます。システムクロックは、UTC(Universal Time, Coordinated、協定世界時) 1970 年 1 月 1 日 00 時 00 分 00 秒(紀元、エポック)からの経過秒数で表されます。Linux システムでは、システムクロックがすべての動作の基準となります。システムクロックを参照、設定するには、**date** コマンドを使用します。

```
[ATDE ~]$ date
2010 年 10 月  5 日 火曜日 04:11:59 JST
```

図 3.26 システムクロックの参照(date コマンド)

ハードウェアクロックは、CPU とは独立した RTC(リアルタイムクロック)によって管理される時計です。システムに電源が供給されていない間も、バッテリや外部電源などで動作しつづけます。Linux システムは、起動時にハードウェアクロックを参照し、システムクロックを設定します。

```
[ATDE ~]$ hwclock
2010 年 10 月 05 日 06 時 22 分 15 秒  -0.244000 seconds
```

図 3.27 ハードウェアクロックの参照(hwclock コマンド)

3.8.1. タイムゾーン

ATDE3 で **date** コマンドを実行すると、「図 3.26. システムクロックの参照(date コマンド)」で示したように JST(Japan Standard Time、日本標準時)で表示されます。システムクロックは、カーネル内部では常に UTC を基準としたエポックからの経過秒数で管理されています。しかし、**date** コマンドはシステムの設定に従ってローカルタイムでの表示を行います。

date コマンドなどの時間を扱うコマンドでどのタイムゾーンを使用するか、即ちローカルタイムのタイムゾーンは TZ 環境変数で指定することができます。環境変数については、「5.3.3. 環境変数」を参照してください。また、TZ 環境変数の指定方法については、**man 3 tzset** を参照してください。

```
[ATDE ~]$ date
2010 年 10 月  5 日 火曜日 06:15:46 JST
[ATDE ~]$ TZ=UTC date
2010年 10 月  4 日 月曜日 21:15:46 UTC
[ATDE ~]$ TZ=America/New_York date
2010 年 10 月  5 日 火曜日 06:16:28 JST
```

図 3.28 システムクロックの参照(タイムゾーンを指定)

TZ 環境変数が指定されていない場合、/etc/localtime ファイルで指定されているタイムゾーンが使用されます。ATDE3 では TZ 環境変数は設定されていないので、「図 3.26. システムクロックの参照(date コマンド)」で表示が JST になっていたのは、このファイルの設定によるものです。/etc/localtime ファイルは、タイムゾーンディレクトリ(/usr/share/zoneinfo)にある tzfile 形式のファイルのコピーとなっています。ATDE3 の標準設定では、/usr/share/zoneinfo/Asia/Tokyo です。Debian GNU/Linux 5.0 では、タイムゾーンの設定は **dpkg-reconfigure tzdata** で変更することができます。

ハードウェアクロックは UTC で保存するかローカルタイムで保存するか、選択することができます。UTC で保存しておけば、タイムゾーンを変更してもハードウェアクロックを変更しなくとも良いので、通常は問題ないでしょう。しかしながら、Windows ではローカルタイムで保存します。そのため、PC Linux ではローカルタイムをハードウェアクロックに保存することが多いようです。

```
[ATDE ~]$ sudo hwclock --systohc --utc
```

図 3.29 システムクロックをハードウェアクロックに設定する(UTC)

```
[ATDE ~]$ sudo hwclock --systohc --localtime
```

図 3.30 システムクロックをハードウェアクロックに設定する(ローカルタイム)

3.8.2. 時刻を正確に保つ

システムクロックとハードウェアクロックは、長期的な視点ではどちらも正確ではありません。通常、二つの時計は異なるクロックソースを元にして動作するので、相対的にずれていきます。また、国際原子時(TAI)と比較すると、どちらの時計も精度が低いので、絶対的な時刻も徐々にずれていきます。

時刻を正しく保つには、いくつかの方法があります。

最も信頼性の高い方法は、NTP(Network Time Protocol)を使用することです。インターネットに接続できるか、信頼できる NTP サーバーを使用できる場合、NTP によりシステムクロックを設定することができます。Debian GNU/Linux 5.0 では、**ntpd**(NTP サーバー)または **ntpdate** コマンド(NTP クライアント)で NTP による 時刻設定を行うことができます。

NTP を使用できない場合、クロックの規則的なずれ(ドリフト)を利用して補正を行なうことができます。システムクロックとハードウェアクロックは、時刻が進むか遅れる方向に同じ程度ずれると想定して、定期的にずれた分時刻を設定しなおすという方法です。

hwclock コマンドの時刻合わせ機能を使用すると、ハードウェアクロックのドリフトを補正することができます。**hwclock** コマンドは、--set オプションまたは--systohc オプションを伴って実行されると、ハードウェアクロックを設定します。このとき、/etc/adjtime ファイルに現在の時刻を最後に時計合わせ(calibration)をした時刻として記録します。ハードウェアクロックがずれた後、再度、--set オプショ

ンまたは--systohc オプションによりハードウェアクロックが設定されると、**hwclock** コマンドは/etc/adjtime ファイルの最後に時計合わせをした時刻を更新するとともに、1 日あたりの時刻のずれを記録します。以降は、--adjust を伴って **hwclock** コマンドを実行すると、1 日あたりの時刻のずれから補正すべき時刻を計算してハードウェアクロックを設定します。また、/etc/adjtime ファイルに最後に時刻を補正(adjustment)した時刻を記録します。詳細は、**man 8 hwclock** を参照してください。

adjtimex コマンドを使用すると、システムクロックのドリフトを徐々に補正することができます。**adjtimex** コマンドでは、NTP やハードウェアクロックを参照して、システムクロックのドリフトを測定し、それを補正するための値をカーネルに設定します。例として、NTP サーバーを参照する方法を以下に示します。

まず、ntpd が動作している場合、停止してください。**adjtimex** コマンドでドリフトを測定している途中に、ntpd がシステムクロックを更新してしまうと、正確な測定ができなくなります。

```
[ATDE ~]$ sudo /etc/init.d/ntp stop
Stopping NTP server: ntpd.
```

図 3.31 adjtimex によるシステムクロックの補正 1: ntpd の停止

次に、**adjtimex** をインストールします。NTP の参照には **ntpdate** コマンドを使用するので、同時にインストールします。

```
[ATDE ~]$ sudo apt-get install adjtimex ntpdate
```

図 3.32 adjtimex によるシステムクロックの補正 2: adjtimex のインストール

adjtimex コマンドに--host オプションを指定すると、**ntpdate** を使用して補正のためのデータを取得します。この例では、NTP サーバーには独立行政法人情報通信研究機構(NICT)のサーバーである、ntp.nict.jp を指定しています。--log オプションも同時に指定することで、補正のためのデータをログファイル(/var/log/clocks.log)に書き込みます。ハードウェアクロックやシステムクロックを **adjtimex** 以外で書き換えていない場合は、二つの質問に **y** と答えてください。

```
[ATDE ~]$ sudo adjtimex --log --host ntp.nict.jp
      reference time is Tue Oct  5 08:28:17 2010
reference time - system time = 1286234897.969 - 1286234791.000 = 106.969 sec
Last clock comparison was at Tue Oct  5 08:27:14 2010
Kernel time variables are unchanged - good.
System clock is synchronized (by ntpd?) - bad.
Checking wtmp file...
System has not booted since Tue Oct  5 08:27:14 2010 - good.
System time has not been changed since Tue Oct  5 08:27:14 2010 - good.
Checking /etc/adjtime...
/sbin/hwclock has not set system time and adjusted the cmos clock
since Tue Oct  5 08:27:14 2010 - good.

Are you sure that, since Tue Oct  5 08:27:14 2010,
  the system clock has run continuously,
  it has not been reset with date' or `/sbin/hwclock,
  the kernel time variables have not been changed, and
  the computer has not been suspended? (y/n) [n] y
The estimated error in system time is -19938.344 +- 187.183 ppm

Are you sure that, since Tue Oct  5 08:27:14 2010,
  the real time clock (cmos clock) has run continuously,
  it has not been reset with `/sbin/hwclock',
  no operating system other than Linux has been running, and
  ntpd has not been running? (y/n) [y] y
The estimated error in the cmos clock is -62097 +- 187 ppm
```

図 3.33 adjtimex によるシステムクロックの補正 3: ntpdate による補正データの測定

--print オプションを指定すると、現在の設定を確認することができます。また、補正のためのデータをカーネルに設定するには、--adjust オプションを使用します。その際、--review オプションを付けると、ログファイルに記録したデータをもとに設定します。

```
[ATDE ~]$ sudo adjtimex --print
         mode: 0
       offset: -682
    frequency: 4550171
     maxerror: 2437460
     esterror: 848
       status: 1
time_constant: 10
    precision: 1
    tolerance: 32768000
         tick: 9799
     raw time:  1286234858s 122967us = 1286234858.122967
[ATDE ~]$ sudo adjtimex --adjust --review
start                    finish                   days    sys - cmos (ppm)
start                    finish                   days    cmos_error (ppm)
start                    finish                   days    sys_error (ppm)
Tue Oct  5 08:22:41 2010  Tue Oct  5 08:25:29 2010  0.0020  -209 +- 49
Tue Oct  5 08:25:29 2010  Tue Oct  5 08:26:31 2010  0.0007  -103 +- 134
least-squares solution:
   cmos_error = 0 +- 100000 ppm
      (no suggestion)
        current adjustment = 0.0012 sec/day
   sys_error = -196 +- 46 ppm
      suggested tick = 10002  freq =   -270990
        current tick =  9799  freq =   4550171
note: clock variations and unstated data errors may mean that the
least squares solution has a bigger error than estimated here
new tick = 10002  freq = -270989
[ATDE ~]$ sudo adjtimex --print
         mode: 0
       offset: -675
    frequency: -270989
     maxerror: 2460460
     esterror: 848
       status: 1
time_constant: 10
    precision: 1
    tolerance: 32768000
         tick: 10002
     raw time:  1286234904s 17221us = 1286234904.017221
```

図 3.34 adjtimex によるシステムクロックの補正 4: 補正データの設定と確認

3.8.3. タイマーの分解能

システムクロックの分解能は、カーネルコンフィギュレーションで定義される HZ 定数によって決まります。HZ が 100 の場合、タイマー割り込みの間隔(jiffy)は 1 秒間に 100 回、つまり、0.01 秒(10 ミリ秒)に 1 回です。タイマー割り込みの度に、カーネル内で管理されている jiffies と呼ばれる値が 1 ずつ増加していきます。システムクロックは jiffies を元に計算されます。i386 や x86_64 アーキテクチャで動作する Linux では、HZ は 100、250(標準の値)、300、1000 を選択することができます。Armadillo-400 シリーズでは、HZ は 100 です。

Linux 2.6.21 より前のカーネルでは、プロセスをスリープさせたりタイマー[13]を扱うシステムコールの精度はシステムクロックの分解能に依存していました。そのため、HZ が 100 の場合、10 ミリ秒以下の時間スリープするといった動作はできませんでした。

しかし、Linux 2.6.21 からハイレゾリューションタイマー(High-Resolusion Timers)がサポートされました。ハイレゾリューションタイマーが有効なシステムでは、スリープやタイマーに関するシステムコールの精度は HZ による制約を受けず、CPU が処理できる限りの短い時間で反応できます。Linux 2.6.21 以降のカーネルを採用しているシステムがすべてハイレゾリューションタイマーを使用できるわけではなく、アーキテクチャごとにサポート状況は異なります。i386 や x86_64 アーキテクチャの PC Linux では、通常ハイレゾリューションタイマーが有効になっていますが、VMware Player 上で動作する ATDE3 では無効です。Armadillo-400 シリーズではハイレゾリューションタイマーが有効になっています。

3.9. ロケール

ロケールとは、多言語を扱うプログラムがどのようなルールに基づいて処理するべきかを定めたルールの集合です。プログラムは、ロケールに基づいて適切な言語や表記でメッセージを表示したり、文字集合を扱うことができます。

ロケールはいくつかのカテゴリに分かれており、それぞれ個別に設定できます。カテゴリには以下のものがあります。

1. LC_COLLATE: アルファベット文字列の比較方法を定義します。
2. LC_CTYPE: 文字の判定、変換操作や多バイト文字操作の方法を定義します。
3. LC_MONETARY: 小数点やカンマの位置など、通貨に関する数字の表示方法を定義します。
4. LC_MESSAGES: メッセージ表示に使用する言語を定義します。
5. LC_NUMERIC: 数字の扱いを定義します。
6. LC_TIME: 時刻の表示方法を定義します。

ロケールは、LANGUAGE と LC_ALL 及び上記のカテゴリに対応する環境変数によって設定できます。複数の環境変数が設定された場合、以下の優先順位に従って反映されます。

1. 環境変数 LC_ALL が設定されている場合、LC_ALL の値が使用されます。
2. LC_ALL 以外の LC_で始まる環境変数が設定されている場合、そのカテゴリにはその値が使用されます。
3. 環境変数 LANG が設定されている場合には、LANG の値が使用されます。
4. いずれの環境変数も設定されていない場合、標準のロケール(C ロケール[14])が使用されます。

環境変数は端末やプロセスごとに設定できます。そのため、ロケールの設定も端末やプロセスごとに行われることになります。

それぞれの環境変数に設定するロケール名は、*language*[_*territory*][.*codeset*][@*modifier*]という書式になります。*language* は ISO639[15]で規程される言語コードです。また、*territory* は ISO 3166 で規

[13]ここでのタイマーは、プロセスが使用する仮想的なタイマーのことです。
[14]Linux システムで互換性のあるロケール。POSIX ロケールとも呼ばれます。
[15]http://www.loc.gov/standards/iso639-2/php/code_list.php

程される国名コード[16]です。*codeset* は、ISO-8859-1 や UTF-8 のような文字集合や文字符号化識別子です。

locale コマンドに-a オプションを付けて実行することで、システムでサポートされているすべてのロケールを得ることができます。

```
[ATDE ~]$ locale -a
C
POSIX
en_US.utf8
ja_JP.utf8
```

図 3.35 システムでサポートされているすべてのロケールを得る(locale -a コマンド)

また、**locale** コマンドを引数なしで実行すると、現在の設定を確認することができます。

```
[ATDE ~]$ locale
LANG=ja_JP.UTF-8
LC_CTYPE="ja_JP.UTF-8"
LC_NUMERIC="ja_JP.UTF-8"
LC_TIME="ja_JP.UTF-8"
LC_COLLATE="ja_JP.UTF-8"
LC_MONETARY="ja_JP.UTF-8"
LC_MESSAGES="ja_JP.UTF-8"
LC_PAPER="ja_JP.UTF-8"
LC_NAME="ja_JP.UTF-8"
LC_ADDRESS="ja_JP.UTF-8"
LC_TELEPHONE="ja_JP.UTF-8"
LC_MEASUREMENT="ja_JP.UTF-8"
LC_IDENTIFICATION="ja_JP.UTF-8"
LC_ALL=
```

図 3.36 現在のロケールを確認する(locale コマンド)

「図 3.26. システムクロックの参照(date コマンド)」で示したように、ATDE3 で **date** コマンドを実行すると日本語で表示されます。これを、英語で表示するには環境変数 LANG を設定して **date** コマンドを実行します。

```
[ATDE ~]$ date
2010 年 10 月  5 日 火曜日 16:44:05 JST
[ATDE ~]$ LANG=en_US date
Tue Oct  5 16:44:05 JST 2010
```

図 3.37 ロケールを指定して date コマンドを実行

3.10. ネットワーク

近年の組み込みシステムでは、ネットワークシステムを持つものが増えています。Armadillo シリーズのすべての製品も、ネットワーク機能を有しています。Linux システムは様々なネットワーク機能を備え

[16]http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm

ており、このことが組み込みシステムで Linux を採用する動機となることも多いようです。ここでは、Linux システムでネットワークを扱う方法について説明します。

3.10.1. ネットワークインターフェース

Linux システムでは、「すべてのものはファイルである(Everything is a file)」という考え方の元、様々なものをファイルとして扱いますが、ネットワークインターフェースは例外です。ブロックデバイスやキャラクタデバイスの場合、デバイスファイル名を指定してファイルをオープンすることで、カーネル内のデバイスドライバにアクセスすることができます。ネットワークインターフェースの場合、インターフェース名でアクセスするインターフェースを指定します。

ネットワークインターフェースの状態を取得、設定するには、**ifconfig** コマンドを使用します。**ifconfig** コマンドを引数を指定せずに実行すると、動作中の(アップ状態の)すべてのインターフェースの状態を表示します。

```
[ATDE ~]$ /sbin/ifconfig
eth0      Link encap:イーサネット  ハードウェアアドレス xx:xx:xx:xx:xx:xx
          inet アドレス:192.168.0.1 ブロードキャスト:192.168.0.255  マスク:255.255.255.0
          inet6 アドレス: xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/64 範囲:リンク
          UP BROADCAST RUNNING MULTICAST  MTU:1500  メトリック:1
          RX パケット:729661 エラー:0 損失:0 オーバラン:0 フレーム:0
          TX パケット:180412 エラー:0 損失:0 オーバラン:0 キャリア:0
          衝突(Collisions):0 TX キュー長:1000
          RX バイト:283450776 (270.3 MiB)  TX バイト:186863017 (178.2 MiB)
          割り込み:18 ベースアドレス:0x2024

lo        Link encap:ローカルループバック
          inet アドレス:127.0.0.1 マスク:255.0.0.0
          inet6 アドレス: ::1/128 範囲:ホスト
          UP LOOPBACK RUNNING  MTU:16436  メトリック:1
          RX パケット:12 エラー:0 損失:0 オーバラン:0 フレーム:0
          TX パケット:12 エラー:0 損失:0 オーバラン:0 キャリア:0
      衝突(Collisions):0 TX キュー長:0
          RX バイト:840 (840.0 B)  TX バイト:840 (840.0 B)
[ATDE ~]$ LANG=C /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr xx:xx:xx:xx:xx:xx
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:730088 errors:0 dropped:0 overruns:0 frame:0
          TX packets:180413 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:283522733 (270.3 MiB)  TX bytes:186863431 (178.2 MiB)
          Interrupt:18 Base address:0x2024

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:840 (840.0 B)  TX bytes:840 (840.0 B)
```

図 3.38 ネットワークインターフェースの状態を取得する(ifconfig コマンド)

「eth0」、「lo」がインターフェース名です。Link encap がネットワークインターフェースの種類を示します。「eth0」は Eternet に対応したインターフェース名です。Ethenet インターフェースが二つ以上あるときは、「eth1」「eth2」と数値部分が増えていきます。「lo」はローカルループバックインターフェースです。自分自身がサーバーにもクライアントにもなるような場合に使用します。

ハードウェアアドレス(HWaddr)は MAC アドレス、inet アドレス(inet addr)は IPv4 の IP アドレス、ブロードキャスト(Bcast)はブロードキャストアドレス、マスク(Mask)はサブネットマスク、inet6 アドレス(inet6 addr)は IPv6 の IP アドレスをそれぞれ示します。それ以外は、ネットワークインターフェースの状態を示しています。

ifconfig コマンドはインターフェース名を指定して実行することで、特定のネットワークインターフェースの状態を取得、設定できます。例えば、eth0 の IP アドレスを「192.168.0.2」に変更するには、以下のようにします。

```
[ATDE ~]$ sudo ifconfig eth0 192.168.0.2
```

図 3.39 ネットワークインターフェースの状態を設定する(ifconfig コマンド)

ifconfig コマンドで設定したネットワークインターフェースの状態は、一時的なもので、再起動すると失われてしまいます。恒久的な設定は、/etc/network/interfaces ファイルに記述します。interfaces ファイルに記述した設定は、**ifup** コマンドでネットワークインターフェースをアップしたときに適用されます。

ATDE3 の標準設定では、interfaces ファイルは以下のようになっています。

```
[ATDE ~]$ cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
allow-hotplug eth0
iface eth0 inet dhcp
```

図 3.40 ATDE3 の interfaces ファイル

「auto lo」という行は、**ifup --all** を実行したときに「lo」をアップするよう指示しています。

「iface lo inet loopback」では、「lo」を TCP/IP ネットワークのローカルループバックインターフェースのインターフェース名として指定しています。

「allow-hotplug eth0」は、ホットプラグイベント[17]が発生した際に「eth0」をアップするよう指示しています。

「iface eth0 inet dhcp」は、「eth0」を TCP/IP ネットワークのインターフェース名として指定しています。また、ネットワーク設定は DHCP によって取得するよう指示しています。

/etc/network/interfaces ファイルの設定方法の詳細は、**man 5 interfaces** を参照してください。

[17]システムが動作しているときにデバイスが接続された場合に発生するイベント。

3.10.2. IP アドレスとポート番号

ネットワークで結ばれたコンピューター同士で通信を行う場合、実際にはそれぞれのコンピューター上で動作するプロセス間で通信を行うことになります。IP アドレスによって、ネットワーク上にあるコンピューターを識別することができます。しかし、IP アドレスだけでは、コンピューター上で動作するプロセスを識別することはできません。そこで、ポート番号を使用します。

接続を受け付けるプロセスは、ポート番号を指定して他のプロセスからの接続を待ち受けます。接続を行うプロセスは、IP アドレスとポート番号を指定して接続することで、特定のプロセスとの通信を開始することができます。

ポート番号は、0〜65535 の範囲内の数値を取ります。このうち、いくつかの番号は用途が決まっています。このようなポート番号をウェルノウンポート(well-known port)と呼びます。どのポート番号がどのような用途に使用されるかは、/etc/services ファイルに記述されています。また、**man 5 services** にも説明がありますので、参照してください。

netstat コマンドを使用すると、どのポートが現在使用されているか調べることができます。例えば、FTP サーバー(21 番ポートを使用)と Telnet サーバー(23 番ポートを使用)が動作しており、それぞれにクライアントからのアクセスがある場合、以下のように表示されます。

```
[ATDE ~]$ netstat -tanp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN      1284/inetd
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN      1284/inetd
```

図 3.41 使用中のポート番号を調べる(netstat コマンド)

3.10.3. ホスト名とリゾルバ

ネットワークに繋がっているコンピューターのことを、ホストまたはノードと呼びます。クロス開発においては、クロスコンパイルを行う作業用 PC をホストと呼び、開発対象をターゲットと呼んでいましたが、ここでは単にネットワークに接続されているコンピューターという意味でホストという言葉を使用します。

IP アドレスを使用すると、ネットワーク上のホストを識別することができます。しかし、IP アドレスは人間にとっては覚えにくいものです。そこで、IP アドレスに対応する名前を付けることができます。この、ホストを識別する名前をホスト名(hostname)といいます。ホスト名は、**hostname** コマンドを使用して調べることができます。

```
[ATDE ~]$ hostname
atde3
```

図 3.42 ホスト名を調べる(hostname コマンド)

ホスト名と IP アドレスの対応は、/etc/hosts ファイルに記述されています。ATDE3 では、以下の内容になっています。

```
[ATDE ~]$ cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       atde3

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
(略)
```

図 3.43 /etc/hosts ファイル

「localhost」というホスト名が「127.0.0.1[18]」、「atde3」というホスト名が「127.0.1.1」に対応付けられています。ホスト名から IP アドレスを特定することを、名前解決といいます。

ホスト名はローカルネットワークだけではなく、インターネットでも使用できます。例えば、「www.atamrk-techno.com」や「armadillo.atmark-techno.com」はホスト名です。「www.atamrk-techno.com」のようなインターネット上にあるホストの IP アドレスをすべて hosts ファイルに書くわけにはいきませんので、インターネットドメインネームシステム(DNS)という名前解決の仕組みがあります。DNS サーバーにホスト名を問い合わせると、対応する IP アドレスを返してくれます。

DNS へのアクセスに使用する機能は、C ライブラリが提供します。この機能のことをリゾルバ(resolver)といいます。リゾルバの設定ファイルは/etc/resolv.conf ファイルです。resolv.conf に DNS サーバーの IP アドレスを指定することができます。詳細は、**man 5 resolv.conf** を参照してください。

3.10.4. ネットワークの状態を調べる

ネットワーク設定が正しく行われたことを確認するため、ホスト同士で通信が可能かを調べるには、**ping** コマンドが使用できます。**ping** コマンドは、対象のホストへパケットを送信し、対象のホストはパケットを受信すると応答パケットを返信します。対象ホストのパケットのやりとりが正常にできれば、最低限のネットワーク設定は正しいことを確認することができます。

通信可能なホストを指定して **ping** コマンドを実行すると、以下のようにホストからの応答が表示されます。

```
[ATDE ~]$ ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=3.99 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.000 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.000 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=0.000 ms
64 bytes from 192.168.0.2: icmp_seq=5 ttl=64 time=0.000 ms
^C
--- 192.168.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
```

図 3.44 ネットワークの到達確認: 成功(ping コマンド)

ホストとの通信ができなかった場合、以下のような表示になります。

[18]ローカルループバックインターフェースの IP アドレス。

```
[ATDE ~]$ ping 192.168.100.2
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
From 172.16.23.10 icmp_seq=2 Destination Host Unreachable
From 172.16.23.10 icmp_seq=3 Destination Host Unreachable
From 172.16.23.10 icmp_seq=4 Destination Host Unreachable
^C
--- 192.168.100.2 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4004ms
```

図 3.45 ネットワークの到達確認: 失敗(ping コマンド)

4. Armadillo 上に Debian GNU/Linux を構築する

この章では、本格的な開発に入る前にターゲット側となる Armadillo の準備をおこないます。開発効率を上げるために、Armadillo に Debian GNU/Linux をインストールします。

Armadillo の標準ルートファイルシステムは、Atmark Dist で作成された initrd です。initrd を使う方法は実環境での運用には向いていますが、ルートファイルシステムの空き容量が少ない、変更を簡単に保存できないなどの問題があるので開発段階には適切ではありません。

Armadillo 上に Debian GNU/Linux を構築しておけば、開発に役立つ様々なツールを **apt-get** コマンド一発でインストールできたり、セルフコンパイルできるようになったりと、大変便利です。Debian GNU/Linux のバージョンは、ATDE3 と同じ 5.0(コードネーム "lenny")を使用します。

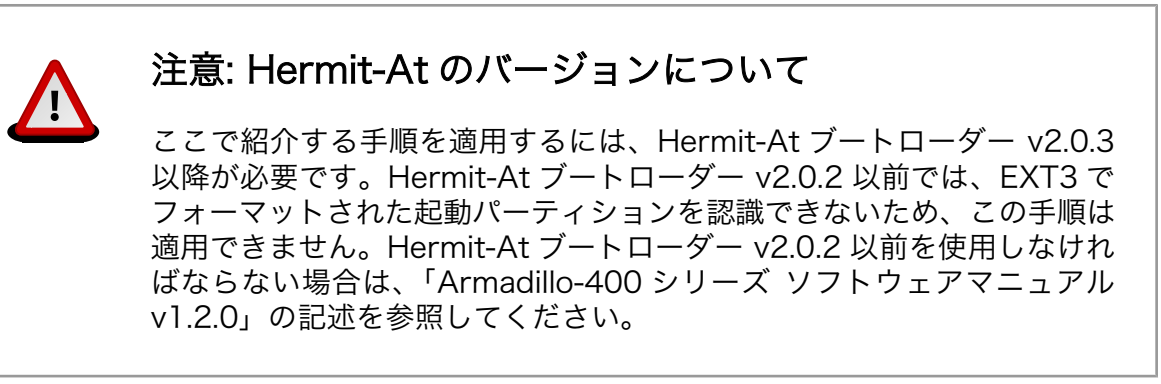
4.1. 構築手順

Armadillo-400 シリーズでは、カーネルイメージは microSD カードに、ユーザーランドのルートファイルシステムは microSD カードまたは USB メモリにも配置することができます。ここでは、例として microSD カードにカーネルイメージとルートファイルシステム両方を配置する手順を説明します。microSD カードは、1GB 以上の容量のものが必要です。

microSD カードにルートファイルシステムを構築し、/boot ディレクトリにカーネルイメージファイルを配置します。どのデバイスからカーネルイメージをロードするかは、Hermit-At のブートオプションで指定します。また、ルートファイルシステムがどこにあるかは、カーネルパラメーターで指定します。これも、Hermit-At で設定します。

Armadillo に Debian GNU/Linux をインストールする具体的な手順は、以下のようになります。

1. microSD カードにパーティションを作成し EXT3 ファイルシステムでフォーマットする。
2. カーネルイメージファイルを microSD カードにコピーする。
3. ルートファイルシステムを microSD カードにコピーする。
4. Hermit-At の設定を行う。



4.1.1. パーティションの作成とフォーマット

まず、microSD に 1 つのパーティションを作成し、EXT3 ファイルシステムでフォーマットします。

Armadillo-420/440 の microSD スロットに microSD カードを挿入[1] した後、標準イメージで起動するとカードが認識され、以下のような表示がコンソール(シリアルインターフェース 1)に表示されます。この表示は、microSD カードによって異なります。

```
mmc0: new high speed SD card at address b368
mmcblk0: mmc0:b368 SD    980992KiB
 mmcblk0: p1
```

図 4.1 microSD カード挿入時のカーネルメッセージ

「mmcblk0: p1」という表示から、microSD カードに対応するブロックデバイスが/dev/mmcblk0 に作成され、一つのパーティションがあることが分かります。microSD カードに二つのパーティションがある場合、「mmcblk0: p1」の次の行に「mmcblk0: p2」と表示されます。

カードが認識されたら、**fdisk** コマンドを使用して microSD カードのパーティションを構成しなおします。既存のパーティションをすべて削除して、一つのパーティションとします。

```
[armadillo ~]# fdisk /dev/mmcblk0
The number of cylinders for this disk is set to 124277.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
(e.g., DOS FDISK, OS/2 FDISK)
Command (m for help): d    ❶
Selected partition 1

Command (m for help): n    ❷
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-124277, default 1):    ❸
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-124277, default 124277):    ❹
Using default value 124277
Command (m for help): w    ❺
The partition table has been altered!
Calling ioctl() to re-read partition table.
mmcblk0: p1
mmcblk0: p1
Syncing disks.
[armadillo ~]#
```

図 4.2 パーティション作成手順

❶　まずは、既存のパーティションを削除します。複数のパーティションがある場合は、すべて削除してください。

[1]Armadillo-400 シリーズの microSD スロットは、ロック式になっています。microSD カードの着脱方法に関しては「Armadillo-400 シリーズハードウェアマニュアル」をご参照ください。

❷ 新しくプライマリパーティションを作成します。

❸ 開始シリンダにはデフォルト値(1)を使用するので、そのまま Enter キーを押してください。

❹ 最終シリンダにもデフォルト値(124277)を使用するので、そのまま Enter キーを押してください。

❺ 変更を microSD に書き込みます。

作成したパーティションに対応するブロックデバイスは、/dev/mmcblk0p1 となります。

パーティション作成での注意点

使用する microSD カードによって仕様が異なるため、表示されるシリンダ数は手順通りとはならない場合があります。

続いて、作成したパーティションを EXT3 ファイルシステムでフォーマットします。フォーマットには **mke2fs** コマンドを使用します。-j オプションを指定することで、EXT3 ファイルシステムとして指定したブロックデバイスをフォーマットします。また、**tune2fs** コマンドでファイルシステムのオプションを変更します。-i 0 オプションを付けることで、時間経過によるファイルシステムのチェックを行わないようにします。これは、Armadillo のリアルタイムクロックが大きくずれていた際に、ファイルシステムチェックが行われるのを抑制するためです。

```
[armadillo ~]# mke2fs -j /dev/mmcblk0p1
mke2fs 1.25 (20-Sep-2001)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
497984 inodes, 994220 blocks
49711 blocks (5%) reserved for the super user
First data block=0
31 block groups
32768 blocks per group, 32768 fragments per group
16064 inodes per group
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 35 mounts or
180.00 days, whichever comes first. Use tune2fs -c or -i to override.
[armadillo ~]# tune2fs -i 0 /dev/mmcblk0p1
tune2fs 1.25 (20-Sep-2001)
Setting interval between check 0 seconds
```

図 4.3 ファイルシステム作成手順

4.1.2. カーネルイメージファイルのコピー

microSD から起動する場合は、起動パーティションの/boot ディレクトリにカーネルイメージを配置する必要があります。対応しているカーネルイメージファイルの形式は、非圧縮カーネルイメージ(Image、linux.bin)または、圧縮イメージ(Image.gz、linux.bin.gz)のどちらかになります。

ここで説明する例では、カーネルイメージの取得に **wget** コマンドを使用します。**wget** コマンドで指定する URL は製品によって異なりますので、以下の表を参照し適宜読み替えてください。

表 4.1 カーネルイメージのダウンロード先 URL

製品	URL
Armadillo-420	http://download.atmark-techno.com/armadillo-420/image/linux-a400-*[バージョン]*.bin.gz
Armadillo-440	http://download.atmark-techno.com/armadillo-440/image/linux-a400-*[バージョン]*.bin.gz

以下に Armadillo-440 での例を示します。

```
[armadillo ~]# mount /dev/mmcblk0p1 /mnt/
[armadillo ~]# mkdir /mnt/boot
[armadillo ~]# cd /mnt/boot
[armadillo /mnt/boot]# wget http://download.atmark-techno.com/armadillo-440/image/linux-a400-[バージョン].bin.gz
[armadillo /mnt/boot]# mv linux-a400-[バージョン].bin.gz /mnt/boot/linux.bin.gz
[armadillo /mnt/boot]# cd
[armadillo ~]# umount /mnt
```

図 4.4 カーネルイメージの配置

4.1.3. ルートファイルシステムの構築

Armadillo-400 シリーズ用に設定済みの Debian GNU/Linux ルートファイルシステムのアーカイブは、付属 DVD の debian ディレクトリか、弊社ダウンロードサイトから取得できます。アーカイブを microSD カードに展開することでルートファイルシステムを構築することができます。

ここで説明する例では、アーカイブの取得に **wget** コマンドを使用します。**wget** コマンドで指定する URL は製品によって異なりますので、以下の表を参照し適宜読み替えてください。

表 4.2 Debian アーカイブのダウンロード先 URL

製品	URL
Armadillo-420/440 共通	http://download.atmark-techno.com/armadillo-4x0/debian/debian-lenny-armel-*#*.tgz[1]
	http://download.atmark-techno.com/armadillo-4x0/debian/debian-lenny-armel-a4x0.tgz[2]

[1]「#」の部分は 1〜5 まで

[2]「a4x0」は置き換えを行わず、そのまま使用してください。

```
[armadillo ~]# mount /dev/mmcblk0p1 /mnt
[armadillo ~]# mkdir tmp
[armadillo ~]# mount -t ramfs ramfs tmp
[armadillo ~]# cd tmp
[armadillo ~/tmp]# for N in 1 2 3 4 5 a4x0; do
> wget http://download.atmark-techno.com/armadillo-4x0/debian/debian-lenny-armel-${N}.tgz;
> gzip -cd debian-lenny-armel-${N}.tgz | (cd /mnt; tar xf -);
> sync;
> rm -f debian-lenny-armel-${N}.tgz;
> done
[armadillo ~/tmp]# cd
[armadillo ~]# umount tmp
[armadillo ~]# rmdir tmp
[armadillo ~]# umount /mnt
```

図 4.5 ルートファイルシステムの構築

4.1.4. ブートデバイスとカーネルパラメーターの設定

カーネルイメージをロードする場所は、Hermit-At のブートデバイス設定で指定します。また、ユーザーランドの場所は、カーネルパラメーターで指定します。

Hermit-At の設定は保守モードでおこないます。一度 Armadillo の電源を切り、タクトスイッチ(SW1)を押しながら電源を投入し、Hermit-At の保守モードで起動してください。

カーネルイメージをロードするデバイスの指定は、**setbootdevice** コマンドを使用します。microSD カードのパーティション 1 に配置したカーネルイメージで起動するためには、以下のコマンドを実行してください。

```
hermit> setbootdevice mmcblk0p1
```

図 4.6 ブートデバイスを microSD カードに指定する

カーネルパラメーターの指定は、**setenv** コマンドを使用します。ルートファイルシステムを microSD カードのパーティション 1 にする場合は、以下のコマンドを実行してください。

```
hermit> setenv console=ttymxc1 root=/dev/mmcblk0p1 noinitrd rootwait
```

図 4.7 ルートファイルシステムを microSD カードに指定する

Hermit-At の設定を元に戻す

Hermit-At の設定を初期状態に戻す(カーネルイメージをロードするデバイスとルートファイルシステムの場所をフラッシュメモリにする)には、以下のコマンドを実行してください。

```
hermit> setbootdevice flash
hermit> clearenv
```

図 4.8 Hermit-At の設定を元に戻す

以上の設定をおこない、**boot** コマンドを実行するか再起動すると、microSD カード上に構築された Debian GNU/Linux で起動します。

4.2. Debian GNU/Linux インストール後にまずすること

Armadillo にインストールした直後の Debian GNU/Linux は、最小限の設定しかされていない状態です。この章では、最低限行わなければならない設定について説明します。

4.2.1. ログインと一般ユーザーの追加

インストール直後の Debian GNU/Linux では、特権ユーザーである root ユーザーしかいません。そのため、ログインは root ユーザーで行います。パスワードなしでログインできます。

```
Debian GNU/Linux 5.0 debian ttymxc1

debian login: root
Linux debian 2.6.26-at10 #1 PREEMPT Thu Aug 19 16:17:48 JST 2010 armv5tejl

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
debian:~#
```

図 4.9 Armadillo 上の Debian GNU/Linux へログインする

以降は、Armadillo 上で動作する Debian GNU/Linux のプロンプトとして、以下の表記を用います。

```
[darmadillo ~]#
```

図 4.10 Armadillo 上の Debian GNU/Linux のプロンプト表記

一般ユーザーがいないと何かと不便ですので、まずは一般ユーザーを追加します。一般ユーザーの名前は guest、パスワードも guest としましょう。

```
[darmadillo ~]# useradd -m guest
[darmadillo ~]# passwd guest
Enter new UNIX password: guest
Retype new UNIX password: guest
passwd: password updated successfully
```

図 4.11 一般ユーザー(guest)の追加

あくまで開発用ですので、root ユーザーのパスワードは設定しません。

4.2.2. 時刻の設定

続いて、時刻の設定をおこないます。時刻を正しく設定しておかないと、**apt-get install** を実行する際にワーニングが大量に表示されるなど、色々な問題が発生します。

まずは、**dpkg-reconfigure tzdata** でタイムゾーンを設定します。テキストベースのメニュー画面が表示されますので、タイムゾーンを JST に設定するには[Asia]-[Tokyo]を選択してください。

```
[darmadillo ~]# dpkg-reconfigure tzdata
Package configuration
 lqqqqqqqqqqqqqqqqqqqqqqqqqqqu Configuring tzdata tqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x Please select the geographic area you live in. Subsequent configuration   x
 x questions will narrow this down by presenting a list of cities,           x
 x representing the time zones in which they are located.                    x
 x                                                                           x
 x Geographic area:                                                          x
 x                                                                           x
 x                              Australia                                    x
 x                              Arctic        a                              x
 x                              Asia                                         x
 x                              Atlantic      a                              x
 x                              Europe        a                              x
 x                              Indian        a                              x
 x                              Pacific       a                              x
 x                              SystemV       a                              x
 x                              Etc                                          x
 x                                                                           x
 x                                                                           x
 x                   <Ok>                       <Cancel>                     x
 x                                                                           x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj

Package configuration
        lqqqqqqqqqqqqqqqqqqqqqu Configuring tzdata tqqqqqqqqqqqqqqqqqqqqqk
        x Please select the time zone corresponding to your location.  x
        x                                                              x
        x Time zone:                                                   x
        x                                                              x
        x                       Samarkand                              x
        x                       Seoul           a                      x
        x                       Shanghai        a                      x
        x                       Singapore       a                      x
        x                       Taipei          a                      x
        x                       Tashkent        a                      x
        x                       Tbilisi         a                      x
        x                       Tehran          a                      x
        x                       Tel_Aviv        a                      x
        x                       Thimphu         a                      x
        x                       Tokyo                                  x
        x                                                              x
        x                                                              x
        x              <Ok>                   <Cancel>                 x
        x                                                              x
        mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj


Current default timezone: 'Asia/Tokyo'
Local time is now:      Wed Oct  6 06:54:23 JST 2010.
Universal Time is now:  Tue Oct  5 21:54:23 UTC 2010.
```

図 4.12 タイムゾーンの設定

次に、システムクロックを設定します。この時点では ntp クライアントが使用できませんので、**date** コマンドで手動設定します。リアルタイムクロックが Armadillo-420/440 に接続されており、適切に

設定されている場合、システムクロックは起動時に自動で設定されます。そのため、この手順は不要です。システムクロックが現在時刻と大きくずれている場合だけ実行してください。

```
[darmadillo ~]# date -s "2010/09/28 00:00:00"
```

