



SOFTWARE USER'S GUIDE

Linux デバイスドライバチュートリアル

製品ドキュメントのご紹介

本製品に関する情報を下記の通りご用意しております。

必要に応じて、適切なものをご利用ください。

Windows ユーザーの為の Linux マニュアル

マニュアル		概要
Linux引越しガイド	無償	WindowsからLinuxに移行した際に、違和感なくLinuxを扱えるようにするために、Windowsとの違いを中心に記載しています。
Linux構築ガイド	無償	LinuxでOSを操作したり、プログラムを作成する際の操作について記載しています。
Linuxデバイスドライバチュートリアル(本書)	無償	Linuxで各種I/O制御について記載しています。

無償:弊社Web site (www.interface.co.jp)からのダウンロードは無料です。

有償:ドキュメント類の冊子(印刷物)およびソフトウェアのCD-ROMでの提供は有料販売となります。

目 次

第1章 はじめに	6
1.1 本書について	6
1.2 概要	6
第2章 デバイスドライバの概要	7
2.1 デバイスドライバとは	7
2.2 デバイスドライバの種類	8
2.3 アプリケーションとデバイスドライバとの関係	9
2.4 デバイスドライバの構造	10
2.4.1 モジュールの組み込み	11
2.4.2 モジュールの取り外し	11
2.4.3 モジュールの組み込み	12
2.4.4 モジュールの基本構造	12
2.4.5 デバイスファイル(デバイスノード)	12
2.5 システムコール	13
2.5.1 openシステムコール	14
2.5.2 closeシステムコール	15
2.5.3 readシステムコール	16
2.5.4 writeシステムコール	17
2.5.5 ioctlシステムコール	18
第3章 弊社デバイスドライバ	20
3.0 デバイスドライバの構造	20
3.0 ソフトウェアのインストール	21
3.0 Windowsからの移行	21
第4章 デジタル入出力	22
4.1 デジタル入出力の概要	22
4.2 対象型式	23
4.3 チュートリアル	24
4.3.1 デジタル入力	24
4.3.2 デジタル出力	28
4.3.3 割り込み処理	31
第5章 アナログ入力	37
5.1 アナログの概要	37
5.1.1 A/Dコンバータ	37
5.1.2 A/Dコンバータの用途	38
5.1.3 分解能と精度	38
5.1.4 シングルエンド入力と差動入力	41
5.1.5 バイポーラとユニポーラ	42
5.1.6 変換時間とサンプリング周波数	42
5.1.7 標本化定理	43

5.1.8	マルチADC方式とマルチプレクサ方式	43
5.1.9	データ転送方式	46
5.1.10	アナログ入力製品を使用時の指針	48
5.2	対象型式	49
5.3	環境図	49
5.4	チュートリアル	50
5.4.1	1 件のAD入力を行う	50
5.4.2	サンプリング	55
5.4.3	割り込み処理	62
第 6 章 アナログ出力		68
6.1	アナログの概要	68
6.1.1	D/Aコンバータ	68
6.1.2	アナログ出力	69
6.1.3	分解能と精度	69
6.1.4	バイポーラとユニポーラ	70
6.1.5	セトリング時間とアナログ出力更新レート	71
6.1.6	データ転送方式	72
6.2	対象型式	74
6.3	環境図	74
6.4	チュートリアル	75
6.4.1	1 件出力	75
6.4.2	連続出力	80
6.4.3	割り込み処理	85
第 7 章 シリアル通信		91
7.1	シリアル通信の概要	91
7.1.1	シリアル通信	91
7.1.2	RS-232CとRS-485	92
7.1.3	同期方式	93
7.1.4	調歩同期通信	93
7.1.5	データ長	94
7.1.6	パリティビット	94
7.1.7	通信速度(bps)とスループット	95
7.1.8	フロー制御	95
7.1.9	その他の用語解説	96
7.1.10	調歩同期通信を行うにあたってのアドバイス	97
7.2	対象型式	99
7.3	環境図	99
7.4	チュートリアル	100
7.4.1	送信	100
7.4.2	受信	105
第 8 章 HDLC		110
8.1	HDLCの概要	110
8.1.1	局とコマンド/レスポンス	111

8.1.2	アドレス.....	112
8.1.3	フレームの構成.....	113
8.1.4	データリンク.....	114
8.1.5	モード.....	115
8.2	対象型式.....	116
8.3	環境図.....	116
8.4	チュートリアル.....	117
8.4.1	フレーム送信.....	117
8.4.2	フレーム受信.....	122
8.4.3	割り込み処理.....	127

第9章 CAN 133

9.1	CANの概要.....	133
9.1.1	CANの特徴.....	134
9.1.2	バスレベルとビット.....	135
9.1.3	調停の仕組み(優先順位の決定方法).....	136
9.1.4	フレームの種類.....	138
9.2	対象型式.....	140
9.3	環境図.....	140
9.4	チュートリアル.....	141
9.4.1	メッセージ送信.....	141
9.4.2	メッセージ受信.....	146
9.4.3	割り込み処理.....	151

第10章 カウンタ 157

10.1	カウンタの概要.....	157
10.2	対象型式.....	157
10.3	チュートリアル.....	158
10.3.1	万能カウンタ.....	158
10.3.2	サンプリング.....	164
10.3.3	割り込み処理.....	171

第1章 はじめに

1.1 本書について

本書はLinuxデバイスドライバチュートリアルです。

弊社のIOモジュールをLinuxから制御するためのプログラムの作成方法を説明しています。

1.2 概要

本書は以下のように構成されています。

1. はじめに
2. デバイスドライバの概要
3. 弊社デバイスドライバ
4. デジタル入出力
5. アナログ入力
6. アナログ出力
7. シリアル通信
8. HDLC
9. CAN
10. カウンタ

第2章 デバイスドライバの概要

2.1 デバイスドライバとは

デバイスとは、比較的大きな単位を持った機器、装置のことを意味します。

例えば、CPU やメモリ、グラフィックボードやディスプレイ、マウスやキーボードなどのことです。

弊社の IO モジュールも同じになります。

しかし、例えば「プリンタ」と言っても、市場には様々なメーカー、機種が出回っています。

メーカーや機種が異なれば、機能や操作方法も異なります。

カラー印刷できるプリンタもあれば、両面印刷できるプリンタもあります。

それはつまり、プリンタの機種が異なれば、その数だけ、カーネル側でもコードが必要となるわけです。

しかし、プリンタの機種が異なっても、同じプリンタですから、共通する機能があります。

そのような共通部分をそれぞれのプリンタごとに用意していると無駄です。

そのため、共通化できる部分はカーネルが受け持ち、それ以外の部分は独立させ、交換可能にしておくのが都合が良いです。

その独立した部分がデバイスドライバです。

デバイスドライバの定義はカーネルに包括されます。

つまり、デバイスドライバもカーネルの一部となります。

2.2 デバイスドライバの種類

Linux のデバイスドライバは、対象となるデバイスの種類に合わせて
キャラクタ型、ブロック型、ネットワークの大きく3つに分類されます。
弊社が提供しているデバイスドライバはキャラクタ型デバイスとして制御します。

●キャラクタ型

キャラクタ型とブロック型の違いは、簡単に言えば扱うデータサイズが異なることです。
キャラクタ型デバイスは普通のファイルの様にバイトストリームとして扱うデバイスのことで、テキストコンソールやシリアルポートなどがこれにあたります。

●ブロック型

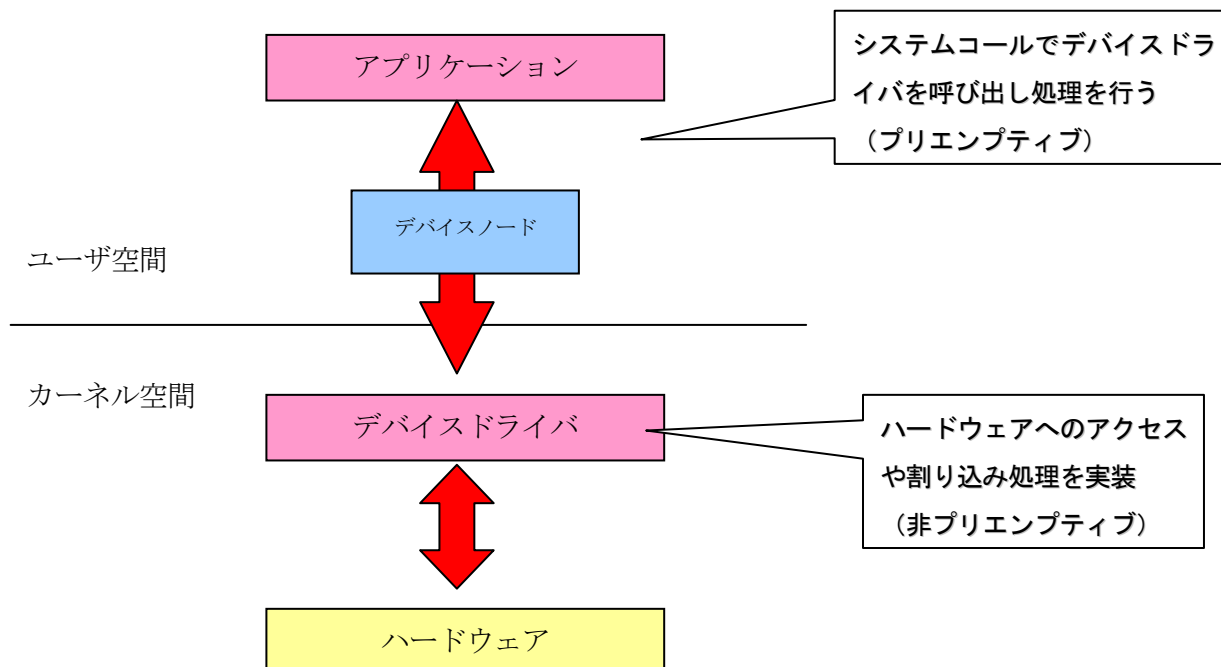
ブロック型デバイスはデータをブロック単位で扱う必要があるデバイスのことです。
(ブロックは 1KB あるいはそれ以外の 2 のべき乗のデータの固まり)
ただし、ブロック型のデバイスでもキャラクタ型のデバイスドライバと同様のインタフェースを持つ為、ブロック単位でもデータを扱えるといった意味となります。
(ブロック型デバイスのインタフェースではファイルシステムをマウント可能です)

●ネットワーク

ネットワークインタフェースは、ネットワーク関連のデバイスに他のホストとの通信を行う為のインタフェースを提供する為のドライバとなります。
ネットワークドライバは通常の上記の 2 つと異なり対応するデバイスノードを持ちません。
固有の名前(eth0)を付けて、パケット送受信の関数を使って呼び出されます。
上記の 2 つのドライバ形式とは大きくことなっています。

2.3 アプリケーションとデバイスドライバとの関係

アプリケーションとデバイスドライバの関係は図の様になります。

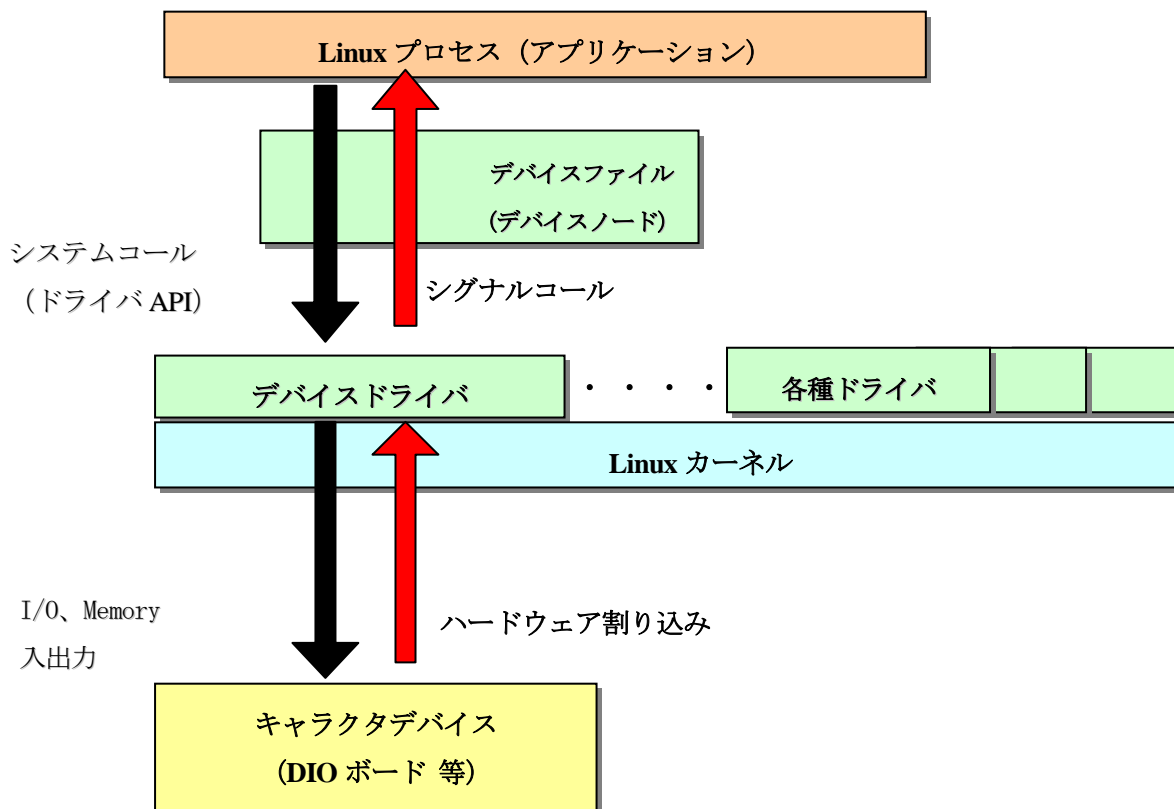


それぞれユーザ空間と、カーネル空間という全く別々のメモリ空間上で動作する為、その特性や使用可能な関数なども大きく異なっている点に注意してください。

ユーザアプリケーションはユーザ空間で実行されるため、アプリケーションの問題でプログラムが暴走しても OS 自体が止まってしまうことはありません、コアダンプを出してプログラムが終了するだけです。ただしハードウェアに自由にアクセスなどができないなどの制限を受けます。

カーネル空間では逆に、ハードウェアに自由にアクセス可能ですが、問題のあるプログラムを動かそうとすると、OS の保護がきかずすぐにフリーズやカーネルパニックなどの致命的な問題を発生しやすいので、注意がひつようとなります。

2.4 デバイスドライバの構造



デバイスドライバは Linux カーネルに動的に組み込まれると、カーネルの一部としてカーネル空間内で動作を行います。

ドライバの呼び出しを行う際にはカーネルの提供するシステムコールを使用して、ドライバに対応づけられたデバイスノードを参照してドライバヘデータ・コマンドの送受信などを行います。

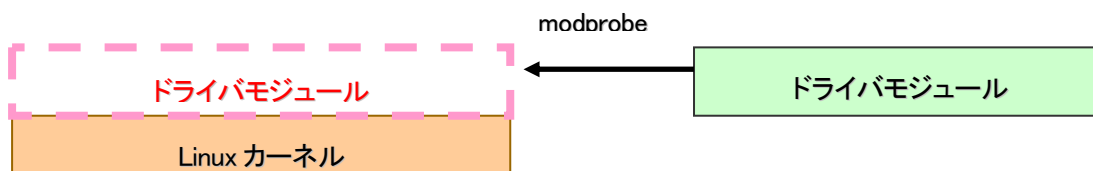
Linux カーネルではシステムコールが呼び出されると、ノードに対応したドライバに実装された処理を実行し、ハードウェアへのアクセスを実行します。

ハードウェアからの割り込みに関しては、ドライバ側で直接ハンドラを実装して処理を実行します。

その後、Linux のプロセス側に伝えるかどうかは、ドライバの設計次第となりますが、カーネル空間からプロセスへの通信にはシグナルが使用されます。

(ドライバ内で割り込み発生時に行う処理が完結する場合にはシグナルなどは必要ありません。)

2.4.1 モジュールの組み込み



※モジュールの組み込み/取り外しには root 権限が必要です。

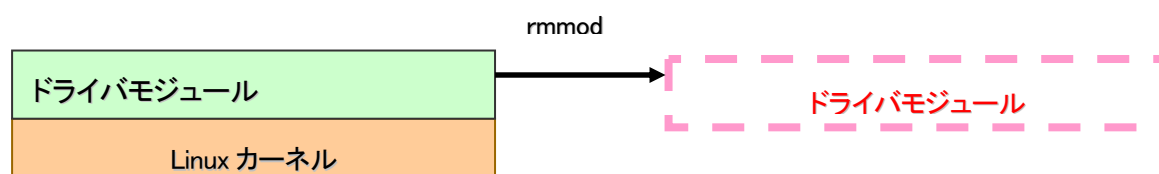
Linux のデバイスドライバはロードブルカーネルモジュールとして作成します。

デバイスドライバは Linux OS 起動後、modprobe コマンドにより動的にモジュールの組み込みを行います。

組み込まれたデバイスドライバは、OS の再起動なしにその時点から使用可能な状態となりますので、使用したいデバイスのドライバを必要になった時点で組み込むことが可能となります。

また、ドライバの作成時にも非常に有効で、修正を行ったドライバをその都度組み込み直すことで、毎回 OS を起動する手間が省けます。

2.4.2 モジュールの取り外し



※モジュールの組み込み/取り外しには root 権限が必要です。

組み込みと同様にモジュールの取り外しも動的に行うことができます。

rmmod コマンドを実行することでカーネル空間に展開されたドライバのデータをアンロードすることが可能です。

2.4.3 モジュールの組み込み

2.4.4 モジュールの基本構造

ドライバのロード・アンロード時には、`init_module/cleanup_module` というハンドラが呼び出されます。
このハンドラ内で、ドライバで使用するリソースの初期化・終了処理を行っています。

2.4.5 デバイスファイル(デバイスノード)

デバイスファイルとは、ハードディスクなどの周辺装置(デバイス)を制御する際に用いられる特別なファイルです。

デバイスファイルはすべて、`/dev` 以下に配置されています。

どのようなデバイスファイルがあるかについては、`/dev` で、`ls` コマンドを実行して確認することができます。

実際に、デバイスファイルは通常のファイルとは異なり、デバイス番号など、ごく少数のデータだけを保持しています。

プログラムからこれらのファイルに対して `open()` などのシステム・コールを用いてアクセスすると、カーネル内に組み込まれたデバイスドライバが呼び出され、ファイルの代わりにデバイスにアクセスします。

2.5 システムコール

アプリケーション(ユーザ空間)からハードウェアを直接操作することはできません。

ハードウェアと直接やりとりができるのはカーネル空間だけになります。

つまり、ソフトウェアはハードウェアを操作したいとき、カーネルに処理を依頼して、間接的に操作する方法になります。

アプリケーション(ユーザ空間)からカーネル空間とのやりとりを行うために「システムコール」を使用します。

システムコールには下記のような種類が用意されています。

- open
- close
- read
- write
- ioctl

上記したのは一部であり、他にもたくさんあります。

ソフトウェアはこのようなシステムコールを介して、ハードウェアを操作することになります。

このデバイスファイルを操作することが、デバイス(ハードウェア)を操作することにつながります。

それはつまり、デバイスファイルがデバイス(ハードウェア)を表しているということです。

2.5.1 openシステムコール

open システムコールについて、説明します。

open システムコールは、指定したデバイスファイルのデバイスのオープンを行い、ファイルディスクリプタを取得します。

open システムコールの書式としては下記になります。

```
#include <fcntl.h>

int open(const char *name, int flags);
```

パラメータについては下記となります。

引数名	内 容
name	オープンを行うデバイスのデバイスファイル名を指定します。
flags	オープン時のオプションを指定します。
	O_RDONLY 読み込み専用でオープンします。
	O_WRONLY 書き込み専用でオープンします。
	O_RDWR 読み書き可能でオープンします。
	O_SYNC ファイルは同期 (synchronous) I/O モードでオープンされる。 write システムコールは、全データの出力が完了するまでドライバからリターンしない。
	O_ASYNC このファイルディスクリプターへの入力または出力が可能になった場合に、シグナルを生成する。 本オプションは一部デバイスのみ有効です。

関数に正常終了した場合は、ファイルディスクリプタが返されます。

エラーの場合は-1 が返され `errno` に以下の値が格納されます。

-EFAULT	name が不正なポインタです
-EACCES	オープンする権限がありません。
-ENODEV	デバイスが存在しません。

ここでは、例としてシリアルポートのオープンを行います。

```
int fd;

fd = open("/dev/ttyS64", O_RDWR);
```

2.5.2 closeシステムコール

close システムコールについて説明します。

close システムコールは、open システムで取得したファイルディスクリプタのデバイスのクローズを行います。

close システムコールの書式としては下記になります。

```
#include <unistd.h>

int close(int fd);
```

パラメータについては下記となります。

引数名	内 容
fd	クローズを行うデバイスのファイルディスクリプタを指定します。

関数に正常終了した場合は、0 が返されます。

エラーの場合は-1 が返され errno に以下の値が格納されます。

-EBADF	fd が不正なファイルディスクリプタです
-EINTR	シグナルにより処理が中断されました

ここでは、例としてシリアルポートのクローズを行います。

```
int fd;

fd = open("/dev/ttyS64", O_RDWR);

close(fd);
```

2.5.3 readシステムコール

read システムコールについて説明します。

read システムコールは、open システムで取得したファイルディスクリプタのデバイスからデータを読み込みます。

read システムコールの書式としては下記になります。

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t length);
```

パラメータについては下記となります。

引数名	内 容
fd	読み込みを行うデバイスのファイルディスクリプタを指定します。
buffer	読み込んだデータを格納する変数へのポインタを指定します。
length	読み込むデータサイズをバイト単位で指定します。

関数に正常終了した場合は、読み込んだバイト数が返されます。

エラーの場合は-1 が返され `errno` に以下の値が格納されます。

-EBADF	fd が不正なファイルディスクリプタです
-EINTR	シグナルにより処理が中断されました
-EINVAL	fd は読み込みできないデバイスです
-EFAULT	buffer が不正なポインタです

ここでは、例としてシリアルポートから 256 バイト受信します。

```
int fd;
ssize_t length;
char buffer[256];

fd = open("/dev/ttyS64", O_RDWR);

length = read(fd, buffer, 256);
if(length > 0) {
    printf("read %ld bytes:%X %X...%n", length, buffer[0], buffer[1]);
}

close(fd);
```


2.5.4 writeシステムコール

write システムコールについて説明します。

write システムコールは、open システムで取得したファイルディスクリプタのデバイスにデータを書き込みます。

write システムコールの書式としては下記になります。

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t length);
```

パラメータについては下記となります。

引数名	内 容
fd	書き込みを行うデバイスのファイルディスクリプタを指定します。
buffer	書き込むデータを格納する変数へのポインタを指定します。
length	書き込むデータサイズをバイト単位で指定します。

関数に正常終了した場合は、読み込んだバイト数が返されます。

エラーの場合は-1 が返され errno に以下の値が格納されます。

-EBADF	fd が不正なファイルディスクリプタです
-EINTR	シグナルにより処理が中断されました
-EINVAL	fd は書き込みできないデバイスです
-EFAULT	buffer が不正なポインタです

ここでは、例としてシリアルポートからデータを送信します。

```
int fd;
ssize_t length;
char buffer[256];

fd = open("/dev/ttyS64", O_RDWR);

strcpy(buffer, "The quick brown fox the jumps the lazy dog.");
length = write(fd, buffer, strlen(buffer));

close(fd);
```

2.5.5 ioctlシステムコール

ioctl システムコールについて説明します。

ioctl システムコールは、open システムで取得したファイルディスクリプタのデバイスを制御します。

ioctl システムコールの書式としては下記になります。

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, char *argp);
```

パラメータについては下記となります。

引数名	内 容
fd	デバイス制御を行うデバイスのファイルディスクリプタを指定します。
request	デバイスに送信するコマンドを指定します指定します。
argp	デバイスに送信する要素を指定します。(データ or 構造体ポインタ 等) ※ ヘッダファイル内では引数のコンパイル時のエラーを回避する目的で、「...」と定義されています。

関数に正常終了した場合は 0 が返されます。

(いくつかの ioctl では出力パラメータとして戻り値を使用していたり、成功した場合に非 0 の値を返したりする可能性があります。)

エラーの場合は -1 が返され、errno に下記の値が格納されます。

-EBADF	fd が不正なファイルディスクリプタです
-EFAULT	argp がアクセス不可能なメモリを参照しています
-ENOTTY	fd がキャラクタ型のスペシャル・デバイスを参照していません
-ENOTTY	指定されたリクエストはディスクリプタ fd が参照する種類のオブジェクトには適用することができません
-EINVAL	cmd または argp が不正です

ここでは、例として標準入力の ECHO をオフにして、キー入力の画面表示を隠します。

```
int fd;
int ret;
struct termio tty;
struct termio tty_org;
char buffer[256];

ioctl(0, TCGETA, &tty);
tty_org = tty;
tty.c_lflag &= ~ECHO;
tty.c_lflag |= ECHONL;
ioctl(0, TCSETA, &tty);

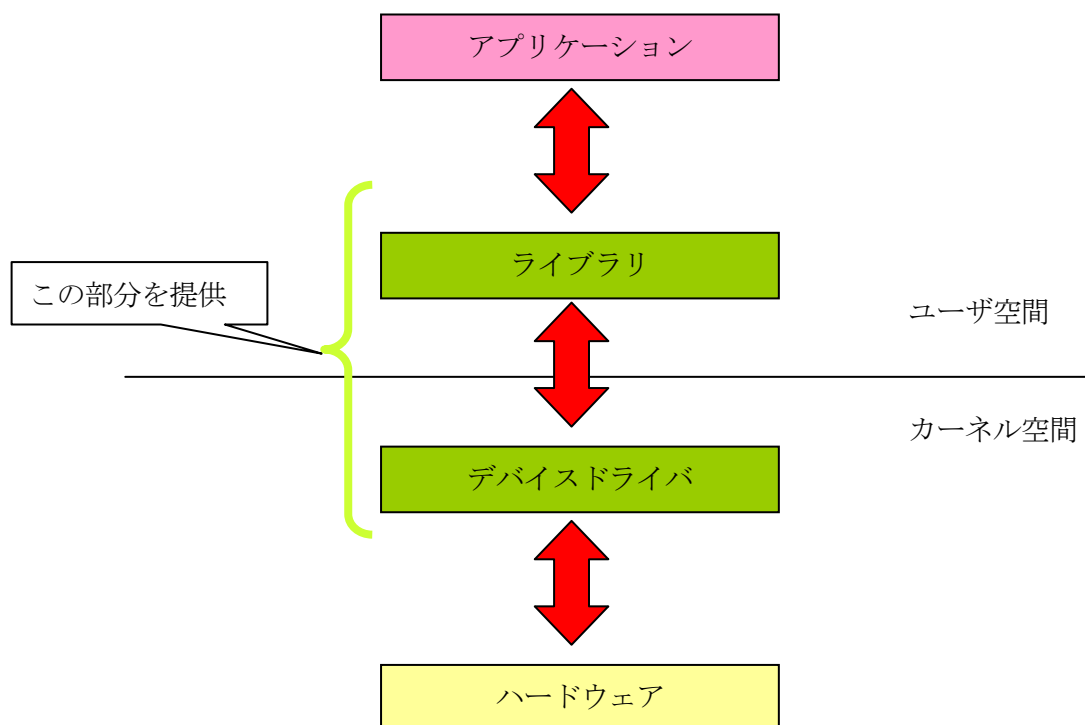
printf("password ? ");
```

```
fgets(buffer, 256, stdin);  
printf("password = %s", buffer);  
  
ioctl(0, TCSETA, &tty_org);
```

第3章 弊社デバイスドライバ

3.1 デバイスドライバの構造

弊社のデバイスドライバの構造を説明します。



ユーザ空間とカーネル空間でドライバとアプリケーションが分かれているのは、先ほど説明したものと同じなのですが、ユーザ空間の方が、最終的なアプリケーションとライブラリに分かれています。

これは、システムコールを実行してドライバを呼び出す処理等を使いやすくする為にAPIを作成してライブラリでラップした形式となっています。

3.2 ソフトウェアのインストール

弊社 IO モデルのプリインストールモデルを購入された場合、出荷時の状態で必要なソフトウェアは全てインストールされています。

よって、ソフトウェアのインストールなどは必要ありません。

3.3 Windowsからの移行

弊社の Linux ドライバについては、Windows ドライバの API 体系とほぼ互換で設計されているため IO 制御の部分については、簡単に移行させることができます。

Windows ドライバとの違いとしては、下記があります。

- ・デバイスを初期化する Open 関数のパラメータが変更
- ・イベント機能が Linux ドライバにはありません。
(コールバック機能で置き換えが必要)

第4章 デジタル入出力

4.1 デジタル入出力の概要

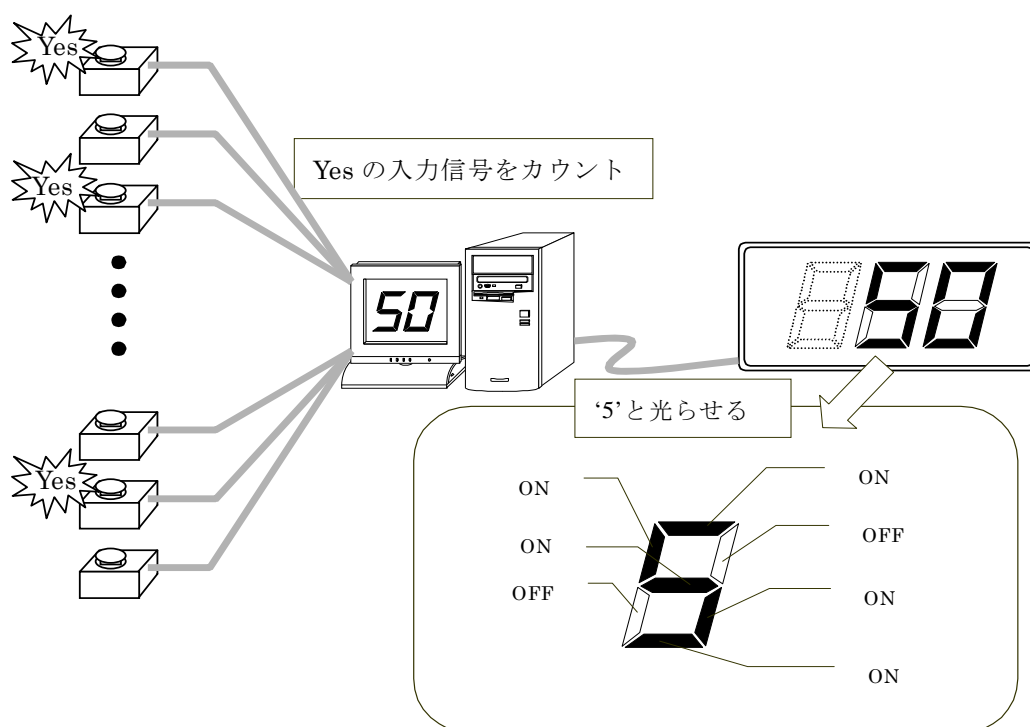
コンピュータはデジタル信号で動作しています。コンピュータに入力される情報、コンピュータから出力する情報は全てデジタルで処理されます。

また、世の中には様々なスイッチ、センサといったものが数多くありますが、電氣的にON/OFFでその状態を示すことが可能な情報は、コンピュータに入力することが可能です。これとは逆に、その状態をコンピュータから作り出す(出力する)ことも可能です。

身近な例でいえば、よくテレビ番組で「お手元のスイッチでYESかNOかお答えください」という司会者の問いかけに観客が答え、その集計結果が電光パネルに表示されるといったものをご覧になったことがあると思います。

これはスイッチの情報、つまりは「YES」か「NO」かといったデジタル情報がコンピュータに入力され、その集計結果の情報が電光パネルに出力され表示されるといった、デジタル入出力システムの一例です。

集計結果を電光パネルに表示する際、コンピュータは電光パネルに対し「特定の箇所を光らせる」といったことを行っています。言い換えれば、発光部品への「ON」、「OFF」のデジタル情報を電光パネルに送っているわけです。



4.2 対象型式

ここでは、実際に弊社製品を用いて、簡単なプログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N3620	タフコン CD シリーズ複合モデル
-----------	-------------------

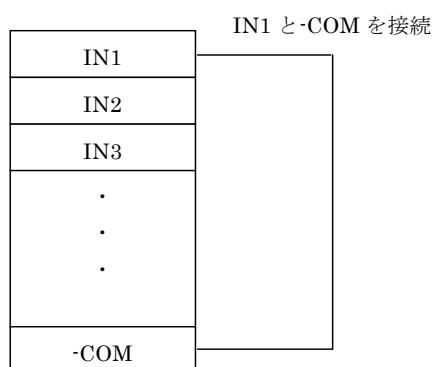
4.3 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

4.3.1 デジタル入力

環境図としては、下記になります。



Step1 プログラム作成

デジタル入力を行うには、DioInputByte / DioInputWord / DioInputDword 関数を使用します。
それぞれ、指定した 8 点を Byte 単位 / 16 点を WORD 単位 / 32 点を DWORD 単位で読み込みます。
エディタで下記コードを入力し「inputbyte.c」という名前で保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbidio.h"

int main(void)
{
    int          dnum, nRet;
    unsigned char bValue;

    dnum = 1;
    nRet = DioOpen(dnum, FBIDIO_FLAG_SHARE);
    if (nRet) {
        printf("Error: ret=%Xh¥n", nRet);
        return -1;
    }

    // バイト単位で1件の入力を実行
    nRet= DioInputByte( dnum, FBIDIO_IN1_8, &bValue);
    if (nRet) {
        printf("Error: ret=%Xh¥n", nRet);
    } else {
        printf("InputData: %Xh¥n", bValue);
    }

    DioClose(dnum);

    return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: inputbyte

inputbyte: inputbyte.o
    $(CC) inputbyte.o -o inputbyte -lgpg2000 -lpthread

inputbyte.o: inputbyte.c
    $(CC) -Wall -c inputbyte.c -o inputbyte.o

clean:
    rm -f *.o ¥##* *~ inputbyte
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「inputbyte.o」と実行ファイル「inputbyte」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./inputbyte
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
InputData: 1h
```

