
エンベデッドコンピュータAR2000シリーズ
フロントオペレーションモデル
RAS制御 チュートリアル (Linux編)

このページは空白です。

はじめに

このたびは、弊社のエンベデッドコンピュータ AR2000 シリーズをお買い求めいただき、誠にありがとうございます。

本製品は、省スペース設計の組み込み用コンピュータです。

本書は、本製品のシステム構築方法（Linux 編）を具体的な例に基づいて解説しています。

本書をご覧いただき、本製品を正しくお使いいただきますよう、お願いいたします。

2007 年 5 月

Microsoft および Windows は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。

Linux は、Linus Torvalds 氏の米国およびその他の国における登録商標または商標です。

Red Hat および Red Hat をベースとしたすべての商標とロゴは、米国およびその他の国における Red Hat, Inc. の商標または登録商標です。

Adobe、Reader は、Adobe Systems Incorporated（アドビシステムズ社）の登録商標です。

その他の社名および商品名は各社の登録商標または商標です。

All Rights Reserved, Copyright © PFU LIMITED 2006-2007

本書の読み方

本書の構成内容、本書での表記に関する注意事項などについて説明します。

●マニュアル体系

本製品には以下のマニュアルが用意されています。必要に応じてお読みください。

『取扱説明書』^(*1)

本製品の仕様や基本的な取り扱い方法、トラブルの対処方法について説明しています。必ずお読みください。

『BIOS 説明書』

添付のドライバ CD 内に PDF データとして格納されています^(*2)。

OS のインストール方法や、ハードウェア環境を設定するためのプログラムである BIOS Setup の仕様と操作方法について説明しています。OS をインストールする際や、ご購入時にあらかじめ設定されている BIOS の設定を変更する場合にお読みください。

『RAS 制御 API 説明書 (Windows 編、Linux 編)』

添付のドライバ CD 内に PDF データとして格納されています^(*2)。

各種 API の仕様について説明しています。

『RAS 制御チュートリアル (Windows 編、Linux 編 (本書))』

添付のドライバ CD 内に PDF データとして格納されています^(*2)。

具体的なプログラムを例に、本製品を使ったシステム構築について説明しています。

*1) 取扱説明書のみ、英語版が存在します。

*2) PDF 形式のファイルをご覧いただく際は、Adobe® Reader® が必要です。

Adobe® Reader® は、アドビシステムズ社のサイトから無償でダウンロードできます。

●本書の構成について

第1章 概要

本製品の概要について説明しています。

第2章 チュートリアル

RAS 制御 API を使用したアプリケーションの作成方法を、具体的なソースコード例を交えて説明しています。

●略称

本書では、以下の用語について省略して表記する場合がありますので、ご了承ください。

製品名称	略称
Microsoft [®] Windows [®] XP Professional operating system	Windows [®] XP
Microsoft [®] Windows [®] 2000 Professional operating system	Windows [®] 2000
Red Hat [®] Enterprise Linux v3	Linux
Red Hat [®] