Armadillo 標準ガイド ソフトウェア開発編

～組み込み Linux の導入から製品化まで～

Version 1.0.1
2022/08/29

株式会社アットマークテクノ [https://www.atmark-techno.com]

Armadillo サイト [https://armadillo.atmark-techno.com]

Armadillo 標準ガイド ソフトウェア開発編: 〜組み込み Linux の導入から製品化まで〜

株式会社アットマークテクノ

Version 1.0.1
2022/08/29

目次

図目次

Armadillo 標準ガイド ソフトウェア開発編

表目次

1. はじめに

「Armadillo 入門編」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。本書「ソフトウェア開発編」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、「3. 開発環境の整備と運用」で、開発環境 ATDE7 について「Armadillo 入門編」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。

「4. 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。

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. Python 言語による実践プログラミング」は、スクリプト言語である Python によるサンプルプログラムを紹介します。Linux システムや開発環境に依存した独特な部分がないアプリケーションであれば、Python を使用したほうが開発効率を上げられる可能性があります。

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、「Armadillo 入門編」から読むことをお勧めします。

1.2. 表記について

1.2.1. コマンド入力例

本書に記載されているコマンドの入力例は、表示されているプロンプトによって、それぞれに対応した実行環境を想定して書かれています。「 / 」の部分はカレントディレクトリによって異なります。各ユーザのホームディレクトリは「 ~ 」で表します。

表 1.1 表示プロンプトと実行環境の関係

プロンプト	コマンドの実行環境
[PC /]#	作業用 PC(Linux)の root ユーザで実行
[PC /]$	作業用 PC(Linux)の一般ユーザで実行
[ATDE /]#	ATDE 上の root ユーザで実行

プロンプト	コマンドの実行環境
[ATDE /]$	ATDE 上の一般ユーザで実行
[armadillo /]#	Armadillo 上 Linux の root ユーザで実行
[armadillo /]$	Armadillo 上 Linux の一般ユーザで実行
⇒	Armadillo 上 U-Boot の保守モードで実行

コマンド中で、変更の可能性のあるものや、環境により異なるものに関しては以下のように表記します。適宜読み替えて入力してください。

表 1.2 コマンド入力例での省略表記

表記	説明
[version]	ファイルのバージョン番号

1.2.2. アイコン

本書では以下のようにアイコンを使用しています。

注意事項を記載します。

役に立つ情報を記載します。

用語の説明や補足的な説明を記載します。

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、 https://download.atmark-techno.com/armadillo-guide-std/sample/ 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 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]https://armadillo.atmark-techno.com

Armadillo サイトでも知りたい情報が見つからない場合は、「Armadillo フォーラム」[3]で質問してみてください。Armadillo フォーラムは、アットマークテクノユーザーズサイト内に設けられた、Armadillo ブランド製品での開発や周辺技術に関する話題を扱うユーザー向けコミュニティです。Armadillo に関する技術的な話題なら何でも投稿できます。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。

フォーラムに参加するときの心構え

Armadillo フォーラムには、その前身となったメーリングリストから引き続き、数百人のユーザーが参加しています。また、フォーラムへ投稿した内容は Web 上で誰でも閲覧・検索可能になるほか、通知を希望しているユーザーにメールで送信されます。

フォーラムには多くの人が参加しており、投稿内容は多くの人の目に触れますので、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一定の配慮はすべきです。技術系コミュニティに不慣れな方は、投稿する前に「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」[4] をご一読されることをお勧めします。メーリングリストに投稿するときの心構えや、適切な回答を得るために有用なテクニックが分かりやすく紹介されています。メーリングリストとフォーラムの違いはあれど、基本的な考え方は共通しており、とても参考になります。

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。

1.5. お問い合わせ先

本書に関するご意見やご質問は、Armadillo フォーラム[3]にご連絡ください。

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

図 1.1 クリエイティブコモンズライセンス

[3]https://armadillo.atmark-techno.com/forum/armadillo
[4]結城浩氏によるサイトより http://www.hyuki.com/writing/techask.html

1.8. 謝辞

Armadillo 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の成果によってなりたっています。この場を借りて感謝の意を表します。

2. 注意事項

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。

3. 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一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 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。

Armadillo-600 シリーズ用の標準開発環境は、ATDE7 となっています。ATDE7 のベースは Debian GNU/Linux 9 (コードネーム "stretch")です。ATDE7 のインストール及び基本的な設定方法については、Armadillo 入門編「開発環境の構築と基本操作」を参照してください。

この章では、Armadillo 入門編で触れなかった ATDE7 の運用方法について説明をおこないます。

3.1. ATDE7 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 ATDE7 へインストールする方法について説明します。

3.1.1. 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。

- Debian のパッケージ検索[1]から検索する。
- apt 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。

apt-get と aptitude

apt と同様の機能を持つコマンドに apt-get と aptitude がありますが、現在は apt の使用が推奨されています。

3.1.1.1. 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 のパッケージ検索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。

[1]https://packages.debian.org/ja/

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。

ATDE7 では、2019 年 4 月現在の安定版(stable)である Debian GNU/Linux 9 (コードネーム "stretch")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。ATDE7 では、このアーキテクチャを使用します。

ARM CPU 用には、armel[2]と armhf[3]の二つのアーキテクチャがありますが、Armadillo-600 シリーズは armhf アーキテクチャ用のパッケージを使用します。

3.1.1.2. apt コマンドで検索する

apt 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。

```
[ATDE ~]$ apt search coreutils
ソート中... 完了
全文検索... 完了
bsdmainutils/oldstable,now 9.0.12+nmu1 i386 [インストール済み]
  FreeBSD 由来のユーティリティ集

coreutils/oldstable,now 8.26-3 i386 [インストール済み]
  GNU コアユーティリティ

gotail/oldstable 1.0.0+git20160415.b294095-3+b2 i386
  Go implementation of tail

libsemanage1/oldstable,now 2.6-2 i386 [インストール済み]
  SELinux ポリシー管理ライブラリ

mktemp/oldstable,oldstable 8.26-3 all
  coreutils mktemp 過渡的なパッケージ

policycoreutils/oldstable 2.6-3 i386
  SELinux core policy utilities

policycoreutils-dev/oldstable 2.6-3 i386
```

[2]Debian/armel はハードウェア浮動小数点演算ユニット (FPU) をサポートしていない、古い 32 ビットの ARM プロセッサを対象としています。
[3]Debian/armhf は最低でも ARMv7 アーキテクチャに ARM ベクトル浮動小数点演算仕様のバージョン 3 (VFPv3) を実装した新しい 32 ビットの ARM プロセッサを対象としています。

```
  SELinux core policy utilities (development utilities)

policycoreutils-gui/oldstable,oldstable 2.6-3 all
  SELinux core policy utilities (graphical utilities)

policycoreutils-python-utils/oldstable 2.6-3 i386
  SELinux core policy utilities (Python utilities)

policycoreutils-sandbox/oldstable 2.6-3 i386
  SELinux core policy utilities (graphical sandboxes)

progress/oldstable 0.13.1-1 i386
  Coreutils Progress Viewer (formerly known as 'cv')

python3-sepolgen/oldstable,oldstable 2.6-3 all
  Python3 module used in SELinux policy generation

realpath/oldstable,oldstable 8.26-3 all
  coreutils realpath 過渡的なパッケージ
```

図 3.1 apt 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。すでにインストール済みのパッケージについては一番右に[インストール済み]と表示されます。

3.1.1.3. 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

ATDE7 にはインストールされていないソフトウェアが、Debian GNU/Linux 9.0 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 3.2 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。

3.1.2. 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 install でパッケージをインストールできます。apt 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、SSH サーバーの openssh-server パッケージをインストールする場合、「図 3.3. apt install によるパッケージのインストール」のようになります。

```
[ATDE ~]$ sudo apt install openssh-server
[sudo] atmark のパスワード:
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自動でインストールされましたが、もう必要とされていません:
  icedtea-netx icedtea-netx-common
これを削除するには 'sudo apt autoremove' を利用してください。
以下の追加パッケージがインストールされます:
  openssh-sftp-server
提案パッケージ:
  molly-guard monkeysphere rssh ssh-askpass ufw
以下のパッケージが新たにインストールされます:
  openssh-server openssh-sftp-server
アップグレード: 0 個、新規インストール: 2 個、削除: 0 個、保留: 0 個。
419 kB 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 1,189 kB のディスク容量が消費されます。
続行しますか? [Y/n]
取得:1 http://ftp.jp.debian.org/debian stretch/main i386 openssh-sftp-server i386                ⏎
1:7.4p1-10+deb9u7 [45.0 kB]
取得:2 http://ftp.jp.debian.org/debian stretch/main i386 openssh-server i386 1:7.4p1-10+deb9u7   ⏎
[374 kB]
419 kB を 0秒 で取得しました (644 kB/s)
パッケージを事前設定しています ...
以前に未選択のパッケージ openssh-sftp-server 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 185232 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。)
.../openssh-sftp-server_1%3a7.4p1-10+deb9u7_i386.deb を展開する準備をしています ...
openssh-sftp-server (1:7.4p1-10+deb9u7) を展開しています...
以前に未選択のパッケージ openssh-server を選択しています。
.../openssh-server_1%3a7.4p1-10+deb9u7_i386.deb を展開する準備をしています ...
openssh-server (1:7.4p1-10+deb9u7) を展開しています...
openssh-sftp-server (1:7.4p1-10+deb9u7) を設定しています ...
systemd (232-25+deb9u12) のトリガを処理しています ...
man-db (2.7.6.1-2) のトリガを処理しています ...
openssh-server (1:7.4p1-10+deb9u7) を設定しています ...

Creating config file /etc/ssh/sshd_config with new version
Creating SSH2 RSA key; this may take some time ...
2048 SHA256:IeCEsCfh2kQ9vClX2pLTY9BmWxRxc9ki4PLS/BM43CM root@atde7 (RSA)
Creating SSH2 ECDSA key; this may take some time ...
256 SHA256:PjOjXxOksmocHq4pSn6ufXQ+BKdL8hnIDk/iiPpqCFs root@atde7 (ECDSA)
Creating SSH2 ED25519 key; this may take some time ...
256 SHA256:YhoEp0UJblQR3eIEiZmpdUP9nsPSHhDKajfgrHIdZOw root@atde7 (ED25519)
Created symlink /etc/systemd/system/sshd.service → /lib/systemd/system/ssh.service.
Created symlink /etc/systemd/system/multi-user.target.wants/ssh.service → /lib/systemd/system/  ⏎
ssh.service.
systemd (232-25+deb9u12) のトリガを処理しています ...
```

図 3.3 apt install によるパッケージのインストール

apt 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 を参照してください。

3.2. ATDE7 にクロス開発用ライブラリをインストールする

ここでは、クロス開発用ライブラリパッケージのインストール方法を紹介します。ATDE7 に、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手順は難しくありません。

まずは、次のコマンドを実行します。

```
[ATDE ~]$ sudo dpkg --add-architecture armhf
```

図 3.4 armhf アーキテクチャを dpkg の architecture リストに追加する

このコマンドを実行すると、armhf アーキテクチャ用のパッケージを--force-architecture を使用せずにインストールできるようになります。

次のコマンドを実行すると、architecture リストを表示できます。「図 3.4. armhf アーキテクチャを dpkg の architecture リストに追加する」がうまくいっていれば、システムコンソールに armhf が表示されます。

```
[ATDE ~]$ dpkg --print-foreign-architectures
armhf
```

図 3.5 dpkg の architecture リストを表示する

以上で準備は完了です。

例として、armhf 用の libboost の開発用パッケージ(libboost-system-dev:armhf)をインストールしてみます。

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。

```
[ATDE ~]$ sudo apt update && sudo apt install libboost-system-dev:armhf
無視:1 http://download.atmark-techno.com/debian stretch InRelease
ヒット:2 http://download.atmark-techno.com/debian stretch Release
ヒット:4 http://security.debian.org/debian-security stretch/updates InRelease
無視:5 http://ftp.jp.debian.org/debian stretch InRelease
```

```
ヒット:6 http://ftp.jp.debian.org/debian stretch-updates InRelease
ヒット:7 http://ftp.jp.debian.org/debian stretch Release
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
パッケージはすべて最新です。
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
以下の追加パッケージがインストールされます:
  libasan3:armhf libatomic1:armhf libboost-system1.62-dev:armhf
  libboost-system1.62.0:armhf libboost1.62-dev:armhf libgcc-6-dev:armhf
  libgomp1:armhf libstdc++-6-dev:armhf libubsan0:armhf
提案パッケージ:
  libboost1.62-doc:armhf libboost-atomic1.62-dev:armhf
  libboost-chrono1.62-dev:armhf libboost-context1.62-dev:armhf
  libboost-coroutine1.62-dev:armhf libboost-date-time1.62-dev:armhf
  libboost-exception1.62-dev:armhf libboost-fiber1.62-dev:armhf
  libboost-filesystem1.62-dev:armhf libboost-graph1.62-dev:armhf
  libboost-graph-parallel1.62-dev:armhf libboost-iostreams1.62-dev:armhf
  libboost-locale1.62-dev:armhf libboost-log1.62-dev:armhf
  libboost-math1.62-dev:armhf libboost-mpi1.62-dev:armhf
  libboost-mpi-python1.62-dev:armhf libboost-program-options1.62-dev:armhf
  libboost-python1.62-dev:armhf libboost-random1.62-dev:armhf
  libboost-regex1.62-dev:armhf libboost-serialization1.62-dev:armhf
  libboost-signals1.62-dev:armhf libboost-test1.62-dev:armhf
  libboost-thread1.62-dev:armhf libboost-timer1.62-dev:armhf
  libboost-type-erasure1.62-dev:armhf libboost-wave1.62-dev:armhf
  libboost1.62-tools-dev:armhf libmpfrc++-dev:armhf libntl-dev:armhf
  libstdc++-6-doc:armhf
以下のパッケージが新たにインストールされます:
  libasan3:armhf libatomic1:armhf libboost-system-dev:armhf
  libboost-system1.62-dev:armhf libboost-system1.62.0:armhf
  libboost1.62-dev:armhf libgcc-6-dev:armhf libgomp1:armhf
  libstdc++-6-dev:armhf libubsan0:armhf
アップグレード: 0 個、新規インストール: 10 個、削除: 0 個、保留: 0 個。
9,587 kB 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 137 MB のディスク容量が消費されます。
続行しますか? [Y/n]
:
:
:
```

図 3.6 armhf 用の libboost の開発用パッケージのインストール

「図 3.6. armhf 用の libboost の開発用パッケージのインストール」のログを見ると分かりますが、libgcc1:armhf や libgcc-6-dev:armhf など、libboost が依存しているパッケージも一緒にインストールされます。このため、アプリケーションプログラムの開発者がパッケージの依存関係を気にする必要はありません。

libboost を利用したアプリケーションプログラムを Armadillo 上で動作させる際は、libboost を Armadillo にインストールします。もちろん、この時もパッケージの依存関係は自動的に処理され、libboost と一緒に libboost の動作に必要なパッケージがインストールされます。

```
[armadillo ~]# apt update && apt install l
ibboost-system1.62.0
```

```
Ign:1 http://download.atmark-techno.com/debian stretch InRelease
Get:2 http://download.atmark-techno.com/debian stretch Release [8872 B]
Get:3 http://download.atmark-techno.com/debian stretch Release.gpg [833 B]
Ign:4 http://ftp.jp.debian.org/debian stretch InRelease
Get:5 http://security.debian.org stretch/updates InRelease [94.3 kB]
Hit:6 http://ftp.jp.debian.org/debian stretch Release
Get:7 http://download.atmark-techno.com/debian stretch/main Sources [7365 B]
Get:8 http://download.atmark-techno.com/debian stretch/non-free Sources [2018 B]
Get:9 http://download.atmark-techno.com/debian stretch/main armhf Packages [9711
 B]
Get:10 http://download.atmark-techno.com/debian stretch/non-free armhf Packages
[15.4 kB]
Get:12 http://security.debian.org stretch/updates/main Sources [211 kB]
Get:13 http://security.debian.org stretch/updates/main armhf Packages [503 kB]
Get:14 http://security.debian.org stretch/updates/main Translation-en [232 kB]
Fetched 1085 kB in 7s (153 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
8 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  libboost-system1.62.0
0 upgraded, 1 newly installed, 0 to remove and 8 not upgraded.
Need to get 32.1 kB of archives.
After this operation, 56.3 kB of additional disk space will be used.
Get:1 http://ftp.jp.debian.org/debian stretch/main armhf libboost-system1.62.0 a
rmhf 1.62.0+dfsg-4 [32.1 kB]
Fetched 32.1 kB in 0s (231 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package libboost-system1.62.0:armhf.
(Reading database ... 41508 files and directories currently installed.)
Preparing to unpack .../libboost-system1.62.0_1.62.0+dfsg-4_armhf.deb ...
Unpacking libboost-system1.62.0:armhf (1.62.0+dfsg-4) ...
Processing triggers for libc-bin (2.24-11+deb9u4) ...
Setting up libboost-system1.62.0:armhf (1.62.0+dfsg-4) ...
Processing triggers for libc-bin (2.24-11+deb9u4) ...
```

図 3.7 Armadillo への libboost のインストール

4. 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。

4.1. 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) to standard output.

FILE の指定がなかったり, - であった場合, 標準入力から読み込みます.

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

図 4.1 --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 ~]$ which --help
Illegal option --
Usage: /usr/bin/which [-a] args
```

図 4.2 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 4.3 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。ATDE7 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

ATDE7 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。

4.2. 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
Debian-exim
sbuild
sshd
```

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、「4.3.4. ファイルの所有権とパーミッション」で説明します。

4.2.1. 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。

ATDE7 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ー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 4.4 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。

ATDE7 では atmark 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 atmark ユーザー)のパスワードです。

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

図 4.5 一時的なユーザーの切り替え(sudo コマンド)

4.2.2. 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 4.6 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 4.7 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 4.8 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 atmark 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 4.9 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)

4.2.3. 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、「4.3.4. 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 4.10 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 4.11 ユーザーが所属するグループを変更する(usermod コマンド)

ユーザーを別のグループにも所属させるには usermod コマンドの-aG オプションを使用します。

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

図 4.12 ユーザーを別のグループにも所属させる(usermod コマンド)

ユーザーが所属しているグループを確認するには、groups コマンドを使用します。

```
[ATDE ~]$ groups newuser
newuser : newuser newgroup
```

図 4.13 ユーザーが所属しているグループを確認する(groups コマンド)

4.3. 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。

4.3.1. 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。

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、「4.3.2. 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。

4.3.2. 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 4.1 inode が持つ情報

情報	説明
種類	「4.3.1. 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[a]

[a]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。

[2]物理的なデバイスを操作するためのプログラム

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 34676  2 月 22  2017 /bin/cat
```

図 4.14 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 4.2 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、「4.3.4. 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。

「34676」はバイト単位のファイルサイズです。

「2 月 22 2017」は、ファイルの最終修正時刻を表します。

4.3.3. 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、ext3 ファイルシステム、ext4 ファイルシステムがあります。Armadillo-600 シリーズのルートファイルシステムは、eMMC 上に構成された ext4 ファイルシステムです。ext4 ファイルシステムは、ジャーナリング機能[3]を持っており、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。

4.3.4. 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 4.14. 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[4]、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-

[3]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。
[4]rwx を 2 進数の各桁に見立てています。

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 4.15 ファイルのパーミッションを変更する(chmod コマンド)

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 4
drwxr-xr-x 2 atmark atmark 4096  4 月  7 16:39 dir
-rw-r--r-- 1 atmark atmark    0  4 月  7 16:38 file
```

図 4.16 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 1412  4 月  7 11:18 /etc/shadow
```

```
[ATDE ~]$ ls -l $(which passwd)
-rwsr-xr-x 1 root root 57972  5 月 17  2017 /usr/bin/passwd
```

図 4.17 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 16 root root 4096  4 月  7 17:17 /tmp
```

図 4.18 /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。

4.3.5. 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、ほとんどの場合、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  4 月  7 09:41 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65  4 月  7 09:41 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66  4 月  7 09:41 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67  4 月  7 09:41 /dev/ttyS3
```

図 4.19 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-[version]-at/Documentation/admin-guide/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 4.20 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[5]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。

4.4. 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[6]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)に分かれており、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。

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

```
[ATDE ~]$ ps
  PID TTY          TIME CMD
 3580 pts/0    00:00:00 bash
 4233 pts/0    00:00:00 ps
```

図 4.21 プロセス一覧の確認(ps コマンド)

「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。

4.5. 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 CtrlC)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[7]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 4.3 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(CtrlC)
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

[7]プロセスの状態を保存したもの。

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。

4.6. 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協調動作をしたり、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)

4.7. 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[8]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。

4.7.1. 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

[8]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。

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。

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

図 4.22 シリアルポートの読み書き

❶ 文字列「hello」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 4.23 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-600 シリーズでは、/dev/ttymxc#(#は 10 進数値文字)となり、/dev/ttymxc0 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-600 シリーズ 製品マニュアル」の「Linux ドライバー覧」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。

4.7.2. 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[9]。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 9.0 では、CtrlAltF#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[10]。CtrlAltF1 で一番目の仮想コンソール(/dev/tty1)に切り替えます。Debian GNU/Linux 9.0 では、F3 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。CtrlAltF2 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。CtrlAltF3 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。

[9]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。
[10]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。CtrlAlt スペースを入力したあと、CtrlAlt を押したままで、CtrlAltF#を入力してください。

```
[ATDE ~]$ tty
/dev/tty3
```

図 4.24 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-600 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=ttymxc0,115200」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。ATDE7 では「BOOT_IMAGE=/vmlinuz-4.9.0-12-686-pae root=/dev/mapper/atde7—vg-root ro quiet」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
BOOT_IMAGE=/vmlinuz-4.9.0-12-686-pae root=/dev/mapper/atde7--vg-root ro quiet
```

図 4.25 カーネルパラメーターの確認(proc/cmdline ファイル)

4.7.3. 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。ATDE7 の[アクティビティ]-[アプリケーションを表示する]-[ユーティリティ]-[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 4.26 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。

4.8. 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
2020 年  4 月  8 日 水曜日 10:20:58 JST
```

図 4.27 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 ~]$ sudo hwclock
2020-04-08 10:22:26.888416+0900
```

図 4.28 ハードウェアクロックの参照(hwclock コマンド)