Step3 プログラムの解説

DIO製品を制御する前段階として、DIO製品をオープンするDioOpen関数を呼び出しています。
以下に引数と対応を示します。

```
Int DioOpen(  
    int          nDevice,          /* デバイス番号 */  
    unsigned long ulFlag           /* オープンフラグ */  
);
```

引数名	内 容
NDevice	オープンするデバイス番号を指定します。
UIFlag	0 を指定すると、デバイスを排他オープンします。 FBIDIO_FLAG_SHARE を指定した場合は、同じデバイスを複数のプロセスから重複(共有)してオープンすることを許可します。

これらの引数を与えて関数を呼び出すと、ドライバは引数に適合するDIO製品を検索し、合致した製品がある場合は、戻り値として0を返します。

本プログラムでは、共有オープンを有効にしてデバイスをオープンしています。
プログラマは、このデバイス番号を使って以降のAPIの呼び出しを行います。

デバイスを1件単位で読み込むためには、DioInputByte / DioInputWord / DioInputDword 関数を使用します。8点/ 16点 / 32点単位で読み込みます。

ここでは、DioInputByte関数を用い、IN1～IN8の入力を行っています。

関数を呼び出すと、第二引数で指定された入力ピンの入力を行い、第3引数で渡された変数のポインタに結果を返します。

本プログラムでは、printf関数を使用し、IN1～IN8の入力データを16進で表示しています。

最後にDIO製品の制御を終了するためには、DioClose関数を用います。
オープンしたDIO製品は、使用後は必ずクローズしてください。

★DioClose関数を呼ばないと、どうなるか？

DioClose関数を呼び出さないと、DIO製品はオープンしたままの状態になります。
オープン状態なので、DioOpen関数を呼び出した場合、エラーが返ります。
オープンしたら必ずクローズしてください。

4.3.2 デジタル出力

環境図としては、下記になります。

IN1-8 と OUT1-8 をそれぞれ接続

IN1		OUT1
IN2		OUT2
IN3		OUT3
IN4		OUT4
IN5		OUT5
IN6		OUT6
IN7		OUT7
IN8		OUT8

Step1 プログラム作成

デジタル出力を行うには、DioOutputByte / DioOutputWord / DioOutputDword 関数を使用します。
それぞれ、指定した 8 点を Byte 単位 / 16 点を WORD 単位 / 32 点を DWORD 単位で出力を制御します。
エディタで下記コードを入力し「outputbyte.c」という名前で保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbidio.h"

int main(void)
{
    int          dnum, ret;
    unsigned char bValue;

    dnum = 1;
    ret = DioOpen( dnum, FBIDIO_FLAG_SHARE);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
        return -1;
    }

    // バイト単位で1件の出力を実行。出力データは55h
    bValue = 0x55;
    ret = DioOutputByte( dnum, FBIDIO_OUT1_8, bValue);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
    } else {
        printf("OutputData: %Xh¥n", bValue);
    }
}
```

```
}  
  
DioClose(dnum);  
  
return 0;  
}
```

MakeFile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include  
LIB = /usr/X11R6/lib  
CC = gcc  
  
all: outputbyte  
  
outputbyte: outputbyte.o  
$(CC) outputbyte.o -o outputbyte -lgpg2000 -lpthread  
  
outputbyte.o: outputbyte.c  
$(CC) -Wall -c outputbyte.c -o outputbyte.o  
  
clean:  
rm -f *.o ¥#* *~ outputbyte
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「outputbyte.o」と実行ファイル「outputbyte」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./outputbyte
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
OutputData:55h
```

ここで、デジタル入力を実行することで、以下の出力を得る事が出来ます。

```
InputData:55h
```

Step3 プログラムの解説

DIO製品を制御する前段階として、DIO製品をオープンするDioOpen関数を呼び出しています。

以下に引数と対応を示します。

```
Int DioOpen(  
    int          nDevice,          /* デバイス番号 */  
    unsigned long ulFlag           /* オープンフラグ */  
);
```

引数名	内 容
NDevice	オープンするデバイス番号を指定します。
UIFlag	0を指定すると、デバイスを排他オープンします。 FBIDIO_FLAG_SHARE を指定した場合は、同じデバイスを複数のプロセスから重複(共有)してオープンすることを許可します。

これらの引数を与えて関数を呼び出すと、ドライバは引数に適合するDIO製品を検索し、合致した製品がある場合は、戻り値として0を返します。

本プログラムでは、共有オープンを有効にしてデバイスをオープンしています。

プログラムは、このデバイス番号を使って以降のAPIの呼び出しを行います。

デバイスを1件単位で出力するためには、DioOutputByte / DioOutputWord / DioOutputDword 関数を使用します。8点 / 16点 / 32点単位で出力を行ないます。

ここでは、DioOutputByte関数を用い、OUT1～OUT8の出力制御を行っています。

関数を呼び出すと、第二引数で指定された出力ピンの出力を行い、第3引数で渡された変数に格納された出力値を出力します。

本プログラムでは、printf関数を使用し、OUT1～OUT8の出力結果をを16進で表示しています。

最後にDIO製品の制御を終了するためには、DioClose関数を用います。

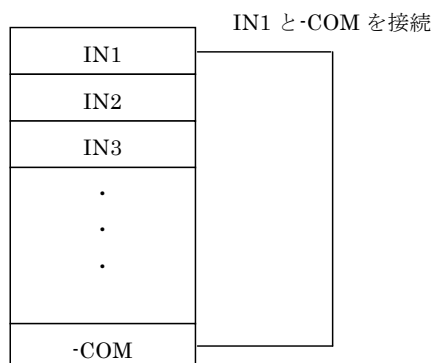
オープンしたDIO製品は、使用後は必ずクローズしてください。

★DioClose関数を呼ばないと、どうなるか？

DioClose関数を呼び出さないと、DIO製品はオープンしたままの状態になります。
オープン状態なので、DioOpen関数を呼び出した場合、エラーが返ります。
オープンしたら必ずクローズしてください。

4.3.3 割り込み処理

環境図としては、下記になります。



Step1 プログラム作成

デジタル入力割り込みを行うには、DioSetIrqConfig 関数で割り込み要因を設定し、DioSetIrqMask 関数で割り込みマスクを解除します。

下記のコードでは、割り込み要因を IN1-4 の立下りエッジに設定し、外部から立下りエッジを入力することで割り込みを発生させます。

発生した割り込みを確認するため、DioRegistIsr 関数でコールバック関数、ユーザデータの登録を行い、割り込み発生時に割り込み要因、ユーザデータ、デバイス番号を表示しています。

エディタで下記コードを入力し「eventcheck.c」という名前で保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "fbidio.h"

int gnflag = 0;

// コールバック関数
void CallBackProc (unsigned long IUserData, unsigned char bEvent, unsigned short nDeviceNum)
{
    printf("Call CallBackProc¥n");
    printf("bEvent[%x] IUserData[%lx] nDeviceNum[%x]¥n", bEvent, IUserData, nDeviceNum);

    // Display the data
    gnflag = 1;
}
```

```
    return;
}

int main(void)
{
    int dnum, ret;
    dnum = 1;

    // デバイスのオープン
    ret = DioOpen(dnum, FBIDIO_FLAG_SHARE);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
        return -1;
    }

    // コールバック関数と、ユーザデータの登録
    ret = DioRegistIsr(dnum, 0x12345678, CallBackProc );
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
        DioClose(dnum);
        return -1;
    }

    // 割り込み要因の割り当て、割り込み発生論理
    // IN1, IN2, IN3 IN4の立下りエッジを指定
    ret = DioSetIrqConfig(dnum, 0xF0);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
        DioClose(dnum);
        return -1;
    }

    // IN1, IN2, IN3 IN4の割り込みをアンマスク
    ret = DioSetIrqMask(dnum, 0x0F);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
        DioClose(dnum);
        return -1;
    }
    printf("Wait for interrupt from IN1, IN2, IN3, IN4¥n");

    // 割り込み発生待ち
    while (!gnflag) {
        usleep(1000);
    }

    // IN1, IN2, IN3 IN4の割り込みをマスク
    ret = DioSetIrqMask(dnum, 0x00);
    if (ret) {
        printf("Error: ret=%Xh¥n", ret);
    }
}
```



```
DioClose(dnum);
return -1;
}

// コールバック関数の登録を解除
ret = DioRegistIsr(dnum, 0x00000000, NULL );
if (ret) {
    printf("Error: ret=%Xh¥n", ret);
    DioClose(dnum);
    return -1;
}

// デバイスのクローズ
DioClose(dnum);

return 0;
}
```

MakeFile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all:eventcheck

eventcheck: eventcheck.o
$(CC) eventcheck.o -o eventcheck -lgpg2000 -lpthread

eventcheck.o: eventcheck.c
$(CC) -Wall -c eventcheck.c -o eventcheck.o

clean:
rm -f *.o ¥#* *~ eventcheck
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「eventcheck.o」と実行ファイル「eventcheck」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./eventcheck
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
Wait for interrupt from IN1, IN2, IN3, IN4
```

この時、IN1 と-COM の接続を外すことで、割り込みが発生し、以下の出力が行なわれます。

```
bEvent[1] IUserData[12345678] nDeviceNum[1]
```

Step3 プログラムの解説

DIO製品を制御する前段階として、DIO製品をオープンするDioOpen関数を呼び出しています。
以下に引数と対応を示します。

```
int DioOpen(  
    int          nDevice,          /* デバイス番号 */  
    unsigned long ulFlag           /* オープンフラグ */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。
ulFlag	0を指定すると、デバイスを排他オープンします。 FBIDIO_FLAG_SHARE を指定した場合は、同じデバイスを複数のプロセスから重複(共有)してオープンすることを許可します。

これらの引数を与えて関数を呼び出すと、ドライバは引数に適合するDIO製品を検索し、合致した製品がある場合は、戻り値として0を返します。

本プログラムでは、共有オープンを有効にしてデバイスをオープンしています。
プログラムは、このデバイス番号を使って以降のAPIの呼び出しを行います。

コールバック関数の登録と、ユーザデータの登録は、DioRegistIsr関数を使用します。

コールバック関数は、デジタル入出力デバイスで割り込み発生時に呼び出される関数のプレースホルダです。(コールバック関数に戻り値はありません。)

割り込みが発生するたびに、DioRegistIsr関数で登録したコールバック関数が呼び出されます。

ユーザデータは、複数のデバイスで割り込みを登録した場合、どのデバイスに対する割り込みかを判別するときに使用します。

それぞれ、第二引数にユーザデータとして、12345678h、第三引数にコールバック関数CallBackProcを指定して、関数を実行しています。

割り込み要因の割り当て、発生論理を指定するには、DioSetIrqConfig関数を使用します。

ここでは、IN1～IN4の立下りエッジで割り込みが発生するように割り当てを行なっています。

通常、割り込みは全てマスクされ、割り込みが発生しない状態となっています。

割り込みを発生させるためには、DioSetIrqMask関数を使用し、割り込みをアンマスクします。

ここでは、IN1～IN4に対して割り込みをアンマスクしています。

本プログラムでは、whileループを使用して割り込みの発生を待機しています。

whileループの判定条件であるgnflagは、コールバック関数の中で書き換えられるため、割り込みが発生してコールバック関数が呼ばれると、whileループを抜け、次の処理に移るようになっています。

割り込みを発生させるためには、接続したIN1と-COMの接続を外します。

IN1の入力状態がHigh状態から、Low状態となり、立下りのエッジが作られます。この時、本デバイスは立下りエッジを検出して割り込みを発生させます。

割り込みが発生すると、コールバック関数内で、割り込みが発生した要因・ユーザデータ・デバイス番号をprintf関数を使用して表示しています。

次に、gnflagに1をいれ、コールバック関数は処理を返します。

この時、コールバック関数内でgnflagが更新されたため、メインルーチンでは割り込み発生を待機している処理を抜け、終了処理に移ります。

割り込みを使う必要がなくなった場合は、割り込みをマスクし、コールバック関数の登録を解除します。

割り込みをマスクし、コールバック関数の登録を解除するには、再度DioSetIrqMask関数とDioRegistIsr関数を使用します。

DioSetIrqMask関数の第二引数に0を指定することで、全ての割り込みの発生をマスクします。以後、割り込みが発生することはありません。

また、DioRegistIsr関数の第三引数にNULLを指定することで、コールバック関数の登録を解除することが出来ます。

最後にDIO製品の制御を終了するためには、DioClose関数を用います。

オープンしたDIO製品は、使用後は必ずクローズしてください。

★DioClose関数を呼ばないと、どうなるか？

DioClose関数を呼び出さないと、DIO製品はオープンしたままの状態になります。

オープン状態なので、DioOpen関数を呼び出した場合、エラーが返ります。

オープンしたら必ずクローズしてください。

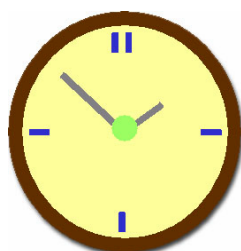
第5章 アナログ入力

5.1 アナログの概要

これまでに、アナログ(analog)という言葉は一度は耳にしたことがあると思います。

ではアナログとは何でしょう。しばしばデジタルという言葉と対比し使われますが、デジタルが「1つ2つと区切って数えられる」のに対し、アナログとは「連続した量で、1つ2つと区切って数えられない」ものを言います。

デジタル時計とアナログ時計を例に挙げてみます。



アナログ時計



デジタル時計

デジタル時計では、「1:53」と単純に読み取ればよいわけですが、アナログ時計で仮に小数点以下を読み取っていくとしたらいかがでしょう。「1:53」、「1:53.1」、「1:53.124」、「1:53.1245.....」と、きりがありません。

5.1.1 A/Dコンバータ

私たちが普段の生活で目や耳にする「光」、「音」の情報は全てアナログ信号です。

今のコンピュータでは音楽や映像の録音や録画ができますが、本来デジタル信号しか処理できないコンピュータが、一体どうやってこのようなアナログ信号を取り扱っているのでしょうか？

実は、ある段階でアナログ信号をデジタル信号に変換し、コンピュータに処理させているのです。これをAD変換といいます。

アナログ信号からデジタル信号に変換する装置を、**A/Dコンバータ**と言います。A/Dコンバータとは、Analog to Digital Converterの略で、アナログ・デジタル変換器と訳され、これによりアナログ信号をデジタル信号に変換することができます。

「光」、「音」、「熱」等の変化は、専用の「センサ」を使用することで、電圧の変化として捉えることができます。この電圧はアナログ信号となり、A/Dコンバータによってデジタル信号に変換され、ハードディスク等の記録媒体に記録されます。

5.1.2 A/Dコンバータの用途

私たちの身の回りにある、全ての事象はアナログ的な情報として成り立っています。しかし、コンピュータは直接アナログ情報を扱うことができないため、A/Dコンバータでデジタル値に変換して、処理を行います。

A/Dコンバータを使った主な用途としては、

- ・音声入力/音声認識
- ・画像入力/画像認識
- ・温度/湿度測定
- ・レーダー
- ・電圧/電流の計測

等があります。

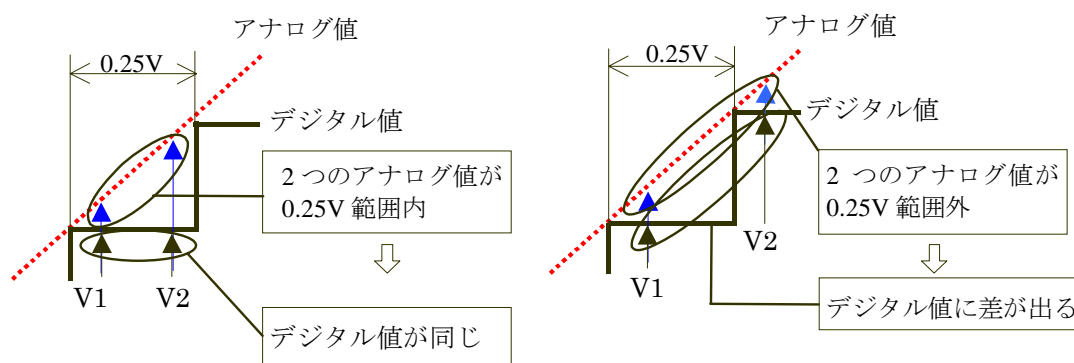
5.1.3 分解能と精度

アナログをデジタルに変換することは、連続した量を不連続な値に変換することを意味します。

ここで、0～1.0Vの電圧値を2ビットのデジタル値に変換するとします。

2ビットで表現できる値は、「00」「01」「10」「11」の4つとなります。従って、電圧をアナログ量からデジタル量に変換するには、これら4つのデジタル値のうちいずれか1つに変換しなければなりません。つまり、0～1.0Vまでの範囲を4で割った値(=0.25V)がデジタル値1つ分の幅となります。

2つの電圧値V1とV2がこの幅(0.25V)を越えると、デジタル値が変化しV1とV2の差が区別できるようになります。



この区別できる電圧の範囲、言い換えれば最小単位の1ビットに対するアナログ量を**分解能**といい、LSB(Least Significant Bit)として表現されます。一般的には分解能nビットといった具合に表現されています。入力電圧が0～5Vで分解能12ビットの時は、

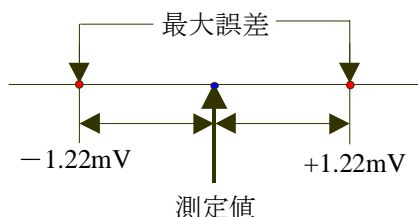
$$5 \div 4096 \doteq 0.00122(\text{V})$$

が、1LSBとなります。

AD変換されたデジタル値は、1LSBを単位とした離散値になります。このように、それ以上分割できない最小の単位(この場合LSB)に分割することを**量子化**といいます。

また、量子化することによって、連続した量が不連続な値になる際、誤差が必ず含まれてしまいます。この誤差のことを**量子化誤差**といいます。

上記の場合、精度が±1LSBならば、-1.22mV～+1.22mVの範囲で誤差があることを示しています。したがって、誤差がもっとも大きい場合、測定値+1.22mVまたは測定値-1.22mVとなります。



分解能をnビットと表現する場合、本来の分解能はフルスケール(FSR:Full Scale Range)を2のビット数乗で割った値となります。入力電圧が0～+5Vの場合はFSR=5、入力電圧が-5V～+5Vの場合はFSR=10となります。

$$1LSB = \frac{FSR}{2^n}$$

12ビットのAD変換では、最小値0、最大値4095(FFFh)となり、精度±0.04%FSRの場合、入力電圧が0～+10Vならば、FSRの±0.04%、つまり-0.004V～+0.004Vの範囲内で誤差があることを意味します。

★レンジの上限値

変換データのデジタル値の最大値に相当する電圧は、レンジの上限になりません。
例えば、12ビット分解能、±5Vレンジの時は4095が最大値となり、それに相当する電圧は
 $10 \div 4096 \times 4095 - 5 \doteq 4.9975\text{V}$ となります。

★量子化誤差

高分解能のA/Dコンバータを使えば、より高精度のAD変換が可能です。

入力電圧が0～+5Vの時、12ビット分解能と16ビット分解能では、以下の違いが生じます。

12ビット分解能: $5 \div 4096 \approx 0.00122\text{V}$ (1.22mV)

16ビット分解能: $5 \div 65536 \approx 0.00007629\text{V}$ (76μV)

ただ、高分解能のA/Dコンバータは比例して高価なので、計測に要求される精度に合わせて選択する必要があります。

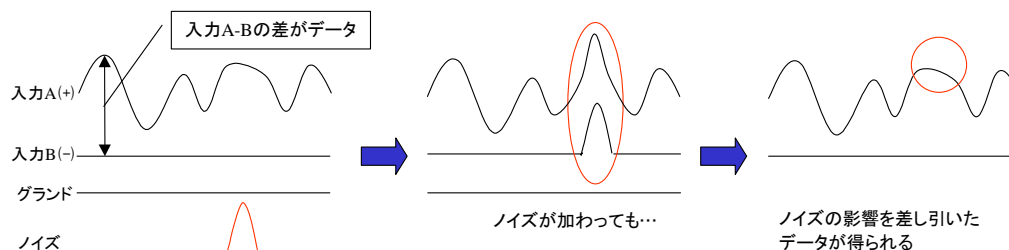
5.1.4 シングルエンド入力と差動入力

シングルエンド入力は、もっとも一般的な入力方式です。

信号源を、グランド線と信号線の2本の線で入力し、グランド線と信号線間の電圧が入力電圧としてAD変換されます。

これに対し、差動入力は信号源を3本の信号線で接続し入力する方式です。3本の信号線はそれぞれグランド線、信号線(+), 信号線(-)です。入力するデータは、グランド線と信号線(+)間の電圧(A)とグランド線と信号線(-)間の電圧(B)の差、つまり $A - B$ の値が入力電圧としてAD変換されます。

アナログ信号はノイズにより信号そのものが変化します。このような場合、差動入力を行うことで、同相ノイズの影響を受けにくくすることができます。一般的にノイズは、グランドに対して影響することが多く、シングルエンド入力ではデータに影響がでますが、差動入力についてはA, Bの電圧は影響を受けますが、 $A - B$ の値は、ノイズ分が相殺され影響を受けていないデータが得られます。差動入力を図にすると下図のようになります。



差動入力の問題点は、入力に使用する信号線が、シングルエンド入力方式が2本の信号線で構成できるのに対して、3本の信号線を必要とすることです。

このため、

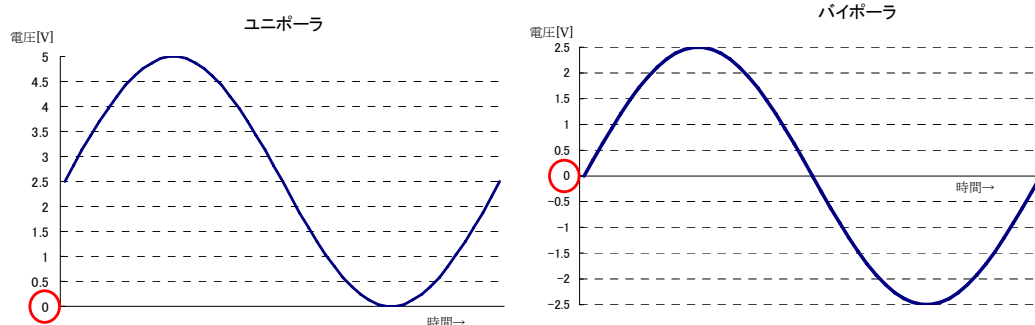
- ・入力に使用するチャンネル数を多く確保したい場合は、シングルエンド入力
- ・ノイズの影響をなるべく排したい場合は、差動入力

といった具合に、目的に応じて使い分けます。

5.1.5 バイポーラとユニポーラ

アナログ入力、入力電圧域の違いにより「ユニポーラ(unipolar)」と「バイポーラ(bipolar)」に区別することができます。ポーラとは「極」を意味し、それぞれ「単極性」、「双極性」と訳されます。

電位差(最大入力電圧－最小入力電圧)が5Vの場合、ユニポーラでは最小入力電圧0Vから最大入力電圧+5Vの入力を行います。一方、バイポーラでは0Vを中心に最小入力電圧-2.5Vから最大入力電圧+2.5Vの出力を行います。



5.1.6 変換時間とサンプリング周波数

かかる時間とコンピュータのプログラムが、そのデータを受け取る時間は、同じではありません。
変換時間とは、インタフェースモジュール上のA/Dコンバータが変換結果を算出する時間です。
実際には、そこからデータがPCI/CompactPCIバスのバス信号線を経由しCPUを通して、データを格納する領域に値が書き込まれるまで、様々な要因が複雑に絡みます。
(それは、OS、割込みのオーバーヘッド、メモリ、CPUの速度等です)

これとは別の速度指標として、「サンプリング周波数」という用語があります。

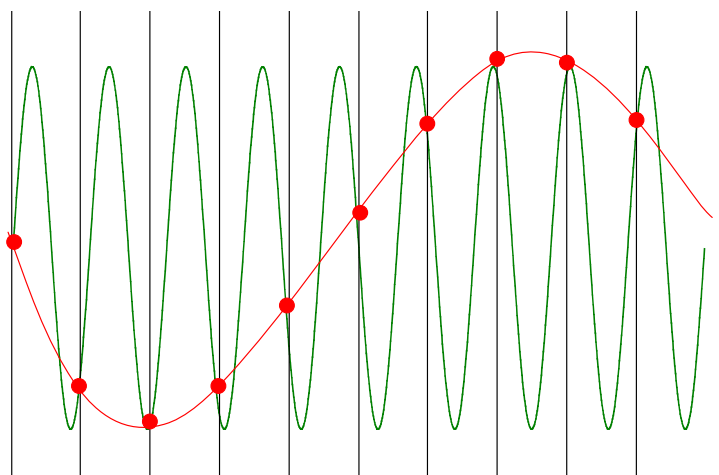
サンプリング周波数とは、1秒間に何回AD変換するかを示します。単位はHzで表されます。

例えば、サンプリング周波数500Hzとは、1秒間に500回周期的にAD変換させることを意味します。

お客様がデバイスドライバに対して指示する速度指標は、通常こちらが使われます。

5.1.7 標本化定理

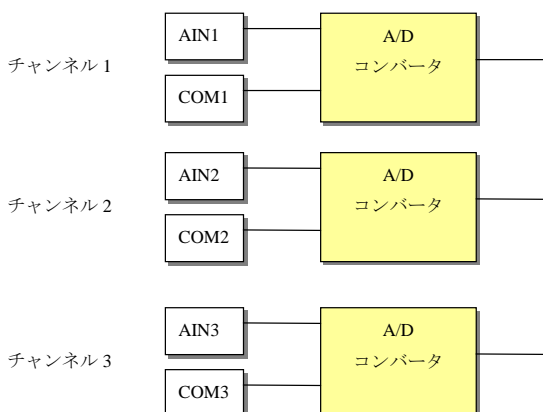
下図は、9kHzの入力に対し10kHzでサンプリングした例です。
サンプリングした点を繋いでいくと、予想外の信号が現れてきます。
この予想外の擬似信号を「折り返し信号」と言います。



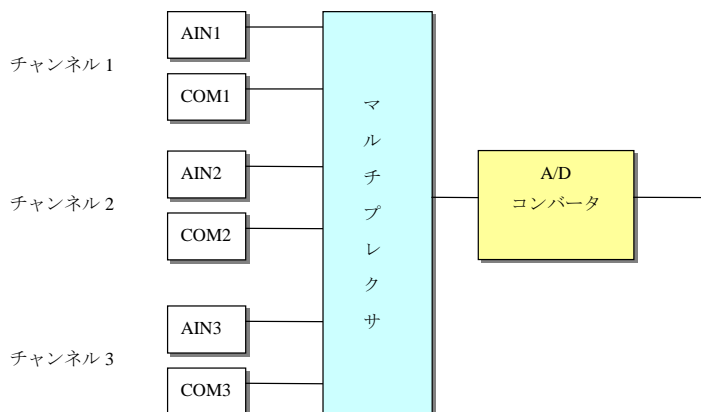
このように、音声等の入力信号に対し正確にサンプリング(標本化)を行うためには、入力信号の周波数に対して、2倍以上の周波数でサンプリングする必要があります。
これを、標本化定理と言います。

5.1.8 マルチADC方式とマルチプレクサ方式

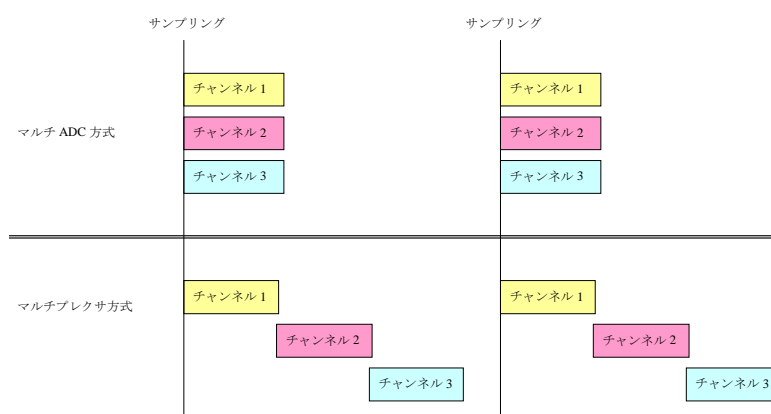
複数チャンネルの入力信号を取り扱う方式として、マルチプレクサ方式とマルチADC(同時変換)方式があります。
マルチADC方式とは、1つのチャンネルに対して、1つのA/Dコンバータが対に対応している方式です。



対して、マルチプレクサ方式とは、複数のチャンネルが1つのA/Dコンバータをマルチプレクサという切り替えスイッチで共有する方式です。



このため、複数のチャンネルをサンプリングすると、マルチADC方式とマルチプレクサ方式で実際のAD変換の時間にズレが生じます。



上図は、マルチADC方式とマルチプレクサ方式で、AD変換のタイミングを表したものです。

マルチADC方式では、各チャンネルに対してA/Dコンバータがあるため、同時にAD変換します。対してマルチプレクサ方式では、複数のチャンネルをマルチプレクサによって切り替えながら使用するため、AD変換の開始時刻に差が生じます。

両方式のメリット/デメリットを示します。

	マルチADC方式	マルチプレクサ方式
メリット	各チャンネルのAD変換に時間的なズレは生じません。	A/Dコンバータが1つで済むため、比較的安価に購入できます。
デメリット	A/Dコンバータが複数搭載されるため、どうしても高価になります。	各チャンネルのAD変換に時間的なズレが生じます。

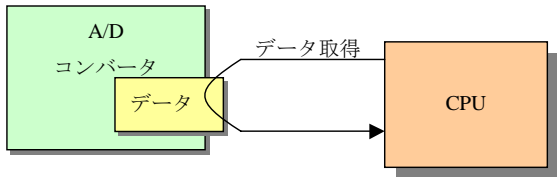
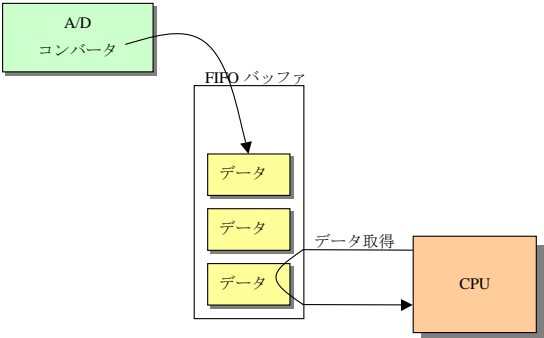
マルチプレクサ方式では時間的なズレは生じますが、位相は保たれています。

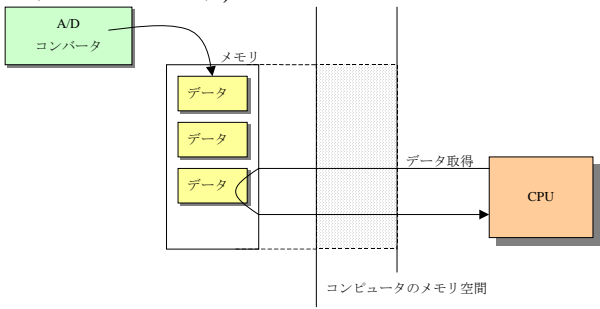
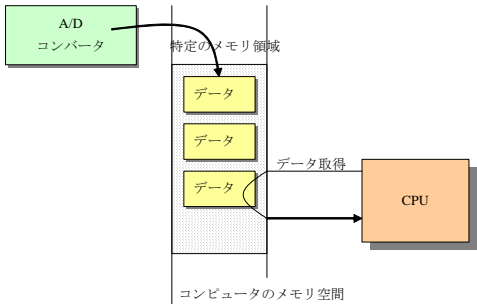
サンプリングの同時性を重視したい場合はマルチADC、複数信号の同時性は重要視されないならマルチプレクサ、という風を選択すると良いでしょう。

5.1.9 データ転送方式

アナログ信号からデジタル信号にAD変換された情報は、何らかの形でコンピュータ内のCPUに取り込む必要があります。

このCPUが、AD変換された情報にアクセスする方式にも様々なやり方があります。

方式	内容
I/O 方式	<p>AD 変換された情報は、CPU が I/O ポートを経由して 1 つずつアクセスします。</p>  <p>AD 変換されたデータは、次のサンプリングが開始される前に、そのデータを取り込まないと、次のデータで上書きされてしまいます。</p> <p>システム構成は比較的簡素なため、一般に安価です。</p>
FIFO 方式	<p>AD 変換された情報は、一旦インタフェースモジュール内の FIFO と呼ばれるバッファ領域に格納されます。</p> <p>CPU は、この FIFO バッファにアクセスして、情報を取り出します。</p>  <p>AD 変換されたデータは、FIFO バッファに溜め込まれるため、バッファがオーバーフローしない限り、次の AD 変換データに上書きされてしまうことはありません。</p> <p>このため、高速なサンプリングを行うのに向いています。</p> <p>ただし、システム構成は複雑になるため、I/O 方式に比べて高価になります。</p>

方 式	内 容
メモリ方式	<p>AD 変換された情報は、インタフェースモジュール内のメモリ領域に書き込まれます。 CPU は、このメモリ領域にアクセスして、データを取り出します。 (コンピュータのメモリ空間に空いた窓から、インタフェースモジュールのメモリにアクセスするイメージです)</p>  <p>A/D コンバータとメモリは直結されており、AD 変換されたデータが高速に書き込まれていきます。 このため、非常に高速なサンプリングを行うのに向いています。 ただし、システム構成は複雑になるため、高価になります。</p>
バスマスタ方式	<p>AD 変換された情報は、コンピュータ内の指定されたメモリ領域に対して書き込まれます。 CPU は、このメモリ領域にアクセスして、データを取り出します。</p>  <p>AD 変換されたデータを書き込むメモリは、コンピュータのメモリに対して行われるため、コンピュータのメモリを消費します。 しかし、メモリ方式と異なり、コンピュータのメモリが多ければ、それだけ多くのバッファを確保できます。 (メモリ方式では、インタフェースモジュール上のメモリに左右されます) メモリ方式同様、非常に高速なサンプリングを行うのに向いています。 ただし、システム構成は複雑になるため、高価になります。</p>

5.1.10 アナログ入力製品を使用時の指針

アナログ入力製品を使って計測を行う際一番重要なのは、ノイズ等の外的要因を受けずに安定して計測できる環境を構築することです。

計測に望ましい環境の構築の指針を幾つか提示します。

- ・コンピュータあるいはユニットの電源を安定させる。

アナログ入力製品は、コンピュータ/ユニットから電源を供給されています。

従って、コンピュータ/ユニットの電源が安定すれば、自ずと測定環境も改善されます。

例えば、電源を供給する個所を主電源の近くに配置し、タコ足状態のタップに接続しない、あるいはノイズフィルタ付きのUPS等の装置を介して電源を供給する等、3端子コンセントを使用し、グラウンドをアースする等の対策があります。

- ・計測環境と入力源とのグラウンドを同電位にする。

アナログ入力製品のグラウンドと入力源のグラウンドをケーブル等で接続し、同電位状態とします。

- ・絶縁タイプのインタフェースモジュールを使用する。

バス絶縁タイプと呼ばれるインタフェースモジュールを使用します。

インタフェースモジュール上のアナログ回路とデジタル回路が絶縁されているため、デジタルノイズがアナログ回路に混入するのを防ぐことができるので、計測に悪影響を及ぼす恐れを低減できます。

- ・入力方式を差動入力にする。

差動方式を採用することで、ノイズによる計測の悪影響を抑えることが可能です。

高精度な計測を行う時、一番推奨できる入力方式です。

- ・アナログ入力製品と信号源との間を短くする。

短いケーブルを使用します。長いケーブルでアナログ入力製品と信号源を接続すると、それだけ外部からノイズが混入する恐れが高まります。

アナログ信号の取り扱いには、オシロスコープ等の計測機器を扱うのと同程度の慎重さが求められます。

環境によっては、計測精度は大きく異なりますので、これらの事項を参考に環境を整えてください。

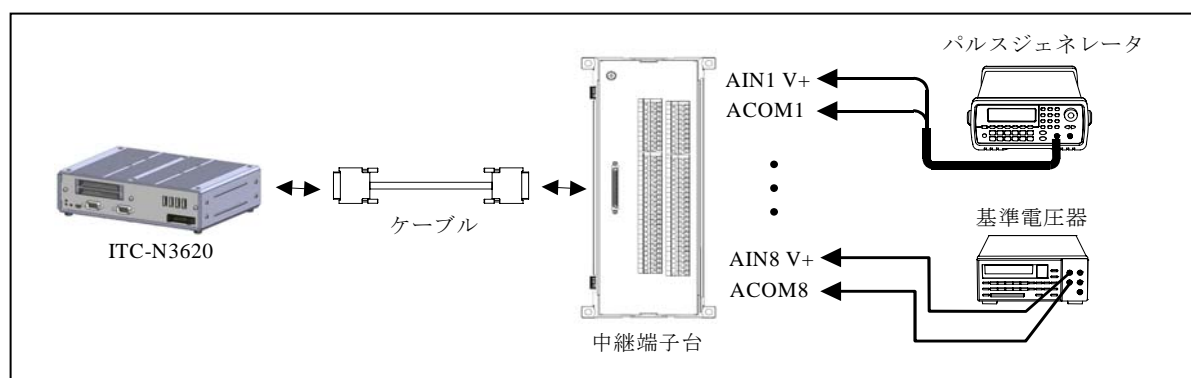
5.2 対象型式

ここでは、実際に弊社製品を用いて、簡単なプログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N3620	タフコン CD シリーズ複合モデル
-----------	-------------------

5.3 環境図



5.4 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

5.4.1 1件のAD入力を行う

Step1 プログラム作成

ここでは、AdInputAD関数を使って、1件のAD入力を行うプログラムを作成します。

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「inputad.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbiad.h"

// デバイスのオープン
int DeviceOpen(int dnum)
{
    int ret;
    ret = AdOpen(dnum);
    if (ret) {
        printf("AdOpen error: ret=%Xh¥n", ret);
        return -1;
    }
    return 0;
}

// 1件のアナログデータを取得します
int InputData(int dnum, unsigned long ChCount)
{
    ADSMPLCHREQ   AdSmplChReq[8];
    unsigned short wData[8];
    unsigned long i;

    for (i = 0; i < ChCount; i++) {
        AdSmplChReq[i].ulChNo = i+1;
        AdSmplChReq[i].ulRange = AD_10V;
    }

    // 1件のアナログデータを取得します
    ret = AdInputAD(dnum, ChCount, AD_INPUT_SINGLE, AdSmplChReq, wData);
    if (ret) {
```

```
        AdClose(dnum);
        printf("AdInputAD error: ret=%Xh¥n", ret);
        return -1;
    }

    // 取得データの表示
    for (i = 0; i < ChCount; i++) {
        printf("CH%d: %04xh¥n", i+1, wData[i]);
    }

    return 0;
}

int main(void)
{
    int  ret;
    int  dnum = 1;

    // デバイスのオープン
    ret = DeviceOpen(dnum);
    if (ret) {
        exit(EXIT_FAILURE);
    }

    // 1件のアナログ入力データを取得します
    reet = InputData(dnum, 8);
    if (ret) {
        AdClose(dnum);
        exit(EXIT_FAILURE);
    }

    // デバイスのクローズ
    AdClose(dnum);

    return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: inputad

sample1: inputad.o
$(CC) sample1.o -o sample1 -lgpg3100 -lpthread

inputad.o: inputad.c
$(CC) -Wall -c inputad.c -o inputad.o

clean:
rm -f *.o ¥##* *~ inputad
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「inputad.o」と実行ファイル「inputad」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./inputad
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
CH1:d333h
CH2:5553h
CH3:8000h
CH4:d44h
CH5:d33h
CH6:d333h
CH7:d333h
CH8:d333h
```

