Armadillo 実践開発ガイド

～組み込み Linux の導入から製品化まで～

第 3 部

Version 3.0.0
2015/10/26
linux-3.14-at 対応

Armadillo 実践開発ガイド: 〜組み込み Linux の導入から製品化まで〜: 第 3 部

株式会社アットマークテクノ

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

Version 3.0.0
2015/10/26

目次

図目次

表目次

1. はじめに

第 1 部では、Armadillo を使った組み込みシステムを構築する方法の全体像について説明を行いました。また、第 2 部では、システムの開発段階で役に立つ、より実践的な事柄について説明しました。第 3 部では、特別なデバイスを扱うなど具体的な事例を取り上げ、Howto 形式で紹介します。

1.1. 対象読者

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

1.2. 表記方法

本書で使用している表記方法について説明します。

1.2.1. 使用するフォント

フォントは以下のものを使用します。

表 1.1 使用するフォント

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

1.2.2. コマンド入力例の表記方法

1.2.2.1. Linux システムの場合

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

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

図 1.1 コマンド入力表記例(Linux システム)

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

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

表 1.2 コマンドの実行環境と対応する表記

表記	実行環境
PC	作業用 PC

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

[a]アットマークテクノ社製品用の開発環境

「~/」の部分は、カレントディレクトリのパスを表します。

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

表 1.3 ユーザーの種類と対応する表記

表記	権限
#	特権ユーザー
$	一般ユーザー

1.2.2.2. 保守モードの場合

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

```
hermit> info
```

図 1.2 コマンド入力表記例(保守モード)

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

1.2.3. コラムの表記方法

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

メモ

用語の説明や補足的な説明は、このアイコンで示します。

ヒント

知っていると便利な情報は、このアイコンで示します。

注意

ユーザーの注意が必要な情報は、このアイコンで示します。このアイコンが付いているコラムの内容に従わない場合、ハードウェアやシステムを破

壊したり、以降の作業に支障をきたす場合があります。再度、ご確認ください。

注意: 本書の内容を実践する前に

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

1.3. サンプルソースコード

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

1.4. 困った時は

本書を読んでわからなかったり困ったことがあった際は、ぜひ Armadillo 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。

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。

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

> 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。何らかの事情でフォーラムに投稿する事ができない場合は、以下にご連絡ください。

株式会社アットマークテクノ 横浜営業所
〒 221-0835 横浜市神奈川区鶴屋町 3 丁目 30-4 明治安田生命横浜西口ビル 7F
電話 045-548-5651
FAX 050-3737-4597
電子メール sales@atmark-techno.com

1.6. 商標

1.7. ライセンス

図 1.3 クリエイティブコモンズライセンス

1.8. 謝辞

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

2. ハードウェア機能をカスタマイズする

本章では、Armadillo-400 シリーズのハードウェア機能をカスタマイズする方法について説明します。

2.1. シリアルインターフェース

2.1.1. コンソールとして別のシリアルインターフェースを使用する

Armadillo-400 シリーズは、標準状態でシリアルインターフェース 1(CON3)をコンソールとして使用します。コンソールには、起動ログやカーネルメッセージなどが出力されるため、標準の設定ではシリアルインターフェース 1 に外部機器を接続して使用するといったことはできません。ここでは、コンソールとして別のシリアルインターフェースを使用する方法について説明します。例として、コンソールをシリアルインターフェース 2(CON9_3、CON9_5)に変更します。

第 1 部「起動の仕組み」でも説明したように、コンソールに文字を表示するプログラムには、ブートローダー、Linux カーネル、ユーザーランドアプリケーションプログラムの 3 種類があります。

まず、ブートローダーの起動ログとカーネルメッセージを出力する先を変更します。カーネルメッセージの出力先は、カーネルパラメータの `console` オプションで指定できます。カーネルパラメータは、ブートローダーの **setenv** コマンドで設定します。ブートローダーは、`console` オプションが指定されている場合、それと同じシリアルインターフェースに起動ログを出力します。

カーネルパラメータを設定するには、Armadillo を保守モードで起動して、「図 2.1. コンソールをシリアルインターフェース 2 に変更する」のように、使用するシリアルデバイスのデバイスファイル名を入力します。シリアルインターフェースとデバイスファイルの対応は、「Armadillo-400 シリーズ ソフトウェアマニュアル」の「UART」の章を参照してください。

```
hermit> setenv console=ttymxc2,115200
```

図 2.1 コンソールをシリアルインターフェース 2 に変更する

現在のカーネルパラメータは、**setenv** コマンドを引数なしで実行することで確認できます。また、カーネルパラメータの指定を解除し、標準状態にもどすには、**clearenv** コマンドを使用します。

```
hermit> setenv
1: console=ttymxc2,115200
```

図 2.2 カーネルパラメータの確認

`console=ttymxc2,115200` を指定した状態で起動すると、起動ログやカーネルメッセージなどはシリアルインターフェース 2 に出力されるようになります。但し、ログインプロンプトはまだシリアルインターフェース 1 に出力されます。

```
atmark-dist v1.45.0 (AtmarkTechno/Armadillo-420)
Linux 3.14.36-at4 [armv5tejl arch]

armadillo420-0 login:
```

図 2.3 ログインプロンプトの表示

ログインプロンプトをシリアルインターフェース 2 に表示するには、/etc/inittab と/etc/securetty を修正する必要があります。標準の、Atmark Dist で作成したユーザーランドの場合、/etc/inittab は「図 2.4. 標準の inittab」のようになっています。3 行目の ttymxc1 を ttymxc2 に変更すると、ログインプロンプトをシリアルインターフェース 2 に表示するようになります。また、/etc/securetty は「図 2.5. 標準の securetty」のようになっています。ttymxc2 を追加すると、シリアルインターフェース 2 に表示されたログインプロンプトから、root ユーザーでログインできるようになります。

```
::sysinit:/etc/init.d/rc

::respawn:/sbin/getty -L 115200 ttymxc1 vt102

::shutdown:/etc/init.d/reboot
::ctrlaltdel:/sbin/reboot
```

図 2.4 標準の inittab

```
ttymxc1
```

図 2.5 標準の securetty

inittab や securetty を変更するには、ユーザーランドを再構築する必要があります。第 1 部「Atmark Dist を使ったルートファイルシステムの作成」などを参照し、使用するプロダクト用に基本的な設定をして、一度ビルドした Atmark Dist を用意してください。そして、atmark-dist/vendors/AtmarkTechno/*プロダクト名*/etc/inittab と atmark-dist/vendors/AtmarkTechno/*プロダクト名*/etc/securetty を「図 2.6. ログインプロンプトをシリアルインターフェース 2 にした inittab」、「図 2.7. シリアルインターフェース 2 からの root ログインを許可した securetty」に示すように修正してユーザーランドをビルドし、作成されたルートファイルシステムイメージ(romfs.img.gz)を Armadillo のフラッシュメモリのユーザーランド領域に書き込んでください。Armadillo を再起動すると、ログインプロンプトもシリアルインターフェース 2 に表示されるようになります。

```
::sysinit:/etc/init.d/rc

::respawn:/sbin/getty -L 115200 ttymxc2 vt102

::shutdown:/etc/init.d/reboot
::ctrlaltdel:/sbin/reboot
```

図 2.6 ログインプロンプトをシリアルインターフェース 2 にした inittab

```
ttymxc1
ttymxc2
```

図 2.7 シリアルインターフェース 2 からの root ログインを許可した securetty

ユーザーランドを Debian GNU/Linux で構築している場合は、/etc/inittab のログインプロンプトに関連する部分は「図 2.8. Armadillo-400 シリーズ用 Debian GNU/Linux の inittab(抜粋)」のようになっています。ログインプロンプトをシリアルインターフェース 2 に出力するには、ttymxc1 を ttymxc2 に変更します。また、securetty は、「図 2.9. Armadillo-400 シリーズ用 Debian GNU/Linux の securetty(抜粋)」のようになっているので、ttymxc2 を追加します。

```
# Example how to put a getty on a serial line (for a terminal)
#
#T0:23:respawn:/sbin/getty -L ttyS0 9600 vt100
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100

# Example how to put a getty on a modem line.
#
#T3:23:respawn:/sbin/mgetty -x0 -s 57600 ttyS3

T1:23:respawn:/sbin/getty -L ttymxc1 115200 vt100
```

図 2.8 Armadillo-400 シリーズ用 Debian GNU/Linux の inittab(抜粋)

```
# MXC serial ports
ttymxc0
ttymxc1
```

図 2.9 Armadillo-400 シリーズ用 Debian GNU/Linux の securetty(抜粋)

2.1.2. コンソールへの出力を止める

実際の製品においては、コンソールが使える事自体が問題となる場合もあるでしょう。コンソールへの出力を止めるのも、「2.1.1. コンソールとして別のシリアルインターフェースを使用する」と同様の手順で行うことができます。

コンソールへの出力を止めるには、カーネルパラメータの console に none を指定します。

```
hermit> setenv console=none
```

図 2.10 コンソールへの出力を止める

また、/etc/inittab の **getty** に関する行は、削除するかコメントアウトします。/etc/securetty に関する設定は、ログインプロンプトを表示しなければ関係ないので、そのままで構いません[1]。

```
::sysinit:/etc/init.d/rc
```

[1]もちろん、セキュリティ面を考慮すれば、securetty にはログインを許可するインターフェースのみを記述すべきです。

```
#::respawn:/sbin/getty -L 115200 ttymxc2 vt102

::shutdown:/etc/init.d/reboot
::ctrlaltdel:/sbin/reboot
```

図 2.11 ログインプロンプトを表示しない(標準の inittab)

```
# Example how to put a getty on a serial line (for a terminal)
#
#T0:23:respawn:/sbin/getty -L ttyS0 9600 vt100
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100

# Example how to put a getty on a modem line.
#
#T3:23:respawn:/sbin/mgetty -x0 -s 57600 ttyS3

#T1:23:respawn:/sbin/getty -L ttymxc1 115200 vt100
```

図 2.12 ログインプロンプトを表示しない(Armadillo-400 シリーズ用 Debian GNU/Linux の inittab)

2.2. I2C 接続 A/D コンバーター

Armadillo-400 シリーズでは、CON11 と CON14 に I2C バスが出ており、外部のデバイスと接続することができます。Armadillo-400 シリーズ LCD 拡張ボードや Armadillo-400 シリーズ RTC オプションモジュール、Armadillo-400 シリーズ WLAN モジュールでは、I2C バスにリアルタイムクロックを接続しています。ここでは、I2C バスに A/D コンバーターを接続する方法を紹介します。

使用するソフトウェア、デバイスは以下のとおりです。

1. Linux カーネル: linux-3.14-at4 以降
2. A/D コンバーター: PCF8591(NXP 社製)

2.2.1. I2C 概要

I2C(Inter Integrated Circuit)は、IC 間のデータ転送に使われる 2 線式の通信方式です。正式には I^2C と記述し、I-squared-C(アイ・スクエアド・シー)と読みます。[2]

I2C ではシリアルデータライン(SDA)とシリアルクロック(SCL)の 2 本の信号線のみを使用して通信をおこないます。この 2 本の信号線に複数のデバイスを接続し、バスを構成することができます。I2C バスに接続するデバイスの出力段はオープンドレイン(またはオープンコレクタ)とし、信号線はプルアップします。そのため、全てのデバイスが High を出力しているときだけ信号線は High となり、どれかひとつのデバイスが Low を出力すると信号線は Low となります[3]。

I2C バスに接続されたデバイスは、その役割によってマスタとスレーブに分かれます。マスタとスレーブは、一つの I2C バスにそれぞれ複数接続することができます。通信は必ずマスタが開始し、バスに接続されたスレーブとデータのやりとりを行います。スレーブはそれぞれ固有のアドレスを持っており、マスタはアドレスを指定することで通信をおこなうスレーブを特定します。Armadillo-400 シリーズは、I2C マスタとなることができます。

[2] I2C と書かれることから、アイ・ツー・シーと発音される場合もあります。
[3] このような接続を、ワイヤード・AND といいます。

I2C では、1 クロックにつき 1bit のデータの転送を行います。そのため、データ転送速度はクロックの速度によって決まります。I2C にはいくつかのモードがあり、モードごとに転送速度の上限が決まっています。標準モードでは 0 から 100kbit/sec、ファーストモードでは 400kbit/sec まで、ハイスピードモードでは 3.4Mbit/sec までとなっています。Armadillo-400 シリーズは、ファーストモードまで対応しています。

データの転送はクロックに同期して行われます。クロックは転送を開始するマスタが生成します。SCL が High の時に SDA を High から Low に変化させることで転送が開始されます。これをスタートコンディションと呼びます。また、SCL が High の時に SDA を Low から High に変化させることでデータの転送を終了します。これをストップコンディションといいます。スタートコンディションとストップコンディションの発行は、必ずマスターによって行われます。

I2C では、1 回の転送で複数のバイトを送受信することができます。各バイトの長さは必ず 8bit になります。データは最上位ビット(MSB)から順に送信されます。SCL が High の時の SDA のレベルによって、論理が 0(SDA=Low)か 1(SDA=High)かが決定します。SCL が High の間、SDA のレベルは一定でなければなりません。SDA のレベルを変更できるのは、SCL が Low の時だけです。

各バイト(8bit)の転送ごとに、アクノリッジ(ACK)が必要になります。受信側は、正常に通信がおこなえている場合、アクノリッジ信号として、SDA を Low にします。アクノリッジ信号として SDA を High にすることで、送信側にデータの終了を知らせることができます。

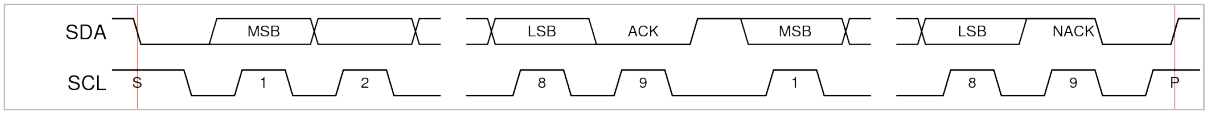


図 2.13 I2C プロトコル

2.2.2. サンプル回路

今回使用する A/D コンバーター PCF8591 は、以下の特長を持ちます。

1. 単一電源(2.5V から 6V)動作
2. I2C 接続
3. 3 つのハードウェアピンでアドレス指定可能
4. オンチップ サンプルアンドホールド回路
5. 8bit 分解能逐次比較型 A/D 入力×4
6. 8bit 分解能 D/A×1

Armadillo-400 シリーズと、A/D コンバーターとの接続を「図 2.14. I2C 接続 A/D コンバーター回路図」に示します。Armadillo-400 シリーズの CON14 から出ている I2C2 に PCF8591 を接続します。アドレスを指定する A0、A1、A2 ピンは全てプルダウンしておきます。AIN0 から AIN3 ピンがアナログ入力ピンです。アナログ入力にかかる電圧を、20kΩ の可変抵抗で変えられるようにしています。リファレンス電圧 VREF に電源電圧と同じ 3.3V を入力しているため、0V から 3.3V の範囲のアナログ入力を 8bit(256 段階)のデジタル値に変換します。

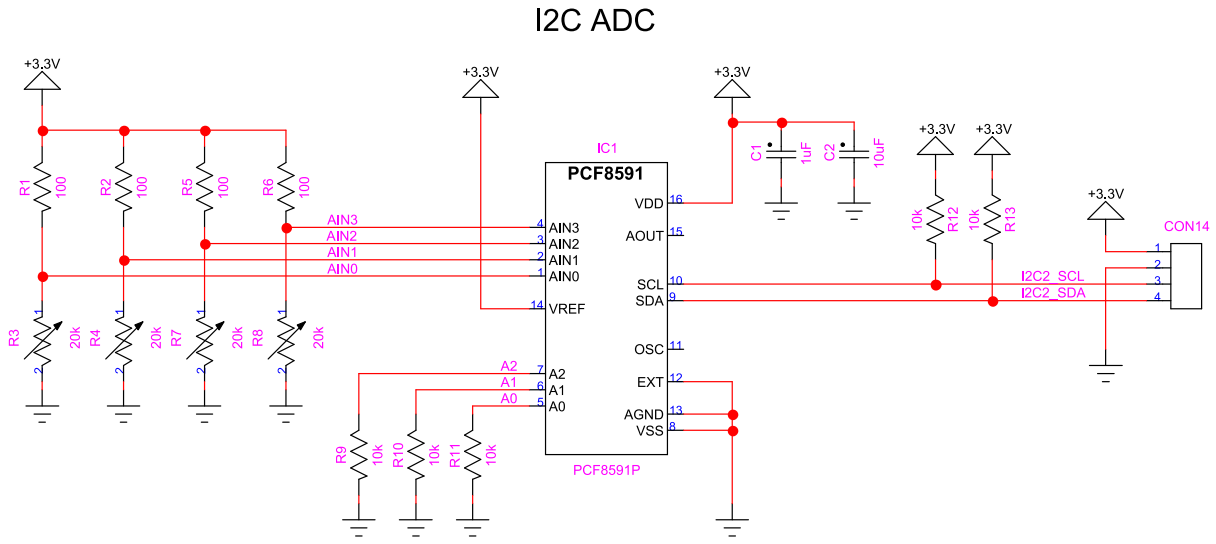
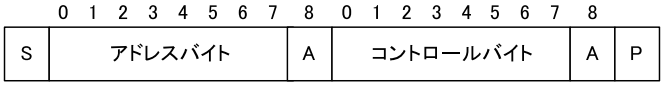


図 2.14 I2C 接続 A/D コンバーター回路図

2.2.3. PCF8591 通信プロトコル

PCF8591 は内部にレジスタを持っており、レジスタの値を変更することで、デバイスの機能を変更することができます。レジスタへの書き込みは、「図 2.15. PCF8591 通信フォーマット(コントロールバイト書き込み)」に示すフォーマットにしたがっておこないます。



S: スタートコンディション
P: ストップコンディション
A: ACK

図 2.15 PCF8591 通信フォーマット(コントロールバイト書き込み)

まず、マスタがスタートコンディションを発行し、アドレスバイトを送信します。スレーブからの ACK が返ってきたら、続いてコントロールバイトを送信します。再びスレーブからの ACK が返ってきたら、ストップコンディションを発行して、レジスタへの書き込みを完了します。

アドレスバイトのフォーマットを「図 2.16. PCF8591 通信フォーマット(アドレスバイト)」に示します。上位 7bit でスレーブのアドレスを指定します。A2、A1、A0 は、それぞれ PCF8591 の A2、A1、A0 ピンのレベルに対応した値とします。今回の例では全てプルダウンしたので、A2、A1、A0 は全て 0 を指定します。書き込み転送の場合は、R/W*を 0 とします。

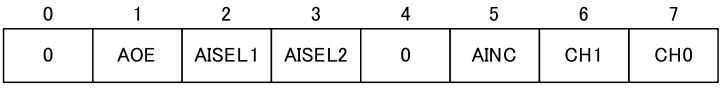
0	1	2	3	4	5	6	7
1	0	0	1	A2	A1	A0	R/W*

A2、A1、A0: PCF8591のA2、A1、A0ピンのレベルに対応した値
R/W*: 読み込み転送時は1, 書き込み転送時は0

図 2.16 PCF8591 通信フォーマット(アドレスバイト)

コントロールバイトのフォーマットを「図 2.17. PCF8591 通信フォーマット(コントロールバイト)」に示します。AOE はアナログ出力有効の場合、1 を指定します。AISEL1 と AISEL0 で、どのようにアナログ入力ピンを使用するか指定します。シングルエンド(方線接地)入力×4 とする場合は 0b00、ディファレンシャル(差動)入力×3 とする場合は 0b01、シングルエンド入力×2+ディファレンシャル入力×1

とする場合は 0b10、ディファレンシャル入力×2 とする場合は 0b11 となります。AINC はオートインクリメント有効の時、1 を指定します。CH1 と CH0 にはアナログ入力に使用するチャンネルを指定します。今回の例では、AOE、AISEL1、AISEL0、AINC を全て 0 とします。



AOE: アナログ出力有効時に1
AISEL1、AISEL2: 00=シングルエンド入力×4　チャンネル0: AIN0
チャンネル1: AIN1
チャンネル2: AIN2
チャンネル3: AIN3

01=差動入力×3　チャンネル0: AIN0 − AIN3
チャンネル1: AIN1 − AIN3
チャンネル2: AIN2 − AIN3

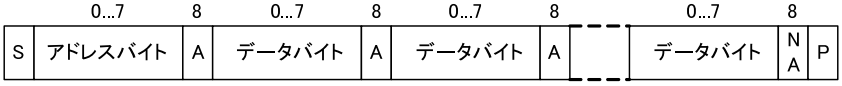
10=シングルエンド入力×2＋差動入力×1　チャンネル0: AIN0
チャンネル1: AIN1
チャンネル2: AIN2 − AIN3

11=差動入力×2　チャンネル0: AIN0 − AIN1
チャンネル1: AIN2 − AIN3

AINC: オートインクリメント有効時に1
CH1、CH2: アナログ入力チャンネル指定

図 2.17 PCF8591 通信フォーマット(コントロールバイト)

PCF8591 からデータを読み出すと、CH で指定したチャンネルの値を得ることができます。データの読み出しは、「図 2.18. PCF8591 通信フォーマット(データバイト読み出し)」に示すフォーマットでおこないます。



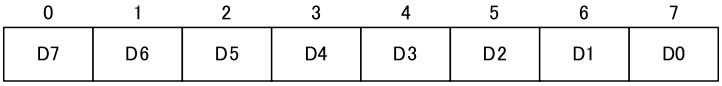
S: スタートコンディション
P: ストップコンディション
A: ACK
NA: NACK

図 2.18 PCF8591 通信フォーマット(データバイト読み出し)

まず、マスタがスタートコンディションを発行し、アドレスバイトを送信します。スレーブはアドレスバイトへの ACK を返し、続いてデータバイトを送信します。マスタはデータバイトを受信したあと、続けてデータが欲しい場合は ACK を返します。データバイトの受信を終了したい場合は NACK を返し、ストップコンディションを発行します。

アドレスバイトのフォーマットは、R/W*が 1 になる以外、コントロールバイト書き込みの場合と同じです。

データバイトのフォーマットを「図 2.19. PCF8591 通信フォーマット(データバイト)」に示します。PCF8591 では、アクノリッジ信号のトレイリングエッジで、指定されたチャンネルの入力電圧がサンプルされ、データバイトの送信中に A/D 変換がおこなわれます。そのため、データバイトで転送される値は、一つ前のデータバイト送信中に変換された値となります。なお、パワーオンリセット後の最初のデータバイトで転送される値は、0x80 となります。



D7〜D0: データバイト

図 2.19 PCF8591 通信フォーマット(データバイト)

2.2.4. i2cdev ドライバー

一般的に、I2C 接続のデバイスを Linux システムで使用する場合、I2C スレーブデバイス用のデバイスドライバーを作成して、デバイスの制御をおこないます。今回は、PCF8591 専用のデバイスドライバーを作成するのではなく、汎用の i2cdev ドライバーを使用することにします。i2cdev ドライバーを使用すると、デバイスファイルインターフェースを経由して、ユーザーランドで動作するアプリケーションプログラムからデバイスの制御をおこなうことができます。Armadillo-400 シリーズでは、標準のカーネルで i2cdev ドライバーが有効になっているため、特に何も設定しなくとも使用可能です。

i2cdev ドライバーでは、/dev/i2c-*N*(*N* は 0 から始まる数値文字)デバイスファイルに対して、open/close/read/write/ioctl システムコールを発行することで、I2C デバイスの制御をおこないます。Armadillo-400 シリーズで使用できるデバイスファイルを「表 2.1. I2C バスとデバイスファイルの対応」に示します。

表 2.1 I2C バスとデバイスファイルの対応

I2C バス	コネクタ	デバイスファイル
I2C1	なし[a]	/dev/i2c-0
I2C2	CON14	/dev/i2c-1
I2C3	CON11	/dev/i2c-2

[a]ボード内蔵バスとして使用。

アプリケーションプログラムで I2C デバイスの制御をおこなうには、まず、デバイスファイルをオープンします。

```
int fd;

fd = open("/dev/i2c-0", O_RDWR);
```

図 2.20 I2C デバイスファイルのオープン

デバイスをオープンした後、通信を行うスレーブデバイスを特定するため、スレーブデバイスのアドレスを設定します。これには、ioctl システムコールを使用します。I2C_SLAVE などの i2cdev を使用する際に必要となる定義は、<linux/i2c-dev.h>で定義されています。

```
#include <linux/i2c-dev.h>

int addr = 0x40; /* The I2C address */

ioctl(fd, I2C_SLAVE, addr);
```

図 2.21 I2C スレーブデバイスのアドレス指定

I2C スレーブデバイスとのデータ転送をおこなうには、単純に read/write システムコールを実行するだけです。スタート/ストップコンディションの発行、アドレスバイトの生成と送信、アクノリッジの処

理などは、全てドライバーがおこなってくれるので、アプリケーションプログラム側ではそれらを意識する必要はありません。

i2cdev ドライバーに関する詳しい情報は、linux-3.14-at*[version]*/Documentation/i2c/dev-interface を参照してください。

2.2.5. サンプルプログラム

PCF8591 と通信をおこない、A/D 変換結果を表示するサンプルプログラムを紹介します。プログラムは「図 2.22. PCF8591 を使用した A/D 変換プログラム」に示すように、オプションとしてデバイスファイル名と A/D 変換をおこなうチャンネルを指定することにします。

```
adc_pcf8591 <-d|--device FILENAME> [-c|--channel CHANNEL]
```