4.8.1. タイムゾーン

ATDE7 で date コマンドを実行すると、「図 4.27. 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
2020 年  4 月  8 日 水曜日 10:29:58 JST
[ATDE ~]$ TZ=UTC date
2020 年  4 月  8 日 水曜日 01:30:10 UTC
[ATDE ~]$ TZ=America/New_York date
2020 年  4 月  7 日 火曜日 21:30:26 EDT
```

図 4.29 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。ATDE7 では TZ 環境変数は設定されていないので、「図 4.27. 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。ATDE7 の標準設定では、/usr/share/zoneinfo/Asia/Tokyo です。Debian GNU/Linux 9.0 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 4.30 システムクロックをハードウェアクロックに設定する(UTC)

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

図 4.31 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)

4.8.2. 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 9.0 では、systemd-timesyncd というプログラムが自動的に NTP サーバーを使用して時刻設定をしてくれます。

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、systemd-timesyncd を停止してください。adjtimex コマンドでドリフトを測定している途中に、systemd-timesyncd 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 systemctl stop systemd-timesyncd.service
```

図 4.32 adjtimex によるシステムクロックの補正 1: systemd-timesyncd 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 install adjtimex ntpdate
```

図 4.33 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 Wed Apr  8 11:07:11 2020
reference time - system time = 1586311631.836 - 1586311631.836 = 0.000 sec
Last clock comparison was at Wed Apr  8 11:05:51 2020
Kernel time variables are unchanged - good.
System clock is synchronized (by ntpd?) - bad.
Checking wtmp file...
System has not booted since Wed Apr  8 11:05:51 2020 - good.
System time has not been changed since Wed Apr  8 11:05:51 2020 - good.
Checking /etc/adjtime...
/sbin/hwclock has not set system time and adjusted the cmos clock
since Wed Apr  8 11:05:51 2020 - good.

Are you sure that, since Wed Apr  8 11:05:51 2020,
  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 -0.2372 +- 6.9418 ppm

Are you sure that, since Wed Apr  8 11:05:51 2020,
  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]
The estimated error in the cmos clock is -5179 +- 7 ppm
```

図 4.34 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: 90244
    frequency: 28746
     maxerror: 296500
     esterror: 0
       status: 8193
time_constant: 7
    precision: 1
    tolerance: 32768000
```

```
         tick: 10000
     raw time:  1586311688s 534312689us = 1586311688.534312689
[ATDE ~]$ sudo adjtimex --adjust --review
start                     finish                    days    sys - cmos (ppm)
Wed Apr  8 11:05:51 2020  Wed Apr  8 11:07:11 2020  0.0009  5185.1 +- 0.2
start                     finish                    days    cmos_error (ppm)
Wed Apr  8 11:04:41 2020  Wed Apr  8 11:05:51 2020  0.0008  -5886 +- 5
Wed Apr  8 11:05:51 2020  Wed Apr  8 11:07:11 2020  0.0009  -5211 +- 5
start                     finish                    days    sys_error (ppm)
Wed Apr  8 11:05:51 2020  Wed Apr  8 11:07:11 2020  0.0009  1 +- 5
least-squares solution:
   cmos_error = -5580 +- 4 ppm
      suggested adjustment = 482.1483 sec/day
        current adjustment = 0.0000 sec/day
   sys_error = -367 +- 4 ppm
      suggested tick = 10004  freq =  -2161506
        current tick = 10000  freq =     28746
note: clock variations and unstated data errors may mean that the
least squares solution has a bigger error than estimated here
new tick = 10004  freq = -2161506
[ATDE ~]$ sudo adjtimex --print
         mode: 0
       offset: 72356
    frequency: -2161506
     maxerror: 353000
     esterror: 0
       status: 8193
time_constant: 7
    precision: 1
    tolerance: 32768000
         tick: 10004
     raw time:  1586311801s 957000028us = 1586311801.957000028
```

図 4.35 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

4.8.3. 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-600 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[11]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

[11]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。

る ATDE7 では無効です。Armadillo-600 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。

4.9. 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[12])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]+という書式になります。language は ISO639[13]で規程される言語コードです。また、territory は ISO 3166 で規程される国名コード[14]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
C.UTF-8
```

[12]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。
[13]http://www.loc.gov/standards/iso639-2/php/code_list.php
[14]http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm

```
POSIX
ja_JP.utf8
```

図 4.36 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
LANGUAGE=
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 4.37 現在のロケールを確認する(locale コマンド)

「図 4.27. システムクロックの参照(date コマンド)」で示したように、ATDE7 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
2020 年  4 月  8 日 水曜日 14:43:16 JST
[ATDE ~]$ LANG=en_US date
Wed Apr  8 14:43:30 JST 2020
```

図 4.38 ロケールを指定して date コマンドを実行

4.10. 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。

4.10.1. 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、ip コマンドを使用します。

ip コマンドと同等の機能を提供するコマンドに ifconfig というものもありますが、現在では ip コマンドの使用が推奨されています。

ip コマンドは以下のような構文で、オブジェクトと呼ばれるものとサブコマンドを組み合わせて実行します。

```
ip [オブジェクト] [サブコマンド]
```

図 4.39 ip コマンドの構文

オブジェクトには以下のようなものがあります。

表 4.4 ip コマンドのオブジェクト(一部)

オブジェクト	省略形	意味
address	addr, a	ネットワークインターフェースの IP アドレス
link	l	ネットワークインターフェース
route	r	ルーティングテーブル

サブコマンドには以下のようなものがあります。

表 4.5 ip コマンドのサブコマンド(一部)

サブコマンド	動作
show, list	表示する
add	設定する
del	削除する

ここで取り上げたオブジェクトやサブコマンドは一部です。詳細は ip コマンドの man ページや、ip コマンド自体の help を参照してください。

例えば、address オブジェクトに対してどのようなサブコマンドが用意されているのか確認するには、以下のように実行します。

```
[ATDE ~]$ ip address help
```

図 4.40 help の表示する(ip コマンド)

ip address show として実行すると、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 ~]$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

```
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp1s0 ❶: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default ⏎
qlen 1000
    link/ether xx:xx:xx:xx:xx:xx ❷ brd ff:ff:ff:ff:ff:ff
    inet 192.168.xxx.xxx/24 ❸ brd 192.168.xxx.xxx ❹ scope global dynamic enp1s0
       valid_lft 3011sec preferred_lft 3011sec
    inet6 xxxx::xxxx:xxxx:xxxx:xxxx/64 ❺ scope link
       valid_lft forever preferred_lft forever
```

図 4.41 ネットワークインターフェースの状態を取得する(ip コマンド)

❶ インターフェース名です。

❷ MAC アドレスです。

❸ IPv4 の IP アドレスです。

❹ ブロードキャストアドレスです。

❺ IPv6 の IP アドレスです。

「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。

特定のインターフェースの状態を取得する場合は以下のように指定します。

```
[ATDE ~]$ ip address show enp1s0
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen ⏎
1000
    link/ether xx:xx:xx:xx:xx:xx brd ff:ff:ff:ff:ff:ff
    inet 192.168.xxx.xxx/24 brd 192.168.xxx.xxx scope global dynamic enp1s0
       valid_lft 3319sec preferred_lft 3319sec
    inet6 xxxx::xxxx:xxxx:xxxx:xxxx/64 scope link
       valid_lft forever preferred_lft forever
```

図 4.42 特定のネットワークインターフェースの状態を取得する(ip コマンド)

ip 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設定できます。例えば、enp1s0 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。

ここで説明する add や del による設定は、Armadillo 上でも行えますが Armadillo-X1 や G3 シリーズでは、NetworkManager が自動的に実行しています。これらの製品では、各製品の製品マニュアルに従って NetworkManager 経由での設定を実施してください。

すでに IP アドレスが設定されている場合は、削除します。

```
[ATDE ~]$ sudo ip address del 192.168.xxx.xxx/24 dev enp1s0
```

図 4.43 ネットワークインターフェースから IP アドレスを削除する(ip コマンド)

次に IP アドレスを設定します。

```
[ATDE ~]$ sudo ip address add 192.168.0.2/24 dev enp1s0
[ATDE ~]$ ip address show enp1s0
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen ⏎
1000
    link/ether xx:xx:xx:xx:xx:xx brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.2/24 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 xxxx::xxxx:xxxx:xxxx:xxxx/64 scope link
       valid_lft forever preferred_lft forever
```

図 4.44 ネットワークインターフェースに IP アドレスを設定する(ip コマンド)

IP アドレスが設定されていることが確認できます。

ip 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、networking.service を再起動すると適用されます。

```
[ATDE ~]$ sudo systemctl restart networking.service
```

図 4.45 networking.service を再起動する

ATDE7 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).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback
```

図 4.46 ATDE7 の interfaces ファイル

「auto lo」という行は、「lo」を自動的に起動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。

/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。

4.10.2. 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。

ss 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、SSH サーバー(22 番ポートを使用)と HTTP サーバー(80 番ポートを使用)が動作している場合、以下のように表示されます。

```
[ATDE ~]$ ss -tanp
State      Recv-Q Send-Q Local Address:Port               Peer Address:Port
LISTEN     0      128          *:80                       *:*
LISTEN     0      128          *:22                       *:*
LISTEN     0      20     127.0.0.1:25                       *:*
LISTEN     0      128         :::80                      :::*
LISTEN     0      128         :::22                      :::*
LISTEN     0      20         ::1:25                      :::*
```

図 4.47 使用中のポート番号を調べる(ss コマンド)

4.10.3. 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
atde7
```

図 4.48 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。ATDE7 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       atde7

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

図 4.49 /etc/hosts ファイル

「localhost」というホスト名が「127.0.0.1[15]」、「atde7」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.atmark-techno.com」や「armadillo.atmark-techno.com」はホスト名です。「www.atmark-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。

4.10.4. 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=0.017 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.042 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=0.033 ms
^C
--- 192.168.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3079ms
rtt min/avg/max/mdev = 0.017/0.033/0.043/0.012 ms
```

図 4.50 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。

```
[ATDE ~]$ ping 192.168.100.2
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
```

[15]ローカルループバックインターフェースの IP アドレス。

```
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 4.51 ネットワークの到達確認: 失敗(ping コマンド)

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。

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 5.0 以降では、/bin/sh は/bin/dash (Debian Almquist shell)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。

```
[armadillo ~]# for dir in /*; do echo $dir; done
/bin
/boot
/dev
/etc
/home
/lib
/lost+found
/media
/mnt
/opt
/proc
/root
/run
/sbin
```

```
/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。

```
[armadillo ~]# for dir in /*
> do
> echo $dir
> done
/bin
/boot
/dev
/etc
/home
/lib
/lost+found
/media
/mnt
/opt
/proc
/root
/run
/sbin
/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。

```
[armadillo ~]# chmod +x example.sh
[armadillo ~]# ./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 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。

```
[armadillo ~]# echo hello world
hello world
[armadillo ~]# 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。

```
[armadillo ~]# 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。

```
[armadillo ~]# variable=hello
[armadillo ~]# echo $variable
hello
[armadillo ~]# variable=’hello world’
[armadillo ~]# echo $variable
hello world
[armadillo ~]# variable=1+2
[armadillo ~]# 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。

```
[armadillo ~]# variable=1
[armadillo ~]# echo $variable
1
[armadillo ~]# variable=$(($variable+1))
[armadillo ~]# echo $variable
2
[armadillo ~]# variable=$(($variable*2))
[armadillo ~]# 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。

```
[armadillo ~]# a[0]=”hello”
[armadillo ~]# a[1]=”world”
[armadillo ~]# echo ${a[0]}
hello
```

```
[armadillo ~]# echo ${a[1]}
world
```

図 5.9 配列の例

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	現在のユーザー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。

```
[armadillo ~]# env
XDG_SESSION_ID=c3
HUSHLOGIN=FALSE
USER=root
PWD=/root
HOME=/root
MAIL=/var/mail/root
SHELL=/bin/bash
TERM=vt220
SHLVL=1
LOGNAME=root
XDG_RUNTIME_DIR=/run/user/0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/usr/bin/env
```

図 5.10 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。

```
[armadillo ~]# export MY_ENV=value
[armadillo ~]# echo $MY_ENV
value
```

図 5.11 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。[a]
#	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。

[a]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。

```
[armadillo ~]# echo $HOME
/root
[armadillo ~]# echo ¥$HOME
$HOME
[armadillo ~]# echo '$HOME'
$HOME
[armadillo ~]# echo "$HOME"
/root
```

[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。

```
[armadillo ~]# echo "¥$HOME"
$HOME
```

図 5.12 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、sh 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ー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、sh ではおこなわれません。)

```
[armadillo ~]# echo ~root
/root
[armadillo ~]# echo ~atmark/file
/home/atmark/file
[armadillo ~]# echo ~unavailuser
~unavailuser
[armadillo ~]# pwd
/root
[armadillo ~]# cd /
[armadillo /]# echo ~+
/
[armadillo /]# echo ~-
/root
```

図 5.13 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。

```
[armadillo ~]# var=$(date)
[armadillo ~]# echo $var
Thu Apr 9 13:00:19 JST 2020
```

図 5.14 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、クォートの削除がおこなわれます。ク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。

```
[armadillo ~]# echo hello
hello
[armadillo ~]# echo $?
0
[armadillo ~]# sleep 100
Ctrl-C
[armadillo ~]# echo $?
130
[armadillo ~]# noexistent-command
bash: noexistent-command: not found
```

```
[armadillo ~]# echo $?
127
[armadillo ~]# touch not-executable-file
[armadillo ~]# ./not-executable-file
bash: ./not-executable-file: Permission denied
[armadillo ~]# echo $?
126
```

図 5.15 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、標準入力(stdin)、標準出力(stdout)、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、標準入力、標準出力、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。

```
[armadillo ~]# echo hello > log
```

図 5.16 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。

```
[armadillo ~]# echo hello >> log
```

図 5.17 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。標準入力のファイルディスクリプタは 0、標準出力は 1、標準エラー出力は 2 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。

```
[armadillo ~]# cat nonexistent_file 2> error
[armadillo ~]# cat error
cat: nonexistent_file: No such file or directory
```

図 5.18 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。

```
[armadillo ~]# echo hello > log
[armadillo ~]# echo hello 1> log
```

図 5.19 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。

```
[armadillo ~]# somecommand > log 2> error
```

図 5.20 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。

```
[armadillo ~]# somecommand > log 2>&1
```

図 5.21 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。

```
[armadillo ~]# somecommand > /dev/null 2>&1
```

図 5.22 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。

```
[armadillo ~]# cat < /etc/hostname
armadillo
```

図 5.23 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。

```
[armadillo ~]# echo hello | cat
hello
```

図 5.24 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。

```
[armadillo ~]# zcat /proc/config.gz | grep ARMADILLO | wc -l
4
```

図 5.25 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 5.26 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。

```
[armadillo ~]# cat <<EOF
> hello
> world
> EOF
```

```
hello
world
```

図 5.27 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。

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

図 5.28 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 5.29. 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 5.29 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 5.30 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語である!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 5.31 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 5.32 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 5.33 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 5.34 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 5.35 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 CtrlZ)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 5.36 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。

記載されていない比較の条件式については、man test を参照してください。

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
-e 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

条件式	結果
-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 5.37 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 5.38 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 5.39. 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 5.39 case 文の例(case_sample.sh)

5.3.10.3. for 文

for 文は、「図 5.40. 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 5.40 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 5.41. 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 5.41 for 文の例 1(for_sample1.sh)

「word」はパス名展開されますので、「図 5.42. 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 5.42 for 文の例 2(for_sample2.sh)

コマンド展開も行われますので、「図 5.43. 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 5.43 for 文の例 3(for_sample3.sh)

5.3.10.4. while 文

while 文は、「図 5.44. while 文の構文」のように書くことができます。

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

図 5.44 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 5.45 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 5.46 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 5.47 関数の例(function_sample1.sh)

```
[armadillo ~]# ./function_sample1.sh
call foo function
foo
```

図 5.48 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 5.49 関数の例(function_sample2.sh)

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

図 5.50 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 5.51 関数の例(function_sample3.sh)

```
[armadillo ~]# ./function_sample3.sh
in global
global
global
global
in function
function
function
global
in global
function
global
global
```

図 5.52 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 5.53 関数の例(function_sample4.sh)

```
[armadillo ~]# ./function_sample4.sh
foo returns 10
bar returns string
```

図 5.54 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

「Armadillo 入門編」の「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。

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

「Armadillo 入門編」の「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、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-gnueabihf-gcc を使用します。このコマンド名の前に付いている arm-linux-gnueabihf-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-gnueabihf-as、ARM クロスリンカであれば arm-linux-gnueabihf-ld になります。

6.1.3. make と makefile

「Armadillo 入門編」の「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。

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。

[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。
[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。

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 apt 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

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。

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、

[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。

```
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

「Armadillo 入門編」の「開発の基本的な流れ」で使用した sin.c と 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 sin.c から実行ファイル sin が生成されます。

```
CROSS := arm-linux-gnueabihf ❶

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

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

TARGET = sin ❹

all: $(TARGET) ❺

sin: sin.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-gnueabihf 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 sin は、sin.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 sin.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=arm-linux-gnueabihf-
arm-linux-gnueabihf-gcc -Wall -Wextra -O2   -c -o sin.o sin.c
arm-linux-gnueabihf-gcc -lm sin.o  -o sin
[ATDE ~]$ file sin
sin: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /
lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0,
BuildID[sha1]=49b950072b23d33d3e92be67f9eee226cb45779b, 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  sin  sin.c  sin.o
[ATDE ~]$ make clean
rm -f *~ *.o
[ATDE ~]$ ls
Makefile  sin.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 sin.o sin.c
gcc -lm sin.o  -o sin
[ATDE ~]$ ./sin
sin(0.5) = 0.479426
```

「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。

sin.c と cos.c と tan.c から実行ファイル trigonometric が生成され、asin.c と acos.c と atan.c 実行ファイル invtrigonometric 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-gnueabihf

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

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

TARGET = trigonometric invtrigonometric
```

```
all: $(TARGET)

hellotrigonometric: sin.o cos.o tan.o
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

invtrigonometric: asin.o acos.o atan.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]。

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。

[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。

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)

```
[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。

[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 ~]$ ./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。

[8]man 3 stdio 参照

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 errno 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。

```
#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
atde7
[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 1012996  4月 10 10:12 /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。

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
```

[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(2020 年 4 月現在の URL)
[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(2020 年 4 月現在の URL)

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
#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;
        static 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。

もし、GLib(ARM アーキテクチャ用)や pkg-config がインストールされていない場合は apt でインストールできます。

```
[ATDE ~]$ sudo apt install pkg-config libglib2.0-dev:armhf
```

図 6.18 pkg-config と GLib のインストール

```
CROSS   := arm-linux-gnueabihf

ifneq ($(CROSS),)
CROSS_PREFIX            := $(CROSS)-
PKGCONFIG_LIBDIR        := PKG_CONFIG_LIBDIR=/usr/lib/$(CROSS)/pkgconfig
endif

CC      = $(CROSS_PREFIX)gcc
CFLAGS  = -O2 -Wall -Wextra -I../common

PKGCONFIG_CFLAGS        = `$(PKGCONFIG_LIBDIR) pkg-config --cflags glib-2.0`
PKGCONFIG_LIBS          = `$(PKGCONFIG_LIBDIR) pkg-config --libs glib-2.0`

TARGET  = dispcsv2

all: $(TARGET)

dispcsv2: dispcsv2.c
        $(CC) $(CFLAGS) $(PKGCONFIG_CFLAGS) -o $@ $< $(PKGCONFIG_LIBS)
```

図 6.19 dispcsv2 のための Makefile

指定された CROSS(=アーキテクチャ名)から PKGCONFIG_LIBDIR 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_LIBDIR 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 6.20 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。

dispcsv2.conf を編集した dispcsv2 の実行結果.

```
  [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
```

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。

```
[ATDE ~]$ ./dispbmp midomadillo.bmp
```

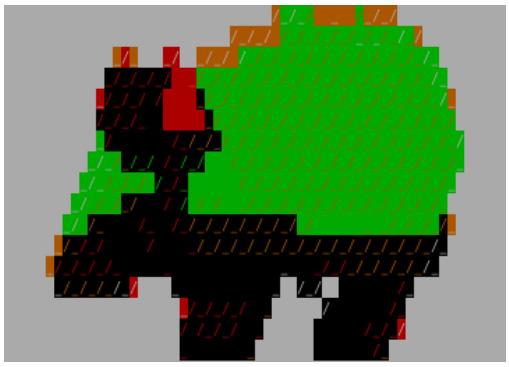

図 6.23 dispbmp の実行結果

6.4. デバイスの操作

「4.3.1. 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。

[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

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、「4.3.1. 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 シリーズソフトウェアマニュアル」を参照してください。

Armadillo-600 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

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ない)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 /dev/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"

#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;
```

[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。

```
}

/**
 * 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。

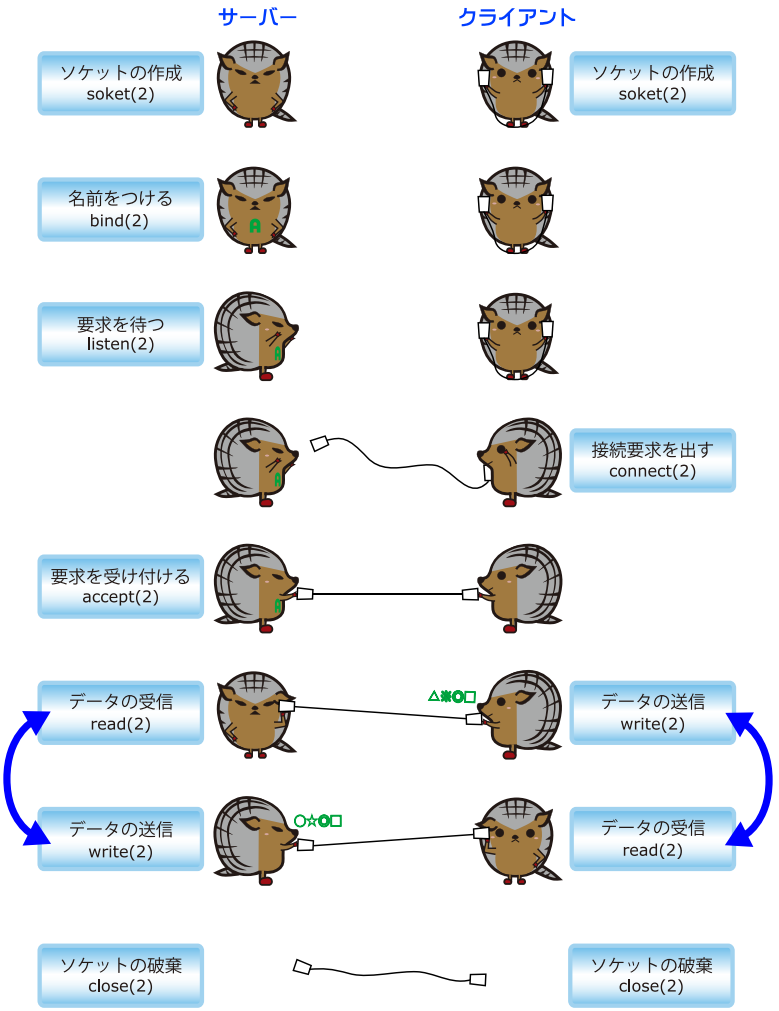

図 6.31 TCP/IP プログラムの基本的な流れ

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. サーバーは、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. サーバーは、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ーに糸電話を投げる(接続要求を出す)
5. サーバー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。

network_hello_server への telnet.

```
  [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.
```