Step3 プログラムの解説

まず、サブルーチンのDeviceOpenについて説明します。

AD製品を制御する前段階として、AD製品をオープンするAdOpen関数を呼び出しています。

以下に引数と対応を示します。

```
int AdOpen(  
    int          nDevice      /* デバイス番号 */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。

この引数を与えて関数を呼び出すと、ドライバは引数に適合するAD製品を検索し、合致した製品がある場合は、戻り値として0を返します。

プログラマは、このデバイス番号を使って以降のAPIの呼び出しを行います。

サブルーチンのInputDataについて説明します。

ここでは、AdInputAD関数を用い1件のAD入力を行っています。

1件のAD入力では、1から複数チャンネルのAD入力が可能です。

関数を呼び出すと、引数で指定された各チャンネルのAD入力を1件だけ行い、第5引数で渡された配列に結果を返します。返されるADデータは、バイナリ値です。

第5引数では結果を格納する配列のポインタを渡す必要があります。

配列の大きさは、以下の式で求めることができます。

配列の大きさ = AD 入力に必要なデータサイズ × チャンネル数(第 2 引数の値)

AD 入力に必要なデータサイズは、AD 製品の分解能で求めることができます。

分解能	データサイズ
8 ビット	1 バイト
12 ビット	2 バイト
16 ビット	2 バイト
24 ビット	4 バイト

サンプルでは、8チャンネルのAD入力を行っています。例えば4チャンネルのAD入力を行う場合、必要な配列の大きさは、2(バイト)×4(チャンネル数)=8バイトとなります。

配列に格納されるAD入力データの順は、第4引数のチャンネル指定順となります。

サンプルでは、1, 2, 3, 4, 5, 6, 7, 8チャンネルの順にチャンネルを指定しているので、配列のデータも、1, 2, 3, 4, 5, 6, 7, 8チャンネルの順に格納されます。

もし、8, 7, 6, 5, 4, 3, 2, 1チャンネルの順にチャンネルを指定すると、配列のデータも8, 7, 6, 5, 4, 3, 2, 1チャンネルの順に格納されます。

ここでは、AdInputAD関数で得たAD入力データを、printf関数を使って全チャンネルのデータを出力しています。

最後にAD製品の制御を終了するためには、AdClose関数を用います。

オープンしたAD製品は、使用後は必ずクローズしてください。

★AdClose関数を呼ばないと、どうなるか？

AdClose関数を呼び出さないと、AD製品はオープンしたままの状態になります。

オープン状態なので、AdOpen関数を呼び出した場合、エラーが返ります。

オープンしたら必ずクローズしてください。

5.4.2 サンプリング

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「sampling.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbid.h"

unsigned short wSmplData[1024][8]; // サンプリングデータを格納するバッファ

// サンプリング条件の設定
int SetConfig(int dnum, unsigned long ChCount)
{
    int          ret;
    unsigned long i;
    ADSMPLREQ    AdSmplConfig;

    ret = AdGetSamplingConfig( dnum, &AdSmplConfig );
    if (ret) {
        printf("AdGetSamplingConfig error: ret=%Xh¥n", ret);
        return -1;
    }

    AdSmplConfig.ulChCount = ChCount;
    for (i = 0; i < ChCount; i++) {
        AdSmplConfig.SmplChReq[i].ulChNo = i+1
        AdSmplConfig.SmplChReq[i].ulRange = AD_10V;
    }
    AdSmplConfig.ulSmplNum = 1024;
    ret = AdSetSamplingConfig(dnum, &AdSmplConfig);
    if (ret) {
        printf("AdSetSamplingConfig error: ret=%Xh¥n", ret);
        return -1;
    }
    return 0;
}

// サンプリングデータの取得
int GetData(int dnum, unsigned long ChCount , unsigned long ulSmplNum)
{
    int          ret;
    unsigned long i;
    unsigned long GetDataNum = ulSmplNum;

    // サンプリングデータの取得
    ret = AdGetSamplingData(dnum, wSmplData, &GetDataNum);
    if (ret) {
```

```
        printf("AdGetSamplingData error: ret=%Xh¥n", ret);
        return -1;
    }

    // サンプリングデータの表示
    for (i = 0; i < GetDataNum; i++) {
        printf("%04Xh ", wSmplData[i][0]);
        printf("%04Xh ", wSmplData[i][1]);
        printf("%04Xh ", wSmplData[i][2]);
        printf("%04Xh ", wSmplData[i][3]);
        printf("%04Xh ", wSmplData[i][4]);
        printf("%04Xh ", wSmplData[i][5]);
        printf("%04Xh ", wSmplData[i][6]);
        printf("%04Xh ", wSmplData[i][7]);
        printf("¥n");
    }
}

int main(void)
{
    int ret;
    int dnum = 1;

    // デバイスのオープン
    ret = AdOpen(dnum);
    if (ret) {
        printf("Open error: ret=%Xh¥n", ret);
        return -1;
    }

    // サンプリング情報の設定
    ret = SetConfig ( dnum, 8 );
    if(ret){
        AdClose(dnum);
        return -1;
    }

    // サンプリングスタート.
    ret = AdStartSampling(dnum, FLAG_SYNC);
    if (ret) {
        AdClose(dnum);
        return -1;
    }

    // サンプリングデータの取得
    ret = GetData(dnum, 10);
    if (ret) {
        AdClose(dnum);
        return -1;
    }

    // デバイスのクローズ
```



```
AdClose(dnum);

return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: sampling

sampling: sampling.o
$(CC) sampling.o -o sampling -lgpg3100 -lpthread

sampling.o: sampling.c
$(CC) -Wall -c sampling.c -o sampling.o

clean:
rm -f *.o ¥##* *~ sampling
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「sampling.o」と実行ファイル「sampling」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./sampling
```

プログラムの実行に成功すれば、以下のようにチャンネル 1～8 までのデータが件数分の出力が行なわれます。
下記に表示しているデータは例となります。実際には入力されているデータが取得されます。

```
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h
:
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h
```

Step3 プログラムの解説

基本的には、サンプリングの開始を指示する関数、停止を指示する関数、サンプリング条件を設定する関数、取得したデータを取り出す関数の4組でサンプリング処理を行うことができます。

下表に、各組に該当する関数の一覧(一部)を示します。

カテゴリ	関数名
サンプリング開始関数	AdStartSampling
サンプリング停止関数	AdStopSampling
サンプリング条件設定関数	AdGetSamplingConfig, AdSetSamplingConfig
サンプリングデータ操作関数	AdGetSamplingData, AdClearSamplingData

サンプリングの条件を設定する場合には、AdGetSamplingConfig関数とAdSetSamplingConfig関数を使用します。

AdGetSamplingConfig関数とAdSetSamplingConfig関数の書式は、それぞれ下記になります。

```
int AdGetSamplingConfig(  
    int          nDevice,          /* デバイス番号 */  
    PADSMPLEQ    pSmplConfig      /* サンプリング構造体へのポインタ */  
);
```

```
int AdSetSamplingConfig(  
    int          nDevice,          /* デバイス番号 */  
    PADSMPLEQ    pSmplConfig      /* サンプリング構造体へのポインタ */  
);
```

引数名	内 容
nDevice	AdOpen 関数でオープンしたデバイス番号を指定してください。
pSmplConfig	AD サンプリング構造体(ADSMPLEQ 構造体)へのポインタを指定します。

AdGetSamplingConfig関数を実行することによって、AdOpen関数実行直後にした場合には、デフォルト値を取得でき、AdSetSamplingConfig関数で設定変更した後に実行した場合に、現在設定されているサンプリング条件を取得することができます。

AdSetSamplingConfig関数では、サンプリング条件のチャンネル数、サンプリング周波数、サンプリング件数、トリガ条件等を設定することができます。

ここでは、チャンネル数とサンプリング件数の変更を行っています。

他のサンプリング条件については、デフォルト値を設定しています。

サンプリング条件の設定が終われば、次はサンプリングを開始させます。

サンプリングを開始するには、AdStartSampling関数を実行します。

AdStartSampling関数の書式については、下記になります。

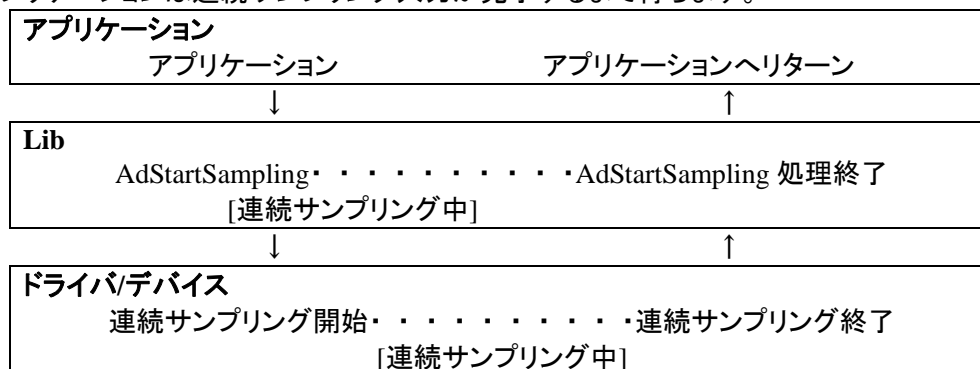
```
int AdStartSampling(
    int          nDevice,          /* デバイス番号 */
    int          nSyncFlag         /* サンプリング入力処理 */
);
```

引数名	内 容		
nDevice	AdOpen 関数でオープンしたデバイス番号を指定してください。		
nSyncFlag	サンプリング入力処理を同期で行うか非同期で行うかを指定します。		
	識別子	値	内容
	FLAG_SYNC	1	同期処理でサンプリング入力を行います。
	FLAG_ASYNC	2	非同期処理でサンプリング入力を行います。

同期／非同期処理について

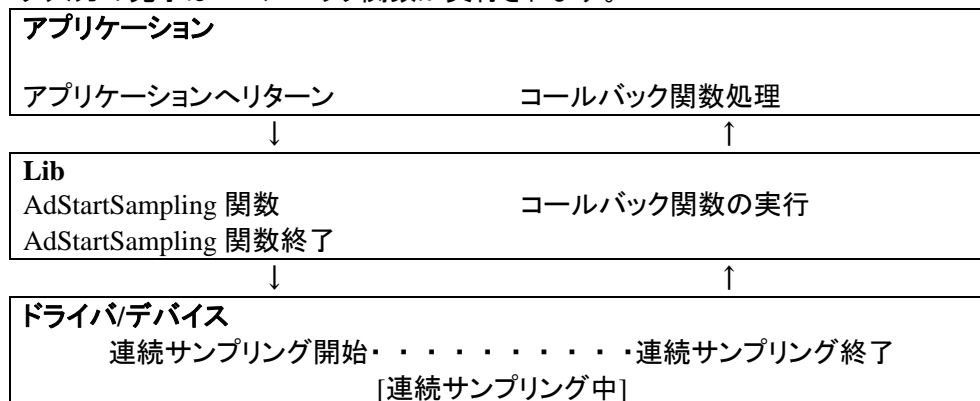
・同期入力(FLAG SYNC)

アプリケーションは連続サンプリング入力が完了するまで待ちます。



・非同期入力(FLAG ASYNC)

アプリケーションは連続サンプリング入力の完了まで待たず、制御が API から即、戻ります。連続サンプリング入力の完了はコールバック関数が実行されます。



ここでは、同期サンプリングを指定します。

よって、サンプリングが完了するまで、AdStartSampling関数からは返ってこないことになります。

サンプリングが完了すれば、次はサンプリングデータを取得することになります。

サンプリングデータの取得には、AdGetSamplingData関数を実行します。

AdGetSamplingData関数の書式については下記になります。

```
int AdGetSamplingData(  
    int                nDevice,          /* デバイス番号 */  
    void*              pSmplData,        /* サンプリングデータへのポインタ */  
    unsigned long*     ulSmplNum         /* サンプリングデータ件数へのポインタ */  
);
```

引数名	内 容
nDevice	AdOpen 関数でオープンしたデバイス番号を指定してください。
pSmplData	アナログ入力デバイスから取得するサンプリングデータを格納するためのバッファへのポインタを指定します。
ulSmplNum	アナログ入力デバイスから取得するサンプリングデータ件数を格納している変数へのポインタを指定します。関数実行後、サンプリングデータ件数をulSmplNumに格納します。

サンプリングデータの取得を行うにあたって、ulSmplNumで指定したサンプリング件数のバッファを確保する必要があります。

バッファサイズは、指定チャンネル数×サンプリング件数×データサイズになります。

本関数を実行すると、ulSmplNumには取得できた件数が返ります。

もし、指定した件数より少ない件数しかサンプリングされていない場合に本関数を実行した場合には取得できる件数しか取得できません。

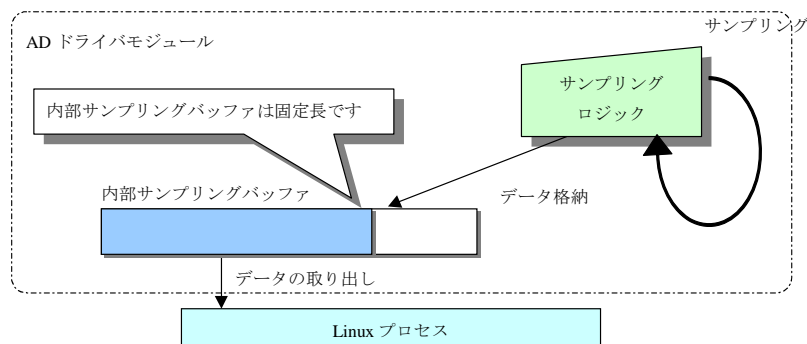
サンプリング関連の関数は、サンプリング処理を肩代わりするだけでなく、トリガ条件を指定して、サンプリングを行わせることができます。

例えば、「入力電圧が2.5V以上になったらサンプリングを開始する」「外部トリガ信号がHighからLow状態になるとサンプリングを開始する」「入力電圧が-2.5～2.5Vの範囲を超えたらサンプリングを停止する」等です。

他にも、トリガを検出して少し経ってからサンプリングを停止、数件のデータを取り込み、平均化したデータをサンプリングデータとして保存する等といった機能が提供されています。

サンプリング関連の関数を使用する時、内部バッファ構造について知っておくべき事項があります。

まず、ADドライバモジュール内では、固定長のバッファ領域をサンプリングデータの格納に使用します。(この固定長のバッファは、お客様が用意して頂く必要があります)

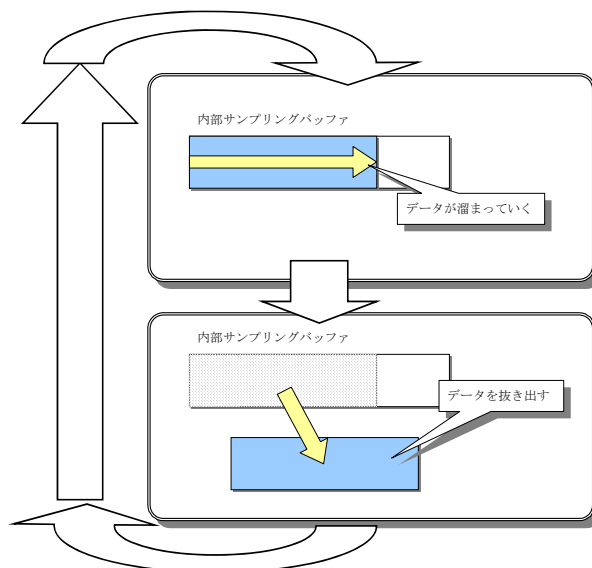


ADドライバモジュールの内部サンプリングバッファは固定長

バッファは固定長のリングバッファ構造として使用されます。従って、サンプリングデータを溜め続けることはできません。古いデータから上書きされてしまうことになります。

これを避けるには、一定のデータが溜まると、バッファからデータを抜き出す必要があります。

こうすることで、内部のバッファ以上のデータをサンプリングすることができます。



サンプルでは、一定件数のデータが溜まるとコールバックされるイベントコールバック関数を用い、サンプリング中にデータを抜き取ることで、永続的なサンプリングを実現しています。

5.4.3 割り込み処理

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「adevent.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbid.h"

int flag;
unsigned short wSmplData[1024][8]; // サンプリングデータを格納するバッファ

// コールバック関数
void event_proc( int dummy )
{
    printf( "The sampling is successfully completed.\n");
    flag = 0;
}

// サンプリング条件の設定
int SetConfig(int dnum, unsigned long ChCount)
{
    int ret;
    unsigned long i;
    ADSMPLREQ      AdSmplConfig;

    // サンプリング条件の取得
    ret = AdGetSamplingConfig( dnum, &AdSmplConfig );
    if (ret) {
        printf("AdGetSamplingConfig error: ret=%Xh\n", ret);
        return -1;
    }

    // サンプリング条件の設定
    AdSmplConfig.ulChCount = ChCount;
    for (i = 0; i < ChCount; i++) {
        AdSmplConfig.SmplChReq[i].ulChNo = i+1
        AdSmplConfig.SmplChReq[i].ulRange = AD_10V;
    }
    AdSmplConfig.ulSmplNum = 1024;
    ret = AdSetSamplingConfig(dnum, &AdSmplConfig);
    if (ret) {
        printf("AdSetSamplingConfig error: ret=%Xh\n", ret);
        return -1;
    }

    return 0;
}
```

```
// サンプリングデータの取得
int GetData(int dnum, unsigned long ChCount, unsigned long ulSmplNum)
{
    int ret;
    unsigned long i;
    unsigned long GetDataNum = ulSmplNum;

    // サンプリングデータの取得
    ret = AdGetSamplingData(dnum, wSmplData, &GetDataNum);
    if (ret) {
        printf("AdGetSamplingData error: ret=%Xh¥n", ret);
        return -1;
    }

    // サンプリングデータの表示
    for (i = 0; i < ulSmplNum; i++) {
        printf("%04Xh ", wSmplData[i][0]);
        printf("%04Xh ", wSmplData[i][1]);
        printf("%04Xh ", wSmplData[i][2]);
        printf("%04Xh ", wSmplData[i][3]);
        printf("%04Xh ", wSmplData[i][4]);
        printf("%04Xh ", wSmplData[i][5]);
        printf("%04Xh ", wSmplData[i][6]);
        printf("%04Xh ", wSmplData[i][7]);
        printf("¥n");
    }
}

int main(void)
{
    int ret, dnum;
    unsigned long ulSmplNum; // Number of the sampling data acquisition
    unsigned long i;

    // Open a device.
    ret = AdOpen(dnum);
    if (ret) {
        printf("Open error: ret=%Xh¥n", ret);
        return -1;
    }

    // コールバック関数の登録
    ret = AdSetBoardConfig(dnum, 0, event_proc, 0);
    if (ret) {
        printf("AdSetBoardConfig error: ret=%Xh¥n", ret);
        AdClose(dnum);
        return -1;
    }

    // サンプリング情報の設定
    ret = SetConfig(dnum, 8);
```

```
if(ret){
    AdClose(dnum);
    return -1;
}

flag = 1;

// サンプリングスタート.
ret = AdStartSampling(dnum, FLAG_ASYNC);
if (ret) {
    AdClose(dnum);
    return -1;
}

// コールバック関数が呼び出されるまで待機.
while(flag);

// サンプリングデータの取得
ret = GetData(dnum, wSmplData, &ulSmplNum);
if (ret) {
    AdClose(dnum);
    return -1;
}

// デバイスのクローズ
AdClose(dnum);

return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: adevent

adevent: adevent.o
    $(CC) adevent.o -o adevent -lgpg3100 -lpthread

adevent.o: adevent.c
    $(CC) -Wall -c adevent.c -o adevent.o

clean:
    rm -f *.o ¥#* *~ adevent
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。

© 2013 Interface Corporation. All rights reserved.