図 2.22 PCF8591 を使用した A/D 変換プログラム

main 関数を「図 2.23. adc_pcf8591.c」に示します。pcf8591_で始まる名前の関数で、実際の制御をおこないます。main 関数では、以下の処理をおこなっています。

1. pcf8591_open()で、デバイスファイル名とアドレスを指定してデバイスをオープンする
2. pcf8591_read()で、チャンネルを指定してデジタル値を読み出す
3. デジタル値を電圧に変換して表示
4. pcf8591_close()で、デバイスをクローズする

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

#include "pcf8591.h"

#define MAIN_C
#include "exitfail.h"

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

#define REF_VOLTAGE     3.3     /* 基準電圧[V] */
#define I2C_ADDR        0x48    /* PCF8591 の I2C アドレス */
#define DEFAULT_CH      0       /* オプションで指定されなかった場合に使用するチャンネル */

static void usage(char *prog)
{
        printf("Usage: %s <-d|--device FILENAME> [-c|--channel CHANNEL]\n", BASENAME(prog));
}

static void parse_arg(int argc, char *argv[], char **device, int *ch)
{
        int c;
        char *endptr;

        *device = NULL;
```

```
        for(;;) {
                int option_index = 0;
                static struct option long_options[] = {
                        /* name,        has_arg,           flag, val*/
                        {"device",      required_argument, NULL, 'd'},
                        {"channel",     required_argument, NULL, 'c'},
                        {0,             0,                 0,    0},
                };

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

                switch (c) {
                case 'd':
                        *device = optarg;
                        break;
                case 'c':
                        errno = 0;
                        *ch = strtol(optarg, &endptr, 0);
                        if (errno != 0 || optarg == endptr)
                                goto err;

                        if (*ch < PCF8591_CH_MIN || PCF8591_CH_MAX < *ch)
                                goto err;
                        break;
                default:
                        goto err;
                }
        }

        if (*device == NULL)
                goto err;

        return;

err:
        usage(argv[0]);
        exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
        struct pcf8591 *adc;
        char *device;
        int channel = DEFAULT_CH;
        double voltage;
        uint8_t digital_code;
        int ret;

        exitfail_init();

        parse_arg(argc, argv, &device, &channel);

        adc = pcf8591_open(device, I2C_ADDR);
        if (adc == NULL)
                exitfail_errno("pcf8591_open");
```

```
        ret = pcf8591_read(adc, channel, &digital_code);
        if (ret != 0)
                exitfail_errno("pcf8591_read");

        voltage = digital_code * REF_VOLTAGE /
                  ((1 << PCF8591_RESOLUTION_BITS) - 1);

        printf("%1.3fV\n", voltage);

        ret = pcf8591_close(adc);
        if (ret != 0)
                exitfail_errno("pcf8591_close");

        return EXIT_SUCCESS;
}
```

図 2.23 adc_pcf8591.c

実際の処理は「図 2.25. pcf8591.c」に記述してあります。「図 2.24. pcf8591.h」には、「図 2.25. pcf8591.c」で記述されている関数のプロトタイプ宣言が記述されています。

```
#ifndef PCF8591_H
#define PCF8591_H

#include <stdint.h>

#define PCF8591_RESOLUTION_BITS 8   /* PCF8591 の分解能(bit) */

#define PCF8591_CH_MIN 0 /* PCF8591 で指定できるアナログ入力チャンネルの最小値 */
#define PCF8591_CH_MAX 3 /* PCF8591 で指定できるアナログ入力チャンネルの最大値 */

struct pcf8591;

struct pcf8591 *pcf8591_open(const char *dev_path, int addr);
int pcf8591_read(struct pcf8591 *adc, int ch, uint8_t *digit);
int pcf8591_close(struct pcf8591 *adc);

#endif /* PCF8591_H */
```

図 2.24 pcf8591.h

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/i2c-dev.h>

#include "pcf8591.h"

#define PCF8591_ADDR_MIN 0x48 /* PCF8591 のアドレスの最小値 */
#define PCF8591_ADDR_MAX 0x4F /* PCF8591 のアドレスの最大値 */
```

```
#define PCF8591_RETRY_MAX 3 /* read()と write()のリトライ回数 */

/* PCF8591 の CONTROL BYTE の設定 */
#define PCF8591_ANALOG_OUTPUT_ENABLE          (1 << 6)
#define PCF8591_ANALOG_OUTPUT_DISABLE         (0 << 6)

#define PCF8591_SINGLE_ENDED                  (0 << 4)
#define PCF8591_THREE_DIFFERENTIAL            (1 << 4)
#define PCF8591_SINGLE_ENDED_AND_DIFFERENTIAL (2 << 4)
#define PCF8591_TWO_DIFFERENTIAL              (3 << 4)

#define PCF8591_AUTO_INCREMENT_ENABLE         (1 << 2)
#define PCF8591_AUTO_INCREMENT_DISABLE        (0 << 2)

#define PCF8591_CH_SHIFT                      (0)


struct pcf8591 {
        int fd;
};

static ssize_t write_uninterruptible(int fd, const void *buf, size_t count,
                                     int retry)
{
        ssize_t ret;
        int i;

        for (i = 0; i < retry; i++) {
                ret = write(fd, buf, count);
                if (ret < 0 && errno != EINTR)
                        return -1;

                if ((size_t)ret == count)
                        return ret;
        }

        errno = ETIMEDOUT;
        return -1;
}

static ssize_t read_uninterruptible(int fd, void *buf, size_t count, int retry)
{
        size_t read_length = 0;
        ssize_t ret;
        int i;

        for (i = 0; i < retry; i++) {
                ret = read(fd, buf + read_length, count - read_length);
                if (ret < 0) {
                        if (errno == EINTR)
                                continue;
                        else
                                return -1;
                }

                read_length += ret;
                if (read_length == count)
```

```
                        return read_length;
        }

        errno = ETIMEDOUT;
        return -1;
}

/**
 * 指定されたデバイスファイルをオープンする
 *
 * @param dev_path  PCF8591 が接続された i2cdev のデバイスファイルへのパス。
 * @param addr      PCF8591 のアドレス。
 *
 * @return 成功すると struct pcf8591 へのポインタを返す。
 *         失敗すると NULL を返す。その際、適切な errno を設定する。
 */
struct pcf8591 *pcf8591_open(const char *dev_path, const int addr)
{
        struct pcf8591 *adc;
        int error;
        int ret;

        if (dev_path == NULL ||
            addr < PCF8591_ADDR_MIN || PCF8591_ADDR_MAX < addr) {
                errno = EINVAL;
                return NULL;
        }

        adc = calloc(1, sizeof(struct pcf8591));
        if (adc == NULL)
                return NULL;

        adc->fd = open(dev_path, O_RDWR);
        if (adc->fd < 0) {
                error = errno;
                goto err1;
        }

        ret = ioctl(adc->fd, I2C_SLAVE, addr);
        if (ret < 0) {
                error = errno;
                goto err2;
        }

        return adc;

err2:
        close(adc->fd);
err1:
        free(adc);
        errno = error;
        return NULL;
}

/**
 * 指定されたチャンネルの値を A/D コンバータにセットし、
 * PCF8591 から A/D 変換結果を読み込む。
 * 読み込んだ A/D 変換結果を digit に格納する。
```

```
 *
 * @param adc    オープン済みの PCF8591 デバイスへのポインタ。
 * @param ch     サンプリングするチャンネル。
 * @param digit  A/D 変換された値が格納される。
 *
 * @return 成功すると 0 を返し、digit に A/D 変換された値を格納する。
 *         失敗すると-1 を返す。その際、適切な errno を設定する。
 */
int pcf8591_read(struct pcf8591 *adc, const int ch, uint8_t *digit)
{
        uint8_t buf[2];
        int ret;

        if (adc == NULL || digit == NULL ||
            ch < PCF8591_CH_MIN || PCF8591_CH_MAX < ch) {
                errno = EINVAL;
                return -1;
        }

        buf[0] = PCF8591_ANALOG_OUTPUT_DISABLE |
                 PCF8591_SINGLE_ENDED |
                 PCF8591_AUTO_INCREMENT_DISABLE |
                 (ch << PCF8591_CH_SHIFT);

        ret = write_uninterruptible(adc->fd, buf, 1, PCF8591_RETRY_MAX);
        if (ret < 0)
                return -1;

        /* 現在のアナログ入力を変換した値を取得するために、2 バイト読み込む */
        ret = read_uninterruptible(adc->fd, buf, 2, PCF8591_RETRY_MAX);
        if (ret < 0)
                return -1;

        *digit = buf[1];

        return 0;
}

/**
 * 指定された PCF8591 デバイスをクローズする
 *
 * @param adc オープン済みの PCF8591 デバイスへのポインタ。
 *
 * @return 成功すると 0 を返す。
 *         失敗すると-1 を返す。その際、適切な errno を設定する。
 */
int pcf8591_close(struct pcf8591 *adc)
{
        int fd;

        if (adc == NULL) {
                errno = EINVAL;
                return -1;
        }

        fd = adc->fd;
        free(adc);
```

```
        return close(fd);
}
```

図 2.25 pcf8591.c

サンプルプログラムをビルドする makefile を「図 2.26. adc_pcf8591 をビルドする makefile」に示します。

```
CROSS   := arm-linux-gnueabi

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

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

TARGET  = adc_pcf8591

all: $(TARGET)

adc_pcf8591: adc_pcf8591.o pcf8591.o
        $(CC) $(CFLAGS) -o $@ $^

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

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

図 2.26 adc_pcf8591 をビルドする makefile

adc_pcf8591.c、pcf8591.h、pcf8591.c、Makefile を同じディレクトリに置き、一つ上の common ディレクトリに第 2 部でも使用した exitfail.h を置いておきます。make コマンドを実行すると、**adc_pcf8591** がビルドされます。

```
[ATDE ~/i2c-adc]$ make
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -c -o adc_pcf8591.o adc_pcf8591.c
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -c -o pcf8591.o pcf8591.c
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -o adc_pcf8591 adc_pcf8591.o pcf8591.o
```

図 2.27 adc_pcf8591 のビルド

生成された **adc_pcf8591** を Armadillo にコピーして、実行権限をつけてください。**adc_pcf8591** を実行すると、「図 2.28. adc_pcf8591 コマンドの実行」に示すような結果が得られます。

```
[armadillo ~]# chmod +x adc_pcf8591
[armadillo ~]# ./adc_pcf8591 --device /dev/i2c-1 --channel 0
3.235V
```

図 2.28 adc_pcf8591 コマンドの実行

2.3. SPI 接続 A/D コンバーター

Armadillo-400 シリーズでは、CON9 を SPI バスとして使用することができます。ここでは、SPI バスに A/D コンバータを接続する方法を紹介します。

使用するソフトウェア、デバイスは以下のとおりです。

1. Linux カーネル: linux-3.14-at4 以降
2. A/D コンバーター: MCP3204(Microchip 社製)

2.3.1. SPI 概要

SPI(Serial Peripheral Interface)は、IC 間のデータ転送に使われる 4 線式の通信方式です。SPI では、シリアルクロック(SCLK)、マスタアウトプット/スレーブインプット(MOSI)、マスタインプット/スレーブアウトプット(MISO)、スレーブセレクト(SS)の 4 本の信号線を使用して通信をおこないます。これらの信号線に、複数のデバイスを接続してバスを構成することができます。

SPI バスに接続されたデバイスは、その役割によってマスタとスレーブに分かれます。SPI では、1 つのバスに 1 つのマスタと複数のスレーブを接続できます。通信は必ずマスタが開始し、バスに接続されたスレーブとデータのやりとりを行います。マスタは、通信をおこなうスレーブを SS 信号によって特定します。そのため、通常、スレーブ 1 つにつき 1 本の SS 信号を接続します。Armadillo-400 シリーズは、SPI マスタとなることができます。

SPI では、1 クロックにつき 1bit のデータ転送をおこないます。そのため、データ転送速度はクロックの速度によって決まります。Armadillo-400 シリーズは、約 16MHz まで対応できます。

データの転送は、マスタが SS 信号をアサートすることで開始されます。SCLK 信号に同期して、マスターからスレーブへのデータを MOSI に出力し、スレーブからマスタへのデータを MISO から入力します。データの入出力をおこなう線が分かれているため、全二重の通信をおこなうことができます。一度の転送で送受信できるビット数はデバイスごとに異なり、SPI プロトコルとしての制限はありません。

SPI では、クロックのポラリティとフェーズを指定できます。それぞれの設定を CPOL と CPHA で表します。

1. CPOL=0: クロックを出力していないとき SCLK を Low に保ちます。
    a. CPHA=0: クロックの立ち上がりでデータをラッチします。
    b. CPHA=1: クロックの立ち下がりでデータをラッチします。
2. CPOL=1: クロックを出力していないとき SCLK を High に保ちます。
    a. CPHA=0: クロックの立ち下がりでデータをラッチします。
    b. CPHA=1: クロックの立ち上がりでデータをラッチします。

CPOL と CPHA の組み合わせを、SPI モードで表現する場合もあります。

表 2.2 SPI モード

モード	CPOL	CPHA
0	0	0
1	0	1
2	1	0

モード	CPOL	CPHA
3	1	1

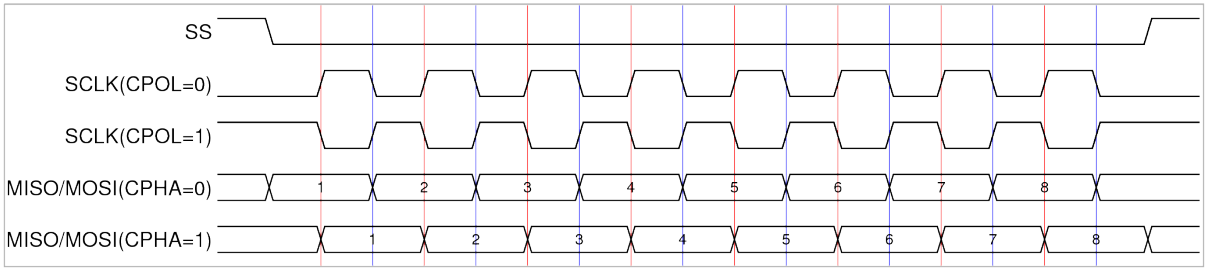


図 2.29 SPI プロトコル

2.3.2. サンプル回路

今回使用する A/D コンバーター MCP3204 は、以下の特長を持ちます。

1. 単一電源(2.7V から 5.5V)動作
2. SPI 接続
3. サンプリング速度 最大 100ksps(Vdd=5V 時)、50ksps(Vdd=2.7V 時)
4. オンチップ サンプルアンドホールド
5. 12bit 分解能逐次比較型 A/D 入力×4

Armadillo-400 シリーズと、A/D コンバーターとの接続を「図 2.30. SPI 接続 A/D コンバーター回路図」に示します。Armadillo-400 シリーズの CON9 から出ている CSPI3 に MCP3204 を接続します。SS 信号には、CSPI3 の SS0 を使用します。AIN0 から AIN3 ピンがアナログ入力ピンです。アナログ入力にかかる電圧を、20kΩ の可変抵抗で変えられるようにしています。リファレンス電圧 VREF に電源電圧と同じ 3.3V を入力しているため、0V から 3.3V の範囲のアナログ入力を 12bit(4096 段階)のデジタル値に変換します。

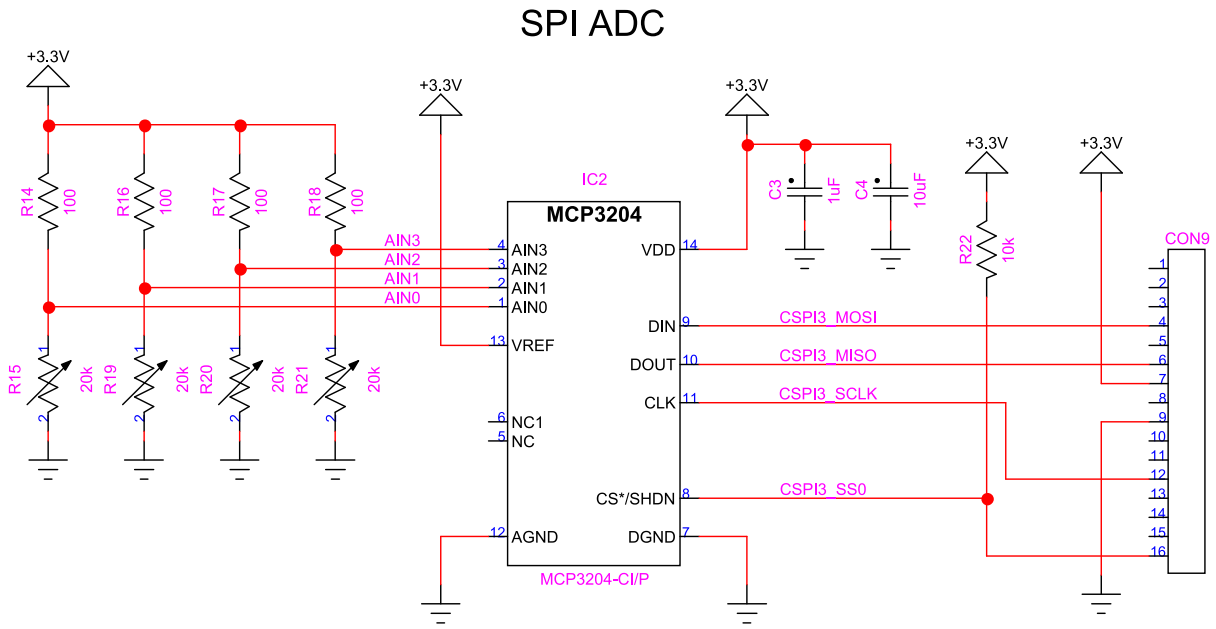


図 2.30 SPI 接続 A/D コンバーター回路図

2.3.3. MCP3204 通信プロトコル

MCP3204 は、SPI モード 0(CPOL=0、CPHA=0)または SPI モード 3(CPOL=1、CPHA=1)で通信をおこないます。MCP3204 の通信フォーマットを「図 2.31. MCP3204 通信フォーマット」に示します。

	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
MOSI	0	0	0	0	0	1	SGL/DIFF*	D2	D1	D0	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.	d.c.
MISO	x	x	x	x	x	x	x	x	x	x	x	0	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

d.c. : Don't care
x: 不定
D2、D1、D0: A/D変換を行うチャンネル指定
B11～B0: A/Dの変換結果

図 2.31 MCP3204 通信フォーマット

MOSI から 1 を出力することで、転送の開始を MCP3204 に指示します。SGL/DIFF*、D2、D1、D0 の組み合わせにより、A/D 変換をおこなうチャンネルを指定します。D0 以降、MOSI から出力されるデータは意味を持ちません。

表 2.3 MCP3204 チャンネル指定

SGL/DIFF*	D2[a]	D1	D0	入力構成	チャンネル
1	d.c.	0	0	シングルエンド	CH0
1	d.c.	0	1	シングルエンド	CH1
1	d.c.	1	0	シングルエンド	CH2
1	d.c.	1	1	シングルエンド	CH3
0	d.c.	0	0	ディファレンシャル	CH0=IN+、CH1=IN-
0	d.c.	0	1	ディファレンシャル	CH0=IN-、CH1=IN+

SGL/DIFF*	D2[a]	D1	D0	入力構成	チャンネル
0	d.c.	1	0	ディファレンシャル	CH2=IN+、CH3=IN-
0	d.c.	1	1	ディファレンシャル	CH2=IN-、CH3=IN+

[a]MCP3204 では D2 は意味を持ちません。

MCP3204 は、11 番目のクロックの上昇部でアナログ入力のサンプリングを開始し、次のクロックの下降部で完了します。A/D 変換結果は、B11 から B0 に出力されます。

2.3.4. spidev ドライバー

一般的に、SPI 接続のデバイスを Linux システムで使用する場合、SPI スレーブデバイス用のデバイスドライバーを作成して、デバイスの制御をおこないます。今回は、MCP3204 専用のデバイスドライバーを作成するのではなく、汎用の spidev ドライバーを使用することにします。spidev ドライバーを使用すると、デバイスファイルインターフェースを経由して、ユーザーランドで動作するアプリケーションプログラムからデバイスの制御をおこなうことができます。Armadillo-400 シリーズでは、標準のカーネルでは spidev ドライバーが有効になっていないため、spidev を使用するにはカーネルの設定を変更する必要があります。

spidev ドライバーでは、/dev/spidev*N.M*(*N*、*M* は 0 から始まる数値文字)デバイスファイルに対して、open/close/read/write/ioctl システムコールを発行することで、SPI デバイスの制御をおこないます。Armadillo-400 シリーズで使用できるデバイスファイルを「表 2.4. SPI バスとデバイスファイルの対応」に示します。

表 2.4 SPI バスとデバイスファイルの対応

SPI バス	コネクタ	デバイスファイル
CSPI1	CON9	/dev/spidev0.*M*(*M* は 0 または 1)
CSPI3	CON9	/dev/spidev2.*M*(*M* は 0、1、2、3 のいずれか)

アプリケーションプログラムで SPI デバイスの制御をおこなうには、まず、デバイスファイルをオープンします。

```
int fd;

fd = open("/dev/spidev0.0", O_RDWR);
```

図 2.32 SPI デバイスファイルのオープン

ioctl システムコールにより、SPI の設定を変更することができます。

1. SPI_IOC_RD_MODE, SPI_IOC_WR_MODE: 読み出しまたは書き込み時に使用する SPI モードを設定します。
2. SPI_IOC_RD_LSB_FIRST, SPI_IOC_WR_LSB_FIRST: 読み出しまたは書き込み時に LSB から転送するか、MSB から転送するか設定します。
3. SPI_IOC_RD_BITS_PER_WORD, SPI_IOC_WR_BITS_PER_WORD: 1 回の読み出しまたは書き込みで転送するビット数を設定します。
4. SPI_IOC_RD_MAX_SPEED_HZ, SPI_IOC_WR_MAX_SPEED_HZ: 読み出しまたは書き込みの最大転送速度を設定します。

read/write システムコールを使用すると、半二重通信をおこなうことができます。全二重通信をおこなうには、ioctl システムコールの SPI_IOC_MESSAGE(N) メッセージを使用します。SPI_IOC_MESSAGE(N)の使用方法は、サンプルプログラムで解説します。

spidev ドライバーに関する詳しい情報は、linux-3.14-at*[version]*/Documentation/spi/spidev を参照してください。

2.3.5. カーネルコンフィギュレーション

Armadillo-400 シリーズの標準のカーネルでは、SPI ドライバーは有効になっていません。そのため、SPI マスタドライバーと spidev ドライバーが有効になったカーネルを作成する必要があります。

まず、カーネルのソースコードアーカイブを取得します。ここでは、Armadillo サイトからダウンロードしてくることにします。

```
[ATDE ~]$ wget http://armadillo.atmark-techno.com/files/downloads/armadillo-4x0/source/kernel/
linux-3.14-at[version].tar.gz
[ATDE ~]$ tar xzvf linux-3.14-at[version].tar.gz
```

図 2.33 Linux カーネルの取得と展開

次に Armadillo-400 シリーズの標準コンフィギュレーションを適用します。

```
[ATDE ~]$ cd linux-3.14-at[version]/
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm armadillo4x0_defconfig
```

図 2.34 Linux カーネルに Armadillo-400 シリーズ標準コンフィギュレーションを適用する

続いて、menuconfig を使用して、「図 2.36. SPI ドライバーを有効にする」及び「図 2.37. SPI に使用するピンを指定する」に示すようにカーネルコンフィギュレーションを変更します。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm menuconfig
```

図 2.35 menuconfig を使用してカーネルコンフィギュレーションを変更する

```
Linux Kernel Configuration
  Device Drivers  --->
    [*] SPI support  --->                          チェックを入れる
      {*}   Utilities for Bitbanging SPI masters M から * に変更する
      <*>   Freescale i.MX SPI controllers       M から * に変更する
      <*>   User mode SPI device driver support  チェックを入れる
```

図 2.36 SPI ドライバーを有効にする

```
Linux Kernel Configuration
  System Type  --->
    Freescale i.MX support  --->
      Armadillo-400 Board options  --->
        [ ] Enable UART5 at CON9           チェックを外す
```

```
        [*] Enable SPI3 at CON9           チェックを入れる
        [*]   Enable SPI3_SS0 at CON9_16  標準で選択されているのでそのまま
        [ ]   Enable SPI3_SS1 at CON9_18  チェックを外す
        [ ]   Enable SPI3_SS2 at CON9_15  チェックを外す
        [ ]   Enable SPI3_SS3 at CON9_17  チェックを外す
```

図 2.37 SPI に使用するピンを指定する

また、カーネルのソースコードにも一部修正が必要になります。SPI スレーブデバイスドライバーを使用するには、spi_board_info を登録する必要があります。spi_board_info の登録は、linux-3.14-at*[version]*/arch/arm/mach-imx/armadillo4x0_extif.c でおこなっています。armadillo4x0_spi2_board_info を、「図 2.38. CSPI3、SS0 を spidev で使用する修正」に示すように修正してください。

```
static struct spi_board_info armadillo4x0_spi2_board_info[] __initdata = {
        {
                .modalias = "spidev",
                .max_speed_hz = 1000000,
                .bus_num = 2,
                .chip_select = 0,
        },
};
```