図 4.13 システムクロックを手動設定する

システムクロックを設定した後は、それをハードウェアクロック(リアルタイムクロック)にも反映させます。ハードウェアクロックの設定は **hwclock** コマンドを使用しますが、DebianGNU/Linux ではそれをラップした hwclock.sh スクリプトがあるので、それを使用します。

```
[darmadillo ~]# /etc/init.d/hwclock.sh restart
```

図 4.14 システムクロックをハードウェアクロックに反映する

4.2.3. パッケージのアップデート

Debian GNU/Linux では、セキュリティアップデートやバグフィックスのためパッケージが随時更新されています。Armadillo にインストールした Debian GNU/Linux のアーカイブは最新の状態ではありませんので、パッケージをアップデートする必要があります。

パッケージの管理には、**apt-get** コマンドを使用します。**apt-get update** でローカルに持っているパッケージのリストを最新のものに更新し、**apt-get upgrade** で更新可能なパッケージをすべてインストールします。

```
[darmadillo ~]# apt-get update
[darmadillo ~]# apt-get upgrade
```

図 4.15 Debian パッケージのアップデート

4.2.4. FTP サーバーのインストール

ATDE と Armadillo で簡単にファイルを送受信するための方法の 1 つとして FTP で転送する方法があります。この章では Armadillo 上の Debian GNU/Linux に FTP サーバーを構築する方法について説明します。

パッケージのインストールには、**apt-get install** を使用します。FTP サーバーのパッケージ名は、ftpd です。

```
[darmadillo ~]# apt-get install ftpd
[darmadillo ~]# /etc/init.d/openbsd-inetd start
```

図 4.16 FTP サーバーのインストール

ftpd は、スーパーサーバー(inetd)を経由して起動されます。そのため、ftpd パッケージをインストールすると、openbsd-inetd パッケージも一緒にインストールされます。Armadillo 上で動作する FTP サーバーに外部からアクセス可能にするには、inetd を起動しておく必要があります。なお、次回起動時からは inetd は自動起動されるように、インストールした時点で設定されています。

```
[darmadillo ~]# /etc/init.d/openbsd-inetd start
Starting internet superserver: inetd.
```

図 4.17 inetd の起動

ATDE3 などから FTP で Armadillo に接続する際、ユーザー名とパスワードを要求されます。ftpd が使用するユーザー名とパスワードは、一般ユーザーのものになります。そのため、ユーザー名 guest、パスワード guest でアクセスすることができます。

```
[ATDE ~]$ lftp Armadillo の IP アドレス -u guest
パスワード: guest
```

図 4.18 lftp で Armadillo の FTP サーバーに接続

lftp の便利な使い方

lftp コマンドでは、-e オプションで FTP コマンドを指定することができます。何度も同じファイルを Armadillo に転送する必要がある場合、一行でコマンドを書いておき、シェルのヒストリー機能を活用することで何度も同じコマンドを入力せずに済みます。

以下のコマンド例では、guest ユーザーでログインし、ファイルを送信しています。一度ファイルを削除しているのは、**lftp** では同じファイル名のファイルがある場合、上書きしないため何度も同じコマンドを実行してもファイルが更新されないためです。また、パスワードをコマンドラインに書いてしまっています。コマンドの履歴は同じコンピューターを使用する別のユーザーが見ることができるので、通常、コマンドラインに直接パスワードを書くことは望ましくありません。コンピューターを共有している場合は気をつけてください。

```
[ATDE ~]$ lftp -u guest,guest Armadillo の IP アドレス -e "rm ファイル
名;put ファイル名;quit"
```

図 4.19 lftp を使用して Armadillo にファイルを転送

4.2.5. Telnet サーバーのインストール

Armadillo 上で動作する Debian GNU/Linux に Telnet サーバーを動作させておくと、ATDE からネットワーク経由で Armadillo を操作できるようになります。シリアルインターフェースを端末として使用できない場合などに便利ですので、本章では Telnet サーバーの構築方法について説明します。

Telnet サーバーのインストールは、いつものように **apt-get install** を使用します。パッケージ名は telnetd です。

```
[darmadillo ~]# apt-get install telnetd
```

図 4.20 Telnet サーバーのインストール

telnetd も ftpd と同様に、inetd を介して起動されます。そのため、inetd が起動していない場合は、起動する必要があります。「図 4.17. inetd の起動」を参照してください。

ATDE3 などから Armadillo の Telnet サーバーにアクセスするには、**telnet** コマンドを使用します。FTP サーバーと同様に、一般ユーザー guest でログインすることができます。

```
[ATDE ~]$ telnet Armadillo の IP アドレス
Trying Armadillo の IP アドレス...
Connected to Armadillo の IP アドレス.
Escape character is '^]'.
Debian GNU/Linux 5.0
debian login: guest
Password: guest
```

図 4.21 telnet で Armadillo の Telnet サーバーに接続

4.3. セルフコンパイル用ツールチェインのインストール

複雑な依存関係を持ったソフトウェアなどでは、クロスコンパイルが難しい場合もあります。そのような時のために、Armadillo 上にツールチェインをインストールしてセルフコンパイルできるようにする方法を紹介します。

build-essential パッケージをインストールすると、開発に最低限必要なパッケージをインストールすることができます。

```
[darmadillo ~]# apt-get install build-essential
```

図 4.22 ツールチェインのインストール

依存関係に基づいて、gcc、g++、binutils、libc6-dev、make などがインストールされます。その他のライブラリなどは、必要に応じてインストールしてください。これは、ATDE3 上でセルフコンパイルする場合と、変わりありません。

高速な CPU を搭載した作業用 PC でクロスコンパイルする場合と比べ、Armadillo 上でセルフコンパイルすると格段に時間がかかります。しかし、クロスコンパイルがどうしても難しいという場合もあるので、最後の手段として、このような方法もあることを覚えておいてください。本書では、これ以降セルフコンパイルについては取り上げません。

5. シェルスクリプトプログラミング

C 言語でのプログラミングに入る前に、本章では、シェルスクリプトを用いたプログラミングについて説明します。

GUI を前提とした Windows とは異なり、Linux を含む UNIX システムでは、基本的にすべての操作をコマンドラインインターフェースから行うことができます。

シェルは、コマンドラインプロンプトから入力されたコマンドを、Linux システムに受け渡す機能を持ったプログラムです。シェルは、ユーザーから入力されたコマンドを一つ一つ実行するだけでなく、ファイルに記述されたコマンドを逐次実行するインタープリタとしての機能も有しています。

シェルが解釈できる形式で記述されたプログラムを、シェルスクリプトと呼びます。

シェルは構造化プログラミングが可能で、その機能は非常に強力です。Linux の豊富なコマンドと合わせることで、アプリケーションプログラムのプロトタイプをシェルスクリプトで記述することもできるでしょう。

5.1. シェルの種類

Linux で最も一般的なシェルは、bash(Bourne-Again Shell)です。Debian GNU/Linux でも、標準のシェルとして採用されています。

Atmark Dist を使用した場合は、bash の代わりに BusyBox の ash が使われます。ash は、bash と良く似ていますが一部の機能が省かれており、軽量ですので組み込みシステムに向いています。

一つの Linux システムに複数のシェルをインストールすることも可能です。システム標準のシェルを使用したい場合は、`/bin/sh` を使用します。Debian GNU /Linux などでは、`/bin/sh` は`/bin/bash` へのシンボリックリンクとなっています。Atmark Dist を使用した場合は、`/bin/busybox`(ash)へのシンボリックリンクとなっています。

5.2. シェルスクリプトの書き方

シェルは、単にコマンドを一つ一つ実行するだけでなく、変数も使うことができ、for や if などの構造化プログラミングが可能な構文を解釈することができます。

例えば、ルートディレクトリ直下のディレクトリをすべて表示するには、以下のようにします。構文については、詳しくは「5.3. シェルの構文」で説明しますが、ここではルートディレクトリ直下のディレクトリ名を dir という変数に代入して、順番に **echo** していると考えてください。

```
[darmadillo ~]$ for dir in /*; do echo $dir; done
/bin
/boot
/dev
/etc
/home
/lib
/lost+found
/media
/mnt
/opt
/proc
/root
/sbin
/selinux
/srv
/sys
/tmp
/usr
/var
```

図 5.1 for 文の例 1

これは、以下のように複数行に分けて書くこともできます。

```
[darmadillo ~]$ for dir in /*
> do
> echo $dir
> done
/bin
/boot
/dev
/etc
/home
/lib
/lost+found
/media
/mnt
/opt
/proc
/root
/sbin
/selinux
/srv
/sys
/tmp
/usr
/var
```

図 5.2 for 文の例 2

コマンド入力の 2 行目から、プロンプトが「>」に変わっていることに注目してください。コマンドの終了はシェルが自動的に判断し、入力されたコマンドを実行してプロンプトを元に戻します。

シェルは、シェルスクリプトとしてファイルに記述されたコマンドを実行することもできます。

```
#!/bin/sh

#シェルスクリプトの例

for dir in /*; do
    echo $dir
done
```

図 5.3 シェルスクリプトの例(example.sh)

シェルスクリプトの決まり事として、先頭行にある「#!」の後にそのスクリプトを処理するプログラムを指定することできます。今回の場合は、「/bin/sh」を指定しているのでシステムの標準シェルでスクリプトが実行されます。

先頭行以外にある「#!」以外の「#」以降の文字列はコメントして扱われます。そのため、「#シェルスクリプトの例」という行はコメントになり、実行結果に影響を与えません。

上記プログラムを実行すると、以下のようになります。

```
[darmadillo ~]$ chmod +x example.sh
[darmadillo ~]$ ./example.sh
/bin
/boot
/dev
/etc
/home
/lib
/lost+found
/media
/mnt
/opt
/proc
/root
/sbin
/selinux
/srv
/sys
/tmp
/usr
/var
```

図 5.4 シェルスクリプト実行例

シェルは拡張子を判別しないので、「.sh」という拡張子はつける必要はありません。

5.3. シェルの構文

本章では、シェルの基本的な構文について説明します。bash と ash の構文の違いについても指摘します。bash/ash の構文規則は、本章で説明するものがすべてではありません。より詳細な説明は、bash の man ページを参照してください。

5.3.1. 基本的なコマンドの実行方法

シェルでコマンドを実行するには、以下のように記述します。

```
[darmadillo ~]$ echo hello world
hello world
[darmadillo ~]$ echo hello          world
hello world
```

図 5.5 シェルでコマンドを実行する

先頭に実行するコマンドの名前を指定します。「echo」は、複数の引数をとり、各引数を表示するプログラムです。コマンドと引数、引数と引数の間はスペースで区切ります。スペースで区切られた、シェルが 1 単位とみなす文字列を単語といいます。単語は、スペース以外に「|、&、;、(、)、タブ」で区切ることができます。単語の区切りとなる文字をメタ文字といいます。

メタ文字は複数あっても構いませんので、上記二つのコマンドは等価です。

引数に空白を含めたい場合は、「"」か「'」で囲みます。

```
[darmadillo ~]$ echo "hello          world"
hello          world
```

図 5.6 シェルでコマンドを実行する(空白を含む引数)

5.3.2. 変数

シェルでは、変数を使用することができます。C 言語と異なり、変数の宣言は必要ありません。標準では、すべての変数は文字列型として扱われます。

変数名は、英数字と「_」(アンダースコア)だけから構成され、かつ最初の文字が英字か「_」である必要があります。英字の大文字と小文字は区別されます。

変数に値を代入する際には、「=」を使用します。「=」の前後に空白を入れてはいけません。空白を含む文字列を代入する場合は、「"」又は「'」で囲む必要があります。また、変数の値を参照するときには、変数名の前に「$」を付けます。または、{}(ブレース)を使用して${変数名}と記述することもできます。

```
[darmadillo ~]$ variable=hello
[darmadillo ~]$ echo $variable
hello
[darmadillo ~]$ variable='hello world'
[darmadillo ~]$ echo $variable
hello world
[darmadillo ~]$ variable=1+2
[darmadillo ~]$ echo $variable
1+2
```

図 5.7 変数の例

変数に対して算術演算を行う場合は、$(())構文を使います。

```
[darmadillo ~]$ variable=1
[darmadillo ~]$ echo $variable
1
[darmadillo ~]$ variable=$(($variable+1))
[darmadillo ~]$ echo $variable
2
[darmadillo ~]$ variable=$(($variable*2))
[darmadillo ~]$ echo $variable
4
```

図 5.8 算術演算の例

bash では変数の 1 次元配列を扱うこともできます。残念ながら、ash では配列を扱うことができないので、ここでは説明を省きます。

5.3.3. 環境変数

bash では、いくつかの変数が環境変数(シェル変数)としてあらかじめ設定されています。環境変数の名前は、大文字になっていますので、これらと混同しないようにユーザーが設定する変数には小文字を使うのが良いでしょう。

以下に、主な環境変数のリストを示します。

表 5.1 主な環境変数のリスト

環境変数	説明
HOME	現在のユーザのホームディレクトリ。
PWD	現在の作業ディレクトリ。
OLDPWD	1 つ前の作業ディレクトリ。
PATH	コマンドの検索パス。シェルがコマンドを検索するディレクトリをコロンで区切って並べたリスト。
IFS	内部フィールド区切り文字。デフォルト値は``<空白><タブ><改行>''。
PS1	プライマリのプロンプト文字列。PS1 を変更することでプロンプトの表示をカスタマイズすることができる。
PS2	セカンダリのプロンプト文字列。「図 5.2. for 文の例 2」のように複数行に分けてコマンドを記述した場合に使用される。

env コマンドを使用すると、すべての環境変数を表示することができます。

```
[darmadillo ~]$ env
SHELL=/bin/sh
TERM=vt100
HUSHLOGIN=FALSE
USER=Armadillo
MAIL=/var/mail/Armadillo
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games
PWD=/home/Armadillo
SHLVL=1
HOME=/home/Armadillo
LOGNAME=Armadillo
_=/usr/bin/env
```

図 5.9 すべての環境変数の表示

新しく環境変数を設定するには、**export** コマンドを使います。

```
[darmadillo ~]$ export MY_ENV=value
[darmadillo ~]$ echo $MY_ENV
value
```

図 5.10 環境変数の設定

5.3.4. パラメータ

パラメータは、値を保持するためのものです。パラメータは名前、数字、特定の特殊文字のいずれかで表現されます。変数とは、名前で表現されたパラメータにすぎません。

変数以外のパラメータとして、位置パラメータ、特殊パラメータがあります。

位置パラメータは、1 以上の数値で表されるパラメータです。位置パラメータには、シェルの引数が代入されます。第 1 引数は$1 に、第 2 引数は$2 にというように順番に代入されていきます。10 以上の位置パラメータを参照する場合には、${10}のように、{}をつけます。

シェル関数の中では、位置パラメータは関数の引数に置き換えられます。(「5.3.11. 関数」参照。)

特殊パラメータは参照だけが可能であり、値を代入することはできないパラメータです。「表 5.2. 特殊パラメータのリスト」に、bash と ash で使用可能な特殊パラメータの一覧を示します。

表 5.2 特殊パラメータのリスト

特殊パラメータ	説明
*	すべての位置パラメータに展開される。
@	すべての位置パラメータに展開される。[1]
#	位置パラメータの個数に展開される。
?	最後に実行されたフォアグラウンドプロセスの終了ステータスに展開される。
!	最後に実行されたバックグラウンドプロセスのプロセス ID に展開される。
$	プロセス ID に展開される。
0	シェルまたはシェルスクリプトの名前に展開される。

[1]ダブルクォート内部での展開のされ方が$*と異なります。「5.3.6. 展開」参照。

5.3.5. クォート

クォートを使うと、特殊文字の意味や後述する展開を無効にすることができます。クォートの方法には、エスケープ文字、シングルクォート、ダブルクォートの 3 種類があります。

クォートされていないバックスラッシュ「\」[1]をエスケープ文字といいます。エスケープ文字の直後の特殊文字は、特殊文字としての意味を失います。

シングルクォート「'」で文字を囲むと、クォート内部では特殊文字は特殊文字としての意味を失い、展開もおこなわれません。シングルクォートの間にシングルクォートを置くことはできません。

ダブルクォート「"」で文字を囲むと、クォート内部では「$、`、\」以外の特殊文字は、特殊文字としての意味を失います。「$」と「`」は、特殊文字としての意味を保持します。バックスラッシュは、直後の文字が「$、`、"、<newline>」のいずれかの場合、特殊文字としての意味を保持します。

```
[darmadillo ~]$ echo $HOME
/home/Armadillo
[darmadillo ~]$ echo \$HOME
$HOME
[darmadillo ~]$ echo '$HOME'
$HOME
[darmadillo ~]$ echo "$HOME"
/home/Armadillo
[darmadillo ~]$ echo "\$HOME"
$HOME
```

図 5.11 クォートの例

5.3.6. 展開

シェルでは、特定の書き方をした場合、別の文字列として置き換えられて解釈されることがあります。この置換処理を展開といいます。

展開は、ブレース展開、チルダ展開、パラメータ・変数・算術式展開、コマンド置換(左から右へ)、単語分割、パス名展開の順番で行われます。

ブレース展開では、a{D,C,B}e が aDe aCe aBe に展開されます。bash では使用可能ですが、ash では使用できません。

チルダ「~」で単語が始まった場合、スラッシュよりも前にある文字が、チルダプレフィックスと解釈されます。有効なチルダプレフィックスの場合、チルダ展開が行われます。

チルダプレフィックスがユーザー名と一致した場合、そのユーザーのホームディレクトリに展開されます。ユーザー名が指定されない場合は、シェルを実行しているユーザのホームディレクトリに展開されます。

チルダプレフィックスが+の場合、環境変数 PWD に、-の場合、環境変数 OLDPWD に展開されます。(この展開は、ash ではおこなわれません。)

[1]環境によっては「¥」と表示されることもありますが同じ意味です。

```
[darmadillo ~]$ echo ~root
/root
[darmadillo ~]$ echo ~Armadillo/file
/home/Armadillo/file
[darmadillo ~]$ echo ~unavailuser
~unavailuser
[darmadillo ~]$ pwd
/home/Armadillo
[darmadillo ~]$ cd /
[darmadillo /]$ echo ~+
/
[darmadillo /]$ echo ~-
/home/Armadillo
```

図 5.12 チルダ展開の例

「$」文字があると、パラメータ展開、コマンド置換、算術式展開が行われます。展開されるパラメータ名やシンボルは、ブレースで括ることもできます。ブレースは省略可能ですが、変数の直後に変数名の一部と解釈できる文字が置かれた場合に、その文字と共にパラメータが展開されてしまうのを防ぐために用意されています。

パラメータ展開では、これまで説明してきた変数、環境変数、位置パラメータ、特殊パラメータの展開が行われます。

ダブルクォートの内部で$*の展開が行われた時は、それぞれのパラメータを IFS での最初の文字で区切って並べた 1 つの単語に展開されます。IFS が設定されていなければ、パラメータは空白で区切られます。IFS が空文字列の場合、すべてのパラメータはつなげられます。

ダブルクォートの内部で$@の展開が行われた時は、それぞれのパラメータは別々の単語に展開されます。位置パラメータがない場合は、空文字に展開されます。(つまり、取り除かれます。)

$(command)または、`command`と書くと、command の実行結果に置き換えられます。

```
[darmadillo ~]$ var=$(date)
[darmadillo ~]$ echo $var
Fri Sep 3 15:21:23 JST 2010
```

図 5.13 コマンド置換の例

$((expression))と書くと、expression を算術式評価して、その結果に置き換えられます。

シェルは上記の展開を行った後、IFS に指定されたそれぞれの文字を区切り文字として、単語に分割します。

単語分割に続いて、パス名展開をおこないます。単語分割を行った後の単語が、「*、?、[」を含んでいた場合、その単語はパターンとみなされます。パターンに一致するファイルがあれば、その単語はマッチするファイル名をアルファベット順にソートしたリストに置換されます。マッチするファイル名がない場合、その単語は置き換えられません。

「*」は、空文字列を含む任意の文字列にマッチします。また、「?」は、任意の 1 文字にマッチします。

「[」と「]」で文字列を括ると、マッチする文字のパターンを指定できます。単に 1 文字以上の文字を指定した場合は、指定した文字のうち、任意の 1 文字にマッチします。2 つの文字の間にハイフンをい

れたものを指定した場合、指定した文字とその間の範囲にある文字にマッチします。また、「[」の直後に「!」または「^」を指定した場合は、指定した文字以外の任意の文字がマッチします。

「[」と「]」の間には、文字クラスを指定することができます。文字クラスは、[:class:]のように「:」コロンで括ります。文字クラスには、alnum、alpha、ascii、blank、cntrl、digit、graph、lower、print、punct、space、upper、xdigit があります。

最後に、クウォートの削除がおこなわれます。クウォートされていない「\、'、"」のうち、上記の展開の結果でないものはすべて削除されます。

5.3.7. 終了ステータス

シェルは、終了ステータス 0 で終了したコマンドは正常終了したとみなします。0 以外の終了ステータスは失敗を意味します。

あるコマンドが、シグナル N で終了したときには、終了ステータスは 128+N になります。コマンドが見つからなかった場合の終了ステータスは 127 で、コマンドが見つかったけれど実行できなかった場合には、126 になります。

```
[darmadillo ~]$ echo hello
hello
[darmadillo ~]$ echo $?
0
[darmadillo ~]$ sleep 100
Ctrl-C
[darmadillo ~]$ echo $?
130
[darmadillo ~]$ noexistent-command
bash: noexistent-command: not found
[darmadillo ~]$ echo $?
127
[darmadillo ~]$ touch not-executable-file
[darmadillo ~]$ ./not-executable-file
bash: ./not-executable-file: Permission denied
[darmadillo ~]$ echo $?
126
```

図 5.14 終了ステータスの例

exit コマンドを使用すると、シェルの終了ステータスを指定することができます。

5.3.8. 入出力の扱い

5.3.8.1. 標準入出力

C 言語でプログラミングする際に、標準出力(stdout)、標準入力(stdin)、標準エラー出力(stderr)を使用したことがあると思います。printf 関数で出力される先が標準出力で、scanf 関数での入力元が標準入力です。

シェルを介してプログラムを実行する際、標準出力、標準入力、標準エラー出力は、通常、コンソールとなります。

シェルでは、以降で説明するリダイレクトとパイプという機能を用いて、標準入出力の入力元や出力先を、コンソールではなくファイルや他のプログラムにすることができます。

5.3.8.2. リダイレクト

シェルには、プログラムの入出力の入力元や出力先を変更する機能があります。その機能のことを、リダイレクトといいます。

標準出力をコンソールではなく、ファイルにする場合には以下のようにします。

```
[darmadillo ~]$ echo hello > log
```

図 5.15 出力のリダイレクト

このコマンドを実行すると、「log」という名前のファイルに「hello」と書き込まれます。fopen("log", "w")とした時と同様に、ファイルが存在しなければファイルが新たに作成され、また、ファイルが存在する場合はファイルの既存の内容はすべて上書きされます。

「log」に追記したい場合は、以下のように>>を使用します。

```
[darmadillo ~]$ echo hello >> log
```

図 5.16 出力のリダイレクト(追記)

リダイレクトする際には、ファイルディスクリプタを指定することもできます。標準出力のファイルディスクリプタは 1、標準入力は 2、標準エラー出力は 3 と決まっています。

そのため、標準エラー出力への出力だけ、ファイルに出力したい場合は、以下のように書けます。

```
[darmadillo ~]$ cat nonexistent_file 2> error
[darmadillo ~]$ cat error
cat: nonexistent_file: No such file or directory
```

図 5.17 標準エラー出力のリダイレクト

「nonexistent_file」(存在しないファイル)を **cat** コマンドで表示しようとした際のエラー出力を「error」という名前のファイルに保存しています。

ファイルディスクリプタを指定しない場合は、標準出力として扱われるので、以下の二つのコマンドは等価です。

```
[darmadillo ~]$ cat hello > log
[darmadillo ~]$ cat hello 1> log
```

図 5.18 標準出力のリダイレクト

また、リダイレクトは複数指定することもできます。

```
[darmadillo ~]$ somecommand > log 2> error
```

図 5.19 複数の出力のリダイレクト

上記の記述では、「*somecommand*」の標準出力への出力を「log」というファイルに書き込み、標準エラー出力への出力を「error」というファイルに書き込みます。

```
[darmadillo ~]$ somecommand > log 2>&1
```

図 5.20 標準出力と標準エラー出力を同じファイルにリダイレクト

このように記述すると、「*somecommand*」の標準出力と標準エラー出力への出力を同時に「log」というファイルに書き込む事ができます。

コマンドの途中経過をコンソールに表示する必要がない場合もあります。そのような場合は、/dev/null へ書き込むことで出力を捨てることができます。

```
[darmadillo ~]$ somecommand > /dev/null 2>&1
```

図 5.21 出力を/dev/null にリダイレクト

出力のリダイレクトと同様に、入力もリダイレクトすることができます。

```
[darmadillo ~]$ cat < /etc/hostname
debian
```

図 5.22 入力のリダイレクト

5.3.8.3. パイプ

リダイレクト機能を使うと、入出力をファイルと結びつけることができました。パイプ機能を使うと、あるプログラムの出力と別のプログラムの入力を結びつけることができます。

以下の例では、**echo** プログラムの標準出力を **cat** プログラムの標準入力に結びつけています。

```
[darmadillo ~]$ echo hello | cat
hello
```

図 5.23 パイプ

以下の例では、カーネルコンフィギュレーションの中に、「ARMADILLO」と書かれた行が何行あるか表示しています。

```
[darmadillo ~]$ zcat /proc/config.gz | grep ARMADILLO | wc -l
21
```

図 5.24 パイプの例

まず、**zcat** コマンドが/proc/config.gz(カーネルコンフィギュレーションを gzip 圧縮したファイル)を展開して標準出力に出力します。その出力を、**grep** コマンドの標準入力にパイプで繋いでいます。**grep** コマンドは、標準入力からの入力のうち、「ARMADILLO」という文字列を含む行だけを標準出力に出力します。**wc** コマンドは、-l オプションが指定されたとき、標準入力から入力された行数を表示します。

このように、パイプを使うことで、シンプルな機能を持ったプログラムを組み合わせて、複雑な処理を簡単に記述することができます。

5.3.8.4. ヒアドキュメント

ヒアドキュメントを使うと、複数行に渡る長い文をコマンドの標準入力にすることができます。

```
<<[-]word
        here-document
delimiter
```

図 5.25 ヒアドキュメントの文法

「<<」に続き、「word」を指定した次の行からヒアドキュメントとして、扱われます。ヒアドキュメントは、「word」のクォートを取り除いた「delimiter」が単独で現れる行まで続きます。「word」には、パラメータ展開、コマンド置換、算術展開、パス名展開は行われません。

```
[darmadillo ~]$ cat <<EOF
> hello
> world
> EOF
hello
world
```

図 5.26 ヒアドキュメントの例

「word」がクォートされていない場合、ヒアドキュメントはパラメータ展開、コマンド置換、算術式展開されます。「word」が一部でもクォートされている場合、ヒアドキュメントの展開は行われません。

```
[darmadillo ~]$ var="hello world"
[darmadillo ~]$ cat <<EOF
> $var
> EOF
hello world
[darmadillo ~]$ cat <<'EOF'
> $var
> EOF
$var
```

図 5.27 ヒアドキュメントの例(クォート)

「<<」と「word」の間に「-」を指定した場合、ヒアドキュメントの各行の先頭のタブが取り除かれます。この機能により、シェルスクリプト中にヒアドキュメントを記述する際に、自然なインデントで記述することができます。

5.3.9. 様々なコマンドの実行方法

ここで、コマンドの実行方法を再度確認しておきます。

5.3.9.1. 単純なコマンド

「5.3.1. 基本的なコマンドの実行方法」では、コマンド名の後に引数を指定してコマンドを実行すると紹介しました。

コマンドの構文を、「図 5.28. 単純なコマンドの文法」に示します。

コマンド名の前には、複数の変数の代入を書くことができます。

最初の単語が、コマンド名になります。コマンド名だけは必須です。

コマンド名の後には、複数の単語を引数として書くことができます。

引数の後には、リダイレクトに関する記述を書くことができます。

最後に制御演算子を書くことができます。制御演算子は「||、&、&&、;、;;、(、)、|、<newline>」のいずれかの文字列です。

```
[変数の代入...] <単語> [単語...] [リダイレクション] [制御演算子]
```

図 5.28 単純なコマンドの文法

5.3.9.2. パイプライン

パイプラインは、「|」で区切った一つ以上の単純なコマンドの列です。

```
[!] <command1> [| command2...]
```

図 5.29 パイプラインの文法

「5.3.8.3. パイプ」で説明したように、パイプで接続すると、「command1」の標準出力は「command2」の標準入力に接続されます。この接続は、リダイレクトよりも先に実行されます。

パイプラインの終了ステータスは、最後のコマンドの終了ステータスになります。パイプラインの前に予約後である!がある場合、そのパイプラインの終了ステータスは、最後のコマンドの終了ステータスの論理否定をとった値になります。また、終了ステータスを返す前に、シェルはパイプライン中のすべてのコマンドの終了を待ちます。

パイプライン中の各コマンドは、サブシェル内で、それぞれ別のプロセスとして実行されます。

5.3.9.3. リスト

リストは、1 つ以上のパイプラインを演算子「;、&、&&、||」のいずれかで区切って並べ、最後に「;、&、<newline>」のいずれかを置いたものです。

```
<pipeline1> [演算子 pipeline2...] [; or & or <newline>]
```

図 5.30 リストの文法

リストコマンドが制御演算子&で終わっている場合、シェルはコマンドをサブシェル内でバックグラウンド実行します。シェルはコマンドが終了するのを待たずに、返却ステータス 0 を返します。

コマンドを;で区切った場合には、これらは順番に実行されます。シェルはそれぞれのコマンドが終了するのを順番に待ちます。返却ステータスは、最後に実行したコマンドの終了ステータスになります。

制御演算子&&は AND リストを示し、「||」は OR リストを示します。AND リストの場合は「pipeline1」が終了ステータス 0 を返した場合に限り「pipeline2」が実行されます。OR リストの場合は、「pipeline1」が 0 以外の終了ステータスを返した場合に限り「pipeline2」が実行されます。AND リストと OR リストの返却ステータスは、リスト中で最後に実行されたコマンドの終了ステータスです。

5.3.9.4. 複合コマンド

```
(list)
$(list)
```

図 5.31 サブシェルの文法

()で括られた「list」はサブシェル内で実行されます。返却ステータスは「list」の終了ステータスです。

$()で括られた場合は、コマンド置換が行われます。

```
{ list; }
```

図 5.32 グループコマンドの文法

「list」が単に現在のシェル環境で実行されます。「list」の最後は改行文字かセミコロンでなければなりません。これはグループコマンドと呼ばれます。返却ステータスは「list」の終了ステータスです。

```
((expression))
$((expression))
```

図 5.33 算術評価の文法

「expression」が算術式評価の規則に従って評価されます。式の値が 0 でない場合、返却ステータスは 0 になります。そうでない場合、返されるステータスは 1 になります。

$(())で括られた場合は、算術式展開が行われます。

```
[[ expression ]]
```

図 5.34 条件式評価の文法

条件式「expression」の評価値に従って 0 または 1 を返します。単語分割とパス名展開はの間の単語に対しては行われません。チルダ展開、パラメータと変数の展開、算術式展開、コマンド置換、プロセス置換、クォート除去は行われます。

==演算子と!=演算子が使われたとき、演算子の右の文字列はパターンと解釈され、パターンマッチング規則に従ってマッチングが行われます。文字列がパターンにマッチすれば返り値は 0 であり、マッチしなければ返り値は 1 になります。パターンの任意の部分をクォートして、文字列としてマッチさせることもできます。

5.3.9.5. バックグラウンド実行

リストの最後に「&」をつけて実行すると、そのリストはバックグラウンドプロセスとして実行されます。対して、「&」を付けずに実行したプロセスはフォアグラウンドプロセスといいます。

バックグラウンドプロセスは、コンソールからの入力を受け付けられません。

フォアグラウンドプロセスの実行中に、サスペンド文字(通常は Ctrl+Z)を入力すると、そのプロセスは停止させられ、シェルに制御が戻ります。

この時、bg と入力するとバックグラウンドプロセスとして、プロセスを再開します。また、fg と入力するとフォアグラウンドプロセスとして実行を再開します。

フォアグラウンドプロセスは、最大一つだけしか実行できませんが、停止中のプロセスやバックグラウンドプロセスは複数存在することができます。それらには、サスペンド文字の入力によって停止したときや、「&」を付けて実行されたときにジョブ番号が割り振られます。bg や fg は、ジョブ番号を指定することができます。ジョブ番号は、「%」の後に続く数値で指定できます。

5.3.10. 制御構文

if、for などの構造化プログラミングを行うために必要な制御構文について説明します。

5.3.10.1. if 文

if 文の書式を以下に示します。

```
if list1; then list2; [elif list3; then list4; ] ... [ else list5; ] fi
```

図 5.35 if 文の構文

最初に、「if list1」が実行されます。「list1」の終了ステータスが 0 ならば、「then list2」が実行されます。「list1」の終了ステータスが 0 以外で、「elif list3」があれば、「list3」が実行されます。「list3」の終了ステータスが 0 ならば「list4」が実行され、「list3」の終了ステータスが 0 以外でさらに elif があれば、そのリストが実行されます。if、elif のリストがすべて 0 以外のステータスで終了し、「else list5」が指定されていた場合、「list5」が実行されます。

if 文の終了ステータスは、最後に実行されたコマンドの終了ステータスとなります。真と評価された条件が一つもなかった場合は、0 となります。

シェルでの真偽

シェルでは、0 を真として扱います。

C 言語での真偽とは逆なので混同しないようにしてください。

if 文は、多くの場合条件式と共に用いられます。条件式は、「[[」または test、「[」コマンドで使用でき、ファイルの属性を調べたり、文字列比較、算術式比較を行うことができます。

条件式には、「表 5.3. 文字列比較の条件式」、「表 5.4. 算術比較の条件式」、「表 5.5. ファイル比較の条件式」に示すものがあります。

表 5.3 文字列比較の条件式

条件式	結果
string	文字列が null(空文字、長さ 0)でなければ真
-n *string*	文字列が null(空文字、長さ 0)でなければ真
-z *string*	文字列が null(空文字、長さ 0)であれば真
string1 = *string2*	2 つの文字列が等しければ真
string1 != *string2*	2 つの文字列が等しくなければ真

表 5.4 算術比較の条件式

条件式	結果
arg1 -eq *arg2*	2 つの式が等しければ真
arg1 -ne *arg2*	2 つの式が等しくなければ真
arg1 -gt *arg2*	*arg1* が *arg2* より大きければ真
arg1 -ge *arg2*	*arg1* が *arg2* と等しい、又はより大きければ真
arg1 -lt *arg2*	*arg1* が *arg2* より小さければ真
arg1 -le *arg2*	*arg1* が *arg2* と等しい、又はより小さければ真
! *arg1*	*arg1* が偽ならば真、*arg1* が真ならば偽

表 5.5 ファイル比較の条件式

条件式	結果
-a *file*	*file* が存在すれば真
-d *file*	*file* が存在し、ディレクトリならば真
-f *file*	*file* が存在し、通常ファイルならば真
-c *file*	*file* が存在し、キャラクタデバイスファイルならば真
-b *file*	*file* が存在し、ブロックデバイスファイルならば真
-r *file*	*file* が存在し、読み込み可能ならば真
-w *file*	*file* が存在し、書き込み可能ならば真
-x *file*	*file* が存在し、実行可能ならば真
-s *file*	*file* が存在し、サイズが 0 より大きければ真

```
#!/bin/sh

if [ ! -e $1 ]; then
        echo "$1: No such file or directory."
        exit 0
fi

if [ -d $1 ]; then
        echo "$1 is directory."
elif [ -f $1 ]; then
        echo "$1 is regular file."
elif [ -c $1 ]; then
        echo "$1 is character device file."
elif [ -b $1 ]; then
        echo "$1 is block device file."
else
```

```
        echo "$1 is unknown type file."
fi
```

図 5.36 if 文の例(if_sample.sh)

test コマンド

「[」は、引数の最後に「]」を取る、**test** コマンドの別名です。if 文などを自然にかけるように、このような名前になっています。

「[」はコマンド名なので、「[」の後ろと「]」の前には、空白が必要です。

5.3.10.2. case 文

C 言語での switch 文は、シェルでは case 文で記述することができます。

```
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

図 5.37 case 文の構文

case 文では、まず「word」を展開し、各「pattern」にマッチするか調べます。「word」にはチルダ展開、パラメータ展開、変数展開、算術式展開、コマンド展開、クォート除去が行われます。「pattern」は、「word」と同様に展開されたあと、評価されます。

いずれかの「pattern」にマッチすると、対応する「list」が実行されます。返却コードは、最後に実行された「list」の終了コードになります。マッチするパターンがなかった場合、返却コードは 0 となります。

case 文の使用例を「図 5.38. case 文の例(case_sample.sh)」に示します。第 1 引数に a または b を指定すると、「alfa」または「blabo」と表示します。c または C を指定すると、「charlie」と表示します。第 1 引数に、これら以外の文字列を指定する又は何も指定ない場合、「other string」と表示します。

```
#!/bin/sh

case $1 in
        a) echo alfa;;
        (b) echo bravo;;
        c|C) echo charlie;;
        *) echo "other string";;
esac
```

図 5.38 case 文の例(case_sample.sh)

5.3.10.3. for 文

for 文は、「図 5.39. for 文の構文」のように書くことができます。

```
for name [in word]; do list; done
```

図 5.39 for 文の構文

「in」の後ろに記述された「word」が展開され、各単語が順に変数「name」に代入されます。代入の都度、「list」が実行されます。「in word」が省略された場合、すべての位置パラメータに対して「list」が実行されます。

返却ステータスは最後に実行されたコマンドの終了ステータスになります。「word」の展開結果が空となった場合、「list」は一度も実行されず、返却ステータスは 0 となります。

「word」を固定の文字列の並びで書く例を、「図 5.40. for 文の例 1(for_sample1.sh)」に示します。

```
#!/bin/sh

for a in alfa bravo charlie; do
        echo $a
done
```

図 5.40 for 文の例 1(for_sample1.sh)

「word」はパス名展開されますので、「図 5.41. for 文の例 2(for_sample2.sh)」のように書くと、ルートディレクトリ直下のディレクトリ、ファイルをすべて表示します。

```
#!/bin/sh

for f in /*; do
        echo $f
done
```

図 5.41 for 文の例 2(for_sample2.sh)

コマンド展開も行われますので、「図 5.42. for 文の例 3(for_sample3.sh)」のように書いても等価です。

```
#!/bin/sh

for f in $(ls /); do
        echo $f
done
```

図 5.42 for 文の例 3(for_sample3.sh)

5.3.10.4. while 文

for 文は、「図 5.43. while 文の構文」のように書くことができます。

```
while list1; do list2; done
```

図 5.43 while 文の構文

while 文では「list1」が実行され、終了ステータスが 0 であれば、「list2」を実行します。これを、「list1」の終了ステータスが 0 である限り続けます。返却ステータスは、最後に実行された「list2」の終了ステータスになります。一度も実行されていないときは、0 です。

```
#!/bin/sh

while true; do
        date
        sleep 1
done
```

図 5.44 while 文の例(while_sample.sh)

5.3.11. 関数

シェルでは、C 言語と同様に関数を使うこともできます。関数の構文は、以下の通りです。

```
name() { list; }
```

図 5.45 関数の構文

関数を定義したあと、関数名を記述することで、その関数を実行することができます。

```
#!/bin/sh

foo() {
        echo foo
}

echo "call foo function"
foo
```

図 5.46 関数の例(function_sample1.sh)

```
[darmadillo ~]$ ./function_sample1.sh
call foo function
foo
call bar function
bar
```

図 5.47 function_sample1.sh の実行結果

関数は、通常のコマンドと同様に引数を取ることができます。関数内では、一時的に位置パラメータが関数の引数に置き換えられます。

```
#!/bin/sh

foo() {
        echo ”in function”
        echo $@
}

echo ”in global”
echo $@

foo A B C

echo ”in global”
echo $@
```

図 5.48 関数の例(function_sample2.sh)

```
[darmadillo ~]$ ./function_sample2.sh 1 2 3
in global
1 2 3
in function
A B C
in global
1 2 3
```

図 5.49 function_sample2.sh の実行結果

シェルでは、変数のスコープは標準でグローバルです。**local** コマンドを使うことで、関数内のみのスコープを持つローカル変数を作ることができます。

```
#!/bin/sh

A=”global”
B=”global”
C=”global”

foo() {
        echo ”in function”
        A=”function”
        local B=”function”

        echo $A
        echo $B
        echo $C
}

echo ”in global”
echo $A
echo $B
echo $C

foo

echo ”in global”
```

```
echo $A
echo $B
echo $C
```

図 5.50 関数の例(function_sample3.sh)