下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「adevent.o」と実行ファイル「adevent」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./adevent
```

プログラムの実行に成功すれば、以下のようにチャンネル 1～8 までのデータが件数分の出力が行なわれます。
下記に表示しているデータは例となります。実際には入力されているデータが取得されます。

```
The sampling is successfully completed.  
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h  
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h  
:  
1111h 2222h 3333h 4444h 5555h 6666h 7777h 8888h
```

Step3 プログラムの解説

サンプリングの基本的な処理については、「5.4.2 サンプリング」で解説しているため省略します。

ここでは割り込み機能を使用するために追加したソースコード部分について説明します。

まず、割り込み機能を使用するためには、AdSetBoardConfig関数でコールバック関数を登録する必要があります。

コールバック関数とは、割り込みが発生した場合に呼び出される関数となります。

AdSetBoardConfig関数の書式は下記になります。

```
int AdSetBoardConfig(  
    int          nDevice,          /* デバイス番号 */  
    int          nReserved1,       /* 予約 */  
    PLPADCALLBACK pEventProc,      /* ユーザ・コールバック関数 */  
    int          uData             /* ユーザデータ */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。
nReserved1	予約です
pEventProc	コールバック関数を指定します。
uData	コールバック関数へ渡すユーザデータを指定します。

コールバック関数の登録としては、第三引数にて、呼び出したいコールバック関数を指定します。

プログラムでは、「event_proc」という関数を呼び出すように設定しています。

コールバック関数の書式は、下記である必要があります。

```
void CallbackProc(  
    int          uData /* ユーザ・データ */  
);
```

ユーザデータは、AdSetBoardConfig関数の第四引数で指定します。

ユーザデータは、コールバック関数の第一引数に値がそのまま渡されます。

こういった場合に使用するかは、複数のデバイスで割り込みを登録した場合、どのデバイスに対する割り込みかを判別するときに使用します。

本プログラムで、サンプリングが終了した場合に、コールバック関数が呼び出されるため、コールバック関数内で、「The sampling is successfully completed.」という文字を出力し、サンプリングを終了しているかのフラグとして使用しているグローバル変数のflagを0にしています。

本プログラムでは、AdStartSampling関数で非同期サンプリングとして実行し、while文でコールバック関数がコールされるまで待機するように作成しています。

コールバック関数が発生すると、サンプリングデータの取得を行い、画面にサンプリングデータを出力するようになります。

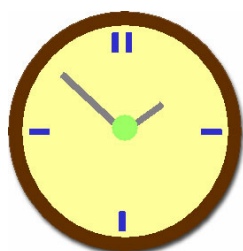
第6章 アナログ出力

6.1 アナログの概要

これまでに、アナログ(analog)という言葉を一度は耳にしたことがあると思います。

ではアナログとは何でしょう。しばしばデジタルという言葉と対比し使われますが、デジタルが「1つ2つと区切って数えられる」のに対し、アナログとは「連続した量で、1つ2つと区切って数えられない」ものを言います。

デジタル時計とアナログ時計を例にあげてみます。



アナログ時計



デジタル時計

デジタル時計では、「1:53」と単純に読み取ればよいわけですが、アナログ時計で仮に小数点以下を読み取っていくとしたらどうでしょう。「1:53」、「1:53.1」、「1:53.124」、「1:53.1245.....」と、きりがありませんね。

6.1.1 D/Aコンバータ

D/Aコンバータとは、Digital to Analog Converterの略で、デジタル・アナログ変換器と訳されます。デジタル・アナログ変換器とはデジタル値をアナログ量に変換してくれる装置のことです。

本冊子で紹介するDA製品もD/Aコンバータが搭載された、インタフェース製品です。

コンピュータはデジタル信号で動いています。一方、私たちが普段、目や耳にすることのできる「光」や「音」の情報は全てアナログ信号です。

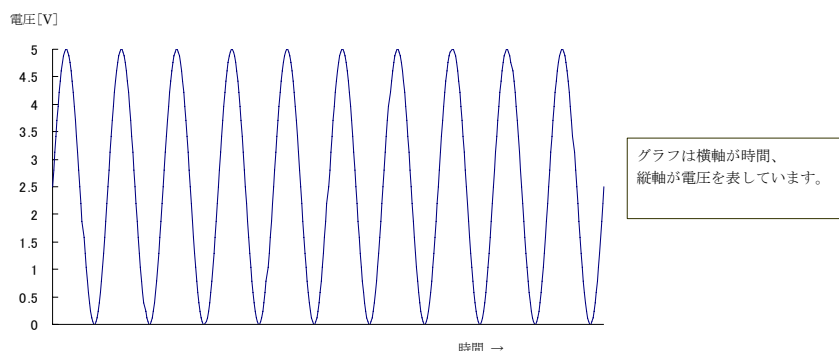
ところで、今のコンピュータはCDの再生ができたり、音声の再生が行えたりと、このアナログ信号である「音」の出力が行えます。

これは、コンピュータ内部のデジタルな情報がD/Aコンバータによってアナログ情報に変換され、スピーカーより出力されているからです。

6.1.2 アナログ出力

DA製品は、アナログ信号を出力することができるインタフェース製品です。

DA製品からは、電圧または電流の固定値または時間とともに連続的に変化させた、電気信号が出力されます。

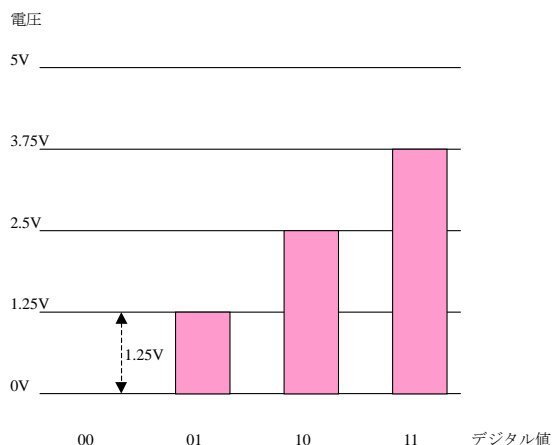


6.1.3 分解能と精度

D/Aコンバータは、デジタルデータを比例する電圧値に変換して出力します。

ここで、2ビットのデジタル値を0～5Vの範囲の電圧に変換するとします。

2ビットで表現できる値は、「00」、「01」、「10」、「11」の4つです。従って、デジタル量をアナログ量に変換するには、これら4つのデジタル値を、0～5Vの電圧値のどれかに割り当てなければなりません。つまり、0～5Vまでの範囲を4で割った値(=1.25V)が、デジタル量1つに対する電圧の幅に対応します。



この区別できる電圧の範囲、言い換えれば最小単位の1ビットに対するアナログ量を**分解能**といい、LSB(Least Significant Bit)として表現されます。一般的には分解能nビットといった具合に表現されています。出力電圧が0～5Vで分解能12ビットの時は、

$$5 \div 4096 \div 0.00122(\text{V})$$

が、1LSBとなります。

★レンジの上限値

デジタル値の最大値に相当する電圧は、レンジの上限になりません。

例えば、12ビット分解能、±5Vレンジの時は4095が最大値となり、それに相当する電圧は、 $10 \div 4096 \times 4095 - 5 \div 4.9975\text{V}$ となります。

★量子化誤差

高分解能のD/Aコンバータを使えば、より高精度のDA変換が可能です。

出力電圧が0～+5Vの時、12ビット分解能と16ビット分解能では、以下の違いが生じます。

12ビット分解能: $5 \div 4096 \div 0.00122\text{V}(1.22\text{mV})$

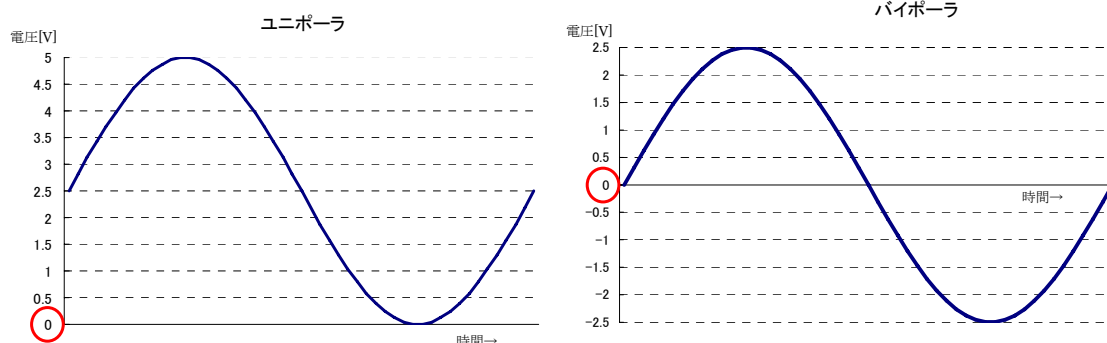
16ビット分解能: $5 \div 65536 \div 0.00007629\text{V}(76\mu\text{V})$

ただ、高分解能のD/Aコンバータは、比例して高価なので、使用用途に要求される精度に合わせて選択する必要があります。

6.1.4 バイポーラとユニポーラ

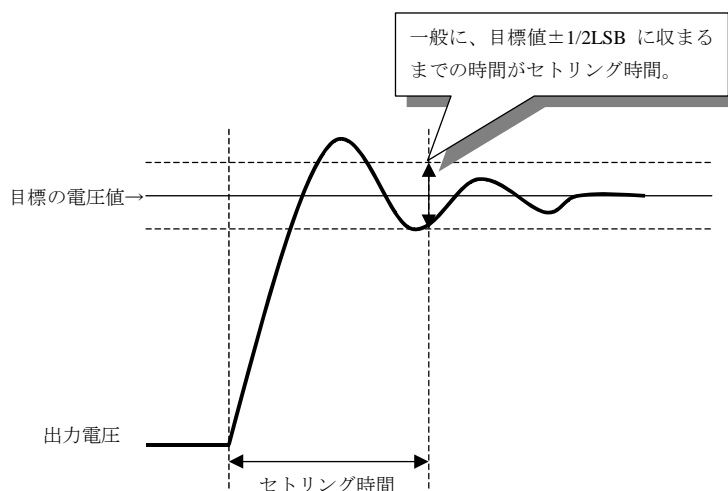
アナログ出力は、出力電圧域の違いにより「ユニポーラ(unipolar)」と「バイポーラ(bipolar)」とに区別することができます。ポーラとは「極」を意味し、それぞれ「単極性」、「双極性」と訳されます。

出力電位差(最大出力電圧－最小出力電圧)が5Vの場合、ユニポーラでは最小出力電圧0Vから最大出力電圧+5Vの出力を行います。一方、バイポーラでは0Vを中心に最小出力電圧-2.5Vから最大出力電圧+2.5Vの出力を行います。



6.1.5 セトリング時間とアナログ出力更新レート

D/Aコンバータを用いて、デジタル信号をアナログ信号に変換する時、アナログ出力が規定の精度に収まるまでの時間をセトリング時間と呼びます。



多くのD/Aコンバータでは、出力のフルスケール変化から出力電圧が $\pm 1/2\text{LSB}$ に収まるまでの時間を、セトリング時間として定義しています。

カタログやマニュアルの仕様等に記載されている $5\mu\text{s}$ や $0.2\mu\text{s}$ 等の値は、デジタル値を与えられ、D/Aコンバータが目標の電圧/電流にまでアナログ信号を変化させるまでの時間を指します。

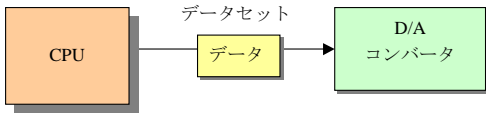
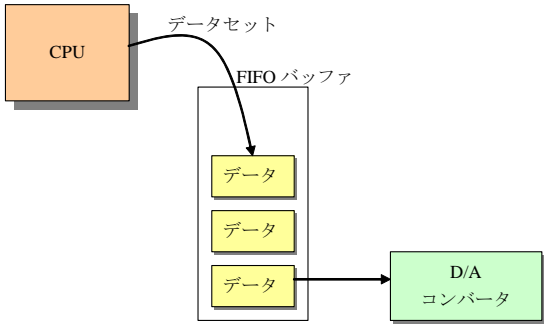
これとは別の速度指標として、「アナログ出力更新レート」という用語があります。

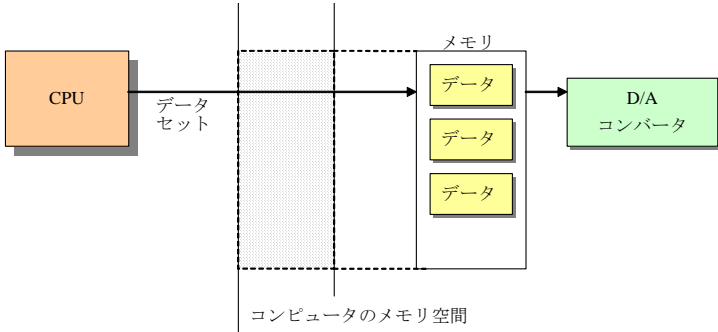
アナログ出力更新レートとは、1秒間に何回アナログ出力するかを示します。単位はHzで表されます。

例えば、アナログ出力更新レート500Hzとは、1秒間に500回 周期的にアナログ出力させることを意味します。お客様がデバイスドライバに対して指示する速度指標は、通常こちらが使われます。

6.1.6 データ転送方式

D/Aコンバータにデータを渡す方法には、下記に示す方法があります。

方式	内容
I/O 方式	<p>CPU は、I/O ポートを経由して D/A コンバータにデータを渡します。</p>  <p>アナログ出力させるために、都度 I/O ポートにアクセスしなければなりません。従って、高速にアナログ出力を繰り返す用途には向いていません。システム構成は比較的簡素なため、一般に安価です。</p>
FIFO 方式	<p>CPU は、I/O ポートを経由して、一旦製品内の FIFO と呼ばれるバッファ領域に、データを格納します。</p>  <p>D/A コンバータと FIFO メモリは直結されており、アナログ出力されるデータが高速に読み込まれます。このため、高速かつ連続的なアナログ出力を行うのに向いています。ただし、システム構成は複雑になるため、I/O 方式に比べて高価になります。</p>

方式	内容
メモリ方式	<p>CPU は、製品内のメモリ領域に D/A コンバータに渡すデータを書き込み、指令を与えると、製品がメモリからデータを読み取って、自分で D/A コンバータにデータを渡します。</p>  <p>D/A コンバータとメモリは直結されており、アナログ出力されるデータが高速に読み込まれます。データのセットと D/A コンバータのアナログ出力は同期する必要が無いので、非常に高速かつ連続的なアナログ出力を行うのに向いています。ただし、システム構成は複雑になるため、高価になります。</p>

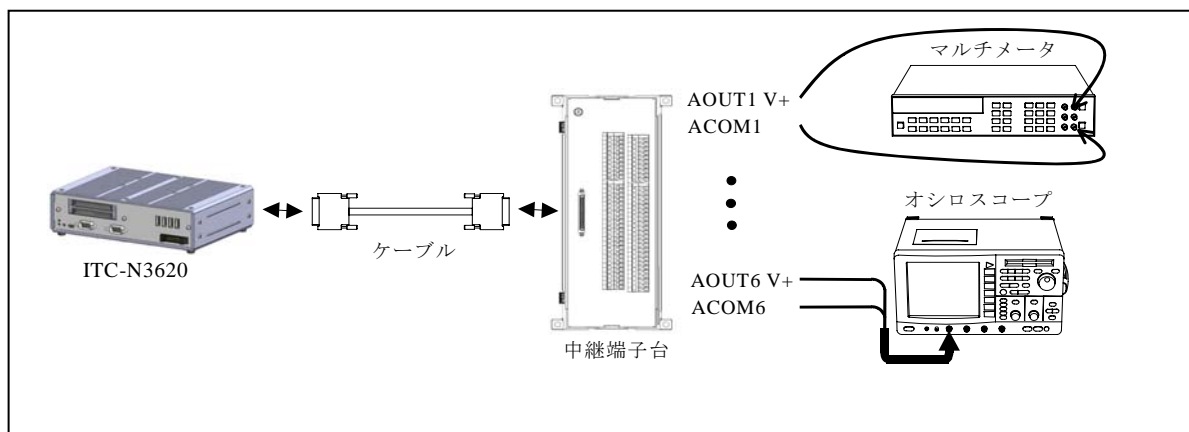
6.2 対象型式

ここでは、実際に弊社製品を用いて、簡単なプログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N3620	タフコン CD シリーズ複合モデル
-----------	-------------------

6.3 環境図



6.4 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

6.4.1 1件出力

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「outputda.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbida.h"

// 1 件のデータを出力
int OutputData(int dnum, unsigned long ChCount)
{
    int    ret;
    unsigned long i;
    unsigned short    wData[ChCount]; // Output data storage area
    DASMPCHREQ    DaSmplChReq[ChCount]; // Output conditions setting structure

    // チャンネルとレンジの設定
    for (i = 0; i < ChCount; i++) {
        DaSmplChReq[i].ulChNo = i+1;
        DaSmplChReq[i].ulRange = DA_10V;
        wData[0] = 0x2000 * (i+1);
    }

    // Start the analog output.
    ret = DaOutputDA( dnum, ChCount, DaSmplChReq, wData );
    if (ret) {
        printf("DaOutputDA error: ret=%Xh¥n", ret);
        return -1;
    }

    return 0;
}
```

```
}

// メイン
int main(void)
{
    int    ret;
    int    dnum = 1;

    // デバイスのオープン
    ret = DaOpen(dnum);
    if(ret){
        printf("Open error: ret=%Xh¥n", ret);
        return -1;
    }

    // 1 件のデータを出力
    ret = OutputData(dnum, 6);
    if(ret){
        DaClose( dnum );
        return -1;
    }

    printf("analog output ¥n");
    sleep(1);

    // デバイスのクローズ
    DaClose( dnum );

    return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: outputda

outputda: outputda.o
$(CC) adevent.o -o outputda -lgpg3300 -lpthread

outputda.o: outputda.c
$(CC) -Wall -c outputda.c -o outputda.o

clean:
rm -f *.o ¥##* *~ outputda
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
Make
```

コンパイルが成功すると、オブジェクトファイル「outputda.o」と実行ファイル「outputda」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./outputda
```

プログラムの実行に成功すれば、以下の出力が行なわれます。
1 秒間出力を行いプログラムが終了します。

```
analog output
```

Step3 プログラムの解説

製品の制御では、オープン処理を行わないと何の制御もできません。ファイルに読み書きする時、open関数を使ってオープンしないと、何もできないのと同じことです。

そして、製品の制御を終える時、必ずクローズ処理を行う必要があります。

サンプルでも、この形式に則って処理を行っています。

以降では細かく動きを見ていきますが、この基本事項を守っていることに注意して読み進めてください。

(DaOpen関数)

DA製品を制御する前段階として、DA製品をオープンするDaOpen関数を呼び出しています。

以下に引数と対応を示します。

引数名	内 容
pnDevice	上記の組み合わせに対するデバイス番号が、呼出し後にセットされます。

この引数を与えて関数を呼び出すと、ドライバモジュールは、引数に適合するDA製品を検索し、合致した製品に対してデバイス番号を割り振り、第4引数に返します。

プログラマは、このデバイス番号を使って、以降のAPIの呼び出しを行います。

(DaOutputDA関数)

ここでは、DaOutputDA関数を用い、1件のDA出力を行っています。

1件のDA出力では、1チャンネルから複数チャンネルのDA出力が可能です。

第1引数で、DaOpenEx関数で得たデバイス番号を、第2引数でDA出力するチャンネル数、第3引数で各チャンネルの詳細設定、第4引数で出力するデータの配列のポインタを渡します。この引数を与えて関数を呼び出すと、ドライバモジュールは、受け取った引数に従って1件のDA出力を、指定されたチャンネル数分行います。

第4引数ではDA出力するデータの配列のポインタを渡す必要があります。

配列の大きさは、以下の式で求めることができます。

配列の大きさ = DA 出力に必要なデータサイズ × チャンネル数(第 2 引数の値)

DA 出力に必要なデータサイズは、DA 製品の分解能で求めることができます。

分解能	データサイズ
8 ビット	1 バイト
12 ビット	2 バイト
16 ビット	2 バイト
24 ビット	4 バイト

サンプルでは、2チャンネルのDA出力を行っていますが、例えば4チャンネルのDA出力を行う場合、必要な配列の大きさは、 $2(\text{バイト}) \times 4(\text{チャンネル数}) = 8\text{バイト}$ となります。

配列に格納されるDA出力データの順は、第3引数のチャンネル指定順に設定します。

サンプルでは、1、2チャンネルの順にチャンネルを指定しているので、配列のデータも、1、2チャンネルの順に格納します。

もし、2、1チャンネルの順にチャンネルを指定すると、配列のデータも2、1チャンネルの順に格納します。

(DaClose関数)

DA製品の制御を終了するためには、DaClose関数を用います。

オープンしたDA製品は、使用後は必ずクローズしてください。

★DaClose関数を呼ばないと、どうなるか？

DaClose関数を呼び出さないと、DA製品はオープンしたままの状態になります。

オープン状態なので、DaOpen関数を呼び出した場合、エラーが返ります。

オープンしたら必ずクローズしてください。

6.4.2 連続出力

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「update.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbida.h"

unsigned short wSmplData[1024][6]; // Output data storage area

// 連続出力条件の設定
int SetUpdateConfig(int dnum)
{
    int ret;
    int i;
    DASMPREQ SmplConfig; // Output conditions setting structure

    // 連続出力条件の取得
    ret = DaGetSamplingConfig(dnum, &SmplConfig);
    if(ret){
        printf("DaGetSamplingConfig error: ret=%Xh¥n", ret);
        return -1;
    }

    SmplConfig.ulChCount = 6;
    for (i = 0; i < SmplConfig.ulChCount; i++) {
        SmplConfig.SmplChReq[i].ulChNo = i+1;
        SmplConfig.SmplChReq[i].ulRange = DA_10V;
    }

    // 連続出力条件の設定
    ret = DaSetSamplingConfig(dnum, &SmplConfig);
    if(ret){
        printf("DaSetSamplingConfig error: ret=%Xh¥n", ret);
        return -1;
    }

    return 0;
}

// 連続出力データの設定
int SetUpdateData(int dnum, unsigned long ulSmplBufferSize)
{
    unsigned long i;

    // 連続出力データの生成
    for (i = 0; i < ulSmplBufferSize; i++) {
```



```
        if (i%2) {
            wSmplData[i][0]=0xFFFF;
            wSmplData[i][1]=0x0000;
            wSmplData[i][2]=0xFFFF;
            wSmplData[i][3]=0x0000;
            wSmplData[i][4]=0xFFFF;
            wSmplData[i][5]=0x0000;
        } else {
            wSmplData[i][0]=0x0000;
            wSmplData[i][1]=0xFFFF;
            wSmplData[i][2]=0x0000;
            wSmplData[i][3]=0xFFFF;
            wSmplData[i][4]=0x0000;
            wSmplData[i][5]=0xFFFF;
        }
    }

    // 連続出力データの設定
    ret = DaSetSamplingData(dnum, wSmplData, ulSmplBufferSize);
    if(ret){
        printf("DaSetSamplingData error: ret=%Xh¥n", ret);
        return -1;
    }

    return 0;
}

// メイン
int main(void)
{
    int    ret;
    int    dnum = 1;
    int    i;
    unsigned long    ulSmplBufferSize;// Output buffer size

    // デバイスのオープン
    ret = DaOpen(dnum);
    if(ret){
        printf("Open error: ret=%Xh¥n", ret);
        return -1;
    }

    // バッファ数とコールバック関数の設定
    ulSmplBufferSize = 1024;
    ret = DaSetBoardConfig(dnum, ulSmplBufferSize, NULL, NULL, 0);
    if(ret){
        printf("DaSetBoardConfig error: ret=%Xh¥n", ret);
        DaClose(dnum);
        return -1;
    }

    // 連続出力条件の設定
```

```
ret = SetUpdateConfig(dnum);
if (ret) {
    DaClose(dnum);
    return -1;
}

// 連続出力データの設定
ret = SetUpdateData(dnum, ulSmplBufferSize);
if (ret) {
    DaClose(dnum);
    return -1;
}

// 連続出力の開始
ret = DaStartSampling(dnum, FLAG_SYNC);
if (ret) {
    printf("DaStartSampling error: ret=%Xh¥n", ret);
    DaClose(dnum);
    return -1;
}

// デバイスのクローズ
DaClose(dnum);

return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: update

update: update.o
    $(CC) adevent.o -o update -lgpg3300 -lpthread

outputda.o: outputda.c
    $(CC) -Wall -c update.c -o update.o

clean:
    rm -f *.o ¥#* *~ update
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
#make
```

コンパイルが成功すると、オブジェクトファイル「update.o」と実行ファイル「update」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
#./update
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

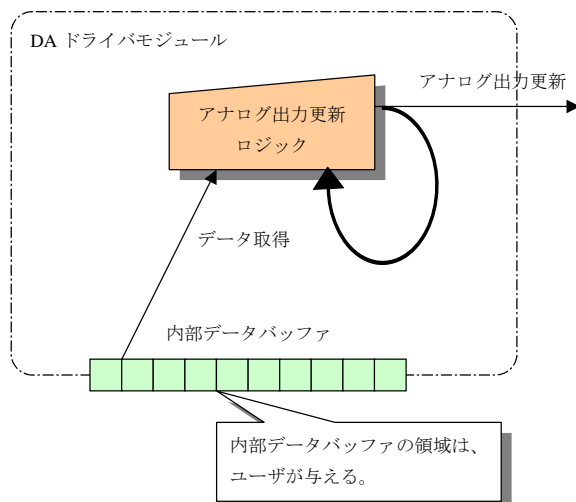
```
#InputData: 1h
```

Step3 プログラムの解説

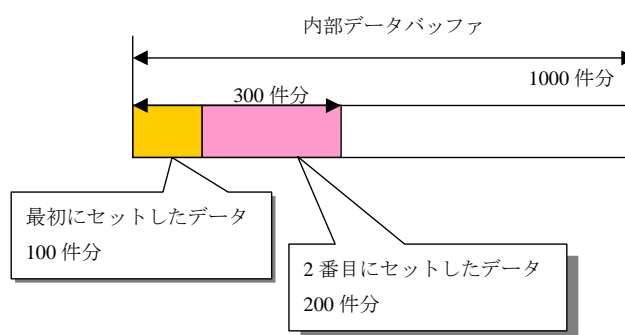
サンプルでは、あまり見えていませんが、DAドライバは、内部で様々な仕事をを行います。ここでは、アナログ出力更新を行う際の構造について、簡単に紹介します。

DaStartSampling関数等の、アナログ出力更新をDAドライバモジュールが行う場合、アナログ出力更新を行うために必要な内部データバッファ領域を、ユーザから提供します。(DaSetBoardConfig関数にて行います)

ドライバモジュールは、得られたバッファ領域を使って、アナログ出力更新を行います。



DaSetSamplingData関数でデータをセットされると、内部データバッファの未使用領域にデータが追加されます。例えば、1000件分の内部データバッファを確保しておき、最初に100件のデータをセット後、次に200件のデータをセットしようとした時、内部データバッファは、合計300件のデータを内部にセットした状態になります。



アナログ出力更新は、セットされたデータ部分に対してのみ行われます。

従って、上図の例では、300件分のデータが、アナログ出力更新の対象となります。

データのセットを繰り返すと、内部データバッファが一杯になってしまいます。

DaClearSamplingData関数を使うと、内部データバッファ内の全てのデータを消去することができます。

6.4.3 割り込み処理

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「daevent.c」として保存してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fbida.h"

int          flag;
unsigned short wSmplData[1024][6]; // Output data storage area

// コールバック関数
void event_proc( int dummy )
{
    printf( "The sampling is successfully completed.\n");
    flag = 0;
}

// 連続出力条件の設定
int SetUpdateConfig(int dnum)
{
    int ret;
    int i;
    DASAMPLREQ    SmplConfig;    // Output conditions setting structure

    // 連続出力条件の取得.
    ret = DaGetSamplingConfig(dnum, &SmplConfig);
    if(ret){
        printf("DaGetSamplingConfig error: ret=%Xh\n", ret);
        return -1;
    }

    SmplConfig.ulChCount = 6;
    for (i = 0; i < SmplConfig.ulChCount; i++) {
        SmplConfig.SmplChReq[i].ulChNo = i+1;
        SmplConfig.SmplChReq[i].ulRange = DA_10V;
    }

    // 連続出力条件の設定
    ret = DaSetSamplingConfig(dnum, &SmplConfig);
    if(ret){
        printf("DaSetSamplingConfig error: ret=%Xh\n", ret);
        return -1;
    }

    return 0;
}
```

```
// 連続出力データの設定
int SetUpdateData(int dnum, unsigned long ulSmplBufferSize)
{
    unsigned long i;

    // 連続出力データの生成
    for (i = 0; i < ulSmplBufferSize; i++) {
        if (i%2) {
            wSmplData[i][0]=0xFFFF;
            wSmplData[i][1]=0x0000;
            wSmplData[i][2]=0xFFFF;
            wSmplData[i][3]=0x0000;
            wSmplData[i][4]=0xFFFF;
            wSmplData[i][5]=0x0000;
        } else {
            wSmplData[i][0]=0x0000;
            wSmplData[i][1]=0xFFFF;
            wSmplData[i][2]=0x0000;
            wSmplData[i][3]=0xFFFF;
            wSmplData[i][4]=0x0000;
            wSmplData[i][5]=0xFFFF;
        }
    }
}

// 連続出力データの設定
ret = DaSetSamplingData(dnum, wSmplData, ulSmplBufferSize);
if(ret){
    printf("DaSetSamplingData error: ret=%Xh¥n", ret);
    return -1;
}

return 0;
}

// メイン
int main(void)
{
    int    ret;
    int    dnum = 1;
    int    i;
    unsigned long    ulSmplBufferSize;// Output buffer size

    // デバイスのオープン
    ret = DaOpen(dnum);
    if(ret){
        printf("Open error: ret=%Xh¥n", ret);
        return -1;
    }

    // バッファ数とコールバック関数の設定
    ulSmplBufferSize = 1024;
```

```
ret = DaSetBoardConfig(dnum, ulSmplBufferSize, NULL, event_proc, 0);
if (ret) {
    printf("DaSetBoardConfig error: ret=%Xh¥n", ret);
    DaClose(dnum);
    return -1;
}

// 連続出力条件の設定
ret = SetUpdateConfig(dnum);
if (ret) {
    DaClose(dnum);
    return -1;
}

// 連続出力データの設定
ret = SetUpdateData(dnum, ulSmplBufferSize);
if (ret) {
    DaClose(dnum);
    return -1;
}

flag = 1;

// 連続出力の開始
ret = DaStartSampling(dnum, FLAG_ASYNC);
if (ret){
    printf("DaStartSampling error: ret=%Xh¥n", ret);
    DaClose(dnum);
    return -1;
}

// コールバック関数がコールされるまで待機
while(flag);

// デバイスのクローズ
DaClose( dnum );

return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
INCLUDE = /usr/X11R6/include
LIB = /usr/X11R6/lib
CC = gcc

all: daevent

daevent: daevent.o
$(CC) daevent.o -o daevent -lgpg3300 -lpthread

daevent.o: daevent.c
$(CC) -Wall -c daevent.c -o daevent.o

clean:
rm -f *.o ¥##* *~ daevent
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「daevent.o」と実行ファイル「daevent」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./daevent
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
The sampling is successfully completed
```


Step3 プログラムの解説

連続出力の基本的な処理については、「6.4.2 連続出力」で解説しているため省略します。

ここでは割り込み機能を使用するために追加したソースコード部分について説明します。

まず、割り込み機能を使用するためには、DaSetBoardConfig関数でコールバック関数を登録する必要があります。

コールバック関数とは、割り込みが発生した場合に呼び出される関数となります。

DaSetBoardConfig関数の書式は下記になります。

```
int DaSetBoardConfig(  
    int          nDevice,          /* デバイス番号 */  
    unsigned long ulSmplBufferSize, /* バッファサイズ */  
    void*        pReserved,        /* 予約 */  
    PLPDACALLBACK pCallBackProc,    /* コールバック関数 */  
    int          dwUser            /* ユーザデータ */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。
ulSmplBufferSize	バッファサイズ
pReserved	予約
pCallBackProc	コールバック関数
dwUser	ユーザデータ

コールバック関数の登録としては、第三引数にて、呼び出したいコールバック関数を指定します。

プログラムでは、「event_proc」という関数を呼び出すように設定しています。

コールバック関数の書式は、下記である必要があります。