図 2.38 CSPI3、SS0 を spidev で使用する修正

コンフィギュレーションの変更と、ソースの修正をおこなったら、カーネルをビルドします。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- && gzip -c arch/ ⏎
arm/boot/Image > linux.bin.gz
```

図 2.39 Linux カーネルをビルドする

正常にビルドが完了すると、linux-3.14-at*[version]*/linux.bin.gz にカーネルイメージが作成されます。linux.bin.gz を Armadillo のフラッシュメモリのカーネル領域に書き込んでください。

2.3.6. サンプルプログラム

MCP3204 と通信をおこない、A/D 変換結果を表示するサンプルプログラムを紹介します。プログラムは「図 2.40. MCP3204 を使用した A/D 変換プログラム」に示すように、オプションとしてデバイスファイル名と A/D 変換をおこなうチャンネルを指定することにします。

```
adc_mcp3204 <-d|--device FILENAME> [-c|--channel CHANNEL]
```

図 2.40 MCP3204 を使用した A/D 変換プログラム

main 関数を「図 2.41. adc_mcp3204.c」に示します。mcp3204_というプレフィックスがついた関数で、実際の制御をおこないます。main 関数では、以下の処理をおこなっています。

1. mcp3204_open()で、デバイスファイル名を指定してデバイスをオープンする
2. mcp3204_read()で、チャンネルを指定してデジタル値を読み出す

3. デジタル値を電圧に変換して表示

4. mcp3204_close()で、デバイスをクローズする

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <getopt.h>

#include "mcp3204.h"

#define MAIN_C
#include "exitfail.h"

#define BASENAME(p)   ((strrchr((p), '/') ? : ((p) - 1)) + 1)

#define REF_VOLTAGE     3.3     /* 基準電圧[V] */
#define DEFAULT_CH      0       /* オプションで指定されなかった場合に使用するチャンネル */

static void usage(const char *prog)
{
        printf("usage: %s <-d|--device FILENAME> [-c|--channel CHANNEL]\n", BASENAME(prog));
}

static void parse_arg(int argc, char *argv[], const char **device, int *ch)
{
        int c;
        char *endptr;

        *device = NULL;

        for(;;) {
                int option_index = 0;
                static struct option long_options[] = {
                        /* name,        has_arg,           flag, val*/
                        {"device",      required_argument, NULL, 'd'},
                        {"channel",     required_argument, NULL, 'c'},
                        {0,             0,                 0,    0},
                };

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

                switch (c) {
                case 'd':
                        *device = optarg;
                        break;
                case 'c':
                        errno = 0;
                        *ch = strtol(optarg, &endptr, 0);
                        if (errno != 0 || optarg == endptr)
                                goto err;

                        if (*ch < MCP3204_CH_MIN || MCP3204_CH_MAX < *ch)
                                goto err;
```

```
                        break;
                default:
                        goto err;
                }
        }

        if (*device == NULL)
                goto err;

        return;

err:
        usage(argv[0]);
        exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
        struct mcp3204 *adc;
        const char *device;
        int channel = DEFAULT_CH;
        double voltage;
        uint16_t digital_code;
        int ret;

        exitfail_init();

        parse_arg(argc, argv, &device, &channel);

        adc = mcp3204_open(device);
        if (adc == NULL)
                exitfail_errno("mcp3204_open");

        ret = mcp3204_read(adc, channel, &digital_code);
        if (ret != 0)
                exitfail_errno("mcp3204_read");

        voltage = digital_code * REF_VOLTAGE /
                ((1 << MCP3204_RESOLUTION_BITS) - 1);

        printf("%1.3fV\n", voltage);

        ret = mcp3204_close(adc);
        if (ret != 0)
                exitfail_errno("mcp3204_close");

        return EXIT_SUCCESS;
}
```

図 2.41 adc_mcp3204.c

実際の処理は「図 2.43. mcp3204.c」に記述してあります。「図 2.42. mcp3204.h」には、「図 2.43. mcp3204.c」で記述されている関数のプロトタイプ宣言が記述されています。

```
#ifndef MCP3204_H
#define MCP3204_H
```

```
#include <stdint.h>

#define MCP3204_RESOLUTION_BITS 12 /* MCP3204 の分解能(bit) */

#define MCP3204_CH_MIN 0 /* MCP3204 で指定できるアナログ入力チャンネルの最小値 */
#define MCP3204_CH_MAX 3 /* MCP3204 で指定できるアナログ入力チャンネルの最大値 */

struct mcp3204;

struct mcp3204 *mcp3204_open(const char *dev_path);
int mcp3204_read(struct mcp3204 *adc, int ch, uint16_t *digit);
int mcp3204_close(struct mcp3204 *adc);

#endif /* MCP3204_H */
```

図 2.42 mcp3204.h

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>

#include <linux/types.h>
#include <linux/spi/spidev.h>

#include "mcp3204.h"

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

#define MCP3204_START_BIT    (1 << 2)
#define MCP3204_SINGLE_ENDED (1 << 1)
#define MCP3204_DIFFERENTIAL (0 << 1)
#define MCP3204_CH_SHIFT     (6)

#define MCP3204_SPI_MODE        (SPI_CPOL | SPI_CPHA)
#define MCP3204_SPI_SPEED_HZ    1000000
#define MCP3204_SPI_DELAY_USECS 0
#define MCP3204_SPI_BITS        8

struct mcp3204 {
        int fd;
};

/**
 * 指定されたデバイスファイルをオープンする
 *
 * @param dev_path  MCP3204 が接続された spidev デバイスファイルへのパス。
 *
 * @return 成功すると struct mcp3204 へのポインタを返す。
 *         失敗すると NULL を返す。その際、適切な errno を設定する。
 */
struct mcp3204 *mcp3204_open(const char *dev_path)
{
        struct mcp3204 *adc;
```

```
        uint8_t mode = MCP3204_SPI_MODE;
        int error;
        int ret;

        if (dev_path == NULL) {
                errno = EINVAL;
                return NULL;
        }

        adc = calloc(1, sizeof(struct mcp3204));
        if (adc == NULL)
                return NULL;

        adc->fd = open(dev_path, O_RDWR);
        if (adc->fd < 0) {
                error = errno;
                goto err1;
        }

        /* SPI モードを設定する */
        ret = ioctl(adc->fd, SPI_IOC_WR_MODE, &mode);
        if (ret < 0) {
                error = errno;
                goto err2;
        }

        return adc;

err2:
        close(adc->fd);
err1:
        free(adc);
        errno = errno;
        return NULL;
}

/**
 * 指定されたチャンネルの A/D 変換結果を読み込む。
 * 読み込んだ A/D 変換結果を digit に格納する。
 *
 * @param adc     オープン済みの MCP3204 デバイスへのポインタ。
 * @param ch      サンプリングするチャンネル。
 * @param digit   A/D 変換結果のデジタル値が格納される。
 *
 * @return 成功すると 0 を返し、voltage に電圧を格納する。
 *         失敗すると-1 を返す。その際、適切な errno を設定する。
 */
int mcp3204_read(struct mcp3204 *adc, int ch, uint16_t *digit)
{
        uint8_t tx[3] = {0, };
        uint8_t rx[3] = {0, };
        struct spi_ioc_transfer tr;
        int ret;

        if (adc == NULL || digit == NULL ||
            ch < MCP3204_CH_MIN || MCP3204_CH_MAX < ch) {
                errno = EINVAL;
                return -1;
```

```
        }

        /* 送信バッファにスタートビット、SGL/DIFF*、D2、D1、D0 をセットする */
        tx[0]            = MCP3204_START_BIT | MCP3204_SINGLE_ENDED;
        tx[1]            = ch << MCP3204_CH_SHIFT;

        /* 転送設定をセットする */
        tr.tx_buf        = (unsigned long)tx;
        tr.rx_buf        = (unsigned long)rx;
        tr.len           = ARRAY_SIZE(tx);
        tr.delay_usecs   = MCP3204_SPI_DELAY_USECS;
        tr.speed_hz      = MCP3204_SPI_SPEED_HZ;
        tr.bits_per_word = MCP3204_SPI_BITS;
        tr.cs_change     = 0;

        /* 全二重通信をおこなう */
        ret = ioctl(adc->fd, SPI_IOC_MESSAGE(1), &tr);
        if (ret < 1)
                return -1;

        /* 受信バッファから A/D 変換結果を取り出す */
        *digit = (rx[1] & 0x0f) << 8;
        *digit |= rx[2];

        return 0;
}

/**
 * 指定された MCP3204 デバイスをクローズする
 *
 * @param adc オープン済みの MCP3204 デバイスへのポインタ。
 *
 * @return 成功すると 0 を返す。
 *         失敗すると-1 を返す。その際、適切な errno を設定する。
 */
int mcp3204_close(struct mcp3204 *adc)
{
        int fd;

        if (adc == NULL) {
                errno = EINVAL;
                return -1;
        }

        fd = adc->fd;
        free(adc);

        return close(fd);
}
```

図 2.43 mcp3204.c

サンプルプログラムをビルドする makefile を「図 2.44. adc_mcp3204 をビルドする makefile」に示します。

```
CROSS   := arm-linux-gnueabi
```

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

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

TARGET  = adc_mcp3204

all: $(TARGET)

adc_mcp3204: adc_mcp3204.o mcp3204.o
        $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

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

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

図 2.44 adc_mcp3204 をビルドする makefile

adc_mcp3204.c、mcp3204.h、mcp3204.c、Makefile を同じディレクトリに置き、一つ上の common ディレクトリに第 2 部でも使用した exitfail.h を置いておきます。make コマンドを実行すると、**adc_mcp3204** がビルドされます。

```
[ATDE ~/spi-adc]$ make
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -c -o adc_mcp3204.o adc_mcp3204.c
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -c -o mcp3204.o mcp3204.c
arm-linux-gnueabi-gcc -Wall -Wextra -O2 -I../common -o adc_mcp3204 adc_mcp3204.o mcp3204.o
```

図 2.45 adc_mcp3204 のビルド

生成された **adc_mcp3204** を Armadillo にコピーして、実行権限をつけてください。**adc_mcp3204** を実行すると、「図 2.46. adc_mcp3204 コマンドの実行」に示すような結果が得られます。