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 6.34 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 6.35 network_echo_server1 の実行結果

PC から telnet で接続してみます。

network_echo_server1 への telnet.

```
  [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.
```

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 6.36 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。

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>
```

[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 10 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。

```
#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 6.37 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。

6.7. プログラムをデバッグする

C 言語で開発したプログラムをデバッグする際、printf を埋め込むいわゆる print デバッグがよく行われます。print デバッグもデバッグ手法の 1 つではありますが、プログラムの規模が大きくなるとこれだけでデバッグするのは効率が悪く、かつメモリリークなどの重大なバグは見逃してしまう可能性もあります。

ここではより効率的な C 言語プログラムのデバッグ手法を紹介します。

6.7.1. gdb によるデバッグ

gdb とはデバッグを行うためのツールで、デバッガと呼ばれるものの 1 つです。プログラムを 1 行ずつ実行したり、ブレークポイントを設定した行で処理を中断できたり、変数に格納されている値を確認できたりと、様々なことができます。

ここでは、gdb の基本的な使い方について説明します。

デバッグ対象のプログラムとして、1 から 100 までの数を全て加算する簡単なものを用意しました。

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

int sum(int a)
{
    int i, sum = 0;
    for (i = 1; i < a; i++) {
        sum += i;
    }
    return sum;
}

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

```
    s = sum(100);
    printf("%d¥n", s);

    return EXIT_SUCCESS;
}
```

図 6.38 1 から 100 までの和を計算するプログラム (sum.c)

このソースコードを -g -O0 オプションを付けてコンパイルします。

```
[armadillo ~]# gcc -g -O0 sum.c -o sum
```

-g オプションは gdb でデバッグできるようにするために必要なオプションです。-O0 オプションはコンパイラによる最適化をなしにするためのオプションです。最適化されてしまうとデバッグしづらくなってしまいます。

このプログラムを実行すると以下のように表示されます。

```
[armadillo ~]# ./sum
4950
```

図 6.39 sum の実行結果

期待していた結果である 5050 とは違った値となりました。gdb を使ってデバッグを行います。

gdb がインストールされていない場合、apt でインストールできます。

```
[armadillo ~]# apt install gdb
```

gdb を使ってプログラムを起動します。

```
[armadillo ~]# gdb sum
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sum...done.
(gdb)
```

(gdb)プロンプトが表示されました。今後はこのプロンプトにコマンドを入力してデバッグを進めます。

start と入力すると main 関数まで進んで実行が止まります。

```
(gdb) start
Temporary breakpoint 1 at 0x610: file sum.c, line 17.
Starting program: /root/work/developers_guide/c/gdb/sum

Temporary breakpoint 1, main (argc=1, argv=0xbefffd84) at sum.c:17
17          s = sum(100);
(gdb)
```

ソースコードの 17 行目の処理で止まっていることを示しています。

next と入力すると次の行へ進んで実行が止まります。

```
(gdb) next
18          printf("%d¥n", s);
```

ここで、print コマンドを使うと変数 s に入っている値を表示することができます。

```
(gdb) print s
$1 = 4950
```

プログラムの実行を再開するには continue と入力します。

```
(gdb) continue
Continuing.
4950
[Inferior 1 (process 1812) exited normally]
```

プログラムが最後まで実行されました。gdb を終了するには quit と入力します。

```
(gdb) quit
[armadillo ~]#
```

プログラムが最後まで実行されても、再度 start と入力すると最初から実行することができます。

```
(gdb) start
Temporary breakpoint 2 at 0x400610: file sum.c, line 17.
Starting program: /root/work/developers_guide/c/gdb/sum

Temporary breakpoint 2, main (argc=1, argv=0xbefffd84) at sum.c:17
17          s = sum(100);
```

ここで、next ではなく step と入力すると sum 関数の中へ入ることができます。next も step も 1 行ずつ実行するコマンドですが、next は関数の中には入らず、step は関数の中に入ります。

```
(gdb) step
sum (a=100) at sum.c:6
6           int i, sum = 0;
```

sum 関数の中に入って処理が止まりました。

目的の行がある場合、その行にたどり着くまで next や step を入力し続けるのは大変なので、目的の行にブレークポイントを設定し、そこまで一気に実行してみます。

list と入力するとソースコードが表示されるので、目的の行が何行目なのかを確認します。

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int sum(int a)
5       {
6           int i, sum = 0;
7           for (i = 1; i < a; i++) {
8               sum += i;
9           }
10          return sum;
```

ソースコード内にコメントがある場合はコメントも表示されます。この時、日本語のコメントがあった場合、環境によっては文字化けが発生することがありますが、gdb の実行には影響ありません。

break コマンドで 8 行目にブレークポイントを設定します。

```
(gdb) break 8
Breakpoint 3 at 0x4005e2: file sum.c, line 8.
(gdb) continue
Continuing.

Breakpoint 3, sum (a=100) at sum.c:8
8               sum += i;
(gdb) print sum
$2 = 0
(gdb) print i
$3 = 1
```

ブレークポイントを設定し、continue するとブレークポイントの行で処理が止まります。そこで、変数 sum と i の値を表示しています。

display コマンドを使うと、変化していく変数の値を常に表示できます。

```
(gdb) display sum
1: sum = 0
(gdb) display i
```

```
2: i = 1
(gdb) continue
Continuing.

Breakpoint 3, sum (a=100) at sum.c:8
8               sum += i;
1: sum = 1
2: i = 2
(gdb) continue
Continuing.

Breakpoint 3, sum (a=100) at sum.c:8
8               sum += i;
1: sum = 3
2: i = 3
(gdb) continue
Continuing.

Breakpoint 3, sum (a=100) at sum.c:8
8               sum += i;
1: sum = 6
2: i = 4
```

for 文を抜けるまで continue し続けるのは大変なので、一度ブレークポイントを削除し、関数の return 文の所に設定し直します。

ブレークポイントは delete コマンドで削除できますが、この時はソースコードの行数ではなく、ブレークポイント番号を指定する必要があります。

```
(gdb) delete 3
(gdb) break 10
Breakpoint 4 at 0x4005f8: file sum.c, line 10.
(gdb) continue
Continuing.

Breakpoint 4, sum (a=100) at sum.c:10
10          return sum;
1: sum = 4950
2: i = 100
```

ブレークポイント番号 3 を削除し、10 行目にブレークポイントを設定し直してそこまで処理を進めました。その時の、変数 sum と i の値が表示されています。

ここで、変数 i の値に注目すると 100 となっていることから、for ループは 99 回しか実行されておらず、for ループの終了条件に問題があることがわかりました。

このように、gdb を使うとプログラムの実行の中断や変数の値の確認などができ、デバッグの効率化が見込めます。

最後に、ここでの説明で使用した gdb のコマンド一覧と、使用はしていませんが有用なコマンドを示します。またほとんどのコマンドは、省略形でも使うことができます。

表 6.3 使用したコマンド

コマンド	省略形	動作
start	-	実行を開始し main 関数で停止する
step	s	1 行ずつ実行し、関数の場合は中に入る
next	n	1 行ずつ実行し、関数には入らない
print	p	変数の値やアドレスを表示する
continue	c	実行を再開する
quit	q	gdb を終了する
list	l	ソースコードを表示する
break n	b n	n 行目にブレークポイントを設定する
delete n	d n	n 番目のブレークポイントを削除する
display	-	常に変数の値やアドレスを表示する

表 6.4 使用していないが有用なコマンド

コマンド	省略形	動作
run	r	実行を開始する
finish	fin	関数を return まで実行する
return n	-	n を return し、強制的に関数から抜ける
help	h	help を表示する

ここで紹介したコマンド以外にも多くのコマンドが用意されています。詳細に関しては、gdb の help コマンドおよび公式ドキュメント[17]を参照してください。

6.7.2. strace でシステムコールをトレースする

プログラムが呼び出すシステムコールをトレース(追跡)することで、デバッグの役に立てることができます。

ここでは、システムコールをトレースする方法を簡単に説明します。

システムコールとは、open、close、write、read などの関数をコールして Linux カーネルが提供している機能をユーザー空間から使う仕組みです。

システムコールをトレースするには strace というコマンドを使います。

ここでは「6.7.1. gdb によるデバッグ」で紹介したプログラムを strace でトレースします。

一番簡単なやり方は、strace の引数としてプログラムを渡すだけです。

```
[armadillo ~]# strace ./sum
execve("./sum", ["./sum"], [/* 14 vars */]) = 0
brk(NULL)                               = 0x197c000
uname({sysname="Linux", nodename="armadillo", ...}) = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=30847, ...}) = 0
mmap2(NULL, 30847, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6fde000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "¥177ELF¥1¥1¥1¥3¥0¥0¥0¥0¥0¥0¥0¥0¥3¥0(¥0¥1¥0¥0¥0¥331e¥1¥0004¥0¥0¥0"..., 5
12) = 512
```

[17]https://www.gnu.org/software/gdb/documentation/

```
lseek(3, 900076, SEEK_SET)              = 900076
read(3, "¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0"..., 2
920) = 2920
lseek(3, 896524, SEEK_SET)              = 896524
read(3, "A4¥0¥0¥0aeabi¥0¥1*¥0¥0¥0¥0057-A¥0¥6¥n¥7A¥10¥1¥t¥2¥n¥3¥f"..., 53) = 53
fstat64(3, {st_mode=S_IFREG|0755, st_size=902996, ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6
fdc000
mmap2(NULL, 972112, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6
ed1000
mprotect(0xb6faa000, 61440, PROT_NONE)  = 0
mmap2(0xb6fb9000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRI
TE, 3, 0xd8000) = 0xb6fb9000
mmap2(0xb6fbc000, 9552, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOU
S, -1, 0) = 0xb6fbc000
close(3)                                = 0
set_tls(0xb6fdcb80, 0xb6fdd258, 0xb6fe7050, 0xb6fdcb80, 0xb6fe7050) = 0
mprotect(0xb6fb9000, 8192, PROT_READ)   = 0
mprotect(0x4ce000, 4096, PROT_READ)     = 0
mprotect(0xb6fe6000, 4096, PROT_READ)   = 0
munmap(0xb6fde000, 30847)               = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(207, 16), ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x197c000
brk(0x199e000)                          = 0x199e000
write(1, "4950¥n", 54950
)                    = 5
exit_group(0)                           = ?
+++ exited with 0 +++
```

open や read などのほかにも、多くのシステムコールが使われていることが確認できます。

特定のシステムコールのみを表示したい場合は-e オプションを使います。例えば open のみトレースしたい場合は以下のように指定します。

```
[armadillo ~]# strace -e open ./sum
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
4950
+++ exited with 0 +++
```

複数のシステムコールを指定したい場合は、-e trace=として、システムコールをカンマ区切りで指定します。open と close の 2 つを指定すると以下のように表示されます。

```
[armadillo ~]# strace -e trace=open,close ./sum
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
open("/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
4950
+++ exited with 0 +++
```

-c オプションを使うと、システムコールの統計情報を出力することができます。

```
[armadillo ~]# strace -c ./sum
4950
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 43.27    0.000074          74         1           write
 29.82    0.000051          17         3           brk
 16.37    0.000028          28         1           ioctl
 10.53    0.000018           6         3           fstat64
  0.00    0.000000           0         3           read
  0.00    0.000000           0         2           open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2           lseek
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         5           mmap2
  0.00    0.000000           0         1           set_tls
------ ----------- ----------- --------- --------- ----------------
100.00    0.000171                    33         3 total
```

この統計情報で、コール回数やエラーの有無などの情報も確認できます。

このように strace を使ってシステムコールをトレースし結果を確認することができます。

コール回数が異常に多い場合や、エラーが発生しているような場合にそのシステムコールを使用している個所を重点的に調べることで、バグの発見につなげることができます。

strace の使い方の詳細については、strace の man ページを参照してください。

6.7.3. メモリ破壊やメモリリークのデバッグ

C 言語によるプログラミングにおいて、発見しづらいバグとしてメモリ破壊やメモリリークがあります。

これらのバグがあっても、プログラムは正常動作することもあり見逃されがちですが、長時間稼働するようなプログラムにおいては、致命的な問題を引き起こす可能性もあります。

ここではツールを用いたメモリ破壊やメモリリークの検出方法を簡単に説明します。

6.7.3.1. Electric Fence を使ったメモリ破壊検出

メモリ破壊とは確保したメモリ領域以外にアクセスしてしまうことで発生します。

ここではメモリ破壊検出ツールである Electric Fence を使います。

Electric Fence は apt でインストールすることができます。

```
[armadillo ~]# apt install electric-fence
```

メモリ破壊を行うプログラムを以下の通り準備します。

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

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

    p = (int *) malloc(sizeof(int) * 5);
    p[5] = 100;
    free(p);

    return EXIT_SUCCESS;
}
```

図 6.40 メモリ破壊を行うプログラム (mem_corruption.c)

変数 p に int 型のサイズで 5 つ分の領域を確保していますが、p[5]へのアクセスしており、これは範囲外のアクセスとなります。仮に p[5]が必要な領域を参照していた場合、その領域を書き換えてしまっていることになります。

しかしコンパイルして、実行すると正常に終了します。

```
[armadillo ~]# gcc -g -O0 mem_corruption.c -o mem_corruption
[armadillo ~]# ./mem_corruption
[armadillo ~]#
```

このように、偶然にも動いてしまうため、メモリ破壊は発見しづらいバグです。

Electric Fence を使うためには、コンパイルするときに libefence.so をリンクするだけです。

```
[armadillo ~]# gcc -g -O0 mem_corruption.c -o mem_corruption -lefence
[armadillo ~]# ./mem_corruption

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Segmentation fault
```

実行すると、Segmentation fault が発生してプログラムが終了しました。gdb と組み合わせると、どこで発生しているのかも確認することができます。

```
[armadillo ~]# gdb mem_corruption

(省略)

(gdb) run
Starting program: /root/work/developers_guide/c/gdb/mem_corruption
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/arm-linux-gnueabihf/libthread_db.so.1".

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x00400732 in main (argc=1, argv=0xbefffd44) at mem_corruption.c:9
9           p[5] = 100;
(gdb)
```

このように、メモリ破壊を検出することができます。Electric Fence の詳細については man ページ(man efence)を参照してください。

6.7.3.2. MemWatch を使ったメモリリーク検出

Electric Fence ではメモリ破壊は検出できますが、メモリリークを検出することはできません。

ここでは、メモリリーク検出ツールである MemWatch を使います。

MemWatch はソースコードとして提供されており、デバッグしたいプログラムと一緒にコンパイルすることで動作します。

MemWatch のソースコードは GitHub からダウンロードしてくることができます。

https://github.com/502110983/MemWatch

メモリリークが発生するプログラムを以下の通り準備します。MemWatch のヘッダである memwatch.h をインクルードする必要がある点に注意してください。

```
#include <stdio.h>
#include <stdlib.h>
#include "memwatch.h"

int main(int argc, char *argv[])
{
    int *p1, *p2;

    p1 = (int *) malloc(sizeof(int));
    p2 = (int *) malloc(sizeof(int));

    p2 = p1;

    free(p1);
    free(p2);

    return EXIT_SUCCESS;
}
```

図 6.41 メモリリークが発生するプログラム (mem_leak.c)

変数 p2 として確保した領域が解放されないままになっており、かつ p1 として確保した領域が二重に解放されています。

このプログラムを memwatch.c と一緒にコンパイルします。コンパイルする時に-DMEMWATCH -DNW_STDIO というオプションを設定します。

```
[armadillo ~]# gcc -DMEMWATCH -DMW_STDIO mem_leak.c memwatch.c -o mem_leak
[armadillo ~]# ./mem_leak
MEMWATCH detected 2 anomalies
[armadillo ~]#
```

実行すると、memwatch.log というファイルができるので、このファイルを確認します。

```
[armadillo ~]# cat ./memwatch.log
```

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Wed Apr 29 21:58:47 2020

Modes: __STDC__ 64-bit mwDWORD==(unsigned long)
mwROUNDALLOC==8 sizeof(mwData)==32 mwDataSize==32

double-free: < 4 > mem_leak.c(15), 0x8261d8 was freed from mem_leak.c(14)

Stopped at Wed Apr 29 21:58:47 2020

unfreed: < 2 > mem_leak.c(10), 4 bytes at 0x826210        {FE FE FE FE .. .. .. ..
 .. .. .. .. .. .. .. .. ....}

Memory usage statistics (global):
 N)umber of allocations made: 2
 L)argest memory usage      : 8
 T)otal of all alloc() calls: 8
 U)nfreed bytes totals      : 4
```

このファイルの以下の出力に注目すると、メモリの二重解放と未解放があることがわかります。

```
ソースコード 15 行目で二重解放
double-free: < 4 >  mem_leak.c(15), 0x8261d8 was freed from mem_leak.c(14)

ソースコード 10 行目で確保した領域が未解放
unfreed: < 2 > mem_leak.c(10), 4 bytes at 0x826210
```

このように通常であれば発見しづらいメモリリークを発見することができます。

MemWatch の詳細については MemWatch のソースコード一式に含まれる README と FAQ を参照してください。

7. Python 言語による実践プログラミング

この章では、Python 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。

Python はスクリプト言語であり、コンパイル言語の C と違い PC で作成したプログラムを Armadillo 用にクロスコンパイルする必要がないので、効率よく開発ができる可能性があります。

さらに、非常に多くのパッケージ(ライブラリ)が提供されており、かつそれらを簡単に導入できます。

現在では、ディープラーニングなどにも利用されており、一般的なアプリケーションから研究開発用アプリケーションまで幅広く利用されている言語です。

クラウドサービスを利用したアプリケーションを開発する場合などでも、各クラウドサービスともに Python 向けの SDK も提供していることが多いです。

実行速度が求められるアプリケーションやデバイスドライバの開発をするのでなければ、開発言語として Python は選択肢の一つになります。

ここでは Python の文法に関する詳細な解説はしません。詳細な解説は公式サイト[1][2]を参照してください。

7.1. Python 実行環境をインストールする

Python の実行環境は apt で簡単にインストールできます。Python にはバージョン 2 系と 3 系があり、現在ではバージョン 3 系が主流となっているので、ここでもバージョン 3 系をインストールします。

インストールは apt でできます。

```
[armadillo ~]# apt install python3
```

図 7.1 python バージョン 3 のインストール

インストール後に動作を確認できます。

```
[armadillo ~]# python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello, world.")
hello, world.
>>> a = 10
>>> b = a * 2
>>> print(b)
20
```

[1]https://docs.python.org/ja/3/tutorial/(日本語チュートリアル)
[2]https://wiki.python.org/moin/BeginnersGuide(英語)

```
>>> quit()
[armadillo ~]#
```

図 7.2 python の動作確認

単に python3 として実行すると対話モードと呼ばれるモードで python が起動します。このモードでは入力されたコードが随時実行されていきます。対話モードを終了する場合は quit()関数を呼び出します。

次に、Python のパッケージ管理ツールである pip をインストールします。pip を使うことで必要なパッケージ(ライブラリ)を手軽にインストールできるようになります。

```
[armadillo ~]# apt install python3-pip
```

図 7.3 pip のインストール

7.2. ファイルの取り扱い

まずは、「6.3. ファイルの取り扱い」と同様にファイルの取扱について取り上げます。

7.2.1. テキストファイルを扱う

ここでは、「6.3.1. テキストファイルを扱う」と同じ仕様のサンプルプログラムを紹介します。Python では CSV を扱うパッケージが標準で用意されているのでこれを使います。

```
import csv
import sys
import os
import copy

# 表示する文字数
DISP_WIDTH = 60  # 幅
DISP_HEIGHT = 17 # 高さ

# CSV データ 1 行の要素数
COLUMN_NUM = 11

line = 0   # 一画面に表示した行数
count = 0  # 表示したデータ総数


def printline(csvline):

    """ 行表示関数

    Args:
        csvline(string の配列): 表示する 1 行分の CSV データ

    """

    global line
    global count

    # CSV データの各要素を表示
```

```
    if csvline is not None:
        # データがない場合表示しない
        if len(csvline) < 1:
            return

        #各要素を表示フォーマットに展開
        buf = "%6d %s %s %s %s %s %s %s %s %s %s %s" % ¥
                (count, csvline[0], csvline[1], csvline[2], csvline[3], csvline[4], ¥
                 csvline[5], csvline[6], csvline[7], csvline[8], csvline[9], csvline[10])
        count += 1
    else:
        # CSV データの総数を表示
        # データ総数を表示フォーマットに展開
        buf = "Count: %6d" % (count)

    # 今回追加される表示行数を計算
    newline = int((len(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:
        input()
        # 表示行数を初期化
        line = 0
    # 実際に表示する
    print(buf)
    # 表示行数を更新
    line += newline

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <csvfile>" % sys.argv[0])
        sys.exit(os.EX_OK)

    # CSV ファイルオープン
    with open(sys.argv[1]) as csvfile:
        pzip = "00000000"
        csvline = [] # CSV データ初期化
        # CSV ファイルから 1 行読み込む
        csvreader = csv.reader(csvfile)
        for row in csvreader:
            # 要素数が不足している場合、次行にスキップ
            if len(row) < COLUMN_NUM:
                continue

            # 新しいデータの場合
            if pzip != row[1]:
                printline(csvline)
                csvline = copy.copy(row)
            else:
                # 既存データへの追加の場合
                # 町域名の続きを追加
                csvline[2] += row[2]
            pzip = row[1]

        # 保持済みのデータを表示
        printline(csvline)
```

```
        # データ総数を表示
        printline(None)
```

図 7.4 CSV ファイルの内容を表示するプログラム(dispcsv1.py)

一見して C 言語のプログラムより短くなっていることがわかります。特に CSV ファイルを読み込んでカンマ区切りでトークンに分割する処理については、Python では 2 行で収まっています。

```
import csv
(中略)
        # CSV ファイルから 1 行読み込む
        csvreader = csv.reader(csvfile)
        for row in csvreader:
```

変数 row にはすでに分割されたトークンが配列として代入されているので、後の処理は内容を整形して表示するだけです。

このように Python では適切なパッケージを import することによって、煩雑な処理を簡潔に書くことができ、バグを作り込む可能性も低減できます。

実際に動作させると C 言語版と同じ結果が表示されます。

```
[armadillo ~]# python3 ./dispcsv1.py ./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 7.5 dispcsv1.py 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。

7.2.2. 設定ファイルに対応する

ここでも、「6.3.2. 設定ファイルに対応する」と同じ仕様のサンプルプログラムを紹介します。Python には設定ファイルを取り扱う configparser パッケージも標準で用意されているのでこれを使います。

```
import csv
import configparser
import sys
import os
import copy
import json

# 表示する文字数
DISP_WIDTH = 60  # 幅
DISP_HEIGHT = 17 # 高さ
```

```
# CSV データ 1 行の要素数
COLUMN_NUM = 11

line = 0   # 一画面に表示した行数
count = 0  # 表示したデータ総数

conf = configparser.ConfigParser() # 設定ファイル管理オブジェクト

def printline(csvline):

    """ 行表示関数

    Args:
        csvline(string の配列): 表示する 1 行分の CSV データ

    """

    global line
    global count

    # CSV データの各要素を表示
    if csvline is not None:
        # データがない場合表示しない
        if len(csvline) < 1:
            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 int(conf['data']['Code']) >= 0 and csvline[0] != conf['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 conf['data']['Zipcode'] != "" and conf['data']['Zipcode'] not in csvline[1]:
            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 conf['data']['Street'] != "" and conf['data']['Street'].lower() not in csvline[2].lower():
            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 conf['data']['City'] != "" and conf['data']['City'].lower() not in csvline[3].lower():
            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 conf['data']['Pref'] != "" and conf['data']['Pref'].lower() not in csvline[4].lower():
            return

        for idx, flag in enumerate(json.loads(conf['data']['Flag'])):
            if int(flag) >= 0 and flag != csvline[5 + idx]:
                return

        #各要素を表示フォーマットに展開
        buf = "%6d %s %s %s %s %s %s %s %s %s %s %s %s" % ¥
                (count, csvline[0], csvline[1], csvline[2], csvline[3], csvline[4], ¥
                 csvline[5], csvline[6], csvline[7], csvline[8], csvline[9], csvline[10])
        count += 1
    else:
        # CSV データの総数を表示
        # 総数表示無効なら表示しない
        if not conf['control']['Count']:
            return
        # データ総数を表示フォーマットに展開
        buf = "Count: %6d" % (count)
```

```
    # 今回追加される表示行数を計算
    disp_width = int(conf['display']['Width'])
    disp_height = int(conf['display']['Height'])
    newline = int((len(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 conf['control']['Pause']:
            input()
        # 表示行数を初期化
        line = 0
    # 実際に表示する
    print(buf)
    # 表示行数を更新
    line += newline


def readconf(conf_file_name):

    """ conf ファイル読み込み関数

    Args:
        conf_file_name(string): conf ファイルのファイル名

    """

    global conf
    if not os.path.exists("./" + conf_file_name):
        # ファイルが存在していない場合は新規作成
        # デフォルト設定
        conf['display'] = {
            'Width': 60,
            'Height': 17
        }
        conf['control'] = {
            'Pause': True,
            'Count': True
        }
        conf['data'] = {
            'Code': -1,
            'Zipcode': "",
            'Street': "",
            'City': "",
            'Pref': "",
            'Flag': [-1, -1, -1, -1, -1, -1]
        }
        with open(conf_file_name, 'w') as conf_file:
            conf.write(conf_file)
    else :
        conf.read(conf_file_name)

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <csvfile>" % sys.argv[0])
        sys.exit(os.EX_OK)
```

```
    # conf ファイルを読み込み
    readconf(os.path.splitext(sys.argv[0])[0] + ".conf")

    # CSV ファイルオープン
    with open(sys.argv[1]) as csvfile:
        csvreader = csv.reader(csvfile)
        pzip = "00000000"
        csvline = [] # CSV データ初期化
        # CSV ファイルから 1 行読み込む
        for row in csvreader:
            # 要素数が不足している場合、次行にスキップ
            if len(row) < COLUMN_NUM:
                continue

            # 新しいデータの場合
            if pzip != row[1]:
                printline(csvline)
                csvline = copy.copy(row)
            else:
                # 既存データへの追加の場合
                # 町域名の続きを追加
                csvline[2] += row[2]
            pzip = row[1]

        # 保持済みのデータを表示
        printline(csvline)
        # データ総数を表示
        printline(None)
```

図 7.6 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.py)

設定ファイルの作成・読み込みを行う readconf 関数を追加しています。この関数の中で configparser パッケージを使って処理を行っています。

設定ファイルの新規作成時のデフォルト値が連想配列のように書くことができたり、読み込みは 1 行で済んだりと、こちらも C 言語版と比べて簡潔に書くことができています。

注意点として、読み込んだ設定値はすべて文字列として保持されるため、数値として扱う場合や配列として扱う場合は、適宜変換する必要があります。

初回実行した後に、dispcsv2.conf ファイルができています。

```
[display]
width = 60
height = 17

[control]
pause = True
count = True

[data]
code = -1
zipcode =
street =
city =
```

```
pref =
flag = [-1, -1, -1, -1, -1, -1]
```

C 言語版と同様に、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

```
[display]
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 ~]# python3 ./dispcsv2.py ./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 7.7 dispcsv2.conf を編集した dispcsv2.py の実行結果

7.2.3. JSON ファイルを扱う

Python での JSON ファイルを扱いについて説明します。現在ほとんどのクラウドサービスでは、通信のレスポンスとして JSON 形式を採用しており、クラウドサービスと連携したアプリケーションを開発するには、JSON ファイルを扱うことがほぼ必須となっています。

当然、C 言語でも JSON ファイルを扱うプログラムを書くことはできますが、C 言語では書きづらい文字列処理を多く実装しなくてはならないため、簡潔に書くのは難しくなります。

ここでは、「7.2.2. 設定ファイルに対応する」で取り上げた設定ファイルを、JSON 形式として保存・読み込みを行うサンプルプログラムを示します。

```
import csv
import sys
import os
import copy
import json

# 表示する文字数
DISP_WIDTH = 60  # 幅
DISP_HEIGHT = 17 # 高さ

# CSV データ 1 行の要素数
COLUMN_NUM = 11

line = 0   # 一画面に表示した行数
count = 0  # 表示したデータ総数

conf = {} # 設定ファイル管理オブジェクト

def printline(csvline):

    """ 行表示関数

    Args:
        csvline(string の配列): 表示する 1 行分の CSV データ

    """

    global line
    global count

    # CSV データの各要素を表示
    if csvline is not None:
        # データがない場合表示しない
        if len(csvline) < 1:
            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 int(conf['data']['Code']) >= 0 and csvline[0] != conf['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 conf['data']['Zipcode'] != "" and conf['data']['Zipcode'] not in csvline[1]:
            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 conf['data']['Street'] != "" and conf['data']['Street'].lower() not in csvline[2].lower():
            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 conf['data']['City'] != "" and conf['data']['City'].lower() not in csvline[3].lower():
            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 conf['data']['Pref'] != "" and conf['data']['Pref'].lower() not in csvline[4].lower():
            return

        for idx, flag in enumerate(conf['data']['Flag']):
            if flag >= 0 and flag != csvline[5 + idx]:
                return

        #各要素を表示フォーマットに展開
        buf = "%6d %s %s %s %s %s %s %s %s %s %s %s" % ¥
                (count, csvline[0], csvline[1], csvline[2], csvline[3], csvline[4], ¥
                 csvline[5], csvline[6], csvline[7], csvline[8], csvline[9], csvline[10])
        count += 1
    else:
        # CSV データの総数を表示
        # 総数表示無効なら表示しない
        if not conf['control']['Count']:
            return
        # データ総数を表示フォーマットに展開
        buf = "Count: %6d" % (count)

    # 今回追加される表示行数を計算
    disp_width = conf['display']['Width']
    disp_height = conf['display']['Height']
    newline = int((len(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 conf['control']['Pause']:
            input()
        # 表示行数を初期化
        line = 0
    # 実際に表示する
    print(buf)
    # 表示行数を更新
    line += newline


def readconf(conf_file_name):

    """ conf ファイル読み込み関数

    Args:
        conf_file_name(string): conf ファイルのファイル名

    """

    global conf
    if not os.path.exists("./" + conf_file_name):
        # ファイルが存在していない場合は新規作成
        # デフォルト設定
        conf['display'] = {
            'Width': 60,
            'Height': 17
        }
```

```
        conf['control'] = {
            'Pause': True,
            'Count': True
        }
        conf['data'] = {
            'Code': -1,
            'Zipcode': "",
            'Street': "",
            'City': "",
            'Pref': "",
            'Flag': [-1, -1, -1, -1, -1, -1]
        }
        # JSON 形式のデータとして書き込む
        with open(conf_file_name, 'w') as conf_file:
            json.dump(conf, conf_file, ensure_ascii=False, indent=4, sort_keys=False, ⏎
separators=(",", ": "))
    else :
        # JSON 形式のデータとして読み込む
        with open(conf_file_name, 'r') as conf_file:
            conf = json.load(conf_file)

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <csvfile>" % sys.argv[0])
        sys.exit(os.EX_OK)

    # conf ファイルを読み込み
    readconf(os.path.splitext(sys.argv[0])[0] + ".json")

    # CSV ファイルオープン
    with open(sys.argv[1]) as csvfile:
        csvreader = csv.reader(csvfile)
        pzip = "00000000"
        csvline = [] # CSV データ初期化
        # CSV ファイルから 1 行読み込む
        for row in csvreader:
            # 要素数が不足している場合、次行にスキップ
            if len(row) < COLUMN_NUM:
                continue

            # 新しいデータの場合
            if pzip != row[1]:
                printline(csvline)
                csvline = copy.copy(row)
            else:
                # 既存データへの追加の場合
                # 町域名の続きを追加
                csvline[2] += row[2]
            pzip = row[1]

        # 保持済みのデータを表示
        printline(csvline)
```

```
        # データ総数を表示
        printline(None)
```

図 7.8 CSV ファイルの内容を表示するプログラムの JSON 形式の conf ファイル対応版 (dispcsv3.py)

内容としては、「図 7.6. 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.py)」の readconf 関数で configparser パッケージを使っている箇所を JSON パッケージを使うように置き換えただけです。加えて、configparser とは違い、設定ファイルを読み込んだ際に、数値は数値として読み込まれるので、文字列を数値に変換する必要はありません。

初回実行した後に、dispcsv3.json ファイルができています。

```
{
    "display": {
        "Width": 60,
        "Height": 17
    },
    "control": {
        "Pause": true,
        "Count": true
    },
    "data": {
        "Code": -1,
        "Zipcode": "",
        "Street": "",
        "City": "",
        "Pref": "",
        "Flag": [
            -1,
            -1,
            -1,
            -1,
            -1,
            -1
        ]
    }
}
```

ここでも、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

```
{
    "display": {
        "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 dispcsv3.json を変更して実行すると、「図 7.7. dispcsv2.conf を編集した dispcsv2.py の実行結果」と同じ結果となります。

```
[armadillo ~]# python3 ./dispcsv3.py ./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 7.9 dispcsv3.json を編集した dispcsv3.py の実行結果

7.2.4. XML ファイルを扱う

XML ファイルは JSON と同様に広く使われているファイルフォーマットです。

Python では XML を扱うパッケージも標準で用意されているので、C 言語で実装する場合と比較すると簡単に扱うことができます。

ここでも、「7.2.2. 設定ファイルに対応する」で取り上げた設定ファイルを、XML 形式として保存・読み込みを行うサンプルプログラムを示します。

```
import csv
import sys
import os
import copy
from xml.etree import ElementTree # XML 操作用パッケージのインポート
import xml.dom.minidom            # XML 操作用パッケージのインポート

# 表示する文字数
DISP_WIDTH = 60  # 幅
DISP_HEIGHT = 17 # 高さ

# CSV データ 1 行の要素数
COLUMN_NUM = 11

line = 0   # 一画面に表示した行数
count = 0  # 表示したデータ総数

conf = {} # 設定ファイル管理オブジェクト

def printline(csvline):

    """ 行表示関数

    Args:
        csvline(string の配列): 表示する 1 行分の CSV データ

    """

    global line
    global count

    # CSV データの各要素を表示
    if csvline is not None:
        # データがない場合表示しない
        if len(csvline) < 1:
            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 int(conf['data']['Code']) >= 0 and csvline[0] != conf['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 conf['data']['Zipcode'] != None and conf['data']['Zipcode'] not in csvline[1]:
            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 conf['data']['Street'] != None and conf['data']['Street'].lower() not in
csvline[2].lower():
            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 conf['data']['City'] != None and conf['data']['City'].lower() not in csvline[3].lower():
            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 conf['data']['Pref'] != None and conf['data']['Pref'].lower() not in csvline[4].lower():
            return
```

```
        for idx, flag in enumerate(conf['data']['Flag']):
            if flag >= 0 and flag != csvline[5 + idx]:
                return

        #各要素を表示フォーマットに展開
        buf = "%6d %s %s %s %s %s %s %s %s %s %s %s" % ¥
                (count, csvline[0], csvline[1], csvline[2], csvline[3], csvline[4], ¥
                 csvline[5], csvline[6], csvline[7], csvline[8], csvline[9], csvline[10])
        count += 1
    else:
        # CSV データの総数を表示
        # 総数表示無効なら表示しない
        if not conf['control']['Count']:
            return
        # データ総数を表示フォーマットに展開
        buf = "Count: %6d" % (count)

    # 今回追加される表示行数を計算
    disp_width = conf['display']['Width']
    disp_height = conf['display']['Height']
    newline = int((len(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 conf['control']['Pause']:
            input()
        # 表示行数を初期化
        line = 0
    # 実際に表示する
    print(buf)
    # 表示行数を更新
    line += newline


def readconf(conf_file_name):

    """ conf ファイル読み込み関数

    Args:
        conf_file_name(string): conf ファイルのファイル名

    """

    global conf
    if not os.path.exists("./" + conf_file_name):
        # ファイルが存在していない場合は新規作成
        # デフォルト設定
        conf['display'] = {
            'Width': 60,
            'Height': 17
        }
        conf['control'] = {
            'Pause': True,
            'Count': True
        }
        conf['data'] = {
            'Code': -1,
            'Zipcode': "",
```

```
            'Street': "",
            'City': "",
            'Pref': "",
            'Flag': [-1, -1, -1, -1, -1, -1]
        }

        # conf 要素(ルート)の作成
        conf_elem = ElementTree.Element("conf")

        # display 要素の作成とその下にある子要素の作成
        display_elem = ElementTree.SubElement(conf_elem, "display")
        width_elem = ElementTree.SubElement(display_elem, "width")
        width_elem.text = str(conf['display']['Width'])
        height_elem = ElementTree.SubElement(display_elem, "height")
        height_elem.text = str(conf['display']['Height'])

        # control 要素の作成とその下にある子要素の作成
        control_elem = ElementTree.SubElement(conf_elem, "control")
        pause_elem = ElementTree.SubElement(control_elem, "pause")
        pause_elem.text = str(conf['control']['Pause'])
        count_elem = ElementTree.SubElement(control_elem, "count")
        count_elem.text = str(conf['control']['Count'])

        # data 要素の作成とその下にある子要素の作成
        data_elem = ElementTree.SubElement(conf_elem, "data")
        code_elem = ElementTree.SubElement(data_elem, "code")
        code_elem.text = str(conf['data']['Code'])
        zipcode_elem = ElementTree.SubElement(data_elem, "zipcode")
        zipcode_elem.text = ""
        street_elem = ElementTree.SubElement(data_elem, "street")
        street_elem.text = ""
        city_elem = ElementTree.SubElement(data_elem, "city")
        city_elem.text = ""
        pref_elem = ElementTree.SubElement(data_elem, "pref")
        pref_elem.text = ""
        flag_elem = ElementTree.SubElement(data_elem, "flag")
        flag_elem.text = "-1,-1,-1,-1,-1,-1"

        # ファイルに書き出したときに見やすくするため整形する
        doc = xml.dom.minidom.parseString(ElementTree.tostring(conf_elem, "utf-8"))
        with open(conf_file_name, "w") as conf_file:
            doc.writexml(conf_file, newl="¥n", indent="", addindent="    ")
    else :
        # XML 形式のデータとして読み込む
        conf_elem = ElementTree.parse(conf_file_name)
        conf['display'] = {}
        conf['display']['Width'] = int(conf_elem.find(".//width").text)
        conf['display']['Height'] = int(conf_elem.find(".//height").text)
        conf['control'] = {}
        conf['control']['Pause'] = bool(conf_elem.find(".//pause").text)
        conf['control']['Count'] = bool(conf_elem.find(".//count").text)
        conf['data'] = {}
        conf['data']['Code'] = int(conf_elem.find(".//code").text)
        conf['data']['Zipcode'] = conf_elem.find(".//zipcode").text
        conf['data']['Street'] = conf_elem.find(".//street").text
        conf['data']['City'] = conf_elem.find(".//city").text
        conf['data']['Pref'] = conf_elem.find(".//pref").text
```

```
        flag = []
        for val in conf_elem.find(".//flag").text.split(","):
            flag.append(int(val))
        conf['data']['Flag'] = flag

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <csvfile>" % sys.argv[0])
        sys.exit(os.EX_OK)

    # conf ファイルを読み込み
    readconf(os.path.splitext(sys.argv[0])[0] + ".xml")

    # CSV ファイルオープン
    with open(sys.argv[1]) as csvfile:
        csvreader = csv.reader(csvfile)
        pzip = "00000000"
        csvline = [] # CSV データ初期化
        # CSV ファイルから 1 行読み込む
        for row in csvreader:
            # 要素数が不足している場合、次行にスキップ
            if len(row) < COLUMN_NUM:
                continue

            # 新しいデータの場合
            if pzip != row[1]:
                printline(csvline)
                csvline = copy.copy(row)
            else:
                # 既存データへの追加の場合
                # 町域名の続きを追加
                csvline[2] += row[2]
            pzip = row[1]

        # 保持済みのデータを表示
        printline(csvline)
        # データ総数を表示
        printline(None)
```

図 7.10 CSV ファイルの内容を表示するプログラムの XML 形式の conf ファイル対応版 (dispcsv4.py)

実行方法や実行結果に関してはこれまでのサンプルプログラムと同じなのでここでは省略し、初回実行時に生成される XML ファイルのみ示します。

```
<?xml version="1.0" ?>
<conf>
    <display>
        <width>60</width>
        <height>17</height>
    </display>
    <control>
        <pause>True</pause>
        <count>True</count>
    </control>
```

```
    <data>
        <code>-1</code>
        <zipcode/>
        <street/>
        <city/>
        <pref/>
        <flag>-1,-1,-1,-1,-1,-1</flag>
    </data>
</conf>
```

XML ファイル内の設定値を変更してプログラムを実行すると、JSON ファイル版のプログラムと同じように動作することを確認できます。

ここで取り上げたものだけではなく、Python には様々なファイル形式に対応したパッケージがあるので、何かファイルを処理したい場合はまずパッケージがないか探してみると見つかる可能性があります。

7.3. ネットワークを使う

Python でもソケットを使ったネットワークプログラムを書くことができます。

TCP/IP プログラムの流れは「6.6.1. TCP/IP」を参照してください。Python でもこの流れと同じです。

7.3.1. Python 版 TCP/IP で Hello!

「6.6.2. TCP/IP で Hello!」の Python 版を作ってみます。

```
import socket
import sys
import os
import traceback

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <port>" % sys.argv[0])
        sys.exit(os.EX_OK)

    port = int(sys.argv[1])
    if port < 49152 or 65535 < port:
        print("Specify the port 49152-65535¥n")
        sys.exit(os.EX_OK)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
        # アドレスとポートを割り当て
        server.bind(("", port))

        # クライアントからの接続を待つ
        server.listen(socket.SOMAXCONN)

        # 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
        client, client_addr = server.accept()

        try:
            # メッセージを送信
            client.sendall(b'Hello!¥r¥n')
```

```
        except:
            print(traceback.format_exc())
```

図 7.11 ネットワークで Hello!を返すサーバー Python 版(network_hello_server.py)

ソケット作成から、メッセージの送信までの流れは C 言語版と同じです。

実際に動作させてみます。

```
[armadillo ~]# python3 ./network_hello_server.py 65432
```

図 7.12 network_hello_server.py 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!
Connection closed by foreign host.
```

図 7.13 network_hello_server への telnet

このように Hello!と表示されることが確認できました。

7.3.2. HTTP 通信を行う

Python で通信系アプリケーションを開発しようとする場合は、Socket のような低位層のインターフェースを使ったものより、HTTP のような上位層の通信を要求されることのほうが多いかもしれません。

ここでは Python における HTTP 通信の例を紹介します。

7.3.2.1. HTTP サーバーを準備する

ATDE7 では、標準状態で lighttpd という HTTP サーバが可動していますので、これを利用します。

ATDE を起動している状態で、他の PC などからウェブブラウザで ATDE の IP アドレスにアクセスすると lighttpd の初期画面が表示されます。

この画面を簡単な html に置き換えます。

```
<html>
    <body>
        hello, Python.
    </body>
</html>
```

図 7.14 取得する index.html ファイル(index.html)

作成した html ファイルを、www ディレクトリに配置してパーミッションを変更します。

```
[ATDE ~]$ mv index.html /var/www/html
[ATDE ~]$ sudo chown www-data:www-data /var/www/html/index.html
```

ここまでで、再度ブラウザから ATDE の IP アドレスにアクセスすると、「hello, Python.」と表示されることが確認できます。

7.3.2.2. HTTP サーバーに接続する

HTTP サーバーにアクセスする Python プログラムは、urllib パッケージを使うと簡単に書けます。

```
import sys
import os
import traceback
import urllib.request

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <url>" % sys.argv[0])
        sys.exit(os.EX_OK)

    url = sys.argv[1]
    # 接続先 URL と送信パラメータからリクエストを作成
    request = urllib.request.Request(url)
    try:
        # 接続してレスポンスを取得
        with urllib.request.urlopen(request) as response:
            # レスポンスを読み込む
            body = response.read()
            print(body.decode())
    except:
        print(traceback.format_exc())
```

図 7.15 HTTP サーバーにアクセスするプログラム(http_access.py)

実行すると、index.html の内容が表示されます。

```
[armadillo ~]# python3 ./http_access.py http://(ATDE の IP アドレス)
<html>
    <body>
        hello, Python.
    </body>
</html>
```

図 7.16 http_access.py の実行結果

URL にパラメータをつけてリクエストすることもできます。

```
import sys
import os
import traceback
import urllib.request
```

```
if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <url>" % sys.argv[0])
        sys.exit(os.EX_OK)

    url = sys.argv[1]
    # 送信するパラメータを準備
    params = {
        "pref": "hokkaido",
        "city": "sapporo"
    }

    # 接続先 URL と送信パラメータからリクエストを作成
    request = urllib.request.Request("%s?%s" % (url, urllib.parse.urlencode(params)))
    try:
        # 接続してレスポンスを取得
        with urllib.request.urlopen(request) as response:
            # レスポンスを読み込む
            body = response.read()
            print(body.decode())
    except:
        print(traceback.format_exc())
```

図 7.17 パラメータ付きで HTTP サーバーにアクセスするプログラム(http_access_param.py)

ここまでは GET リクエストでしたが、POST でリクエストすることもできます。

POST で JSON データを送信するプログラム例です。

```
import sys
import os
import traceback
import urllib.request
import json

if __name__ == "__main__":
    # 引数が指定されなかった場合、usage 表示して終了
    if len(sys.argv) < 2:
        print("Usage: %s <url>" % sys.argv[0])
        sys.exit(os.EX_OK)

    url = sys.argv[1]
    # 送信する JSON データを準備
    data = {
        "pref": "hokkaido",
        "city": "sapporo"
    }
    # 送信ヘッダを準備
    headers = {
        'Content-Type': 'application/json'
    }

    # 接続先 URL と送信パラメータからリクエストを作成
    request = urllib.request.Request(url, json.dumps(data).encode(), headers)
    try:
```

```
        # 接続してレスポンスを取得
        with urllib.request.urlopen(request) as response:
            # レスポンスを読み込む
            body = response.read()
            print(body.decode())
    except:
        print(traceback.format_exc())
```

図 7.18 パラメータ付きで HTTP サーバーにアクセスするプログラム(http_access_post.py)

7.4. データベースを使う

センサーなどから取得したデータやサーバーから受信したデータを保存しておく場合、ファイルとして保存しておくのも手ですが、データ同士に関連があったりまとまったデータに対して、なにか条件をつけて目的のデータを検索したいとなった場合、データベースを構築してそこに保存しておく方法もあります。

ここでは Python でデータベースを扱う例について取り上げます。

Python 向けに peewee というデータベースを扱うパッケージがあるので、それを使います。

peewee は SQLite、MySQL、PostgreSQL、CockroachDB といったデータベースシステムを扱えますが、ここでは組み込み向けに利用されることが多い SQLite を選択します。

まずは、pip で peewee をインストールします。

```
[armadillo ~]# pip3 install peewee
```

例として「6.3.1. テキストファイルを扱う」で使用した、住所の CSV ファイルのデータをデータベースに保存するプログラムを示します。

仕様は以下のとおりです。

1. データベースファイルの名前は address.db とする
2. CSV ファイルの先頭から 10 件分の住所をデータベースに保存する
3. 保存するデータは、郵便番号、都道府県、市区町村、住所とする
4. サブコマンドとして add、show、update、delete の 4 つを用意する
5. add -F <CSV ファイル名> で読み込む CSV ファイルを指定する
6. show で保存しているデータを表示する
7. update -I <id> -C <column 名> -V <値> で指定した id のデータの<column 名>を<値>に変更する
8. delete -I <id> で指定した id のデータを削除する

```
import sys
import os
import csv
```

```
import traceback
import argparse
from peewee import *

# CSV データ 1 行の要素数
COLUMN_NUM = 11

# データベースファイルのオープン
db = SqliteDatabase("address.db")

# 住所クラスの定義
class Address(Model):
    zipcode = CharField()
    pref = CharField()
    city = CharField()
    street = CharField()

    class Meta:
        database = db

# Address テーブルの作成
db.create_tables([Address])

def add(args):

    """ データベースへのデータの追加

    Args:
        args: コマンド引数
            args.file: CSV ファイル名

    """

    with open(args.file) as csvfile:
        csvreader = csv.reader(csvfile)
        count = 0
        try:
            # トランザクションの開始
            with db.transaction():
                # CSV ファイルから 1 行読み込む
                # 10 行目まで読み込む
                for row in csvreader:
                    # 要素数が不足している場合、次行にスキップ
                    if len(row) < COLUMN_NUM:
                        continue
                    if count >= 10:
                        break

                    # データベースへのデータの追加(INSERT)
                    Address.create(zipcode=row[1], pref=row[4], city=row[3], street=row[2])

                    count += 1

                # トランザクション終了 データをコミット
                db.commit()
        except:
            # エラー発生の場合データベースをロールバック
            db.rollback()
```

```
            print(traceback.format_exc())

def show(args):

    """ データベースに保存されているデータの表示

    Args:
        args: コマンド引数

    """

    for address in Address.select():
        print("%s %s %s %s %s" ¥
                % (address.id, address.zipcode, address.pref, address.city, address.street))

def update(args):

    """ データベースに保存されているデータを変更する

    Args:
        args: コマンド引数
            args.id: 変更したいデータの id
            args.column: 変更したいデータのカラム名
            args.val: 変更後の値

    """

    try:
        # トランザクションの開始
        with db.transaction():
            # id を指定してデータを取得
            address = Address.get(id=args.id)

            # 指定されたカラム名毎に値を変更
            if args.column == "zipcode":
                address.zipcode = args.val
            elif args.column == "pref":
                address.pref = args.val
            elif args.column == "city":
                address.city = args.val
            elif args.column == "street":
                address.street = args.val

            # 変更を保存してトランザクション終了
            address.save()
            db.commit()
    except:
        # エラー発生の場合データベースをロールバック
        db.rollback()
        print(traceback.format_exc())

def delete(args):

    """ データベースに保存されているデータを削除

    Args:
        args: コマンド引数
            args.id: 削除したいデータの id
```

```
    """

    try:
        # トランザクションの開始
        with db.transaction():
            # id を指定してデータを取得
            address = Address.get(id=args.id)

            # データを削除
            address.delete_instance()

            # トランザクション終了
            db.commit()
    except:
        # エラー発生の場合データベースをロールバック
        db.rollback()
        print(traceback.format_exc())

def commandline_parser():

    """ コマンドオプションとハンドラを設定する

    """

    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers()

    # add コマンド
    add_command = subparsers.add_parser("add")
    add_command.add_argument("-F", "--file", required=True, help="csv file")
    add_command.set_defaults(handler=add)

    # show コマンド
    show_command = subparsers.add_parser("show")
    show_command.set_defaults(handler=show)

    # update コマンド
    update_command = subparsers.add_parser("update")
    update_command.add_argument("-I", "--id", required=True, help="id number")
    update_command.add_argument("-C", "--column", required=True, help="column name")
    update_command.add_argument("-V", "--val", required=True, help="new value")
    update_command.set_defaults(handler=update)

    # delete コマンド
    delete_command = subparsers.add_parser("delete")
    delete_command.add_argument("-I", "--id", required=True, help="id number")
    delete_command.set_defaults(handler=delete)

    return parser

if __name__ == "__main__":
    parser = commandline_parser()
    args = parser.parse_args()
    if hasattr(args, 'handler'):
        # ハンドラが登録されていれば実行
        args.handler(args)
    else:
```

```
        # 未知のサブコマンドの場合はヘルプを表示
        parser.print_help()
```

図 7.19 データベースを操作するプログラム(handling_database.py)

実際に動作させてみます。まず、CSV ファイルからデータベースに保存します。

サブコマンド add にオプション-F で CSV ファイルを指定して実行します。

```
[armadillo ~]# python3 ./handling_database.py add -F ./ken_all_rome.csv
```

図 7.20 handling_database.py の実行結果(add サブコマンド)

サブコマンド show でデータが保存されていることが確認できます。

```
[armadillo ~]# python3 ./handling_database.py show
1 0600000 HOKKAIDO CHUO-KU SAPPORO-SHI IKANIKEISAIGANAIBAAI
2 0640941 HOKKAIDO CHUO-KU SAPPORO-SHI ASAHIGAOKA
3 0600041 HOKKAIDO CHUO-KU SAPPORO-SHI ODORIHIGASHI
4 0600042 HOKKAIDO CHUO-KU SAPPORO-SHI ODORINISHI(1-19-CHOME)
5 0640820 HOKKAIDO CHUO-KU SAPPORO-SHI ODORINISHI(20-28-CHOME)
6 0600031 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JOHIGASHI
7 0600001 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(1-19-CHOME)
8 0640821 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(20-28-CHOME)
9 0600032 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JOHIGASHI
10 0600002 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JONISHI(1-19-CHOME)
```

図 7.21 handling_database.py の実行結果(show サブコマンド)

次に、サブコマンド update で 5 番目のデータを変更してみます。

```
[armadillo ~]# python3 ./handling_database.py update -I 5 -C zipcode -V 9000002
[armadillo ~]# python3 ./handling_database.py update -I 5 -C pref -V OKINAWA
[armadillo ~]# python3 ./handling_database.py update -I 5 -C city -V NAHA-SHI
[armadillo ~]# python3 ./handling_database.py update -I 5 -C street -V AKEBONO
[armadillo ~]# python3 ./handling_database.py show
1 0600000 HOKKAIDO CHUO-KU SAPPORO-SHI IKANIKEISAIGANAIBAAI
2 0640941 HOKKAIDO CHUO-KU SAPPORO-SHI ASAHIGAOKA
3 0600041 HOKKAIDO CHUO-KU SAPPORO-SHI ODORIHIGASHI
4 0600042 HOKKAIDO CHUO-KU SAPPORO-SHI ODORINISHI(1-19-CHOME)
5 9000002 OKINAWA NAHA-SHI AKEBONO
6 0600031 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JOHIGASHI
7 0600001 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(1-19-CHOME)
8 0640821 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(20-28-CHOME)
9 0600032 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JOHIGASHI
10 0600002 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JONISHI(1-19-CHOME)
```

図 7.22 handling_database.py の実行結果(update サブコマンド)

5 番目のデータが変更されていることが確認できます。

最後に、サブコマンド delete で 6 番目のデータを削除してみます。

```
[armadillo ~]# python3 ./handling_database.py delete -I 6
[armadillo ~]# python3 ./handling_database.py show
1 0600000 HOKKAIDO CHUO-KU SAPPORO-SHI IKANIKEISAIGANAIBAAI
2 0640941 HOKKAIDO CHUO-KU SAPPORO-SHI ASAHIGAOKA
3 0600041 HOKKAIDO CHUO-KU SAPPORO-SHI ODORIHIGASHI
4 0600042 HOKKAIDO CHUO-KU SAPPORO-SHI ODORINISHI(1-19-CHOME)
5 9000002 OKINAWA NAHA-SHI AKEBONO
7 0600001 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(1-19-CHOME)
8 0640821 HOKKAIDO CHUO-KU SAPPORO-SHI KITA1-JONISHI(20-28-CHOME)
9 0600032 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JOHIGASHI
10 0600002 HOKKAIDO CHUO-KU SAPPORO-SHI KITA2-JONISHI(1-19-CHOME)
```

図 7.23 handling_database.py の実行結果(delete サブコマンド)

6 番目のデータが削除されていることが確認できます。

このように、Python から SQL データベースを複雑な SQL 文を意識することなく、操作できます。

peewee には他にも様々な機能がありますので詳細は公式サイト[3]を参照してください。

ここで作成したデータベースファイル address.db は Windows などにある SQL データベースブラウザのようなアプリケーションで中身を直接確認することもできます。

7.5. プログラムをデバッグする

Python でアプリケーションを開発しても、C 言語に比べてコード量が少ないとはいえバグは存在します。Python はスクリプト言語なので、怪しいと思った箇所に print 文を埋め込んでコンパイル不要ですぐに動作を確認することができますが、それでも数が多くなると print 文を埋め込むのもデバッグ後に削除するのも大変です。

ここでは Python プログラムのデバッグ手法について紹介します。

7.5.1. トレースバックを読む

Python では、プログラムがクラッシュした場合、その理由やプログラムのどこで発生したのかなどの情報を含んだトレースバックを出力します。ほとんどの場合は、このトレースバックを読んで原因箇所を修正すれば解決します。

以下は 0 除算を行う可能性のあるプログラムです。

```
import os
import sys

def divide(a, b):
    """ a を b で割った商を返す
    """
    c = a / b
    return c

if __name__ == "__main__":
    n = [5, 2, 4, 0, 1, 3]
    for i in n:
```

[3]http://docs.peewee-orm.com/en/latest/(英語)

```
        d = divide(10, i)
        print(d)
```

図 7.24 0 除算を行う可能性のあるプログラム (dividezero.py)

実行すると以下のようになります。

```
[armadillo ~]# python3 ./dividezero.py
2.0
5.0
2.5
Traceback (most recent call last):
  File "dividezero.py", line 13, in <module>
    d = divide(10, i)
  File "dividezero.py", line 7, in divide
    c = a / b
ZeroDivisionError: division by zero
```

図 7.25 dividezero.py の実行結果

トレースバックが出力されています。下から上に向かって読むと、まず

```
ZeroDivisionError: division by zero
```

このメッセージで 0 除算が発生していることが分かります。次に

```
  File "dividezero.py", line 7, in divide
    c = a / b
```

ここで、divide 関数の中、プログラム 7 行目で発生していることが分かります。

別な例として、配列の範囲外へのアクセスが発生した場合です。

```
import os
import sys

def printarray(a):
    """ 配列を表示する
    """
    for i in range(0, 10):
        print(a[i])

if __name__ == "__main__":
    n = [5, 2, 4, 0, 1, 3]
    printarray(n)
```

図 7.26 配列の範囲外へアクセスする可能性のあるプログラム (outofrange.py)

実行すると以下のようになります。

```
[armadillo ~]# python3 ./outofrange.py
5
2
4
0
1
3
Traceback (most recent call last):
  File "outofrange.py", line 12, in <module>
    printarray(n)
  File "outofrange.py", line 8, in printarray
    print(a[i])
IndexError: list index out of range
```

図 7.27 outofrange.py の実行結果

トレースバックの内容から、プログラム 8 行目で配列の範囲外へのアクセスが発生していることが分かります。

このように、クラッシュが発生するようなバグの場合はトレースバックに表示される情報ですぐに発生原因と発生箇所を特定できるので、まずはトレースバックに目を通すのがよいです。

7.5.2. デバッガを使う

クラッシュはしないが期待していた動作と違うといった場合は、デバッガを使うのが有効です。

Python では標準で pdb というデバッガが用意されていますので、これを使います。

デバッグ対象の簡単なプログラムです。

```
import os
import sys

def sum(a):
    """ 1 から a までの和を計算する
    """
    s = 0
    for i in range(1, a):
        s += i

    return s

if __name__ == "__main__":
    s = sum(100)
    print(s)
```

図 7.28 1 から 100 までの和を計算するプログラム (sum.py)

このプログラムを実行すると以下のように表示されます。

```
[armadillo ~]# python3 ./sum.py
4950
```

図 7.29 sum.py の実行結果

結果として 5050 と表示されるはずですが期待と違っていました。pdb を使ってデバッグしてみます。

pdb を使うためにプログラム実行時にオプション引数を渡します。

```
[armadillo ~]# python3 -m pdb sum.py
> /home/atmark/workspace/source/sum.py(1)<module>()
-> import os
(Pdb)
```

1 行目で止まり、(Pdb)プロンプトが表示されました。今後はこのプロンプトにコマンドを入力してデバッグを進めます。next と入力すると次の行へ進んで実行が止まります。

```
[armadillo ~]# python3 -m pdb sum.py
> /root/work/developers_guide/python/debug/sum.py(1)<module>()
-> import os
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(2)<module>()
-> import sys
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(4)<module>()
-> def sum(a):
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(11)<module>()
-> if __name__ == "__main__":
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(12)<module>()
-> s = sum(100)
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(13)<module>()
-> print(s)
```

ここで p コマンドを使うと、変数 s に入っている値を表示することができます。

```
(Pdb) p s
4950
```

continue コマンドを使うとプログラムを最後まで実行します。最後まで実行された後は、再び最初から実行が開始されます。

```
(Pdb) continue
4950
The program finished and will be restarted
> /root/work/developers_guide/python/debug/sum.py(1)<module>()
-> import os
```

pdb を終了するには quit と入力します。

```
(Pdb) quit
[armadillo ~]#
```

再び、pdb を開始し sum 関数の呼び出し箇所まで処理を進めます。

```
# python3 -m pdb sum.py
> /root/work/developers_guide/python/debug/sum.py(1)<module>()
-> import os
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(2)<module>()
-> import sys
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(4)<module>()
-> def sum(a):
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(11)<module>()
-> if __name__ == "__main__":
(Pdb) next
> /root/work/developers_guide/python/debug/sum.py(12)<module>()
-> s = sum(100)
```

ここで、next ではなく step と入力すると sum 関数の中へ入ることができます。next も step も 1 行ずつ実行するコマンドですが、next は関数の中には入らず、step は関数の中に入ります。

```
(Pdb) step
--Call--
> /root/work/developers_guide/python/debug/sum.py(4)sum()
-> def sum(a):
```

sum 関数の中へ入り、処理が止まりました。

ここまで、next や step で一行ずつ実行してきましたが、目的の行にたどり着くまで何度も入力するのは大変です。

そこで、ブレークポイントを設定しそこまで一気に処理を実行してみます。

まず、ブレークポイントを設定する行を確認するために、list コマンドでソースコードを表示します。

```
[armadillo ~]# python3 -m pdb sum.py
> /home/atmark/work/developer_guide/python/sum.py(1)<module>()
-> import os
(Pdb) list
  1  -> import os
  2     import sys
  3
  4     def sum(a):
  5         s = 0
  6         for i in range(1, a):
  7             s += i
  8
```

```
  9         return s
 10
 11     if __name__ == "__main__":
(Pdb)
```

ソースコード内にコメントがある場合はコメントも表示されます。この時、日本語のコメントがあった場合、環境によっては文字エンコードに関する例外が発生することがありますが、デバッグは可能です。

break コマンドで 7 行目にブレークポイントを設定し、continue コマンドで設定した行まで一気に実行します。

```
(Pdb) break 7
Breakpoint 1 at /root/work/developers_guide/python/debug/sum.py:7
(Pdb) continue
> /root/work/developers_guide/python/debug/sum.py(7)sum()
-> s += i
(Pdb) p s
0
(Pdb) p i
1
```

ブレークポイントで実行が停止しました。そこでの変数 s と i の値も表示しています。

display コマンドを使うと値が変化した変数が自動的に表示されるようになります。

```
(Pdb) display s
display s: 0
(Pdb) display i
display i: 1
(Pdb) continue
> /root/work/developers_guide/python/debug/sum.py(7)sum()
-> s += i
display s: 1  [old: 0]
display i: 2  [old: 1]
(Pdb) continue
> /root/work/developers_guide/python/debug/sum.py(7)sum()
-> s += i
display s: 3  [old: 1]
display i: 3  [old: 2]
(Pdb) continue
> /root/work/developers_guide/python/debug/sum.py(7)sum()
-> s += i
display s: 6  [old: 3]
display i: 4  [old: 3]
```

for 文を抜けるまで continue し続けるのは大変なので、clear コマンドでブレークポイントを削除し、return コマンドで関数の最後まで処理を実行します。

clear コマンドでブレークポイントを削除する時は、ソースコードの行数ではなくブレークポイント番号を指定する必要があります。ブレークポイント番号は、break と入力することで確認できます。

```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /root/work/developers_guide/python/debug/sum.py:7
        breakpoint already hit 6 times
(Pdb) clear 1
Deleted breakpoint 1 at /root/work/developers_guide/python/debug/sum.py:7
(Pdb) return
--Return--
> /root/work/developers_guide/python/debug/sum.py(9)sum()->4950
-> return s
display s: 4950  [old: 15]
display i: 99  [old: 6]
```

ブレークポイントを削除し、関数の最後まで処理を進めたところ変数 i の値が 99 となっており、for ループが 99 回しか実行されておらず、for ループの終了条件に問題があることがわかりました。

このように、デバッガを使うとデバッグのための print 文を埋め込むことなく処理の流れや変数の内容を確認できるので、効率的にデバッグができるようになります。

最後に、ここでの説明で使用した pdb のコマンド一覧を示します。またほとんどのコマンドは、省略形でも使うことができます。

表 7.1 使用したコマンド

コマンド	省略形	動作
step	s	1 行ずつ実行し、関数の場合は中に入る
next	n	1 行ずつ実行し、関数には入らない
p	-	変数の値を表示する
continue	c	実行を再開する
quit	q	pdb を終了する
list	l	ソースコードを表示する
break n	b n	n 行目にブレークポイントを設定する
clear n	d n	n 番目のブレークポイントを削除する
return	r	関数を最後まで実行する
display	-	変化のあった変数の値を表示する

他にもコマンドがありますので詳細は公式サイト[4]を参照してください。

[4]https://docs.python.org/ja/3/library/pdb.html

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には systemd を利用します。

本章では、この方法について説明します。

8.1.1. systemd とは

systemd とは Linux のシステム管理デーモンの一つです。並列実行による効率的なシステム起動/終了や設定ファイルによるシステム管理の共通化、柔軟なプロセス起動を行うことができます。Armadillo-640 ではデフォルトのシステム管理デーモンに systemd を採用しています。

詳細な情報は公式サイトを参照してください。

https://www.freedesktop.org/wiki/Software/systemd/

8.1.2. 自動起動するコマンドの作成

起動時に自動実行するテストコマンドを作成します。一秒ごとに/tmp/system_init_test.log に"system_init_test"という文字列を書き込むだけの簡単なシェルスクリプトです。

```
[armadillo ~]# vi system_init_test.sh
[armadillo ~]# chmod +x system_init_test.sh ❶
```

❶　実行権限を付けます。

シェルスクリプトの内容は以下のように実装します。

```
#!/bin/bash

while true
do
   echo system_init_test >> /tmp/system_init_test.log
   sleep 1
done
```

図 8.1 シェルスクリプトの実装内容(system_init_test.sh)

8.1.3. Unit ファイルの作成

systemd では Unit と呼ばれる定義ファイルに起動時の定義を記載することで各起動処理を行っています。自作のコマンドの場合も、Unit ファイルを作成することでシステム起動時に自動起動することができます。

ここでは「system_init_test.service」という名前の Unit ファイルの設定を行います。

```
[armadillo ~]# vi /etc/systemd/system/system_init_test.service
```

Unit ファイルの内容は以下のように実装します。

```
[Unit]
Description = system_init_test daemon

[Service]
ExecStart = /root/system_init_test.sh
Restart = always
Type = simple

[Install]
WantedBy = multi-user.target
```

図 8.2 Unit ファイルの実装内容(system_init_test.service)

Unit ファイルの各項目と意味については以下の通りです。

- [Unit]セクション：起動時に必要な依存関係など
  - Description：この Unit の説明文を記載する。
  - Before：この Unit よりも後に起動する Unit を指定する。(指定した Unit より前に、この Unit を起動する。)
  - After：この Unit よりも前に起動する Unit を指定する。(指定した Unit より後に、この Unit を起動する。)
  - Wants：この Unit の起動に必要な Unit を指定する。可能な限り同時起動する。
  - Requires：この Unit の起動に必要な Unit を指定する。必ず同時起動する。
- [Service]セクション：起動時に必要なパラメータなど
  - Type：この Unit が起動完了したかどうかの判定方法を指定する。
    - simple：コマンドが実行された時に起動完了となる。
    - forking：コマンドが完了した時に起動完了となる。
    - oneshot：コマンドが完了した時に起動完了&終了となる。
    - notify：この Unit のプログラム内から systemd へシグナルを送った時に起動完了となる。
  - ExecStart：この Unit を起動する際に実行するコマンドを指定する。（絶対パスを指定）

- ExecReload：この Unit をリロードする際に実行するコマンドを指定する。
- ExecStop：この Unit を停止する際に実行するコマンドを指定する。
- Restart：この Unit のメインプロセスが停止した際の動作を指定する。
  - always：常に再起動を試みる。
  - no：再起動を行わない。（デフォルト値）
  - on-success：終了コード 0 で停止した際に再起動する。
  - on-failure：終了コード 0 以外で停止した際に再起動する。
- [Install]セクション：systemd で有効にした場合の設定など
  - WantedBy：この Unit が必要とする target を指定する。可能な限り同時起動する。
  - RequiredBy：この Unit が必要とする target を指定する。必ず同時起動する。

一般的な Unit の場合、WantedBy/RequiredBy に指定する値は multi-user.target または、graphical.target を指定します。

WantedBy/RequiredBy に target を指定すると /etc/systemd/system/[target].wants/配下に対象となる Unit へのシンボリックリンクが作成されます。

これは[Unit]セクションの Wants/Requires に依存関係が追加されるのと同じ効果を持ち、以下のように使い分けをします。

- システム的に必要な依存関係 → [Unit]セクションの Wants/Requires に記載
- システム管理者が環境に応じて設定する依存関係 → WantedBy/RequiredBy に記載

8.1.4. 自動起動の設定

systemctl コマンドを用いて自動起動する service の設定をします。

```
[armadillo ~]# systemctl enable system_init_test.service
Created symlink /etc/systemd/system/multi-user.target.wants/system_init_test.service -> /etc/ ⏎
systemd/system/system_init_test.service.

[armadillo ~]# systemctl list-unit-files | grep system_init_test
system_init_test.service              enabled ❶
```

❶ 有効になっていることを確認できます。

systemctl の start サブコマンドで service を起動できます。

```
[armadillo ~]# systemctl start system_init_test.service
[armadillo ~]# LANG=C systemctl status system_init_test.service ❶
* system_init_test.service - system_init_test daemon
   Loaded: loaded (/etc/systemd/system/system_init_test.service; enabled; vendor
   Active: active (running) since Tue 2019-04-16 09:49:53 JST; 46s ago
 Main PID: 747 (system_init_tes)
    Tasks: 2 (limit: 4915)
   CGroup: /system.slice/system_init_test.service
           |-747 /bin/bash /root/system_init_test.sh
           `-806 sleep 1
```

❶ service のステータスが確認できます。

LANG=C を設定しないと表示結果が文字化けすることがあります。

実行結果は以下の通り確認できます。

```
[armadillo ~]# tail -f /tmp/system_init_test.log
system_init_test
system_init_test
system_init_test
```

systemctl の stop サブコマンドで service を停止できます。

```
[armadillo ~]# systemctl stop system_init_test.service
```

Armadillo を再起動することで設定した service が自動起動されているかを確認できます。

```
[armadillo ~]# reboot
(省略)
[armadillo ~]# tail -f /tmp/system_init_test.log
system_init_test
system_init_test
system_init_test
```

8.1.5. 自動起動の解除

systemctl の disable サブコマンドで自動起動を解除できます。

```
[armadillo ~]# systemctl disable system_init_test.service
[armadillo ~]# systemctl list-unit-files | grep system_init_test
system_init_test.service              disabled ❶
```

❶ 解除されていることを確認できます。

8.1.6. 順序関係のあるコマンドの自動起動

あるコマンドを起動した後に別のコマンドを実行したい、といったような順序関係のあるコマンドの自動起動について説明します。

2 つのテストコマンドを作成します。/tmp/system_init_test_ab.log に"system_init_test_a","system_init_test_b"という文字列を書き込むだけの簡単なシェルスクリプトです。

```
[armadillo ~]# vi system_init_test_a.sh
[armadillo ~]# vi system_init_test_b.sh
[armadillo ~]# chmod +x system_init_test_a.sh
[armadillo ~]# chmod +x system_init_test_b.sh
```

シェルスクリプトの内容は以下のように実装します。

```
#!/bin/bash

echo system_init_test_a >> /tmp/system_init_test_ab.log
```

図 8.3 シェルスクリプトの実装内容(system_init_test_a.sh)

```
#!/bin/bash

echo system_init_test_b >> /tmp/system_init_test_ab.log
```

図 8.4 シェルスクリプトの実装内容(system_init_test_b.sh)

Unit ファイルを作成します。ここでは system_init_test_a.sh を起動した後に system_init_test_b.sh を起動するように設定します。

```
[armadillo ~]# vi /etc/systemd/system/system_init_test_a.service
[armadillo ~]# vi /etc/systemd/system/system_init_test_b.service
```

Unit ファイルの内容は以下のように実装します。

```
[Unit]
Description = system_init_test_a

[Service]
ExecStart = /root/system_init_test_a.sh
Type = oneshot

[Install]
WantedBy = multi-user.target
```

図 8.5 Unit ファイルの実装内容(system_init_test_a.service)

```
[Unit]
Description = system_init_test_b
After = system_init_test_a.service ❶

[Service]
ExecStart = /root/system_init_test_b.sh
Type = oneshot

[Install]
WantedBy = multi-user.target
```

図 8.6 Unit ファイルの実装内容(system_init_test_b.service)

❶ 先に起動する Unit を指定します。ここで指定する Unit(この場合は「図 8.5. Unit ファイルの実装内容(system_init_test_a.service)」)が "Type = oneshot" であることに注意してください。"Type = oneshot" であれば、system_init_test_b.service は、確実に system_init_test_a.service が終了した後に起動されます。

自動起動の設定を行います。

```
[armadillo ~]# systemctl enable system_init_test_a.service
[armadillo ~]# systemctl enable system_init_test_b.service
```

Armadillo を再起動して確認します。

```
[armadillo ~]# reboot
(省略)
[armadillo ~]# cat /tmp/system_init_test_ab.log
system_init_test_a
system_init_test_b ❶
```

❶ 設定した順に実行されていることが確認できます。

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、systemd を利用する方法を紹介します。

8.2.1. systemd で定期実行する

systemd で定期実行するには、定期実行したいプログラムを起動する Unit ファイルとは別に、もう一つ Unit ファイルを作成します。

ここでは、「8.1.6. 順序関係のあるコマンドの自動起動」で作成した「図 8.5. Unit ファイルの実装内容(system_init_test_a.service)」を 1 分毎に起動するようにしてみます。

system_init_test_a.sh、system_init_test_a.service の内容を確認します。もし削除してしまった場合は、もう一度作成してください。

```
[armadillo ~]# vi system_init_test_a.sh
[armadillo ~]# chmod +x system_init_test_a.sh
[armadillo ~]# vi /etc/systemd/system/system_init_test_a.service
```

シェルスクリプトの内容は以下のように実装します。

```
#!/bin/bash

echo system_init_test_a >> /tmp/system_init_test_ab.log
```

図 8.7 シェルスクリプトの実装内容(system_init_test_a.sh)

Unit ファイルの内容は以下のように実装します。

```
[Unit]
Description = system_init_test_a

[Service]
ExecStart = /root/system_init_test_a.sh
Type = oneshot

[Install]
WantedBy = multi-user.target
```

図 8.8 Unit ファイルの実装内容(system_init_test_a.service)

タイマーを設定するために、system_init_test_a.timer という Unit ファイルを system_init_test_a.service と同じディレクトリに作成します。

```
[armadillo ~]# vi /etc/systemd/system/system_init_test_a.timer
```

Unit ファイルの内容は以下のように実装します。

```
[Unit]
Description = launch system_init_test_a

[Timer]
Persistent = true
OnBootSec = 0s
OnCalendar = *-*-* *:*:00
AccuracySec = 0.1s

[Install]
WantedBy = timers.target
```

図 8.9 Unit ファイルの実装内容(system_init_test_a.timer)

Unit ファイル(.timer)の [Timer]の意味については以下の通りです。[Unit] と [Install] については、Unit ファイル(.service)と同じなので省きます。

- [Timer]セクション: 対象の Unit を起動するタイミングを指定する。
  - OnActiveSec: タイマーが active になってから指定時間が経過した後、対象の Unit を起動する。
  - OnBootSec: システムがブートしてから指定時間が経過した後、対象の Unit を起動する。
  - OnStartupSec: systemd がスタートしてから指定時間が経過した後、対象の Unit を起動する。
  - OnUnitActiveSec: 対象の Unit が最後に起動になってから指定時間が経過した後、対象の Unit を起動する。
  - OnUnitInactiveSec: 対象の Unit が最後に停止になってから指定時間が経過した後、対象の Unit を起動する。
  - OnCalendar: 指定時間になった時、対象の Unit を起動する。
    - 曜日 年-月-日 時:分:秒: 曜日(Mon, Tue, Wed, Thu, Fri, Sat, Sun)は省略できる。 各フィールドにはアスタリスク(*)も指定できる。 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る。

```
minutely: *-*-* *:*:00 と同じ。
hourly: *-*-* *:00:00 と同じ。
daily: *-*-* 00:00:00 と同じ。
quarterly: *-01,04,07,10-01 00:00:00 と同じ。
```

  - Persistent=: 電源停止などで前回未実行だった場合の動作を指定する。
    - true: 対象の Unit を即座に起動する。
    - false: 即座に起動はせず、次のタイマーの時に対象の Unit を起動する。
  - AccuracySec=: 対象の Unit を起動するタイミングの精度を指定する。デフォルトでは 1min 。 (秒単位の精度がほしい場合は 0.1s などとする。)

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 5 systemd.timer や man 7 systemd.time を参照してください。

タイマーの自動起動の設定を行います。また、system_init_test_a.service が system_init_test_a.timer 以外から起動されると混乱の元になってしまうので、system_init_test_a.service の自動起動しないように設定します。

```
[armadillo ~]# systemctl enable system_init_test_a.timer
[armadillo ~]# systemctl disable system_init_test_a.service
```

Armadillo を再起動して確認します。

```
[armadillo ~]# reboot
(省略)
```

```
[armadillo ~]# tail -f /tmp/system_init_test_ab.log
system_init_test_b ❶
system_init_test_a ❷
system_init_test_a ❸
```

❶ system_init_test_b.service には、After = system_init_test_a.service が設定されていますが、system_init_test_a.service の自動起動が無効化されているので、system_init_test_a.service とは無関係に system_init_test_b.service が起動しています。

❷ system_init_test_a.timer の OnBootSec=0s によって system_init_test_a.service が起動されています。

❸ しばらく待つと system_init_test_a.timer の OnCalendar=*-*-* *:*:00 によって system_init_test_a.service が起動されます。

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. systemd によるプロセスの自動再起動

「8.1.3. Unit ファイルの作成」で紹介したように、systemd 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、Unit ファイルの定義に従って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、systemd 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[1]。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 では、標準で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。

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。

[1] 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。

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。

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 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。

Armadillo-600 シリーズで採用している Debian GNU/Linux 9.0(コードネーム "stretch") では、rsyslogd が標準です。[2]

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。

ローテーションの設定は、/etc/logrotate.conf と/etc/logrotate.d/以下のファイルで行います。/etc/logrotate.d ディレクトリの中を見てみると apt や rsyslog などの Armadillo にインストールされているソフトウェア毎に、設定ファイルが作られていることがわかります。

/etc/logrotate.conf の中には、説明とともに設定が記述されています。

```
[armadillo ~]# cat /etc/logrotate.conf
# see "man logrotate" for details
# rotate log files weekly
weekly ❶

# keep 4 weeks worth of backlogs
rotate 4 ❷

# create new (empty) log files after rotating old ones
create ❸

# uncomment this if you want your log files compressed
#compress ❹

# packages drop log rotation information into this directory
include /etc/logrotate.d ❺
```

[2]Armadillo-400 シリーズや Armadillo-800 シリーズ等で採用していた Atmark Dist では、syslog デーモンとして BusyBox の syslogd を使用していました。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。

```
# no packages own wtmp, or btmp -- we'll rotate them here
/var/log/wtmp { ❻
    missingok
    monthly
    create 0664 root utmp
    rotate 1
}

/var/log/btmp {
    missingok
    monthly
    create 0660 root utmp
    rotate 1
}

# system-specific logs may be configured here ❼
```

❶ ログファイルをローテーションする頻度を weekly (毎週) に設定しています。

❷ ローテーションしたログファイルを ４つ まで保存するように設定しています。

❸ ログファイルをローテーションした直後に、空のログファイルを作成するように設定しています。

❹ コメントアウトされているのでこの設定は無効です。コメントアウトを外すとローテーションしたログファイルを圧縮するように設定します。

❺ /etc/logrotate.d ディレクトリから設定ファイルを読み込んでいます。

❻ /var/log/wtmp というログファイルに対して、{}で囲まれた行に記述された設定を適用します。

❼ 最後の行に設定を記述することで、その前までの行で記述された設定を上書きすることができます。

rsyslog のローテーションの設定を変更したい場合は、/etc/logrotate.d/rsyslog を修正します。

```
[armadillo ~]# cat /etc/logrotate.conf
/var/log/syslog
{ ❶
        rotate 7
        daily
        missingok
        notifempty
        delaycompress
        compress
        postrotate
                invoke-rc.d rsyslog rotate > /dev/null
        endscript
}

/var/log/mail.info
/var/log/mail.warn
/var/log/mail.err
/var/log/mail.log
/var/log/daemon.log
/var/log/kern.log
/var/log/auth.log
/var/log/user.log
/var/log/lpr.log
/var/log/cron.log
```

```
/var/log/debug
/var/log/messages
{ ❷
        rotate 4
        weekly
        missingok
        notifempty
        compress
        delaycompress
        sharedscripts
        postrotate
                invoke-rc.d rsyslog rotate > /dev/null
        endscript
}
```

❶ /var/log/syslog の設定をこの {} 内に記述しています。

❷ 直前に記述された/var/log/mail.info から /var/log/messages までの全てのファイルの設定をこの {} 内に記述しています。

表 8.1 logrotate 設定ファイルの主な設定項目

項目	説明
missingok	対象のログファイルが存在しなくてもエラーを出さない。
notifempty	ログファイルが空ならローテーションしない。
rotate	ローテーションで保存するログファイル数。
size	指定したファイルサイズ以上になったらローテーションする。 追加指定された時間間隔より前にはローテーションしない。
maxsize	指定したファイルサイズ以上になったらローテーションする。 追加指定された時間間隔よりも前であってもローテーションする。
create	ローテーション後に空のログファイルを作成する。 パーミッション、ユーザ名、グループ名も設定できる。
daily	ログファイルを毎日ローテーションする。
weekly	ログファイルを毎週ローテーションする。
monthly	ログファイルを毎月ローテーションする。
yearly	ログファイルを毎年ローテーションする。
compress	ローテーションしたログファイルを gzip で圧縮する。
delaycompress	最新のローテーションは圧縮しない。
sharedscripts	対象のログファイルを複数指定している場合、 ローテーションするログファイルの数に関わらず、 postrotate または prerotate で指定したスクリプトを 1 回だけ呼び出す。
nosharedscripts	ローテーションするログファイル一つに対して 1 回ずつ postrotate または prerotate で指定したスクリプトを呼び出す。 スクリプトの第 1 引数には、ローテーションするログファイルのフルパスが与えられる。
postrotate/ endscript	ローテーションの後に postrotate から endscript までの間の行に記述されたスクリプトを呼び出す。
prerotate/ endscript	ローテーションの前に prerotate から endscript までの間の行に記述されたスクリプトを呼び出す。

logrotate ではこの他にも様々な設定が可能です。詳しくは、man logrotate の CONFIGURATION FILE を参照してください。

8.4.2. ログをリモートサーバーに送る

rsyslogd 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。

メッセージを転送するには、リモートサーバー(ここでは ATDE を使用します)とクライアント(ここでは Armadillo を使用します)の両方で適切な設定を行う必要があります。

大まかな手順は、次のようになります。

1. /etc/rsyslog.conf の修正 (リモートサーバーのみ)
2. /etc/rsyslog.d/aguide.conf の作成
3. rsyslog の restart
4. log の出力 (クライアントのみ)
5. log の確認 (リモートサーバーのみ)

ファイアーウォールの設定をしている場合は、上記の手順に、rsyslogd が使用するポートを解放する手順が加わります。

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を指定します。デフォルトでは、514 番ポートに転送します。

具体的な設定方法を、Armadillo から ATDE にログを転送する場合を例として説明します。ATDE 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、ATDE 側の設定をおこないます。/etc/rsyslog.conf の TCP の待ち受けに関する設定を有効にし、/etc/rsyslog.d/aguide.conf でログの出力先を設定します。/etc/ 以下のファイルの編集には、特権ユーザー権限が必要なことに注意してください。非特権ユーザーが特権ユーザー権限を行使するには、sudo コマンドを使います。

```
[ATDE ~]$ sudo vi /etc/rsyslog.conf
```

/etc/rsyslog.conf の次の部分を

```
: (省略)
# provides TCP syslog reception
# module(load="imtcp")
# input(type="imtcp" port="514")
: (省略)
```

以下のように修正します。

```
: (省略)
# provides TCP syslog reception
module(load="imtcp")
input(type="imtcp" port="514")
: (省略)
```

/etc/rsyslog.d/aguide.conf を作成し、Armadillo から受け取ったログの出力先を記述します。

```
[ATDE ~]$ sudo vi /etc/rsyslog.d/aguide.conf
```

内容は以下のようにします。

```
armadillo.*        /var/log/armadillo.log
```

設定を反映させるために、rsyslogd を再起動します。

```
[ATDE ~]$ sudo service rsyslog restart
```

次に、Armadillo 側の設定をおこないます。/etc/rsyslog.d/aguide.conf でログの出力先を設定します。

```
[armadillo ~]# vi /etc/rsyslog.d/aguide.conf
```

内容は以下のようにします。

```
armadillo.*        @@192.168.0.1
```

設定を反映させるために、rsyslogd を再起動します。

```
[armadillo ~]# service rsyslog restart
```

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、ATDE5 の rsyslogd によって/var/log/armadillo.log に記録されます。

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、ext4 ファイルシステムです。

ext4 ファイルシステムで、デバイスをフォーマットするには、mkfs.ext4 コマンドを使用します。

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. ファイアーウォールを設定する

Linux では iptables コマンドを使用して、パケットフィルタリング型ファイアーウォールを構築することができます。パケットフィルタリングとは、通信されるデータ（パケット）に対して、IP アドレスやポート番号などの情報によって、送られてきたパケットを許可、破棄、または拒否の判断を行う機能です。iptables コマンドはこのパケットフィルタリングのルールを設定することができます。

Armadillo-640 で iptables を使用するためには Linux カーネルコンフィギュレーションを変更してイメージファイルをビルドする必要があります。

```
Networking support --->
    Networking options --->
        Network packet filtering framework (Netfilter) --->
            Core Netfilter Configuration --->
                [*] Netfilter connection tracking support
Networking support --->
    Networking options --->
        Network packet filtering framework (Netfilter) --->
            Core Netfilter Configuration --->
```

```
                Netfilter Xtables support (required for ip_tables) --->
                    [*]  "conntrack" connection tracking match support
Networking support --->
    Networking options --->
        Network packet filtering framework (Netfilter) --->
            IP: Netfilter Configuration --->
                [*] IPv4 connection tracking support (required for NAT)
```

iptables の詳細な使用方法は man 8 iptables を参照してください。

8.6.1. すべてのパケットを破棄する

ここでは Armadillo に入ってくるすべてのパケットを破棄する設定を行います。まず、現在の iptables の設定を確認するために、以下のように実行します。

```
[armadillo ~]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

現在の設定は INPUT が ACCEPT になっているので、Armadillo に入ってくるすべてのパケットを通してしまいます。すべてのパケットを破棄するには INPUT と FORWARD を DROP に変更します。

```
[armadillo ~]# iptables -P INPUT DROP
[armadillo ~]# iptables -P FORWARD DROP
[armadillo ~]# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

これですべてのパケットを破棄するようになりましたが、このままでは Armadillo から自分自身へのパケットも破棄されてしまうので、ローカルループバックからのパケットは許可するように設定します。さらに、Armadillo から外部へのリクエストに対するレスポンスのパケットも許可するように設定します。

```
[armadillo ~]# iptables -A INPUT -i lo -j ACCEPT
[armadillo ~]# iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
[armadillo ~]# iptables -L
```

```
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTA
BLISHED

(省略)
[armadillo ~]# ping localhost ❶
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.163 ms
```

❶ 自身からのパケットは許可されていることを確認できます。

8.6.2. SSH を許可する

SSH による接続を許可したい場合は、以下のように設定します。

```
[armadillo ~]# iptables -A INPUT -p tcp -m tcp --dport 22 -j ACCEPT
[armadillo ~]# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTA
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:ssh

(省略)
```

これで外部から Armadillo への SSH 接続ができるようになります。

8.6.3. HTTPS を許可する

SSH と同様の手順で HTTPS による接続も許可することができます。

```
[armadillo ~]# iptables -A INPUT -p tcp -m tcp --dport 443 -j ACCEPT
[armadillo ~]# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             state RELATED,ESTA
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:ssh
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:https

(省略)
```

これで外部から Armadillo への HTTPS 接続ができるようになります。

8.6.4. 設定値を保存する

ここまで iptables を使用してパケットフィルタリングを設定してきましたが、このままでは Armadillo を再起動すると設定値が元に戻ってしまうので、次のように設定値を保存する必要があります。

```
[armadillo ~]# apt-get update
[armadillo ~]# apt-get install iptables-persistent ❶
[armadillo ~]# iptables-save > /etc/iptables/rules.v4
```

❶ すでにインストール済みの場合は不要です。

これで次回起動時から自動的に設定値が適用されるようになります。

8.7. SSH のセキュリティ設定を見直す

ネットワークに接続されている Armadillo に対して外部からログインするセキュアな方法として SSH が挙げられます。しかしデフォルトの設定ではパスワードでの認証となっており十分にセキュアとは言えません。この章では、パスワードの他にセキュリティを高める方法について説明します。

8.7.1. パスワードの代わりに RSA 公開鍵認証を使う

ここでは RSA 公開鍵認証を使用した SSH ログイン方法について説明します。

まず、クライアント側の環境で鍵ペアを作成します。ユーザ名は atmark とします。

```
[PC ~]$ mkdir .ssh ❶
[PC ~/.ssh]$ cd .ssh
[PC ~/.ssh]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/atmark/.ssh/id_rsa): ❷
Enter passphrase (empty for no passphrase): ❸
Enter same passphrase again:

[PC ~/.ssh]$ ls
id_rsa  id_rsa.pub
```