```
void CallbackProc(  
    int          uData /* ユーザ・データ */  
);
```

ユーザデータは、AdSetBoardConfig関数の第四引数で指定します。

ユーザデータは、コールバック関数の第一引数に値がそのまま渡されます。

こういった場合に使用するかは、複数のデバイスで割り込みを登録した場合、どのデバイスに対する割り込みかを判別するときに使用します。

本プログラムで、サンプリングが終了した場合に、コールバック関数が呼び出されるため、コールバック関数内で、「The sampling is successfully completed.」という文字を出力し、サンプリングを終了しているかのフラグとして使用しているグローバル変数のflagを0にしています。

本プログラムでは、DaStartSampling関数で非同期サンプリングとして実行し、while文でコールバック関数がコールされるまで待機するように作成しています。

第7章 シリアル通信

7.1 シリアル通信の概要

コンピュータや計測器等の装置との間で、データをやり取りする方法の1つである調歩同期通信について概要を説明します。

この調歩同期通信とは、「シリアル通信」と呼ばれる通信方式で、データをやり取りするために用いる同期方式の一種です。

コンピュータで、モデムなどを使って通信を行う場合、「シリアルポート」と呼ばれるインタフェースの口に、「シリアルケーブル」と呼ばれるケーブルを挿して、機器と繋ぐと思います。

こうして通信を行う時、コンピュータと通信機器の間では、データのやり取りが行われています。このやり取りは、実は「調歩同期通信」と呼ばれる方式でデータのやり取りが行われています。

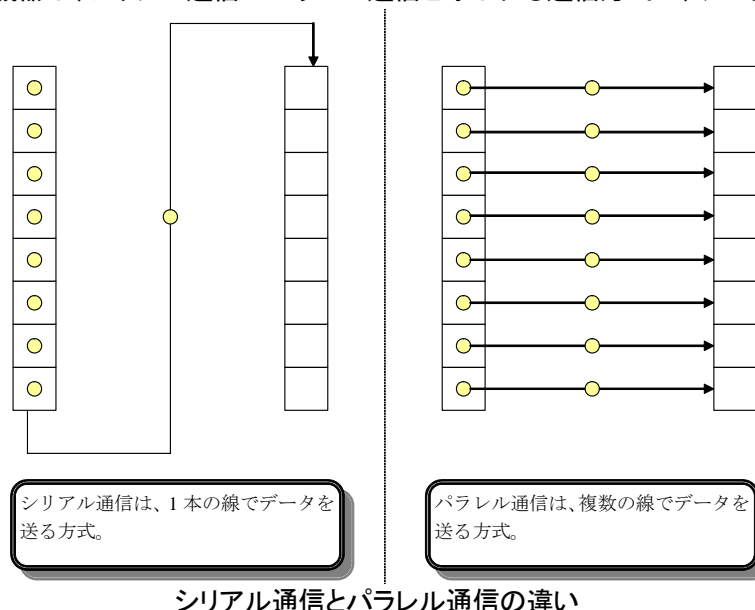
弊社では、この調歩同期通信が行えるインタフェース製品を提供しております。

このインタフェース製品を使用することで、お客様は、複数の機器とデータのやり取りが行えるようになります。

7.1.1 シリアル通信

シリアル通信とは、1本の線を使ってデータをやり取りする通信方式です。対語として、パラレル通信があり、これは複数の線を使ってデータをやり取りする通信方式です。

コンピュータに接続する機器は、シリアル通信かパラレル通信と呼ばれる通信方式で、データをやり取りします。



一般に、シリアル通信方式を採用した通信インタフェースは、他の通信インタフェースに比べ、ケーブル敷設等のコストが安価であることが多いです。逆に、パラレル通信方式を採用した通信インタフェースは、高価であることが多いですが、一度に複数のデータを送ることが可能なため、大量のデータを高速に送るのに向いていると言われています。しかし最近では、高速なシリアル通信方式が開発され、このようなことは一概には言えなくなってきました。シリアル通信方式の代表的なものとしては、RS-232C、RS-485、USB、Ethernet等が挙げられます。この内、調歩同期通信をサポートする通信方式は、RS-232CとRS-485です。

7.1.2 RS-232CとRS-485

RS-232Cとは通信インタフェースの規格の一つです。電氣的仕様、信号線の種類、機能、特性等が規定されています。

この規格は、米国電子工業会(EIA = Electronic Industries Association)が定めました。

なお、RS-232Cという呼び方は正式なものではなく、ANSI/TIA/EIA-232-Fと言います。日本規格では、JIS X5101(旧名:JIS C6361)が、これにあたります。しかし、今でもRS-232Cと呼ばれることが多いです。

コンピュータで、シリアルポートと一般に呼ばれているものは、このRS-232Cを通信インタフェースに採用したシリアル通信を指しています。

RS-485も同じくEIAが規定した規格です。RS-422(ANSI/EIA/TIA-422-B)という規格があり、その上位に相当する規格です。

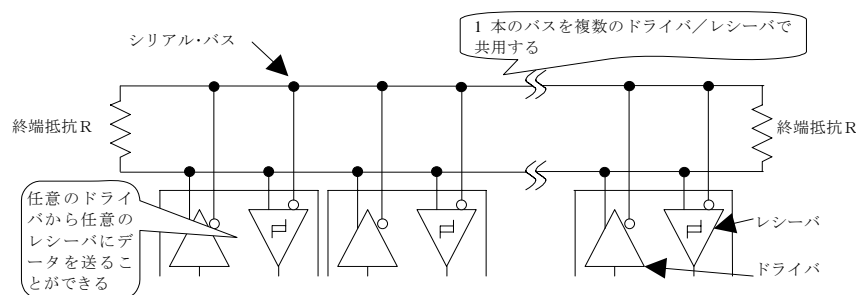
RS-232Cに比べ、以下の特長を有しています。

- ・長距離の通信が可能

RS-232C は最大 15m まで、RS-485 では最大 1.2km まで通信可能です。

- ・1 対 n 接続が可能

RS-232C は 1 対 1 の通信しかできませんが、RS-485 は 1 対 31 までの多点間(マルチドロップ)の通信が行えます。下図に示すように、1 本のシリアル・バスに複数のドライバ/レシーバを接続して、任意のドライバ、レシーバ間でデータの送信を行います。



ドライバレシーバ間でのデータ転送

この接続では複数のドライバが動作するとバス上でデータが衝突しますので、送信を行うときのみドライバをデータラインに接続し、送信を行わないときはデータラインから切り離す必要があります。

7.1.3 同期方式

ある機器から別の機器にデータを送る時、2つの機器はタイミングを合わせて、データの受け渡しを行う必要があります。

タイミングを合わせないと、受け手はどの時点が受け取るべきデータなのか判断することができません。このタイミングを合わせることを「同期を取る」と言います。

同期の方法は千差万別です。RS-232CやRS-485の通信インタフェースで使用される同期方法は、大きく分けて、調歩同期、キャラクタ同期、フラグ同期が挙げられます。

調歩同期は、1キャラクタのデータを送るごとに、データの先頭にスタートビット、末尾にストップビットと呼ばれる制御ビットを挿入して同期を取る方式です。

受け手側では、この特定のビットを検出することで、1キャラクタの受信データの始まりと終わりを認識します。

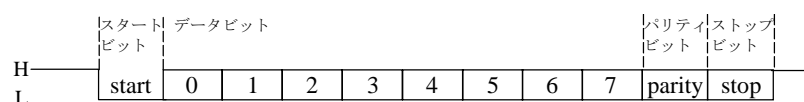
キャラクタ同期は、ブロックと呼ばれる複数キャラクタの集まりの先頭に、特定のコードを付加して同期を取る方式です。

フラグ同期は、フラグと呼ばれる特定のビット列で、フレームと呼ばれるデータ列を挟んで同期を取る方式です。本書では、この内、調歩同期を使用する通信制御について解説します。

7.1.4 調歩同期通信

調歩同期は、通信の効率という面から見ると、キャラクタごとにスタートビット、ストップビットを挿入しなくてはならないため、通信効率率は、他の同期方式に比べて良くありません。

ただしデータに同期したクロックを送信する必要がなく、装置が簡略化できるため、現在のコンピュータ通信で広く普及している通信方式です。



調歩同期通信のデータ構成

データがやりとりされていないとき(アイドル時)、データを送る信号線は、High状態 即ち”1”になっているため、最初のデータが”1”の場合には、アイドル時との区別が付きません。

そこで、データの送信開始を受信装置に知らせるために、まずLow状態 即ち”0”を1ビット送信します。この最初に送信する”0”がスタートビットです。

スタートビットに続いて送信するビット列をデータビットと呼びます。

送信側は、まずスタートビットを送り、次に1データを最下位ビット(LSB)から送信します。

最後のデータビットを送り終わるとデータラインを再び”1”(High状態)に戻し、次のデータが連続している場合は、一定時間待ってから次のスタートビットを送信します。

この最後に送信する”1”がストップビットです。ストップビットの長さは、1 / 1.5 / 2ビットの中から選択できます(通信コントローラによっては選択できない長さもあります)。

7.1.5 データ長

データ長は、スタートビットとストップビットの間のパリティビットを除いた、データビット数のことです。

コンピュータ間の通信では、7ビットまたは8ビットがよく使用されます。

一般的なデータ通信は、送信するキャラクタをキャラクタコードで表し、1キャラクタずつ送ります。キャラクタとキャラクタコードの対応は、キャラクタ符号規格で定められています。昔は5ビット符号もありましたが、現在は7ビットまたは8ビット符号(ASCII/JIS符号)が使用されます。

バイナリデータを送信する場合は、データ長は8ビットにする必要があります。

7.1.6 パリティビット

パリティビットは、通信ではあるデータを送信するとき、データが正確に送られたかどうか検査するためのものです。

例えばASCII 7ビットコードの”a”は16進数で61h、2進数で1100001bとなります。

ここでパリティチェックを偶数パリティに設定した場合、送信側は、転送する”1”の総数が偶数個になるようにします。

この例では”1”の数が3個ですので、8ビット目を”1”にして、11100001bとなる8ビットのデータ列を作り、”1”の数を4個(偶数)にして転送します。

受信側では、”1”の数を数え、偶数であるか確認します。もし、”1”の数が奇数の場合、データの受信に失敗したと判断できます。

このようにデータビットの後にパリティ情報として付加されるビットを、パリティビットと呼びます。

パリティには偶数と奇数があり、奇数パリティの場合は、転送する”1”の数が奇数になるようにパリティビットを付加します。

7.1.7 通信速度(bps)とスループット

シリアル通信では、データを送る速さの指標として、“bps”がよく使われます。

“bps”は1秒間に送れるビット数を表す単位です。SI単位系の原則からいえば“bit/s”と書くべきですが、“bps”を使うのが一般的です。

通信速度が9600bpsの時、1ビットの時間幅が1/9600 sということを表し、

この速度でデータビットを連続して送ることができれば、9600ビット/秒の実効通信速度(スループット)が得られます。

しかし、調歩同期通信の場合、1キャラクタのデータビットには最低2ビット(1ビットのスタートビットと、1ビットのストップビット)が付加されます。

そのため8ビット送るのに少なくとも10ビット分の時間がかかることから、キャラクタ間が隙間なく連続して送ったとしても、実効通信速度は7680bpsしか得られません。

7.1.8 フロー制御

高速通信の際、通信機器のオーバーラン(受信処理をできない時に、データが損失すること)を防ぐために データ受信の準備ができるまで通信の流れを止めておくような制御のことです。

フロー制御にはハードウェアとソフトウェアの 2 つのタイプがあります。

ハードウェアフロー制御

通常の信号線とは別にフロー制御専用の信号線を用意する方式。

制御専用の信号線が必要なため設計は複雑になりますが、ソフトウェアフロー制御と比べ、送受信するデータに制御データを埋め込まずに済むため、転送の効率がよいという利点があります。

ソフトウェアフロー制御

通常の信号線で送受信するデータ列の中に制御データを埋め込む方式

ハードウェアフロー制御と比べ、制御専用の信号線が必要無いため設計は単純になりますが、制御データを埋め込む分だけデータ転送の効率は低下します。

7.1.9 その他の用語解説

その他、調歩同期通信を行う際に、出てくる用語について解説します。

用 語	解 説
シリアルポート	<p>シリアル通信を行う際、送受信を行う口を、一般に「シリアルポート」と呼びます。</p> <p>個々のシリアルポートは、ドライバソフトウェアから見た場合、独立しており、個別に取り扱うことが可能です。</p> <p>カタログ等の仕様では、チャンネル数が 2 チャンネル,4 チャンネルと記載していますが、これを 2 ポート,4 ポートと呼ぶ書籍等もあります。</p> <p>本書では、通信を行う 1 つの口を特に、シリアルポートと称しています。</p>
全二重/半二重	<p>全二重通信とは両方向通信が可能で、かつ送受信が同時に行える通信方式です。</p> <p>半二重通信における送受信切り替えの時間的ロスがなく、データの送信効率を高めることができます。</p> <p>半二重通信とは両方向通信が可能で、かつ送受信を同時に行えないものをいいます。送受信が同時に行えないため、送受信の切り替えを行う必要があります。</p> <p>RS-232C インタフェースを搭載した半二重モデムと接続する場合、制御信号(一般には RS 信号)により送受信の切り替えを行います。</p> <p>RS-485 マルチドロップ接続の場合、ドライバ・レシーバの接続変更により、送受信を切り替えます。</p>

7.1.10 調歩同期通信を行うにあたってのアドバイス

調歩同期通信I/Oモジュールを使って調歩同期通信を行う際、知っておくための事項をご紹介します。

項 目	内 容
RS-232C と RS-485 のどちらを使うべきか？	<p>RS-232C と RS-485 のどちらを通信方式として採用すべきかについては、接続相手、使用用途、環境等を鑑みる必要があります。</p> <p>例えば、計測器や接続相手が決まっている時は、相手の通信インタフェースに合わせる必要があります。</p> <p>通信インタフェースを自由に選択できる場合は、使用用途で選択した方が良いと思われます。</p> <ul style="list-style-type: none">・ コンピュータのシリアルポートの口を増やしたいのならば、RS-232C をお勧めします。・ 接続相手との距離が 15m を越えるようならば、RS-485 を選択してください。・ 1 対 1 の接続でなく、1 対 2～1 対 31 の接続を行いたいならば、RS-485 を選択してください。・ 通信速度が 1Mbps を越えるような場合、RS-485 を選択してください。
家電品店で売っているシリアルケーブルは使用できるか？	<p>家電品店で一般にシリアルケーブルとして売られているものは、RS-232C 用のものがほとんどです。</p> <p>コンピュータとモデムを接続する場合は、ほとんどの場合、ストレートタイプのケーブルを使用します。</p> <p>機器同士を直接接続する場合は、リバース(クロス)タイプのケーブルを使用します。</p> <p>ただし、購入前に結線図をよく確認して、使用目的と合致するケーブルを選択してください。</p>
送受信バッファメモリとは、何のためにあるのか？	<p>シリアル通信でよく問題となるのが、データの受け取り側が、今あるデータを取り出す前に、次のデータが送られる状況が発生することです。</p> <p>これは、ベルトコンベアに載せられて送られてくる物品を、捌くのが追いつかなくなる状況に似ています。これを俗に「取りこぼしが発生した」と言います。</p> <p>取りこぼしが発生する状況としては、幾つか原因が考えられます。代表的な例としては、CPU の処理速度が遅いために取りこぼしたり、OS がリアルタイムに受け取り処理に入らないため、取りこぼすといったものです。</p> <p>受け取ったデータを内部で溜め込んでおき、一括して取り出すことができれば、取りこぼしを回避できると思われます。</p> <p>送受信バッファメモリは、この送受信するデータを溜め込む領域です。このサイズ</p>

項 目	内 容
	が多ければ多い程、受け取り処理に入るまでの反応が遅い OS でも、より確実にデータを受け取ることが可能となります。
標準 COM ポート互換とは、どういう意味か？	PC/AT 互換機の PC に内蔵されているシリアルポートは、標準 COM ポートと呼ばれます。 シリアル通信 I/O モジュールの中で、PC/AT 互換機のシリアルポートの通信コントローラ 16550 と互換の通信コントローラを搭載した I/O モジュールを、標準 COM ポート互換通信 I/O モジュールと呼んでいます。 また Windows のシリアルポート(COM ポート)と API 互換であるという意味で使用されることもあります。
通信を行うために、最低何本の線が必要か？	全二重の場合、送信ライン,受信ライン,グラウンドの 3 本のラインを接続すれば、最低限の通信は行えます。 ハードウェアフロー制御等通信制御を行う場合、制御信号を別途接続する必要があります。 接続する制御信号は、接続相手の通信機器の仕様により変わります。 詳細については、ハードウェアマニュアルを参照してください。
通信速度は、1bps ずつ変更することは可能か？	通信に使用するクロックは、I/O モジュールに搭載されている基準クロックを分周して生成されます。 分周するため、ビットレートは全ての整数値に設定することはできません。 次の計算式の結果が整数値になる値が、設定できる通信速度となります。 [基準クロック] _____ [通信速度(bps)] × 16

7.2 対象型式

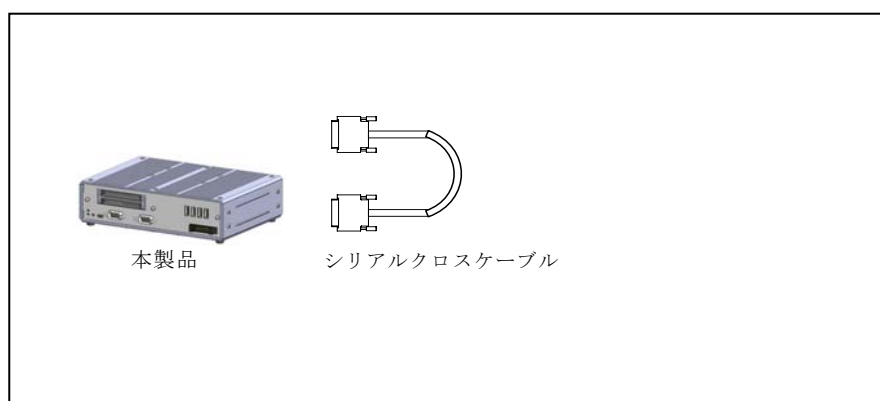
ここでは、実際に弊社製品を用いて、簡単な通信プログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N4007	タフコン CD シリーズ通信モデル
-----------	-------------------

7.3 環境図

ITC-N4007 の「UART CN1」と「UART CN2」を、シリアルクロスケーブルを用いて接続します。



シリアルクロスケーブルの結線図を下記に示します。

信号名	ピン番号		信号名	ピン番号
RD	2	↗	RD	2
SD	3	↘	SD	3
ER	4	↗	ER	4
SG	5	↘	SG	5
DR	6	↗	DR	6
RS	7	↘	RS	7
CS	8	↗	CS	8

7.4 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

7.4.1 送信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「Send.c」として保存してください。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <linux/serial.h>
#include <asm/ioctls.h>
#define BAUDRATE B38400

char Send_Buf[255];

int main(void)
{
    int fd;
    int Ret;
    struct termios Newtio;
    int Length;

    sprintf(Send_Buf, "0123456789ABCDEFGH¥n");

    // シリアルポートのオープン
    fd = open( "/dev/ttyG0", O_RDWR | O_NOCTTY );
    if ( fd < 0 ) { printf( "Open fd Error ! Exit !¥n " ); exit( -1 ); }

    bzero( &Newtio, sizeof( Newtio ) );
    Newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
    Newtio.c_iflag = IGNPAR | ICRNL;
    Newtio.c_oflag = 0;
    Newtio.c_lflag = ICANON;
```

```
// バッファのクリア
Ret = tcflush( fd, TCIOFLUSH );
if ( Ret != 0 ){
    printf( " tcflush Error %n" );
    close( fd );
    exit( -1 );
}
// 通信パラメータの設定
Ret = tcsetattr( fd, TCSANOW , &Newtio );
if (Ret != 0 ){
    printf( " tcsetattr Error %n" );
    close( fd );
    exit( -1 );
}

// データの送信
Length = strlen( Send_Buf );
if ( ( Ret = write( fd, Send_Buf, Length ) ) == -1 ){
    printf( " Write Return Error %n" );
    close( fd );
    exit( -1 );
}

printf("Transmission was completed.%n");

// ポートのクローズ
close( fd );
}
```

次に Makefile を作成します。

エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = Send

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「Send.o」と実行ファイル「Send」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./Send
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
Transmission was completed.
```

今回作成したプログラムはシリアルデータの送信を行うプログラムです。

シリアルデータの送信を行うには、write 関数を使用します。

ITC-N4007 の UART CN1 から「0123456789ABCDEFGH」というデータを送信します。

送信が正常に行われたかどうかは「7.4.2 受信」で作成するプログラムを実行して確認出来ます。

Step3 プログラムの解説

まず、オープンシステムコールでシリアルポートをオープンします。

第一引数に「/dev/ttyG0」という値を指定しています。この値は、ITC-N4007 の UART CN1 に割当てられているデバイスノードです。※1

この値を指定し、実行したオープンシステムコールの戻り値を用いて ITC-N4007 の UART CN1 の制御を行います。

次に tcflush 関数を使用して受信バッファ、送信バッファをクリアします。

次に通信パラメータの設定を行います。tcsetattr 関数を用いることで設定を行うことができます。

今回作成したプログラムでは以下の設定を行っております。

- ・ボーレート : 38400bps
- ・データ長 : 8bit
- ・ストップビット : 1bit
- ・パリティチェック: なし
- ・フロー制御 : ハードウェアフロー制御 ※2
- ・カノニカルモード: 有効 ※3

次に write 関数を用いてデータの送信を行います。

第一引数にはオープンシステムコール関数で取得したファイルディスクリプタ、第二引数には送信するデータを格納した配列、第三引数には送信するデータサイズを指定します。

最後にオープンしたシリアルポートのクローズを行います。

オープンしたシリアルポートは終了時には必ずクローズを行ってください。

※1 シリアルポートのデバイスノード「/dev/ttyGxx」はデバイスの認識した順番に番号が割り振られていきます。そのため、デバイスごとの一意な名前ではありません。

デバイス名を確認するにはコンソール上で「cat /proc/tty/driver/cp4161」と入力します。

入力後、デバイスノード、製品型式が表示されます。

```
#cat /proc/tty/driver/cp4161
ttyG0: PCI/LPC-466120(P)(bid=0h) CH1 [9600bps] tx:0 rx:0
ttyG1: PCI/LPC-466120(P)(bid=0h) CH2 [9600bps] tx:0 rx:0
```

※ITC-N4007 の UART CN1, CN2 はそれぞれ PCI-466120 CH1, CH2 として表示されます。

前面下側には 16550 互換の UART が 2 ポート搭載されています。

デバイスノードはそれぞれ、/dev/ttyS0、/dev/ttyS1 となります。

※2 フロー制御については「7.1.8 フロー制御」のハードウェアフロー制御を参照してください。

© 2013 Interface Corporation. All rights reserved.



信号線(RS、CS)が配線されていない場合正常に送受信を行うことができません。
ケーブルの結線については「7.3 環境図」を参照してください。

※3 カノニカルモードが有効な場合、送受信が行単位での動作となります。

送信の場合、行区切り文字を write 関数に渡すまでデータの送信を行いません。

受信の場合、行区切り文字を受け取るまで read 関数の処理から抜けません。

7.4.2 受信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「Receive.c」として保存してください。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <linux/serial.h>
#include <asm/ioctls.h>
#define BAUDRATE B38400

char Recv_Buf[255];

int main(void)
{
    int fd;
    int Ret;
    struct termios Newtio;

    // ポートのオープン
    fd = open("/dev/ttyG1", O_RDWR | O_NOCTTY);
    if ( fd < 0 ) {
        printf( "Open fd Error ! Exit !%n " );
        return -1;
    }

    bzero( &Newtio, sizeof( Newtio ) );
    Newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
    Newtio.c_iflag = IGNPAR | ICRNL;
    Newtio.c_oflag = 0;
    Newtio.c_lflag = ICANON;

    // バッファのクリア
    Ret = tcflush( fd, TCIOFLUSH );
    if (Ret != 0 ){
        printf( " tcsetattr Error %n " );
        close( fd );
        exit( -1 );
    }

    // 通信パラメータの設定
    Ret = tcsetattr( fd, TCSANOW, &Newtio );
```

```
if (Ret != 0){
    printf( " tcsetattr Error ¥n" );
    close( fd );
    exit( -1 );
}

// 受信データを取得
if ( ( Ret = read( fd, Recv_Buf, 255 ) ) == -1 ) {
    printf( "Read Return Error ¥n" );
    close( fd );
    return -1;
}

// 受信データの表示
Recv_Buf[Ret] = 0;
printf( "Receive Characters¥n%s ",Recv_Buf );

// ポートのクローズ
close( fd );
}
```

次に Makefile を作成します。エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = Receive

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「Receive.o」と実行ファイル「Receive」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./Receive
```

実行すると何も表示されず処理が停止します。

この状態で「送信」で作成したプログラムを実行してください。

「送信」のプログラム実行後、正常に処理が終了すると下記が表示されます。

```
Receive Characters  
0123456789ABCDEFGH
```

今回作成したプログラムはシリアルデータの受信を行うプログラムです。

データの受信を行うには、read 関数を使用します。

ITC-N4007 の UART CN2 でデータを受信し、受信したデータを表示します。

Step3 プログラムの解説

まず、オープンシステムコールでシリアルポートをオープンします。

第一引数に「/dev/ttyG1」という値を指定しています。この値は、ITC-N4007 の UART CN2 に割り当てられているデバイスノードです。※1

この値を指定し、実行したオープンシステムコールの戻り値を用いて ITC-N4007 の UART CN2 の制御を行います。次に tcflush 関数を使用して受信バッファ、送信バッファをクリアします。

次に通信パラメータの設定を行います。tcsetattr 関数を用いることで設定を行うことができます。

今回作成したプログラムでは以下の設定を行っております。

- ・ボーレート : 38400bps
- ・データ長 : 8bit
- ・ストップビット : 1bit
- ・パリティチェック: なし
- ・フロー制御 : ハードウェアフロー制御 ※2
- ・カノニカルモード: 有効 ※3

次に read 関数を用いてデータの送信を行います。

第一引数にはオープンシステムコール関数で取得したファイルディスクリプタ、第二引数には受信したデータを格納する配列のポインタ、第三引数には受信する最大データサイズを指定します。関数の戻り値には実際に受信できたデータサイズが返されます。

最後にオープンしたシリアルポートのクローズを行います。

オープンしたシリアルポートは終了時には必ずクローズを行ってください。

※1 シリアルポートのデバイスノード「/dev/ttyGxx」はデバイスの認識した順番に番号が割り振られていきます。そのため、デバイスごとの一意な名前ではありません。

デバイス名を確認するにはコンソール上で「cat /proc/tty/driver/cp4161」と入力します。

入力後、デバイスノード、製品型式が表示されます。

```
#cat /proc/tty/driver/cp4161
ttyG0: PCI/LPC-466120(P)(bid=0h) CH1 [9600bps] tx:0 rx:0
ttyG1: PCI/LPC-466120(P)(bid=0h) CH2 [9600bps] tx:0 rx:0
```

※ITC-N4007 の UART CN1, CN2 はそれぞれ PCI-466120 CH1, CH2 として表示されます。

前面下側には 16550 互換の UART が 2 ポート搭載されています。

デバイスノードはそれぞれ、/dev/ttyS0、/dev/ttyS1 となります。

※2 フロー制御については「7.1.8 フロー制御」のハードウェアフロー制御を参照してください。

信号線(RS、CS)が配線されていない場合正常に送受信を行うことができません。

ケーブルの結線については「7.3 環境図」を参照してください。

※3 カノニカルモードが有効な場合、送受信が行単位での動作となります。

送信の場合、行区切り文字を write 関数に渡すまでデータの送信を行いません。

受信の場合、行区切り文字を受け取るまで read 関数の処理から抜けません。

第8章 HDLC

8.1 HDLCの概要

HDLC (High level Data Link Control procedure:ハイレベルデータリンク制御手順)とは、データ伝送制御手順の一つです。信頼性が高く、効率良くデータを送ることができます。

HDLCには、下記のような特長があります。

●効率の良い伝送

HDLCでは、フレームと呼ばれるデータのかたまりを、両方向同時伝送(全二重)できます。さらに、送信側は、受信側からの応答を待たずに、連続してデータを伝送できるため、効率良くデータを伝送することができます。

●高い信頼性

CRC(Cyclic Redundancy Check)方式による誤り検出を、伝送するフレーム内の全ての情報に対して行うため、非常に高い信頼性を誇ります。

★CRC

巡回符号方式と呼ばれる誤り検出方式のこと。送るデータをあらかじめ定められた生成多項式で割り、余りをデータのあとに付加して送信します。受信側でも同じ生成多項式で割って、余りがなければ伝送データは正しいとします。

●自由な長さ・内容のデータ伝送

HDLCでは、伝送制御上の制限を受けず、自由なビットの組み合わせのデータを、自由な長さで伝送できます。

このような特長から、HDLCはコンピュータ間のデータ通信等、幅広く用いられます。



8.1.1 局とコマンド/レスポンス

HDLC ではデータ伝送する端末のことを局と呼びます。局は、その役割により、下記の3つが定義されています。

●1次局

通信経路の確保(『8.1.4 データリンク』を参照してください)し、誤り検出等の制御を行う局です。

1次局から送信されるフレーム(2次局への指示)はコマンドと呼び、1次局が受信するフレーム(2次局からの応答)はレスポンスと呼びます(下図 1次局と2次局)。

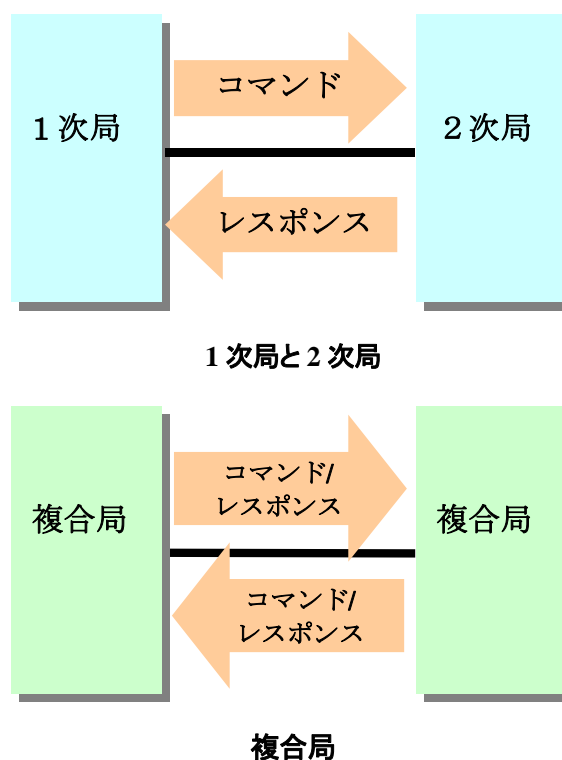
●2次局

1次局からの命令に従い動作する局です。

1次局とは逆に、2次局から送信されるフレーム(1次局への応答)はレスポンス、受信するフレーム(1次局からの指示)はコマンドになります(下図 1次局と2次局)。

●複合局

1次局,2次局を兼ね備える局です(下図 複合局)。

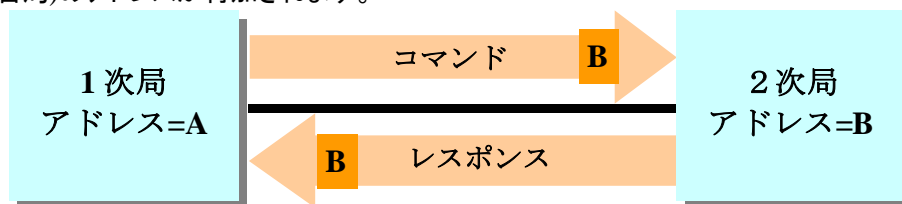


8.1.2 アドレス

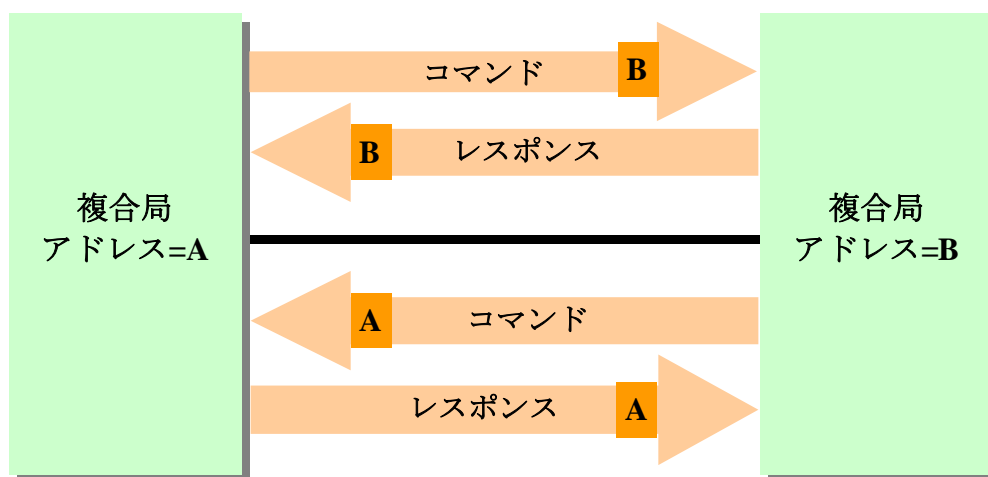
HDLCにおけるコマンドやレスポンス等のフレームには、全て「アドレス」が付加されています。

アドレスを付加することで、どの局に対するコマンドか、どの局からのレスポンスかを判断でき、全二重通信が可能となるのです。

コマンドフレームには、それを受け取る2次局(または複合局)のアドレスを付加し、レスポンスフレームには、それを送信した2次局(または複合局)のアドレスが付加されます。



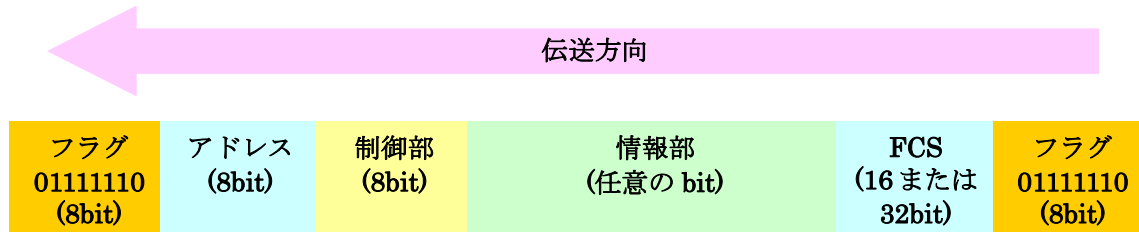
1次局と2次局のアドレス



複合局のアドレス

8.1.3 フレームの構成

HDLCでは、全ての情報を下図に示すようなフレーム単位で伝送します。伝送される順に従って説明します。



フレームの構成

●フラグ

フラグは、フレームの始まりと終わりを示し、フラグ同期の機能を持っています。受信側では、これらのフラグを検出し、フレームの先頭と末尾を識別します。

★フレームの認識

このフラグが検出されなければ、フレームの先頭や末尾が識別できずに正常に通信できなくなります。信号のなまり等には注意する必要があります。

●アドレス

局を識別するためのアドレスです(『8.1.2 アドレス』を参照してください)。

●制御部

接続局に対する命令や、その命令に対する応答を格納します。

●情報部

ここに実際に送信を行うデータを格納します。データはどのようなビットパターンでも構わず、またフィールドの長さに制限はありません。

●FCS(Frame Check Sequence:フレーム検査シーケンス)

「アドレス」、「制御部」、「情報部」の誤り制御のためのビットパターンをCRC方式で格納します。

8.1.4 データリンク

データリンクとは、データの伝送経路のことで、物理的な伝送経路(通信回線等)と論理的な伝送経路(送受信可能な状態)との両方がそろった状態のことを言います。

例えば、データを送信する前に相手側の受信準備状態を確認した時、はじめて「データリンクが設定された」といいます。HDLCのデータリンクには、不平衡型データリンク(Unbalanced Data Link)と平衡型データリンク(Balanced Data Link)の2種類があります。

●不平衡型データリンク

1次局と2次局で構成され、データリンクの設定や障害の復旧等は、全て1次局が行います。

●平衡型データリンク

複合局同士で構成され、データリンクの設定や障害の復旧等は、それぞれの局が対等に行います。

8.1.5 モード

通信拠点が、2次局や複合局の場合には、その状態により3種類の動作モードと、2種類の切断モードおよび初期モードがあります。1次局にモードはありませんが、2次局に対してモードを設定する機能を有しています。

モード一覧

動作モード	正規応答モード(NRM) Normal Response Mode	2次局は1次局から送信許可コマンドを受信した時のみ、1次局に対しレスポンスを送信できる。
	非同期応答モード(ARM) Asynchronous Response Mode	2次局は1次局から送信許可コマンドを受信せず、1次局に対しレスポンスを送信できる。
	非同期平衡モード(ABM) ASynchronous Balanced Mode	複合局は接続先の複合局の許可がなくても、コマンド/レスポンスの送受信が行える。
切断モード	正規切断モード(NDM) Normal Disconnected Mode	不平衡型データリンクでの非動作モード。
	非同期切断モード(ADM) Asynchronous Disconnected Mode	不平衡型または平衡型データリンクでの非動作モード。
初期モード	初期モード(IM) Initialization Mode	1次局または相手の複合局の状態により、データリンクの初期化/再生が行える。

★プロトコルに関して

弊社I/Oモジュールおよびドライバは、フレーム送信に関しては、FCS自動付加以外のデータ編集は行いませんので、アドレス、制御コード等は、アプリケーション側で作成する必要があります。また、受信に関してはFCSの自動検査のみデータ編集を行います。

弊社HDLCL/Oモジュールの特長としては以下のようなものが挙げられます。

- ・自由なフォーマットでHDLCLフレームの送受信が行えます。
- ・FCS付加、チェックを自動的に行います。
- ・一つのフレームで最大16384バイトのデータ送受信が行えます。
- ・I/Oモジュール上に送信バッファを用意してあるため、連続してデータ送信が行えます。
- ・I/Oモジュール上に受信バッファを用意してあるため、アプリケーションプログラムが他の処理を実行していても、データ取りこぼしが発生しません。
- ・I/Oモジュールからのイベント発生時に、ユーザが登録した処理(コールバックルーチン)を実行することができます。

8.2 対象型式

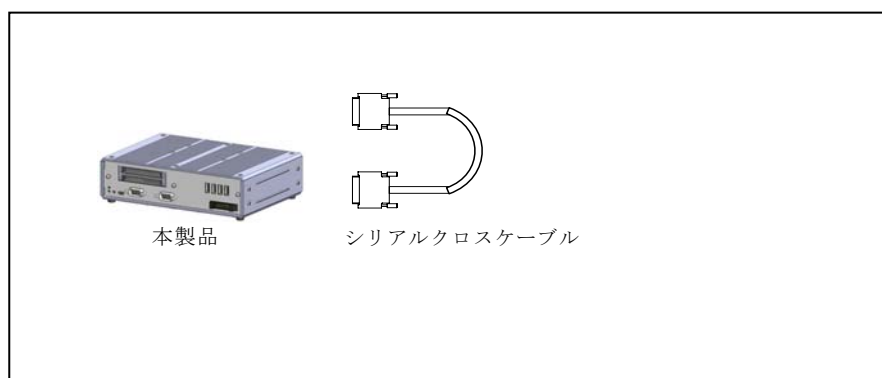
ここでは、実際に弊社製品を用いて、簡単な通信プログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N4007	タフコン CD シリーズ通信モデル
-----------	-------------------

8.3 環境図

ITC-N4007 の「HDLC CN1」と「HDLC CN2」を、シリアルクロスケーブルを用いて接続します。



シリアルクロスケーブルの結線図を下記に示します。



8.4 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

8.4.1 フレーム送信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「FrameSend.c」として保存してください。

```
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include "pcihdlc.h"

int main(void)
{
    int                portNum;
    long               ret;
    unsigned long      frameLength;
    char               sendData[512];
    HDLCPORTINITDATA   sInit;
    HDLCPORTSTATUS     status;

    portNum = HDLC_PORTNO_467X02_0_2;

    memset(&sInit, 0, sizeof(HDLCPORTINITDATA));
    sInit.ulFormat = HDLC_FORMAT_NRZI;
    sInit.ulFcs = HDLC_FCS_16;
    sInit.ulAddressMode = HDLC_ADDRESS_NONE;
    sInit.ulLineMode = HDLC_LINE_FULL;
    sInit.ulTxc = HDLC_SCLK_PTC;
    sInit.ulRxc = HDLC_RCLK_DPLL;
    sInit.ulSourceClock = HDLC_CLOCK_32000000;
    sInit.ulBaudRate = 250000;
    sInit.ulInterface = HDLC_INTERFACE_485;
    sInit.ulTxcMode = HDLC_STOUT_NONE;
    sInit.ulSendTiming = 100 | HDLC_TIME_MICRO_SEC;
    sInit.ulCloseTiming = 100 | HDLC_TIME_MICRO_SEC;
    sInit.lpCallbackProc = NULL;
    sInit.ulUserData = 0;

    // ポートのオープン
    ret = HdlcOpen(portNum, &sInit);
```

```
if(ret != HDLC_ERROR_SUCCESS) {
    printf("The specified device could not be opened.%n");
    exit(0);
}

// フレームの送信
sprintf(sendData,"The quick brown fox jumps over the lazy dog.");
frameLength = strlen(sendData);
ret = HdlcSendFrame(portNum, sendData, frameLength);
if(ret != HDLC_ERROR_SUCCESS) {
    printf("The Transmission process failed.%n");
    HdlcClose(portNum);
    exit(0);
}

// 送信が完了するまで待つ
do{
    ret = HdlcGetStatus(portNum, &status);
    if(ret != HDLC_ERROR_SUCCESS) {
        printf("Failed to get status.%n");
        HdlcClose(portNum);
        exit(0);
    }
}while(status.ulSendFrame);

printf("Transmission was completed.%n");

// ポートのクローズ
ret = HdlcClose(portNum);
if(ret != HDLC_ERROR_SUCCESS) {
    printf("Failed to close the device.%n");
    exit(0);
}

return 0;
}
```

次に Makefile を作成します。エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = FrameSend
CFLAGS = -lgpg4116

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「FrameSend.o」と実行ファイル「FrameSend」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./FrameSend
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
Transmission was completed.
```

今回作成したプログラムはフレームの送信を行うプログラムです。

フレームの送信には HdlcSendFrame 関数を使用します。

ITC-N4007 の HDLC CN2 から「The quick brown fox jumps over the lazy dog.」というメッセージを送信します。メッセージの送信後、HdlcGetStatus 関数にて送信待ちのフレーム数をポーリングし、メッセージがすべて送信されたことを確認した後、プログラムを終了します。

送信が正常に行われたかどうかは「8.4.2 フレーム受信」で作成するプログラムを実行して確認出来ます。

Step3 プログラムの解説

HDLC製品を制御する前段階として、HDLC製品をオープンするHdlcOpen関数を呼び出しています。
HdlcOpen関数では第一引数にポート番号、第二引数にポート初期化構造体のポインタを指定します。

第一引数のポート番号は製品型式・チャンネル・RSW1設定値により一意に定まるポート番号を指定します。
ITC-N4007のHDLC CN2は「HDLC_PORTNO_467X02_0_2」となります。(値では50)

第二引数のポート初期化構造体では下記の設定を行っています。

- ・符号化フォーマット : NRZI
- ・FCS生成多項式 : ITU-T 16bit
- ・アドレスの扱い : アドレス検出なし
- ・ラインモード : 全二重
- ・送信クロック : 内部クロック
- ・受信クロック : DPLL
- ・ソースクロック : 32MHz
- ・ボーレート : 250000bps
- ・インタフェース : RS-485
- ・送信クロックモード : 常に出力しない
- ・送信前切替時間 : 100 μ s
- ・送信後切替時間 : 100 μ s
- ・コールバックルーチン: 設定なし
- ・ユーザデータ : 設定なし

次に HdlcSendFrame 関数を用いてフレームの送信を行います。

HdlcSendFrame 関数では第一引数にポート番号、第二引数に送信するデータを格納した配列のアドレス、第三引数に送信するデータサイズを指定します。

次に do～while 文を使用して、ステータスをポーリングします。ステータスの取得には HdlcGetStatus 関数を使用します。

HdlcGetStatus 関数では第一引数にポート番号、第二引数にステータスを格納する HDLCPORTSTATUS 構造体のポインタを指定します。

ここでは送信待ちのフレーム数が 0 になるまで(送信が完了するまで)ここで繰り返しポーリングを行います。

ポーリングから抜けると HdlcClose 関数を用いてポートをクローズします。

オープンしたポートは終了時には必ずクローズを行ってください。

★送信の完了を待たずに実行するとどうなるか？

送信待ちのフレーム数が0になる前にHdlcClose関数が呼び出される事になり、送信が途中までしか行われなくなります。
送信が完了したことを確認した後でクローズを行ってください。

8.4.2 フレーム受信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「FrameReceive.c」として保存してください。

```
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include "pcihdlc.h"

int main(void)
{
    int                portNum;
    long               ret;
    unsigned long      frameLength;
    char               receiveBuffer[512];
    HDLCPORTINITDATA   sInit;
    HDLCPORTSTATUS     status;

    portNum = HDLC_PORTNO_467X02_0_1;

    memset(&sInit, 0, sizeof(HDLCPORTINITDATA));
    sInit.ulFormat = HDLC_FORMAT_NRZI;
    sInit.ulFcs = HDLC_FCS_16;
    sInit.ulAddressMode = HDLC_ADDRESS_NONE;
    sInit.ulLineMode = HDLC_LINE_FULL;
    sInit.ulTxc = HDLC_SCLK_PTC;
    sInit.ulRxc = HDLC_RCLK_DPLL;
    sInit.ulSourceClock = HDLC_CLOCK_32000000;
    sInit.ulBaudRate = 250000;
    sInit.ulInterface = HDLC_INTERFACE_485;
    sInit.ulTxcMode = HDLC_STOUT_NONE;
    sInit.ulSendTiming = 100 | HDLC_TIME_MICRO_SEC;
    sInit.ulCloseTiming = 100 | HDLC_TIME_MICRO_SEC;
    sInit.lpCallbackProc = NULL;
    sInit.ulUserData = 0;

    // ポートのオープン
    ret = HdlcOpen(portNum, &sInit);
    if(ret != HDLC_ERROR_SUCCESS) {
        printf("The specified device could not be opened.%n");
        exit(0);
    }

    // フレームが受信されるまで待つ
    do{
        ret = HdlcGetStatus(portNum, &status);
        if(ret != HDLC_ERROR_SUCCESS) {
            printf("Failed to get status.%n");
        }
    }
```

```
        HdlcClose(portNum);
        exit(0);
    }
}while(status.ulReceiveFrame == 0);

// フレームの受信
ret = HdlcReceiveFrame(portNum, receiveBuffer, &frameLength);
if(ret != HDLC_ERROR_SUCCESS) {
    printf("Failed to receive frame.¥n");
    HdlcClose(portNum);
    exit(0);
}

// 受信したフレームの表示
receiveBuffer[frameLength] = ¥0';
printf("%s¥n", receiveBuffer);

// ポートのクローズ
ret = HdlcClose(portNum);
if(ret != HDLC_ERROR_SUCCESS) {
    printf("Failed to close the device.¥n");
    exit(0);
}

return 0;
}
```

次に Makefile を作成します。エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = FrameReceive
CFLAGS = -lgpg4116

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
#make
```

コンパイルが成功すると、オブジェクトファイル「FrameReceive.o」と実行ファイル「FrameReceive」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
# ./FrameReceive
```

プログラムを実行すると何も表示されず処理が停止します。。
この状態で「フレーム送信」で作成したプログラムを実行してください。
「8.4.1 フレーム送信フレーム送信」のプログラム実行後、正常に処理が終了すると下記が表示されます。

```
The quick brown fox jumps over the lazy dog.
```

今回作成したプログラムはフレームの受信を行うプログラムです。

フレームの受信を行うには HdlcReceiveFrame 関数を使用します。

ITC-N4007 の HDLC CN1 がメッセージを受信したらメッセージを取得し、そのメッセージを表示します。

メッセージが受信されるまで HdlcGetStatus 関数にて受信フレーム数をポーリングしながら受信処理の実行を待ちます。フレームが受信されていることを確認した後、受信処理を行います。

Step3 プログラムの解説

HDLC製品を制御する前段階として、HDLC製品をオープンするHdlcOpen関数を呼び出しています。
HdlcOpen関数では第一引数にポート番号、第二引数にポート初期化構造体のポインタを指定します。

第一引数のポート番号は製品型式・チャンネル・RSW1設定値により一意に定まるポート番号を指定します。
ITC-N4007のHDLC CN1は「HDLC_PORTNO_467X02_0_1」となります。(値では49)

第二引数のポート初期化構造体では下記の設定を行っています。

- ・符号化フォーマット : NRZI
- ・FCS生成多項式 : ITU-T 16bit
- ・アドレスの扱い : アドレス検出なし
- ・ラインモード : 全二重
- ・送信クロック : 内部クロック
- ・受信クロック : DPLL
- ・ソースクロック : 32MHz
- ・ボーレート : 250000bps
- ・インタフェース : RS-485
- ・送信クロックモード : 常に出ししない
- ・送信前切替時間 : 100 μ s
- ・送信後切替時間 : 100 μ s
- ・コールバックルーチン: 設定なし
- ・ユーザデータ : 設定なし

次に do～while 文を使用して、ステータスをポーリングします。ステータスの取得には HdlcGetStatus 関数を使用します。

HdlcGetStatus 関数では第一引数にポート番号、第二引数にステータスを格納する HDLCPORTSTATUS 構造体のポインタを指定します。

ここでは受信しているフレーム数に 0 でなくなるまで(フレームを受信するまで)ここで繰り返しポーリングを行います。

次に HdlcReceiveFrame 関数を用いてフレームの受信を行います。

HdlcReceiveFrame 関数では第一引数にポート番号、第二引数に受信データを格納する配列のアドレス、第三引数に受信したデータサイズを格納する変数のポインタを指定します。

HdlcReceiveFrame 関数実行後、受信したデータを printf 関数で表示し、HdlcClose 関数を用いてポートをクローズします。

オープンしたポートは終了時には必ずクローズを行ってください。

8.4.3 割り込み処理

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「IntFrameReceive.c」として保存してください。

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "pcihdlc.h"

char SendBuffer[512];
char ReceiveBuffer[MAX_FRAME_LENGTH];
unsigned long ReceivedCount = 0;

// コールバックルーチン
void ReceiveHandler(unsigned long EventMask, unsigned long UserData)
{
    unsigned long FrameCount;
    unsigned long FrameLength;
    unsigned long i;
    long ret;
    int nPort;

    // 受信割り込み要因かどうかの確認
    if (EventMask != 1) return;

    // フレームの受信、受信したフレームの表示
    nPort = (int)UserData;
    ret = HdlcGetFrameCount(nPort, &FrameCount);
    if (ret == HDLC_ERROR_SUCCESS) {
        for(i = 0; i < FrameCount; i++) {
            ret = HdlcReceiveFrame(nPort, ReceiveBuffer, &FrameLength);
            if (ret == HDLC_ERROR_SUCCESS) {
                if(FrameLength > 50) {
                    ReceiveBuffer[50] = '\0';
                } else {
                    ReceiveBuffer[FrameLength] = '\0';
                }
                printf("receive data=%s (size=%ld)\n", ReceiveBuffer, FrameLength);
                ReceivedCount++;
            }
        }
    }
}

int main(void)
{
    long ret;
    int nPort;
```

```
int i;
unsigned long FrameLength;
HDLCPORTINITDATA sInit;

nPort = HDLC_PORTNO_467X02_0_1;

memset(&sInit, 0, sizeof(HDLCPORTINITDATA));
sInit.ulFormat = HDLC_FORMAT_NRZI;
sInit.ulFcs = HDLC_FCS_16;
sInit.ulAddressMode = HDLC_ADDRESS_NONE;
sInit.ulLineMode = HDLC_LINE_FULL;
sInit.ulTxc = HDLC_SCLK_PTC;
sInit.ulRxc = HDLC_RCLK_DPLL;
sInit.ulSourceClock = HDLC_CLOCK_32000000;
sInit.ulBaudRate = 250000;
sInit.ulInterface = HDLC_INTERFACE_485;
sInit.ulTxcMode = HDLC_STOUT_NONE;
sInit.ulSendTiming = 100 | HDLC_TIME_MICRO_SEC;
sInit.ulCloseTiming = 100 | HDLC_TIME_MICRO_SEC;
sInit.lpCallbackProc = ReceiveHandler;
sInit.ulUserData = nPort;

// ポートのオープン
ret = HdlcOpen(nPort, &sInit);
if(ret) {
    printf("HdlcOpen Error[%x]¥n", ret);
    return -1;
}

// イベントマスクの設定
ret = HdlcSetEventMask(nPort, 1, 0);
if (ret != HDLC_ERROR_SUCCESS) {
    printf(" HdlcSetEventMask Error[%x]¥n", ret);
    HdlcClose(nPort);
    return -1;
}

// 10秒間、または10件のフレームが受信されるまで待つ
for(i = 0; i < 10; i++) {
    if(ReceivedCount == 10) break;
    usleep(1000*1000);
}

// イベントマスクの設定
ret = HdlcSetEventMask(nPort, 0, 0);
if (ret != HDLC_ERROR_SUCCESS) {
    printf(" HdlcSetEventMask Error[%x]¥n", ret);
    HdlcClose(nPort);
    return -1;
}

// ポートのクローズ
```



```
HdlcClose(nPort);  
  
    return 0;  
}
```

次に Makefile を作成します。

エディタを起動し、下記に示すコードを記述します。

```
CC = gcc  
TARGET = IntFrameReceive  
CFLAGS = -lgpg4116  
  
all: $(TARGET)  
  
clean:  
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。

保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「IntFrameReceive.o」と実行ファイル「IntFrameReceive」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./IntFrameReceive
```

実行すると何も表示されず処理が停止します。約 10 秒後に自動的にプログラムが終了します。
処理の停止している 10 秒以内の間に「フレーム送信」で作成したプログラムを実行してください。

「8.4.1 フレーム送信」で作成したプログラムを実行するたびに一行ずつ表示されます。

```
receive data=The specified device could not be opened. (size=44)
receive data=The specified device could not be opened. (size=44)
...
```

今回作成したプログラムはフレームの受信を割り込みのコールバック機能を使用して行うプログラムです。

割り込みを発生させるには HdlcOpen 関数の第二引数のパラメータでコールバックルーチンを登録し、HdlcSetEventMask 関数で割り込みを発生させたい要因を設定します。

割り込みの要因に受信割り込みを設定し、ITC-N4007 の HDLC CN1 がメッセージを受信したらメッセージ受信割り込みが発生し、登録しているコールバックルーチン内で受信処理を行い、メッセージを表示します。

Step3 プログラムの解説

まずmainルーチンの処理から説明を行います。

HDLC製品を制御する前段階として、HDLC製品をオープンするHdlcOpen関数を呼び出しています。

HdlcOpen関数では第一引数にポート番号、第二引数にポート初期化構造体のポインタを指定します。

第一引数のポート番号は製品型式・チャンネル・RSW1設定値により一意に定まるポート番号を指定します。

ITC-N4007のHDLC CN1は「HDLC_PORTNO_467X02_0_1」となります。(値では49)

第二引数のポート初期化構造体では下記の設定を行っています。

- ・符号化フォーマット :NRZI
- ・FCS生成多項式 :ITU-T 16bit
- ・アドレスの扱い :アドレス検出なし
- ・ラインモード :全二重
- ・送信クロック :内部クロック
- ・受信クロック :DPLL
- ・ソースクロック :32MHz
- ・ボーレート :250000bps
- ・インタフェース :RS-485
- ・送信クロックモード :常に出力しない
- ・送信前切替時間 :100 μ s
- ・送信後切替時間 :100 μ s
- ・コールバックルーチン:ReceiveHandler関数を指定
- ・ユーザデータ :ポート番号を指定

次に HdlcSetEventMask 関数を用いて割込みイベントマスクの設定を行います。

HdlcSetEventMask 関数では第一引数にポート番号、第二引数にイベント発生要因のマスク設定値、第三引数にイベント発生要因条件を指定します。

ここでは受信割込み要因をアンマスクに設定しています。(データを受信し、そのデータが受信バッファに格納されたときにコールバックルーチンが呼ばれます。)

次に for 文を使用して 10 秒間経過する、または、受信したフレーム数が 10 件を超えた場合に処理を抜ける処理を行っています。

その後、HdlcClose 関数を用いてポートをクローズします。

オープンしたポートは終了時には必ずクローズを行ってください。

次にコールバックルーチン(ReceiveHandler 関数)の説明を行います。

HdlcOpen 関数の第二引数のポート初期化構造体で設定された ReceiveHandler 関数は HdlcSetEventMask 関数で設定した割り込み要因が発生した時に呼ばれます。

ここでは受信割り込みが発生したとき、この関数が呼ばれます。

まず、HdlcGetFrameCount 関数を用いて受信バッファ内に格納されているフレームの数を取得します。

HdlcGetFrameCount では第一引数にポート番号、第二引数にフレーム数を格納する変数のポインタを指定します。

for 文を使用し、受信バッファ内のフレーム数で繰り返し HdlcReceiveFrame 関数を実行し、フレームの受信を行います。

HdlcReceiveFrame 関数では第一引数にポート番号、第二引数に受信データを格納する配列のアドレス、第三引数に受信したデータサイズを格納する変数のポインタを指定します。

HdlcReceiveFrame 関数実行後、受信したデータを printf 関数で表示し、受信したフレーム数のカウント用変数をインクリメントして関数を抜けます。

再度受信割り込みが発生すれば、ReceiveHandler 関数はまた呼ばれます。

第9章 CAN

9.1 CANの概要

CANとは「Controller Area Network」の略で、シリアル通信プロトコルの一種です。

CANと聞くと多くの方が、自動車内ネットワークを連想されると思います。

元々CANは自動車内ネットワークで使用することを前提に開発されました。

私たちの生活に欠かせない自動車ですが、近年では高機能化が進み、これまでには必要の無かった様々な部品が自動車内に搭載されるようになってきました。部品が増えるということはそれらを接続するための配線もまた、増えるということになります。

また、省エネルギー化の波が押し寄せ、少ない燃料でより効率の良い走りが期待されています。

このように一見矛盾した様に見える二つの要望を満たす為、新たな技術を開発する必要が出てきました。

そこで考えられたのが、自動車内で使われている配線の数を減らせないかということです。

例えば自動車内にスイッチや、センサを取り付けていくと、それらを集中管理しているところまで配線を行う必要があります。そこで自動車メーカ各社は、各種信号をシリアルデータに変換し独自のプロトコルで通信を行い、配線の数を減らすことに成功しました。

しかし、規格が統一されていない為、開発の効率、部品の流通性の面では思うように改善が進みません。

1980年代後半、ドイツの電装メーカ BOSCH 社により開発された CAN は、その仕様をオープンとすることで、多くの半導体メーカにより対応チップが開発、ISO による規格化も行われ、瞬く間に自動車内ネットワークの標準として位置づけられることになりました。

また、フィールドバス(工業用途データバス)の一種である、DeviceNet の基礎技術としても使用され自動車の中だけにとどまることなく、幅広い分野で活用され始めています。

9.1.1 CANの特徴

CAN には下記のような特徴があります。

- ・低コスト・省配線

CAN は各デバイスをシリアルで接続するため、全体の配線コストを下げるすることができます。
また、多くの半導体メーカーより CAN 対応 IC が販売されており、価格競争面で優れています。

- ・高信頼性

自動車内ネットワークということで、高い信頼性を誇っています。外部からのノイズに強い差動信号で外部からのノイズの影響を受けにくくし、データの正当性を保証する CRC など、エラーを確実に検出します。

- ・調停

CAN はマルチマスタ方式で、バスが空いていれば、どの端末からでも送信を開始することができます。もし複数の端末から同時にデータが送信された場合、バス上で衝突を検知し、ID による優先順位の決定が行われます。衝突によりデータが無くなる事はありません。

→ CSMA/NBA 方式(Carrier Sense Multiple Access Nondestructive Bitwise Arbitration)

この調停により、無駄の無い通信が可能です。

優先順位の高い端末は周期を乱されず、高いリアルタイム性を保つことができます。

調停の仕組みについては「9.1.3 調停の仕組み(優先順位の決定方法)」を参照してください。

- ・柔軟なシステム構築

端末にアドレスを設定する必要が無いので、機器の増設が容易です。

9.1.2 バスレベルとビット

CAN バスには二つのレベルがあります。ドミナントとレセシブと呼ばれる状態です。

CAN バスは、必ずこの二つのレベルのうちどちらかのレベルになります。

論理的にはドミナントが“0”、レセシブが“1”を指します。

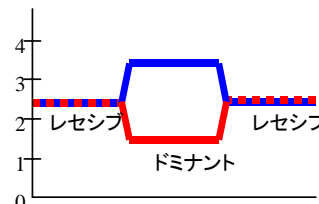
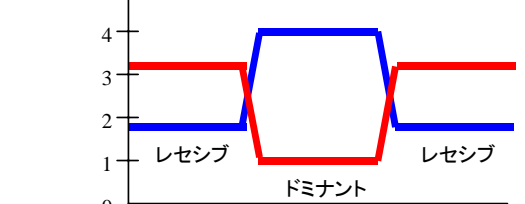
CAN はバス接続なので、同一バス上に複数の端末が接続され、それぞれが送信を行います。

この際、一つでもドミナントを出力する端末があると、バスのレベルはドミナントになります。

全ての端末からレセシブが出力されていればバスのレベルはレセシブです。

ドミナント、レセシブのそれぞれのバスレベルは、対応する規格により変わってきます。

それらをまとめたのが以下の表です。

	高速CAN	低速CAN	
	ISO11898-2	ISO11519-2	Fault-tolerant CAN ISO11898-3
バスの形状	ループバス	オープンバス	オープンバス
通信速度	125Kbps ~ 1Mbps / 40m	最高 125Kbps	最高 125Kbps / 40m
バスレベル			
障害発生時の自動復旧 (片線での通信継続)	×	×	○

このうち、弊社製品で対応しているのは ISO11898-2 と ISO11898-3 の二つです。

CAN バスのレベルの判断は CAN トランシーバが行います。

CAN バスには CAN_High と CAN_Low という 2 本の信号線があり、この 2 本の電位差によりレベルの判断が行われます。

上の表の通り、高速 CAN と低速 CAN では、バスレベルの判断の仕方が異なります。

高速 CAN の場合、CAN_High と CAN_Low の間に電位差が無いときがレセシブレベル、CAN_High が高い電圧で CAN_Low が低い電圧になり、電位差が生じた場合がドミナントレベルとなります。

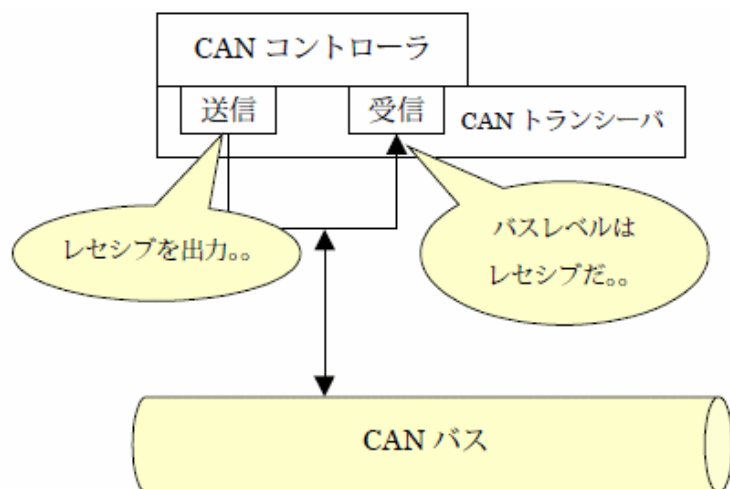
低速 CAN の場合、CAN_Low が CAN_High を上回っている場合、レセシブレベルで、CAN_High が CAN_Low を上回っている場合ドミナントレベルとなります。

9.1.3 調停の仕組み(優先順位の決定方法)

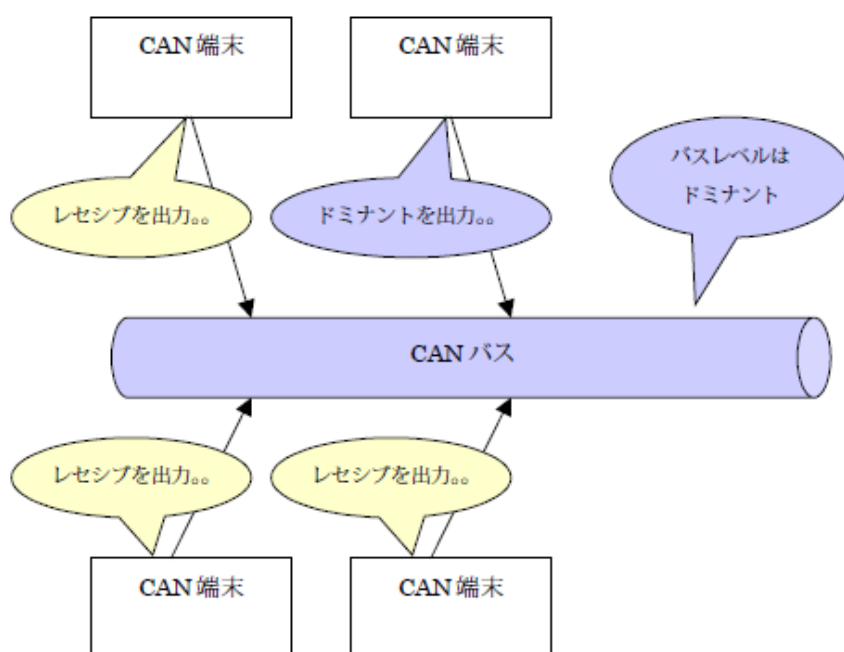
CAN の一つの大きな特長に、この調停の仕組みが挙げられます。

バス上で複数の端末が同時に送信を開始した場合、送信されたメッセージに含まれる ID 等を用い優先度を判断し、より優先度の高いメッセージがバス上に生き残り、優先度の低いメッセージは再送することになります。

では、どうやって優先度の判断を行っているのでしょうか？ ここで重要になってくるのが「バスレベルとビット」になります。CAN では、自分の出力したレベルと、バス上の実際のレベルを常に監視しています。(※バスレベルとビットについては「9.1.2 バスレベルとビット」を参照してください。)



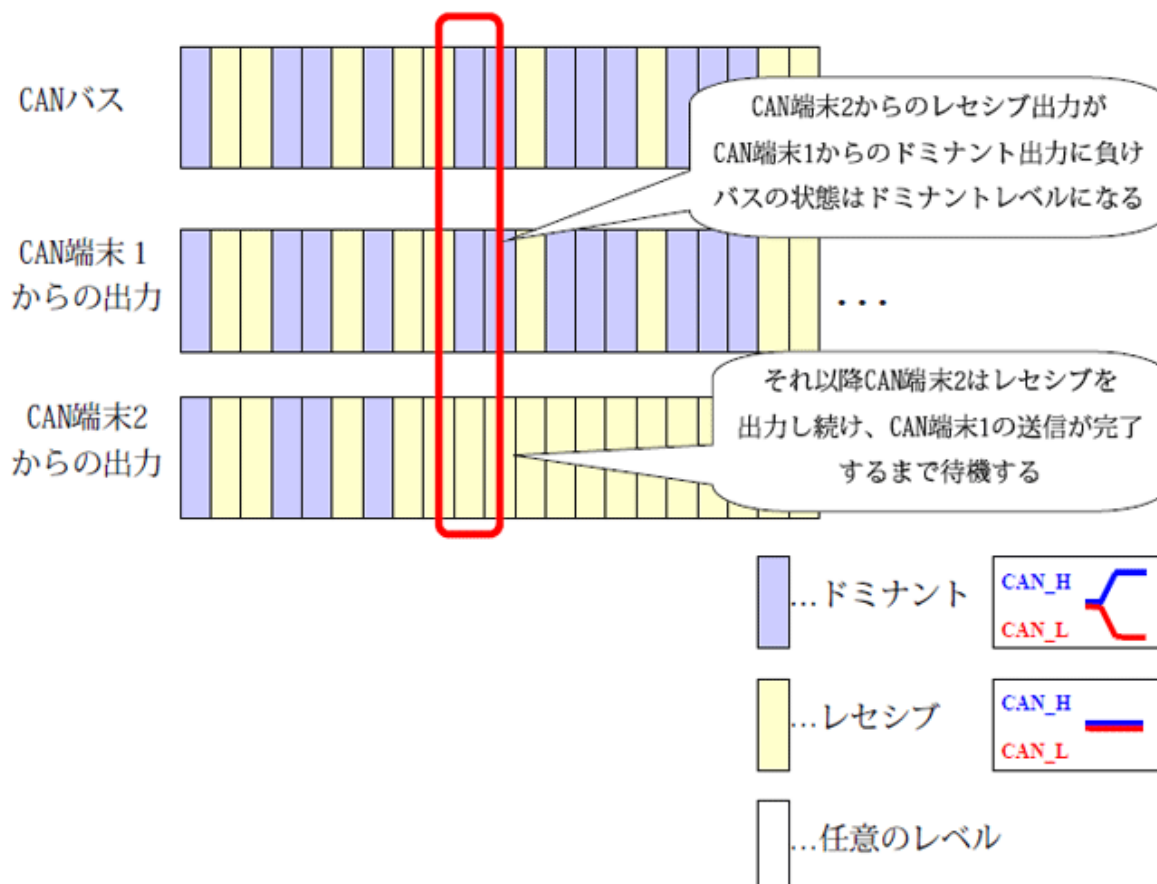
そして一つでもドミナントレベルを出力する端末がいたと、バスはドミナントレベルになります。



この仕組みをつかい、調停を行います。下の図のように 2 台の端末から同時に送信が行われたとします。CAN のメッセージ送信は、ID の上位ビットからそれぞれ送信されます。

自分の出力がドミナントレベルで、バス上もドミナントレベルであれば次のビットの出力を行います。自分の出力がレセシブレベルで、バス上もレセシブレベルだった場合も同様です。

もし自分がレセシブレベルを出力したのに、バス上がドミナントレベルだった場合には、他の端末からドミナントレベルが出力されていることになります。



これが調停に負けたということです。調停に負けた端末はバスが空くのを待ち、再送を行います。

※ このとき、自分よりも優先度の高い端末からの送信が入った場合、また調停に負け再送を行うことになります。時間を考慮した設計が必要になってきます。

9.1.4 フレームの種類

CAN では、フレームという単位で送受信を行います。以下の種類のフレームが CAN では定義されています。

フレームの種類	主な役割
データフレーム	データを送信する際に使用します。
リモートフレーム	通信相手に送信を要求する際に使用します。
エラーフレーム	送受信エラー発生するとき、他のユニットにエラーを知らせる際に使用します。
オーバーロードフレーム	データを受信する準備ができていないときに使用します。
インターフレームスペース	フレームとフレームの間に挿入し、間隔をあける際に使用します。

このうち、データフレームとリモートフレームについては、標準フォーマットと拡張フォーマットという 2 種類が存在します。標準フォーマットでは ID が 11 ビット、拡張フォーマットでは ID が 29 ビットという違いがあります。

・データフレーム

データフレームについて解説致します。

データフレームはメッセージの送信の際に使用し、最大で 8 バイトまでのデータを含むことができます。

以下にデータフレームの構成を示します。

Start of Frame	Arbitration field	Control field	Data field	CRC field	Acknowledge field	End of Frame
1bit	12bit or 32bit	6bit	0~8Byte (0~64bit)	16bit	2bit	7bit

・リモートフレーム

リモートフレームは、データ本体を持たないデータフレームの様に見えます。通常、受信側の端末から、送信側の端末へデータ送信を要求する際に使用します。

以下にリモートフレームの構成を示します。

Start of Frame	Arbitration field	Control field	CRC field	Acknowledge field	End of Frame
1bit	12bit or 32bit	6bit	16bit	2bit	7bit

・エラーフレーム

エラーフレームはエラーを検出した端末が他の端末にエラーを通知する際に使用します。

以下にデータフレームの構成を示します。

エラーフラグ	エラーデリミタ
6~12bit	8bit

- ・ オーバーロードフレーム

オーバーロードフレームは受信側端末の受信準備が整っていないときに送信されます。

以下にオーバーロードフレームの構成を示します。

オーバーロードフラグ	オーバーロードデリミタ
6～12bit	8bit

- ・ インターフレームスペース

インターフレームスペースは、データフレーム、リモートフレームの、フレーム間を分離する為に送信されます。

以下にインターフレームスペースの構成を示します。

インターミッション	サスペンドトランスミッション	バスアイドル
3bit	8bit	0bit～無制限

9.2 対象型式

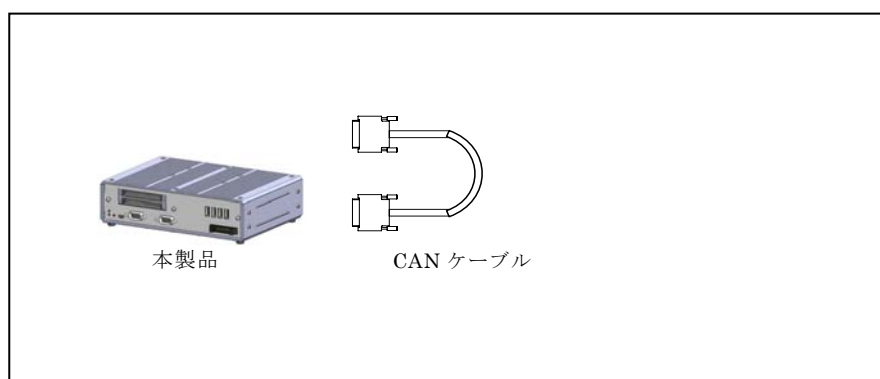
ここでは、実際に弊社製品を用いて、簡単な通信プログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N4005	タフコン CD シリーズ通信モデル
-----------	-------------------

9.3 環境図

ITC-N4005 の「CAN CN1」と「CAN CN2」を、CAN ケーブルを用いて接続します。



CAN ケーブルには D-sub 9pin のストレートケーブルを使用してください。

9.4 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

9.4.1 メッセージ送信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「CanSend.c」として保存してください。

```
#include <stdio.h>
#include "ifcan.h"

int main(void)
{
    int    portHandle;
    int    ret;
    unsigned long    msgCount;
    CAN_PORT_CONFIG Config;
    CAN_MESSAGE sendMessage;
    CAN_MESSAGE completionMessage;

    // ポートのオープン
    portHandle = CanOpenPort("ifcan2");
    if(portHandle < 0) {
        printf("Failed to open the port.¥n");
        return -1;
    }

    // 現在のCANの通信条件設定値取得
    ret = CanGetConfig(portHandle, &Config);
    if(ret) {
        printf("Failed to get configure the port.¥n");
        CanClosePort(portHandle);
        return -1;
    }

    // CANの通信条件を設定
    Config.ulLineMode = CAN_NORMAL_MODE;
    Config.ulFilterMode = CAN_SINGLE_FILTER;
    Config.ulTXBSize = 64;
    Config.ulRXBSize = 64;
    Config.ulERBSize = 64;
    Config.ulErrorLimit = 96;
```

```
Config.ulBaudRate = CAN_BAUDRATE_125k;
ret = CanSetConfig(portHandle, &Config);
if(ret) {
    printf("Failed to set configure the port.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANバスへの接続を有効
ret = CanActivate(portHandle);
if(ret) {
    printf("Failed to CanActivate.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANメッセージの送信
sendMessage.ulLength = 1;
sendMessage.ulFlag = 0;
sendMessage.ulID = 0x30;
sendMessage.bData[0] = 0x55;
sendMessage.ulTime = 0;
msgCount = 1;
ret = CanSendMessage(portHandle, &sendMessage, msgCount);
if(ret) {
    printf("Failed to CanSendMessage.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANメッセージの送信完了を待つ
while(1){
    msgCount = 1;
    ret = CanGetCompletionMessage(portHandle, &completionMessage, &msgCount);
    if((ret == IFCAN_ERROR_SUCCESS) && (msgCount == 1))break;
}
printf("Transmission was completed.%n");

// CANバスへの接続を無効
ret = CanDeactivate(portHandle);
if(ret) {
    printf("Failed to CanDeactivate.%n");
    CanClosePort(portHandle);
    return -1;
}

// ポートのクローズ
CanClosePort(portHandle);

return 0;
}
```

コードの記述が終わったら保存してください。
保存するファイル名は「CanSend.c」としてください。

次にMakefileを作成します。エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = CanSend
CFLAGS = -lifcan

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「CanSend.o」と実行ファイル「CanSend」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

※相手機器と接続していない場合、送信完了とならず、プログラムが待ちの状態になります。

```
./CanSend
```

正常に処理が終了すると下記が表示されます。

```
Transmission was completed.
```

今回作成したプログラムは CAN メッセージの送信を行うプログラムです。

CAN メッセージの送信を行うには CanSendMessage 関数を使用します。

ITC-N4005 の CAN CN1 から、ID = 30[h]、Data = 55[h]のデータを送信し、

CanGetCompletionMessage 関数にて送信が完了した事を確認し、プログラムを終了します。

Step3 プログラムの解説

CAN製品を制御する前段階として、CAN製品をオープンするCanOpenPort関数を呼び出しています。

CanOpenPort関数では第一引数にCANインタフェースのデバイス名を指定します。※1

ITC-N4005のCAN CN1は「ifcan2」となります。

関数の戻り値にはデバイスハンドルが戻ります。この値を用いてCANデバイスの制御を行います。

次にCanGetConfig関数を用いて、現在のCANの通信条件設定値を取得します。

CanGetConfig関数は第一引数にデバイスハンドル、第二引数に通信条件設定値を格納するCAN_PORT_CONFIG構造体のポインタを指定します。

次に CanSetConfig 関数を用いて、CAN の通信条件を設定します

CanSetConfig 関数は第一引数にデバイスハンドル、第二引数に設定する通信条件を格納した CAN_PORT_CONFIG 構造体のポインタを指定します。

ここでは下記の設定を行っています。

- ・通信モード: 通常の通信モード
- ・アクセプタンスフィルタモード: 1 つのフィルタ設定可能
- ・送信バッファサイズ: 64 メッセージ分
- ・受信バッファサイズ: 64 メッセージ分
- ・エラーバッファサイズ: 64 × 16 バイト
- ・エラーリミット: 96 件
- ・ボーレート: 125Kbps

次に CanActivate 関数を用いて CAN バスへの接続を有効(データの送受信が行える状態)にします。

CanActivate 関数では第一引数にデバイスハンドルを指定します。

次に CanSendMessage 関数を用いて CAN メッセージの送信を行います。

CanSendMessage 関数では第一引数にデバイスハンドル、第二引数に送信メッセージを格納した CAN_MESSAGE 構造体のポインタ、第三引数には送信するメッセージの件数を指定します。

ここでは下記のメッセージを 1 件送信します。

- ・送信メッセージの長さ : 1
- ・送信メッセージの特殊フラグ: なし
- ・送信メッセージ ID : 0x30
- ・送信メッセージ : 0x55
- ・ディレイ時間 : なし

次に while 文を用いて送信完了メッセージの取得を繰り返し実行し、送信完了を待ちます。

送信完了メッセージの取得には CanGetCompletionMessage 関数を使用します。

CanGetCompletionMessage 関数では第一引数にデバイスハンドル、第二引数に送信完了メッセージを格納する CAN_MESSAGE 構造体、第三引数に取得する送信完了メッセージの件数を設定した変数のポインタ(関数終了後には実際に取得した件数が格納されます)を指定します。

ここでは先ほど CanSendMessage 関数で送信した 1 件のメッセージの送信完了メッセージが取得できるまで(送信が完了するまで)処理を待ちます。

処理待ちの while 文を抜けた後、CanDeactivate 関数を用いて CAN バスへの接続を無効(データの送受が行えない状態)にします。

CanDeactivate 関数では第一引数にデバイスハンドルを指定します。

最後に CanClosePort 関数を用いてポートをクローズします。

CanClosePort 関数では第一引数にデバイスハンドルを指定します。

オープンしたポートは終了時には必ずクローズを行ってください。

★送信の完了を待たずに実行するとどうなるか？

送信待ちのフレーム数が0になる前にHdlcClose関数が呼び出される事になり、送信が途中までしか行われなくなります。送信が完了したことを確認した後でクローズを行ってください。

9.4.2 メッセージ受信

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「CanReceive.c」として保存してください。

```
#include <stdio.h>
#include "ifcan.h"

int main(void)
{
    int    ret;
    int    portHandle;
    unsigned long i, j, msgCount;
    CAN_PORT_CONFIG Config;
    CAN_PORT_STATUS portStatus;
    CAN_MESSAGE receiveMessage[64];

    // ポートのオープン
    portHandle = CanOpenPort("ifcan3");
    if(portHandle < 0) {
        printf("Failed to open the port.¥n");
        return -1;
    }

    // 現在のCANの通信条件設定値取得
    ret = CanGetConfig(portHandle, &Config);
    if(ret) {
        printf("Failed to get configure the port.¥n");
        CanClosePort(portHandle);
        return -1;
    }

    // CANの通信条件を設定
    Config.ulLineMode = CAN_NORMAL_MODE;
    Config.ulFilterMode = CAN_SINGLE_FILTER;
    Config.ulTXBSize = 64;
    Config.ulRXBSize = 64;
    Config.ulERBSize = 64;
    Config.ulErrorLimit = 96;
    Config.ulBaudRate = CAN_BAUDRATE_125k;
    ret = CanSetConfig(portHandle, &Config);
    if(ret) {
        printf("Failed to set configure the port.¥n");
        CanClosePort(portHandle);
        return -1;
    }

    // CANバスへの接続を有効
    ret = CanActivate(portHandle);
```

```
if(ret) {
    printf("Failed to CanActivate.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANメッセージが受信されるまで待つ
while(1){
    ret = CanGetStatus(portHandle, &portStatus);
    if(ret) {
        printf("Failed to CanGetStatus.%n");
        CanClosePort(portHandle);
        return -1;
    }
    if(portStatus.ulRXBCount >= 1)break;
}

// CANメッセージの受信
msgCount = portStatus.ulRXBCount;
ret = CanReceiveMessage(portHandle, &receiveMessage[0], &msgCount);
if(ret) {
    printf("Failed to CanReceiveMessage.%n");
    CanClosePort(portHandle);
    return -1;
}

// 受信したCANメッセージの表示
for(i = 0; i < msgCount; i++){
    printf("[ID:%#lx] ", receiveMessage[i].ulID);
    for(j = 0; j < receiveMessage[i].ulLength; j++){
        printf("%#x ", receiveMessage[i].bData[j]);
    }
    printf("%n");
}

// CANバスへの接続を無効
ret = CanDeactivate(portHandle);
if(ret) {
    printf("Failed to CanDeactivate.%n");
    CanClosePort(portHandle);
    return -1;
}

// ポートのクローズ
CanClosePort(portHandle);

return 0;
}
```

次にMakefileを作成します。エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = CanReceive
CFLAGS = -lifcan

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「CanReceive.o」と実行ファイル「CanReceive」が作られます。
※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./CanReceive
```

実行すると、CAN メッセージを受信するまでプログラムは待ち状態となります。

「9.4.1 メッセージ送信」で作成したプログラムを実行すると下記が表示されます。

```
[ID:0x30] 0x55
```

今回作成したプログラムは CAN メッセージの受信を行います。

CAN メッセージの受信を行うには CanReceiveMessage 関数を使用します。

ITC-N4005 の CAN CN2 に CAN メッセージが受信されるまで CanGetStatus 関数にて受信 CAN メッセージ数をポーリングしながら受信処理を待ちます。

CAN メッセージの受信後、CAN メッセージの取得を行い、取得した CAN メッセージの ID、DATA を表示します。

Step3 プログラムの解説

CAN製品を制御する前段階として、CAN製品をオープンするCanOpenPort関数を呼び出しています。

CanOpenPort関数では第一引数にCANインタフェースのデバイス名を指定します。※1

ITC-N4005のCAN CN2は「ifcan3」となります。

関数の戻り値にはデバイスハンドルが戻ります。この値を用いてCANデバイスの制御を行います。

次に CanSetConfig 関数を用いて、CAN の通信条件を設定します

CanSetConfig 関数は第一引数にデバイスハンドル、第二引数に設定する通信条件を格納した CAN_PORT_CONFIG 構造体のポインタを指定します。

ここでは下記の設定を行っています。

- ・通信モード: 通常の通信モード
- ・アクセプタンスフィルタモード: 1 つのフィルタ設定可能
- ・送信バッファサイズ: 64 メッセージ分
- ・受信バッファサイズ: 64 メッセージ分
- ・エラーバッファサイズ: 64 × 16 バイト
- ・エラーリミット: 96 件
- ・ボーレート: 125Kbps

次に CanActivate 関数を用いて CAN バスへの接続を有効(データの送受信が行える状態)にします。

CanActivate 関数では第一引数にデバイスハンドルを指定します。

次に while 文を使用して、ステータスをポーリングします。ステータスの取得には CanGetConfig 関数を使用します。

CanGetConfig 関数では第一引数にデバイスハンドル、第二引数にステータスを格納する CAN_PORT_STATUS 構造体のポインタを指定します。

ここでは受信バッファに蓄えられている CAN メッセージ数が 0 でなくなるまで(CAN メッセ時を受信するまで)ここで繰り返しポーリングを行います。

次に CanReceiveMessage 関数を用いて CAN メッセージの受信を行います。

CanReceiveMessage 関数では第一引数にデバイスハンドル、第二引数に受信メッセージを格納する CAN_MESSAGE 構造体のポインタ、第三引数には受信するメッセージの件数を格納した変数のポインタ(関数終了後には実際に取得した件数が格納されます)を指定します。

CanReceiveMessage 関数実行後、printf 関数にて取得した CAN メッセージを表示した後、CanDeactivate 関数を用いて CAN バスへの接続を無効(データの送受が行えない状態)にします。

CanDeactivate 関数では第一引数にデバイスハンドルを指定します。

最後に CanClosePort 関数を用いてポートをクローズします。

CanClosePort 関数では第一引数にデバイスハンドルを指定します。

オープンしたポートは終了時には必ずクローズを行ってください。

9.4.3 割り込み処理

Step1 プログラム作成

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「CanIntReceive.c」として保存してください。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include "ifcan.h"
#include "ifcanpf.h"

int portHandle;

// コールバックルーチン
void EventCallback(unsigned long Event, unsigned long dummy)
{
    unsigned long i;
    int ret;
    CAN_MESSAGE CanMsg;
    unsigned long length;

    // CANメッセージの受信、受信したメッセージの表示
    length = 1;
    ret = CanReceiveMessage(portHandle, &CanMsg, &length);
    if(ret){
        printf("CanReceiveMessage Error¥n");
    } else {
        printf("time:%8ld ID:%4lx Data:", CanMsg.ulTime, CanMsg.ulID);
        for(i = 0; i < CanMsg.ulLength; i++) {
            printf("%02X ", CanMsg.bData[i]);
        }
        printf("¥n");
    }
    return;
}

int main(void)
{
    int ret, i;
    CAN_PORT_CONFIG Config;
    CAN_EVENT_REQ Event;

    // ポートのオープン
    PortHandle = CanOpenPort("ifcan3");
    if (portHandle < 0) {
```

```
    printf("Failed to open the port.%n");
    return -1;
}

// 現在のCANの通信条件設定値取得
ret = CanGetConfig(portHandle, &Config);
if(ret) {
    printf("Failed to get configure the port.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANの通信条件を設定
Config.ulLineMode = CAN_NORMAL_MODE;
Config.ulFilterMode = CAN_SINGLE_FILTER;
Config.ulTXBSize = 64;
Config.ulRXBSize = 64;
Config.ulERBSize = 64;
Config.ulErrorLimit = 96;
Config.ulBaudRate = CAN_BAUDRATE_125k;
ret = CanSetConfig(portHandle, &Config);
if(ret) {
    printf("Failed to set configure the port.%n");
    CanClosePort(portHandle);
    return -1;
}

// コールバックルーチンの設定
memset(&Event, 0, sizeof(CAN_EVENT_REQ));
Event.lpCallBackProc = EventCallback;
Event.dwUser = 0;
ret = CanSetEvent(portHandle, &Event);
if(ret) {
    printf("Failed to CanSetEvent.%n");
    CanClosePort(portHandle);
    return -1;
}

// 割り込みイベントマスクの設定
ret = CanSetEventMask(portHandle, CAN_EVENT_RECV);
if(ret) {
    printf("Failed to CanSetEventMask.%n");
    CanClosePort(portHandle);
    return -1;
}

// CANバスへの接続を有効
ret = CanActivate(portHandle);
if(ret) {
    printf("Failed to CanActivate.%n");
    CanClosePort(portHandle);
    return -1;
}
```



```
}

// 10秒間待つ
for(i=0; i<10; i++){
    usleep(1000*1000);
}

// CANバスへの接続を無効
ret = CanDeactivate(portHandle);
if(ret) {
    printf("Failed to CanDeactivate.%n");
    CanClosePort(portHandle);
    return -1;
}

// ポートのクローズ
CanClosePort(portHandle);

return 0;
}
```

次にMakefileを作成します。
エディタを起動し、下記に示すコードを記述します。

```
CC = gcc
TARGET = CanIntReceive
CFLAGS = -lifcan

all: $(TARGET)

clean:
    rm -f $(TARGET) *~
```

コードの記述が終わったら保存してください。
保存するファイル名は「Makefile」としてください。

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「CanIntReceive.o」と実行ファイル「CanIntReceive」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./CanReceive
```

実行すると何も表示されず処理が停止します。

約 10 秒後に自動的にプログラムが終了します。

処理の停止している 10 秒以内の間に「メッセージ送信」で作成したプログラムを実行してください

「9.4.1 メッセージ送信」で作成したプログラムを実行するたびに一行ずつ表示されます。

```
time: xxxxxxxx ID: 30 Data:55  
time: xxxxxxxx ID: 30 Data:55  
...
```

※time の値は割り込みを発生させた時間により変化します。

今回作成したプログラムは CAN メッセージの受信を割り込みのコールバック機能を使用して行うプログラムです。

割り込みを発生させるには CanSetEvent 関数にて割り込み発生時に起動するコールバックルーチン、またはシグナルの設定を行い、CanSetEventMask 関数にて割り込みを発生させたい要因を設定します。

割り込みの要因に受信割り込みを設定し、ITC-N4005 の CAN CN2 が CAN メッセージを受信すると受信割り込みが発生し、登録しているコールバックルーチンの中で受信処理を行い、CAN メッセージの受信時間、ID、Data を表示します。

Step3 プログラムの解説

mainルーチンの処理から説明を行います。

CAN製品を制御する前段階として、CAN製品をオープンするCanOpenPort関数を呼び出しています。

CanOpenPort関数では第一引数にCANインタフェースのデバイス名を指定します。※1

ITC-N4005のCAN CN2は「ifcan3」となります。

関数の戻り値にはデバイスハンドルが戻ります。この値を用いてCANデバイスの制御を行います。

次に CanSetConfig 関数を用いて、CAN の通信条件を設定します。

CanSetConfig 関数は第一引数にデバイスハンドル、第二引数に設定する通信条件を格納した CAN_PORT_CONFIG 構造体のポインタを指定します。

ここでは下記の設定を行っています。

- ・通信モード: 通常の通信モード
- ・アクセプタンスフィルタモード: 1 つのフィルタ設定可能
- ・送信バッファサイズ: 64 メッセージ分 (64 × 24 バイト)
- ・受信バッファサイズ: 64 メッセージ分 (64 × 24 バイト)
- ・エラーバッファサイズ: 64 × 16 バイト
- ・エラーリミット: 96 件
- ・ボーレート: 125Kbps

次に CanSetEvent 関数を用いて割り込み要因発生時に呼ばれるコールバックルーチンの設定を行います。

CanSetEvent 関数では第一引数にデバイスハンドル、第二引数に割り込み設定を格納した CAN_EVENT_REQ 構造体のポインタを指定します。

ここでは下記の設定を行っています。

コールバックルーチン: EventCallback 関数
ユーザデータ : 0

次に CanSetEventMask を用いて割り込みイベントマスクの設定を行います。

CanSetEvent 関数では第一引数にデバイスハンドル、第二引数にイベントマスクの設定値を指定します。ここでは受信割り込みが発生した場合にコールバックルーチンが呼ばれるように設定しています。

次に CanActivate 関数を用いて CAN バスへの接続を有効(データの送受信が行える状態)にします。

CanActivate 関数では第一引数にデバイスハンドルを指定します。

次に for 文を使用して、約 10 秒間処理を待ちます。

for 文を抜けた後、CanDeactivate 関数を用いて CAN バスへの接続を無効(データの送受が行えない状態)にします。
CanDeactivate 関数では第一引数にデバイスハンドルを指定します。

最後に CanClosePort 関数を用いてポートをクローズします。
CanClosePort 関数では第一引数にデバイスハンドルを指定します。
オープンしたポートは終了時には必ずクローズを行ってください。

次にコールバックルーチン(EventCallback 関数)の説明を行います。
CanSetEvent 関数で設定された EventCallback 関数は CanSetEventMask 関数で設定した割り込み要因が発生した時に呼ばれます。
ここでは受信割り込みが発生したとき、この関数が呼ばれます。

EventCallback 関数では、CanReceiveMessage 関数を用いて CAN メッセージの受信を行います。
CanReceiveMessage 関数では第一引数にデバイスハンドル、第二引数に受信メッセージを格納する CAN_MESSAGE 構造体のポインタ、第三引数には受信するメッセージの件数を格納した変数のポインタ(関数終了後には実際に取得した件数が格納されます)を指定します。
CanReceiveMessage 関数実行後、printf 関数にて取得した CAN メッセージを表示し、関数を抜けます。
再度受信割り込みが発生すれば、ReceiveHandler 関数はまた呼ばれます。

※1 CAN インタフェースのデバイス名は「ifcanxx」はデバイスの認識した順番に番号が割り振られていきます。そのため、デバイスごとの一意な名前ではありません。

デバイス名を確認するにはコンソール上で「cat /proc/driver/can/cp4851」と入力します。

入力後、製品型式とデバイス名が表示されます。

```
#cat /proc/driver/can/cp4851
ifcan1: PCI/LPC/PAZ-485110 (bid=0h) CH1 (High-speed) [125000bps] tx:0 rx:0
ifcan2: PCI/LPC/PAZ-485120 (bid=1h) CH1 (High-speed) [125000bps] tx:0 rx:0
ifcan3: PCI/LPC/PAZ-485120 (bid=1h) CH2 (High-speed) [125000bps] tx:0 rx:0
```

※ ITC-N4005 の CAN CN1, CN2 はそれぞれ PCI-485120 CH1, CH2 として表示されます。

※ 背面にある CAN は PCI-485110 CH1 として表示されます。

第10章 カウンタ

10.1 カウンタの概要

単純にパルスの数だけをカウントするものから、周波数や周期を計測したり、2つのパルスから位置を特定したりとカウンタの用途は色々です。また、一定周期毎にクロックが発生するタイマカウンタや、パルスを出力するものもあります。

カウントする用途、接続する機器によって使い分けます。

10.2 対象型式

ここでは、実際に弊社製品を用いて、簡単なプログラムの作成・実行を行います。

チュートリアルでは以下の製品を使用しています。

ITC-N3620	タフコン CD シリーズ複合モデル
-----------	-------------------

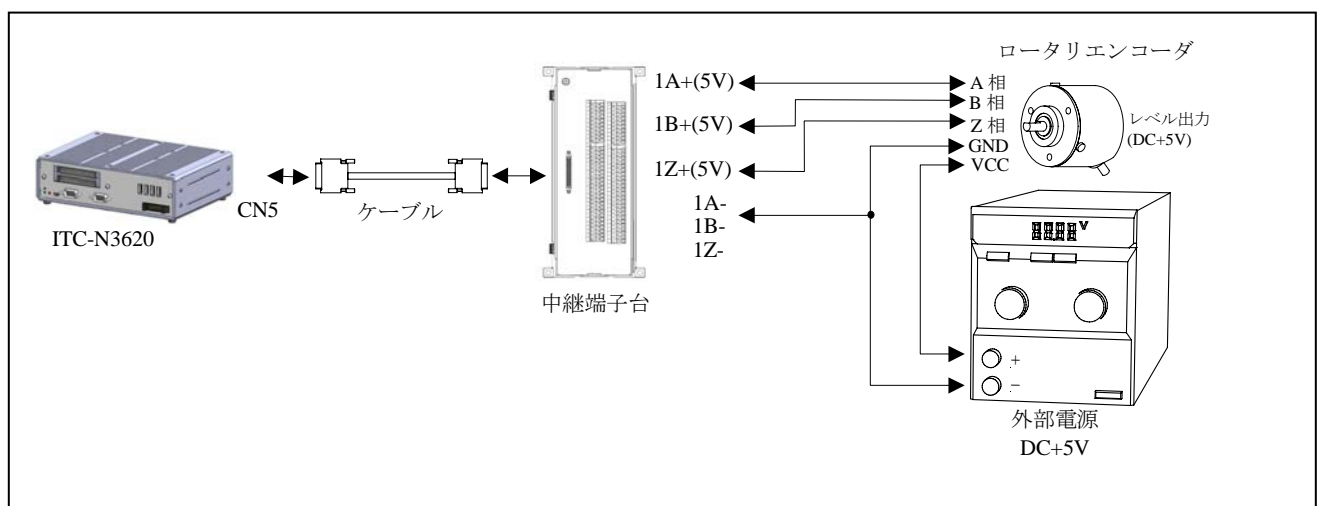
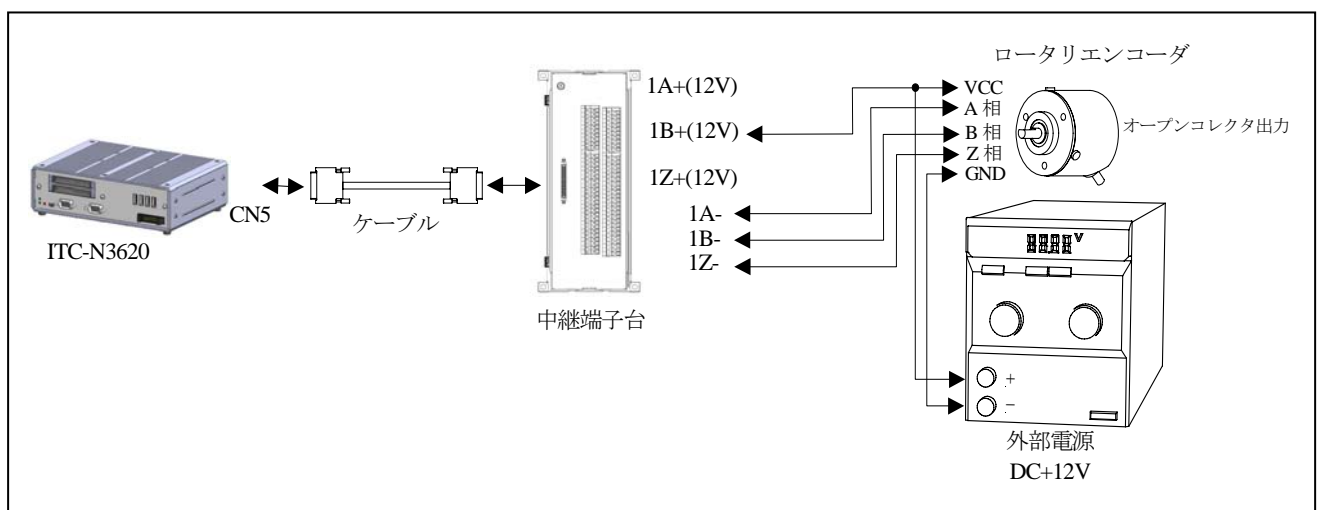
10.3 チュートリアル

★プログラム中のコメントについて

サンプルプログラムには、理解し易いよう、日本語コメントを用いています。
しかし、プログラム中に日本語を用いてコンパイルすると、正常にコンパイルできない場合があります。
その時は、日本語コメントを削除してコンパイルしてください。

10.3.1 万能力カウンタ

環境図は下記になります。



Step1 プログラム作成

パルスカウントモード、平均周波数測定モード、周期測定モード、位相差幅測定モード、タイマモード、分周期モード、パルスジェネレータモードの7つの機能モードから選択して使用することが出来ます。

例えば、パルスカウントモードでの制御手順では、UcntSetPulseCountMode 関数で機能モード設定、UcntSetBaseClock 関数や UcntSetTriggerConfig 関数でカウンタ設定を行います。

UcntStartCount 関数でカウントを開始し、UcntReadCounter 関数でカウント値の取得を行います。

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「encoderpulsecount.c」として保存してください。

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

#include "ifucnt.h"

int main(int argc, char *argv[])
{
    int i, nRet, nDevice;
    int nChannel;
    unsigned long dwChSel;
    unsigned long dwCountMode;
    unsigned long dwLoadMode;
    unsigned long dwLatchMode;
    unsigned long dwLoadData;
    unsigned long dwCounter[4];

    // デバイス番号 1 のデバイスを制御
    nDevice = 1;