```
[armadillo ~]# chmod +x adc_mcp3204
[armadillo ~]# ./adc_mcp3204 --device /dev/spidev2.0 --channel 0
3.235V
```

図 2.46 adc_mcp3204 コマンドの実行

2.4. 1-Wire 接続温度センサ

Armadillo-400 シリーズでは、CON9 2 ピンと CON9 26 ピンを 1-Wire バスとして使用することができます。ここでは、1-Wire バスに温度センサ IC を接続する方法を紹介します。

使用するソフトウェア、デバイスは以下のとおりです。

1. Linux カーネル: linux-3.14-at4 以降
2. 温度センサ: DS18B20(MAXIM 社製)

2.4.1. 1-Wire 概要

1-Wire は、IC 間のデータ転送に使われる 1 線式の通信方式です。最低限、1 本の信号線と接地線の 2 本だけでバスを構成することができます。このとき、電力は信号線から得ます。なお、電源線を別途用意し、3 線で構成することもできます。1-Wire バスに接続されたデバイスの出力段はオープンドレインとし、信号線はプルアップします。

1-Wire バスに接続されたデバイスは、その役割によってマスタとスレーブに分かれます。1-Wire では、1 つのバスに 1 つのマスタと複数のスレーブを接続できます。通信は必ずマスタが開始し、バスに接続されたスレーブとデータのやりとりを行います。スレーブデバイスは、チップごとに固有な 64bit の ROM ID を持っており、マスタは ROM ID を指定することで通信をおこなうスレーブを特定します。Armadillo-400 シリーズは、1-Wire マスタとなることができます。

1-Wire では、クロック信号がないため、タイムスロットに基づいてデータの転送をおこないます。マスタからスレーブに値を書き込む場合、マスタからローパルスを出力します。パルスの立ち下がりエッジでスレーブ内の単安定マルチバイブレーターが開始し、立ち下がりエッジから約 30μsec の時点でサンプリングをおこないます。そのため、マスタは 1 を書き込む場合は 1 から 15μsec の短いローパルスを出力し、0 を書き込む場合は 60μsec の長いローパルスを出力します。

スレーブからの値を読み出す場合、まず、マスタがローパルス 1 から 15μsec の短いローパルスを出力します。スレーブ側は、1 を送信したい場合何もしません。0 を送信したい場合、60μsec の間信号線をローに引っ張ります。マスタは、立ち下がりエッジから 30μsec の時点でサンプリングをおこない、スレーブからの出力をサンプリングします。

なお、タイムスロットにはスタンダード(1 タイムスロット 60μsec)とオーバードライブ(1 タイムスロット 8μsec)の二つがあります。上記の説明はスタンダードの場合のタイミングです。

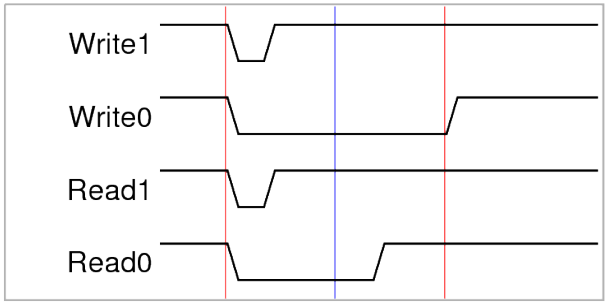


図 2.47 1-Wire プロトコル(ビット転送)

マスターとスレーブ間でのデータ転送は 3 つのシーケンスでおこないます。3 つのシーケンスは、リセットシーケンス、ROM コマンドシーケンス、ファンクションシーケンスの順番に実行されます。

リセットシーケンスでは、まず、マスタがリセットパルスを出力します。1-Wire バスにスレーブが接続されている場合、スレーブはプレゼンスパルスを出力します。

ROM コマンドシーケンスでは、マスタが 8bit の ROM コマンドを出力した後、64bit の ROM ID を出力します。ROM ID の先頭 8bit はデバイス種類を示すファミリーコードです。続く 48bit がシリアルナンバーになっています。最後の 8bit は CRC です。ROM コマンドには次のものがあります。

1. SEARCH ROM(0xF0): バスに接続されているスレーブデバイスの ROM ID を得ることができます。1 回の SEARCH ROM コマンドで 1 つのデバイスの ROM ID を特定することができます。

2. READ ROM(0x33): バスに接続されているスレーブデバイスが一つだけの場合、SEARCH ROM コマンドの代わりに READ ROM コマンドを使用して、ROM ID を得ることができます。

3. MATCH ROM(0x55): MATCH ROM コマンドでマスタが出力した ROM ID に一致したスレーブデバイスが、続くファンクションコマンドに応答します。それ以外のデバイスは、次のリセットシーケンスを待ちます。

4. SKIP ROM(0xcc): SKIP ROM コマンドに続いて送信されたファンクションコマンドは、バスに接続されているスレーブデバイス全てに同時に適用されます。

ファンクションシーケンスは、スレーブデバイスごとに異なります。基本的には、マスタが 8bit のフォワードコマンド出力した後、データの読み出しまたは書き込みをおこないます。

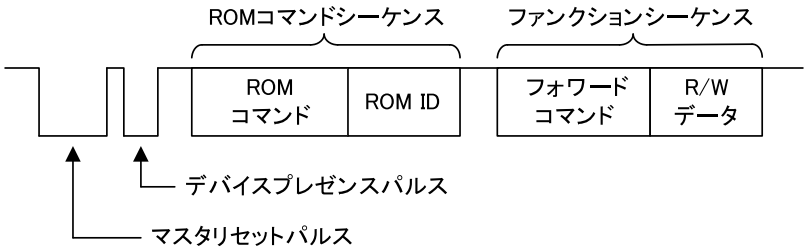


図 2.48 1-Wire プロトコル

2.4.2. DS18B20

今回使用する温度センサ DS18B20 は、以下の特長を持ちます。

1. 単一電源(3.0V から 5.5V)動作
2. 電源は信号線から供給可能
3. 外部部品不要
4. 温度計測範囲-55℃から+125℃
5. 分解能 9bit から 12bit
6. 変換時間 750msec(最大 12bit 時)

DS18B20 のファンクションコマンドには以下のものがあります。

1. CONVERT T(0x44): このコマンドにより、温度変換がおこなわれます。変換結果は、DS18B20 の内蔵 2 バイトレジスタに格納されます。
2. WRITE SCRATCHPAD(0x48): DS18B20 の内蔵メモリに書き込みをおこないます。書き込むデータは 3 バイト長で、T_H、T_L、Configuration Register の順番に送信します。
3. READ SCRATCHPAD(0xbe): DS18B20 の内蔵メモリを読み出します。読み出すデータのバイト数は最大 9 バイトです。途中で、マスタからリセットパルスを送信することで、データの読み出しを中断できます。

DS18B20 内蔵レジスタは次のようになっています。

表 2.5 DS18B20 内蔵レジスタ

バイト	内容
0	Temperature Register LSB

バイト	内容
1	Temperature Register MSB
2	T_H or User Byte 1
3	T_L or User Byte 2
4	Configuration Register
5	Reserved (0xff)
6	Reserved
7	Reserved (0x10)
8	CRC

DS18B20 の温度センサ分解能は、Configuration Register の 5bit 目と 6 ビット目で決まります。それ以外の Configuration Register のビットは内部的に使用され、上書きすることはできません。

表 2.6 DS18B20 温度センサ分解能

BIT 6	BIT 5	分解能
0	0	9 bit
0	1	10 bit
1	0	11 bit
1	1	12 bit[a]

[a]パワーオンリセット時の設定

Temperature Register のフォーマットは次のようになっています。温度は摂氏で格納されています。12 ビット分解能の場合は、BIT10 から BIT0 全てのビットが有効です。11 ビット分解能の場合、BIT0 が不定となります。10 ビット、9 ビット分解能の場合も同様です。BIT15 から BIT11 は、温度が正の場合 0、負の場合 1 となります。

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
LS_BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
MS_BYTE	S	S	S	S	S	2^6	2^5	2^4

S = SIGN

図 2.49 DS18B20 Temperature Register フォーマット

2.4.3. サンプル回路

Armadillo-400 シリーズと温度センサとの接続を、「図 2.50. 1-Wire 接続温度センサ回路図」に示します。信号線は、CON9 2 ピンに接続します。また、今回は VDD に+3.3V を接続し電源は外部から供給します。

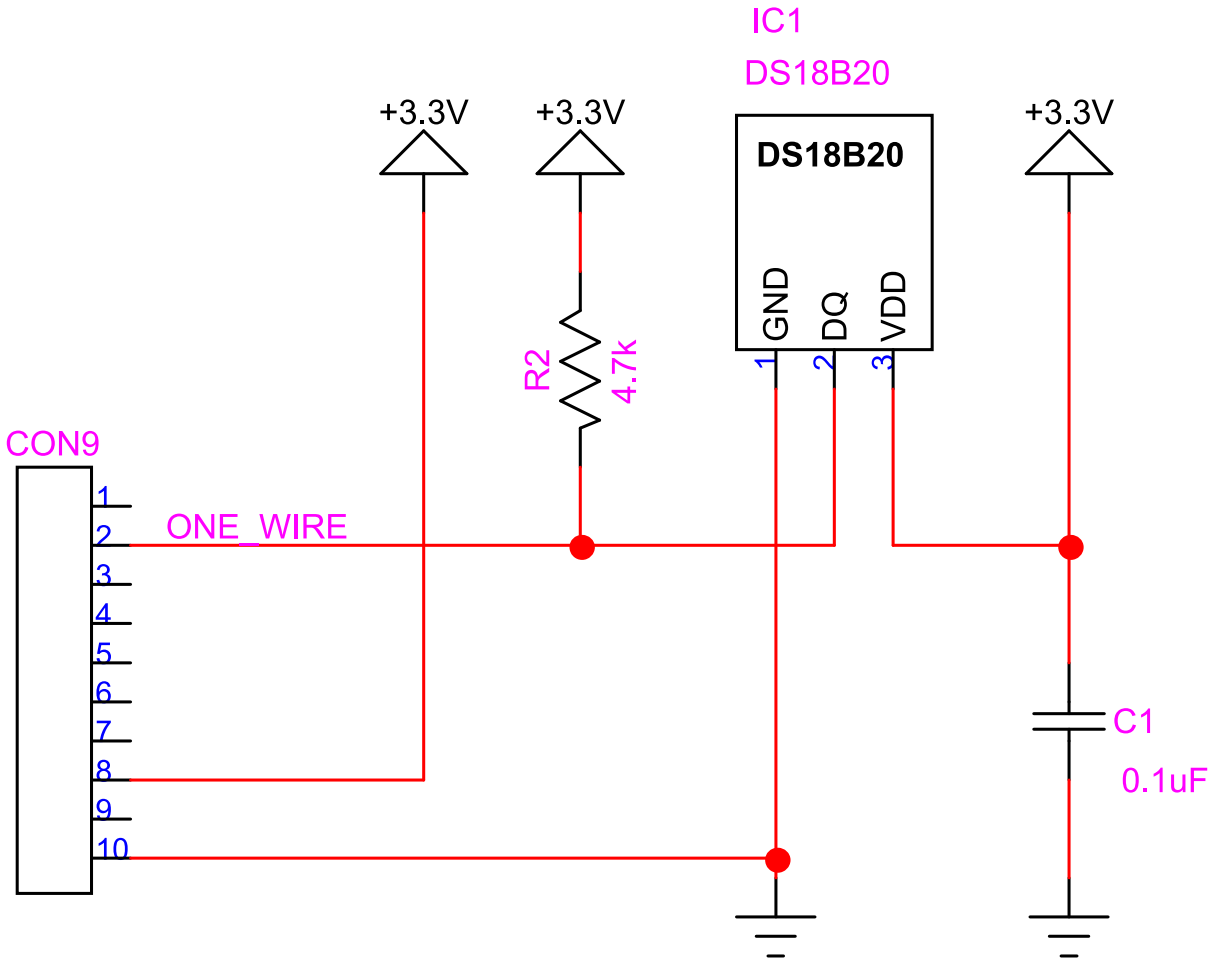


図 2.50 1-Wire 接続温度センサ回路図

2.4.4. 温度センサドライバー

一般的に、1-Wire 接続のデバイスを Linux システムで使用する場合、1-Wire スレーブデバイス用のデバイスドライバーを作成して、デバイスの制御をおこないます。今回は、1-Wire スレーブデバイスドライバーの「Thermal family implementation」を使用します。

「Thermal family implementation」を使用すると、1-Wire バスに接続された温度センサデバイスを自動で検出し、sysfs 経由で温度データの読み出しを可能にします。

```
[armadillo ~]# cd /sys/devices/w1_bus_master1/
[armadillo /sys/devices/w1_bus_master1]# ls
28-0000022e2355/           w1_master_name
driver@                    w1_master_pointer
power/                     w1_master_search
subsystem@                 w1_master_slave_count
uevent                     w1_master_slaves
w1_master_attempts         w1_master_timeout
w1_master_max_slave_count
[armadillo /sys/devices/w1_bus_master1]# cat 28-0000022e2355/w1_slave
c4 01 4b 46 7f ff 0c 10 3b : crc=3b YES
c4 01 4b 46 7f ff 0c 10 3b t=28250
c3 01 4b 46 7f ff 0d 10 2f : crc=2f YES
c3 01 4b 46 7f ff 0d 10 2f t=28187
```

図 2.51 1-Wire 接続温度センサドライバーの使用例

/sys/devices/w1_bus_master1/が、1-Wire に関連する sysfs ディレクトリです。「Thermal family implementation」が有効になっていて、スレーブデバイスが検出されると、28-0000022e2355 のようにデバイスの ROM ID に対応したディレクトリが作成されます。その中の w1_slave を読み出すと、ドライバーは CONVERT T コマンドを実行したあと READ SCRATCHPAD コマンドを実行し、DS18B20 の内蔵レジスタを表示します。「crc=*xx* YES」で CRC が一致したことを表します。また、「t=28250」は温度(摂氏)を 1000 倍した値を示します。1 度の読み出しで、2 回分の変換結果を表示します。

2.4.5. カーネルコンフィギュレーション

Armadillo-400 シリーズの標準のカーネルでは、1-Wire ドライバーは有効になっていません。そのため、1-Wire マスタドライバーと「Thermal familyimplementation」ドライバーが有効になったカーネルを作成する必要があります。

まず、カーネルのソースコードアーカイブを取得します。ここでは、Armadillo サイトからダウンロードしてくることにします。

```
[ATDE ~]$ wget http://armadillo.atmark-techno.com/files/downloads/armadillo-4x0/source/kernel/
linux-3.14-at[version].tar.gz
[ATDE ~]$ tar xzvf linux-3.14-at[version].tar.gz
```

図 2.52 Linux カーネルの取得と展開

次に Armadillo-400 シリーズの標準コンフィギュレーションを適用します。

```
[ATDE ~]$ cd linux-3.14-at[version]/
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm armadillo4x0_defconfig
```

図 2.53 Linux カーネルに Armadillo-400 シリーズ標準コンフィギュレーションを適用する

続いて、menuconfig を使用して、「図 2.55. 1-Wire ドライバーを有効にする」に示すようにカーネルコンフィギュレーションを変更します。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm menuconfig
```

図 2.54 menuconfig を使用してカーネルコンフィギュレーションを変更する

```
Linux Kernel Configuration
  System Type  --->
    Freescale i.MX support  --->
      Armadillo-400 Board options  --->
        [*] Enable one wire at CON9_2  ←チェックを入れる

  Device Drivers  --->
    <*> Dallas's 1-wire support  --->  ←チェックを入れる
      1-wire Bus Masters  --->
        <*> Freescale MXC 1-wire busmaster  ←チェックを入れる
      1-wire Slaves  --->
        <*> Thermal family implementation  ←チェックを入れる
```