❶ すでにある場合は不要です。

❷ 鍵の保存先です。ここではデフォルト値とします。

❸ 秘密鍵を使用するためのパスフレーズです。設定したほうがよりセキュアです。

作成した公開鍵 id_rsa.pub を Armadillo へ転送します。ここでは scp を使った例を示します。

```
[PC ~/.ssh]$ scp id_rsa.pub atmark@[Armadillo の IP アドレス]:~/
```

クライアント側での作業はここまでとなります。

秘密鍵の扱い

秘密鍵がコピーされてしまうと、なりすましが可能になってしまうので、絶対に漏らさないように気をつけてください。

ここからは Armadillo 側での作業となります。

```
[armadillo ~]# su atmark ❶
[armadillo /root]$ cd ❷
[armadillo ~]$ mkdir .ssh ❸
[armadillo ~]$ cat ~/id_rsa.pub >> ~/.ssh/authorized_keys ❹
[armadillo ~]$ chmod 700 ~/.ssh ❺
[armadillo ~]$ chmod 600 ~/.ssh/authorized_keys ❻
[armadillo ~]$ exit ❼
```

❶ atmark ユーザになります。

❷ atmark のホームディレクトリへ移動します。

❸ すでにある場合は不要です。

❹ 公開鍵の内容を authorized_keys に追記します。

❺ パーミッションを適切に設定します。

❻

❼ root に戻ります。

次に SSH サーバの設定を変更します。

```
[armadillo ~]# vi /etc/ssh/sshd_config
```

RSA 公開鍵認証でログイン可能となるように設定します。加えてパスワードでのログインを不可とします。

```
:（省略）
RSAAuthentication yes ❶
PubkeyAuthentication yes ❷
# Expect .ssh/authorized_keys2 to be disregarded by default in future.
AuthorizedKeysFile      .ssh/authorized_keys .ssh/authorized_keys2 ❸
:（省略）
PasswordAuthentication no ❹
:（省略）
```

図 8.10 sshd_config の修正箇所

❶ 追加します。

❷ コメントアウトされているのでコメントアウトを外します。

❸

❹ コメントアウトを外しして no にします。

sshd を再起動します。

```
[armadillo ~]# systemctl restart sshd
```

これでクライアント側から RSA 公開鍵認証で Armadillo に SSH ログインできるようになりました。

```
[PC ~]$ ssh -i ~/.ssh/id_rsa [Armadillo の IP アドレス]
Enter passphrase for key '/home/atmark/.ssh/id_rsa': ❶

[armadillo ~]$
```

❶ 秘密鍵にパスフレーズを設定した場合はここで入力します。

8.7.2. 使用するポートを変更する

ここでは通常 22 である SSH のポート番号を変更する手順を説明します。SSH サーバの設定を変更します。

```
[armadillo ~]# vi /etc/ssh/sshd_config
```

変更箇所は以下のとおりです。例として 8022 に変更しています。

```
:（省略）
Port 8022 ❶
:（省略）
```

図 8.11 sshd_config の修正箇所（ポート番号）

❶ コメントアウトを外し任意のポート番号に変更します。

sshd を再起動します。

```
[armadillo ~]# systemctl restart sshd
```

これでポート番号が変更されました。クライアント側からは以下のようにポート番号を指定して接続します。

```
[PC ~]$ ssh -i ~/.ssh/id_rsa -p 8022 [Armadillo の IP アドレス]
Enter passphrase for key '/home/atmark/.ssh/id_rsa': ❶

[armadillo ~]$
```

❶ 秘密鍵にパスフレーズを設定した場合はここで入力します。

ポート番号

ポート番号はどのプログラムからも使われていない番号を指定してください。

8.8. ソフトウェアアップデート

Armadillo の製品マニュアルの「特定のイメージファイルだけを書き換える」に記載されいている方法は、Armadillo に PC などの端末を接続しての操作を必要としています。製品出荷後でもソフトウェアを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仕組みが望ましいところです。

また、ソースコードの管理も重要です。これは前述したアップデートの仕組みの前段階として、「ブートローダーやカーネル等が更新されて新しいソースコードが配布された場合、古くなったソースコードに行っていたカスタマイズを、どうやって新しいソースコードに適用するのか」という話になります。

本章では、ソースコード管理のためのツールと、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。

8.8.1. Git によるソースコードの管理

ソースコードの管理には所謂バージョン管理システムを利用します。バージョン管理システムには様々ありますが、ここでは Git を使用していきます。

Git を利用すると、古くなったソースコードに行っていたカスタマイズを、新しいソースコードに適用することが簡単にできます。また、バージョン管理システムの基本的な機能として、ソースコードの変更内容を確認したり、変更前の状態に戻したりすることもできます。

ここからは、Armadillo-640 のブートローダーのソースコードを例に、「古くなったソースコードに行っていたカスタマイズを、新しいソースコードに適用する」ための具体的な手順を説明していきます。

流れとしては次のとおりです。

1. Git リポジトリの作成
2. ベースとなるソースコード (u-boot-a600-v2018.03-at7) を登録
3. ソースコードの修正
4. ベースとなるソースコードの更新 (u-boot-a600-v2018.03-at7 から u-boot-a600-v2018.03-at8 へ)
5. 更新後のベースとなるソースコードにソースコードの修正を適用

8.8.1.1. Git リポジトリの作成

まずは、https://download.atmark-techno.com/armadillo-640/source/u-boot-a600-v2018.03-at7.tar.gz を ATDE にダウンロードして、tar コマンドで展開します。

```
[ATDE ~]$ wget https://download.atmark-techno.com/armadillo-640/source/u-boot-a600-v2018.03-
at7.tar.gz
[ATDE ~]$ tar xf u-boot-a600-v2018.03-at7.tar.gz
[ATDE ~]$ ls --all -1 u-boot-a600-v2018.03-at7
.
..
.checkpatch.conf
.gitignore
.mailmap
.travis.yml
Documentation
```

```
Kbuild
Kconfig
Licenses
MAINTAINERS
Makefile
README
api
arch
board
cmd
common
config.mk
configs
disk
doc
drivers
dts
env
examples
fs
include
lib
net
post
scripts
snapshot.commit
test
tools
```

後で新しいソースコードに更新する作業を体験するため、ここでは、あえて最新ではないソースコードをダウンロードしています。

実際の開発では最新のソースコードをダウンロードしてください。

展開が終わったら、mv コマンドでディレクトリ名を変更します。ソースコードのバージョン管理は Git で行えるのでバージョンを除いたディレクトリ名とします。

```
[ATDE ~]$ mv u-boot-a600-v2018.03-at7 u-boot-a600-v2018.03-at
```

ディレクトリ名を変更したら、cd コマンドでディレクトリを移動し、空の Git リポジトリを作成します。

```
[ATDE ~]$ cd u-boot-a600-v2018.03-at
[ATDE ~/u-boot-a600-v2018.03-at]$ git init
Initialized empty Git repository in /home/atmark/u-boot-a600-v2018.03-at/.git/
```

8.8.1.2. ベースとなるソースコードの登録

ベースとなるソースコードを Git リポジトリに登録します。git add で対象となるファイルを指定し、git commit で対象のファイルをひとまとまりの修正として登録します。登録の際は、対象のファイルでどのような修正が行われたか、を説明するためのコミットメッセージを記述します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git add -f $(find . -mindepth 1 -maxdepth 1)
[ATDE ~/u-boot-a600-v2018.03-at]$ git commit -m"Imported from u-boot-a600-v2018.03-at7.tar.gz"
```

ここで登録した「ひとまとまりの修正」を Git ではコミットと呼びます。

Git はコミット単位でバージョンを管理します。

git commit の -m オプションは、コマンドライン引数でコミットメッセージを記述するためのものです。

-m オプションを指定しなかった場合は自動でエディタが起動するので、エディタでコミットメッセージを記述します。

$()は、()内の find . -mindepth 1 -maxdepth 1 を実行し出力された文字列をコマンドライン上に展開します。

git add $(find . -mindepth 1 -maxdepth 1)はカレントディレクトリのファイル全てを git add するという意味になります。つまり、以下のコマンドを実行するのと同じです。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git add ./.git ./tools ./test ./
snapshot.commit ./scripts ./post ./net ./lib ./include ./fs ./examples ./
env ./dts ./drivers ./doc ./disk ./configs ./config.mk ./common ./cmd ./
board ./arch ./api ./README ./Makefile ./MAINTAINERS ./Licenses ./
Kconfig ./Kbuild ./
Documentation ./.travis.yml ./.mailmap ./.gitignore ./.checkpatch.conf
```

8.8.1.3. ソースコードの修正

Git でのソースコードの修正は、以下の流れで行います。

1. 新しい branch(ブランチ)の作成
2. ファイルの編集
3. 編集したファイルの git add
4. git commit

先程 git commit したベースとなるソースコードは、デフォルトのブランチである master に積まれています。ここから新しいブランチを作成して、そのブランチ上でソースコードのカスタマイズを行っていきます。master とブランチを分けることで、「ベースとなるソースコードの更新」と「ソースコードのカスタマイズ」を区別することができます。

git log を実行すると現在のブランチに積まれているコミットを確認できます。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git log
commit 8c22d30fcb0c254d3f0cf28cb9da4611a7100d70
Author: atmark <atmark@atde7>
Date:   Wed May 13 14:39:35 2020 +0900

    Imported from u-boot-a600-v2018.03-at7.tar.gz
```

ここから新しいブランチを作成して、そのブランチをチェックアウトする(現在のブランチからそのブランチに切り替える)には git checkout -b を実行します。ブランチの名称は製品プロジェクトの名称等で構いません。ここでは x11a としています。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git checkout -b x11a
Switched to a new branch 'x11a'
```

現在のブランチを確認するには、git branch を実行します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git branch
  master
* x11a
```

新しいブランチをチェックアウトしたら、ソースコードのカスタマイズをしていきます。ここでは、U-Boot のデフォルトの環境変数に aguide=Software Development を追加していきます。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ vi include/configs/armadillo-640.h
```

```
#define CONFIG_EXTRA_ENV_SETTINGS ¥
        "setup_mmcargs=setenv bootargs root=/dev/mmcblk0p2 rootwait ${optargs};¥0"¥
        BOOTCOMMAND_USB¥
       "tftpboot=tftpboot uImage; tftpboot 0x83000000 a640.dtb; bootm ${loadaddr} - 0x83000000;¥0"¥
        STOP_NR3225SA_ALARM_ENV_NAME "=" STOP_NR3225SA_ALARM_DEFAULT ";¥0"¥
        ENABLE_PF3000_LPM_ENV_NAME "=" ENABLE_PF3000_LPM_DEFAULT "¥0"

#define CONFIG_BOARD_LATE_INIT


#endif
```

図 8.12 編集前の include/configs/armadillo-640.h(末尾のみ抜粋)

を次のように修正します。

```
#define CONFIG_EXTRA_ENV_SETTINGS ¥
        "setup_mmcargs=setenv bootargs root=/dev/mmcblk0p2 rootwait ${optargs};¥0"¥
        BOOTCOMMAND_USB¥
       "tftpboot=tftpboot uImage; tftpboot 0x83000000 a640.dtb; bootm ${loadaddr} - 0x83000000;¥0"¥
        STOP_NR3225SA_ALARM_ENV_NAME "=" STOP_NR3225SA_ALARM_DEFAULT ";¥0"¥
        ENABLE_PF3000_LPM_ENV_NAME "=" ENABLE_PF3000_LPM_DEFAULT "¥0"¥
        "aguide=Software Development¥0"
```

```
#define CONFIG_BOARD_LATE_INIT


#endif
```

図 8.13 編集後の include/configs/armadillo-640.h(末尾のみ抜粋)

編集が終わったら git diff を実行します。git diff により、現状のソースコードにどのような変更が行われたのか確認することができます。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git diff
diff --git a/include/configs/armadillo-640.h b/include/configs/armadillo-640.h
index 7235241..78ad680 100644
--- a/include/configs/armadillo-640.h
+++ b/include/configs/armadillo-640.h
@@ -121,7 +121,8 @@ int wlan_rtc_i2c_read(void);
        BOOTCOMMAND_USB¥
       "tftpboot=tftpboot uImage; tftpboot 0x83000000 a640.dtb; bootm ${loadaddr} - 0x83000000;¥0"¥
        STOP_NR3225SA_ALARM_ENV_NAME "=" STOP_NR3225SA_ALARM_DEFAULT ";¥0"¥
-       ENABLE_PF3000_LPM_ENV_NAME "=" ENABLE_PF3000_LPM_DEFAULT "¥0"
+       ENABLE_PF3000_LPM_ENV_NAME "=" ENABLE_PF3000_LPM_DEFAULT "¥0"¥
+       "aguide=Software Development¥0"

 #define CONFIG_BOARD_LATE_INIT
```

正しく修正できていたら、git add と git commit でコミットを作成します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git add include/configs/armadillo-640.h
[ATDE ~/u-boot-a600-v2018.03-at]$ git commit -m"デフォルトの環境変数に aguide を追加"
[ATDE ~/u-boot-a600-v2018.03-at]$ git log
commit 280d3336a73f0f80269dc166e6653dfcec1a15ca
Author: atmark <atmark@atde7>
Date:   Wed May 13 15:13:38 2020 +0900

    デフォルトの環境変数に aguide を追加

commit 8c22d30fcb0c254d3f0cf28cb9da4611a7100d70
Author: atmark <atmark@atde7>
Date:   Wed May 13 14:39:35 2020 +0900

    Imported from u-boot-a600-v2018.03-at7.tar.gz
```

8.8.1.4. ベースとなるソースコードの更新

ベースとなるソースコードの更新は、master ブランチに対して行っていきます。

流れとしては次のとおりです。

1. master ブランチのチェックアウト
2. 古くなったソースコードの削除
3. 新しいソースコードのダウンロード

4. 新しいソースコードの展開

5. 新しいソースコードの git add, git commit

master ブランチをチェックアウトし、古くなったソースコードの削除していきますが、その前に git commit していない変更がないかを git status で確認してください。当然ながら、ファイルを削除すると、git commit していない変更は失われ、復元できなくなります。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git status
On branch x11a
nothing to commit, working tree clean
```

ソースコードの削除には git rm を使用します。git rm は Git リポジトリに登録されているファイルを削除するコマンドです。

それでは、master ブランチをチェックアウトし、古くなったソースコードの削除していきます。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git checkout master
Switched to branch 'master'
[ATDE ~/u-boot-a600-v2018.03-at]$ git rm -r --ignore-unmatch $(find . -mindepth 1 -maxdepth 1)
```

git rm -r --ignore-unmatch は、指定された Git リポジトリに登録されているファイルを全て削除する、という意味になります。

-r オプションでディレクトリ内のファイルを再帰的に削除し、--ignore-unmatch オプションで Git リポジトリに登録されていないファイルが指定されていても、コマンドがエラーしないようにしています。

Git リポジトリ登録されていないファイルがなければ、以下のように".git"だけがある状態になります。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ ls --all -1
.
..
.git
```

次に新しいソースコード(u-boot-a600-v2018.03-at8.tar.gz)をダウンロードし、Git リポジトリに展開し、不要なファイルは削除します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ wget https://download.atmark-techno.com/armadillo-640/source/u-
boot-a600-v2018.03-at8.tar.gz
[ATDE ~/u-boot-a600-v2018.03-at]$ tar xf u-boot-a600-v2018.03-at8.tar.gz
[ATDE ~/u-boot-a600-v2018.03-at]$ ls --all -1 u-boot-a600-v2018.03-at8
.
..
.checkpatch.conf
.gitignore
.mailmap
.travis.yml
Documentation
```

```
Kbuild
Kconfig
Licenses
MAINTAINERS
Makefile
README
api
arch
board
cmd
common
config.mk
configs
disk
doc
drivers
dts
env
examples
fs
include
lib
net
post
scripts
snapshot.commit
test
tools
[ATDE ~/u-boot-a600-v2018.03-at]$ mv $(find u-boot-a600-v2018.03-at8 -mindepth 1 -maxdepth 1) ./
[ATDE ~/u-boot-a600-v2018.03-at]$ rmdir u-boot-a600-v2018.03-at8/
[ATDE ~/u-boot-a600-v2018.03-at]$ rm u-boot-a600-v2018.03-at8.tar.gz
```

新しいソースコードを Git リポジトリに登録します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git add -f $(find . -mindepth 1 -maxdepth 1)
[ATDE ~/u-boot-a600-v2018.03-at]$ git commit -m"Update to u-boot-a600-v2018.03-at8"
[ATDE ~/u-boot-a600-v2018.03-at]$ git log
commit 761d05239a5deeedbba1a915cab099cf31a06b53
Author: atmark <atmark@atde7>
Date:   Wed May 13 16:23:03 2020 +0900

    Update to u-boot-a600-v2018.03-at8

commit 8c22d30fcb0c254d3f0cf28cb9da4611a7100d70
Author: atmark <atmark@atde7>
Date:   Wed May 13 14:39:35 2020 +0900

    Imported from u-boot-a600-v2018.03-at7.tar.gz
```

これで、ベースとなるソースコードの更新は完了です。

8.8.1.5. 更新後のベースとなるソースコードにソースコードの修正を適用

更新後のベースとなるソースコードに「8.8.1.3. ソースコードの修正」を適用していきます。この作業は、コンフリクトが起きなければ、非常に簡単です。

Git が自動でブランチを統合できないことをコンフリクト(conflict)といいます。

流れとしては次のとおりです。

1. 適用するソースコードの修正を行ったブランチをチェックアウト
2. git rebase で更新後のベースとなるソースコードのブランチにリベース

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git checkout x11a
Switched to branch 'x11a'
[ATDE ~/u-boot-a600-v2018.03-at]$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: デフォルトの環境変数に aguide を追加
Using index info to reconstruct a base tree...
M       include/configs/armadillo-640.h
Falling back to patching base and 3-way merge...
Auto-merging include/configs/armadillo-640.h
Auto packing the repository in background for optimum performance.
See "git help gc" for manual housekeeping.
[ATDE ~/u-boot-a600-v2018.03-at]$ git log
commit 40a9afcdcfc585feb23575338b93ee279a2c2a44
Author: atmark <atmark@atde7>
Date:   Wed May 13 15:13:38 2020 +0900

    デフォルトの環境変数に aguide を追加

commit 761d05239a5deeedbba1a915cab099cf31a06b53
Author: atmark <atmark@atde7>
Date:   Wed May 13 16:23:03 2020 +0900

    Update to u-boot-a600-v2018.03-at8

commit 8c22d30fcb0c254d3f0cf28cb9da4611a7100d70
Author: atmark <atmark@atde7>
Date:   Wed May 13 14:39:35 2020 +0900

    Imported from u-boot-a600-v2018.03-at7.tar.gz
```

以上で完了となります。

コンフリクトが起きた際は、コマンド(git rebase)が出力したメッセージに従って修正作業をしていくことになります。どのような修正を行うかは、その時々で異なるので、具体的な説明はできません。修正前のソースコードはどんなものだったのか、どんな修正なのか、といった視点からその修正が必要なのかを判断し、修正していく必要があります。

git rebase でコンフリクトが起き、とりあえずブランチをもとの状態(git rebase 前の状態)に戻したい場合は、git rebase --abort を実行します。

```
[ATDE ~/u-boot-a600-v2018.03-at]$ git rebase --abort
```

8.8.2. Armadillo のソフトウェア更新における 2 つの考え方

Armadillo のソフトウェア更新には、2 つの考え方があります。一つは「イメージファイルのアップデート」、もう一つは「ソフトウェア単体のアップデート」です。

「イメージファイルのアップデート」は、ソフトウェアをブートローダー、カーネル、ユーザーランドに大別してそれぞれをアップデートする、という考え方です。これは、Armadillo の製品マニュアルに記載されている「イメージファイルの書き換え方法」と同じ考え方です。

「ソフトウェア単体のアップデート」は、ブートローダー、カーネル、そしてユーザーランドに含まれるソフトウェアを一つ一つアップデートする、という考え方です。この考え方は、ブートローダーとカーネルについては「イメージファイルのアップデート」と同じですが、ユーザーランドについては大きく異なります。アップデートの仕組みは複雑になってしまいますが、アップデートが必要なソフトウェアのみアップデートできるため、データ通信量やアップデートにかかる時間を最小限に抑えることができます。

8.8.3. インストールディスクを使用してのアップデート

インストールディスクは、イメージファイルの書き換え方法として、各製品の製品マニュアルで紹介されています。「インストールディスクを使用してのアップデート」とは、インストールディスクを自作し、自作したインストールディスクを使用して Armadillo のソフトウェアを更新する、ということです。

大まかな手順は次のとおりです。

1. ブートローダーイメージのビルド
2. Linux カーネルイメージおよび DTB のビルド
3. ユーザーランドアーカイブのビルド
4. インストールディスクイメージの作成
5. インストールディスクの作成
6. インストールの実行

「ブートローダーイメージの作成」「Linux カーネルイメージおよび DTB の作成」「ユーザーランドアーカイブの作成」「インストールディスクの作成」「インストールの実行」は各製品の製品マニュアルで、「インストールディスクイメージの作成」は「Armadillo 標準ガイド Armadillo 入門編」で説明されていますので、そちらを参照してください。

8.8.4. インターネットを使用してのアップデート

インターネットから最新のソフトウェアをダウンロードしてアップデートする方法を紹介します。

8.8.4.1. APT でインストールしたソフトウェアのアップデート

実は apt コマンドでインストールしたソフトウェア(debian package)については既に、定期的に最新のソフトウェアをインターネットからダウンロードしてインストールする機能が、有効になっています。この機能は、 APT(debian package) に含まれている複数の Unit ファイルによって実現されています。

この APT の自動アップデート機能が有効になっているかどうかは、次のようにして確認できます。apt-daily-upgrade.timer と apt-daily.timer が enabled なら有効になっています。

```
[armadillo ~]# systemctl list-unit-files | grep apt
apt-daily-upgrade.service              static
```

```
apt-daily.service                   static
apt-daily-upgrade.timer             enabled
apt-daily.timer                     enabled
```

apt の自動アップデート機能を利用するだけであれば、以上の情報があれば十分ですが、ここからは、APT の Unit ファイルについて簡単に解説しておきます。インターネットを使用してのアップデートの実装の一つとして、参考になるはずです。

```
[Unit]
Description=Daily apt download activities

[Timer]
OnCalendar=*-*-* 6,18:00 ❶
RandomizedDelaySec=12h ❷
Persistent=true

[Install]
WantedBy=timers.target
```

図 8.14 /lib/systemd/system/apt-daily.timer

❶ 6 時および 18 時にタイマーが起動するよう設定しています。

❷ タイマーの起動を設定時刻から最大 12 時間後まで、 ランダムに遅らせるよう設定しています。

RandomizedDelaySec の項目は非常に重要です。Armadillo を量産した際、全ての Armadillo に「図 8.14. /lib/systemd/system/apt-daily.timer」が書き込まれて動作します。この場合、RandomizedDelaySec の設定をしていなければ、全ての Armadillo が、6 時または 18 時ちょうど(つまりほぼ同時)に、サーバーにアクセスするため、サーバーに多大な負荷がかかってしまいます。

```
[Unit]
Description=Daily apt upgrade and clean activities
After=apt-daily.timer ❶

[Timer]
OnCalendar=*-*-* 6:00 ❷
RandomizedDelaySec=60m ❸
Persistent=true

[Install]
WantedBy=timers.target
```

図 8.15 /lib/systemd/system/apt-daily-upgrade.timer

❶ apt upgrade は、 apt update を実行した後でなければ実行する意味がないため、 apt-daily.timer の後に動作するよう設定しています。

❷ 6 時にタイマーが起動するよう設定しています。

❸ タイマーの起動を設定時刻から最大 60 分後まで、 ランダムに遅らせるよう設定しています。

```
[Unit]
Description=Daily apt download activities
```

```
Documentation=man:apt(8)
ConditionACPower=true
After=network-online.target ❶
Wants=network-online.target ❷

[Service]
Type=oneshot
ExecStart=/usr/lib/apt/apt.systemd.daily update
```

図 8.16 /lib/systemd/system/apt-daily.service

❶ インターネット接続後でなければアップデートできないため、 network-online.target の後に起動するよう設定しています。

❷ インターネット接続後でなければアップデートできないため、 network-online.target が起動される場合のみ起動するよう設定しています。

```
[Unit]
Description=Daily apt upgrade and clean activities
Documentation=man:apt(8)
ConditionACPower=true
After=apt-daily.service ❶

[Service]
Type=oneshot
ExecStart=/usr/lib/apt/apt.systemd.daily install
KillMode=process
TimeoutStopSec=900 ❷
```

図 8.17 /lib/systemd/system/apt-daily-upgrade.service

❶ apt upgrade は、 apt update を実行した後でなければ実行する意味がないため、 apt-daily.service の後に動作するよう設定しています。

❷ 停止の要求を出してから、停止まで 900 秒間は待つよう設定しています。

8.8.4.2. APT でインストールしたソフトウェア以外のソフトウェアのアップデート

APT でインストールしたソフトウェア以外のソフトウェアと言うと、次のようなソフトウェアが該当します。

- ブートローダーイメージ
- Linux カーネルイメージ
- DTB(Device Tree Blob)
- APT 以外のパッケージ管理ソフトウェア[3]でインストールしたソフトウェア
- C や Python などで自作したアプリケーション
- アプリケーションの起動を管理するための自作 Unit ファイル

[3] pip(Python) や gem(Ruby) が該当します。

「APT 以外のパッケージ管理ソフトウェアでインストールしたソフトウェア」については、パッケージ管理ソフトウェアでのアップデートを定期的に実行すれば良いでしょう。それ以外のソフトウェアについては、大量にあると制御が大変なので、tar コマンドで 1 つに結合したファイルを Web サーバーに配置しておくことにします。

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. pip でインストールしたパッケージのアップデート確認
    b. 必要であれば、pip でインストールしたパッケージのアップデート
    c. Web サーバーにアクセスし、更新すべきソフトウェアがあるか確認する。
    d. ファイルがある場合、ダウンロードしてストレージに書き込みアップデート。
    e. 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には、systemd を利用することにします。systemd については、「8.2.1. systemd 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設定は以下のようになるでしょう。ただし、製品を量産した時のことを考慮して RandomizedDelaySec を設定し、4:00 ちょうどには実行しないようにします。イメージのアップデートを行うスクリプトは、/root/web_software_update.sh という名前だとします。

```
[Unit]
Description = web image update
After=network-online.target
Wants=network-online.target

[Service]
ExecStart = /root/web_software_update.sh
Type = oneshot
```

図 8.18 毎日 4:00 に処理を実行する Unit ファイルの実装内容(web_image_update.service)

```
[Unit]
Description = launch web image updater

[Timer]
Persistent = true
OnCalendar = *-*-* 04:00:00
RandomizedDelaySec=1h

[Install]
WantedBy=timers.target
```

図 8.19 毎日 4:00 に処理を実行する Unit ファイルの実装内容(web_image_update.timer)

systemd から実行される web_software_update.sh は、次のようになります。pip-review により pip でインストールした Python の各種パッケージをアップデート、その後、ブートローダー、Linux カーネ

ル、DTB、アプリケーションについて、最新バージョンのチェックを行い、最新バージョンがインストールされていなかった場合は、ソフトウェアの更新を行ってから Armadillo を再起動させます。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)等の各変数は、環境に合わせて書き換えてください。

wget コマンドは、https プロトコルを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='https'
USER_NAME=''
PASSWORD=''

PRODUCT_PATH='products/x11a'
URL_PATH="${PROTCOL}://${SERVER_IP_ADDRESS}/${PRODUCT_PATH}"

BOOTLOADER_IMG_PREFIX='u-boot-a600-v'
BOOTLOADER_IMG_SUFFIX='.imx'
KERNEL_IMG_PREFIX='uImage-a600-v'
KERNEL_IMG_SUFFIX=''
DTB_IMG_PREFIX='armadilo-640-v'
DTB_IMG_SUFFIX='.dtb'
APP_ARCHIVE_PREFIX='app-v'
APP_ARCHIVE_SUFFIX='.tar.xz'

VERSIONSDIR='/var/x11a/versions'
LOCKDIR='/var/lock/auto_image_update'
TMPDIR=''
FILE_IS_UPDATED=false

WGET_OPS=''
if [ "$USER_NAME" ]; then
    WGET_OPS="--http-user=$USER_NAME"
    if [ "$PASSWORD" ]; then
        WGET_OPS="$WGET_OPS --http-password=$PASSWORD"
    fi
fi

log()
{
    logger -p user.$1 -t "$0[$$]" -- "$2"
}

ledctrl()
{
    local LEDCLASS="/sys/class/leds"
    local led="$1"
    shift
    local op="$1"
    shift

    case "${op}" in
        "blink_on")
            local interval="$1"
```

```
            echo "timer" > "${LEDCLASS}/${led}/trigger"
            echo $interval > "${LEDCLASS}/${led}/delay_on"
            echo $interval > "${LEDCLASS}/${led}/delay_off"
            ;;
        "blink_off")
            echo "none" > "${LEDCLASS}/${led}/trigger"
            echo 0 > "${LEDCLASS}/${led}/brightness"
            ;;
    esac
}

die()
{
    log err "$1"

    if [ "$TMPDIR" ]; then
        cd
        umount $TMPDIR
        rm -rf $TMPDIR
    fi

    ledctrl red blink_off
    rmdir $LOCKDIR

    exit 1
}

bootloader_update_x1_iotg3()
{
    x1-bootloader-install "$1"
}

kernel_update_x1_iotg3()
{
    mount -t vfat /dev/mmcblk2p1 /mnt && ¥
    cp "$1" /mnt/uImage && ¥
    umount /mnt
}

dtb_update_x1_iotg3()
{
    mount -t vfat /dev/mmcblk2p1 /mnt && ¥
    cp "$1" "$DTB_PATH" && ¥
    umount /mnt
}

bootloader_update_a600series()
{
    dd if="$1" of=/dev/mmcblk0 ¥
       bs=1k seek=1 conv=fsync
}

kernel_update_a600series()
{
    mv "$1" /boot/uImage
}
```

```
dtb_update_a600series()
{
    mv "$1" /boot/a640.dtb
}

python_update()
{
    # pip3 install pip-review
    pip-review --auto
}

get_latest_filename()
{
    local prefix="$1"
    local suffix="$2"
    local url="$3"
    # for example, if
    #   prefix="uImage-a600-v"
    #   suffix=""
    #   url="https://download.atmark-techno.com/armadillo-640/image/"
    # then get_latest_filename puts
    #   "uImage-a600-v4.14-at21"
    # to stdout as of 2020-04-21.

    wget $WGET_OPS -q -O - "$url" | html2 | grep @href | ¥
        if [ -n "${prefix}" ]; then
            grep "${prefix}"
        else
            sed '' # pass through
        fi | ¥
        if [ -n "${suffix}" ]; then
            grep "${suffix}"
        else
            sed '' # pass through
        fi | grep -v ".md5" | ¥
        sort -V | tail -n 1 | sed -r 's/^.*@href=(.*)$/¥1/'
}

check_and_set_version()
{
    local version="$1"

    mkdir -p "${VERSIONSDIR}"

    if find "${VERSIONSDIR}" -mindepth 1 | ¥
       grep -q "^${VERSIONSDIR}/${version}$"; then
        if tail -n 1 "${VERSIONSDIR}/${version}" | ¥
           grep -q "^install OK"; then
            return 0 # already installed
        else
            return 1 # previous installation failed
        fi
    fi

    touch "${VERSIONSDIR}/${version}"
    return 2 # then installation start
}
```

```
complete_set_version()
{
    echo "install OK" >> "${VERSIONSDIR}/$1"
}

incomplete_set_version()
{
    local version=$1
    shift
    local msg="$@"

    echo "install NG${msg}" >> "${VERSIONSDIR}/$1"
}

download_and_md5sum()
{
    local url_path="$1"
    local filename="$2"

    wget $WGET_OPS -q "${url_path}/${filename}" -O "${filename}" || ¥
        log err "download ${filename} failed"

    wget $WGET_OPS -q "${url_path}/${filename}.md5" -O "${filename}.md5"
    if [ $? -ne 0 ]; then
        md5sum "${filename}" > "${filename}.md5"
        rm -f "${filename}"

        wget $WGET_OPS -q "${url_path}/${filename}" -O "${filename}"
    fi

    md5sum -c "${filename}.md5"
}

update()
{
    local fail
    local msg
    local update

    python_update

    local LATEST_BOOTLOADER_NAME="$(get_latest_filename "${BOOTLOADER_IMG_PREFIX}" "$ ⏎
{BOOTLOADER_IMG_SUFFIX}" "${URL_PATH}")"
    local LATEST_KERNEL_NAME="$(get_latest_filename "${KERNEL_IMG_PREFIX}" "${KERNEL_IMG_SUFFIX}" ⏎
"${URL_PATH}")"
    local LATEST_DTB_NAME="$(get_latest_filename "${DTB_IMG_PREFIX}" "${DTB_IMG_SUFFIX}" "$ ⏎
{URL_PATH}")"
    local LATEST_APP_ARCHIVE_NAME="$(get_latest_filename "${APP_ARCHIVE_PREFIX}" "$ ⏎
{APP_ARCHIVE_SUFFIX}" "${URL_PATH}")"

    msg=""
    fail=1
    if [ -n "${LATEST_BOOTLOADER_NAME}" ]; then
        if check_and_set_version "${LATEST_BOOTLOADER_NAME}"; then
            : "nothing to do"
        else
            log info "update bootloader to ${LATEST_BOOTLOADER_NAME}"
```

```
        download_and_md5sum "${URL_PATH}" ¥
            "${LATEST_BOOTLOADER_NAME}" || ¥
            download_and_md5sum "${URL_PATH}" ¥
                "${LATEST_BOOTLOADER_NAME}" # retry
        if [ $? -ne 0 ]; then
            log err "download ${URL_PATH}/${LATEST_BOOTLOADER_NAME} failed"
            msg="${msg}: donwload failed"
        else
            $bootloader_update "${LATEST_BOOTLOADER_NAME}"
            if [ $? -eq 0 ]; then
                fail=0
            else
                log err "update ${LATEST_BOOTLOADER_NAME} failed"
                msg="${msg}: copy failed"
            fi
        fi

        if [ $fail -eq 0 ]; then
            complete_set_version "${LATEST_BOOTLOADER_NAME}"
            FILE_IS_UPDATED=true
        else
            incomplete_set_version "${LATEST_BOOTLOADER_NAME}" "$msg"
        fi
    fi
fi

msg=""
fail=1
if [ -n "${LATEST_KERNEL_NAME}" ]; then
    if check_and_set_version "${LATEST_KERNEL_NAME}"; then
        : "nothing to do"
    else
        log info "update kernel to ${LATEST_KERNEL_NAME}"

        download_and_md5sum "${URL_PATH}" ¥
            "${LATEST_KERNEL_NAME}" || ¥
            download_and_md5sum "${URL_PATH}" ¥
                "${LATEST_KERNEL_NAME}" # retry
        if [ $? -ne 0 ]; then
            log err "download ${URL_PATH}/${LATEST_KERNEL_NAME} failed"
            msg="${msg}: donwload failed"
        else
            $kernel_update "${LATEST_KERNEL_NAME}"
            if [ $? -eq 0 ]; then
                fail=0
            else
                log err "update ${LATEST_KERNEL_NAME} failed"
                msg="${msg}: copy failed"
            fi
        fi

        if [ $fail -eq 0 ]; then
            complete_set_version "${LATEST_KERNEL_NAME}"
            FILE_IS_UPDATED=true
        else
            incomplete_set_version "${LATEST_KERNEL_NAME}" "$msg"
        fi
    fi
```

```
    fi

    msg=""
    fail=1
    if [ -n "${LATEST_DTB_NAME}" ]; then
        if check_and_set_version "${LATEST_DTB_NAME}"; then
            : "nothing to do"
        else
            log info "update dtb to ${LATEST_DTB_NAME}"

            download_and_md5sum "${URL_PATH}" ¥
                "${LATEST_DTB_NAME}" || ¥
                download_and_md5sum "${URL_PATH}" ¥
                    "${LATEST_DTB_NAME}" # retry
            if [ $? -ne 0 ]; then
                log err "download ${URL_PATH}/${LATEST_DTB_NAME} failed"
                msg="${msg}: donwload failed"
            else
                $dtb_update "${LATEST_DTB_NAME}"
                if [ $? -eq 0 ]; then
                    fail=0
                else
                    log err "update ${LATEST_DTB_NAME} failed"
                    msg="${msg}: copy failed"
                fi
            fi

            if [ $fail -eq 0 ]; then
                complete_set_version "${LATEST_DTB_NAME}"
                FILE_IS_UPDATED=true
            else
                incomplete_set_version "${LATEST_DTB_NAME}" "$msg"
            fi
        fi
    fi

    msg=""
    fail=1
    if [ -n "${LATEST_APP_ARCHIVE_NAME}" ]; then
        if check_and_set_version "${LATEST_APP_ARCHIVE_NAME}"; then
            : "nothing to do"
        else
            log info "update applications by extracting ${LATEST_APP_ARCHIVE_NAME}"

            download_and_md5sum "${URL_PATH}" ¥
                "${LATEST_APP_ARCHIVE_NAME}" || ¥
                download_and_md5sum "${URL_PATH}" ¥
                    "${LATEST_APP_ARCHIVE_NAME}" # retry
            if [ $? -ne 0 ]; then
                log err "download ${URL_PATH}/${LATEST_APP_ARCHIVE_NAME} failed"
                msg="${msg}: donwload failed"
            else
                tar xf "${LATEST_APP_ARCHIVE_NAME}" -C /
                if [ $? -eq 0 ]; then
                    fail=0
                else
                    log err "update ${LATEST_APP_ARCHIVE_NAME} failed"
                    msg="${msg}: extract archive failed"
```

```
                fi
            fi

            if [ $fail -eq 0 ]; then
                complete_set_version "${LATEST_APP_ARCHIVE_NAME}"
                FILE_IS_UPDATED=true
            else
                incomplete_set_version "${LATEST_APP_ARCHIVE_NAME}" "$msg"
            fi
        fi
    fi
}

model="$(cat /proc/device-tree/model)" > /dev/null
case "$model" in
    *"Atmark Techno Armadillo-6"*)
        bootloader_update=bootloader_update_a600series
        kernel_update=kernel_update_a600series
        dtb_update=dtb_update_a600series
        case "$model" in
            *"Armadillo-64"*)
                DTB_PATH="/boot/a640.dtb"
                ;;
            *"Armadillo-61"*)
                DTB_PATH="/boot/a610.dtb"
                ;;
            *)
                die "unknown model($model)"
                ;;
        esac
        ;;
    *"Atmark-Techno Armadillo-X1"*"miniaml"*)
        die "detect minimal model($model)"
        ;;
    *"Atmark-Techno Armadillo-X1"*|*"Atmark-Techno Armadillo-IoT"*"G3"*)
        bootloader_update=bootloader_update_x1_iotg3
        kernel_update=kernel_update_x1_iotg3
        dtb_update=dtb_update_x1_iotg3
        case "$model" in
            *"Armadillo-X1 Board"*)
                DTB_PATH="/mnt/armadillo_x1.dtb"
                ;;
            *"Armadillo-X1L Board"*)
                DTB_PATH="/mnt/armadillo_x1l.dtb"
                ;;
            *"Armadillo-IoT"*"G3 Board"*)
                DTB_PATH="/mnt/armadillo_iotg_g3.dtb"
                ;;
            *"Armadillo-IoT"*"G3 M1 Board"*)
                DTB_PATH="/mnt/armadillo_iotg_g3_m1.dtb"
                ;;
            *"Armadillo-IoT"*"G3 W2 Board"*)
                DTB_PATH="/mnt/armadillo_iotg_g3_w2.dtb"
                ;;
            *)
                die "unknown model($model)"
                ;;
        esac
```

```
        ;;
    *)
        die "unknown model($model)"
        ;;
esac

mkdir -p $LOCKDIR || die "making LOCKDIR($LOCKDIR) is failed"

ledctrl red blink_on 100

TMPDIR=$(mktemp -d /tmp/web_software_update.XXXXXXXX)
if [ -z "$TMPDIR" ]; then
    die "mktemp failed"
fi

cd $TMPDIR

update

cd
rm -rf $TMPDIR


ledctrl red blink_off

rmdir $LOCKDIR

if [ "$FILE_IS_UPDATED" = true ]; then
    log info "reboot"
    reboot
fi
```

図 8.20 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_software_update.sh)

html2 コマンドを使用するには、xml2 をインストールしておく必要があります。

```
[armadillo ~]# apt install xml2
```

8.9. PC と Armadillo 間でファイルを共有する

Armadillo 上で動くプログラムを開発する場合、開発作業自体は PC 上やあるいは PC 上で動いている ATDE 上で行うことが多いです。

PC 上で作成したプログラムを動作確認する場合、Armadillo へ転送する必要がありますが、動作確認のたびに scp などでファイル転送するのは大変です。そのような煩わしさを解決する方法の 1 つとして PC と Armadillo 間でのファイル共有があります。

ここでは、samba を使ったファイル共有の方法について説明します。

8.9.1. samba をインストールする

samba は apt でインストールできます。

```
[armadillo ~]# apt install samba
```

8.9.2. samba を設定する

まず、共有ディレクトリへアクセスできるユーザを追加します。このユーザは Linux 上に存在しているユーザでなければなりません。新たなユーザを追加したい場合は「4.2.2. ユーザーの追加と削除」を参考に Linux にユーザを追加してください。

ここでは Linux 上のユーザである atmark を共有ディレクトリにアクセスできるユーザとして追加します。追加には pdbedit コマンドを使います。

password には共有ディレクトリにアクセスするためのパスワードを設定してください。

```
[armadillo ~]# pdbedit -a atmark
new password:
retype new password:
Unix username:        atmark
NT username:
Account Flags:        [U          ]
User SID:             S-1-5-21-1214478273-341451916-2061574801-1001
Primary Group SID:    S-1-5-21-1214478273-341451916-2061574801-513
Full Name:
Home Directory:       ¥¥armadillo¥atmark
HomeDir Drive:
Logon Script:
Profile Path:         ¥¥armadillo¥atmark¥profile
Domain:               ARMADILLO
Account desc:
Workstations:
Munged dial:
Logon time:           0
Logoff time:          never
Kickoff time:         never
Password last set:    Fri, 08 May 2020 12:45:59 JST
Password can change:  Fri, 08 May 2020 12:45:59 JST
Password must change: never
Last bad password   : 0
Bad password count  : 0
Logon hours         : FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

次に共有ディレクトリを作成します。共有ディレクトリは/home/atmark の下に作成し、所有ユーザとグループを atmark に変更します。

```
[armadillo ~]# mkdir /home/atmark/share
[armadillo ~]# chown atmark:atmark /home/atmark/share
```

次に samba の設定ファイルを修正し、作成したディレクトリを共有します。

/etc/samba/smb.conf ファイルの末尾に以下を追記します。

```
[armadillo ~]# vi /etc/samba/smb.conf
(省略)

[Share] ❶
   path = /home/atmark/share ❷
   read only = no ❸
   guest only = no ❹
   guest ok = no ❺
```

❶ 任意の共有名称です。Share でなくてもかまいません。

❷ 共有ディレクトリのパスです。

❸ 読み取り専用とはしません。

❹ ゲストユーザのアクセスを不可にします。

❺ ゲストユーザのアクセスを不可にします。

最後に samba を再起動します。

```
[armadillo ~]# systemctl restart smbd nmbd
```

ここまでで、基本的な samba の設定は完了です。

samba の設定ファイルにはここで紹介したものの他にも数多くの設定項目があります。詳細は Samba ユーザ会のサイトを参照してください。[4]

8.9.3. Windows からアクセスする

前章で Armadillo 上に作成した samba の共有ディレクトリに Windows からアクセスするには、ネットワークドライブの割り当てをします。

ネットワークドライブの割り当てをする際に表示される「フォルダー」入力欄に以下のように Armadillo の IP アドレスと共有ディレクトリ名を入力します。

```
¥¥<Armadillo の IP アドレス>¥share
```

次に、ユーザ名とパスワードの入力を求められるので、「8.9.2. samba を設定する」で設定したユーザ名(ここでは atmark)とパスワードを入力します。

これで Windows から share ディレクトリに対してファイルの読み書きができるようになります。

8.9.4. ATDE からアクセスする

共有ディレクトリに ATDE からアクセスするには mount コマンドで共有ディレクトリを任意の場所にマウントします。

[4]http://www.samba.gr.jp/project/translation/current/htmldocs/manpages/smb.conf.5.html

```
[ATDE ~]$ mkdir share ❶
[ATDE ~]$ sudo mount -t cifs -o username=<ユーザ名>,password=<パスワード> //<Armadillo の IP アドレス>/share share ❷
```

❶　マウントするためのディレクトリを作成します。

❷　作成したディレクトリにマウントします。

マウントしたのちに、share ディレクトリに移動しファイルを作成すると、Armardillo 側からもファイルを確認できます。

このように、samba によるファイル共有を行うことで、PC と Armadillo 間でのファイルのやり取りがスムーズになり、開発効率の向上につなげることができます。

Armadillo 標準ガイド ソフトウェア開発編

改訂履歴

バージョン	年月日	改訂内容
1.0.0	2020/05/20	・初版発行
1.0.1	2022/08/24	・誤記修正

Armadillo 標準ガイド ソフトウェア開発編
Version 1.0.1
2022/08/29