```
[darmadillo ~]$ ./function_sample3.sh
in global
global
global
global
in function
function
function
global
in global
function
global
global
```

図 5.51 function_sample3.sh の実行結果

return コマンドを使用することで、関数の戻り値を指定することができます。**return** コマンドがなかったり、**return** コマンドに戻り値を指定しなかった場合は、関数の中で最後に実行されたコマンドの終了ステータスが戻り値になります。

シェルの関数は、戻り値に文字列を指定することはできません。関数の結果を文字列で受け取りたい場合は、グローバル変数を使うか、関数内で標準出力へ文字列を出力しコマンド置換する方法があります。

```
#!/bin/sh

foo() {
        return 10
}

bar() {
        echo "string"
}

foo
echo "foo returns $?"
echo "bar returns $(bar)"
```

図 5.52 関数の例(function_sample4.sh)

```
[darmadillo ~]$ ./function_sample4.sh
foo returns 10
bar returns string
```

図 5.53 function_sample4.sh の実行結果

6. C 言語による実践プログラミング

この章では、C 言語を使用した実践的なプログラミングを取り上げます。

一口にプログラミングといっても、ちょっとしたファイルの読み書きやデバイス操作を実現するだけの簡単なものから、複雑な演算を行ったりネットワークを介してサービスを提供し続けるような高度なものまで、多岐に渡ります。ここではその中から、誰もが様々な場面で使うであろう基本的技術と、Armadillo が持つインターフェースを通じて行う操作の代表的なものを中心に、分野ごとに分けて紹介していきます。

Linux や開発環境に依存した独特な部分に留意しつつ、組み込みならではの使用方法を想定した応用例やノウハウについても多く記載したつもりです。プログラミング経験豊富な方であってもおさらいのつもりで読んでみて、一般的なプログラミング本では解説されていない情報を見つけていただければ幸いです。

6.1. C 言語プログラミングのためのツールたち

C 言語で書かれたプログラムは、実行できる状態にするためにコンパイルが必要です。このためのツールが C コンパイラでありツールチェーンですが、この他にも一連のビルド作業を手助けしてくれる色々なツールが存在します。C 言語プログラミングのための基礎知識として、これらビルド用ツールの機能や使い方について説明します。

6.1.1. C コンパイラ: gcc

第 1 部「開発の基本的な流れ」の「アプリケーションプログラムの作成」で説明したように、C 言語で記述したソースコードのコンパイルには gcc(GNU C Compiler[1])を使用します。

gcc はいくつかの動作形態を持っており、また多くの機能を備えています。これらは gcc に与えるコマンドラインオプションにより制御されますが、このオプションはかなりの多種に及びます。ここでは、gcc を使いこなすために必須といえるオプションをピックアップして紹介します。

6.1.1.1. 動作全体に関わるオプション

gcc にソースファイル名のみを与えると、コンパイル、アセンブル、リンクの一連の処理を自動で行って、実行ファイルを出力します。これが基本動作です。

`-o` *出力ファイル名*オプションを付けることで、出力ファイルの名前を指定することができます。このオプションを付けなかった場合、実行ファイルは `a.out` という名前になります。

`-c` オプションを付けると、コンパイルからアセンブルまでを行い、リンク処理を行いません。出力ファイルは、アセンブラが出力したオブジェクトファイルになります。

6.1.1.2. 警告オプション

`-W` で始まるものは、警告オプションです。コンパイル時の警告表示を制御することができます。

本書のサンプルプログラムでは、バグを生みやすいコードの書き方をしていれば警告表示が出るように、コンパイル時のオプションとして`-Wall` と`-Wextra` を必ず指定するようにしています。これらのオプ

[1]GCC と大文字で書いた場合は、通常 GNU Compiler Collection(C/C++、Objective-C/C++、Java、Fortran、Ada を扱える統合コンパイラパッケージ全体)を指します。

ションを付けても警告が出ないような書き方を目指すことで、C 言語の構文が原因であるバグの大半を防ぐことができます。

6.1.1.3. 最適化オプション

-O で始まるものは、最適化オプションです。コンパイラの最適化レベルを制御することができます。

-O0 は、最適化を行いません。最適化オプション未指定のときも同じ動作です。

-O1 では、コードサイズと実行時間を小さくするいくつかの最適化を行います。-O のように数字をつけなかったときも、この動作になります。

-O2 では、サポートするほとんどの種類の最適化を行います。ただし、コードサイズと実行速度のどちらかを大きく犠牲にするようなもの(例えば関数の自動インライン化)は、このレベルには含まれません。

-O3 では、さらに高速にするための最適化を行います。コードサイズは大きくなるが実行速度を稼ぐことのできる関数の自動インライン化は、このレベルで有効になります。

-Os では、コードサイズが小さくなるように最適化を行います。-O2 で有効になる最適化のうちコードサイズが大きくならないものすべてに加え、さらにコードサイズが小さくなるように設計された特別な最適化も行います。

6.1.1.4. 処理対象となるディレクトリやファイルを追加するオプション

-I *ディレクトリ名*オプションを付けると、ヘッダファイルの検索対象に指定ディレクトリが追加されます。ここで指定したディレクトリは、標準のシステムインクルードディレクトリよりも先に検索されます。

-iquote *ディレクトリ名*オプションを付けると、ローカルヘッダファイル(#include "*ヘッダファイル名*"という形で、ダブルクォート囲みでヘッダ指定したもの)の検索対象に指定ディレクトリが追加されます。システムヘッダファイル(#include <*ヘッダファイル名*>と指定したもの)については、この指定ディレクトリからは検索されません。

-L や-l は、リンク時のリンカ動作に影響を与えるオプションです。

-L *ディレクトリ名*オプションを付けると、ライブラリファイルの検索対象に指定ディレクトリが追加されます。

-l *ライブラリ名*オプションを付けると、lib *ライブラリ名*.so または lib *ライブラリ名*.a という名前のライブラリファイルを検索し、リンクします。

6.1.1.5. プロセッサ固有のオプション

PC には一般的に x86 と呼ばれる種類のプロセッサが搭載されていますが、Armadillo に搭載されているプロセッサは ARM コアを採用したものです。各々のプロセッサ固有の機能を制御するときは、-m で始まるオプションを使用します。

ARM プロセッサ固有のオプションには、アーキテクチャや浮動小数点演算ユニット、ABI の種類などを指定するものがあります。

-march=*アーキテクチャ名*を付けると、指定アーキテクチャ向けのインストラクションセットを用いたコンパイルが行われます。

ARM インストラクションセットの互換性

ARM のインストラクションセットは、後方互換性が維持されています[2]。このため異なるプロセッサを搭載したマシン間であっても、より古い方のアーキテクチャを指定してコンパイルすることで、バイナリレベルでの互換性を保つことができます。しかしながら、新しいアークテクチャ上で古いアーキテクチャ向けのインストラクションセットを使用することは、使用可能な新しいインストラクション(例えば、ARMv3 まではハーフワード単位の入出力インストラクションが使えません)を使用しないことになりますから、実行効率面から見ればもったいない状態になります。

この辺りのバランスを考えて、Debian GNU/Linux の ARM バイナリは ARMv4(`-march=armv4`)向けとしてコンパイルされたものになっており、ARMv5TEJ(`-march=armv5te`)コアを持つ Armadillo-400 シリーズ上でそのまま動作します。一方、Ubuntu の ARM バイナリは、より実行効率を優先したものになっているようです。Ubuntu 10.04 では ARMv7(`-march=armv7-a`)向けとされており、Armadillo-400 シリーズ上では動作しないバイナリになっています。

6.1.2. クロスツールチェーン

ツールチェーンには、C コンパイラ(gcc)を始めとして、C プリプロセッサ(cpp)、アセンブラ(as)、リンカ(ld)、アーカイバ(ar)、デバッガ(gdb) などが含まれます。ARM 向けにクロス開発する際は、クロスツールチェーンを使用します。

6.1.2.1. プレフィックス

第 1 部「開発基本的な流れ」の「アプリケーションプログラムの作成」で説明したように、作業用 PC 上で ARM 向けにクロスコンパイルする際には **arm-linux-gnueabi-gcc** を使用します。このコマンド名の前に付いている arm-linux-gnueabi-の部分を、プレフィックス(前頭詞)といいます。

クロスツールチェーンは、すべてこのプレフィックスが付いたコマンド名になっています。例えば ARM クロスアセンブラであれば **arm-linux-gnueabi-as**、ARM クロスリンカであれば **arm-linux-gnueabi-ld** になります。

6.1.3. make と makefile

第 1 部「開発の基本的な流れ」の「make」で紹介したように、C 言語で開発する際にはコンパイル、アセンブル、リンクといった一連のビルド作業を自動化するために、makefile を記述して make を使用することが一般的です。

ここでは make の使い方と、makefile の書き方について紹介します。

6.1.3.1. make

make は、プログラムのビルドを簡単にするツールです。makefile にプログラムのビルド手順ルールを記述しておくと、make はそのルールに従って次に行うべき手順を自動的に見つけ出し、必要なコマンドだけを実行してくれます。

[2]新しいアーキテクチャでは、以前のアーキテクチャに存在したインストラクションがすべて使用可能です。

何のオプションも付けずに make を実行すると、カレントディレクトリにある GNUmakefile、makefile、Makefile といった順にファイルを検索し、最初に見つかったものをルールとして使用します[3]。

-C *ディレクトリ名*オプションを使用して指定したディレクトリに移動した状態で実行したり、-f *ファイル名*オプションを使用して指定したファイル名を makefile として読み込むことなども可能です。

make の詳細情報

gcc と同様に、make のオプションや makefile の書き方などの詳細情報については、info ページが充実しています。

make の info ページは make-doc パッケージに含まれており、ATDE などの Debian 環境では **aptitude** や **apt-get** コマンドでインストールすることができます。

6.1.3.2. makefile へのルールの記述

make は、makefile に記載されたルールに従ってビルドを行います。このルールの書き方について説明します。

makefile には、複数のルールを記述することができます。1 つのルールは必ず 1 つのターゲットを持ち、このターゲットがそのルールで生成されるファイルとなります。ターゲットと組み合わせて、そのターゲットを生成するための依存ファイル(事前に必要なファイル)の名称と、実行するコマンドラインを記述します。

```
ターゲット 1: 依存ファイル 1
        コマンドライン 1

ターゲット 2: 依存ファイル 2 依存ファイル 3
        コマンドライン 2
        コマンドライン 3
```

図 6.1 ルールの記述方法

依存ファイルは、ターゲット名:の後にスペース区切りで複数記述することができます。コマンドラインは、次の行の先頭からタブ(スペースではありません)を入力した後に記述します。コマンドラインを複数行書くことも可能です。

複数のルールが記述された makefile に対し make を実行すると、一番上に記述されているターゲットに対するルールのみが適用されます。このターゲット(「図 6.1. ルールの記述方法」でいえばターゲット1)を、デフォルトゴールと呼びます。

デフォルトゴール以外のターゲットを指定して make したい場合、make にターゲット名を与えて実行します。「図 6.1. ルールの記述方法」でターゲット 2 を make する場合は、**make ターゲット 2** になります。

```
ターゲット 1: ターゲット 2
        コマンドライン 1
```

[3]一般的には、大文字始まりの Makefile がよく使用されるようです。

```
ターゲット 2:
        コマンドライン 2
```

図 6.2 ルールの記述方法 2

「図 6.2. ルールの記述方法 2」のように、別のルールのターゲットを依存ファイルとして指定することが可能です。この場合、ターゲット 1 を生成するためにターゲット 2 が必要となるため、先にコマンドライン 2 が実行されます。

あるルールを適用する際に、必ずコマンドラインが実行されるわけではありません。make はルールを評価する際、必ずターゲットの存在と更新日時を確認します。ターゲットが存在しない場合、またはターゲットの更新日時より依存ファイルのいずれかの更新日時が新しくなっていた場合のみ、コマンドラインを実行します。

つまり、2 回目以降に make した際は、依存ファイルが更新されたルールのコマンドラインのみが実行されるわけです。このように、make はビルド時間を短縮してくれます。

より実際に近い、makefile の例を見てみます。

```
target1: target2
        cat target2

target2: depend1 depend2
        cat depend1 > target2
        cat depend2 >> target2
```

図 6.3 makefile の実例

「図 6.3. makefile の実例」では、デフォルトゴールは target1 です。target1 は target2 に依存するので、target2 がない場合は先に target2 を作成しにいきます。

target2 は、depend1 と depend2 に依存します。target2 という名前のファイルがないか、target2 が作られた後に depend1 や depend2 が変更されていた場合、下のコマンドライン 2 行が実行されます。

target2 が存在して target1 という名前のファイルがない場合、または target1 作成後に target2 が再作成されていた場合、cat target2 が実行されます。この例では target1 が作成されることはありませんので、make をするたびに毎回 cat target2 が実行されることになります。

「図 6.3. makefile の実例」を Makefile という名前でファイル保存し、適当な内容のファイル depend1 と depend2 を作成してから make すると、以下のように動作します。

```
[ATDE ~]$ ls ❶
Makefile  depend1  depend2
[ATDE ~]$ cat depend1 ❷
hello
[ATDE ~]$ cat depend2
world
[ATDE ~]$ make ❸
cat depend1 > target2 ❹
cat depend2 >> target2
cat target2 ❺
```

```
hello
world
[ATDE ~]$ ls ❻
Makefile  depend1  depend2  target2
[ATDE ~]$ make target2 ❼
make: `target2' は更新済みです
[ATDE ~]$ echo "byebye" > depend2 ❽
[ATDE ~]$ make target2 ❾
cat depend1 > target2
cat depend2 >> target2
[ATDE ~]$ make ❿
cat target2
hello
byebye
```

図 6.4 makefile の実例: 実行結果

❶ Makefile と依存ファイルを用意します。

❷ depend1 は hello、depend2 は world と書かれたテキストファイルです。

❸ make すると、target1 に対するルールが適用されます。

❹ target1 は target2 に依存するので、target2 に対するルールが適用されコマンドラインが実行されます。

❺ target2 が作成されると、target1 のためのコマンドラインが実行されます。

❻ target2 が作成されています。

❼ target2 を make しますが、既に存在する target2 が依存ファイルより新しいので、何も実行されません。

❽ target2 の依存ファイルを変更してみます。

❾ target2 を make すると、今度は依存ファイルが更新されているので、コマンドラインが実行されます。

❿ target1 が作成されることはないので、target1 に対するコマンドラインは毎回実行されます。

6.1.3.3. makefile での変数の使用

makefile 内では、変数[4]を使うことができます。

変数名には、前後がスペースでなく、「:」(コロン)、「#」(ナンバー記号[5])、「=」(イコール)を含まない文字列を使用できますが、通常は英数字と「_」(アンダースコア)のみで構成するのが無難です。なお、大文字と小文字は区別されます。

変数の定義は、**変数名 = 値**という形式で初期値を代入することによって成されます。シェルスクリプトの場合とは異なり、=の前後にはスペースを入れることができます。変数の値は文字列、または文字列のリストです。リストの場合は、**変数名 = 値 1 値 2 ...**と、スペースで値を区切って指定します。

変数を参照するには、**$(変数名)**または**${変数名}**とします。変数を参照すると展開され、展開された文字列と置き換えられます。

[4]マクロとも呼ばれます。
[5]日本では通常、シャープと呼ばれる記号。

変数名 = 値という形式で定義された変数は、正確には再帰展開変数(recursivery expanded variable)といいます。再帰展開変数は記述されたままの形で値を保持し、参照するまで展開されません。そのため、

```
FOO = foo $(BAR)
BAR = bar baz
```

と定義することができます。このとき、**$(FOO)**を展開すると「foo bar baz」となります。ただし、

```
FOO = foo
FOO = $(FOO) bar baz
```

とすると無限ループになるため、定義することはできません。

変数は、**変数名 := 値**という形式でも定義することができます。この形式で定義された変数を、単純展開変数(simply expanded variable)といいます。単純展開変数は、定義された時点で値を展開して保持します。そのため以下のように記述することで、変数に値を追加することができます。

```
FOO := foo
FOO := $(FOO) bar baz
```

また、**変数名 += 値**とすることでも変数に文字列を追加できます。

```
FOO = foo bar
FOO += baz
```

とした場合、**$(FOO)**を展開すると「foo bar baz」となります。**+=**で文字列を追加した場合、追加する文字列の前に半角スペースが一つ追加されます。展開がいつ行われるかは、文字列を追加する変数がどのように定義されたかに準じます。再帰展開変数に**+=**で文字列を追加した場合、参照の際に展開されます。また、単純展開変数に文字列を追加した場合は、代入の際に展開されます。

変数定義は、以下のように書くこともできます。

```
FOO ?= bar
```

こうすると**変数 FOO** が定義されていない場合だけ、**変数 FOO** に「bar」を代入します。

6.1.3.4. makefile で使用される暗黙のルールと定義済み変数

makefile では、よく使われるルールは明示的に記述しなくても暗黙のルールとして適用されます。

C/C++言語のソースファイルをビルドする際に適用される暗黙のルールには、以下のものがあります。

1. C プログラムのコンパイル

   オブジェクトファイル(.o)が、C ソースファイル(.c)から**$(CC) -c $(CPPFLAGS) $(CFLAGS)** ***C ソースファイル名***というコマンドラインで生成されます。

2. C++プログラムのコンパイル

オブジェクトファイル(.o)が、C++ソースファイル(.cc/.cpp/.C のいずれか)から**$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)** ***C++ソースファイル名***というコマンドラインで生成されます。

3. アセンブラソースのプリプロセス

   プリプロセス済みアセンブラソースファイル(.s)が、アセンブラソース(.S)から**$(CPP) $(CPPCFLAGS)** ***アセンブラソース名***というコマンドラインで生成されます。

4. プリプロセス済みアセンブラソースのアセンブル

   オブジェクトファイル(.o)が、プリプロセス済みアセンブラソースファイル(.s)から**$(AS) $(ASFLAGS)** ***プリプロセス済みアセンブラソースファイル名***というコマンドラインで生成されます。

5. リンク

   実行ファイル(拡張子なし)が、オブジェクトファイル(.o)から**$(CC) $(LDFLAGS)** ***オブジェクトファイル名*** **$(LOADLIBES) -o** ***実行ファイル名***というコマンドラインで生成されます。

   このルールは、複数のオブジェクトファイルに対して適用することもできます。ルールに、実行ファイル名:オブジェクト 1 ファイル名 オブジェクト 2 ファイル名とだけ記述しておくと、オブジェクト 1 ファイルとオブジェクト 2 ファイルをルールに従って生成した後、2 つのオブジェクトファイルから実行ファイルを生成します。

ここに登場した CC や CFLAGS といった変数は、暗黙のうちに定義されている変数です。主なものを挙げます。

表 6.1 暗黙のルールで使用される変数

変数名	デフォルト値	説明
AR	ar	アーカイバ
AS	as	アセンブラ
CC	cc	C コンパイラ。Linux システムでは、cc は **gcc** コマンドへのリンクになっています。
CXX	g++	C++コンパイラ
CPP	$(CC) -E	C プリプロセッサ
RM	rm -f	ファイル削除コマンド
ARFLAGS	rv	アーカイバに渡されるフラグ
ASFLAGS		アセンブラに渡される拡張フラグ
CFLAGS		C コンパイラに渡される拡張フラグ
CXXFLAGS		C++コンパイラに渡される拡張フラグ
CPPFLAGS		C プリプロセッサとそれを使うプログラムに渡される拡張フラグ
LDFLAGS		コンパイラがリンカ(ld)を呼び出すときに渡される拡張フラグ

パターンルールを使用して、新しい暗黙のルールを定義することもできます。パターンルールは、ターゲットと依存ファイルの一部に%を用いて記述します。%は、空でない任意の文字列に適合します。

例えば、C プログラムのコンパイルを行う暗黙のルールとして、あえてデフォルト状態と同じものをパターンルールで記述すると、このようになります。

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

ここで、$<や$@は自動変数と呼ばれる特殊な変数です。自動変数はルールが実行されるたびに、ターゲットと依存ファイルに基づいて設定される変数です。この例では$@はターゲットとなるオブジェクトファイル名に、$<は依存ソースファイル名に展開されます。

自動変数には、以下のようなものがあります。

表 6.2 自動変数

自動変数	説明
$@	ターゲットファイル名
$<	最初の依存ファイル名
$?	ターゲットより新しいすべての依存ファイル名
$^	すべての依存ファイル名

6.1.3.5. 変数の外部定義とオーバーライド

変数は、makefile の外で定義することもできます。定義方法は 2 種類あります。

1 つ目の方法は、**make** コマンドの引数として指定する方法です。**make 変数名=値**とすることで、変数が定義されます。なお、makefile 内に同じ名前の変数定義があった場合でも、こちらの引数による定義の方が優先(オーバーライド)されます。

2 つ目の方法は、環境変数として指定する方法です。すべての環境変数は、make の変数と同等に扱われます。しかし、こちらの変数定義はそれほど強いものではありません。make 引数による定義や、makefile 内の定義があった場合、そちらが優先(オーバーライド)されます。

ちなみに、makefile 内で同じ変数を重複定義した場合、最後に定義されたものが優先(オーバーライド)されます。

オーバーライドが発生する例を見てみます。

```
[ATDE ~]$ cat Makefile ❶
VARIABLE = value

all:
        echo $(VARIABLE)
        echo $(SHELL)
[ATDE ~]$ make
echo value
value
echo /bin/sh
/bin/sh
[ATDE ~]$ make VARIABLE=arg ❷
echo arg
arg
echo /bin/sh
/bin/sh
[ATDE ~]$ VARIABLE=env make ❸
echo value
value
echo /bin/sh
/bin/sh
```

図 6.5 オーバーライドの発生例

❶ makefile 内で定義した変数 VARIABLE と環境変数 SHELL を表示するだけの Makefile。

❷ **make** コマンドへの引数で定義した変数が、makefile 内で定義した変数よりも優先されます。

❸ makefile 内で定義した変数が、環境変数よりも優先されます。

6.1.3.6. 条件文

makefile 内には、ある条件が成立したときだけ有効になる行を書くことができます。基本的な構文は以下のようになります。

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
endif
```

CONDITIONAL-DIRECTIVE の条件が真の時に、**TEXT-IF-TRUE** の行が有効になります。

また、else 節を使って以下のように書くこともできます。

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
else
TEXT-IF-FALSE
endif
```

または

```
CONDITIONAL-DIRECTIVE
TEXT-IF-ONE-IS-TRUE
else CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
else
TEXT-IF-FALSE
endif
```

CONDITIONAL-DIRECTIVE は、**ifeq**、**ifneq**、**ifdef**、**ifndef** のいずれかの構文を使って記述します。

ifeq を使う場合、**CONDITIONAL-DIRECTIVE** は以下のようになります。

```
ifeq (ARG1, ARG2)
ifeq 'ARG1', 'ARG2'
ifeq "ARG1", "ARG2"
```

どの書き方をしても意味は同じです。**ARG1** と **ARG2** を展開し両者が等しい場合、真と判定され **TEXT-IF-TRUE** が有効になります。

ifneq も **ifeq** と同様に記述することができますが、**ARG1** と **ARG2** を展開し両者が等しくない場合、真と判定され **TEXT-IF-TRUE** が有効になります。

```
ifneq (ARG1, ARG2)
ifneq 'ARG1', 'ARG2'
ifneq "ARG1", "ARG2"
```

ifdef は、指定された変数名の変数が定義済みの場合、真と判定されます。**ifndef** はその逆です。

```
ifdef VARIABLE-NAME
ifndef VARIABLE-NAME
```

VARIABLE-NAME に変数が指定された場合、変数を展開した後の文字列を変数名として使用します。

```
BAR = true
FOO = BAR
ifdef $(FOO)
BAZ = yes
endif
```

とした場合、**変数 FOO** は「BAR」に展開され、それが変数名として用いられます。**変数 BAR** は定義されているので、**ifdef $(FOO)**は真として判定され、**BAZ = yes** 行が有効になります。

6.1.3.7. make 動作の実際

第 1 部「開発の基本的な流れ」の「make」で使用した makefile がどのように動作しているのか、改めて見てみます。

「図 6.6. 基本的な makefile」では、C ソースファイル hello.c から実行ファイル hello が生成されます。

```
CROSS   := arm-linux-gnueabi ❶

ifneq ($(CROSS),) ❷
CROSS_PREFIX    := $(CROSS)-
endif

CC = $(CROSS_PREFIX)gcc ❸
CFLAGS = -Wall -Wextra -O2
LDFLAGS =

TARGET = hello ❹

all: $(TARGET) ❺

hello: hello.o ❻
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@ ❼

clean: ❽
        $(RM) *~ *.o $(TARGET) ❾
```

```
%.o: %.c ❿
        $(CC) $(CFLAGS) -c -o $@ $<
```

図 6.6 基本的な makefile

❶ デフォルトで CROSS を arm-linux-gnueabi として定義し、クロスコンパイルを行います。

❷ CROSS 変数が空でなければ、CROSS_PREFIX を定義します。

❸ 暗黙のルールで使用される変数 CC と CFLAGS、LDFLAGS を明示的に定義し、オーバーライドしています。これによって、gcc の前に CROSS_PREFIX が付きます。

❹ ファイル名が変わっても使いまわせるように、実行ファイルの名前を変数で定義します。

❺ デフォルトゴール(一般的に all という名前を付けます)は、TARGET に依存します。

❻ hello は、hello.o に依存します。

❼ $@、$^は自動変数、CC、LDFLAGS、LDLIBS は暗黙のルールで使用される変数です。

❽ clean ターゲットは、依存ファイルがないので必ず実行されます。

❾ 生成したファイルや中間ファイルをすべて削除します。

❿ C プログラムのコンパイルを行うパターンルールを定義しています。

「図 6.6. 基本的な makefile」を Makefile という名前で保存し、C ソースコードを hello.c として同じディレクトリに置いてから make すると、ARM (Armadillo)用の実行ファイルが生成されます。

```
[ATDE ~]$ make CROSS_COMPILE=arm-linux-gnueabi-
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -c -o hello.o hello.c
arm-linux-gnueabi-gcc  hello.o  -o hello
[ATDE ~]$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for
 GNU/Linux 2.6.14, not stripped
```

make clean で、生成されたすべてのファイルを削除できます。

```
[ATDE ~]$ ls
Makefile  hello  hello.c  hello.o
[ATDE ~]$ make clean
rm -f *~ *.o
[ATDE ~]$ ls
Makefile  hello.c
```

また、コマンドライン引数による変数定義を使って make CROSS=として変数 CROSS をオーバーライドすると、ホスト PC 用の実行ファイルを生成できます。この例のようにすることで、同じ makefile、同じソースファイルから異なるアーキテクチャ用の実行ファイルを簡単に生成できるわけです。

```
[ATDE ~]$ make CROSS=
gcc -Wall -Wextra -O2 -c -o hello.o hello.c
gcc  hello.o  -o hello
[ATDE ~]$ ./hello
Hello World
```

「図 6.6. 基本的な makefile」では、1 つのソースファイルから 1 つの実行ファイルを生成しています。これを、複数のソースから 1 つの実行ファイルを生成したり、複数の実行ファイルを生成するように変更してみます。

hello.c と world.c から実行ファイル hello が生成され、fiz.c と baz.c から実行ファイル fizbaz が生成されるようにする場合、このようになります。

```
CROSS   := arm-linux-gnueabi

ifneq ($(CROSS),)
CROSS_PREFIX    := $(CROSS)-
endif

CC = $(CROSS_PREFIX)gcc
CFLAGS = -Wall -Wextra -O2
LDFLAGS =

TARGET = hello fizbaz

all: $(TARGET)

hello: hello.o world.o
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

fizbaz: fiz.o baz.o
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

clean:
        $(RM) *~ *.o $(TARGET)

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<
```

図 6.7 複数ファイルを扱う makefile

6.2. C 言語プログラミングの復習

実践的なプログラミングの話題に入る前に、C 言語でプログラムを作成する際に気をつけるべきことについて復習しておきます。Linux システム独特の話題もありますので、他の OS 上での C 言語プログラミングに慣れた方も確認してみてください。

6.2.1. コマンドライン引数の扱いと終了ステータス

標準的な C プログラムでの main 関数は、以下のどちらかの形で定義しなければなりません。

```
int main(void);
int main(int argc, char *argv[]);
```

戻り値は int 型として規定されています。引数は取らないか、または argc,argv の 2 個を取ります[6]。

[6] これ以外に独自拡張的な仕様として、環境変数を使用するために int main(int argc, char *argv[], char *envp[]);と 3 個の引数を取る場合があります。

C ソースをコンパイルして生成したプログラムをシェルから実行すると、自身のコマンド名と渡されたコマンドラインパラメータが main 関数の引数として渡ります。パラメータをつけずコマンド名のみで実行した場合は argc は 1 で、argv はコマンド名の文字列へのポインタです。パラメータをつけると argc は(1+パラメータ数)となり、argv はコマンド名、パラメータ 1、パラメータ 2…といった形の文字列配列になります。

main 関数の戻り値は、コマンドの終了ステータスになります。シェルの世界では 0 が真、0 以外のすべての値を偽として扱いますので、プログラムが正常に終了した場合、main 関数の戻り値は 0 であるべきです。

終了ステータスを表現するためのマクロが、ヘッダファイル stdlib.h で定義されています。正常終了、つまり 0 となる値として EXIT_SUCCESS が、異常終了のための値として EXIT_FAILURE が用意されています。本書のサンプルプログラムでは、終了ステータスとしてこれらを使用しています。

その他の終了ステータス

stdlib.h で定義されている終了ステータス以外の終了ステータスとしては、BSD 由来のものが sysexits.h で定義されています。

プログラムの終了には、exit 関数を呼ぶ方法もあります。

```
void exit(int status);
```

この exit 関数に渡す status が終了ステータスであり、main 関数の戻り値と同様の扱いです[7]。

main 関数周りの動作を実際に見てみましょう。コマンドに渡したパラメータを順番に表示するプログラムです。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;

        for (i = 0; i < argc; i++)
                printf("%d: '%s'\n", i, argv[i]);

        return 0;
}
```

図 6.8 すべてのパラメータを表示するプログラム(show_arg.c)

[7]シェルで扱える終了ステータスは 1 バイト長なので、実際には 0xff でマスクされた値が返ることになります。

```
[ATDE ~]$ ./show_arg 1 "with space"
0: './show_arg'
1: '1'
2: 'with space'
```

図 6.9 show_arg の実行結果

argv[0]にはコマンド名が格納されています。argv[1]以降は、与えたパラメータが順に格納されます。シェルからスペースを含む文字列をパラメータとして渡したい場合は、ダブルクォートかシングルクォートで囲みます。

シェルから呼ぶように作られたコマンドの多くは、「-」(ハイフン)始まりなどのオプション指定に対応しています。このようなオプションの解析を、すべて自前で実装するのはかなり手間のかかることです。これを楽にしてくれるライブラリ関数が存在します。

getopt 関数は、「-」で始まるショートオプションの解析を助けてくれます。getopt_long 関数は、「-」始まりのショートオプションと「--」始まりのロングオプションの両方を扱うことができます。ここでは getopt_long を使ってみます。

```
[ATDE ~]$ ./greeting --name Alice
Hello, Alice!
[ATDE ~]$ ./greeting --name=Bob --time morning
Good morning, Bob!
[ATDE ~]$ ./greeting -n Charlie -t evening --german
Gute Nacht, Charlie!
[ATDE ~]$ ./greeting -nDave -g
Hallo, Dave!
```

図 6.10 greeting の動作

一見してわかるとおり、-n または--name で指定した名前に対して挨拶を表示するプログラムです。-t または--time で時刻を指定することができ、それによって挨拶文が変化します。-g または--german を指定すると、挨拶文がドイツ語になります。

作成したコマンドに指定できるオプションを形式的に表記すると、次のようになります。

```
greeting <-n|--name NAME> [-t|--time TIME] [-g|--german]
```

このプログラムのソースコードが、「図 6.11. greeting.c」です。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>

enum time {
        TIME_MORNING,
        TIME_DAYTIME,
        TIME_NIGHT,
        TIME_UNKNOWN,
};
```

```
enum lang {
        LANG_ENGLISH,
        LANG_GERMAN,
};

static void usage(const char *prg)
{
        printf("usage: %s <-n|--name NAME> [-t|--time TIME] [-g|--german]\n", prg);
}

int main(int argc, char *argv[])
{
        int c;
        char *name = NULL;
        enum time time = TIME_UNKNOWN;
        enum lang lang = LANG_ENGLISH;
        char *greeting;

        while (1) {
                int option_index = 0;
                static struct option long_options[] = {
                        /* name,        has_arg,           flag, val*/
                        {"name",        required_argument, NULL, 'n'},
                        {"time",        required_argument, NULL, 't'},
                        {"german",      no_argument,       NULL, 'g'},
                        {0,             0,                 0,    0},
                };

                c = getopt_long(argc, argv, "n:t:g",
                                long_options, &option_index);
                if (c == -1)
                        break;

                switch (c) {
                case 'n':
                        name = strdup(optarg);
                        if (name == NULL)
                                exit(EXIT_FAILURE);
                        break;
                case 't':
                        if (strcmp(optarg, "morning") == 0)
                                time = TIME_MORNING;
                        else if (strcmp(optarg, "daytime") == 0)
                                time = TIME_DAYTIME;
                        else if (strcmp(optarg, "evening") == 0)
                                time = TIME_NIGHT;
                        else
                                time = TIME_UNKNOWN;
                        break;
                case 'g':
                        lang = LANG_GERMAN;
                        break;
                default:
                        usage(argv[0]);
                        exit(EXIT_SUCCESS);
                }
        }
```

```
        if (name == NULL) {
                /* NAME が指定されなかった */
                usage(argv[0]);
                return EXIT_FAILURE;
        }

        switch (time) {
        case TIME_MORNING:
                greeting =
                        (lang == LANG_ENGLISH) ? "Good Morning" : "Guten Morgen";
                break;
        case TIME_DAYTIME:
                greeting =
                        (lang == LANG_ENGLISH) ? "Good Afternoon" : "Guten Tag";
                break;
        case TIME_NIGHT:
                greeting =
                        (lang == LANG_ENGLISH) ? "Good Night" : "Gute Nacht";
                break;
        default:
        case TIME_UNKNOWN:
                greeting =
                        (lang == LANG_ENGLISH) ? "Hello" : "Hallo";
                break;
        }

        printf("%s, %s!\n", greeting, name);

        free(name);

        return EXIT_SUCCESS;
}
```

図 6.11 greeting.c

6.2.2. 終了処理

C 言語を使用してプログラムを記述する際、プロセスを正常に終了する方法には、以下の 3 種類があります。

1. main 関数から戻る
2. exit 関数を呼ぶ
3. _exit 関数を呼ぶ

また、Linux システムで動作するプロセスは、正常に終了する方法以外に、シグナルを受けて終了する場合があります。

main 関数から戻るか、exit 関数によってプロセスが終了した場合、所定の終了処理が行われます。

まず、atexit 関数や on_exit 関数によって登録された関数が、それらが登録された順番とは逆順に呼ばれます。

次に、オープン中の標準入出力[8]ストリームがすべてクローズされます。ストリームがクローズされると、バッファされている出力データはすべてフラッシュされ、ファイルに書き出されます。

また、tmpfile 関数によって作成されたファイルは削除されます。

exit 関数は、これらの処理を行ったあと、_exit 関数を呼びます。

_exit 関数内では、そのプロセスがオープンしたディスクリプタがすべてクローズされます。

さらに、Linux システムの場合、exit や_exit 関数で行われる処理の他に、プロセス終了時にカーネルが資源の回収を行います。すなわち、プロセスがオープンしているすべてのディスクリプタをクローズし、使用していたメモリなどを開放します。

プロセスがシグナルを受信し、そのシグナルに対する動作がプロセスを終了させるものであった場合、プロセスは直ちに終了します。シグナルは、自プロセス以外のプロセスから送られることもありますし、プログラム中で abort 関数や kill 関数で自プロセスへシグナルを送ることもできます。

これらの終了処理やカーネルによる資源の回収処理があるため、アプリケーションプログラム内では malloc したメモリ領域は必ず free しなければいけないということはありません。終了処理で行われることを把握した上で、資源の後始末を明示的にプログラム中に記述せず、それらに任せるという方法もあります。

6.2.3. エラー処理

C 言語でプログラムを記述する際、エラー処理を怠りがちです。世にある C 言語プログラミングの参考書では、サンプルコードをシンプルに書くために、あえてエラー処理を書いていない場合が多いので、それらを参考にしてコードを記述すると、つい、エラー処理を忘れてしまいます。

しかし、実際に使用するプログラムでは、必ずエラー処理を行うコードを記述してください。特に組み込みシステムでは、PC やサーバーなどと比較して、振動や温湿度などの外部環境が厳しい環境で動作することが多いので、単純な処理でもエラーが発生することがあります。

システムコールまたはライブラリコールのエラーを検出する一般的な方法は、関数の戻り値をチェックすることです。システムコールといくつかのライブラリコールは、エラーが発生した場合 errno を設定します。errno の値を確認することによって、エラーの発生要因を知ることができます。

システムコールやライブラリコールの戻り値や、それらが設定する errno の値は、man ページで確認できます。

例えば、open システムコールの man ページの返り値のセクションには、「open()と creat()は新しいファイル・ディスクリプタを返す。エラーが発生した場合は-1 を返す(その場合は errno が適切に設定される)。」と記述されています。また、エラーのセクションには、open システムコールが返す可能性のある errono の値と、どのような時に設定されるのかが記述されています。

システムコールの man ページをみるコマンドは、**man 2 *関数名***です。ライブラリコールの場合は、**man 3 *関数名***となります。man ページの内容をよく確認し、エラーが発生した場合の処理を忘れずに記述してください。

エラー処理の例として、ファイルの内容を読み込み、標準出力に表示するプログラムのソースコードを「図 6.12. fdump.c」に示します。

このプログラムは、open、close、read、write の 4 つのシステムコールを使用します。エラーが発生した際には、perror 関数で errno に応じたエラーメッセージを表示します。

[8]man 3 stdio 参照

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        int fd;
        char buf[1024];
        ssize_t len;
        int ret;

        if (argc < 2) {
                printf("usage: %s <file name>\n", argv[0]);
                return EXIT_SUCCESS;
        }

        /*
         * open() は新しいファイル・ディスクリプタを返す。エラーが発生
         * した場合は -1 が返され、errno が適切に設定される。
         */
        fd = open(argv[1], O_RDONLY);
        /* エラーが発生した? */
        if (fd == -1) {
                /* errno に応じたエラーメッセージを出力する */
                perror("open");
                return EXIT_FAILURE;
        }

        ret = EXIT_SUCCESS;
        for(;;) {
                /*
                 * 成功した場合、読み込んだバイト数を返す（0 はファイル
                 * の終りを意味する)。エラーの場合は、-1 が返され、
                 * errno が適切に設定される。
                 */
                len = read(fd, buf, sizeof(buf));
                if (len <= 0) {
                        /* ファイルの終わりに達した? */
                        if (len == 0)
                                break;

                        /* エラーが発生した */
                        perror("read");
                        ret = EXIT_FAILURE;
                        break;
                }

                /*
                 * 成功した場合、書き込まれたバイト数が返される（ゼロは
                 * 何も書き込まれなかったことを示す)。エラーならば -1
                 * が返され、errno が適切に設定される。
                 */
                len = write(1, buf, len);
                /* エラーが発生した? */
                if (len == -1) {
                        perror("write");
```

```
                        ret = EXIT_FAILURE;
                        break;
                }
        }

        /*
         * close() は成功した場合は 0 を返す。エラーが発生した場合は
         * -1 を返して、 errno を適切に設定する。
         */
        if (close(fd) == -1) {
                perror("close");
                ret = EXIT_FAILURE;
        }

        return ret;
}
```

図 6.12 fdump.c

fdump の実行結果を以下に示します。引数にファイル名を指定して実行すると、ファイルの内容を表示します。正常に終了したときの終了ステータスは 0(EXIT_SUCCESS)になります。/var/log/message は読み込み権限のないファイルなので、これを引数に指定して fdump を実行すると、open システムコールで失敗します。異常終了時には、エラーメッセージを表示して終了します。その時の終了コードは 1(EXIT_FAILURE)になります。

```
[ATDE ~]$ ./fdump /etc/hostname
atde3
[ATDE ~]$ echo $?
0
[ATDE ~]$ ./fdump /var/log/messages
open: Permission denied
[ATDE ~]$ echo $?
1
[ATDE ~]$ ls -l /var/log/messages
-rw-r----- 1 root adm 211 2010-09-06 06:25 /var/log/messages
```

図 6.13 fdump の実行結果

6.2.4. 共通ヘッダファイル

次章からは、より実際に近いプログラムを目的別に取り上げていきます。

以降のプログラムでは、ソースコードを見やすくし、また何のエラーが起きたかを明確に可視化するために、エラー処理のための共通のヘッダファイル exitfail.h を使用することにします。

```
#ifndef EXITFAIL_H
#define EXITFAIL_H

#include <errno.h>

#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
```

```
#ifdef MAIN_C
static char *progname = NULL;

#define exitfail_init()                                                 \
        (progname = (strrchr(argv[0], '/') ? : (argv[0] - 1)) + 1)

/**
 * プログラムエラー終了関数
 * @param format 書式フォーマットと引数群
 */
void exitfail(const char *format, ...)
{
        va_list va;

        va_start(va, format);
        fprintf(stderr, "%s: ", progname);
        vfprintf(stderr, format, va);
        va_end(va);

        exit(EXIT_FAILURE);
}
#else
extern void exitfail(const char *__format, ...);
#endif /* MAIN_C */

#define exitfail_errno(msg) exitfail(msg " - %s\n", strerror(errno))

#endif /* EXITFAIL_H */
```

図 6.14 エラー内容表示と FAILURE 終了するためのヘッダ(exitfail.h)

main 関数を含んだソースでは、#define マクロ定義で MAIN_C を定義してこのヘッダをインクルードします。exitfail 関数は、printf 関数と同じ引数フォーマット形式で好きな内容のエラーメッセージを表示できます。exitfail_errno は、ライブラリ関数の呼び出しで errno が更新されるタイプのエラー発生直後に呼び出す専用のものです。引数として渡した文字列(通常は関数名を想定しています)と、errno を意味のある文章に変換したエラー文字列をセットで表示します。どちらの関数も、関数名どおりに exit して EXIT_FAILURE を返します。

6.3. ファイルの取り扱い

まずは、ファイルを扱う例です。一般的な C のテキストであっても最初の方で取り上げられるもので、基本的にはそれほど大きくは違いません。但し、PC のように高速だったり、ふんだんなメモリが積まれているわけではありませんから、無駄な処理をしないようにしたり、メモリリークを起こさないように一層の注意が必要といえます。

6.3.1. テキストファイルを扱う

テキストファイルを扱うサンプルプログラムを紹介します。ここでは Comma Separated Values(CSV) ファイルと呼ばれる、データをカンマで区切った形式のものを扱ってみます。日本郵便が公開している住所の郵便番号(ローマ字)(CSV 形式)[9]を処理する例としてみました。

以下のような仕様を満たすものとします。

[9]このデータは「郵便事業株式会社は著作権を主張しません。自由に配布していただいて結構です。」とされています。http://www.post.japanpost.jp/zipcode/dl/readme.html(2010 年 9 月現在の URL)

1. CSV ファイルを中身を整形して表示するアプリケーション
2. コマンド引数として、CSV ファイル(郵便番号データ)を指定する
3. 行頭にインデックス番号を表示し、その後スペース区切りで各項目を表示する
4. 表示データが流れてしまわないように、画面サイズを超える場合は一時停止する
5. 最後にデータ総数を表示する

郵便番号データ CSV ファイルの一行は、以下のように構成されています[10]。

```
01101,"0600035","KITA5-JOHIGASHI","CHUO-KU SAPPORO-SHI","HOKKAIDO",0,0,1,0,0,0
```

先頭から順に、以下の内容を表しています。

```
全国地方公共団体コード,"郵便番号","町域名","市区町村名","都道府県名",フラグ 1,フラグ 2,フラグ 3,フラグ 4,フラグ 5,フラグ 6
```

フラグにはそれぞれ意味があるのですが、今回のプログラムに関して処理する必要のあるものは 1 つだけです。フラグ 4 が 0 であって同じ郵便番号が複数行続く場合は町域名が長いため複数行に分割されたデータとなる、という仕様です。

```
01224,"0660005","KYOWA(88-2.271-10.343-2.404-1.427-","CHITOSE-SHI","HOKKAIDO",1,0,0,0,0,0
01224,"0660005","3.431-12.443-6.608-2.641-8.814.842-","CHITOSE-SHI","HOKKAIDO",1,0,0,0,0,0
01224,"0660005","5.1137-3.1392.1657.1752-BANCHI)","CHITOSE-SHI","HOKKAIDO",1,0,0,0,0,0
```

この時は、複数行を一つのデータとして扱う対応を行うことにします。

ここまでの仕様に基づいたコードは、次のようになりました。

```
#define _GNU_SOURCE /* strchrnul 関数使用のために必要 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

/* 表示する文字数 */
#define DISP_WIDTH  60 /* 幅 */
#define DISP_HEIGHT 17 /* 高さ */

/* CSV データ各要素のサイズ */
#define ZIPCODE_LEN 8
#define STREET_LEN  512
#define CITY_LEN    64
#define PREF_LEN    16
```

[10] 詳細なデータファイルの形式説明については、日本郵便のページを参照。http://www.post.japanpost.jp/zipcode/dl/readme.html(2010 年 9 月現在の URL)

```
#define FLAG_NUM    6
/* CSV データ 1 行の要素数 */
#define COLUMN_NUM  (5 + FLAG_NUM)

/* CSV データ構造体 */
typedef struct {
        long code;                  /* 全国地方公共団体コード */
        char zipcode[ZIPCODE_LEN]; /* 郵便番号 */
        char street[STREET_LEN];   /* 町域名 */
        char city[CITY_LEN];       /* 市区町村名 */
        char pref[PREF_LEN];       /* 都道府県名 */
        long flag[FLAG_NUM];       /* フラグ配列 */
} csvline_t;

#define BASENAME(p) ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/**
 * 行表示関数
 * @param pcsvline 表示する CSV データ構造体へのポインタ(NULL の場合は表示済データの総数を表示)
 */
static void printline(csvline_t *pcsvline)
{
        static int count = 0;   /* 表示したデータ総数 */
        static int line = 0;    /* 一画面中に表示した行数 */
        int disp_width = DISP_WIDTH, disp_height = DISP_HEIGHT, newline;
        char buf[1024];

        /* CSV データの各要素を表示 */
        if (pcsvline) {
                /* 郵便番号が入っていない場合は表示しない */
                if (!pcsvline->zipcode[0])
                        return;

                /* 各要素を表示フォーマットに展開 */
                snprintf(buf, sizeof(buf),
                         "%6d %05ld %s %s %s %s %ld %ld %ld %ld %ld %ld",
                         count++, pcsvline->code, pcsvline->zipcode,
                         pcsvline->street, pcsvline->city, pcsvline->pref,
                         pcsvline->flag[0], pcsvline->flag[1],
                         pcsvline->flag[2], pcsvline->flag[3],
                         pcsvline->flag[4], pcsvline->flag[5]);
        }
        /* CSV データの総数を表示 */
        else {
                /* データ総数を表示フォーマットに展開 */
                sprintf(buf, "Count: %6d", count);
        }

        /* 今回追加される表示行数を計算 */
        newline = (strlen(buf) + (disp_width - 1)) / disp_width;
        /* 1 画面を超える場合、入力があるまで一時停止 */
        if (line + newline >= disp_height) {
                getchar();
                /* 表示行数を初期化 */
                line = 0;
        }
        /* 実際に表示する */
        printf("%s\n", buf);
```

```
        /* 表示行数を更新 */
        line += newline;
}

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として読み込み CSV ファイル名を指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        FILE *pcsvfile; /* CSV ファイルポインタ */
        csvline_t csvline; /* CSV データ */
        char buf[256], *pbuf, *pcol[COLUMN_NUM];
        int i;

        exitfail_init();

        /* 引数が指定されなかった場合、usage 表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <csvfile>\n", BASENAME(argv[0]));
                return EXIT_SUCCESS;
        }

        /* CSV ファイルオープン */
        pcsvfile = fopen(argv[1], "r");
        if (!pcsvfile)
                exitfail_errno("fopen");

        /* CSV データを初期化 */
        memset(&csvline, 0, sizeof(csvline));
        /* CSV ファイルから 1 行読み込む */
        while (fgets(buf, sizeof(buf), pcsvfile)) {
                /* 各要素へのポインタ配列を初期化 */
                memset(pcol, 0, sizeof(pcol));
                /* 次のカンマ(または行末)を見つけてトークン化 */
                pbuf = strtok(buf, ",\n");
                for (i = 0; i < COLUMN_NUM && pbuf; i++) {
                        /* 前後のダブルクォートは除去する */
                        if (*pbuf == '"')
                                *strchrnul(++pbuf, '"') = '\0';
                        /* 要素へのポインタを保持 */
                        pcol[i] = pbuf;
                        /* 次のカンマ(または行末)を見つけてトークン化 */
                        pbuf = strtok(NULL, ",\n");
                }
                /* 要素数が不足している場合、次行にスキップ */
                if (i < COLUMN_NUM)
                        continue;

                /* 新しいデータの場合 */
                if (strcmp(pcol[1], csvline.zipcode) ||
                    strtol(pcol[8], NULL, 10)) {
                        /* 保持済みのデータを表示 */
                        printline(&csvline);

                        /* CSV データ各要素を保持 */
```

```
                        memset(&csvline, 0, sizeof(csvline));
                        csvline.code = strtol(pcol[0], NULL, 10);
                        strncpy(csvline.zipcode, pcol[1],
                                sizeof(csvline.zipcode) - 1);
                        strncpy(csvline.street, pcol[2],
                                sizeof(csvline.street) - 1);
                        strncpy(csvline.city, pcol[3],
                                sizeof(csvline.city) - 1);
                        strncpy(csvline.pref, pcol[4],
                                sizeof(csvline.pref) - 1);
                        for (i = 0; i < FLAG_NUM; i++)
                                csvline.flag[i] = strtol(pcol[5 + i], NULL, 10);
                }
                /* 既存データへの追加の場合 */
                else
                        /* 町域名の続きを追加 */
                        strncat(csvline.street, pcol[2],
                                (sizeof(csvline.street) -
                                 strlen(csvline.street)) - 1);
        }
        /* 保持済みのデータを表示 */
        printline(&csvline);

        /* CSV ファイルクローズ */
        fclose(pcsvfile);

        /* データ総数を表示 */
        printline(NULL);

        return EXIT_SUCCESS;
}
```

図 6.15 CSV ファイルの内容を表示するプログラム(dispcsv1.c)

main 関数から見ていきます。CSV ファイルの操作は基本的に標準 C ライブラリ関数で行っていますので、難しいところはないと思います。データは fgets 関数で一行ずつ読み込み、strtok 関数でカンマ区切りをトークン単位に分解。トークンがダブルクォートで始まっている場合は、これを外します。

ここのところで strchrnul という、標準 C ライブラリにない関数を使用しています。ダブルクォートを見つけるだけなら strchr 関数でよいのですが、この関数は検索文字が見つからなかったときに NULL を返します。これを考慮すると

```
        p = strchr(++pbuf, '"');
        if (p)
                *p  = '\0';
```

のように処理しなくてはなりません。

ここで **man strchr** とすると、似たような関数として以下のような説明が見つけられます。

```
SYNOPSIS
       #define _GNU_SOURCE
       #include <string.h>

       char *strchrnul(const char *s, int c);
```

```
DESCRIPTION
       The strchrnul() function is like strchr() except that if c is not found
       in s, then it returns a pointer to the null  byte  at  the  end  of  s,
       rather than NULL.
```

strchrnul 関数は、(strchr とは違い)未発見時に終端文字'\0'位置へのポインタを返します。このためサンプルプログラムのように条件分岐が不要になります。なお上記 man に説明されていますが、こうした GNU 拡張関数を使う場合はヘッダ(今回の場合 string.h)インクルード前に GNU 拡張関数を使うため GNU_SOURCE マクロを定義(#define)しておく必要があります。今回は小さな例ですが、このように man には便利な情報が多く記載されており大変有用です。

トークン解析処理が終わると、これを数値変換や文字列コピーして保持します。前述した複数行にわたる長い町域名に対応するため、複数回のループにまたがって一つのデータを処理することがあります。

こうして完成した一つのデータを表示しているのは、printline 関数です。この関数では、データを snprintf で読みやすい形の文字列に整形しています。大量の表示データが流れていってしまわないように、画面サイズ(マクロ定義で横 60 文字×縦 17 文字とされています)ごとに一時停止する処理を入れつつ、表示を行います。また、最後に行われるデータ総数表示にもこの関数を使用(引数 pcsvline に NULL を指定)しますので、このための分岐処理も入れてあります。

作成したアプリケーション dispcsv1 に CSV ファイル名を渡すと、データが整形表示されます。

```
[armadillo ~]$ ./dispcsv1 ken_all_rome.csv
     0 01101 0600000 IKANIKEISAIGANAIBAAI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 0 0 0 0
     1 01101 0640941 ASAHIGAOKA CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     2 01101 0600041 ODORIHIGASHI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     3 01101 0600042 ODORINISHI(1-19-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     4 01101 0640820 ODORINISHI(20-28-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     5 01101 0600031 KITA1-JOHIGASHI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     6 01101 0600001 KITA1-JONISHI(1-19-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     7 01101 0640821 KITA1-JONISHI(20-28-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
```

図 6.16 dispcsv1 の実行結果

リターンを入力すると次に進みます。途中で終了したいときは、Ctrl+C を入力してください。

6.3.2. 設定ファイルに対応する

アプリケーションの設定を保存するとき、.ini や.conf といった設定ファイルを用いることがあります。設定ファイルも普通はテキストファイルですので標準 C ライブラリ関数を駆使して作成することができますが、こうした目的のために特別な機能が用意されたライブラリを使うと、短いコードで効率的に扱うことが可能です。ここでは GLib というライブラリを使って conf ファイルを使用する例を紹介します。

以下のような機能を加えてみます。

1. dispcsv2.conf ファイルを参照して、動作の設定を可能にする
2. 表示一時停止判定用の画面サイズを設定できるようにする
3. 表示一時停止を行うかどうか設定できるようにする
4. データ総数の表示を行うかどうか設定できるようにする

5. 各データの条件を設定して一致/部分一致したもののみを表示できるようにする
6. 文字列の条件比較においては大文字小文字を同一視する
7. dispcsv2.conf ファイルが存在しない場合、初期状態が設定された conf ファイルを自動作成する

サンプルプログラムは、先ほど作ったものに手を加えたものです。機能実装のために追加したコードがほとんどで、大きく構造を変更はしていません。

```
#define _GNU_SOURCE /* strchrnul 関数使用のために必要 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <glib.h>

#define MAIN_C
#include "exitfail.h"

/* 表示する文字数 */
#define DISP_WIDTH  60 /* 幅 */
#define DISP_HEIGHT 17 /* 高さ */

/* CSV データ各要素のサイズ */
#define ZIPCODE_LEN 8
#define STREET_LEN  512
#define CITY_LEN    64
#define PREF_LEN    16
#define FLAG_NUM    6
/* CSV データ 1 行の要素数 */
#define COLUMN_NUM  (5 + FLAG_NUM)

/* CSV データ構造体 */
typedef struct {
        long code;                  /* 全国地方公共団体コード */
        char zipcode[ZIPCODE_LEN]; /* 郵便番号 */
        char street[STREET_LEN];   /* 町域名 */
        char city[CITY_LEN];       /* 市区町村名 */
        char pref[PREF_LEN];       /* 都道府県名 */
        long flag[FLAG_NUM];       /* フラグ配列 */
} csvline_t;

#define BASENAME(p) ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/* conf 設定ファイル内で使用するキーワードの定義 */
#define GROUP_DISPLAY "dispaly"
#define KEY_WIDTH     "Width"
#define KEY_HEIGHT    "Height"
#define GROUP_CONTROL "control"
#define KEY_PAUSE     "Pause"
#define KEY_COUNT     "Count"
#define GROUP_DATA    "data"
#define KEY_CODE      "Code"
#define KEY_ZIPCODE   "Zipcode"
#define KEY_STREET    "Street"
#define KEY_CITY      "City"
```

```
#define KEY_PREF      "Pref"
#define KEY_FLAG      "Flag"

/* conf設定保持用構造体 */
typedef struct {
        struct { /* 表示設定グループ */
                gint width;  /* 表示幅 */
                gint height; /* 表示高さ */
        } display;
        struct { /* 制御設定グループ */
                gboolean pause; /* 一時停止の有無 */
                gboolean count; /* 総数表示の有無 */
        } control;
        struct { /* データ条件設定グループ */
                gint code;      /* 全国地方公共団体コード */
                gchar *zipcode; /* 郵便番号(部分一致) */
                gchar *street;  /* 町域名(部分一致) */
                gchar *city;    /* 市区町村名(部分一致) */
                gchar *pref;    /* 都道府県名(部分一致) */
                gint *flag;     /* フラグ配列 */
                gsize flag_len; /* フラグ配列要素数 */
        } data;
} config_t;
/* conf 設定保持用領域ポインタ */
static config_t *pconf;

/**
 * conf ファイル読み込み関数
 * @param conffilename conf ファイル名
 */
static void readconf(char conffilename[])
{
        GKeyFile *keyfile;
        GError *error = NULL;
        gint flag[FLAG_NUM];
        gsize len;
        gchar *pdata;
        FILE *pconffile;
        int i;

        /* conf 設定保持用領域確保 */
        pconf = g_slice_new(config_t);

        /* キーファイル確保 */
        keyfile = g_key_file_new();

        /* conf からキーファイル読み込み、失敗した場合は新規作成 */
        if (!g_key_file_load_from_file(keyfile, conffilename,
                                       G_KEY_FILE_KEEP_COMMENTS |
                                       G_KEY_FILE_KEEP_TRANSLATIONS,
                                       &error)) {
                /* デフォルト設定 */
                pconf->display.width = DISP_WIDTH;
                pconf->display.height = DISP_HEIGHT;
                pconf->control.pause = TRUE;
                pconf->control.count = TRUE;
                pconf->data.code = -1;
                pconf->data.zipcode = "";
```

```
                pconf->data.street = "";
                pconf->data.city = "";
                pconf->data.pref = "";
                for (i = 0; i < FLAG_NUM; i++)
                        flag[i] = -1;
                pconf->data.flag = flag;
                pconf->data.flag_len = FLAG_NUM;
                /* キーファイル書き込み */
                g_key_file_set_integer(keyfile, GROUP_DISPLAY, KEY_WIDTH,
                                       pconf->display.width);
                g_key_file_set_integer(keyfile, GROUP_DISPLAY, KEY_HEIGHT,
                                       pconf->display.height);
                g_key_file_set_boolean(keyfile, GROUP_CONTROL, KEY_PAUSE,
                                       pconf->control.pause);
                g_key_file_set_boolean(keyfile, GROUP_CONTROL, KEY_COUNT,
                                       pconf->control.count);
                g_key_file_set_integer(keyfile, GROUP_DATA, KEY_CODE,
                                       pconf->data.code);
                g_key_file_set_string(keyfile, GROUP_DATA, KEY_ZIPCODE,
                                      pconf->data.zipcode);
                g_key_file_set_string(keyfile, GROUP_DATA, KEY_STREET,
                                      pconf->data.zipcode);
                g_key_file_set_string(keyfile, GROUP_DATA, KEY_CITY,
                                      pconf->data.city);
                g_key_file_set_string(keyfile, GROUP_DATA, KEY_PREF,
                                      pconf->data.pref);
                g_key_file_set_integer_list(keyfile, GROUP_DATA, KEY_FLAG,
                                            pconf->data.flag,
                                            pconf->data.flag_len);
                /* キーファイルからテキストデータ取得 */
                pdata = g_key_file_to_data(keyfile, &len, &error);
                if (!pdata)
                        g_error(error->message);

                /* 新規ファイルを作成してテキストデータ書き込み */
                pconffile = fopen(conffilename, "w");
                if (!pconffile)
                        exitfail_errno("fopen");
                if (fwrite(pdata, len, 1, pconffile) < 1)
                        exitfail_errno("fwrite");
                fclose(pconffile);
        }
        /* conf 読み込みに成功した場合、設定を保持 */
        else {
                pconf->display.width =
                        g_key_file_get_integer(keyfile, GROUP_DISPLAY,
                                               KEY_WIDTH, NULL);
                pconf->display.height =
                        g_key_file_get_integer(keyfile, GROUP_DISPLAY,
                                               KEY_HEIGHT, NULL);
                pconf->control.pause =
                        g_key_file_get_boolean(keyfile, GROUP_CONTROL,
                                               KEY_PAUSE, NULL);
                pconf->control.count =
                        g_key_file_get_boolean(keyfile, GROUP_CONTROL,
                                               KEY_COUNT, NULL);
                pconf->data.code =
                        g_key_file_get_integer(keyfile, GROUP_DATA,
```

```
                                        KEY_CODE, NULL);
                pconf->data.zipcode =
                        g_key_file_get_string(keyfile, GROUP_DATA,
                                        KEY_ZIPCODE, NULL);
                pconf->data.street =
                        g_key_file_get_string(keyfile, GROUP_DATA,
                                        KEY_STREET, NULL);
                pconf->data.city =
                        g_key_file_get_string(keyfile, GROUP_DATA,
                                        KEY_CITY, NULL);
                pconf->data.pref =
                        g_key_file_get_string(keyfile, GROUP_DATA,
                                        KEY_PREF, NULL);
                pconf->data.flag =
                        g_key_file_get_integer_list(keyfile, GROUP_DATA,
                                                KEY_FLAG,
                                                &pconf->data.flag_len,
                                                NULL);
        }

        /* キーファイル開放 */
        g_key_file_free(keyfile);
}

/**
 * 行表示関数
 * @param pcsvline 表示する CSV データ構造体へのポインタ(NULL の場合は表示済データの総数を表示)
 */
static void printline(csvline_t *pcsvline)
{
        static int count = 0;   /* 表示したデータ総数 */
        static int line = 0;    /* 一画面中に表示した行数 */
        int disp_width = DISP_WIDTH, disp_height = DISP_HEIGHT, newline;
        char buf[1024];
        unsigned int i;

        /* CSV データの各要素を表示 */
        if (pcsvline) {
                /* 郵便番号が入っていない場合は表示しない */
                if (!pcsvline->zipcode[0])
                        return;

                /* 全国地方公共団体コード条件設定があり、一致しなかったら表示しない */
                if (pconf->data.code >= 0 && pcsvline->code != pconf->data.code)
                        return;
                /* 郵便番号条件設定があり、部分一致しなかったら表示しない */
                if (pconf->data.zipcode[0] != '\0' &&
                    !strcasestr(pcsvline->zipcode, pconf->data.zipcode))
                        return;
                /* 町域名条件設定があり、部分一致しなかったら表示しない */
                if (pconf->data.street[0] != '\0' &&
                    !strcasestr(pcsvline->street, pconf->data.street))
                        return;
                /* 市区町村名条件設定があり、部分一致しなかったら表示しない */
                if (pconf->data.city[0] != '\0' &&
                    !strcasestr(pcsvline->city, pconf->data.city))
                        return;
                /* 都道府県条件設定があり、部分一致しなかったら表示しない */
```

```
                if (pconf->data.pref[0] != '\0' &&
                    !strcasestr(pcsvline->pref, pconf->data.pref))
                        return;
                for (i = 0; i < FLAG_NUM && i < pconf->data.flag_len; i++) {
                        /* フラグ条件設定があり、一致しなかったら表示しない */
                        if (pconf->data.flag[i] >= 0 &&
                            pcsvline->flag[i] != pconf->data.flag[i])
                                return;
                }

                /* 各要素を表示フォーマットに展開 */
                snprintf(buf, sizeof(buf),
                         "%6d %05ld %s %s %s %s %ld %ld %ld %ld %ld %ld",
                         count++, pcsvline->code, pcsvline->zipcode,
                         pcsvline->street, pcsvline->city, pcsvline->pref,
                         pcsvline->flag[0], pcsvline->flag[1],
                         pcsvline->flag[2], pcsvline->flag[3],
                         pcsvline->flag[4], pcsvline->flag[5]);
        }
        /* CSV データの総数を表示 */
        else {
                /* 総数表示無効なら表示しない */
                if (!pconf->control.count)
                        return;
                /* データ総数を表示フォーマットに展開 */
                sprintf(buf, "Count: %6d", count);
        }

        /* 今回追加される表示行数を計算 */
        disp_width = pconf->display.width;
        disp_height = pconf->display.height;
        newline = (strlen(buf) + (disp_width - 1)) / disp_width;
        /* 1 画面を超える場合、入力があるまで一時停止 */
        if (line + newline >= disp_height) {
                /* 一時停止有効なら */
                if (pconf->control.pause)
                        getchar();
                /* 表示行数を初期化 */
                line = 0;
        }
        /* 実際に表示する */
        printf("%s\n", buf);
        /* 表示行数を更新 */
        line += newline;
}

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として読み込み CSV ファイル名を指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        FILE *pcsvfile; /* CSV ファイルポインタ */
        csvline_t csvline; /* CSV データ */
        char buf[256], *pbuf, *pcol[COLUMN_NUM];
        int i;
```

```
exitfail_init();

/* 引数が指定されなかった場合、usage 表示して終了 */
if (argc < 2) {
        printf("Usage: %s <csvfile>\n", BASENAME(argv[0]));
        return EXIT_SUCCESS;
}

/* conf ファイル名を作成 */
snprintf(buf, sizeof(buf), "%s.conf", argv[0]);
/* conf ファイルを読み込み */
readconf(buf);

/* CSV ファイルオープン */
pcsvfile = fopen(argv[1], "r");
if (!pcsvfile)
        exitfail_errno("fopen");

/* CSV データを初期化 */
memset(&csvline, 0, sizeof(csvline));
/* CSV ファイルから 1 行読み込む */
while (fgets(buf, sizeof(buf), pcsvfile)) {
        /* 各要素へのポインタ配列を初期化 */
        memset(pcol, 0, sizeof(pcol));
        /* 次のカンマ(または行末)を見つけてトークン化 */
        pbuf = strtok(buf, ",\n");
        for (i = 0; i < COLUMN_NUM && pbuf; i++) {
                /* 前後のダブルクォートは除去する */
                if (*pbuf == '"')
                        *strchrnul(++pbuf, '"') = '\0';
                /* 要素へのポインタを保持 */
                pcol[i] = pbuf;
                /* 次のカンマ(または行末)を見つけてトークン化 */
                pbuf = strtok(NULL, ",\n");
        }
        /* 要素数が不足している場合、次行にスキップ */
        if (i < COLUMN_NUM)
                continue;

        /* 新しいデータの場合 */
        if (strcmp(pcol[1], csvline.zipcode) ||
            strtol(pcol[8], NULL, 10)) {
                /* 保持済みのデータを表示 */
                printline(&csvline);

                /* CSV データ各要素を保持 */
                memset(&csvline, 0, sizeof(csvline));
                csvline.code = strtol(pcol[0], NULL, 10);
                strncpy(csvline.zipcode, pcol[1],
                        sizeof(csvline.zipcode) - 1);
                strncpy(csvline.street, pcol[2],
                        sizeof(csvline.street) - 1);
                strncpy(csvline.city, pcol[3],
                        sizeof(csvline.city) - 1);
                strncpy(csvline.pref, pcol[4],
                        sizeof(csvline.pref) - 1);
                for (i = 0; i < FLAG_NUM; i++)
```

```
                                csvline.flag[i] = strtol(pcol[5 + i], NULL, 10);
                        }
                        /* 既存データへの追加の場合 */
                        else
                                /* 町域名の続きを追加 */
                                strncat(csvline.street, pcol[2],
                                        (sizeof(csvline.street) -
                                         strlen(csvline.street)) - 1);
                }
                /* 保持済みのデータを表示 */
                printline(&csvline);

                /* CSV ファイルクローズ */
                fclose(pcsvfile);

                /* データ総数を表示 */
                printline(NULL);

                return EXIT_SUCCESS;
        }
```

図 6.17 CSV ファイルの内容を表示するプログラムの conf ファイル対応版 (dispcsv2.c)

GLib の詳細な API 説明については、以下の URL や市販の書籍などを参照してください。

GLib Reference Manual [http://library.gnome.org/devel/glib/2.16/]

ここでは簡単に流れを説明するに留めます。main 関数から見ていくと、冒頭で conf ファイル名を作成してから readconf 関数を呼んでおり、この中で conf ファイルから設定を読み込んでいます。

readconf 関数ではまず必要なメモリ領域を確保し、g_key_file_load_from_file 関数で conf ファイルを読み込みます。この戻り値で 0 が返って来たときはファイルがなかったものとしてデフォルト設定を使用し、g_key_file_to_data でテキストデータに変換してから新規 conf ファイルとして書き込みます。

conf ファイルが読み込めたときは、pconf から示される領域に各設定を保持します。この GLib の conf ファイルは、グループによって分類されるキーに対して各値が設定される形式になっています。例えば一つ目の項目の場合、グループ display のキー Width について 1 つの数値が設定されています。その後の項目のように 1 つの文字列や真偽値、複数の数値・文字列を設定するキーを作成することも可能です。

読み込まれ保持した設定は、printline 関数内で表示の制御に使用しています。ここは事前に決めた仕様どおりに動作を変更しているだけですから、コード内容を見てください。

GLib を使う時は、makefile にも注意する必要があります。GLib 用のヘッダファイルやライブラリファイルは、標準 C ライブラリ向けのものとは違う特別なディレクトリに配置されるため、これを gcc に教えてあげなくてはならないのです。それらがどこにあるかわかれば、適切に makefile に設定して gcc に渡るようにすれば良いのですが、ATDE にも入っている pkg-config というツールを使ってこれを簡略化できます。

```
CROSS   := arm-linux-gnueabi

ifneq ($(CROSS),)
CROSS_PREFIX    := $(CROSS)-
PKGCONFIG_PATH  := PKGCONFIG_PATH=/usr/$(CROSS)/lib/pkgconfig
endif
```

```
CC      = $(CROSS_PREFIX)gcc
CFLAGS  = -O2 -Wall -Wextra -I../common \

PKGCONFIG_CFLAGS        = `$(PKGCONFIG_PATH) pkg-config --cflags glib-2.0`
PKGCONFIG_LIBS          = `$(PKGCONFIG_PATH) pkg-config --libs glib-2.0`

TARGET  = dispcsv2

all: $(TARGET)

dispcsv2: dispcsv2.c
        $(CC) $(CFLAGS) $(PKGCONFIG_CFLAGS) -o $@ $< $(PKGCONFIG_LIBS)
```

図 6.18 dispcsv2 のための Makefile

指定された CROSS(=アーキテクチャ名)から PKGCONFIG_PATH を作っています。これが、目的のクロス環境 pkgconfig 情報があるパスになります。この PKGCONFIG_PATH が定義された状態で pkg-config コマンドを実行すると、CFLAGS 用のオプション(-I<dir>のヘッダファイルパス)や、LIBS(ライブラリ名)を教えてくれるのです。こうして GLib を使う場合も、それなりにシンプルに makefile を書くことができます。

これを使って make し、プログラムを実行してみます。

```
[armadillo ~]$ ./dispcsv2 ken_all_rome.csv
     0 01101 0600000 IKANIKEISAIGANAIBAAI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 0 0 0 0
     1 01101 0640941 ASAHIGAOKA CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     2 01101 0600041 ODORIHIGASHI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     3 01101 0600042 ODORINISHI(1-19-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     4 01101 0640820 ODORINISHI(20-28-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     5 01101 0600031 KITA1-JOHIGASHI CHUO-KU SAPPORO-SHI HOKKAIDO 0 0 1 0 0 0
     6 01101 0600001 KITA1-JONISHI(1-19-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
     7 01101 0640821 KITA1-JONISHI(20-28-CHOME) CHUO-KU SAPPORO-SHI HOKKAIDO 1 0 1 0 0 0
```

図 6.19 dispcsv2 の実行結果

1 回目の動作は先ほどのものとまったく変わりませんが、Ctrl+C で終了すると dispcsv2.conf ファイルができています。

```
[dispaly]
Width=60
Height=17

[control]
Pause=true
Count=true

[data]
Code=-1
Zipcode=
Street=
City=
Pref=
Flag=-1;-1;-1;-1;-1;-1;
```

角括弧で囲われた単語がグループ、その下にイコール記号を使って値が設定されている単語がキーになります。以下のように動作を変更させてみます。

1. 画面サイズは 80x24
2. 一時停止しない
3. 町域名に KOKUBUNJI を含んだもののみ出力する

```
[dispaly]
Width=80
Height=24

[control]
Pause=false
Count=true

[data]
Code=-1
Zipcode=
Street=kokubunji
City=
Pref=
Flag=-1;-1;-1;-1;-1;-1;
```

テキストエディタでこのように dispcsv2.conf を変更して実行すると、以下のように動作します。

```
[armadillo ~]$ ./dispcsv2 ken_all_rome.csv
     0 09216 3290417 KOKUBUNJI SHIMOTSUKE-SHI TOCHIGI 0 0 0 0 0 0
     1 12219 2900071 KITAKOKUBUNJIDAI ICHIHARA-SHI CHIBA 0 0 1 0 0 0
     2 12219 2900073 KOKUBUNJIDAICHUO ICHIHARA-SHI CHIBA 0 0 1 0 0 0
     3 12219 2900072 NISHIKOKUBUNJIDAI ICHIHARA-SHI CHIBA 0 0 1 0 0 0
     4 12219 2900074 HIGASHIKOKUBUNJIDAI ICHIHARA-SHI CHIBA 0 0 1 0 0 0
     5 12219 2900075 MINAMIKOKUBUNJIDAI ICHIHARA-SHI CHIBA 0 0 1 0 0 0
     6 14215 2430413 KOKUBUNJIDAI EBINA-SHI KANAGAWA 0 0 1 0 0 0
     7 15222 9420088 BISHAMONKOKUBUNJI JOETSU-SHI NIIGATA 0 0 0 0 0 0
     8 15224 9520304 KOKUBUNJI SADO-SHI NIIGATA 0 0 0 0 0 0
     9 27127 5310064 KOKUBUNJI KITA-KU OSAKA-SHI OSAKA 0 0 1 0 0 0
    10 28201 6710234 MIKUNINOCHO KOKUBUNJI HIMEJI-SHI HYOGO 0 0 0 0 0 0
    11 28209 6695341 HIDAKACHO KOKUBUNJI TOYOKA-SHI HYOGO 0 0 0 0 0 0
    12 31201 6800155 KOKUFUCHO KOKUBUNJI TOTTORI-SHI TOTTORI 0 0 0 0 0 0
    13 31203 6820943 KOKUBUNJI KURAYOSHI-SHI TOTTORI 0 0 0 0 0 0
    14 33203 7080843 KOKUBUNJI TSUYAMA-SHI OKAYAMA 0 0 0 0 0 0
    15 35206 7470021 KOKUBUNJICHO HOFU-SHI YAMAGUCHI 0 0 0 0 0 0
    16 37201 7690105 KOKUBUNJICHO KASHIHARA TAKAMATSU-SHI KAGAWA 0 0 0 0 0 0
    17 37201 7690102 KOKUBUNJICHO KOKUBU TAKAMATSU-SHI KAGAWA 0 0 0 0 0 0
    18 37201 7690104 KOKUBUNJICHO SHIMMYO TAKAMATSU-SHI KAGAWA 0 0 0 0 0 0
    19 37201 7690101 KOKUBUNJICHO NII TAKAMATSU-SHI KAGAWA 0 0 0 0 0 0
    20 37201 7690103 KOKUBUNJICHO FUKE TAKAMATSU-SHI KAGAWA 0 0 0 0 0 0
    21 46215 8950073 KOKUBUNJICHO SATSUMASENDAI-SHI KAGOSHIMA 0 0 0 0 0 0
Count:     22
```

図 6.20 dispcsv2.conf を編集した dispcsv2 の実行結果

6.3.3. バイナリファイルを扱う

バイナリファイルを扱う例として、Windows で使われる BMP 形式の画像ファイルを表示してみます。とはいっても、グラフィック画面は使用しません。コンソール上のみで実行できるように、エスケープシーケンスを使用したカラー対応のアスキーアート(文字を使った擬似画像)表示サンプルプログラムです。

```
#ifndef BITMAP_H
#define BITMAP_H

#include <stdint.h>

/* ビットマップファイルヘッダ構造体(オリジナル) */
/*
typedef struct tagBITMAPFILEHEADER {
        uint16_t bfType;
        uint32_t bfSize;        // アラインメント不正(2 番地から始まる 4 バイト変数)
        uint16_t bfReserved1;
        uint16_t bfReserved2;
        uint32_t bfOffBits;     // アラインメント不正(10 番地から始まる 4 バイト変数)
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
*/

/* ビットマップファイルヘッダ構造体 */
typedef struct tagBITMAPFILEHEADER {
        uint16_t bfType;
        uint16_t bfSize_l;      // Size 下位 16 ビット
        uint16_t bfSize_h;      // Size 上位 16 ビット
        uint16_t bfReserved1;
        uint16_t bfReserved2;
        uint16_t bfOffBits_l;   // OffBits 下位 16 ビット
        uint16_t bfOffBits_h;   // OffBits 上位 16 ビット
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;

/* ビットマップファイルヘッダメンバ取得マクロ */
/* Size と OffBits は上位 16bit と下位 16bit を別々に取得して 32bit に合成する */
#define BITMAPFILEHEADER_TYPE(pbf)      (((PBITMAPFILEHEADER)(pbf))->bfType)
#define BITMAPFILEHEADER_SIZE(pbf)                                      \
        (((uint32_t)((PBITMAPFILEHEADER)(pbf))->bfSize_h << 16) |       \
         ((uint32_t)((PBITMAPFILEHEADER)(pbf))->bfSize_l      ))
#define BITMAPFILEHEADER_OFFBITS(pbf)                                   \
        (((uint32_t)((PBITMAPFILEHEADER)(pbf))->bfOffBits_h << 16) |    \
         ((uint32_t)((PBITMAPFILEHEADER)(pbf))->bfOffBits_l      ))

/* ビットマップファイルタイプ識別子 */
#define BF_TYPE (*(uint16_t *)"BM")

/* ビットマップ情報ヘッダ構造体 */
typedef struct tagBITMAPINFOHEADER {
        uint32_t biSize;
        int32_t  biWidth;
        int32_t  biHeight;
        uint16_t biPlanes;
        uint16_t biBitCount;
        uint32_t biCompression;
        uint32_t biSizeImage;
        int32_t  biXPelsPerMeter;
```

```
        int32_t  biYPelsPerMeter;
        uint32_t biClrUsed;
        uint32_t biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;

/* ビットマップ情報ヘッダメンバ取得マクロ */
#define BITMAPINFOHEADER_SIZE(pbi)      (((PBITMAPINFOHEADER)(pbi))->biSize)
#define BITMAPINFOHEADER_WIDTH(pbi)     (((PBITMAPINFOHEADER)(pbi))->biWidth)
#define BITMAPINFOHEADER_HEIGHT(pbi)    (((PBITMAPINFOHEADER)(pbi))->biHeight)
#define BITMAPINFOHEADER_PLANES(pbi)    (((PBITMAPINFOHEADER)(pbi))->biPlanes)
#define BITMAPINFOHEADER_BITCOUNT(pbi)  (((PBITMAPINFOHEADER)(pbi))->biBitCount)
#define BITMAPINFOHEADER_COMPRESSION(pbi)       \
        (((PBITMAPINFOHEADER)(pbi))->biCompression)
#define BITMAPINFOHEADER_SIZEIMAGE(pbi) \
        (((PBITMAPINFOHEADER)(pbi))->biSizeImage)
#define BITMAPINFOHEADER_XPELSPERMETER(pbi)     \
        (((PBITMAPINFOHEADER)(pbi))->biXPelsPerMeter)
#define BITMAPINFOHEADER_YPELSPERMETER(pbi)     \
        (((PBITMAPINFOHEADER)(pbi))->biYPelsPerMeter)
#define BITMAPINFOHEADER_CLRUSED(pbi)   (((PBITMAPINFOHEADER)(pbi))->biClrUsed)
#define BITMAPINFOHEADER_CLRIMPORTANT(pbi)      \
        (((PBITMAPINFOHEADER)(pbi))->biClrImportant)

#endif /* BITMAP_H */
```

図 6.21 BMP ファイル形式構造定義ヘッダファイル(bitmap.h)

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

/* ビットマップファイル形式用ヘッダ */
#include "bitmap.h"

/* 読み込み可能なビットマップファイルの条件(固定値) */
#define BMP_PLANES      1  /* プレーン数 */
#define BMP_BITCOUNT    24 /* 色深度 */
#define BMP_COMPRESSION 0  /* 圧縮方式 */

/* 読み込む最大画素サイズ */
#define MAX_WIDTH    480 /* 幅 */
#define MAX_HEIGHT   272 /* 高さ */
/* キャラクタ化する画素単位 */
#define PIXEL_WIDTH  8  /* 幅 */
#define PIXEL_HEIGHT 16 /* 高さ */
/* 表示するキャラクタ個数 */
#define DISP_WIDTH   (MAX_WIDTH / PIXEL_WIDTH)   /* 幅 */
#define DISP_HEIGHT  (MAX_HEIGHT / PIXEL_HEIGHT) /* 高さ */

/* 色変換用境界値 */
#define BOUNDARY_FG (0x55 * PIXEL_WIDTH * PIXEL_HEIGHT) /* 前景 */
#define BOUNDARY_BG (0xaa * PIXEL_WIDTH * PIXEL_HEIGHT) /* 背景 */
```

```
/* 色変換用構造体 */
/* 同一キャラクタに当たる複数の画素について各色を加算していき、
   境界値と比較して色変換を行う */
typedef struct {
        uint16_t r; /* 赤 */
        uint16_t g; /* 緑 */
        uint16_t b; /* 青 */
} color_t;

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として読み込みビットマップファイル名を指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        FILE *pbmpfile; /* ビットマップファイルポインタ */
        BITMAPFILEHEADER bf; /* ビットマップファイルヘッダ */
        uint32_t offbits; /* ファイル先頭から画素データへのオフセットサイズ */
        BITMAPINFOHEADER bi; /* ビットマップ情報ヘッダ */
        int32_t bmp_width, bmp_height; /* ビットマップファイル画素サイズ */
        color_t pixels[DISP_HEIGHT][DISP_WIDTH]; /* 色変換用画素データ蓄積 */
        uint8_t buf[MAX_WIDTH][3], *ptmp; /* 画素データ読み込み用バッファ */
        int dx, dy, px, py; /* 画素シーク位置 */

        exitfail_init();

        /* 引数が指定されなかった場合、usage 表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <bmpfile>\n", BASENAME(argv[0]));
                return 0;
        }

        /* ビットマップファイルオープン */
        pbmpfile = fopen(argv[1], "r");
        if (!pbmpfile)
                exitfail_errno("fopen");

        /* ビットマップファイルヘッダ読み込み */
        if (fread(&bf, sizeof(bf), 1, pbmpfile) < 1)
                exitfail_errno("fread");
        /* ビットマップファイルタイプが正しい識別子か */
        if (BITMAPFILEHEADER_TYPE(&bf) != BF_TYPE)
                exitfail("Bad bitmap file header type\n");
        /* ビットマップヘッダサイズが十分か */
        offbits = BITMAPFILEHEADER_OFFBITS(&bf);
        if (offbits < sizeof(bf) + sizeof(bi))
                exitfail("Bad bitmap header size\n");

        /* ビットマップ情報ヘッダ読み込み */
        if (fread(&bi, sizeof(bi), 1, pbmpfile) < 1)
                exitfail_errno("fread");
        /* ビットマップ情報ヘッダサイズが十分か */
        if (BITMAPINFOHEADER_SIZE(&bi) < sizeof(bi))
                exitfail("Bad bitmap info header size\n");
```

```
        /* 読み込み可能なビットマップファイルか */
        if (BITMAPINFOHEADER_PLANES(&bi) != BMP_PLANES ||
            BITMAPINFOHEADER_BITCOUNT(&bi) != BMP_BITCOUNT ||
            BITMAPINFOHEADER_COMPRESSION(&bi) != BMP_COMPRESSION)
                exitfail("Bad bitmap type\n");
        /* ビットマップサイズが十分か */
        bmp_width = BITMAPINFOHEADER_WIDTH(&bi);
        bmp_height = BITMAPINFOHEADER_HEIGHT(&bi);
        if (bmp_width < MAX_WIDTH || bmp_height < MAX_HEIGHT)
                exitfail("Bad bitmap size\n");

        /* 画素データ蓄積配列をゼロクリア */
        memset(pixels, 0, sizeof(pixels));

        /* 読み込む最初の画素(MAX_HEIGHT-1 行目)までシーク */
        if (fseek(pbmpfile, offbits + 3 * bmp_width * (bmp_height - MAX_HEIGHT),
                  SEEK_SET) < 0)
                exitfail_errno("fseek");
        /* 最終行から順に画素データを蓄積 */
        for (dy = DISP_HEIGHT - 1; dy >= 0; dy--)
                for (py = PIXEL_HEIGHT - 1; py >= 0; py--) {
                        /* 画素データを 1 行分読み込む */
                        if (fread(buf, sizeof(buf), 1, pbmpfile) < 1)
                                exitfail_errno("fread");
                        /* 画素データを加算していく */
                        for (dx = 0; dx < DISP_WIDTH; dx++)
                                for (px = 0; px < PIXEL_WIDTH; px++) {
                                        ptmp = buf[PIXEL_WIDTH * dx + px];
                                        pixels[dy][dx].r += (uint16_t)ptmp[2];
                                        pixels[dy][dx].g += (uint16_t)ptmp[1];
                                        pixels[dy][dx].b += (uint16_t)ptmp[0];
                                }
                        /* 次行先頭画素までシーク */
                        if (fseek(pbmpfile, 3 * bmp_width - sizeof(buf),
                                  SEEK_CUR) < 0)
                                exitfail_errno("fseek");
                }

        /* ビットマップファイルクローズ */
        fclose(pbmpfile);

        /* 先頭行から順にキャラクタを描画していく */
        for (dy = 0; dy < DISP_HEIGHT; dy++) {
                printf("\n");
                for (dx = 0; dx < DISP_WIDTH; dx++) {
                        /* 境界値と比較して
                           色付けエスケープシーケンスとキャラクタを出力 */
                        printf("\x1b[3%dm\x1b[4%dm%c",
                               ((pixels[dy][dx].r > BOUNDARY_FG) ? 1 : 0) |
                               ((pixels[dy][dx].g > BOUNDARY_FG) ? 2 : 0) |
                               ((pixels[dy][dx].b > BOUNDARY_FG) ? 4 : 0),
                               ((pixels[dy][dx].r > BOUNDARY_BG) ? 1 : 0) |
                               ((pixels[dy][dx].g > BOUNDARY_BG) ? 2 : 0) |
                               ((pixels[dy][dx].b > BOUNDARY_BG) ? 4 : 0),
                               ((dx ^ dy) & 1) ? '_' : '/');
                }
                /* 色付けをクリアするエスケープシーケンスを出力 */
                printf("\x1b[0m");
```

```
        }
        /* 入力があるまで一時停止 */
        getchar();

        return 0;
}
```

図 6.22 BMP 形式画像ファイルのコンソール表示プログラム(dispbmp.c)

bitmap.h は、BMP ファイル形式のヘッダを解析するための構造体を定義したヘッダファイルです。dispbmp.c の main 関数と平行して、見ていきます。

main 関数は、まず引数で指定された bmp ファイルをオープンし、そして BITMAPFILEHEADER 構造体分のデータを読み込んでヘッダの中身を解析していきます。まずは TYPE が正しく BM であることから順次チェックしていくのですが、2 番目の要素である SIZE の読み込みで少々困ったことになります。

BITMAPFILEHEADER 構造体の先頭の要素である bfType が 2 バイトであり、次の bfSize が 4 バイトであるため、構造体メンバとして直接参照しようとするとアドレス 2 から 4 バイト参照することになり、ARM アーキテクチャでは正しく値を読むことができません。このため、このサンプルでは bfSize メンバを 4 バイトとして直接定義せず、bfSize_l/bfSize_h と上位下位 2 バイトずつのメンバとして分断させ、別々に読み込んだものを一つの 4 バイトとして扱うためのサポートマクロ BITMAPFILEHEADER_SIZE を用意する方法を取りました。こうすることで、呼び出し側は分断されたデータであることを意識することなく、要素の比較ができます。

ヘッダ 2 種類の解析の後、その情報に基づいた形で画像データを読み込んでいきます。アスキーアート化するため、いくつかの並んだ画素を一つとして RGB 別に配列に蓄えていき、この値の大きさで出力する色を決定します。色付けの決定方法は、ここでは本筋から外れますので省略します。

最終段の出力は、printf で行っています。色付けはエスケープシーケンスという手法を用いており、0x1b に相当するコントロールコードの後に前景/背景と色を指定するための情報を付加して出力します。

サンプル BMP ファイルの madillon.bmp を指定して実行すると、以下のように出力されます。

```
[armadillo ~]$ ./dispbmp midomadillo.bmp
```

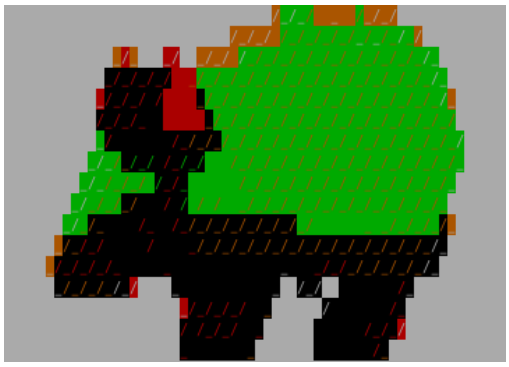


図 6.23 dispbmp の実行結果

6.4. デバイスの操作

「3.3.1. ファイルの種類」で説明したように、Linux システムを含む UNIX システムでは、すべてをファイルとして表現します。

Armadillo というハードウェアが持つ、シリアルインターフェースや GPIO、LED、スイッチなどのデバイスも例外ではありません。Linux カーネルは、これらのデバイスをファイルとして扱えるように抽象化します。

本章では、このようなファイルを扱う方法について説明します。

6.4.1. デバイスファイルを使う

デバイスを抽象化したファイルで最も一般的なものは、デバイスファイル[11]です。

デバイスファイルには、キャラクタデバイスとブロックデバイスの 2 種類があります。それぞれの特徴は、「3.3.1. ファイルの種類」を参照してください。

デバイスファイルは、通常、/dev ディレクトリ以下にあります。**ls -l** を実行したときに、一番左に表示される文字が c のファイルがキャラクタデバイスで、b がブロックデバイスです。

デバイスファイルを C 言語で扱うには、通常ファイルと同様に、open、close、read、write システムコールを使用します。

また、デバイスファイル特有のシステムコールとして、ioctl があります。ioctl では、デバイスのパラメータを変更するなど、通常の read/write 操作とは馴染まないデバイスへの操作を行うために使用されます。ioctl の使い方は、対象となるデバイスによって異なります。

デバイスファイルを扱う例は、「6.5. シリアルポートの入出力」で説明します。

6.4.2. sysfs ファイルシステムを使う

sysfs ファイルシステムは、proc ファイルシステムに似た特殊ファイルシステムです。ユーザーランドアプリケーションは、sysfs ファイルシステムを通して、カーネル内部のデータ構造にアクセスできます。sysfs ファイルシステムが提供するファイルのいくつかは、物理的なデバイスに対応しており、それらに対して読み書きすることで、デバイスを制御することができます。

sysfs ファイルシステムは、通常、/sys ディレクトリにマウントします。

sysfs を扱う例として、LED の制御について説明します。

LED は、Linux システムでは LED クラスとして汎用化されています。LED クラスとして登録された LED に対する操作は、/sys/class/leds/以下のディレクトリによって行います。/sys/class/leds/(LED 名)/brightness という名前のファイルに対して、0 という文字を書き込むと LED が消灯します。また、1 を書き込むと点灯します。詳しい仕様は「Armadillo-400 シリーズソフトウェアマニュアル」の「8. Linux カーネルデバイスドライバー仕様/8.10.LED」を参照してください。

Armadillo-400 シリーズでは、red、green、yellow の 3 個の LED を LED クラスとして扱えます。

シェルから LED を点灯/消灯する例を、以下に示します。

```
[armadillo ~]# echo 1 > /sys/class/leds/red/brightness    ❶
[armadillo ~]# echo 0 > /sys/class/leds/red/brightness    ❷
```

図 6.24 シェルから LED を点灯/消灯する

[11]スペシャルファイル(特殊ファイル)やデバイスノードとも呼ばれます

❶ red LED を点灯します

❷ red LED を消灯します

C 言語での sysfs ファイルシステムのファイルの扱いは、通常ファイルと同じです。LED を扱う簡単な例を、「図 6.25. LED の点灯/消灯を行うプログラム(led_on_off.c)」に示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>

#define LED_CLASS_PATH "/sys/class/leds"

int main(int argc, char *argv[])
{
        int fd;
        char path[PATH_MAX];
        ssize_t len;

        if (argc < 3) {
                printf("usage: %s <led name> <brightness>\n", argv[0]);
                return EXIT_SUCCESS;
        }

        snprintf(path, PATH_MAX, "%s/%s/brightness",
                 LED_CLASS_PATH, argv[1]);

        fd = open(path, O_WRONLY);
        if (fd == -1) {
                perror("open");
                return EXIT_FAILURE;
        }

        len = write(fd, argv[2], strlen(argv[2]));
        if (len == -1) {
                perror("write");
                return EXIT_FAILURE;
        }

        close(fd);

        return EXIT_SUCCESS;
}
```

図 6.25 LED の点灯/消灯を行うプログラム(led_on_off.c)

```
[armadillo ~]# ./led_on_off red 1      ❶
[armadillo ~]# ./led_on_off red 0      ❷
[armadillo ~]# ./led_on_off green 1    ❸
[armadillo ~]# ./led_on_off green 0    ❹
```

図 6.26 led_on_off の実行例

❶ red LED を点灯します

❷ red LED を消灯します

❸ green LED を点灯します

❹ green LED を消灯します

6.5. シリアルポートの入出力

シリアルポートで入出力を行うプログラムを作ってみます。

Linux では、テキストデータを扱うコンソール端末用として、行単位に文字を扱ったり自動的に変換したりしたりしてくれるカノニカルモードがあるのですが、意図したデータをそのまま転送したい場合には不都合です。バイナリデータも自由に扱えるようにするには非カノニカルモードに設定する必要があります。ここでは非カノニカルモードを使います。

6.5.1. シリアルエコーサーバー

シリアルポートから受け取ったデータをそのまま返してくれる、エコーサーバーを作ってみます。

1. 9600bps で接続する。
2. 8bit データ、パリティなし、フロー制御なし

まずはこの条件だけの、シンプルなものにします。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <termios.h>
#include <unistd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

#define SERIAL_BAUDRATE B9600

#define BUF_SIZE 256

static int serial_fd = -1; /* シリアルポートファイルディスクリプタ */
static struct termios old_tio; /* 元のシリアルポート設定 */
```

```
static int terminated = 0; /* 終了シグナル発生フラグ */

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

#define __unused __attribute__((unused))

/**
 * 終了シグナルハンドラ
 * @param signum シグナル番号(不使用)
 */
static void terminate_sig_handler(__unused int signum)
{
        /* 終了シグナル発生を記録 */
        terminated = 1;
}

/**
 * シグナルハンドラ設定関数
 * @param sig_list ハンドラを設定するシグナルのリスト
 * @param num シグナルリストの要素数
 * @param handler 設定するハンドラ関数
 */
static void set_sig_handler(int sig_list[], ssize_t num, __sighandler_t handler)
{
        struct sigaction sa;
        int i;

        /* ハンドラ関数を設定 */
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = handler;

        /* 各シグナルに対して関連付け */
        for(i = 0; i < num; i++)
                if (sigaction(sig_list[i], &sa, NULL) < 0)
                        exitfail_errno("sigaction");
}

/**
 * シリアルポート設定復元関数
 */
static void restore_serial(void)
{
        int ret;

        /* シリアルポートの設定を元に戻す */
        ret = tcsetattr(serial_fd, TCSANOW, &old_tio);
        if (ret < 0)
                exitfail_errno("tcsetattr");
}

/**
 * シリアルポート設定関数
 * @param fd 設定するシリアルポートファイルディスクリプタ
 */
static void setup_serial(int fd)
{
```

```
        struct termios tio;
        int ret;

        /* 現在のシリアルポートの設定を退避する */
        ret = tcgetattr(fd, &old_tio);
        if (ret)
                exitfail_errno("tcgetattr");

        /* 終了時に設定を復元するための関数を登録 */
        if (atexit(restore_serial))
                exitfail_errno("atexit");

        /* 新しいシリアルポートの設定 */
        memset(&tio, 0, sizeof(tio));
        tio.c_iflag = IGNBRK | IGNPAR; /* ブレーク文字無視/パリティなし */
        tio.c_cflag = CS8 | CLOCAL | CREAD; /* フロー制御なし/8bit/非モデム/受信可 */
        tio.c_cc[VTIME] = 0; /* キャラクタ間タイマー無効 */
        tio.c_cc[VMIN] = 1; /* 最低1文字送信/受信するまでブロックする */
        ret = cfsetspeed(&tio, SERIAL_BAUDRATE); /* 入出力ボーレート */
        if (ret < 0)
                exitfail_errno("cfsetspeed");

        /* バッファ内のデータをフラッシュ */
        ret = tcflush(fd, TCIFLUSH);
        if (ret < 0)
                exitfail_errno("tcflush");

        /* 新しいシリアルポート設定を適用 */
        ret = tcsetattr(fd, TCSANOW, &tio);
        if (ret)
                exitfail_errno("tcsetattr");
}

/**
 * main関数
 * @param argc 引数なしの場合はusage表示のみ
 * @param argv 第1引数としてシリアルデバイス名を指定
 * @return exit値
 */
int main(int argc, char *argv[])
{
        int terminate_sig_list[] = { /* 終了シグナル種類 */
                SIGHUP, SIGINT, SIGQUIT, SIGPIPE, SIGTERM
        };
        char buf[BUF_SIZE];
        ssize_t ret, len, wrlen;

        exitfail_init();

        /* 引数が指定されなかった場合、usage表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <device>\n", BASENAME(argv[0]));
                return EXIT_SUCCESS;
        }

        /* シリアルポートを読み書き可能な非制御端末としてオープン */
        serial_fd = open(argv[1], O_RDWR | O_NOCTTY);
        if (serial_fd < 0)
```

```
                exitfail_errno("open");

        /* 終了シグナルに対してハンドラを設定 */
        set_sig_handler(terminate_sig_list, ARRAY_SIZE(terminate_sig_list),
                        terminate_sig_handler);

        /* シリアルポートを設定 */
        setup_serial(serial_fd);

        /* 終了シグナルが発生していない限りループ */
        while (!terminated) {
                /* シリアルポートから読み込み */
                ret = read(serial_fd, buf, BUF_SIZE);
                if (ret < 0) {
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("read");
                }
                len = ret;

                /* すべてのデータを書き込むまでループ(終了シグナル発生で中断) */
                for (wrlen = 0; wrlen < len && !terminated; wrlen += ret) {
                        /* シリアルポートに書き込み */
                        ret = write(serial_fd, buf + wrlen, len - wrlen);
                        if (ret < 0) {
                                if (errno == EINTR) {
                                        /* シグナル発生時はリトライ */
                                        ret = 0;
                                        continue;
                                }
                                exitfail_errno("write");
                        }
                }
        }

        return EXIT_SUCCESS;
}
```

図 6.27 シリアルエコーサーバー(serial_echo_server1.c)

ポイントがいくつかあります。まず先に、シリアルポートの設定を行っているところを見てみます。setup_serial 関数がそれですが、ここで struct termios 構造体 tio のメンバに対して値を書き込んでいるところが重要です。c_iflag や c_cflag に対して、8bit/パリティなし/フロー制御なしであること、また非カノニカルモードにすることを意識して、適切な値を書かなければなりません。ボーレートの設定については、それらとは別に cfsetspeed 関数を使います。

tcsetattr 関数で、作成した tio 状態を実際にシリアルデバイスに反映させるのですが、ここで 1 点重要なことがあります。この設定はこのプログラム内のみならずシステム全体に影響してしまうものであり、プログラムが正常に終了したり、また不正な終了や(Ctrl+C 入力によるような)強制終了が発生した場合、初期状態に戻ってはくれません。つまり、行儀よくバグの少ないコードを書こうとするならば、どのような終了時であっても初期状態に戻してあげるような注意が必要なのです。

このプログラムでは、atexit 関数とシグナルハンドラを使ってこれを実現しています。setup_serial 関数内で呼ばれている atexit 関数は、exit される時(main 関数からの return 時も含みます)に呼ばれて欲しいハンドラ関数を登録するためのものです。これを使って restore_serial 関数が登録されています

ので、終了時には必ずシリアルポート設定の状態が復帰されます。また、いくつかの不正/強制終了に対応するために、set_sig_handle でシグナルハンドラを登録しています。

もう 1 つのポイントは、read/write 関数のエラー処理でしょう。これらの関数は、シグナルが発生された時に中断して、戻り値-1 となることがあります。このプログラムの目的としては、これを致命的なエラーとして扱うのは適切ではありません。このため、errno が EINT(シグナル発生による中断を表す)であった場合は continue してリトライするような作りにしています。なお、この際に Ctrl+C による中断シグナル(SIGINT)などであった場合には、グローバル変数 terminated をチェックしてきちんとループを抜けるようにしてあります。

空いている(現在コンソールとして使っている)シリアルポートと、PC の空いているシリアルポート(もちろん USB シリアルデバイスでも構いません)をクロスケーブルで接続し、PC 側では Tera Term を立ち上げます。ここでの設定は、プログラムに合わせ 9600bps/8bit/パリティなし/フロー制御なしとします。

この状態で、シリアルポート名を引数として「図 6.28. serial_echo_server1 の実行例」を実行します。

```
[armadillo ~]# ./serial_echo_server1 ttymxc2
```

図 6.28 serial_echo_server1 の実行例

Tera Term から入力した文字が、そのまま返ってくるのが確認できます。

6.5.2. 改行コードの違いを吸収する

先ほどのシリアルエコーサーバーを Windows 版 Tera Term で試すと、改行した際に次の行に行かず、現在入力している行の先頭から上書きされるような状態になってしまいます。これは、Linux と Windows で改行を表すコードが異なるためです。

Linux では、改行コードとして LF(ラインフィード)、バイナリで表すと 0x0a[12]を使用します。これに対し、Windows では改行コードとして CR(キャリッジリターン)+LF、バイナリで表すと 0x0d[13], 0x0a という連続 2 文字を使用します。さらに Tera Term のデフォルトの状態は、改行コードとして CR の 1 文字のみを送出する状態になっています[14]。

先ほどのプログラムをちょっと改造して、この違いを吸収してみましょう。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <termios.h>
#include <unistd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"
```

[12]C 言語コード中でキャラクタ文字表現する際の\n にあたります。
[13]C 言語コード中でキャラクタ文字表現する際の\r にあたります。
[14]Tera Term の「設定」-「端末」メニューをクリックするとすると、改行コードの「送信」として「CR」か「CR+LF」を選べることが確認できます。

```
#define SERIAL_BAUDRATE B9600

#define BUF_SIZE  256
#define READ_SIZE (BUF_SIZE / 2)

static int serial_fd = -1; /* シリアルポートファイルディスクリプタ */
static struct termios old_tio; /* 元のシリアルポート設定 */

static int terminated = 0; /* 終了シグナル発生フラグ */

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

#define __unused __attribute__((unused))

/**
 * 終了シグナルハンドラ
 * @param signum シグナル番号(不使用)
 */
static void terminate_sig_handler(__unused int signum)
{
        /* 終了シグナル発生を記録 */
        terminated = 1;
}

/**
 * シグナルハンドラ設定関数
 * @param sig_list ハンドラを設定するシグナルのリスト
 * @param num シグナルリストの要素数
 * @param handler 設定するハンドラ関数
 */
static void set_sig_handler(int sig_list[], ssize_t num, __sighandler_t handler)
{
        struct sigaction sa;
        int i;

        /* ハンドラ関数を設定 */
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = handler;

        /* 各シグナルに対して関連付け */
        for(i = 0; i < num; i++)
                if (sigaction(sig_list[i], &sa, NULL) < 0)
                        exitfail_errno("sigaction");
}

/**
 * シリアルポート設定復元関数
 */
static void restore_serial(void)
{
        int ret;

        /* シリアルポートの設定を元に戻す */
        ret = tcsetattr(serial_fd, TCSANOW, &old_tio);
        if (ret < 0)
                exitfail_errno("tcsetattr");
```

```
}

/**
 * シリアルポート設定関数
 * @param fd 設定するシリアルポートファイルディスクリプタ
 */
static void setup_serial(int fd)
{
        struct termios tio;
        int ret;

        /* 現在のシリアルポートの設定を退避する */
        ret = tcgetattr(fd, &old_tio);
        if (ret)
                exitfail_errno("tcgetattr");

        /* 終了時に設定を復元するための関数を登録 */
        if (atexit(restore_serial))
                exitfail_errno("atexit");

        /* 新しいシリアルポートの設定 */
        memset(&tio, 0, sizeof(tio));
        tio.c_iflag = IGNBRK | IGNPAR; /* ブレーク文字無視/パリティなし */
        tio.c_cflag = CS8 | CLOCAL | CREAD; /* フロー制御なし/8bit/非モデム/受信可 */
        tio.c_cc[VTIME] = 0; /* キャラクタ間タイマー無効 */
        tio.c_cc[VMIN] = 1; /* 最低 1 文字送信/受信するまでブロックする */
        ret = cfsetspeed(&tio, SERIAL_BAUDRATE); /* 入出力ボーレート */
        if (ret < 0)
                exitfail_errno("cfsetspeed");

        /* バッファ内のデータをフラッシュ */
        ret = tcflush(fd, TCIFLUSH);
        if (ret < 0)
                exitfail_errno("tcflush");

        /* 新しいシリアルポート設定を適用 */
        ret = tcsetattr(fd, TCSANOW, &tio);
        if (ret)
                exitfail_errno("tcsetattr");
}

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数としてシリアルデバイス名を指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        int terminate_sig_list[] = { /* 終了シグナル種類 */
                SIGHUP, SIGINT, SIGQUIT, SIGPIPE, SIGTERM
        };
        char buf[BUF_SIZE];
        ssize_t ret, len, wrlen;
        int lf_flag = 0; /* LF 挿入処理発生フラグ */
        int i;

        exitfail_init();
```

```
/* 引数が指定されなかった場合、usage 表示して終了 */
if (argc < 2) {
        printf("Usage: %s <device>\n", BASENAME(argv[0]));
        return EXIT_SUCCESS;
}

/* シリアルポートを読み書き可能な非制御端末としてオープン */
serial_fd = open(argv[1], O_RDWR | O_NOCTTY);
if (serial_fd < 0)
        exitfail_errno("open");

/* 終了シグナルに対してハンドラを設定 */
set_sig_handler(terminate_sig_list, ARRAY_SIZE(terminate_sig_list),
                terminate_sig_handler);

/* シリアルポートを設定 */
setup_serial(serial_fd);

/* 終了シグナルが発生していない限りループ */
while (!terminated) {
        /* シリアルポートから読み込み */
        ret = read(serial_fd, buf, READ_SIZE);
        if (ret < 0) {
                /* シグナル発生時はリトライ */
                if (errno == EINTR)
                        continue;
                exitfail_errno("read");
        }
        len = ret;

        /* LF 挿入処理直後の LF の場合、1 文字無視させる */
        if (lf_flag) {
                if (buf[0] == '\n')
                        memmove(buf, buf + 1, --len);
                lf_flag = 0;
        }
        /* データ最終文字が CR だった場合のために 1 文字潰しておく */
        buf[len] = '\0';
        /* データを最後まで検査 */
        for (i = 0; i < len; i++) {
                /* CR を発見 */
                if (buf[i] == '\r')
                        /* 直後が LF ではない */
                        if (buf[++i] != '\n') {
                                /* LF 挿入処理 */
                                if (i < len) {
                                        /* まだデータがあるなら後ろにずらす */
                                        memmove(buf + i + 1, buf + i,
                                                len - i);
                                }
                                else
                                        /* LF 挿入処理発生を保持 */
                                        lf_flag = 1;
                                buf[i] = '\n';
                                len++;
                        }
        }
```

```
                /* すべてのデータを書き込むまでループ(終了シグナル発生で中断) */
                for (wrlen = 0; wrlen < len && !terminated; wrlen += ret) {
                        /* シリアルポートに書き込み */
                        ret = write(serial_fd, buf + wrlen, len - wrlen);
                        if (ret < 0) {
                                if (errno == EINTR) {
                                        /* シグナル発生時はリトライ */
                                        ret = 0;
                                        continue;
                                }
                                exitfail_errno("write");
                        }
                }
        }

        return EXIT_SUCCESS;
}
```

図 6.29 改行コード変換を行うシリアルエコーサーバー(serial_echo_server2.c)

大きく追加された箇所は、main 関数後半の while ループ内です。CR を受け取り、その次が LF でなかった場合は LF を補ってあげることで、Windows 上でも改行状態として見えるように改変するロジックが入っています。

なお、この LF 挿入処理が発生した場合は、データサイズが大きくなっていく(最大で元データの 2 倍)ことになるため、READ_SIZE を定義して read 時点ではバッファの半分までしか使用しないように変更しています。

6.5.3. より効率的な入出力方法

ここまでのプログラムは、read したものを一旦バッファに蓄えてから、必ずバッファ内のすべてを write して、また read するというシンプルなつくりでした。受信と送信が等速であるような理想的な環境であればこれでも構いませんが、接続相手や機器仕様によってはそう決め付けられないことも多く、その場合はこの手順は効率的とは言えません。

これを改善するためのアプローチとして、複数のプロセスを動作させて送受信を平行動作させる方法もありますが、今回のようなケースでは中間となるバッファの扱い方に工夫を凝らさなければならず、大げさとも言えます。

整理してみると、解決したい問題となるのは以下のような状況です。

1. 送信に時間がかかり待たされる状態なのに、次の受信データが来てしまっている。
2. 送信ができないので受信待ち状態に入ったら、送信の方が先にできるようになった。

どちらも待ち状態に入ってしまい、融通が利かなくなってしまうことが問題点です。であれば、送受信ができない状態になったら即座に中断し、先に可能になったものから優先的に処理するというアプローチであれば解決できそうです。select というシステムコールを使用して、このような実装が可能です。

```
#include <sys/types.h>
#include <sys/select.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
```

```
#include <termios.h>
#include <unistd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

#define SERIAL_BAUDRATE B9600

#define BUF_SIZE  256
#define READ_SIZE (BUF_SIZE / 2)

static int serial_fd = -1; /* シリアルポートファイルディスクリプタ */
static struct termios old_tio; /* 元のシリアルポート設定 */

static int terminated = 0; /* 終了シグナル発生フラグ */

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

#define __unused __attribute__((unused))

/**
 * 終了シグナルハンドラ
 * @param signum シグナル番号(不使用)
 */
static void terminate_sig_handler(__unused int signum)
{
        /* 終了シグナル発生を記録 */
        terminated = 1;
}

/**
 * シグナルハンドラ設定関数
 * @param sig_list ハンドラを設定するシグナルのリスト
 * @param num シグナルリストの要素数
 * @param handler 設定するハンドラ関数
 */
static void set_sig_handler(int sig_list[], ssize_t num, __sighandler_t handler)
{
        struct sigaction sa;
        int i;

        /* ハンドラ関数を設定 */
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = handler;

        /* 各シグナルに対して関連付け */
        for(i = 0; i < num; i++)
                if (sigaction(sig_list[i], &sa, NULL) < 0)
                        exitfail_errno("sigaction");
}

/**
 * シリアルポート設定復元関数
```

```
 */
static void restore_serial(void)
{
        int ret;

        /* シリアルポートの設定を元に戻す */
        ret = tcsetattr(serial_fd, TCSANOW, &old_tio);
        if (ret < 0)
                exitfail_errno("tcsetattr");
}

/**
 * シリアルポート設定関数
 * @param fd 設定するシリアルポートファイルディスクリプタ
 */
static void setup_serial(int fd)
{
        struct termios tio;
        int ret;

        /* 現在のシリアルポートの設定を退避する */
        ret = tcgetattr(fd, &old_tio);
        if (ret)
                exitfail_errno("tcgetattr");

        /* 終了時に設定を復元するための関数を登録 */
        if (atexit(restore_serial))
                exitfail_errno("atexit");

        /* 新しいシリアルポートの設定 */
        memset(&tio, 0, sizeof(tio));
        tio.c_iflag = IGNBRK | IGNPAR; /* ブレーク文字無視/パリティなし */
        tio.c_cflag = CS8 | CLOCAL | CREAD; /* フロー制御なし/8bit/非モデム/受信可 */
        tio.c_cc[VTIME] = 0; /* キャラクタ間タイマー無効 */
        tio.c_cc[VMIN] = 0; /* 送信/受信時にブロックしない */
        ret = cfsetspeed(&tio, SERIAL_BAUDRATE); /* 入出力ボーレート */
        if (ret < 0)
                exitfail_errno("cfsetspeed");

        /* バッファ内のデータをフラッシュ */
        ret = tcflush(fd, TCIFLUSH);
        if (ret < 0)
                exitfail_errno("tcflush");

        /* 新しいシリアルポート設定を適用 */
        ret = tcsetattr(fd, TCSANOW, &tio);
        if (ret)
                exitfail_errno("tcsetattr");
}

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数としてシリアルデバイス名を指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
```

```
int terminate_sig_list[] = { /* 終了シグナル種類 */
        SIGHUP, SIGINT, SIGQUIT, SIGPIPE, SIGTERM
};
fd_set fds_org, rdfds, wrfds, *prdfds, *pwrfds;
int nfds;
char buf[BUF_SIZE];
ssize_t ret, len, rdlen, wrlen;
int lf_flag = 0; /* LF 挿入処理発生フラグ */
int i;

exitfail_init();

/* 引数が指定されなかった場合、usage 表示して終了 */
if (argc < 2) {
        printf("Usage: %s <device>\n", BASENAME(argv[0]));
        return EXIT_SUCCESS;
}

/* シリアルポートを読み書き可能な非制御端末としてオープン */
serial_fd = open(argv[1], O_RDWR | O_NOCTTY);
if (serial_fd < 0)
        exitfail_errno("open");

/* 終了シグナルに対してハンドラを設定 */
set_sig_handler(terminate_sig_list, ARRAY_SIZE(terminate_sig_list),
                terminate_sig_handler);

/* シリアルポートを設定 */
setup_serial(serial_fd);

/* select のため、シリアルポートの設定された fd セットを作成しておく */
FD_ZERO(&fds_org);
FD_SET(serial_fd, &fds_org);
nfds = serial_fd + 1;

len = 0;
/* 終了シグナルが発生していない限りループ */
while (!terminated) {
        /* バッファに空きがある場合、読み込み可能を待つ */
        if (len < READ_SIZE) {
                rdfds = fds_org;
                prdfds = &rdfds;
        }
        else
                prdfds = NULL;
        /* バッファにデータがある場合、書き込み可能を待つ */
        if (len > 0) {
                wrfds = fds_org;
                pwrfds = &wrfds;
        }
        else
                pwrfds = NULL;
        /* 読み書きが可能になるまで待つ */
        ret = select(nfds, prdfds, pwrfds, NULL, NULL);
        if (ret < 0) {
                /* シグナル発生時はリトライ */
                if (errno == EINTR)
                        continue;
```

```
                exitfail_errno("select");
        }

        /* 書き込み可能になった */
        if (pwrfds && FD_ISSET(serial_fd, pwrfds)) {
                /* シリアルポートに書き込み */
                ret = write(serial_fd, buf, len);
                if (ret < 0) {
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("write");
                }
                wrlen = ret;

                /* 書き込んだ分を捨てて、残りデータを前にずらす */
                if (wrlen < len)
                        memmove(buf, buf + wrlen, len - wrlen);
                len -= wrlen;
        }

        /* 読み込み可能になった */
        if (prdfds && FD_ISSET(serial_fd, prdfds)) {
                /* シリアルポートから読み込み */
                ret = read(serial_fd, buf + len, READ_SIZE - len);
                if (ret < 0) {
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("read");
                }
                rdlen = ret;

                /* LF 挿入処理直後の LF の場合、1 文字無視させる */
                if (lf_flag) {
                        if (buf[len] == '\n')
                                memmove(buf + len, buf + len + 1,
                                        --rdlen);
                        lf_flag = 0;
                }
                /* データ最終文字が CR だった場合のために 1 文字潰しておく */
                buf[len + rdlen] = '\0';
                /* データを最後まで検査 */
                for (i = len; i < len + rdlen; i++) {
                        /* CR を発見 */
                        if (buf[i] == '\r')
                                /* 直後が LF ではない */
                                if (buf[++i] != '\n') {
                                        /* LF 挿入処理 */
                                        if (i < len + rdlen) {
                                                /* まだデータがあるなら
                                                   後ろにずらす */
                                                memmove(buf + i + 1,
                                                        buf + i,
                                                        len + rdlen - i);
                                        }
                                        else
                                                /* LF 挿入処理発生を保持 */
```

```
                                                lf_flag = 1;
                                            buf[i] = '\n';
                                            rdlen++;
                                        }
                                }
                                len += rdlen;
                        }
                }

                return EXIT_SUCCESS;
        }
```

図 6.30 改行コード変換を行うシリアルエコーサーバー(serial_echo_server3.c)

main 関数後半部分の while ループ内の構造が、やや大きく変わりました。まず、送受信の可能な条件を考えてみます。受信はバッファが一杯の時はできず、送信はバッファにデータがない時はできません。これを加味して、select 関数に適切な引数を渡します。

select 関数は、fd_set 型で指定されたデバイスに対して、送信・受信ができるようになるまで待ってくれます。プログラム内で、変数 prdfds としているもので受信側、変数 pwrfds としているもので送信側、それぞれに入っているデバイス群を監視します。

送受信どちら側(あるいは両方)が空いたかについては、FD_ISSET というマクロで渡した fd_set の変化をチェックすることで判定できます。なお、select 関数も read や write と同様、シグナルにより中断して-1 を返すことがある点についても注意してください。

実行結果は、見た目上は先ほど作ったものと変わりありません。しかしながら、こちらの方がより効率的であり、構造的にもわかりやすく見えるのではないかと思います。

6.6. ネットワークを使う

Linux などの UNIX 系 OS では、ネットワーク通信を行なうためにソケットという概念を用いた仕組みを使います。ソケットはファイルと同じように扱うことができるので、シリアルポートの時と同様に read や write といった関数でデータ送受信を行なうことができます。

6.6.1. TCP/IP

Ethernet 上で単純にネットワーク通信を行うと、送信したデータの紛失、化け、到達順番の入れ替わりなどが発生する可能性があります。TCP/IP はこれらの問題を吸収し、信頼性の高い通信を提供するためのプロトコルです。例えばデータが紛失した場合、データの再送を行うことで信頼性を確保します。TCP/IP は高い信頼性が必要なネットワーク通信で標準的に使われており、HTTP や FTP など多くのネットワーク通信プロトコルの基盤となっています。

TCP/IP での通信手順を模式化すると、次の図のようになります。サーバーとクライアントは、通信を行なう前に通信経路を確立する必要があります。

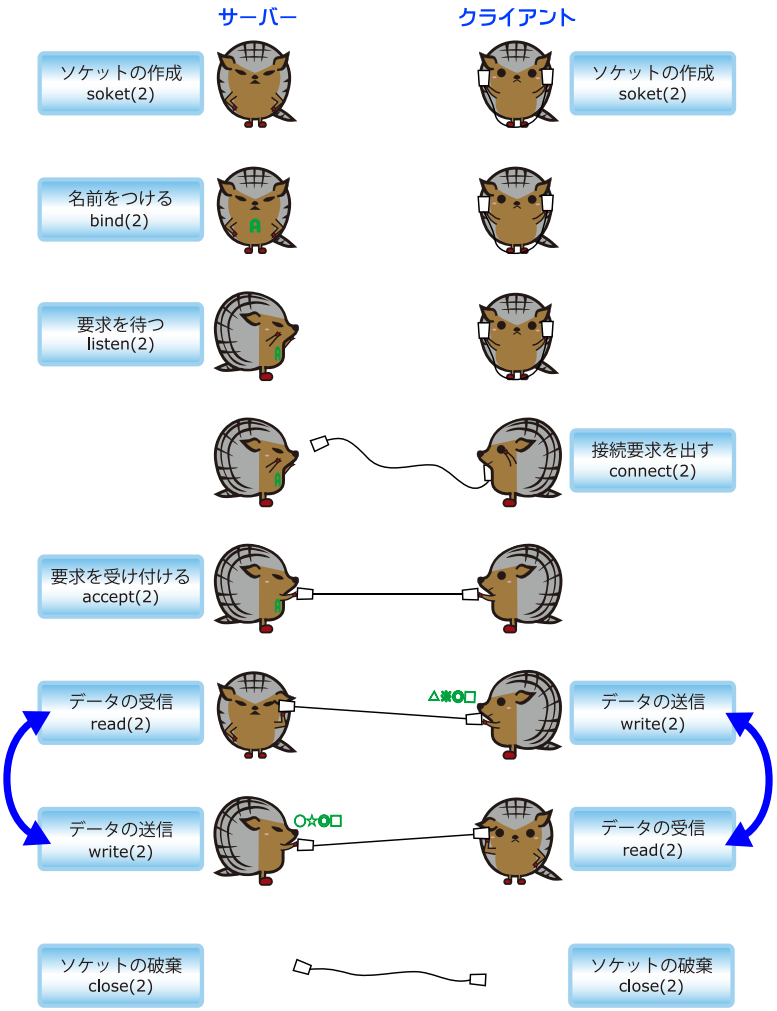


図 6.31 TCP/IP プログラムの基本的な流れ

通信経路を糸電話に例えると、次のようになります。

1. 糸電話で話をする人(ソケット)を作成する
2. サーバは、クライアントから糸電話を投げてもらう場所を指定する(名前を付ける)
3. サーバは、投げてもらうまで待つ(要求を待つ)
4. クライアントはサーバに糸電話を投げる(接続要求を出す)
5. サーバは糸電話を受け取る(要求を受け付ける)
6. 話をする(データの送信/受信)
7. 話が終わったら糸電話で話をした人は帰る(ソケットの破棄)

6.6.2. TCP/IP で Hello!

ソケットと TCP/IP を使って、簡単なサーバーアプリケーションを作ってみます。まずはこんな仕様にしてみます。

1. サーバーが接続を待つポートは、コマンドライン引数で指定する。
2. クライアントからは telnet アプリケーションで接続でき、接続すると Hello!と表示される。

こんなプログラムになります。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として接続待ちポートを指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        in_port_t listen_port;
        static int listen_fd, accept_fd; /* ソケットファイルディスクリプタ */
        struct sockaddr_in server_addr, client_addr;
        socklen_t addr_len;
        char message[] = "Hello!\r\n";
        ssize_t ret;

        exitfail_init();

        /* 引数が指定されなかった場合、usage 表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <port>\n", BASENAME(argv[0]));
                return EXIT_SUCCESS;
        }

        ret = strtoul(argv[1], NULL, 10);
        if (ret == LONG_MIN || ret == LONG_MAX)
                exitfail_errno("strtol");
        if (ret < 49152 || 65535 < ret)
                exitfail("Specify the port 49152-65535\n");
        listen_port = ret;

        /* 接続待ち用のソケットを作成 */
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (listen_fd < 0)
                exitfail_errno("open");

        /* アドレスとポートを割り当て */
```

```
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET; /* IPv4 インターネットプロトコル */
        server_addr.sin_addr.s_addr = INADDR_ANY; /* 任意のアドレス */
        server_addr.sin_port = htons(listen_port); /* 接続を待つポート */
        addr_len = sizeof(server_addr);
        ret = bind(listen_fd, (struct sockaddr *)&server_addr, addr_len);
        if (ret < 0)
                exitfail_errno("bind");

        /* クライアントからの接続を待つ */
        ret = listen(listen_fd, SOMAXCONN);
        if (ret < 0)
                exitfail_errno("listen");

        /* クライアントからの接続を受け付けて、入出力用のソケットを作成 */
        accept_fd = accept(listen_fd, (struct sockaddr *)&client_addr,
                           &addr_len);
        if (accept_fd < 0)
                exitfail_errno("accept");

        /* これ以上の接続を受け付けないため、接続待ち用ソケットを破棄 */
        close(listen_fd);

        /* メッセージを送信 */
        ret = write(accept_fd, message, strlen(message));
        if (ret < 0)
                exitfail_errno("write");

        /* 入出力用のソケットを破棄 */
        close(accept_fd);

        return EXIT_SUCCESS;
}
```

図 6.32 ネットワークで Hello!を返すサーバー (network_hello_server.c)

ソケットは 2 つ作られます。1 つ目は、クライアントからの接続を待つためのソケットです。こちらは socket 関数で作成し、sockaddr_in 型の変数 server_addr に接続待ちするポートを入れて bind、それから listen 関数で接続を待つ動作に入ります。

クライアントから接続があると、accept 関数で新たに入出力用のソケットが作られます。プログラムの作り方次第では複数のクライアントを待つこともできるのですが、このサーバーは 1 接続のみとしますので、ここで接続待ち用のソケットは破棄してしまっています。

入出力用のソケットができてしまえば、後はシリアルポートの時と同じように read/write できます。ここでは Hello!メッセージだけ表示して、ソケットを閉じてプログラムを終了しています。

Armadillo 上でこのネットワークサーバーを動作させます。この例では、ポートは 65432 を指定してみました。

```
[armadillo ~]$ ./network_hello_server 65432
```

図 6.33 network_hello_server の実行結果

PC から telnet で接続してみます。ATDE 上からでも、Windows の DOS プロンプトなどからでも構いません。接続コマンドは同じです。telnet のパラメータとして、サーバー(Armadillo)の IP アドレス(ここでは例として、192.168.1.100 であるとします)と接続ポートを指定します。

```
[ATDE ~]$ telnet 192.168.1.100 65432
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
Hello!
Connection closed by foreign host.
```

図 6.34 network_hello_server への telnet

このように Hello!と表示されてから、すぐに切断されます。

ソケットの再利用

同じポートの指定で network_hello_server を何度も実行すると、エラーメッセージが表示されて起動できないことがあります。

```
[armadillo ~]$ ./network_hello_server 65432
./network_hello_server: bind: Address already in use
```

bind しようとしたアドレスが使用中です、というエラーメッセージです。サーバーの終了時に、ソケットはクローズしているのに…これは、TCP/IP を使用していることに起因しています。

TCP/IP は送信データの到達を保証するものであるため、データが紛失した際は再送しなければなりません。このプログラムのように、送信後すぐにソケットをクローズしてしまった場合であっても、その後に再送する可能性があるため、システムはしばらく(2 ～ 4 分程度)ソケットを破棄しないまま保持し続けます。このようなソケットを、TIME_WAIT 状態であるといいます。なお、(接続しに行った側である)クライアントが先にソケットをクローズした場合は、TIME_WAIT 状態にはなりません。

6.6.3. ネットワークエコーサーバー

Hello!を改造して、シリアルポート送受信で作成したエコーサーバーのネットワーク版を作成してみます。

```
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#include <limits.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

#define BUF_SIZE  256

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として接続待ちポートを指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        in_port_t listen_port;
        static int listen_fd, accept_fd; /* ソケットファイルディスクリプタ */
        struct sockaddr_in server_addr, client_addr;
        socklen_t addr_len;
        char message[] = "Hello!\r\n";
        fd_set fds_org, rdfds, wrfds, *prdfds, *pwrfds;
        int nfds;
        char buf[BUF_SIZE];
        ssize_t ret, len, wrlen;

        exitfail_init();

        /* 引数が指定されなかった場合、usage 表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <port>\n", BASENAME(argv[0]));
                return EXIT_SUCCESS;
        }

        ret = strtoul(argv[1], NULL, 10);
        if (ret == LONG_MIN || ret == LONG_MAX)
                exitfail_errno("strtol");
        if (ret < 49152 || 65535 < ret)
                exitfail("Specify the port 49152-65535\n");
        listen_port = ret;

        /* 接続待ち用のソケットを作成 */
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (listen_fd < 0)
                exitfail_errno("open");

        /* アドレスとポートを割り当て */
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET; /* IPv4 インターネットプロトコル */
        server_addr.sin_addr.s_addr = INADDR_ANY; /* 任意のアドレス */
        server_addr.sin_port = htons(listen_port); /* 接続を待つポート */
        addr_len = sizeof(server_addr);
        ret = bind(listen_fd, (struct sockaddr *)&server_addr, addr_len);
        if (ret < 0)
                exitfail_errno("bind");
```

```
/* クライアントからの接続を待つ */
ret = listen(listen_fd, SOMAXCONN);
if (ret < 0)
        exitfail_errno("listen");

/* クライアントからの接続を受け付けて、入出力用のソケットを作成 */
accept_fd = accept(listen_fd, (struct sockaddr *)&client_addr,
                   &addr_len);
if (accept_fd < 0)
        exitfail_errno("accept");

/* これ以上の接続を受け付けないため、接続待ち用ソケットを破棄 */
close(listen_fd);

/* メッセージを送信 */
ret = write(accept_fd, message, strlen(message));
if (ret < 0)
        exitfail_errno("write");

/* select のため、ソケットの設定された fd セットを作成しておく */
FD_ZERO(&fds_org);
FD_SET(accept_fd, &fds_org);
nfds = accept_fd + 1;

len = 0;
/* 無限ループ */
for (; ; ) {
        /* バッファに空きがある場合、読み込み可能を待つ */
        if (len < BUF_SIZE) {
                rdfds = fds_org;
                prdfds = &rdfds;
        }
        else
                prdfds = NULL;
        /* バッファにデータがある場合、書き込み可能を待つ */
        if (len > 0) {
                wrfds = fds_org;
                pwrfds = &wrfds;
        }
        else
                pwrfds = NULL;
        /* 読み書きが可能になるまで待つ */
        ret = select(nfds, prdfds, pwrfds, NULL, NULL);
        if (ret < 0) {
                /* シグナル発生時はリトライ */
                if (errno == EINTR)
                        continue;
                exitfail_errno("select");
        }

        /* 書き込み可能になった */
        if (pwrfds && FD_ISSET(accept_fd, pwrfds)) {
                /* ソケットに書き込み */
                ret = write(accept_fd, buf, len);
                if (ret < 0) {
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
```

```
                        exitfail_errno("write");
                }
                wrlen = ret;

                /* 書き込んだ分を捨てて、残りデータを前にずらす */
                if (wrlen < len)
                        memmove(buf, buf + wrlen, len - wrlen);
                len -= wrlen;
        }

        /* 読み込み可能になった */
        if (prdfds && FD_ISSET(accept_fd, prdfds)) {
                /* ソケットから読み込み */
                ret = read(accept_fd, buf + len, BUF_SIZE - len);
                if (ret <= 0) {
                        if (ret == 0)
                                break; /* 終了 */
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("read");
                }
                len += ret;
        }
    }

    /* 入出力用のソケットを破棄 */
    close(accept_fd);

    return EXIT_SUCCESS;
}
```

図 6.35 ネットワークエコーサーバー (network_echo_server1.c)

ソースに追加された select, read, write を使う入出力部の基本構造は、シリアルポートの時とまったく一緒です。

先ほどと同じようにサーバーを実行してみます。

```
[armadillo ~]$ ./network_echo_server1 65432
```

図 6.36 network_echo_server1 の実行結果

PC から telnet で接続してみます。

```
[ATDE ~]$ telnet 192.168.1.100 65432
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
Hello!
abc
abc
defg
defg
^]

telnet> quit
Connection closed.
```

図 6.37 network_echo_server1 への telnet

先ほどと違い、Hello!が表示された後にソケットがクローズされません。その後はエコーサーバーですので、入力したものがそのまま返ってきます。

telnet 実行直後に表示されているように、Ctrl キーを押しながら]キーを押すとエスケープすることができます。「telnet> 」というプロンプトが表示されるので、quit と入力して **telnet** コマンドを終了することができます[15]。

このように動作するのは ATDE から telnet した場合です。実は、他の telnet アプリケーションから接続した場合、それぞれちょっとずつ動作が違ってきてしまいます。

例えば Windows の DOS プロンプトから **telnet** コマンドで接続した時は、以下のようになりました[16]。

```
C:\> telnet 192.168.1.100 65432
Microsoft Telnet クライアントへようこそ

エスケープ文字は 'Ctrl+]' です

Hello!
aabbcc

ddeeffgg

^]

Microsoft Telnet> quit
```

図 6.38 network_echo_server1 への telnet(Windows)

このように、1 文字打つたびに文字が返ってきています。

これは、単純に telnet クライアントアプリケーションの挙動が違うだけです。ATDE (つまり Linux) の telnet クライアントは(改行が入力されるたびに)行単位でデータを送信し、Windows の telnet クライアントは 1 文字入力するたびにデータを送信しているということになります。

[15]この場合は接続したクライアント側がソケットをクローズすることになるため、TIME_WAIT 状態にはなりません。
[16]Windows XP の DOS プロンプトから **telnet** コマンドを使用して確認しました。

また、Tera Term にも telnet 機能があります。こちらの場合は(標準の設定では)打ち込んだものがそのまま表示はされません。これは設定の問題なので良いのですが、シリアルエコーサーバーの時と同じように、CR や LF の改行コードの違いによる問題も発生します。

Windows の telnet や Tera Term 相手の時も、Linux の telnet の時と同じように表示するようにサーバーアプリケーションを改造してみます。

```
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAIN_C
#include "exitfail.h"

#define BUF_SIZE  256
#define READ_SIZE (BUF_SIZE / 2)

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

/**
 * main 関数
 * @param argc 引数なしの場合は usage 表示のみ
 * @param argv 第 1 引数として接続待ちポートを指定
 * @return exit 値
 */
int main(int argc, char *argv[])
{
        in_port_t listen_port;
        static int listen_fd, accept_fd; /* ソケットファイルディスクリプタ */
        struct sockaddr_in server_addr, client_addr;
        socklen_t addr_len;
        char message[] = "Hello!\r\n";
        fd_set fds_org, rdfds, wrfds, *prdfds, *pwrfds;
        int nfds;
        char buf[BUF_SIZE];
        ssize_t ret, len, rdlen, wrlen;
        int lf_flag = 0; /* LF 挿入処理発生フラグ */
        int i;

        exitfail_init();

        /* 引数が指定されなかった場合、usage 表示して終了 */
        if (argc < 2) {
                printf("Usage: %s <port>\n", BASENAME(argv[0]));
                return EXIT_SUCCESS;
        }

        ret = strtoul(argv[1], NULL, 10);
        if (ret == LONG_MIN || ret == LONG_MAX)
                exitfail_errno("strtol");
```

```
        if (ret < 49152 || 65535 < ret)
                exitfail("Specify the port 49152-65535\n");
        listen_port = ret;

        /* 接続待ち用のソケットを作成 */
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (listen_fd < 0)
                exitfail_errno("open");

        /* アドレスとポートを割り当て */
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET; /* IPv4 インターネットプロトコル */
        server_addr.sin_addr.s_addr = INADDR_ANY; /* 任意のアドレス */
        server_addr.sin_port = htons(listen_port); /* 接続を待つポート */
        addr_len = sizeof(server_addr);
        ret = bind(listen_fd, (struct sockaddr *)&server_addr, addr_len);
        if (ret < 0)
                exitfail_errno("bind");

        /* クライアントからの接続を待つ */
        ret = listen(listen_fd, SOMAXCONN);
        if (ret < 0)
                exitfail_errno("listen");

        /* クライアントからの接続を受け付けて、入出力用のソケットを作成 */
        accept_fd = accept(listen_fd, (struct sockaddr *)&client_addr,
                           &addr_len);
        if (accept_fd < 0)
                exitfail_errno("accept");

        /* 接続待ち用のソケットは不要になったので破棄 */
        close(listen_fd);

        /* メッセージを送信 */
        ret = write(accept_fd, message, strlen(message));
        if (ret < 0)
                exitfail_errno("write");

        /* select のため、ソケットの設定された fd セットを作成しておく */
        FD_ZERO(&fds_org);
        FD_SET(accept_fd, &fds_org);
        nfds = accept_fd + 1;

        len = 0;
        /* 無限ループ */
        for (; ; ) {
                /* バッファに空きがある場合、読み込み可能を待つ */
                if (len < READ_SIZE) {
                        rdfds = fds_org;
                        prdfds = &rdfds;
                }
                else
                        prdfds = NULL;
                /* バッファにデータがあり改行が含まれていた場合、書き込み可能を待つ */
                if (len > 0 && memchr(buf, '\n', len)) {
                        wrfds = fds_org;
                        pwrfds = &wrfds;
                }
```

```
        else
                pwrfds = NULL;
        /* 読み書きが可能になるまで待つ */
        ret = select(nfds, prdfds, pwrfds, NULL, NULL);
        if (ret < 0) {
                /* シグナル発生時はリトライ */
                if (errno == EINTR)
                        continue;
                exitfail_errno("select");
        }

        /* 書き込み可能になった */
        if (pwrfds && FD_ISSET(accept_fd, pwrfds)) {
                /* ソケットに書き込み */
                ret = write(accept_fd, buf, len);
                if (ret < 0) {
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("write");
                }
                wrlen = ret;

                /* 書き込んだ分を捨てて、残りデータを前にずらす */
                if (wrlen < len)
                        memmove(buf, buf + wrlen, len - wrlen);
                len -= wrlen;
        }

        /* 読み込み可能になった */
        if (prdfds && FD_ISSET(accept_fd, prdfds)) {
                /* ソケットから読み込み */
                ret = read(accept_fd, buf + len, BUF_SIZE - len);
                if (ret <= 0) {
                        if (ret == 0)
                                break; /* 終了 */
                        /* シグナル発生時はリトライ */
                        if (errno == EINTR)
                                continue;
                        exitfail_errno("read");
                }
                rdlen = ret;

                /* LF 挿入処理直後の LF の場合、1 文字無視させる */
                if (lf_flag) {
                        if (buf[len] == '\n')
                                memmove(buf + len, buf + len + 1,
                                        --rdlen);
                        lf_flag = 0;
                }
                /* データ最終文字が CR だった場合のために 1 文字潰しておく */
                buf[len + rdlen] = '\0';
                /* データを最後まで検査 */
                for (i = len; i < len + rdlen; i++) {
                        /* CR を発見 */
                        if (buf[i] == '\r')
                                /* 直後が LF ではない */
                                if (buf[++i] != '\n') {
```

```
                                        /* LF 挿入処理 */
                                        if (i < len + rdlen) {
                                                /* まだデータがあるなら
                                                   後ろにずらす */
                                                memmove(buf + i + 1,
                                                        buf + i,
                                                        len + rdlen - i);
                                        }
                                        else
                                                /* LF 挿入処理発生を保持 */
                                                lf_flag = 1;
                                        buf[i] = '\n';
                                        rdlen++;
                                }
                        }
                        len += rdlen;
                }
        }

        /* 入出力用のソケットを破棄 */
        close(accept_fd);

        return EXIT_SUCCESS;
}
```

図 6.39 ネットワークエコーサーバー (network_echo_server2.c)

Windows 用 telnet 対応のための変更点は 1 点だけ。write 可能を待つかどうか判定する際に、memchr 関数によるデータ内容の確認を行っています。改行文字が来るまでは、write されないようにしているわけです。

そして、Tera Term の telnet での改行文字問題の修正ですが、これはシリアルエコーサーバーでの「図 6.30. 改行コード変換を行うシリアルエコーサーバー(serial_echo_server3.c)」の対策とまったく一緒です。

同じ telnet アプリケーションとはいえ、このように(見た目の)動作に差が出ることもあります。より汎用的で完成度の高いものを作ろうとすれば多くの試験が必要で、対策コストも決して小さくないということは心に留めておくべきでしょう。

7. 詳解 Atmark Dist

Atmark Dist は、uCLinux-dist を元に Atmark Techno 製品用に独自に改良したソースコードベースの開発ディストリビューションです。

uCLinux-dist

uClinux-dist は uClinux.org が配布する、ソースコードベースの開発ディストリビューションです。

uClinux とは、MMU を持たないマイクロコンピュータでも動作するように作成された Linux です。uClinux の成果は 2.6 系の Linux カーネルに取り込まれています。

uClinux-dist は当初 uClinux 用に作成されましたが、現在の uClinux-dist は Clinux 専用というわけではありません。設定時に既存の Linux を選択することで i386 や ARM、PowerPC のような MMU をもった CPU にも対応しています。

Atmark Dist を使うと、Linux カーネルとユーザーランドのルートファイルシステムを統合して開発することができます。コンフィギュレーションを変更することで、ルートファイルシステムに含むアプリケーションプログラムやライブラリ及び Linux カーネルの機能を選択できます。また、コマンド一つでそれらをビルドし、フラッシュメモリに書き込める形式のイメージファイルを作成できます。さらに、ユーザー独自のアプリケーションプログラムを追加したり、コンフィギュレーションを保存しておくこともできるので、Armadillo を使った組み込みシステムの構築を容易にします。

Atmark Dist を使ってイメージをビルドする基本的な手順は第 1 部の「開発の基本的な流れ」で述べました。本章では、Atmark Dist の詳しい動作や、カスタマイズの方法について説明します。

7.1. ディレクトリ構成

この章では、Atmark Dist のディレクトリ構成について説明します。各ディレクトリの概要を説明するとともに、各ディレクトリに関連した説明がなされている章を紹介します。ディレクトリ構成を理解することは、Atmark Dist を用いて開発を行うにあたり、重要なポイントとなります。

Atmark Dist のトップディレクトリは、以下のようになっています。

```
[ATDE ~]$ cd atmark-dist
[ATDE ~/atmark-dist]$ tree -L 1 -F
.
|-- COPYING
|-- Documentation/
|-- Makefile
|-- README
|-- SOURCE
|-- bin/
|-- config/
|-- freeswan/
|-- glibc/
|-- include/
|-- lib/
|-- linux-2.6.x -> ../linux-バージョン/
|-- tools/
|-- uClibc/
|-- user/
|-- vendors/
`-- version

      12 directories, 5 files
```

図 7.1 Atmrk Dist のトップディレクトリ

7.1.1. トップレベル makefile

Atmark Dist は、**make** コマンドによってすべての作業を行います。各ディレクトリに makefile があり、ソースコードのビルドなど、指定されたターゲットに対応する処理が記述されています。

atmark-dist/Makefile をトップレベル makefile と呼び、他のディレクトリにある makefile とは区別することにします。トップレベル makefile は、atmark-dist のビルドをコントロールする大事なファイルです。ここで定義されているターゲットは第 6 章の章で詳しく説明します。

また、**make** コマンドについては、「6.1.3. make と makefile」で説明しました。適宜参照してください。

7.1.2. config ディレクトリ

config ディレクトリには設定に必要なスクリプトや makefile が収録されています。config ディレクトリ内のいくつかのファイルについては、「7.6. プロダクトディレクトリのカスタマイズ」で説明します。

7.1.3. tools ディレクトリ

tools ディレクトリには、ビルドに必要ないくつかのツールが収録されています。このディレクトリに収録されている romfs-inst.sh はプロダクトディレクトリの makefile でよく使います。「7.6. プロダクトディレクトリのカスタマイズ」で、詳しく説明します。

7.1.4. C ライブラリディレクトリ

glibc(GNU C library)と uClibc ディレクトリには、Atmark Dist で選択可能な C ライブラリのソースコードが収録されています。通常は、開発環境 ATDE にインストールされたビルド済みの C ライブラリを使用しますので、これらのディレクトリにあるソースコードは使用されません。

7.1.5. user ディレクトリ

このディレクトリにはユーザーランドアプリケーションプログラムのソースコードが収録されています。多くのアプリケーションは、GNU/Linux 用のアプリケーションを Atmark Dist でも使用できるように変更したものですが、Atmark Dist 専用に開発されたものも含まれます。

7.1.6. vendors ディレクトリ

vendors ディレクトリは以下のようになっています。

```
[ATDE ~/atmark-dist]$ tree -L 1 -F vendors/
vendors/
|-- 3com/
|-- ADI/
|-- Akizuki/
|-- Apple/
|-- Arcturus/
|-- Arnewsh/
|-- AtmarkTechno/
|-- Atmel/
(中略)
|-- Xilinx/
|-- config/
`-- senTec/
```

図 7.2 vendors ディレクトリ

vendors ディレクトリの中には、ベンダー名のディレクトリがたくさん入っています。AtmarkTechno ディレクトリもそのうちのひとつです。

ベンダー名のディレクトリの中には、各製品用のサブディレクトリが複数入っています。この製品用のディレクトリをプロダクトディレクトリと呼びます。プロダクトディレクトリには、個々の製品用のイメージをビルドするためのさまざまなファイルが入っています。プロダクトディレクトリについては、「7.6. プロダクトディレクトリのカスタマイズ」を参照してください。

vendors/AtmarkTechno ディレクトリ内には Armadillo をはじめとした Atmark Dist に対応した Atmark Techno 社製品用のプロダクトディレクトリがあります。

```
[ATDE ~/atmark-dist]$ tree -L 1 -F vendors/AtmarkTechno/
vendors/AtmarkTechno/
|-- Armadillo-210.Base/
|-- Armadillo-210.Common/
|-- Armadillo-210.Recover/
|-- Armadillo-220.Base/
|-- Armadillo-220.Recover/
|-- Armadillo-230.Base/
|-- Armadillo-230.Recover/
|-- Armadillo-240.Base/
|-- Armadillo-240.Recover/
|-- Armadillo-2x0.Common/
|-- Armadillo-300/
|-- Armadillo-420/
|-- Armadillo-440/
|-- Armadillo-4x0.Common/
|-- Armadillo-500/
|-- Armadillo-500-FX.base/
|-- Armadillo-500-FX.dev/
|-- Armadillo-9/
|-- Armadillo-9.Common/
|-- Armadillo-9.PCMCIA/
|-- Common/
|-- SUZAKU-V.Common/
|-- SUZAKU-V.SZ310/
|-- SUZAKU-V.SZ310-SID/
|-- SUZAKU-V.SZ310-SIL/
|-- SUZAKU-V.SZ310-SIL-GPIO/
|-- SUZAKU-V.SZ410/
|-- SUZAKU-V.SZ410-SID/
|-- SUZAKU-V.SZ410-SIL/
|-- SUZAKU-V.SZ410-SIL-GPIO/
`-- SUZAKU-V.SZ410-SIV/
```

図 7.3 vendors/AtmarkTechno ディレクトリ

vendors ディレクトリには、ベンダー名のディレクトリの他に config ディレクトリもあります。アーキテクチャごとのデフォルト設定が config.arch という名前でそれぞれのディレクトリ内に保存されています。この config.arch はプロダクトディレクトリの config.arch から参照されています。詳しくは項 8.1.「config.arch」を参照してください。

```
[ATDE ~/atmark-dist]$ tree -L 2 -F vendors/config/
vendors/config/
|-- arm/
|   `-- config.arch
|-- arm-vfp/
|   `-- config.arch
|-- armel/
|   `-- config.arch
|-- armnommu/
|   `-- config.arch
|-- common/
|   `-- config.arch
|-- config.languages
|-- h8300/
|   `-- config.arch
|-- host/
|   `-- config.arch
|-- i386/
|   `-- config.arch
|-- i960/
|   `-- config.arch
|-- m68knommu/
|   `-- config.arch
|-- microblaze/
|   `-- config.arch
|-- mips/
|   `-- config.arch
|-- powerpc/
|   `-- config.arch
`-- sh/
    `-- config.arch
```

図 7.4 vendors/config ディレクトリ

7.2. 基本ターゲット

Atmark Dist でよくつかうターゲットをここで紹介します。

7.2.1. コンフィギュレーション設定用のターゲット

コンフィギュレーションを設定するためのターゲットには、テキストベースとメニューベースの二つ[1]が用意されています。表示方法が異なるだけで、両方共 Atmark Dist のコンフィギュレーションを設定するためのターゲットです。

7.2.1.1. テキストベースでのコンフィギュレーション設定(config)

config ターゲットを指定すると、テキストベースでコンフィギュレーションを設定できます。標準イメージを作成するような簡単なコンフィギュレーションを選択するときに便利です。

第 1 部の「Atmark Dist を使ったルートファイルシステムの作成」「ソースコードの取得」で説明したように、Atmark Dist と Linux カーネルのソースコードを取得して、それぞれシンボリックリンクを作成し、atmark-dist ディレクトリで **make config** を実行すると、以下のように表示されます。

[1]GUI ベースのコンフィギュレーションもありますが、あまり使いませんのでここでは紹介しません。

まず、ターゲットとなるボードのベンダー名を質問されます。ベンダー名は、vendors ディレクトリ以下にあるディレクトリ名で指定します。「AtmarkTechno」と入力してください。

```
[ATDE ~/atmark-dist]$ make config
config/mkconfig > config.in
#
# No defaults found
#
*
* Vendor/Product Selection
*
*
* Select the Vendor you wish to target
*
Vendor (3com, ADI, Akizuki, Apple, Arcturus, Arnewsh, AtmarkTechno, Atmel, Avnet, Cirrus, Cogent,
Conexant, Cwlinux, CyberGuard, Cytek, Exys, Feith, Future, GDB, Hitachi, Imt, Insight, Intel, Kend
inMicrel, LEOX, Mecel, Midas, Motorola, NEC, NetSilicon, Netburner, Nintendo, OPENcores, Promise,
SNEHA, SSV, SWARM, Samsung, SecureEdge, Signal, SnapGear, Soekris, Sony, StrawberryLinux, TI, Tele
IP, Triscend, Via, Weiss, Xilinx, senTec) [SnapGear] (NEW) AtmarkTechno
```

図 7.5 make config: ベンダーの選択

次に、ボード名を質問されます。選択したベンダーのディレクトリにあるプロダクトディレクトリの名前で指定します。例として、Armadillo-440 を入力します。独自のプロダクトディレクトリを追加している場合は、そのディレクトリ名を指定してください。

```
*
* Select the Product you wish to target
*
AtmarkTechno Products (Armadillo-210.Base, Armadillo-210.Recover, Armadillo-220.Base, Armadillo-22
0.Recover, Armadillo-230.Base, Armadillo-230.Recover, Armadillo-240.Base, Armadillo-240.Recover, A
rmadillo-300, Armadillo-420, Armadillo-440, Armadillo-500, Armadillo-500-FX.base, Armadillo-500-FX
.dev, Armadillo-9, Armadillo-9.PCMCIA, SUZAKU-V.SZ310, SUZAKU-V.SZ310-SID, SUZAKU-V.SZ310-SIL, SUZ
AKU-V.SZ310-SIL-GPIO, SUZAKU-V.SZ410, SUZAKU-V.SZ410-SID, SUZAKU-V.SZ410-SIL, SUZAKU-V.SZ410-SIL-G
PIO, SUZAKU-V.SZ410-SIV) [Armadillo-210.Base] (NEW) Armadillo-440
```

図 7.6 make config: プロダクトの選択

続いて、ターゲットのアーキテクチャを指定します。通常は、「default」のままにします。Enter キーを入力してください。

```
*
* Kernel/Library/Defaults Selection
*
*
* Kernel is linux-2.6.x
*
Cross-dev (default, arm-vfp, arm, armel, armnommu, common, h8300, host, i386, i960, m68knommu, mic
roblaze, mips, powerpc, sh) [default] (NEW) ⏎
```

図 7.7 make config: アーキテクチャの選択

使用する C ライブラリを選択します。 通常は「None」のままにしてください。「None」を選択すると、開発環境にインストールされているビルド済みの C ライブラリを使用します。Enter キーを入力してください。

```
Libc Version (None, glibc, uC-libc, uClibc) [None] (NEW) ⏎
```

図 7.8 make config: C ライブラリの選択

これまで選択してきたコンフィギュレーションの設定を標準にするかどうか質問されます。y(Yes)と入力してください。

```
Default all settings (lose changes) (CONFIG_DEFAULTS_OVERRIDE) [N/y/?] (NEW) y
```

図 7.9 make config: コンフィギュレーションの設定を標準にする

標準のイメージと同じイメージを作成するには、最後の 3 つの質問は n(No)と答えてください。「Customize Kernel Settings」に y(Yes)と答えると、Linux カーネルのコンフィギュレーションを変更することができます。また、「Customize Vendor/User Settings」に y(Yes)と答えると、ルートファイルシステムに含めるアプリケーションプログラムやライブラリを選択することができます。

```
Customize Kernel Settings (CONFIG_DEFAULTS_KERNEL) [N/y/?] n
Customize Vendor/User Settings (CONFIG_DEFAULTS_VENDOR) [N/y/?] n
Update Default Vendor Settings (CONFIG_DEFAULTS_VENDOR_UPDATE) [N/y/?] n
```

図 7.10 make config: コンフィギュレーション設定の終了

すべての質問に答え終わると、コンフィギュレーションが反映されます。

7.2.1.2. メニューベースでのコンフィギュレーション設定(menuconfig)

menuconfig ターゲットを指定すると、メニュー画面を使ってコンフィギュレーションを設定することができます。

メニュー画面の描画

menuconfig ターゲットを指定した場合のメニュー画面は Ncurses(new curses、テキストベースのユーザーインターフェースを作成するためのライブラリ)を使って描画されます。このターゲットが指定されたときに画面などをコントロールするプログラムをビルドするため、Ncurses のライブラリとヘッダファイルが必要になります。ATDE では、Ncurses 用のパッケージがあらかじめインストールされているので必要ありませんが、自前で環境を用意した場合はインストールしてください。

make menucnfig を実行すると、「図 7.12. menuconfig: Main Menu 画面」のような画面が表示されます。メニュー画面によるコンフィギュレーションの設定方法は、「7.4. コンフィギュレーションの設定」で詳しく説明します。

```
[ATDE ~/atmark-dist]$ make menuconfig
```

図 7.11 メニュー画面によるコンフィギュレーション設定の開始

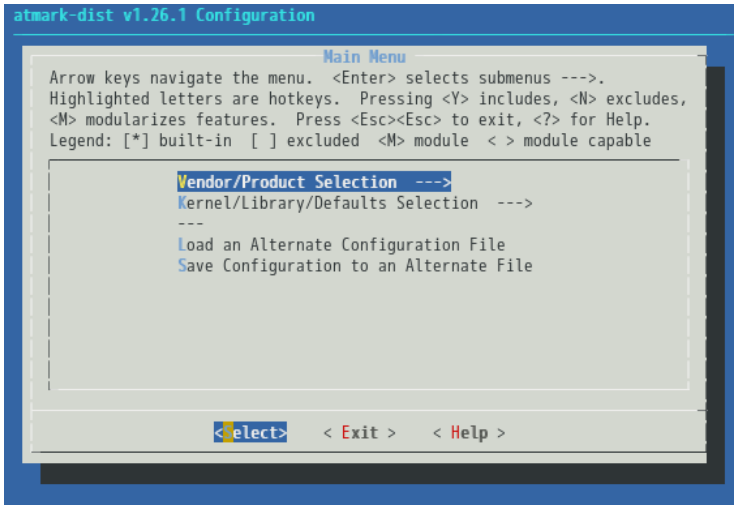


図 7.12 menuconfig: Main Menu 画面

7.2.2. クリーンターゲット(clean)

clean ターゲットは Atmark Dist 内で作成された中間ファイル等を削除し、きれいにするためのターゲットです。Atmark Dist には以下の 2 種類の clean ターゲットが用意されています。

表 7.1 Atmark Dist の clean ターゲット

ターゲット	内容
make clean	サブディレクトリをクリーンし、romfs ディレクトリおよび、images ディレクトリ内のファイルを削除します。
make distclean	すべての中間ファイルなどを削除して、Atmark Dist を初期状態にします。現在のコンフィギュレーションも削除されます。

7.2.3. デフォルトターゲット(all)

all ターゲットは、atmark-dist のデフォルトターゲットです。**make** コマンドをオプションなしで実行することで、このターゲットが実行されます。all ターゲットでは、必要なソースコード(カーネル、ユーザーランドアプリケーション、ライブラリ)のビルドを行い、Armadillo に書き込むことができるイメージファイルを生成します。詳しくは「7.5. ソースコードのビルドとイメージファイルの作成」で説明します。

7.3. イメージファイル作成手順の全体像

Atmark Dist でイメージファイルを作成する際の全体的な流れは、以下に示す手順になります。

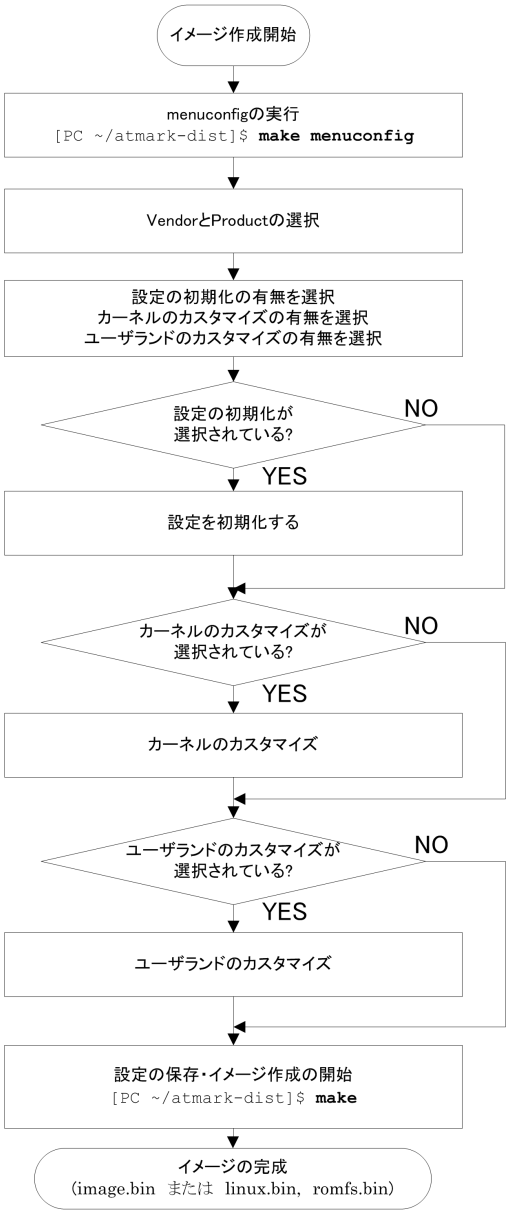


図 7.13 Atmark Dist でイメージファイルを作成する流れ

まず、atamrk-dist ディレクトリで **make menuconfig** を実行することで、コンフィギュレーションの設定を開始します。コンフィギュレーションの設定画面では、まず、ベンダーとプロダクトを選択します。続いて、設定初期化を行うか、カーネル又はユーザーランドのルートファイルシステムのカスタマイズを行うかを選択します。カスタマイズを行うことを選択した場合、それぞれのコンフィギュレーションを設定します。コンフィギュレーション設定画面を終了すると、設定が反映されます。

コンフィギュレーションを正しく設定したあと **make** コマンドを実行すると、イメージファイルの作成を開始します。コンフィギュレーションに従って、必要なソースコードのビルド、ルートファイルシステムの作成、イメージファイルの作成を自動でおこないます。

以降の章では、それぞれの手順を詳しく説明していきます。

7.4. コンフィギュレーションの設定

Atmark Dist を使ってイメージファイルを作成する手順は、大きく以下の二つに分けられます。

1. コンフィギュレーションの設定
2. ソースコードのビルドとイメージファイルの作成

ここではその前半部分、コンフィギュレーションの設定方法について説明します。説明は、make menuconfig を使ったメニューベースのコンフィギュレーション設定画面を用いて行います。設定画面では、「表 7.2. menuconfig の操作方法」に示すキー操作で画面の操作を行います。

表 7.2 menuconfig の操作方法

キー操作	動作
↑ ↓キー	メニューの選択
←→キー	動作の選択
スペースキー	オプションの選択
Enter キー	動作の決定

設定画面は、以下に示す階層構造になっています。

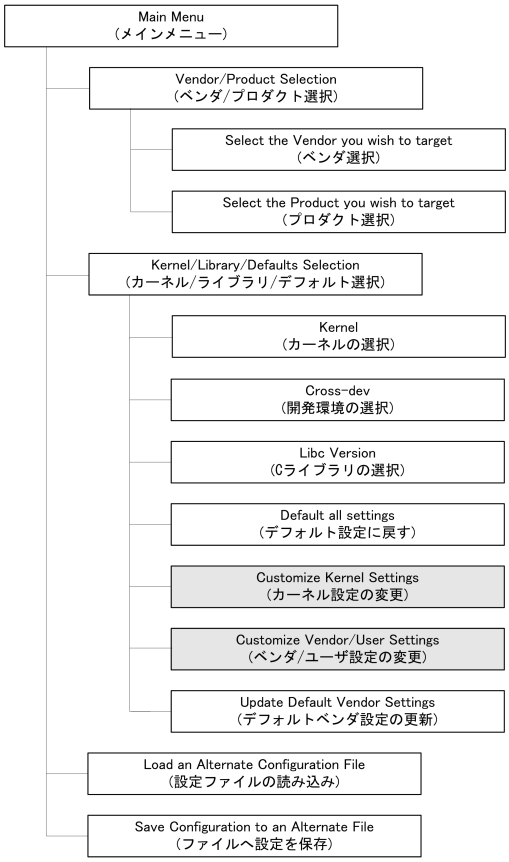


図 7.14 設定画面の階層

以下、それぞれの設定について説明していきます。

7.4.1. ベンダー、プロダクトの選択

まず初めに、ベンダーとプロダクトを選択します。この作業は、他の設定の前に行う必要があります。

atmrk-dist ディレクトリで **make menuconfig** を実行すると、「Main Menu」画面が表示されます。「Main Menu」画面で「Vendor/Product Selection」を選択すると、「Vendor/Product Selection」画面へ移動します。

最初に、ベンダーを選択します。「Vendor/Product Selection」画面で「(SnapGear) Vendor」という行を選択すると、「Vendor」画面に移動します。AtmarkTechno 社製品を使用する場合、「Vendor」画面で「AtmarkTechno」を選択してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    Vendor
      AtmarkTechno
```

図 7.15 ベンダーとして AtmarkTechno を選択する

次に、プロダクトを選択します。「Vendor/Product Selection」画面の「(Armadillo-210.Base) AtmarkTechno Products」という行を選択すると、「AtmarkTechno Products」画面に移動します。「AtmarkTechno Products」画面では、使用するプロダクトを選択してください。独自プロダクト用のプロダクトディレクトリを追加していた場合、ここにディレクトリ名が表示されますので、それを選択してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    AtmarkTechno Products
      プロダクト名
```

図 7.16 プロダクトを選択する

「Vendor/Product Selection」画面で「Exit」を選択すると、「Main Menu」画面に戻ります。そのまま「Exit」を選択して、「Do you wish to save your new kernel configuration?」画面で「Yes」を選択すると、選択したプロダクトの標準のコンフィギュレーションが適用されます。コンフィギュレーションを変更したい場合は、「7.4.2. ユーザーランドコンフィギュレーションの変更」や「7.4.3. カーネルコンフィギュレーションの変更」を参照してください。

7.4.2. ユーザーランドコンフィギュレーションの変更

ユーザーランドのコンフィギュレーションを変更すると、ユーザーランドのルートファイルシステムにどのアプリケーションプログラムやライブラリを含めるか選択することができます。

ユーザーランドのコンフィギュレーションを変更するには、「Main Menu」画面で「Kernel/Library/Defaults Selection」を選択して「Kernel/Library/Defaults Selection」画面へ移動し、「Customize Vendor/User Settings」にチェックを入れ、Main Menu を終了してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    [*] Customize Vendor/User Settings
```

図 7.17 ユーザーランドコンフィギュレーションの変更

すると、続いて「Userland Configuration」画面が表示されます。

「Userland Configuration」画面のメニューは以下のような項目に別れています。

表 7.3 Userland Configuration メニュー一覧

セクション	説明
Vendor specific	ベンダー固有の設定を行います。ルートファイルシステムのサイズや、製品固有のアプリケーションを選択できます。
Core Application	システムとして動作するために必要な基本的なアプリケーションが入っています。システムの初期化を行う init やユーザ認証の login などがこのセクションで選択できます。
Library Configuration	アプリケーションが必要とするライブラリの選択ができます。
Flash Tools	フラッシュメモリに関係のあるアプリケーションが選択できます。netflash と呼ばれるネットワークアップデート用アプリケーションもここで選択できます。
Filesystem Applications	ファイルシステムに関係のあるアプリケーションが選択できます。flatfsd はここで選択することができます。その他、mount、fdisk、ext2 ファイルシステム、Samba などが含まれます。
Network Applications	ネットワークに関係のあるアプリケーションが選択できます。dhcpcd-new、ftpd、ifconfig、inetd、thttpd の他にも ppp やワイヤレスネットワークのユーティリティなども含まれます。
Miscellaneous Applications	上記のカテゴリに属さないアプリケーションが収録されています。Unix の一般的なコマンド(cp、ls、rm 等)やエディタ、オーディオ関連、スクリプト言語インタプリタなどが含まれます。
Busybox	BusyBox のカスタマイズを行います。BusyBox は多くのカスタマイズができるため、別セクションになっています。
Tinylogin	Tinylogin は BusyBox のように複数コマンドの機能をもつアプリケーションです。login や passwd、getty など認証に関係のある機能を提供します。多くのカスタマイズが可能なため別セクションになっています。
MicroWindows	MicroWindows は組み込み機器をターゲットにしたグラフィカルウインド環境です。LCD などを持つ機器を開発する場合に使えます。
Game	ゲームです。説明はいらないですよね。
Miscellaneous Configuration	いろいろな設定がまとめられています。
Debug Builds	デバッグ用のオプションがまとめられています。開発中にアプリケーションのデバッグを行うときに選択します。

ルートファイルシステムに含めたいアプリケーションプログラムやライブラリにはチェックを入れ、含めたくないもののチェックを外してください。

選択が完了したら、「Userland Configuration」画面で「Exit」を選択してください。「Do you wish to save your new kernel configuration?」画面が再度表示されますので、設定を反映する場合には「Yes」を選択してください。

7.4.3. カーネルコンフィギュレーションの変更

カーネルのコンフィギュレーションを変更すると、カーネルに含めるデバイスドライバなどの機能を選択することができます。

カーネルのコンフィギュレーションを変更するには、「Main Menu」画面で「Kernel/Library/Defaults Selection」を選択して「Kernel/Library/Defaults Selection」画面へ移動し、「Customize Kernel Settings」にチェックを入れ[2]、Main Menu を終了してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    [*] Customize Kernel Settings
```

図 7.18 カーネルコンフィギュレーションを変更する

すると、続いて「Linux Kernel Configuration」画面が表示されます。

「Linux Kernel Configuration」画面のメニューは以下のような項目に別れています。

表 7.4 Linux Kernel Configuration メニュー一覧

セクション	説明
General setup	カーネル全般に渡る設定を行います。
Enable loadable module support	機能を有効にするかを含め、カーネルモジュールに関する設定を行います。
Enable the block layer	ブロックレイヤーを有効にするか選択します。ブロックレイヤーを無効にすると、ブロックデバイスを使用できなくなります。
System Type	システム依存の設定を行います。Armadillo-400 シリーズのボードオプションの設定もここで行います。
Bus support	PCMCIA/PC カードバスを有効に関する設定を行います。
kernel Features	ハイレゾリューションサポートの設定など、カーネルの機能に関する設定を行います。
Boot options	ブート時の挙動に関する設定を行います。標準のカーネルオプションはここで設定できます。
Floating point emulation	浮動小数点演算をエミュレートする機能の設定を行います。ABI を EABI にする場合必要ありません。
Userspace binary formats	カーネルがサポートするユーザーランドで動作するプログラムのフォーマットを選択します。
Power management options	電源管理機能に関する設定を行います。
Networking	ネットワーク機能に関する設定を行います。
Device Drivers	カーネルで有効にするデバイスドライバを選択します。多くの場合設定する必要があるのは、このセクションでしょう。
File systems	カーネルがサポートするファイルシステムを選択します。
Kernel hacking	主にカーネルのデバッグ用の設定です。
CodeTEST setup	同じく、カーネルのデバッグ用の設定です。
Security options	セキュリティ機能に関する設定を行います。
Cryptographic API	カーネルで有効にする暗号 API を選択します。
Library routines	カーネルで有効にするライブラリルーチンを選択します。

カーネルに含めたい機能にはチェックを入れ、含めたくないもののチェックを外してください。

[2]「Customize Vendor/User Settings」と「Customize Kernel Settings」は同時に選択することもできます。両方選択した場合、カーネルコンフィギュレーションの変更が完了した後に、「Userland Configuration」画面が表示されます。

選択が完了したら、「Linux Kernel Configuration」画面で「Exit」を選択してください。「Do you wish to save your new kernel configuration?」画面が再度表示されますので、設定を反映する場合には「Yes」を選択してください。

7.4.4. コンフィギュレーションの変更を元に戻す

「7.4.2. ユーザーランドコンフィギュレーションの変更」や「7.4.3. カーネルコンフィギュレーションの変更」で変更したコンフィギュレーションを元に戻すには、「Main Menu」画面で「Kernel/Library/Defaults Selection」を選択して「Kernel/Library/Defaults Selection」画面へ移動し、「Default all settings (lose changes)」にチェックを入れ、Main Menu を終了してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    [*] Default all settings (lose changes)
```

図 7.19 コンフィギュレーションの変更を元に戻す

ベンダーとプロダクトはそのままで、標準のコンフィギュレーションに戻ります。

7.4.5. コンフィギュレーションの保存

変更したコンフィギュレーションをプロダクトディレクトリに保存し、それを標準のコンフィギュレーションとすることができます。この機能は「7.7.1. Atmark Dist がアップデートされた場合」で使用します。

コンフィギュレーションを保存するには、「Main Menu」画面で「Kernel/Library/Defaults Selection」を選択して「Kernel/Library/Defaults Selection」画面へ移動し、「Update Default Vendor Settings」にチェックを入れ、Main Menu を終了してください。

```
Main Menu
  Kernel/Library/Defaults Selection
    [*] Update Default Vendor Settings
```

図 7.20 コンフィギュレーションを保存する

7.4.6. その他のコンフィギュレーション

「Kernel/Library/Defaults Selection」画面では、開発環境と C ライブラリを選択することもできます。これらは、通常変更する必要はありませんので、標準の設定を使用してください。

「Kernel/Library/Defaults Selection」画面で「Cross-dev」を選択すると、開発環境を選択できます。開発環境を変更することで、異なる ABI やアーキテクチャ用のバイナリを生成することができます。

```
Main Menu
  Kernel/Library/Defaults Selection
    Cross-dev
```

図 7.21 開発環境の選択

選択できる開発環境は、プロダクトごとに決まっています。開発環境として Armadillo で選択できる項目は以下の通りです。

表 7.5 開発環境

項目	内容
default	プロダクト毎の標準設定を使用
arm-vfp	OABI で VFP[1]用命令を使用したバイナリを生成
arm	OABI のバイナリを生成
armel	EABI のバイナリを生成

[1]浮動小数点演算ユニットの一つ。Atmark Techno 社製品では、Armadillo-500 のみが VFP を使用可能です。

「Kernel/Library/Defaults Selection」画面で「Libc Version」を選択すると、どの C ライブラリを使用するか指定できます。以下の 4 つが選択対象です。

- None
- glibc
- uC-libc
- uClibc

「None」を選択すると、開発環境にインストールされている C ライブラリを使用します。それ以外を選択すると、Atmark Dist に含まれているソースコードをビルドして使用します。通常は、「None」を選択してください。

7.5. ソースコードのビルドとイメージファイルの作成

本章では、前章で設定したコンフィギュレーションに従って、ソースコードをビルドしイメージファイルを作成する手順について説明します。

ソースコードをビルドしイメージファイルを作成することを、「Atmark Dist をビルドする」と表現することにします。

7.5.1. ビルドの実行

Atmark Dist をビルドすると言っても、すべてをビルドシステムが行ってくれるため、ユーザーは **make** コマンドを実行するだけです。

```
[ATDE ~/atmark-dist]$ make
:
:
[ATDE ~/atmark-dist]$ ls images
linux.bin  linux.bin.gz  romfs.img  romfs.img.gz
```

図 7.22 Atmark Dist のビルド

ビルドが正常に完了すると、images ディレクトリにイメージファイルが作成されます。linux.bin がカーネルの、romfs.img がユーザーランドのイメージファイルです。gz で終わるファイルが、それぞれを圧縮したものです。通常は、圧縮されたイメージファイルを Armadillo に書き込みます。

7.5.2. 詳細なビルドの流れ

デフォルト(all)ターゲットのビルドを行うと、イメージファイルが作成されますが、この間にたくさんのターゲットが実行されています。Atmark Dist では、すべてを一度にビルドするのではなく、対象を指定してそれだけをビルドすることもできます。修正を行った部分だけをビルドしなおすといったことができるので、ビルド時間を大幅に短縮できます。

デフォルトターゲットのルールは、トップレベル makefile に以下のように記載されています。

```
ifeq (.config,$(wildcard .config))
include .config

all: subdirs romfs modules modules_install image
else
all: config_error
endif
```

図 7.23 デフォルトターゲットのルール

デフォルトターゲットを指定してビルドすると subdirs、romfs、modules、modules_install、image の順にビルドが行われるのが分かります。この流れに沿って、各ターゲットで行われている処理を説明します。

7.5.2.1. subdirs ターゲット

subdirs ターゲットのルールは、トップレベル makefile に以下のように記述されています。

```
VENDDIR = $(ROOTDIR)/vendors/$(CONFIG_VENDOR)/$(CONFIG_PRODUCT)/.
DIRS    = include lib include user
:
:
subdirs: linux
        for dir in $(DIRS) ; do [ ! -d $$dir ] || $(MAKEARCH_KERNEL) -C $$dir || exit 1 ; don
        $(MAKEARCH_KERNEL) -C $(VENDDIR)
```

図 7.24 subdirs ターゲットのルール

subdirs ターゲットは linux ターゲットに依存しているため、まず linux ターゲットのルールが適用されます。linux ターゲットでは、Linxu カーネルのビルドを行います。

subdirs ターゲットのルールでは、lib、include、user、VENDDIR(プロダクトディレクトリ)の各ディレクトリを順にビルドします。

lib ディレクトリは、ユーザーランドのライブラリを収録したディレクトリです。uClibc と glibc は lib ディレクトリに入っていませんが、コンフィグ時に lib ディレクトリ内にシンボリックリンクを生成するようになっています。

include ディレクトリは、インクルードファイルを収めたディレクトリです。all ターゲットのルールが適用されると、ビルドしたライブラリのインクルードファイルへのシンボリックリンクを作成します。

user ディレクトリは、ユーザーランドのアプリケーションプログラムを集めたディレクトリです。user ディレクトリには専用の Makefile が用意されており、トップレベル makefile はそちらに制御を任せるようになっています。

最後に、プロダクトディレクトリのビルドが行われます。

7.5.2.2. romfs ターゲット

romfs ターゲットでは、各ディレクトリに対して romfs ターゲットを再帰的に呼びだします。多くの場合、romfs-inst.sh を使って、必要なファイルを atmark-dist/romfs ディレクトリにインストールします。romfs-inst.sh については「7.5.3. romfs インストールツール(romfs-inst.sh)」で説明します。

romfs ディレクトリ以下のディレクトリ構成は、ターゲットシステム上で見えるディレクトリ構成と同じ構成になっています。romfs ディレクトリをルートディレクトリとして、その下に bin や dev、etc などのディレクトリが配置されます。

ディレクトリ名に romfs という名前が使われているのは、Atmark Dist の元となった uClinux-dist が対象としていた組み込みシステムでは、romfs ファイルシステムが使われることが多かったためです。しかし、romfs ディレクトリ自体は、romfs に依存しているわけではありません。Armadillo-400 シリーズでは、ユーザーランドのルートファイルシステムは ext2 です。

各ディレクトリの romfs ターゲットの処理が完了した後、もし、ベンダーディレクトリの atmark-dist/Common/tools ディレクトリに lib-inst.sh と auto-strip.sh があり、それらを使用するように設定されていた場合、以下の処理が行われます。

lib-inst.sh を使用して、romfs ディレクトリにある実行ファイルが使用する共有ライブラリを romfs ディレクトリにインストールします。開発環境として armel を選択していた場合、ATDE3 では/usr/arm-linux-gnueabi/lib ディレクトリ以下にある ARM EABI 用の共有ファイルを使用します。

注意: 自動でインストールされないライブラリ

動的ロードされるライブラリは、lib-inst.sh では検出できないため、romfs ディレクトリにインストールされません。プログラム中でライブラリを動的ロードしている場合は、使用するライブラリを romfs-inst.sh を使うなどして明示的に romfs ディレクトリにインストールする必要があります。

続いて、auto-strip.sh を使用して、romfs ディレクトリにある実行ファイルに対して、**strip** コマンドを適用します。**strip** コマンドはオブジェクトファイルの不要なシンボルを削除し、ファイルサイズ小さくします。

7.5.2.3. modules ターゲット

Linux カーネルでは、ドライバーなどをカーネルモジュールとして Linux カーネルから分離した形で作成することができるようになっています。modules ターゲットでは、Linux のビルドシステムに modules を make ターゲットとして渡し、カーネルモジュールをビルドします。

7.5.2.4. modules_install ターゲット

modules_install ターゲットでは、Linux ビルドシステムで作成されたカーネルモジュールを romfs にインストールします。カーネルモジュールは romfs/lib/modules ディレクトリに置かれます。

7.5.2.5. image ターゲット

image ターゲットは、イメージファイルを作成するためのターゲットです。トップレベル makefile では、単にプロダクトディレクトリにある Makefile の image ターゲットを呼ぶだけになっているので、実際の処理はそちらに書かなければなりません。

通常、以下の処理を記述します。

1. Linux カーネルのイメージファイルを生成。
2. romfs ディレクトリからユーザーランドのルートファイルシステムのイメージファイルを生成。
3. NetFlash[3]用にチェックサムなどを作成し、イメージファイルに添付する。

イメージファイルは、images ディレクトリに作成されます。

7.5.2.6. 指定可能なターゲットのまとめ

ビルド対象を特定するために指定可能なターゲットを以下にまとめます。

1. linux: Linux カーネルをビルドします
2. lib: ライブラリをビルドします
3. user: user ディレクトリのアプリケーションプログラムをビルドします
4. romfs: romfs ディレクトリにファイルをインストールします
5. modules: Linux カーネルモジュールをビルドします
6. modules_install: Linux カーネルモジュールを romfs ディレクトリにインストールします
7. image: イメージファイルを作成します

7.5.3. romfs インストールツール(romfs-inst.sh)

romfs-inst.sh は、romfs ディレクトリへのファイルのインストールを簡単にするスクリプトです。このスクリプトは、atmark-dist/tools ディレクトリにあります。

romfs-inst.sh のパスは、トップレベル makefile で ROMFSINST という変数に代入されています。そのため、プロダクトディレクトリをはじめとする各ディレクトリの makefile では、romfs-inst.sh がどこにあるかを気にせず、ROMFSINST という変数で使用できます。

7.5.3.1. 概要

romfs-inst.sh は、romfs ディレクトリを指定する環境変数 ROMFSDIR が設定されていない場合、簡単な help を出力します。

[3]ネットワーク経由でファイルを取得し、フラッシュメモリに書き込むアプリケーションプログラム

```
[ATDE ~/atmark-dist]$ tools/romfs-inst.sh
ROMFSDIR is not set
tools/romfs-inst.sh: [options] [src] dst
    -v          : output actions performed.
    -e env-var  : only take action if env-var is set to "y".
    -o option   : only take action if option is set to "y".
    -p perms    : chmod style permissions for dst.
    -a text     : append text to dst.
        -A pattern  : only append text if pattern doesn't exist in file
    -l link     : dst is a link to 'link'.
    -s sym-link : dst is a sym-link to 'sym-link'.

    if "src" is not provided,  basename is run on dst to determine the
    source in the current directory.

        multiple -e and -o options are ANDed together.  To achieve an OR affect
        use a single -e/-o with 1 or more y/n/"" chars in the condition.

        if src is a directory,  everything in it is copied recursively to dst
        with special files removed (currently CVS dirs).
```

図 7.25 romfs-inst.sh のヘルプ

romfs-inst.sh のコマンド構文は以下のとおりです。

```
romfs-inst.sh [options] [src] <dst>
```

図 7.26 romfs-inst.sh の構文

options と *src* は省略可能です。もし、*src* が指定されなかった場合、**basename** コマンドが *dst* に適用され、戻り値を *src* として使用します。

options が指定されなかった場合、romfs-inst.sh は、現在のディレクトリ[4]に *src* に指定された値と名前が一致するファイルがあった場合、そのファイルを romfs ディレクトリにコピーします。もし、*src* と名前が一致するディレクトリがあった場合、そのディレクトリ以下のすべてのファイルとディレクトリをコピーします。ただし、CVS ディレクトリはコピーされません。

options を指定することで、単純なコピー以外の処理をさせることができます。*options* には、以下のものが指定できます。

表 7.6 romfs-inst.sh のオプション

オプション	内容
-v	実際に実行した内容を出力
-e *env-var*	*env-var* が y のときだけ、指定されたアクションを実行
-o *option*	*option* が y のときだけ、指定されたアクションを実行
-p *perms*	chmod 方式で *dst* のパーミッションを指定
-a *text* [-A *pattern*]	*text* を *dst* に追加。-A *pattern* が指定されているときは、*pattern* が *dst* に含まれていない場合に text を追加

[4]つまり、プロダクトディレクトリの makefile で実行された場合は、プロダクトディレクトリ

オプション	内容
-l *link*	*dst* で指定された名前で、*link* へのハードリンクを作成
-s *sym-link*	*dst* で指定された名前で、*sym-link* へのシンボリックリンクを作成

7.5.3.2. ファイルのコピー

単純にファイルをコピーする場合は、romfs ターゲットを以下のように makefile に記述します。

```
romfs:
	$(ROMFSINST) src.txt /etc/dst.txt
```

図 7.27 ファイルをコピーするルール

これは、makefile があるディレクトリにある src.txt を romfs ディレクトリ以下の/etc/dst.txt にインストールすることを意味しています。もし、romfs ディレクトリが atmark-dist/romfs であれば、atmark-dist/romfs/etc/dst.txt というファイルができあがります。

src.txt の名前を変更して dst.txt としておくことで、以下のように簡単に記述することができます。

```
romfs:
	$(ROMFSINST) /etc/dst.txt
```

図 7.28 ファイルをコピーするルール(src を省略)

src が省略された場合、romfs-inst.sh は *dst* に **basename** コマンドを適用して、*src* の値とします。

/etc/dst.txt に **basename** コマンドを適用すると dst.txt となるので、上記のルールは以下と同じ意味を持ちます。

```
romfs:
	$(ROMFSINST) dst.txt /etc/dst.txt
```

7.5.3.3. ディレクトリのコピー

多くのファイルをコピーしたい場合は、ディレクトリごとコピーすると簡単です。たとえば、ターゲットの/etc ディレクトリに多くの設定ファイルをコピーする場合などです。

makefile があるディレクトリに etc という名前でディレクトリを作成し、必要なファイルを置きます。そして makefile に以下のように記述します。

```
romfs:
	$(ROMFSINST) /etc
```

図 7.29 ディレクトリをコピーするルール

この例でも *src* が省略されているので、romfs-inst.sh は *dst* に **basename** コマンドを適用します。/etc に **basename** コマンドを適用すると etc となるので、romfs-inst.sh は makefile があるディレクトリ内の etc というファイルまたはディレクトリを探します。そして今回のようにディレクトリの場合、romfs-inst.sh はディレクトリ内にあるファイルとディレクトリをすべてインストールしてくれます。

もちろん、別の名前のディレクトリとしてでインストールすることも可能です。

```
romfs:
        $(ROMFSINST) /etc /var
```

図 7.30 ディレクトリをコピーするルール(ディレクトリ名を指定する)

このコマンドでは、makefile があるディレクトリ内の etc というディレクトリを atmark-dist/romfs/var にコピーします。

7.5.3.4. リンクの作成

romfs-inst.sh を使うと、単純なファイルやディレクトリのコピーだけでなく、リンクを作成することもできます。

シンボリックリンクを作成する場合は、-s オプションを指定します。例として、ターゲットのルートディレクトリ直下の a.txt へのシンボリックリンクを b.txt という名前で作成する場合、以下のようになります。

```
romfs:
        $(ROMFSINST) /a.txt
        $(ROMFSINST) -s a.txt /b.txt
```

図 7.31 シンボリックリンクを作成するルール

このルールを記述して **make romfs** を実行すると、結果は以下のようになります。

```
[ATDE ~/atmark-dist]$ make clean; make romfs
    :
    :
[ATDE ~/atmark-dist]$ ls -l romfs
total 0
-rw-r--r--  1 guest guest 0 Sep 24 05:43 a.txt
lrwxrwxrwx  1 guest guest 5 Sep 24 05:43 b.txt -> a.txt
```

図 7.32 シンボリックリンクの作成結果

ハードリンクを作成する場合は、-l オプションを指定します。

```
romfs:
        $(ROMFSINST) /a.txt
        $(ROMFSINST) -l a.txt /b.txt
```

図 7.33 ハードリンクを作成するルール

しかし、このように指定した場合、romfs ディレクトリ内にできた b.txt は、同じ romfs ディレクトリ内の a.txt ではなく、makefile があるディレクトリの a.txt へのハードリンクとなります。

ハードリンクの使用は混乱を招きますので、よほど開発環境のの容量に困っていないかぎりお勧めしません。現在の Atmark Dist でも、romfs 内へのハードリンクはあまり使用していません。

7.5.3.5. ファイルへの追記

-a オプションを指定すると、ファイルへ文字列を追記できます。

構文は以下のようになります。

```
romfs-inst.sh -a "文字列" romfs ディレクトリ内のファイル名
```

図 7.34 -a オプション指定時の romfs-inst.sh の構文

以下のように記述すると、atmark-dist/romfs/a.txt には、「Hello」と「World」が一行ずつ記述されます。

```
romfs:
        $(ROMFSINST) -a 'Hello' /a.txt
        $(ROMFSINST) -a 'World' /a.txt
```

図 7.35 文字列を追記するルール

7.5.3.6. 条件実行

-e オプションを指定すると、条件が一致した場合だけインストールを実行することができます。

構文は以下のようになります。

```
romfs-inst.sh -e 変数名 実行するコマンド
```

図 7.36 -e オプション指定時の romfs-inst.sh の構文

*変数名*に指定した変数の値が y または Y の場合だけ、コマンドを実行します。*変数名*とは、CONFIG_で始まる環境変数がよく用いられます。CONFIG_で始まる環境変数には、対応するコンフィギュレーションが選択されたときに y が代入されますので、特定のコンフィギュレーションが選択されているときだけインストールしたい場合に使用されます。

7.6. プロダクトディレクトリのカスタマイズ

プロダクトディレクトリには、プロダクト専用のアプリケーションプログラム、設定ファイル、コンフィギュレーションファイルなどを置くことができます。Armadillo をベースに独自のプロダクトを開発する場合、使用する Armadiilo 用のプロダクトディレクトリをコピーして、独自プロダクト用のプロダクトディレクトリとすることで、設定や管理が楽になります。

7.6.1. プロダクトディレクトリに含まれるファイル

プロダクトディレクトリには、最低限以下のファイルが必要です。

1. Makefile
2. config.arch
3. config.vendor
4. tools/config-linux.conf

Makefile には、デフォルト(all)、clean、distclean、romfs、image ターゲットを記述する必要があります。それぞれのターゲットでどのような処理を行うべきかは、「7.2. 基本ターゲット」や「7.5.2. 詳細なビルドの流れ」を参照してください。

config.arch ファイルには、アーキテクチャに依存した設定を記述します。実際には、アーキテクチャごとにデフォルトの値がすでに用意されているため、上書きする設定だけを書けばよいようになっています。

config.vendor ファイルには、ユーザーランドコンフィギュレーションの標準設定を記述します。現在のユーザーランドコンフィギュレーションは、atamrk-dist/.config に保存されています。「図 7.20. コンフィギュレーションを保存する」の手順を実行すると、atamrk-dist/.config ファイルがプロダクトディレクトリの config.vendor ファイルにコピーされます。

tools/config-linux.conf には、Linux カーネルの標準コンフィギュレーションファイルを記述します。最初にカーネルコンフィギュレーションを設定する際、このファイルで指定されたファイルに記述されたコンフィギュレーションを使用します。現在のカーネルコンフィギュレーションは、atmark-dist/linux-2.6.x/.config ファイルに保存されています。「図 7.20. コンフィギュレーションを保存する」の手順を実行すると、atmark-dist/linux-2.6.x/.config ファイルがプロダクトディレクトリの config.linux-2.6.x ファイルにコピーされます。

7.6.2. プロダクトディレクトリの作成

Atamrk Dist を使用して独自プロダクトを開発する場合、まずはベースとなる Armadillo 用のプロダクトディレクトリをコピーして、独自プロダクト用のプロダクトディレクトリとしてください。基本的に、プロダクトに依存するすべての変更作業は、このディレクトリ内で行います。

ここでは Armadillo-440 をベースとして my-product という名前のプロダクトディレクトリを作成します。

```
[ATDE ~/atmark-dist]$ cp -a vendors/AtmarkTechno/Armadillo-440 vendors/AtmarkTechno/my-product
```

図 7.37 プロダクトディレクトリの作成

7.6.3. アプリケーションプログラムの追加

Armadillo を使用した組み込みシステムを開発するにあたり、ほとんどの場合、独自のアプリケーションプログラムを作成することになるでしょう。ここでは、独自に作成したアプリケーションプログラムを Atmark Dist に追加する方法を紹介します。

7.6.3.1. ディレクトリの準備

プロダクトディレクトリ以下に、アプリケーション用のディレクトリを作成します。ここでは hello とします。

```
[ATDE ~/atmark-dist]$ cd vendors/AtmarkTechno/my-product
[ATDE ~/atmark-dist/vendors/AtmarkTechno/my-product]$ mkdir hello
```

図 7.38 アプリケーションプログラム用ディレクトリの追加

7.6.3.2. ソースファイルの追加

ソースコードを作成し、hello ディレクトリに追加します。以下の内容を、hello ディレクトリに hello.c という名前で追加してください。

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        printf("Hello World!\n");

        return EXIT_SUCCESS;
}
```

図 7.39 hello.c

7.6.3.3. Makefile の作成

hello.c をコンパイルするための makefile を作成し、hello ディレクトリに追加します。以下の内容を、hello ディレクトリに Makefile という名前で追加してください。

```
TARGET = hello

all: $(TARGET)

$(TARGET): hello.o
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

romfs:
        $(ROMFSINST) /bin/$(TARGET)

clean:
        $(RM) *~ *.o $(TARGET)

%.o: %.c
        $(CC) -c $(CFLAGS) -o $@ $<
```

図 7.40 Makefile

7.6.3.4. プロダクトディレクトリの makefile を修正

プログラムを追加した場合、プロダクトディレクトリの makefile を変更する必要があります。

atmark-dist/vendors/AtmarkTechno/my-product/Makefile には、SUBDIR_y という変数が用意されています。この変数にアプリケーションプログラム用のディレクトリ名を追加してください。

```
:
:
:
SUBDIR_y =
SUBDIR_$(CONFIG_VENDOR_FUNCTESTER_FUNCTESTER)   += functester/
SUBDIR_$(CONFIG_VENDOR_GPIOCTRL_GPIOCTRL)       += gpioctrl/
SUBDIR_$(CONFIG_VENDOR_LEDCTRL_LEDCTRL)         += ledctrl/
SUBDIR_$(CONFIG_VENDOR_SWMGR_SWMGR)             += swmgr/
SUBDIR_y += hello/  ❶
:
:
:
```

図 7.41 プロダクトディレクトリの makefile にアプリケーションプログラム用のディレクトリを追加

❶ 作成したディレクトリを SUBDIR_y に追加します。

プロダクト専用のアプリケーションプログラムを追加する手順は以上です。**make** コマンドを実行すると、実行ファイル hello を含んだユーザーランドのイメージファイルが作成されます。

7.6.4. ファイルの追加、変更

プロダクトディレクトリには、アプリケーションプログラムのソースコードだけでなく、ルートファイルシステムに含めたい設定ファイルやデータファイルなども置くことができます。

Aramdillo-440 ディレクトリをベースにプロダクトディレクトリを作成した場合、プロダクトディレクトリ以下の etc、home、usr ディレクトリ内にあるファイルやディレクトリはすべて romfs ディレクトリにコピーされるようになっています。通常は、これらのディレクトリに設定ファイルやデータファイルを置くとよいでしょう。

etc、home、usr ディレクトリ以外の場所にファイルやディレクトリを追加したい場合、プロダクトディレクトリの makefile を変更する必要があります。例として、var ディレクトリ内のファイルとディレクトリをすべて romfs ディレクトリにコピーした場合、atmark-dist/vendors/AtmarkTechno/my-product/Makefile を以下のように修正してください。

```
romfs:
(略)
        $(ROMFSINST) -v /etc
        $(ROMFSINST) -v /home
        $(ROMFSINST) -v /usr
        $(ROMFSINST) -v /var    ←ここに追加
```

図 7.42 var ディレクトリの追加

7.7. ソフトウェアアップデートへの対応

Linux カーネルや Atmark Dist は機能追加やバグ修正などで随時アップデートがおこなわれます。この章では、プロダクトの開発中に新しい Atmark Dist や Linux カーネルがリリースされた場合の対応方法について説明します。

7.7.1. Atmark Dist がアップデートされた場合

新しい Atamrk Dist がリリースされた場合、コンフィギュレーションを保存して、プロダクトディレクトリをまるごと新しい Atmark Dist にコピーするだけで、コンフィギュレーションを引き継いだまま、移行することができます。

例として、新しいバージョンの Atmark Dist を atmark-dist-new、古いバージョンの AtmarkDist を atmark-dist-old というディレクトリでホームディレクトリの下に作成しているものとして説明します。また使用しているプロダクトは、ベンダー名を AtmarkTechno、プロダクト名を my-product と設定しているものとします。

まず、**make clean** を実行して、古いバージョンの Atmark Dist の不要なファイルを削除します。

```
[ATDE ~/atmark-dist-old]$ make clean
```

図 7.43 不要なファイルの削除

次に、古いバージョンの Atmark Dist のコンフィギュレーションを保存します。「7.4.5. コンフィギュレーションの保存」の手順を実行してください。コンフィギュレーションファイルは、プロダクトディレクトリに保存されます。

以上で、古いバージョンの Atmark Dist の準備は完了です。

新しいバージョンの Atmark Dist のソースコードを取得して展開し、atmark-dist-new ディレクトリとしてください。atamrk-dist-new ディレクトリには、通常どおり、linux-2.6.x という名前で Linux カーネルのソースコードディレクトリのシンボリックリンクを張ります。

新しいバージョンの Atmark Dist の vendors/AtmarkTechno ディレクトリに古いバージョンで使用していたプロダクトディレクトリをコピーします。

```
[ATDE ~/atmark-dist-new]$ cp -a ~/atmark-dist-old/vendors/AtmarkTechno/my-product vendors/
AtmarkTechno
```

図 7.44 プロダクトディレクトリのコピー

新しい Atmark Dist の atmark-dist-new ディレクトリで **make menuconfig** を実行し、古いバージョンからコピーしたプロダクトを選択します。

```
Main Menu
  Vendor/Product Selection
    (AtmarkTechno) Vendor
    (my-product) AtmarkTechno Products
```

図 7.45 プロダクト設定

my-product を選択し、コンフィギュレーション設定画面を終了すると、Atmark Dist は自動的にプロダクトディレクトリにあるコンフィギュレーションファイルを読み込みます。

デフォルト設定を読み込み中に新しい Atmark Dist で追加されたコンフィギュレーションがあれば、それをどのように設定するか尋ねられます。設定を入力し Enter を押してください。デフォルトの設定

を選択する場合は、何も入力せず Enter を押してください。以下に、追加された項目があった場合の設定例を示します。

```
mjpg-streamer (CONFIG_USER_MJPG_STREAMER_MJPG_STREAMER) [N/y/?] (NEW) ❶
```

図 7.46 新しく追加された項目の設定例

❶ mjpg-streamer の設定をおこないます。[N/y/?]は入力できる値を示しています。「y」を入力して Enter を押すと、mjpg-streamer を有効に、「n」を入力して Enter を押すと mjpg-stremaer を無効に設定します。「?」を入力して Enter を押すとヘルプが表示されます。入力できる値が大文字になっている値はデフォルトの設定値であることを示しています。

新しいコンフィギュレーションがあった場合は、選択した設定を保存するために、再度「7.4.5. コンフィギュレーションの保存」を実行してください。

7.7.2. Linux カーネルがアップデートされた場合

新しい Linux カーネルがリリースされた場合、コンフィギュレーションを保存して、Linux カーネルのソースコードディレクトリへのシンボリックリンクを差し替えるだけで、コンフィギュレーションを引き継いだまま、移行することができます。

この章では例として、新しいバージョンの Linux カーネルを Linux-2.6.26-at-new とし、古いバージョンの Linux カーネルを Linux-2.6.26-at-old として説明します。

まず、**make clean** を実行して、Atmark Dist の不要なファイルを削除します。

```
[ATDE ~/atmark-dist]$ make clean
```

図 7.47 不要なファイルの削除

次に、古い Linux カーネルのコンフィギュレーションを保存します。「7.4.5. コンフィギュレーションの保存」の手順を実行してください。コンフィギュレーションファイルは、プロダクトディレクトリに保存されます。

古いバージョンの Linux カーネルのソースコードディレクトリへのシンボリックリンクを削除します

```
[ATDE ~/atmark-dist]$ rm linux-2.6.x
```

図 7.48 古いバージョンの Linux カーネルへのシンボリックリンクの削除

新しいバージョンの Linux カーネルのソースコードディレクトリへのシンボリックリンクを作成します。

```
[ATDE ~/atmark-dist]$ ln -s ../linux-2.6.26-at-new linux-2.6.x
```

図 7.49 新しいバージョンの Linux カーネルへのシンボリックリンクの作成

make menuconfig を実行して標準のコンフィギュレーションを読み込みます。「図 7.19. コンフィギュレーションの変更を元に戻す」の手順を実行してください。

新しいカーネルで追加されたコンフィギュレーションがあれば、それをどのように設定するか尋ねられます。設定を入力し Enter を押してください。デフォルトの設定を選択する場合は、何も入力せず Enter を押してください。以下に、新しく追加された項目があった場合の設定例を示します。

```
Size of 1 qTD's buffer (USB_STATIC_IRAM_TD_SIZE) [2048] (NEW) ❶
```

図 7.50 新しく追加された項目の設定例

❶ Size of 1 qTD's buffer の設定をおこないます。[2048]は入力できる値が数字であり、デフォルトの値が 2048 であることを示しています。値を入力せず Enter を押した場合はデフォルトの値になります。

新しいコンフィギュレーションがあった場合は、選択した設定を保存するために、再度「7.4.5. コンフィギュレーションの保存」を実行してください。

8. 組み込みシステム構築の定石

本章では、組み込みシステムを構築する際の定石を紹介します。

8.1. 起動時にコマンドを自動実行する

システム起動時にコマンドを自動的に実行する方法には、以下の二種類があります。

1. 初期化スクリプトとして登録する。
2. inittab に登録する。

本章では、これらの方法について説明します。

8.1.1. 初期化スクリプトで自動実行する(一時的な方法)

起動時に特定のコマンドを自動実行したい場合、コマンドをスクリプトに記述し、それを初期化スクリプトとして登録[1]します。

初期化スクリプトは、以下の順番で実行されます。

1. /etc/init.d/rc ファイル。
2. /etc/rc.d/ディレクトリにあるファイル。
3. /etc/config/rc.local ファイル。

通常、/etc/init.d/rc ファイルは固定で、/etc/rc.d ディレクトリ内のファイルや/etc/config/rc.local をシステムに合わせて変更します。

Armadillo では、ルートファイルシステムに RAM ディスクを置いているため、/etc/rc.d ディレクトリのファイルを Armadillo 上で変更しても、次回起動時にはその変更は失われます。/etc/rc.d ディレクトリの内容を変更したい場合には、毎回ルートファイルシステムイメージを作成しなおさなければならないため、開発の初期段階などファイルの変更が頻繁にある場合、効率が良くありません。

一方で、/etc/config ディレクトリの内容は、**flatfsd** コマンドでフラッシュメモリのコンフィグ領域に保存することができ、起動時に復元されます[2]。そのため、/etc/config/rc.local を使うと、ルートファイルシステムイメージを更新せずに自動実行するコマンドを変更することができます。

例として、Armadillo-440 の標準イメージの/etc/config/rc.local は以下のようになっています。

```
#!/bin/sh

. /etc/init.d/functions

PATH=/bin:/sbin:/usr/bin:/usr/sbin

echo -n "Starting functester: "
```

[1]登録といっても、所定のディレクトリに所定の名前でファイルを置くだけです。
[2]この処理は/etc/rc.d/S04flatfsd で行っています。

```
export TZ=JST-9
DISPLAY=:0 functester > /dev/null 2>&1 &
check_status
```

図 8.1 Armadillo-440 の標準イメージの/etc/config/rc.local

「図 8.1. Armadillo-440 の標準イメージの/etc/config/rc.local」では、**functester** コマンド[3]をバックグラウンドプロセスとして実行しています。

/etc/config/rc.local がない場合は、ファイルを新規作成し、実行権限を付けることで起動時に自動で実行されます。

/etc/config/rc.local を新規作成したり、編集した場合、その内容をフラッシュメモリのコンフィグ領域に保存しなければなりません。コンフィグ領域に保存するには、**flatfsd** コマンドを使用します。

/etc/config/rc.local ファイルを扱うコマンドは、以下のようになります。