図 2.55 1-Wire ドライバーを有効にする

コンフィギュレーションの変更をおこなったら、カーネルをビルドします。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- && gzip -c arch/ ⏎
arm/boot/Image > linux.bin.gz
```

図 2.56 Linux カーネルをビルドする

正常にビルドが完了すると、linux-3.14-at*[version]*/linux.bin.gz にカーネルイメージが作成されます。linux.bin.gz を Armadillo のフラッシュメモリのカーネル領域に書き込んでください。

「図 2.50. 1-Wire 接続温度センサ回路図」に示すように Armadillo と DS18B20 を接続してから Armadillo を起動し、「図 2.51. 1-Wire 接続温度センサドライバーの使用例」に示す手順で動作確認をおこなってください。

2.5. CAN

Armadillo-400 シリーズでは、CON14 を CAN バスとして使用することができます。ここでは、Armadillo 同士を CAN で接続する方法を紹介します。

使用するソフトウェア、デバイスは以下のとおりです。

1. Linux カーネル: linux-3.14-at4 以降
2. ユーザーランド: Atmark Dist v20151026 以降
3. ネットワークおよびトラッフィック制御ツール: iproute2 (Atmark Dist に含まれるもの)
4. CAN 通信プログラム: can-utils (Atmark Dist に含まれるもの)
5. CAN トランシーバー: AMIS-42673(ON Semiconductor 社製)

2.5.1. CAN 概要

CAN(Controller Area Network)は、機器間のデータ転送に使われる、2 線差動電圧式の通信方式です。差動電圧式を採用しているため耐ノイズ性に優れる点や、エラー検出方法と検出後の動作が明確化されているといった特長から、比較的信頼性の求められるネットワークに用いられます。

CAN では、CAN+と CAN-の 2 本の信号線間の電圧差を変化させることで通信をおこないます。この 2 本の信号線に複数のノード(機器)を接続し、バスを構成します。CAN の物理的な仕様に関連する規格には、通信速度が 125kbps までの低速 CAN(ISO1159-2)、通信速度 125kbps から 1Mbps の高速 CAN(ISO11898-2)など様々なものがあります。一般的に CAN のノードは、物理層の処理をおこなう CAN トランシーバとその後のデータ処理をおこなう CAN コントローラから構成されます。今回の例では、CAN コントローラは i.MX25 内蔵の FlexCAN コントローラを使用し、CAN トランシーバには ISO 11898-2 に対応した AMIS-42673 を使用します。

CAN プロトコルでは、CAN+と CAN-間の電圧差を、RS-232C 通信のようにあらかじめ決められた通信速度(ビットレート)に従って変化させることで、データの転送をおこないます。転送される各ビットは、ドミナントかリセッシブのいずれかの状態を取ります。高速 CAN では、CAN+と CAN-の電圧差がある場合ドミナント、無い場合リセッシブとなります。通常、ドミナントを論理 0、リセッシブを論理 1 として扱います。CAN はマルチマスタ構成のため、複数のデバイスが同時に通信をおこない、バス上でデータの衝突がおこる場合があります。この場合、どれか一つのノードがドミナントを出力していた場合、バスの状態はドミナントとなります(ドミナントがリセッシブに対して優先される)。CAN では、この特性を利用して調停をおこないます。

データの転送は、フレームという単位でおこないます。フレームには、「表 2.7. CAN プロトコルフレーム」に示す 4 つの種類があります。

表 2.7 CAN プロトコルフレーム

名称	機能
データフレーム	データを送信する。
リモートフレーム	データフレームを要求する。
オーバーロードフレーム	前回のフレーム処理が完了していないことを通知する。
エラーフレーム	エラーが発生したことを通知する。

データフレームとリモートフレームを合わせて、メッセージフレームといいます。CAN では、ノードごとのアドレスというものはなく、その代わりにそれぞれのメッセージが固有な ID(識別子、Identifier)を持っています。受信ノードは、ID によって、自分が処理すべきメッセージかどうか判断します。メッセージに含まれる ID の長さによって、メッセージフレームには標準フォーマット(11bit 長)と拡張フォーマット(29bit 長)の 2 種類の形式があります。

データフレームの形式を「図 2.57. CAN プロトコル(データフレーム)」に示します。上の線はリセッシブを、下の線はドミナントを意味します。データフレームは、データを送信するノードがバスをドミナントにすることから始まります。これをスタート・オブ・フレーム(SOF)と呼びます。SOF に続き、アービトレーションフィールド(ARBI)、コントロールフィールド(CONT)、データフィールド(DATA)、CRC フィールド(CRC)が順に送信されます。続いて、受信ノードは ACK フィールド(ACK)を送信します。最後に、7 ビット分バスをリセッシブに保ちエンド・オブ・フレーム(EOF)とします。

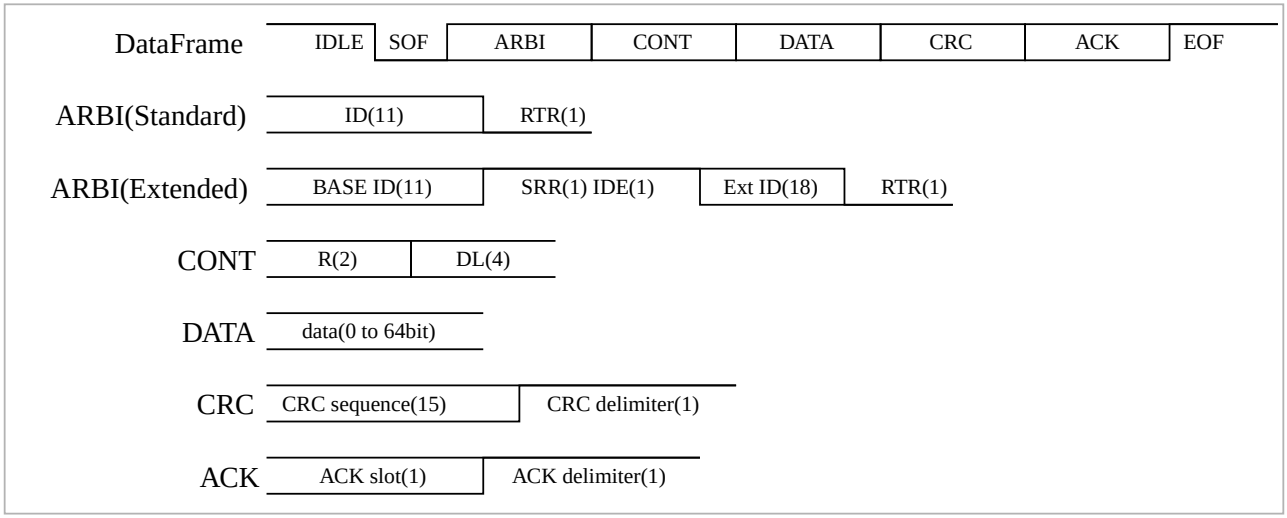


図 2.57 CAN プロトコル(データフレーム)

アービトレーションフィールドは、標準フォーマットか拡張フォーマットかによって異なります。標準フォーマットの場合、11bit の ID を送信したあと、リモート・トランスミッション・リクエスト・ビット(RTR)にドミナントを送信します。拡張フォーマットの場合、11bit のベース ID(BASE ID)を送信したあと、代替リモート・リクエスト・ビット(SRR)、アイデンティファイヤ・エクステンション・ビット(IDE)として、2bit 分バスをリセッシブに保ちます。続いて、18bit の拡張 ID(Ext ID)を送信したあと、RTR にドミナントを送信します。

CAN プロトコルのバリエーション

CAN プロトコルには、バージョン 2.0A と 2.0B の 2 つがあります。プロトコルバージョン 2.0A では、標準フォーマットしか扱うことができません。もし拡張フォーマットのフレームを受信した場合、エラーフレームを送信します。プロトコルバージョン 2.0B パッシブでは、拡張フォーマットの送信はできませんが、拡張フォーマットを受信しても、無視します。プロトコルバージョン 2.0B アクティブでは、拡張フォーマットの送受信が可能です。

Armadillo-400 シリーズは、プロトコルバージョン 2.0B アクティブに対応しています。

コントロールフィールドは、最初の 2 ビットが予約ビットとなっており、常にドミナントとします。続く 4bit のデータ長コード(DLC)に送信するデータのバイト数を送信します。そのため、データフィールドは 0 から 8 バイト(64bit)長となります。

CRC フィールドの CRC シーケンス(CRC sequence)には、SOF からデータフィールドまでの CRC(Cyclic Redundancy Check)を送信します。CRC フィールドの区切りを示す CRC デリミタとして、1bit 分リセッシブとします。

受信ノードは、受信したメッセージの CRC が一致した場合、ACK スロット(ACK slot)でドミナントを送信します。ACK スロットでバスがドミナントとなることで、送信ノードは少なくとも一つの受信ノードがデータフィールドを正常に受信できたことを確認できます。ACK スロットに続いて、ACK フィールドの区切りを示す ACK デリミタ(ACK delimiter)として、1bit 分リセッシブとします。

リモートフレームは、データフレームの要求に使用されます。リモートフレームを受信したノードは、リモートフレームで指定された ID と同じ ID のメッセージを返信します。リモートフレームの形式を、「図 2.58. CAN プロトコル(リモートフレーム)」に示します。RTR をリセッシブにして、データフィールドが無い以外、データフレームと同じです。DLC は、リモートフレームへの返信として帰ってくるデータフレームのデータ長と同一にします。

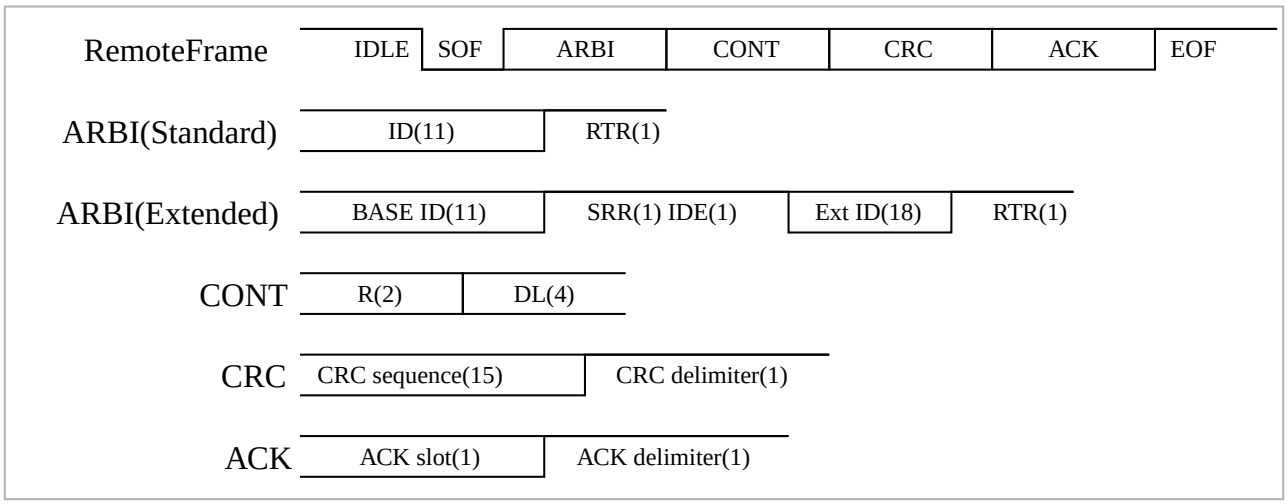


図 2.58 CAN プロトコル(リモートフレーム)

メッセージフレームのアービトレーションフィールドという名前は、このフィールド送信中にバスの調停をおこなうことに由来します。同時に複数のノードがメッセージの送信を開始した場合、バスの衝突が発生します。送信ノードは、各ビットでバスの状態を確認し、もし自身がリセッシブを送信したにも関わらず、バスがドミナントとなっていた場合、以後の送信を中止します。そのため、より小さな ID が優先して送信されます。また、RTR により、リモートフレームよりもデータフレームが優先されます。

オーバーロードフレームとエラーフレームは、一般に CAN コントローラによって自動で処理されます。そのため、ここでは説明を割愛します。

同期とビット・スタッフィング・ルール

CAN では、ビットレートに従ってデータの送受信をおこなうため、ノードごとのクロックに誤差がある場合、タイミングが少しずつずれていきます。これを補正するため、バスがリセッシブからドミナントへ変化するとき、タイミングの同期をおこないます。

しかし、リセッシブやドミナントだけが続いた場合、この同期が行われないことになります。そこで、ビット・スタッフィング・ルールが適用されます。これは、同じ状態が 5bit 連続した場合、反対の状態のビット(スタッフビット)を一つ送信するルールです。このルールにより、一定期間内に必ず同期が行われることを保証しています。

なお、ビット・スタッフィング・ルールの処理は CAN コントローラで自動で行われるため、ユーザー側は通常それを意識することはありません。

2.5.2. サンプル回路

Armadillo-400 シリーズと CAN トランシーバーとを接続する回路図を、「図 2.59. CAN 接続回路図」に示します。Armadillo の CON14 から出ている CAN2 を使用します。CAN トランシーバーには、AMIS-42673(ON Semiconductor 社製)を使用します。LM2731YMF(National Semiconductor 社製)は、3.3V から 5V を生成するスイッチングコンバーターです。CON9 2 ピン(GPIO3_17)で出力の ON/OFF を切り替えることができます。

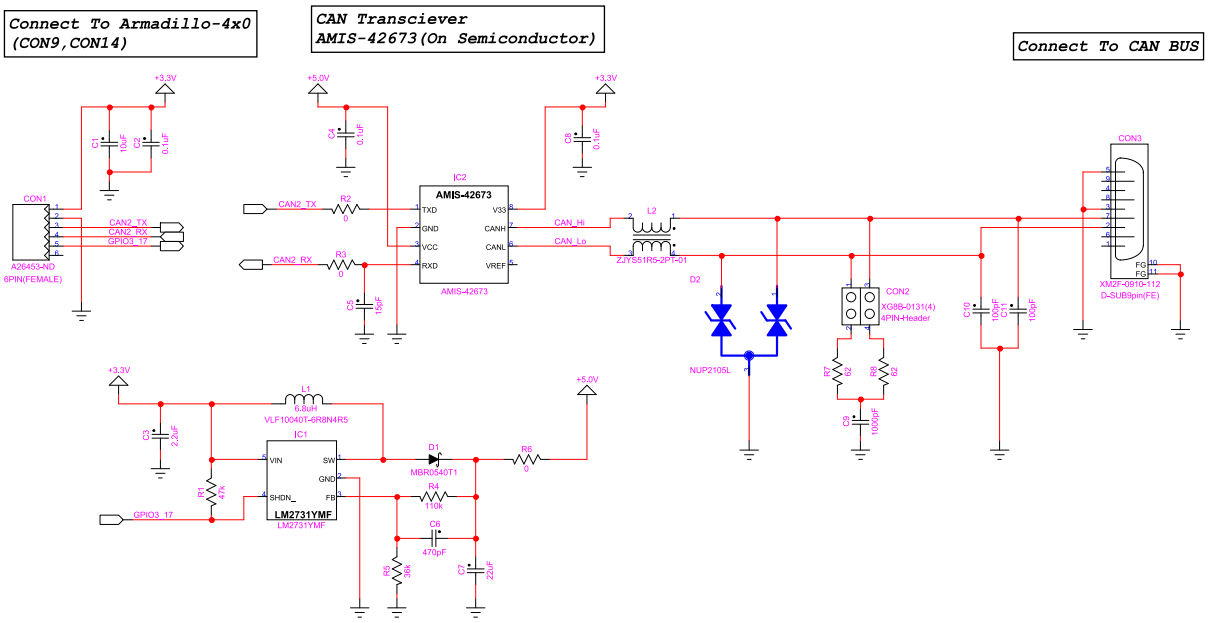


図 2.59 CAN 接続回路図

Armadillo-400 シリーズ同士を接続する場合は、次のように接続してください。3 つ以上の Armadillo を接続しても構いません。

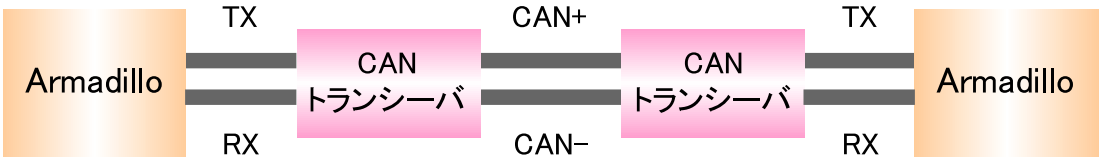


図 2.60 CAN バスを介した Armadillo 同士の接続

2.5.3. CAN ドライバー

Armadillo-400 シリーズの CAN 機能は、SocketCAN フレームワークを使用して実装されています。SocketCAN では、通常のネットワークデバイスと同様に、socket インターフェースを用いてデータの送受信を行います。

CAN 通信をおこなうプログラムは、TCP/IP などを用いたネットワークプログラムと同様に記述できます。「図 2.61. CAN ソケットのオープン」に、CAN 通信用のソケットをオープンし、can0 インターフェースに関連付けるコード例を示します。プロトコルファミリーには、PF_CAN を指定します。プロトコルには、ローソケットプロトコル(CAN_RAW)または、ブロードキャストマネージャ(BCM)を指定します。bind システムコールで CAN インターフェースとソケットを関連付けます。