    // デバイスをオープン
    nRet = UcntOpen(nDevice);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // パルスカウントモードに設定
    nChannel = 1; // チャンネル1
    dwCountMode = IFUCNT_COUNT_PHASE_1 | IFUCNT_DIR_NORMAL; // 位相差パルスカウントモード 1 週
    // 倍, 通常方向(カウントアップ)
    dwLoadMode = 0x10; // 同期エッジ
    dwLatchMode = 0x00; // ラッチ無効

    nRet = UcntSetPulseCountMode(nDevice, nChannel, dwCountMode, dwLoadMode, dwLatchMode);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
```

```
        goto EXIT_CLOSE;
    }

    // プリロードデータに 0 を設定
    dwLoadData = 0;
    nRet = UcntSetLoadData(nDevice, nChannel, dwLoadData);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // カウントを開始
    dwChSel = 0x01 << (nChannel - 1);
    nRet = UcntStartCount(nDevice, dwChSel, IFUCNT_CMD_START);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // カウント値を取得し、表示
    for (i = 0; i < 1000; i++) {
        nRet = UcntReadCounter(nDevice, dwChSel, &dwCounter[0]);
        if (nRet != IFUCNT_ERROR_SUCCESS) {
            goto EXIT_CLOSE;
        } else {
            printf("[%d] %08lXh¥n", i, dwCounter[(nChannel - 1)]);
        }
        usleep(10 * 1000);
    }

    // カウントを停止
    nRet = UcntStopCount(nDevice, dwChSel, IFUCNT_CMD_STOP);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

EXIT_CLOSE:
    // デバイスをクローズ
    UcntClose(nDevice);

    if (nRet != IFUCNT_ERROR_SUCCESS) {
        exit(EXIT_FAILURE);
    }

    return 0;
}
```


Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
all: encoderpulsecount

encoderpulsecount: encoderpulsecount.o
    $(CC) encoderpulsecount.o -o encoderpulsecount -lgpg6320u

encoderpulsecount.o: encoderpulsecount.c
    $(CC) -Wall -c encoderpulsecount.c -o encoderpulsecount.o

clean:
    rm -f *.o encoderpulsecount
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。
下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「encoderpulsecount.o」と実行ファイル「encoderpulsecount」が作られます。

※Error や Warning が表示された場合はコードの記述に誤りがありますので内容を見直してください。

プログラムを実行するには、以下のコマンドを実行します。

```
./encoderpulsecount
```

正常に処理が終了すると下記が表示されます。

```
[0] 00000000h
[1] 00000001h
[2] 00000002h
[3] 00000003h
...
[998] 000003E6h
[999] 000003E7h
```

Step3 プログラムの解説

カウンタ製品を制御する前段階として、カウンタ製品をオープンする UcntOpen 関数を呼び出しています。
以下に引数と対応を示します。

```
int UcntOpen(  
    int          nDevice,          /* デバイス番号 */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。

この引数を与えて関数を呼び出すと、ドライバは引数に適合するカウンタ製品を検索し、合致した製品がある場合は、戻り値として 0 を返します。

プログラムは、このデバイス番号を使って以降の API の呼び出しを行います。

カウンタ製品を制御するためには、まずカウンタのモード設定を行ないます。

カウンタのモードには様々な種類があり、パルスカウントモード、平均周波数測定モード、周期測定モード、位相差幅測定モード、タイマモード、分周器モード、パルスジェネレーターモードがあります。

これらの設定を行なうには、以下の関数を使用します。

カウンタモード	関数
パルスカウントモード	UcntSetPulseCountMode 関数
平均周波数測定モード	UcntSetFreqAvgMode 関数
周期測定モード	UcntSetCycleMode 関数
位相差幅測定モード	UcntSetPhaseDiffMode 関数
タイマモード	UcntSetTimerMode 関数
分周器モード	UcntSetFreqDividerMode 関数
パルスジェネレーターモード	UcntSetPulseGeneratorMode 関数

ここでは、パルスカウントモードの設定を行なうための UcntSetPulseCountMode 関数を使用します。

それぞれの引数で、設定を行なうチャンネル、カウンタモード、プリロードモード、ラッチモードを設定する事が出来ます。

ここでは、設定を行なうチャンネルを 1 とし、位相差パルスカウント・1 通倍・通常方向(カウントアップ)、同期エッジ、ラッチ無効に設定しています。

次にプリロードデータの設定を行ないます。設定を行なうには、UcntSetLoadData 関数をします。
ここでは、設定するチャンネルを 1 とし、プリロードデータを 0 としています。

設定が終わると、カウンタを開始させます。カウンタを開始するには、UcntStartCount関数を使用します。開始するチャンネル、スタートモードを指定します。

開始するチャンネルは複数のチャンネルを一度に指定することができ、bit1~bit4にそれぞれCH1~CH4が割り当てられています。

ここでは、カウントを開始するチャンネルを1のみとし、スタートモードをソフトスタートとしています。

UcntStartCount 関数を実行すると、カウントが開始されます。現在のカウント値を1件ごとに取得するには、UcntReadCounter関数を使用します。UcntStartCount関数同様、取得するチャンネルは複数同時に指定することができ、第3引数で渡された変数のポインタに結果を返します。

ここでは、for文を使用して、現在のカウント値をprintf関数を使用して、1000回表示しています。

接続したロータリエンコーダをまわすことで、カウントアップを行ないます。

カウントを停止するには、UcntStopCount関数を使用します。UcntStartCount関数同様、停止するチャンネルは複数同時に指定することができます。

ここでは、チャンネル1を指定してカウント停止を行なっています。

最後にカウンタ製品の制御を終了するためには、UcntClose関数を用います。

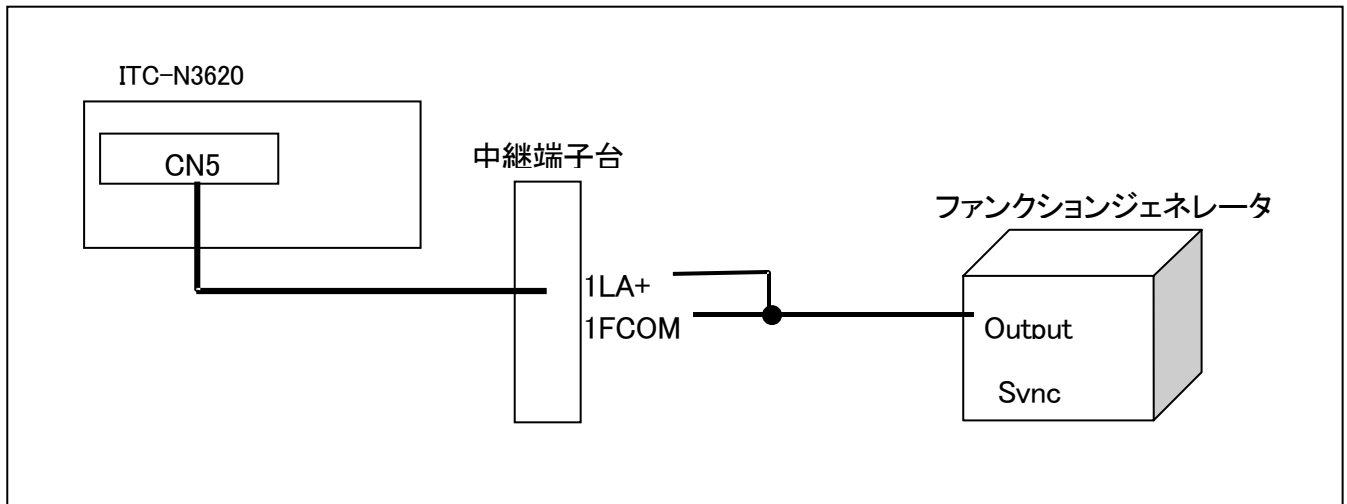
オープンしたカウンタ製品は、使用後は必ずクローズしてください。

★UcntClose関数を呼ばないと、どうなるか？

UcntClose関数を呼び出さないと、カウンタ製品はオープンしたままの状態になります。
オープン状態なので、UcntOpen関数を呼び出した場合、エラーが返ります。
オープンしたら必ずクローズしてください。

10.3.2 サンプルング

環境図としては、下記になります。



Step1 プログラム作成

また、サンプルング機能を使用することで、指定した件数分のデータをメモリ上に保持することができ、内部クロックを使用したラッチを行うことで指定した周期でカウンタ値を取得することもできます。

例えば、周期測定モードでサンプルング機能を使用し、デューティ比の測定を行えます。

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「pulsedutysampling.c」として保存してください。

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

#include "ifucnt.h"

int main(int argc, char *argv[])
{
    int i, nRet, nDevice;
    int nChannel;
    unsigned long dwChSel;
    unsigned long dwCountType;
    unsigned long dwClock;
    unsigned long dwSmplStatus;
    unsigned long dwSmplCount;
    unsigned long dwRepCount;
    unsigned long dwDataNum;
    unsigned long dwCounter[1000][2];
```

```
unsigned long dwHighCount;
unsigned long dwLowCount;

UCNTSMPLCONFIG SmplConfig;

// デバイス番号 1 のデバイスを制御
nDevice = 1;

// デバイスをオープン
nRet = UcntOpen(nDevice);
if (nRet != IFUCNT_ERROR_SUCCESS) {
    goto EXIT_CLOSE;
}

// 周期測定モードに設定
// 基準クロック 1us でHigh, Lowパルス幅を測定
nChannel = 1;
dwCountType = IFUCNT_CYCLE_HIGH_LOW;
dwClock = 10;

nRet = UcntSetCycleMode(nDevice, nChannel, dwCountType, dwClock);
if (nRet != IFUCNT_ERROR_SUCCESS) {
    goto EXIT_CLOSE;
}

// サンプリング設定
SmplConfig.dwSmplNum = 100;
SmplConfig.dwSmplEventNum = 0;
SmplConfig.dwSmplRepeat = 1;
SmplConfig.dwStatusMode = IFUCNT_ADD_STATUS;
SmplConfig.dwErrCtrl = IFUCNT_FREERUN;
nRet = UcntSetSamplingConfig(nDevice, nChannel, &SmplConfig);
if (nRet != IFUCNT_ERROR_SUCCESS) {
    goto EXIT_CLOSE;
}

// カウントを開始
dwChSel = 0x01 << (nChannel - 1);
nRet = UcntStartCount(nDevice,
                      dwChSel,
                      IFUCNT_CMD_START | IFUCNT_CMD_SAMPLING);
if (nRet != IFUCNT_ERROR_SUCCESS) {
    goto EXIT_CLOSE;
}

// サンプリング完了まで待機
printf("Wait for the completion of the sampling.\n");
do {
    nRet = UcntGetSamplingStatus(nDevice,
                                nChannel,
                                &dwSmplStatus,
```

```
                                &dwSmplCount,  
                                &dwRepCount);  
    if (nRet != IFUCNT_ERROR_SUCCESS) {  
        goto EXIT_CLOSE;  
    }  
} while ((dwSmplStatus == IFUCNT_SMPL_START) || (dwSmplCount < 100));  
  
// カウントを停止  
nRet = UcntStopCount(nDevice, dwChSel, IFUCNT_CMD_STOP);  
if (nRet != IFUCNT_ERROR_SUCCESS) {  
    goto EXIT_CLOSE;  
}  
  
// サンプリングデータを取得  
dwDataNum = 100;  
nRet = UcntGetSamplingData(nDevice, nChannel, &dwCounter[0][0], &dwDataNum);  
if (nRet != IFUCNT_ERROR_SUCCESS) {  
    goto EXIT_CLOSE;  
}  
  
dwHighCount = 0;  
dwLowCount = 0;  
for (i = 0; i < (int)dwDataNum; i++) {  
    if ((dwCounter[i][1] & 0x2500) == 0x2100) {  
        // LA 立ち下りエッジでのカウント値 (Highパルス幅)  
        if (dwHighCount == 0) {  
            dwHighCount = dwCounter[i][0];  
            dwLowCount = 0;  
        }  
    } else if ((dwCounter[i][1] & 0x2500) == 0x2400) {  
        // LA 立ち上がりエッジでのカウント値 (Lowパルス幅)  
        if ((dwLowCount == 0) && (dwHighCount != 0)) {  
            dwLowCount = dwCounter[i][0];  
        }  
    }  
}  
// Highパルス幅とLowパルス幅からデューティ比を計算し、表示  
if ((dwHighCount != 0) && (dwLowCount != 0)) {  
    printf("%03ld:(%%)%%n", (dwHighCount * 100) / (dwHighCount + dwLowCount));  
    dwHighCount = 0;  
    dwLowCount = 0;  
}  
}  
  
EXIT_CLOSE:  
// デバイスをクローズ  
UcntClose(nDevice);  
  
if (nRet != IFUCNT_ERROR_SUCCESS) {  
    exit(EXIT_FAILURE);  
}  
  
return 0;
```

```
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
all: pulsedutysampling

pulsedutysampling: pulsedutysampling.o
    $(CC) pulsedutysampling.o -o pulsedutysampling -lgpg6320u

pulsedutysampling.o: pulsedutysampling.c
    $(CC) -Wall -c pulsedutysampling.c -o pulsedutysampling.o

clean:
    rm -f *.o pulsedutysampling
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「pulsedutysampling.o」と実行ファイル「pulsedutysampling」が作られます。

Error や Warning が表示された場合はコードの記述に誤りがあります。内容を見直してください。
プログラムを実行するには、以下のコマンドを実行します。

```
./pulsedutysampling
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
050:(% %)
050:(% %)
050:(% %)
...
```

Step3 プログラムの解説

カウンタ製品を制御する前段階として、カウンタ製品をオープンするUcntOpen関数を呼び出しています。
以下に引数と対応を示します。

```
int UcntOpen(  
    int          nDevice,          /* デバイス番号 */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。

この引数を与えて関数を呼び出すと、ドライバは引数に適合するカウンタ製品を検索し、合致した製品がある場合は、戻り値として0を返します。

プログラマは、このデバイス番号を使って以降のAPIの呼び出しを行います。

カウンタ製品を制御するためには、まずカウンタのモード設定を行ないます。

カウンタのモードには様々な種類があり、パルスカウントモード、平均周波数測定モード、周期測定モード、位相差幅測定モード、タイマモード、分周器モード、パルスジェネレーターモードがあります。

これらの設定を行なうには、以下の関数を使用します。

カウンタモード	関数
パルスカウントモード	UcntSetPulseCountMode関数
平均周波数測定モード	UcntSetFreqAvgMode関数
周期測定モード	UcntSetCycleMode関数
位相差幅測定モード	UcntSetPhaseDiffMode関数
タイマモード	UcntSetTimerMode関数
分周器モード	UcntSetFreqDividerMode関数
パルスジェネレーターモード	UcntSetPulseGeneratorMode関数

ここでは、サンプリングモードの設定を行なうためのUcntSetCycleMode関数を使用します。

この関数はサンプリング周期の設定を行います。ここでは、設定するチャンネルを1に指定し、周期測定モード、基準クロックを1usとしHigh/Lowパルス幅の測定、の設定を行なっています。

サンプリング周期の設定が終わると、次にサンプリング条件の設定を行いません。

サンプリング条件の設定を行なうには、UcntSetSamplingConfig 関数を使用します。

設定は UCNTSMPLCONFIG 構造体で指定します。

構造体のメンバーで、サンプリング件数、イベント通知サンプリング件数、サンプリング繰り返し回数、サンプリングステータスへのステータス付加の有無、サンプリングエラー発生時の処理をそれぞれ指定します。

ここでは、設定するチャンネルを 1 に指定し、サンプリング件数を 100 件、、イベント通知サンプリング件数を 0 件、サンプリング繰り返し回数を 1 回、サンプリングステータスへのステータス付加を有効、サンプリングエラー発生時は何も処理をしない、に設定しています。

サンプリングの設定が終わると、サンプリングを開始させます。サンプリングを開始するには、UcntStartCount 関数を使用します。開始するチャンネル、スタートモードを指定します。

開始するチャンネルは複数のチャンネルを一度に指定することができ、bit1~bit4 にそれぞれ CH1~CH4 が割り当てられています。

ここでは、サンプリングを開始するチャンネルを 1 のみとし、スタートモードをソフトスタート・サンプリングスタートとしています。

UcntStartCount 関数を実行すると、サンプリングが開始され、指定されたサンプリング周期で 100 件のカウントを開始します。

サンプリングは非同期で実行されるため、関数を実行するとサンプリング開始処理を行い、すぐに処理を返します。

このため、サンプリング完了を確認する必要がある、サンプリング完了を確認するには、UcntGetSamplingStatus 関数を使用します。

この関数は、取得するチャンネルを指定することで、サンプリングステータス、サンプリングが完了した件数、現在のサンプリング繰り返し回数を取得する事が出来ます。

ここでは、do/while 文を使用し、ループ条件にサンプリングステータスがサンプリング動作中であること、またはサンプリング完了件数が 100 件未満である事を指定しています。

サンプリングが完了すると、サンプリングステータスはサンプリング停止中となり、サンプリング完了件数は 100 件となるため、do/while 文を抜けます。

サンプリングの完了を確認出来たら、サンプリングを停止させます。サンプリングを停止させるには、UcntStopCount 関数を使用します。UcntStartCount 関数同様、停止するチャンネルは複数同時に指定することができます。

ここでは、チャンネル1を指定してサンプリング停止を行なっています。

次に、サンプリングデータの取得を行ないます。サンプリングデータの取得を行なうには、UcntGetSamplingData 関数を使用します。取得するチャンネル番号、取得するサンプリング件数を指定し、サンプリング結果は、第 3 引数で渡された配列のポインタに結果を返します。

第 4 引数で、取得するサンプリング件数を指定しますが、変数のポインタを渡しています。これは、関数を実行したときにサンプリングしている件数が変数のポインタに格納されます。

したがって、サンプリングを 100 件しか行なっていないのに、取得件数を 200 件と指定した場合、実際に取得したサンプリング件数は 100 件となるため、変数のポインタには 100 が格納されます。

High/Low パルス幅測定モードでは、LA 信号の全ての立ち上がりと立ち下りのエッジでカウンタ値をラッチ、クリアします。取得したサンプリングデータが High パルスと Low パルスのどちらを示しているのかを確認するために、ステータス情報を確認する必要があります。UcntSetSamplingConfig 関数でサンプリングデータにステータス情報を付加するように設定しています。ステータス情報からラッチ直前の LA の状態(LABEF)が High の場合、そのときのサンプリングデータが High パルス幅となり、ラッチ直後の LA の状態(LAAFT)が High の場合、Low パルス幅となります。High パルス幅と Low パルス幅から、デューティ比を計算し、printf 関数で表示を行なっています。

最後にカウンタ製品の制御を終了するためには、UcntClose 関数を用います。

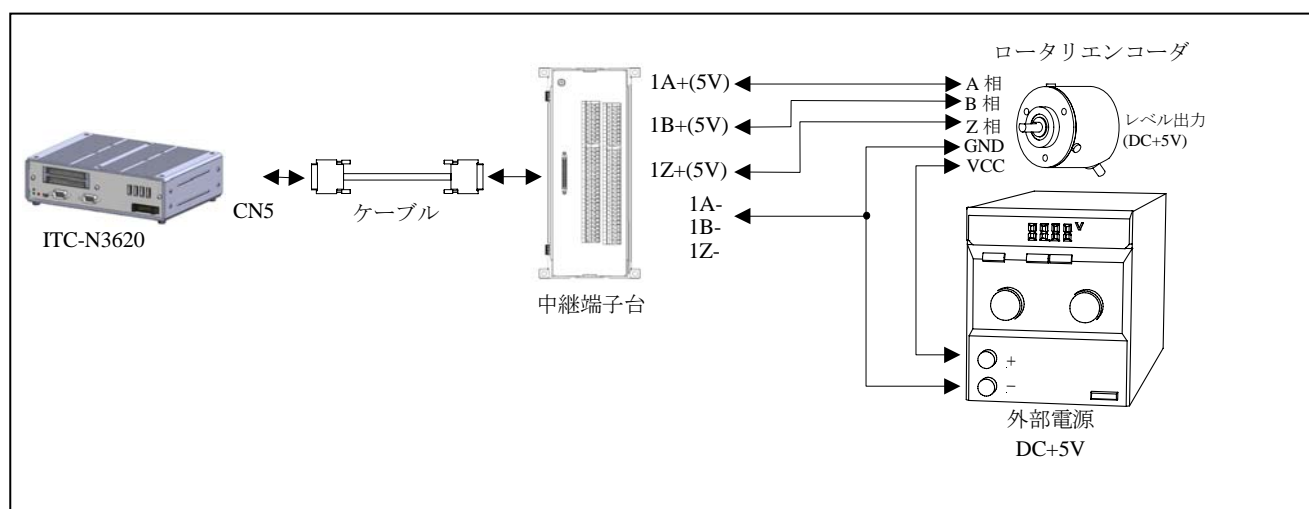
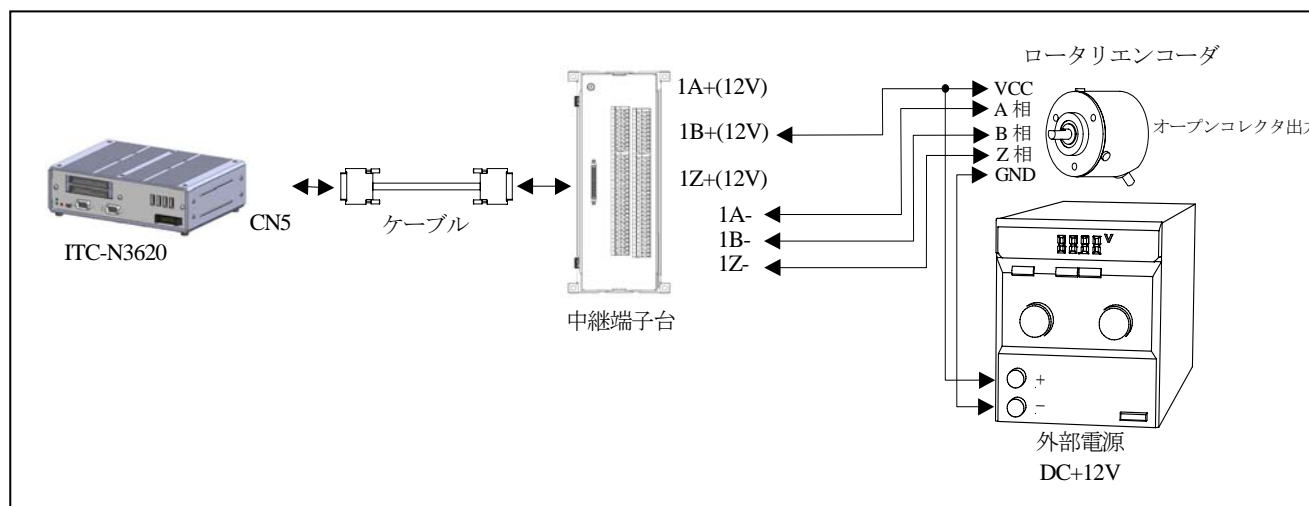
オープンしたカウンタ製品は、使用後は必ずクローズしてください。

★UcntClose関数を呼ばないと、どうなるか？

UcntClose関数を呼び出さないと、カウンタ製品はオープンしたままの状態になります。
オープン状態なので、UcntOpen関数を呼び出した場合、エラーが返ります。
オープンしたら必ずクローズしてください。

10.3.3 割り込み処理

環境図は下記になります。



Step1 プログラム作成

万能カウンタの割り込みイベントには、カウンタのキャリー、ボローや一致検出割り込みなどがあります。割り込みイベントが発生すると、あらかじめ登録しておいたコールバック関数が実行されます。割り込み処理などは、コールバック関数に記述します。

エディタを起動し、下記に示すプログラムを入力して、ファイル名を「ucntevent.c」として保存してください。

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

#include "ifucnt.h"

int isEvent = 0;

// コールバック関数
void EventProc(int nChannel, unsigned long ulEvent, unsigned long ulUser)
{
    if (ulEvent & 0x00000001) { // カウンタキャリーを検知
        printf("カウンタキャリー発生¥n");
    }

    if (ulEvent & 0x00000002) { // カウンタボローを検知
        printf("カウンタボロー発生¥n");
    }

    isEvent = 1;
}

int main(int argc, char *argv[])
{
    int nRet, nDevice;
    int nChannel;
    unsigned long dwChSel;
    unsigned long dwCountMode;
    unsigned long dwLoadMode;
    unsigned long dwLatchMode;
    unsigned long dwLoadData;
    unsigned long dwEventMask;

    // デバイス番号 1 のデバイスを制御
    nDevice = 1;

    // デバイスをオープン
    nRet = UcntOpen(nDevice);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
```

```
        goto EXIT_CLOSE;
    }

    // パルスカウントモードに設定
    nChannel    = 1;    // CH1
    dwCountMode = IFUCNT_COUNT_PHASE_1 | IFUCNT_DIR_NORMAL; // 位相差パルスカウントモ
ード 1 通倍, 通常方向(カウントアップ)
    dwLoadMode  = 0x10; // 同期エッジ
    dwLatchMode = 0x00; // ラッチ無効

    nRet = UcntSetPulseCountMode(nDevice, nChannel, dwCountMode, dwLoadMode, dwLatchMode);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // プリロードデータに 0 を設定
    dwLoadData = 0;
    nRet = UcntSetLoadData(nDevice, nChannel, dwLoadData);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // 割り込みイベントのマスクを設定
    dwEventMask = 0x00000003; // キャリー、ボローの割り込みイベントを有効
    nRet = UcntSetEventMask(nDevice, nChannel, dwEventMask);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // 割り込みイベント発生時に実行されるコールバック関数を登録
    nRet = UcntSetEvent(nDevice, EventProc, 0x1234);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // カウントを開始
    dwChSel = 0x01 << (nChannel - 1);
    nRet = UcntStartCount(nDevice, dwChSel, IFUCNT_CMD_START);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }

    // 割り込みイベントが発生するまで待機
    while (!isEvent) {
        sleep(1);
    }

    // コールバック関数を解除
    nRet = UcntKillEvent(nDevice);
    if (nRet != IFUCNT_ERROR_SUCCESS) {
        goto EXIT_CLOSE;
    }
```

```
}

// 割り込みイベントをすべて無効に設定
dwEventMask = 0x00000000;
nRet = UcntSetEventMask(nDevice, nChannel, dwEventMask);
if (nRet != IFUCNT_ERROR_SUCCESS) {
    goto EXIT_CLOSE;
}

EXIT_CLOSE:
// デバイスをクローズ
UcntClose(nDevice);

if (nRet != IFUCNT_ERROR_SUCCESS) {
    exit(EXIT_FAILURE);
}
return 0;
}
```

Makefile の作成

下記ソースを「Makefile」という名前で、ソースファイルと同一のディレクトリに保存してください。

```
all: ucntevent

ucntevent: ucntevent.o
    $(CC) ucntevent.o -o ucntevent -lgpg6320u

ucntevent.o: ucntevent.c
    $(CC) -Wall -c ucntevent.c -o ucntevent.o

clean:
    rm -f *.o ucntevent
```

Step2 コンパイル/実行

ソースコード・Makefile を保存したディレクトリに移動し、コマンドラインから下記コマンドを実行します。下記コマンドを実行することでコンパイルを行う事が出来ます。

```
make
```

コンパイルが成功すると、オブジェクトファイル「ucntevent.o」と実行ファイル「ucntevent」が作られます。

Error や Warning が表示された場合はコードの記述に誤りがあります。内容を見直してください。
プログラムを実行するには、以下のコマンドを実行します。

```
./ucntevent
```

プログラムの実行に成功すれば、以下の出力が行なわれます。

```
カウンタキャリー発生
```

Step3 プログラムの解説

カウンタ製品を制御する前段階として、カウンタ製品をオープンするUcntOpen関数を呼び出しています。
以下に引数と対応を示します。

```
int UcntOpen(  
    int          nDevice,          /* デバイス番号 */  
);
```

引数名	内 容
nDevice	オープンするデバイス番号を指定します。

この引数を与えて関数を呼び出すと、ドライバは引数に適合するカウンタ製品を検索し、合致した製品がある場合は、戻り値として0を返します。

プログラマは、このデバイス番号を使って以降のAPIの呼び出しを行います。

カウンタ製品を制御するためには、まずカウンタのモード設定を行ないます。

カウンタのモードには様々な種類があり、パルスカウントモード、平均周波数測定モード、周期測定モード、位相差幅測定モード、タイマモード、分周器モード、パルスジェネレーターモードがあります。

これらの設定を行なうには、以下の関数を使用します。

カウンタモード	関数
パルスカウントモード	UcntSetPulseCountMode関数
平均周波数測定モード	UcntSetFreqAvgMode関数
周期測定モード	UcntSetCycleMode関数
位相差幅測定モード	UcntSetPhaseDiffMode関数
タイマモード	UcntSetTimerMode関数
分周器モード	UcntSetFreqDividerMode関数
パルスジェネレーターモード	UcntSetPulseGeneratorMode関数

ここでは、パルスカウントモードの設定を行なうためのUcntSetPulseCountMode関数を使用します。

それぞれの引数で、設定を行なうチャンネル、カウンタモード、プリロードモード、ラッチモードを設定する事が出来ます。

ここでは、設定を行なうチャンネルを1とし、位相差パルスカウント・1通倍・通常方向(カウントアップ)、同期エッジ、ラッチ無効に設定しています。

次にプリロードデータの設定を行ないます。設定を行なうには、UcntSetLoadData関数を行います。ここでは、設定するチャンネルを1とし、プリロードデータを0としています。

次に、割り込みの設定を行ないます。

通常、割り込みは全てマスクされ、割り込みが発生しない状態となっています。

割り込みを発生させるためには、UcntSetEventMask関数を使用し、割り込みをアンマスクします。

ここでは、キャリア・ボローに対して割り込みをアンマスクしています。

割り込みマスクを解除したら、割り込みの発生を確認するために、コールバック関数の登録と、ユーザデータの登録を行ないます。

コールバック関数の登録と、ユーザデータの登録は、UcntSetEvent関数を使用します。

コールバック関数は、カウンタデバイスで割り込み発生時に呼び出される関数のプレースホルダです。(コールバック関数に戻り値はありません。)

割り込みが発生するたびに、UcntSetEvent関数で登録したコールバック関数が呼び出されます。

ユーザデータは、複数のデバイスで割り込みを登録した場合、どのデバイスに対する割り込みかを判別するときに使用します。

それぞれ、第2引数にコールバック関数EventProc、第3引数にユーザデータとして、12345678h、を指定して関数を実行しています。

割り込みの設定が終わると、カウンタを開始させます。カウンタを開始するには、UcntStartCount関数を使用します。開始するチャンネル、スタートモードを指定します。

開始するチャンネルは複数のチャンネルを一度に指定することができ、bit1~bit4にそれぞれCH1~CH4が割り当てられています。

ここでは、カウントを開始するチャンネルを1のみとし、スタートモードをソフトスタートとしています。

本プログラムでは、whileループを使用して割り込みの発生を待機しています。

whileループの判定条件であるisEventは、コールバック関数の中で書き換えられるため、割り込みが発生してコールバック関数が呼ばれると、whileループを抜け、次の処理に移るようになっています。

プリロードデータを0hに設定しているため、ロータリーエンコードよりカウントダウンのパルスが入力されれば、カウンタ値が0hからFFFFFFFFhに変化し、パルスカウンタボロー割り込みが発生します。カウンタ値がFFFFFFFFhからカウントアップのパルスが入力されれば、カウンタ値が0hになるため、パルスカウンタキャリア割り込みが発生します。

割り込みが発生すると、コールバック関数内で、割り込みが発生した要因を確認し、printf関数を使用して割り込み発生要因を表示しています。

次に、isEventに1をいれ、コールバック関数は処理を返します。

この時、コールバック関数内でisEventが更新されたため、メインルーチンでは割り込み発生を待機している処理を抜け、終了処理に移ります。

割り込みを使う必要がなくなった場合は、コールバック関数の登録を解除し、割り込みをマスクします。

コールバック関数の登録を解除し、割り込みをマスクするには、UcntKillEvent関数とUcntSetEventMask関数を使用します。

UcntKillEvent関数を実行することで、登録したコールバック関数を解除し、UcntSetEventMask関数の第3引数に0を指定することで、全ての割り込みの発生をマスクします。以後、割り込みが発生することはありません。

最後にカウンタ製品の制御を終了するためには、UcntClose関数を用います。

オープンしたカウンタ製品は、使用後は必ずクローズしてください。

★UcntClose関数を呼ばないと、どうなるか？

UcntClose関数を呼び出さないと、カウンタ製品はオープンしたままの状態になります。

オープン状態なので、UcntOpen関数を呼び出した場合、エラーが返ります。

オープンしたら必ずクローズしてください。

改訂履歴

Ver.	年 月	改 訂 内 容
1.0	2013 年 11 月	新規作成