```
[armadillo ~]$ vi /etc/config/rc.local ❶
[armadillo ~]$ chmod +x /etc/config/rc.local ❷
[armadillo ~]$ flatfsd -s ❸
```

図 8.2 /etc/config/rc.local ファイルを変更しフラッシュメモリに保存する

❶ rc.local ファイルを新規作成または編集します。

❷ 実行権限を付けます。

❸ フラッシュメモリに保存します。

8.1.2. 初期化スクリプトで自動実行する(恒久的な方法)

/etc/config/rc.local への変更は一時的なものです。開発の最終段階でルートファイルシステムのイメージを固める時には、自動実行するスクリプトは、/etc/rc.d ディレクトリに登録します。

/etc/rc.d ディレクトリのファイルは S で始まり、文字コード順でソートした場合に先にくるものから順番に実行されます。通常、実行される順番が分かりやすいように、S の後には二桁の数値文字が続きます。

大抵の場合、/etc/rc.d ディレクトリのファイルは、/etc/init.d ディレクトリのファイルへのシンボリックリンクになっています。そのため、スクリプト本体のファイル名は固定で、シンボリックリンクの名前を変更するだけで、スクリプトの実行順番を変えることができます。

初期化スクリプトの実行順番には、注意が必要です。例えば、ネットワークの設定を行う前に Web サーバーを起動してはいけません。

また、バックグラウンドプロセスやデーモンとして起動したプロセスの実行順番には、より注意が必要です。バックグラウンドプロセスやデーモンを実行するコマンドは、プロセスを起動するとすぐに戻ってきます。そのため、後続のコマンドを実行する際に、それらのプロセスが行う処理が完了しているとは限りません。

[3]液晶画面に表示され、タッチパネルによる操作で Armadillo-440 の動作確認を行うことができる、ファンクションテストアプリケーションプログラム。

Armadillo-420 や Armadillo-440 の標準イメージでは、/etc/rc.d ディレクトリ内の初期化スクリプトは、以下の順番で実行されます。

1. ファイルやディレクトリの準備
    a. S03udevd：udevd の起動とデバイスノードの作成。
    b. S04flatfsd：コンフィグ領域に保存した内容を/etc/config ディレクトリに復元。
    c. S05checkroot：重要なファイルやディレクトリの権限の設定。
    d. S06checkftp：FTP サーバーが使用する/home/ftp ディレクトリの設定。
2. ログデーモンの起動
    a. S10syslogd：syslog デーモンの起動。
    b. S20klogd：カーネルログデーモンの起動。
3. ネットワークの設定
    a. S30firewall：ファイヤーウォール(iptables)の設定。
    b. S30hostname：ホストネームの設定。
    c. S40networking：ネットワークインターフェースの有効化。
4. 各種ネットワークサーバの起動
    a. S60inetd：inetd(スーパーサーバー)の起動。
    b. S70at-cgi：at-cgi[4]で必要なファイルの設定。
    c. S70lighttpd：lighttpd(Web サーバー)の起動。
    d. S71avahi:avahi-daemon(Zeroconf サーバー)の起動。
5. その他
    a. S99misc：/home/ftp/pub ディレクトリを ramfs でマウント。
6. rc.local の実行
    a. S99rc.local：/etc/config/rc.local ファイルが存在し、実行可能なファイルであれば実行。

新しく初期化スクリプトを追加する場合、どの順番でおこなうべきか、良く検討してください。

Atmark Dist で作成するルートファイルシステムに初期化スクリプトを追加するには、以下のような手順でおこないます。

1. プロダクトディレクトリ以下の etc/init.d ディレクトリに初期化スクリプトを追加する。
2. rc.d ディレクトリのシンボリックリンクを作成するようプロダクトディレクトリ以下の Makefile を修正する。

[4]Web ブラウザで Armadillo の設定管理を行うための CGI。

Atmark Dist のプロダクトディレクトリ以下の etc/init.d ディレクトリにファイルを追加すると、Armadillo のルートファイルシステムの/etc/init.d ディレクトリにファイルが追加されます。

また、Atmark Dist のプロダクトディレクトリの Makefile に、Armadillo のルートファイルシステムの/etc/init.d ディレクトリのファイルへのシンボリックリンクを/etc/rc.d ディレクトリに作成するよう記述しておくと、**make romfs** を実行時に、シンボリックリンクが作成されます。

以下に Makefile の変更箇所を示します。romfs ターゲットで実行するコマンドに、$(ROMFSINST) コマンド[5]を追加します。

```
romfs:
        @rm -f etc/DISTNAME
(中略)⏎
        $(ROMFSINST) -s /etc/init.d/misc /etc/rc.d/S99misc
        [ "$(CONFIG_USER_FLATFSD_FLATFSD)" != "y" ] || \
        $(ROMFSINST) -s /etc/init.d/rc.local /etc/rc.d/S99rc.local
        この行にシンボリックリンクを作成するコマンドを追加する
```

図 8.3 /etc/rc.d ディレクトリにシンボリックリンクを追加するための変更箇所

Atmark Dist への修正を Armadillo に反映するには、通常通り、**make** コマンドを実行してユーザーランドのルートファイルシステムイメージ(romfs.img.gz)を作成しなおし、Armadillo のフラッシュメモリのユーザーランド領域に書き込んだ後、再起動してください。

8.1.3. inittab で自動実行する

起動時に自動実行するコマンドは、inittab に記述することもできます。

第 1 部「Armadillo が動作する仕組み」の「ユーザーランドの初期化処理」にも記述したように、inittab の action に指定できる値のうち、起動時にコマンドを実行するアクションは、sysinit と respawn です。

sysinit アクションのコマンドには、通常、初期化スクリプトの/etc/init.d/rc を実行するよう記述します。

respawn アクションに記述されたコマンドは、sysinit アクションに記述されたコマンドが完了したあと、すなわち、初期化スクリプトがすべて完了した後に実行されます。また、そのコマンドによって起動されたプロセスが終了すると、そのコマンドを再度実行します。

Atmark Dist で作成するルートファイルシステムの/etc/inittab を変更するには、Atmark Dist のプロダクトディレクトリ以下の etc/inittab ファイルを変更します。

respawn アクションの実行間隔

Atmark Dist に含まれる busybox の init では、inittab の respawn アクションで指定されたコマンドが起動したプロセスが、起動してから 5 秒以内に終了した場合、約 5 秒後にコマンドを再度実行するようになっています。

[5]実体は atmark-dist/tools/romfs-inst.sh

また、その状況が 5 回続くと、次のコマンドの実行はプロセスが終了してから約 5 分後に設定されます。

ですので、すぐに終了するようなプログラムは inittab に登録するには向いていません。次章で紹介する cron の利用を検討してください。

一時的な inittab

Atmark Dist に登録されている busybox の init は/etc/inittab のほかに/etc/config/inittab も読み込みます。

そのため、開発初期段階で一時的に使用する初期化スクリプトとして/etc/config/rc.local を使用したように、一時的に使用する inittab として/etc/config/inittab を使用することができます。

なお、/etc/config/inittab には、respawn アクションのみ指定することができます。その他のアクションを指定しても実行されません。

8.2. 定期的にコマンドを実行する

前章では、起動時にコマンドを実行する方法を紹介しました。本章では、指定した時刻に定期的にコマンドを実行する方法として、cron を使う方法を紹介します。

8.2.1. cron で定期実行する

cron は、指定した時刻に指定したコマンドを実行するための仕組みです。crontab と呼ばれる設定ファイルに記述された内容に従って、crond がコマンドを実行します。

コマンドを実行する時刻は分単位で指定可能で、毎分実行するという設定もできるので、間隔が比較的長い周期的な処理を行うこともできます。

crond の設定ファイルである crontab ファイルは、**crontab** コマンドを使用して作成することができます。

crontab ファイルには 1 行に付き、1 つの設定を記述します。1 つの設定は、スペースまたはタブによって区切られた、複数のフィールドを持ちます。各行のフォーマットは以下のとおりです。

```
分 時 月内日 月 曜日 コマンド
```

図 8.4 crontab ファイルのフォーマット

crond は、毎分 crontab ファイルの内容を調べます。分、時、月が現在のシステム時刻と一致し、かつ、月内日または曜日のいずれかが現在のシステム時刻と一致すれば、コマンドが実行されます。

それぞれのフィールドには以下の内容を指定できます。

表 8.1 crontab ファイルのフィールド

フィールド	指定可能な値
分	0-59
時	0-23
月内日	1-31
月	1-12(もしくは名前[1])
曜日	0-7(0 と 7 は日曜日、もしくは名前[2])

[1]「月」フィールドには、月の名前の最初の 3 文字を使用することができます。(例.Jan)

[2]「曜日」フィールドには、曜日の名前の最初の 3 文字を使用することができます。(例.Sun)

各フィールドには、アスタリスク(*)も指定できます。アスタリスクを指定した場合、そのフィールドが取りうるすべての値を指定したことになります。また、上記の指定可能な値以外にも高度な設定が可能です。詳細は **man 1 crontab** や **man 5 crontab** を参照してください。

Armadillo-400 シリーズの標準イメージでは、crond はユーザーランドのルートファイルシステムには含まれていますが、自動起動するようになっていません。そのため、crontab ファイルも含まれていません。Atmark Dist で作成するルートファイルシステムに crontab ファイルを追加し、crond を自動起動するには、以下のような手順でおこないます。

1. プロダクトディレクトリに crontab ファイルを追加する。
2. crontab ファイルをルートファイルシステムに含めるよう、プロダクトディレクトリ以下の Makefile を修正する。
3. crond を起動する初期化スクリプトをプロダクトディレクトリに追加する。
4. crond を起動する初期化スクリプトを自動実行するよう、プロダクトディレクトリ以下の Makefile を修正する。

まず、Atmark Dist のプロダクトディレクトリに、crontab を追加します。

Armadillo の標準イメージに含まれる crond は、BusyBox のものです。BusyBox の crond は、/var/spool/cron/crontabs/*[ユーザー名]*を crontab として読み込みます。

このファイルは、Armadillo-420 と Armadillo-440 の標準イメージにはありませんので、新しく追加する必要があります。root ユーザー用の crontab ファイルは、プロダクトディレクトリ以下の var/spool/cron/crontabs/root というファイル名で追加します。

crontab ファイルをルートファイルシステムイメージに追加するには、プロダクトディレクトリ以下の Makefile を修正する必要があります。具体的な変更方法については「7.6.4. ファイルの追加、変更」を参照してください。

crond を起動する初期化スクリプトは、以下のようになります。この起動スクリプトをプロダクトディレクトリ以下の etc/init.d/crond というファイル名で追加します。

```
#!/bin/sh

. /etc/init.d/functions

PATH=/bin:/sbin:/usr/bin:/usr/sbin

echo -n "Starting crond: "
```

```
crond
check_status
```

図 8.5 crond を起動する初期化スクリプト

「図 8.5. crond を起動する初期化スクリプト」が起動時に実行されるように Makefile を修正する方法は、「8.1.2. 初期化スクリプトで自動実行する(恒久的な方法)」を参照してください。

8.3. 不具合発生時の自動再起動

一度起動したシステムがいつまでも元気に動き続けるという保証はありません。システムが正常に動かなくなる原因には、ソフトウェアのバグやハードウェアの故障などが考えられます。

システムは停止することがある、ということを前提として、停止した場合の対処についても、システム設計時に考慮しておかなければなりません。ここでは、問題が発生した場合、自動的に再起動する方法について説明します。

8.3.1. init によるプロセスの自動再起動

「8.1.3. inittab で自動実行する」で紹介したように、inittab の respawn アクションによって起動されたプロセスがなんらかの理由により意図せず終了してしまった場合、init が自動的にそのプロセスを再起動します。

プロセスで起動時や一定時間毎にログを出力しておけば、いつ頃異常終了したかを特定できるので、障害解析時に重要な情報となります。

8.3.2. ウォッチドッグタイマーによるシステムの再起動

アプリケーションプログラムのプロセスが異常終了したのであれば、init によってそのことを検出し、再起動することができます。しかし、カーネルがハングアップ(停止)してしまった場合、ソフトウェア的にそのことを検出する手段はありません。

システム全体がハングアップしてしまった場合、そのことを検出するための仕組みとして、ウォッチドッグタイマーがあります。

ウォッチドッグタイマーは、有効にされると内部のタイマーのカウントを開始します。システム側は、正常に動作している間はウォッチドッグタイマーに対して、定期的に信号を送ります[6]。ウォッチドッグタイマーは、信号を受け取るとタイマーのカウントを最初からやり直します。もし、システム側に問題が発生し、信号を送ることができなくなったら、ウォッチドッグタイマーがタイムアウトし、システムのリセットをおこないます。

Armadillo-400 シリーズでは、標準で i.MX25 が持つハードウェアウォッチドッグタイマーが有効になっています。

Hermit-At ブートローダーによって、ウォッチドッグタイマーのタイムアウト時間が 10 秒に設定されてから、Linux カーネルが起動されます。カーネルは、自動でウォッチドッグタイマーをキックします。

もし、何らかの要因でカーネルがハングアップしてウォッチドッグタイマーをキックできなくなりタイムアウトが発生すると、システムが再起動します。

[6] この操作を、ウォッチドッグタイマーをキックする、撫でる、起こす、などと表現します。

Linux カーネルに手を入れるようなことがなければ、通常、カーネルのハングアップに気を配る必要はありませんが、このような仕組みが Armadillo-400 シリーズには組み込まれていることを覚えておいてください。

8.4. ログ管理

Linux システムでは、ログの管理は syslog でおこなうことが一般的です。

各アプリケーションプログラムは、**logger** コマンドや C 言語の syslog 関数で、ログ記録用プログラムである syslog デーモンにメッセージを送ります。syslog デーモンは、送られてきたメッセージをファイルに記録したり、別サーバーへの転送、ログファイルのローテーションなど、ログの管理を一括して行います。

syslog デーモンには、オリジナルの syslogd の他に、いくつかのバリエーションがあります。

Atmark Dist で作成したユーザーランドの場合、syslog デーモンとして BusyBox の syslogd を使用します。

BusyBox の syslogd は機能が基本的なものに限られているため、設定ファイルを持たず、設定はすべてコマンドラインオプションで指定します。

Armadillo-400 シリーズの標準設定では、初期化スクリプトの/etc/rc.d/S10syslogd(/etc/init.d/syslogd へのシンボリックリンク)で syslogd を起動しています。ログは、/var/log/messages へ書き込まれます。

Debian GNU/Linux 5.0(コードネーム "lenny")では、rsyslogd が標準です。rsyslogd の設定方法については、「**man rsyslogd**」や「**man rsyslog.conf**」を参照してください。

8.4.1. ログファイルのローテーション

ログファイルにログを書き込み続けると、ファイルサイズがどんどん大きくなり、いずれストレージの限界に達してしまいます。このような事態を避けるため、通常、ログファイルのローテーションをおこないます。ログファイルのローテーションは、一定の期間(1 日や 1 週間など)や、一定のサイズごとに行います。

BusyBox の syslogd では、サイズ毎のローテーションをサポートしています。

ログのローテートに関連するオプションは、-s *SIZE* と-b *NUM* です。*SIZE* [KB]になる前にログファイルをローテートします。-s オプションを指定しない場合の *SIZE* のデフォルト値は 200[KB]です。また、ローテートされたファイルを NUM 個分保持します。-b オプションを指定しない場合のデフォルト値は、1 です。

Armadillo-400 シリーズの標準設定では、-s と-b オプションを指定していないので、200[KB]になる前にローテートが行われ、過去のログファイルを 1 個だけ保持します。最新のログは/var/log/messages に書き込まれ、一つ過去のログファイルは/var/log/messages.0 となります。

8.4.2. ログをリモートサーバーに送る

BusyBox の syslogd は、別のサーバーで動作している syslog デーモンへメッセージを転送できます。

メッセージの転送も、コマンドラインオプションで指定します。メッセージの転送を指定するオプションは、-R *HOST[:PORT]* です。HOST には、転送先サーバーの IP アドレスかホスト名を指定します。PORT には、転送先サーバーのポート番号と転送に使用するプロトコルを指定します。デフォルトの値は、514/UDP で、UDP プロトコルで 514 番ポートに転送します。プロトコルには、TCP も指定できます。UDP は到達保証のないプロトコルですので、TCP を使うのが良いでしょう。

具体的な設定方法を、Armadillo から ATDE3 にログを転送する場合を例として説明します。ATDE3 の IP アドレスは 192.168.0.1 となっているとします。メッセージの待ち受けには、TCP の 514 番ポートを使用します。

まず、ATDE3 側の設定をおこないます。/etc/rsyslog.conf の UDP と TCP の待ち受けに関する設定を有効にします。/etc/rsyslog.conf の編集は、**sudo** コマンドを使い、特権ユーザー権限で行う必要がありますので、注意してください。

```
# provides UDP syslog reception
#$ModLoad imudp
#$UDPServerRun 514

# provides TCP syslog reception
#$ModLoad imtcp
#$InputTCPServerRun 514
```

を以下のように、修正します。

```
# provides UDP syslog reception
$ModLoad imudp
$UDPServerRun 514

# provides TCP syslog reception
$ModLoad imtcp
$InputTCPServerRun 514
```

設定を反映させるために、rsyslogd を再起動します。Debian GNU/Linux では、サーバーの起動、停止、再起動などは、/etc/init.d/ディレクトリ以下のスクリプトでおこないます。

```
[ATDE ~]$ sudo /etc/init.d/rsyslog restart
```

次に、Armadillo の syslogd を、-R オプション付きで起動します。既に syslogd が起動しているので、一度終了させてから再起動します。

```
[armadillo ~]$ ps | grep syslog
  334 root        484 S   syslogd -L
[armadillo ~]$ kill 334
[armadillo ~]$ syslogd -L -R 192.168.0.1:514/TCP
```

以上の設定を行うと、Armadillo で **logger** コマンドまたは C 言語の syslog 関数で送ったメッセージが、ATDE3 の rsyslod によって記録されます。

8.5. 外部ストレージのデータを守る

Armadillo では、USB メモリや SD/microSD カードなどの外部ストレージを使用することができます。これらのストレージデバイスは、大容量のデータを保存するために大変便利ですが、扱い方に注意が必要です。

本章では、外部ストレージのデータを安全に扱う方法を紹介します。

8.5.1. データがストレージに書き込まれたことを保証する

ストレージデバイスに対するデータの読み書き速度は、CPU の動作速度やメモリの読み書きの速度と比べると、非常に遅いです。そのため、Linux システムでは、色々な場所にバッファを設けてストレージとのデータのやりとりを効率化しています。プログラムでファイルに対して書き込みを行った後、そのデータがストレージデバイスに書き込まれたことを保証するためには、すべてのバッファされているデータをフラッシュ(書き出し)しなければなりません。

ストリームに対するフラッシュは、fflush ライブラリ関数でおこないます。しかし、fflush ライブラリ関数は、ユーザー空間でバッファされているデータをフラッシュするだけで、カーネル空間でバッファされているデータはフラッシュされませんので、これでは不十分です。

sync システムコールはカーネル空間のバッファをフラッシュします。sync システムコールは、デバイスへの書き込みが終了するまで返ってきません。そのため、このシステムコールを使用すると、データの書き込みを保証できるように思われます。しかし、最近のストレージデバイスは、デバイス側で大きなキャッシュを持っているため、sync システムコールから返ってきても、データは実際にはストレージデバイスに書き込まれていないかもしれません。**sync** コマンドでも同様です。

データがストレージに書き込まれたことを保証する最も確実な方法は、umount システムコールか **umount** コマンドで、デバイスをアンマウントすることです。アンマウントが完了した時点で、すべてのデータがストレージデバイスに書き込まれていることが保証されます。

8.5.2. 不意な電源断への対応

組み込みシステムでは、予期せぬタイミングで電源が遮断される状況への対応は必須ともいえるでしょう。特に、ストレージにデータを書き込みしている最中に電源断が発生すると、どうしてもデータの不整合が発生して、ファイルシステムが破壊されてしまいます。

このような状況への対処として、ジャーナリングファイルシステムを用いる方法と、ファイルシステムをリードオンリーでマウントする方法があります。

ジャーナリングファイルシステムとは、定期的にファイルシステムの状態(ジャーナル)を保存しておくことで、クラッシュが発生した場合に、ジャーナルを元に状態を復元できるファイルシステムです。ジャーナリングファイルシステムを用いると、フラッシュされていないデータがある状況で不意な電源断が発生した場合でも、少し前の正常な状態にファイルシステムを復元することができます。

Linux システムで標準的なジャーナリングファイルシステムは、ext3 ファイルシステムです。

ext3 ファイルシステムで、デバイスをフォーマットするには、**mke2fs** コマンドに-j オプションを付ける(Atmark Dist ベースのユーザーランドの場合)か、**mkfs.ext3** コマンド(ATDE3 の場合)を使用します。

ストレージへの書き込みが必要ない場合、デバイスをリードオンリー(読み込みのみ)でマウントするのが最も確実で安全な方法です。mount システムコールや **mount** コマンドには、リマウント(再マウント)オプションがあります。書き込みが必要ない場合は、リードオンリーでマウントしておき、書き込みが必要になった時だけ、読み書き可能でリマウントすることで、ファイルシステムが破壊される危険性を少なくすることができます。(もちろん、読み込みの必要すらないときは、アンマウントしておくのが一番安全です。)

ストレージに保存するデータの内容をよく吟味して、リードオンリーで済むデータがあるのならば、デバイスをリードオンリーのパーティションと書き込み可能なパーティションに分割するという方法もあります。

パーティションを分割するには、**fdisk** コマンドを使用します。

8.6. データ溢れを防ぐ

組み込みシステムで制限の厳しいリソースの一つが、ストレージ容量でしょう。

Armadillo では標準のルートファイルシステムとして RAM ディスクを使用しているので、ログを出力しているだけでもルートファイルシステムの容量を使い切ってしまう、という状況に陥りがちです。

本章では、ストレージの使用状況を調べ、使用できるストレージの容量を増やす方法について説明します。

8.6.1. ストレージのサイズと使用量を調べる

システムで使用しているストレージのサイズと、その使用量を調べるには **df** コマンドを使用します。

Armadillo-440 の標準イメージで **df** コマンドを実行すると、以下のように表示されます。-h オプションにより、人間に読みやすい形式にしています。

```
[armadillo ~]# df -h
Filesystem                Size      Used Available Use% Mounted on
/dev/ram0                26.4M     23.9M      1.2M  95% /
```

図 8.6 Armadillo-440 のストレージ使用量

この表示から、/dev/ram0(ルートファイルシステムに使用している RAM ディスク)のサイズは 26.9MByte で、23.9Mbyte 使用されており、残り 1.2MByte なので、使用率は 95%であることが読み取れます。

どのファイルやディレクトリが容量を使用しているのか調べるには、**du** コマンドを使用します。

```
[armadillo ~]# du -h -d 1 /
1.3M    /lost+found
956.0k  /sbin
215.0k  /etc
0       /sys
3.6M    /usr
2.8M    /home
28.0k   /var
1.0k    /mnt
4.0k    /dev
2.0k    /tmp
1.8M    /bin
13.2M   /lib
1.0k    /root
0       /proc
23.9M
```

図 8.7 Armadillo-440 の各ディレクトリのサイズ使用量

ルートディレクトリ直下のディレクトリの使用量を、人間に読みやすい形式で表示しています。全部で 23.9Mbyte 使用しているうち、半分以上の 13.2MByte は/lib ディレクトリが使用していることが分かります。

8.6.2. ルートファイルシステムの空き容量を増やす

Atmark Dist でルートファイルシステムを作成すると、ルートファイルシステムのサイズは、標準では必要最小限になるよう自動で調整されます。概ね、空き容量は 10%以下となります。

ルートファイルシステムのサイズを手動で設定することで、空き容量を増やすことができます。

Atmark Dist の menuconfig の「Userland Configuration → Vendor specific→ generate file-system option」を Auto から Manual にすることで、ルートファイルシステムのサイズを手動で設定できるようになります。

「Size of the image in blocks」で、ユーザーランドのサイズをブロック単位で指定します。1 ブロックは 1024Byte です。

「Maximum number of inodes」で、ファイルシステムに保存できるファイルやディレクトリの数を指定します。

自動でサイズ設定されたユーザーランドがどのようなパラメータを持っているかは、make 時のログから確認することができます。

一度ビルド済みの atmark-dist ディレクトリで、**make image** を実行することで、現在の設定を確認できます。以下は、Armadillo-440 の標準イメージをビルドしたときの設定です。

```
[ATDE ~]$ make image | grep genext2fs
/usr/bin/genext2fs --size-in-blocks 27241 --number-of-inodes 1280 --squash-uids --root /home/
atmark/atmark-dist/romfs --devtable ext2_devtable.txt /home/atmark/atmark-dist/images/romfs.img
```

図 8.8 ルートファイルシステムのブロック数と inode 数の確認

「--size-in-blocks」で指定されているオプションがブロック単位でのサイズで、「--number-of-inodes」で指定されているオプションが inode の数となります。この例では、それぞれ 27241 と 1280 となっています。手動でサイズを指定する場合は、これらの値を参考にしながら、設定してください。

例として、ルートファイルシステムのサイズを約 32MByte、inode 数 2048 に設定するには、以下のようにコンフィギュレーションをおこないます。

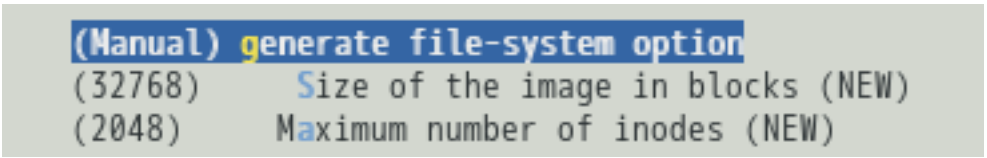


図 8.9 ルートファイルシステムサイズの手動設定

コンフィギュレーションを変更したあと、**make** または **make image** を実行すると、設定したサイズでイメージが生成されます。あとは、通常と同様に romfs.img.gz を Armadillo のユーザーランド領域に書き込んで再起動すると、ルートファイルシステムのサイズが約 32MByte になります。

```
[armadillo ~]# df -h
Filesystem                Size      Used Available Use% Mounted on
/dev/ram0                31.7M     24.1M      6.0M  80% /
```

図 8.10 手動で設定したルートファイルシステムのストレージ使用量

ルートファイルシステムの最大サイズ

ルートファイルシステムは RAM ディスクのサイズより大きくすることはできません。

RAM ディスクのサイズは、カーネルの設定で決まります。Armadillo の標準設定では、32MByte となっています。

RAM ディスクのサイズを増やすには、カーネルコンフィギュレーションの「Linux Kernel Configuration → Device Drivers → Block devices →Default RAM disk size (kbytes)」を変更してください。

8.6.3. 一時 RAM ファイルシステムを使用する

ログローテーションをおこないログファイルを保存するなど、使用するファイルサイズがあらかじめ計算できる場合は、ルートファイルシステムのサイズを変更して空き容量を増やすことで対応できます。しかし、Armadillo が FTP サーバーになっていて最大サイズの分からないファイルを受信する場合など、使用するサイズが計算できない場合は、いくらルートファイルシステムのサイズを大きくしておいても、容量不足の懸念が残ります。

そのような場合は、ルートファイルシステム以外に、別のファイルシステムを用意することで対応します。最大サイズが計算できないファイルをルートファイルシステムとは別のファイルシステムに保存することで、そのファイルシステムが容量不足になったとしても、システム全体が動作不能になる事態は避けることができます。

通常の PC やサーバーなどの Linux システムでは、HDD のパーテーションを分けることで、ファイルシステムを分割します。Armadillo-400 シリーズでは、microSD が使えますので、それを使用するのも良いでしょう。

microSD が使えない場合は、一時 RAM ファイルシステム(tmpfs)を使うことができます。tmpfs は、RAM ディスクと同様に、RAM の一部をストレージとして使用する機能です。

最大 1MByte の RAM を tmpfs に割り当て、/mnt ディレクトリにマウントするには、以下のようにします。

```
[armadillo ~]# mount -t tmpfs -o size=1m tmpfs /mnt/
```

図 8.11 tmpfs をマウントする

tmpfs でマウントした/mnt ディレクトリには、ルートファイルシステムにあるファイルとは別に、合計 1MByte までのファイルやディレクトリを置くことができます。

mount コマンドをオプションを指定せずに実行すると、現在マウントされているすべてのディレクトリを確認することができます。

```
[armadillo ~]# mount
/dev/ram0 on / type ext2 (rw)
proc on /proc type proc (rw)
usbfs on /proc/bus/usb type usbfs (rw)
sysfs on /sys type sysfs (rw)
ramfs on /home/ftp/pub type ramfs (rw)
tmpfs on /mnt type tmpfs (size=1m)
```

図 8.12 マウントされているディレクトリの確認

tmpfs をマウントする際のオプションについては、**man 8 mount** を参照してください。

ramfs

tmpfs とほぼ同じ機能を提供するものとして、ramfs もあります。Armadillo-400 シリーズの標準設定では、/home/ftp/pub ディレクトリが ramfs でマウントされています。

ramfs は最大サイズの指定ができないため、ファイルを追加していくと、いずれ RAM を使い切ってしまいます。実運用するシステムでは、ramfs ではなく最大サイズを指定した tmpfs を使用する方が良いでしょう。

8.7. イメージの自動アップデート

開発段階で、Armadillo のイメージをアップデートする方法には以下のものがあります。

1. Hermit-At ダウンローダーを使用して、シリアル経由で書き換える。
2. Hermit-At ブートローダーの tftpdl 機能を使用して、ネットワーク経由で書き換える。
3. **netflash** コマンドを使用して、ネットワーク経由で書き換える。
4. at-cgi のファームウェアアップデート機能を使用して、ネットワーク経由で書き換える。

これらの方法は、いずれも人手での操作を伴います。製品出荷後でもイメージアップデートできるような機能をつける場合、可能な限り簡単な手順でアップデートできることが望ましいところです。

本章では、なるべく人手を介さない自動アップデート機能の実現方法を紹介します。

8.7.1. USB メモリを使用してのアップデート

udevd を使用して、USB メモリを接続するだけでイメージのアップデートを自動で行う方法を紹介します。

例として、次のような動作を考えます。

1. USB ポートに USB メモリが接続されたら、以下の処理を行う。
    a. USB メモリのマウント。

b. USB メモリにカーネルまたはユーザーランドのイメージファイルがあれば、フラッシュメモリをアップデート。

2. USB メモリが抜かれ、フラッシュメモリのアップデートが正常に終了していたら、以下の処理を行う。

   a. 変更を反映するためにリブート。

USB メモリが接続されたことを検知して、特定の動作を行わせるには udevd の機能を使用します。

udevd は、/etc/udev/rules.d/ディレクトリに、どのようなデバイスが接続された時にどのような動作を行うか、というルールを記述したファイルを置いておくと、そのルールに従って処理をおこないます。udev ルールについての詳細は、**man 7 udev** を参照してください。

上下いずれかの USB ポートにマスストレージクラスの USB メモリが接続されたら、または、USB メモリが抜かれたら、/etc/config/usb_image_update.sh という名前のシェルスクリプトを実行するルールは、以下のようになります。

$KERNEL には、USB メモリが接続された時に/dev ディレクトリに作成される、「sd*[0-9]*」にマッチしたデバイスファイル名が入ります。通常、sda1 になるでしょう。

```
KERNEL=="sd*[0-9]*","BUS=="usb",ACTION=="add",RUN+="/etc/config/usb_image_update.sh start $KERNEL"
KERNEL=="sd*[0-9]*","BUS=="usb",ACTION=="remove",RUN+="/etc/config/usb_image_update.sh stop"
```

図 8.13 USB メモリが接続された時にスクリプトを実行する udev ルール (z99_usb_image_update.rules)

上記の内容を、/etc/udev/rules.d/z99_usb_image_update.rules という名前で保存します。

設定内容をすぐに反映するには、udevd を一度終了して再起動します。

```
[armadillo ~]# killall udevd && udevd --daemon
```

図 8.14 udevd の再起動をおこなうコマンド

udevd から実行される usb_image_update.sh は、次のようになります。

USB メモリが接続されたときは、第一引数に start という文字列が指定されるので、start_action 関数を実行します。/dev/$KERNEL という名前のデバイスファイルを/mnt/auto_image_update ディレクトリにマウントし、ファイル名が linux*.bin.gz または romfs.*.bin.gz にマッチするイメージファイルでフラッシュメモリをアップデートします。処理中は、red LED を点滅させます。

USB メモリが抜かれた時には、第一引数に stop という文字列が指定されるので、stop_action 関数を実行します。フラッシュメモリが正常に更新されていれば、**reboot** コマンドを実行し、再起動します。

```
#!/bin/sh

LOCKDIR='/var/lock/auto_image_update'
SUCCESSDIR='/var/lock/auto_image_update_success'
MOUNTDIR='/mnt/auto_image_update'
KERNEL_IMG_PTN='linux*.bin.gz'
USERLAND_IMG_PTN='romfs*.img.gz'
```

```
ACTION=$1
DEVICE=$2

log()
{
    logger -p user.$1 -t "$0[$$]" -- "$2"
}

die()
{
    log err "$1"
    if [ "$(mount | grep /mnt/auto_image_update | grep -v grep)" ]; then
        umount $MOUNTDIR
    fi
    if [ -e $MOUNTDIR ]; then
        rmdir $MOUNTDIR
    fi
    ledctrl red blink_off
    rmdir $LOCKDIR
    exit 1
}

start_action()
{
    mkdir -p $LOCKDIR || die "making LOCKDIR($LOCKDIR) is failed"

    if [ -e $SUCCESSDIR ]; then
        die "SUCCESSDIR($SUCCESSDIR) is already exist"
    fi

    ledctrl red blink_on 100

    if [ ! -e $MOUNTDIR ]; then
        mkdir $MOUNTDIR
    fi

    MOUNT_SUCCESS=no
    for fs in vfat auto; do
        mount -t $fs -o ro /dev/$DEVICE $MOUNTDIR > /dev/null 2>&1
        if [ $? = 0 ]; then
            MOUNT_SUCCESS=yes
            break;
        fi
    done
    if [ "$MOUNT_SUCCESS" != "yes" ]; then
        die "failed to mount /dev/$DEVICE to $MOUNTDIR"
    fi

    KERNEL_IMG_FILE=$(ls $MOUNTDIR/$KERNEL_IMG_PTN 2>/dev/null | tail -1)
    USERLAND_IMG_FILE=$(ls $MOUNTDIR/$USERLAND_IMG_PTN 2>/dev/null | tail -1)

    if [ -z "$KERNEL_IMG_FILE" ] && [ -z "$USERLAND_IMG_FILE" ]; then
        die "no image file found"
    fi

    if [ "$KERNEL_IMG_FILE" ]; then
        log info "update kernel image: $KERNEL_IMG_FILE"
```

```
        netflash -knubsr /dev/flash/kernel $KERNEL_IMG_FILE > /dev/null 2>&1
        if [ $? != 0 ]; then
            die "update kernel image failed"
        fi
    fi

    if [ "$USERLAND_IMG_FILE" ]; then
        log info "update userland image: $USERLAND_IMG_FILE"
        netflash -knubsr /dev/flash/userland $USERLAND_IMG_FILE > /dev/null 2>&1
        if [ $? != 0 ]; then
            die "update userland image failed"
        fi
    fi

    umount $MOUNTDIR || die "umount $MOUNTDIR failed"
    rmdir $MOUNTDIR || die "rmdir $MOUNTDIR failed"

    mkdir $SUCCESSDIR || die "mkdir $SUCCESSDIR"

    ledctrl red blink_off

    rmdir $LOCKDIR || die "rmdir $LOCKDIR"

    exit 0
}

stop_action()
{
    if [ -e $LOCKDIR ]; then
        log err "LOCKDIR($LOCKDIR) is exist"
        exit 1
    fi

    if [ ! -e $SUCCESSDIR ]; then
        log err "SUCCESSDIR($SUCCESSDIR) is not exist"
        exit 1
    fi

    rmdir $SUCCESSDIR

    log info "reboot"
    reboot

    exit 0
}

log info "$ACTION"
case $ACTION in
    start)
        start_action
        ;;
    stop)
        stop_action
        ;;
    *)
    log err "unknown action"
```

```
        ;;
esac
```

図 8.15 USB メモリ内のイメージファイルでフラッシュメモリをアップデートするスクリプト (usb_image_update.sh)

注意: セキュリティホール

USB メモリを使用してのアップデート機能は、第三者が USB ポートに触れることができる環境下では、その第三者によってイメージを書き換えられる危険性が伴います。

本方法を適用できる状況であるか否かについては、十分ご検討ください。

8.7.2. Web サーバーを使用してのアップデート

Web サーバにイメージファイルを置くと、自動でイメージをアップデートする方法を紹介します。

例として、次のような動作を考えます。

1. 毎日定時に特定のシェルスクリプトを実行する。
2. シェルスクリプトでは、以下の処理をおこなう。
   a. Web サーバーにアクセスし、更新すべきイメージファイルがあるか確認する。
   b. ファイルがある場合取得し、フラッシュメモリをアップデート。
   c. 正常に書き込みが完了したら、変更を反映するためにリブート。

毎日定時にスクリプトを実行する処理は、cron に任せることにします。cron の設定は、「8.2.1. cron で定期実行する」を参照してください。

毎日 4:00 に処理を実行する場合の crontab の設定は以下のようになるでしょう。イメージのアップデートを行うスクリプトは、/etc/config/web_image_update.sh という名前だとします。

```
* 4 * * * /etc/config/web_image_update.sh
```

図 8.16 毎日 4:00 に処理を実行する crontab

cron から実行される web_image_update.sh は、次のようになります。**wget** コマンドでイメージファイルを取得し、**netflash** コマンドでフラッシュメモリを書き換えます。SERVER_IP_ADDRESS(Web サーバーの IP アドレス)、PROTCOL(http または https)、USER_NAME(認証を行う場合のユーザー名)、PASSWORD(認証を行う場合のパスワード)、KERNEL_IMG_PATH(カーネルイメージのパス)、USERLAND_IMG_PATH(ユーザーランドいめーのパス)の各変数は、環境に合わせて書き換えてください。

wget コマンドを使用することで、http プロトコルだけでなく、https プロトコルも使用することができます。また、USER_NAME と PASSWORD 変数を指定することで、basic 認証または digest 認証を利用できます。

```
#!/bin/sh

SERVER_IP_ADDRESS=172.16.2.101
PROTCOL='http'
USER_NAME=''
PASSWORD=''
KERNEL_IMG_PATH='linux.bin.gz'
USERLAND_IMG_PATH='romfs.img.gz'

KERNEL_IMG_URL=${PROTCOL}://${SERVER_IP_ADDRESS}/${KERNEL_IMG_PATH}
USERLAND_IMG_URL=${PROTCOL}://${SERVER_IP_ADDRESS}/${USERLAND_IMG_PATH}
TMPFS_SIZE='32m'
LOCKDIR='/var/lock/auto_image_update'
TMPDIR=''
FILE_IS_UPDATED=false

log()
{
    logger -p user.$1 -t "$0[$$]" -- "$2"
}

die()
{
    log err "$1"

    if [ "$TMPDIR" ]; then
        cd
        umount $TMPDIR
        rm -rf $TMPDIR
    fi

    ledctrl red blink_off
    rmdir $LOCKDIR

    exit 1
}

update_image()
{
    region=$1
    img_url=$2

    wget_ops=""
    if [ "$USER_NAME" ]; then
        wget_ops="--http-user=$USER_NAME"
        if [ "$PASSWORD" ]; then
            wget_ops="$wget_ops --http-password=$PASSWORD"
        fi
    fi

    wget $wget_ops $img_url > /dev/null 2>&1
    if [ $? = 0 ]; then
        log info "update $region image: $(basename $img_url)"
        netflash -knubsr /dev/flash/$region $(basename $img_url) > /dev/null 2>&1
        if [ $? = 0 ]; then
            FILE_IS_UPDATED=true
        else
```

```
            die "update $region image failed"
        fi
        rm -f $(basename $img_url)
    fi
}

mkdir -p $LOCKDIR || die "making LOCKDIR($LOCKDIR) is failed"

ledctrl red blink_on 100

TMPDIR=$(mktemp -d /tmp/web_image_update.XXXXXXXX)
if [ -z "$TMPDIR" ]; then
    die "mktemp failed"
fi

mount -t tmpfs -o size=$TMPFS_SIZE tmpfs $TMPDIR
cd $TMPDIR

if [ "$KERNEL_IMG_PATH" ]; then
    update_image kernel $KERNEL_IMG_URL
fi
if [ "$USERLAND_IMG_PATH" ]; then
    update_image userland $USERLAND_IMG_URL
fi

cd
umount $TMPDIR
rm -rf $TMPDIR


ledctrl red blink_off

rmdir $LOCKDIR

if [ "$FILE_IS_UPDATED" = true ]; then
    log info "reboot"
    reboot
fi
```

図 8.17 Web サーバーにあるイメージファイルでフラッシュメモリをアップデートするスクリプト(web_image_update.sh)

改訂履歴

バージョン	年月日	改訂内容
1.0.0	2010/10/19	・初版リリース
1.1.0	2010/11/18	・全般的に誤記および用語の修正 ・しおりにすべての章が表示されるよう修正 ・「1. はじめに」コマンド入力例に Linux システムの場合と保守モードの場合を追記 ・「2. 開発環境の整備と運用」及び「6. C 言語による実践プログラミング」の章立てを変更 ・「6. C 言語による実践プログラミング」makefile の「+=」についての説明を追記 ・「6. C 言語による実践プログラミング」makefile の条件文に関する説明を追記
2.0.0	2010/12/24	・用語の修正 ・第 3 部リリース

Armadillo 実践開発ガイド
Version 2.0.0
2010/12/24

株式会社アットマークテクノ
060-0035 札幌市中央区北 5 条東 2 丁目 AFT ビル 6F TEL 011-207-6550 FAX 011-207-6570