```
int s;
struct sockaddr_can addr;
```

```
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

図 2.61 CAN ソケットのオープン

以降の処理は、通常のネットワークプログラムと同様です。CAN メッセージの送受信には、read/write システムコールや、send/sendto/sendmsg システムコール、recv/recvfrom/recvmsg システムコールを使用できます。

SocketCAN に関する詳しい情報は、linux-3.14-at*[version]*/Documentation/networking/can.txt を参照してください。

SocketCAN フレームワークでは、sysfs インターフェースを用いて、CAN 通信に関わる設定をおこないます。CAN2 を使用する場合、/sys/devices/platform/FlexCAN.1/以下のファイルを使用します。使用可能な sysfs ファイルの一覧は、「Armadillo-400 シリーズ ソフトウェアマニュアル」の「CAN」を参照してください。

通常の SocketCAN フレームワークには無い、Armadillo-400 シリーズ独自の拡張として、リモートフレームのサポートを追加しています。set_resframe ファイルに、*ID#DATA* という形式で値を書き込むと、対応する ID のリモートフレームワークを受信した場合、自動でデータフレームを返信します。

2.5.4. カーネルコンフィギュレーション

Armadillo-400 シリーズの標準のカーネルでは、CAN ドライバーは有効になっていません。そのため、CAN ドライバーが有効になったカーネルを作成する必要があります。

まず、カーネルのソースコードアーカイブを取得します。ここでは、Armadillo サイトからダウンロードしてくることにします。

```
[ATDE ~]$ wget http://armadillo.atmark-techno.com/files/downloads/armadillo-4x0/source/kernel/
linux-3.14-at[version].tar.gz
[ATDE ~]$ tar xzvf linux-3.14-at[version].tar.gz
```

図 2.62 Linux カーネルの取得と展開

次に Armadillo-400 シリーズの標準コンフィギュレーションを適用します。

```
[ATDE ~]$ cd linux-3.14-at[version]/
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm armadillo4x0_defconfig
```

図 2.63 Linux カーネルに Armadillo-400 シリーズ標準コンフィギュレーションを適用する

続いて、menuconfig を使用して、「図 2.65. CAN ドライバーを有効にする」及び「図 2.66. CAN に使用するピンを指定する」に示すようにカーネルコンフィギュレーションを変更します。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm menuconfig
```

図 2.64 menuconfig を使用してカーネルコンフィギュレーションを変更する

```
Linux Kernel Configuration
  Networking  --->
    <*>   CAN bus subsystem support  --->  ← チェックを入れる
      <*>   Raw CAN Protocol (raw access with CAN-ID filtering)  ← チェックが入っているのでそのまま
      <*>   Broadcast Manager CAN Protocol (with content filtering)  ← チェックが入っているのでそ ⏎
のまま
      CAN Device Drivers  --->
        <*>   Support for Freescale FLEXCAN based chips  ← チェックを入れる
```

図 2.65 CAN ドライバーを有効にする

```
Linux Kernel Configuration
  System Type  --->
    Freescale MXC Implementations  --->
      Armadillo-400 Board options  --->
        [ ] Enable I2C2 at CON14  ← チェックを外す
        [*] Enable CAN2 at CON14  ← チェックを入れる
```

図 2.66 CAN に使用するピンを指定する

コンフィギュレーションの変更と、ソースの修正をおこなったら、カーネルをビルドします。

```
[ATDE ~/linux-3.14-at[version]]$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- && gzip -c arch/ ⏎
arm/boot/Image > linux.bin.gz
```

図 2.67 Linux カーネルをビルドする

正常にビルドが完了すると、linux-3.14-at*[version]*/linux.bin.gz にカーネルイメージが作成されます。linux.bin.gz を Armadillo のフラッシュメモリのカーネル領域に書き込んでください。

2.5.5. ip コマンドの準備

CAN の通信速度設定は **ip** コマンドを使用します。**ip** コマンドは Atmark Dist のユーザーランドコンフィギュレーションで iproute2 を選択する[4]事で組み込む事が可能です。

第 1 部の「Atmark Dist を使ったルートファイルシステムの作成」などを参照し、使用するプロダクト用に基本的な設定をして一度ビルドした Atmark Dist を用意してください。Atmark Dist のユーザーランドコンフィギュレーションで以下の項目にチェックを入れます。

```
Userland Configuration
  Network Applications  --->
```

[4]ユーザーランドには Busybox の ip コマンドが標準で組み込まれています。しかし、BusyBox の ip コマンドは CAN の通信速度をサポートしていません。従って、iproute2 を組み込む必要があります。

```
    [*] iproute2   ← チェックを入れる
    [*]   ip       ← チェックを入れる
```

図 2.68 iproute2 を選択する

2.5.6. CAN 通信プログラムの準備

Atmark Dist には、CAN 通信プログラムのサンプルとして can-utils が含まれています。can-utils には、一つのメッセージを送信する **cansend**、複数のメッセージを連続して送信する **cangen**、受信したメッセージを表示する **candump** があります。今回は、これらを使用して CAN の動作確認をおこなうことにします。

can-utils を使用可能にするには、Atmark Dist のユーザーランドコンフィギュレーションで以下の項目にチェックを入れます。

```
Userland Configuration
  Network Applications  --->
    [*] can-utils  ← チェックを入れる
    [*]   cansend  ← チェックを入れる
    [*]   candump  ← チェックを入れる
    [*]   cangen   ← チェックを入れる
```

図 2.69 can-utils を選択する

これらを選択した状態でユーザーランドをビルドし、作成されたルートファイルシステムイメージ(romfs.img.gz)を Armadillo のフラッシュメモリのユーザーランド領域に書き込んでください。

2.5.7. 使用例

実際に、CAN バスを通じて Armadillo 同士で通信をおこなう手順を説明します。

まず、通信速度を設定します。通信速度は送受信をおこなうノード全てで一致している必要があるので、それぞれの Armadillo でおこなってください。

通信速度は **ip** コマンドで設定します。以下の例では通信速度を 125kbps に設定しています。

```
[armadillo ~]# ip link set can0 type can bitrate 125000 loopback off
```

図 2.70 ip コマンドによる CAN 通信速度の設定例

次に、CAN インターフェースを有効にします。これも、それぞれの Armadillo で実行します。

```
[armadillo ~]# ifconfig can0 up
```

図 2.71 CAN インターフェースの有効化

CAN メッセージを受信する Armadillo で、**candump** を実行しておきます。

```
[armadillo ~]# candump can0
```

図 2.72 CAN メッセージの受信準備

別の Armadillo で **cansend** を実行すると、一つのメッセージを送信できます。「図 2.73. CAN メッセージの送信」の例では、ID=0x5a5、データ=0x01234567 を送信しています。

```
[armadillo ~]# cansend can0 5a5#01234567
```

図 2.73 CAN メッセージの送信

candump コマンドを実行している受信側の Armadillo では、メッセージを受信すると「図 2.74. CAN メッセージの受信結果」に示しすような表示が得られます。

```
[armadillo ~]# candump can0
  can0  5a5  [4] 01 23 45 67
```

図 2.74 CAN メッセージの受信結果

また、**cangen** を実行すると、連続したメッセージを送信できます。オプションに CAN インターフェース名だけを指定した場合、**cangen** はアドレス、データ共にランダムな値を送信します。

```
[armadillo ~]# cangen can0
```

図 2.75 連続した CAN メッセージの送信

candump を実行している受信側の Armadillo では、「図 2.76. 連続した CAN メッセージの受信」に示すような受信結果が得られます。

```
[armadillo ~]# candump can0
  can0  567  [6] 69 98 3C 64 73 48
  can0  451  [8] 4A 94 E8 2A EC 58 55 62
  can0  729  [8] BA 58 1B 3D AB D7 7E 50
  can0  1F2  [8] E3 A9 E2 79 46 E1 45 75
  can0   7C  [2] 54 08
  :
  :
  :
```

図 2.76 連続した CAN メッセージの受信

改訂履歴

バージョン	年月日	改訂内容
3.0.0	2015/10/26	・Linux 3.14 対応のため全面改版 **linux-2.6.26-at に対応した情報は、旧版(v2.x.x)のドキュメントを参照してください。下記 URL からダウンロードすることができます。** http://download.atmark-techno.com/armadillo-420/document/

Armadillo 実践開発ガイド
Version 3.0.0
2015/10/26

株式会社アットマークテクノ

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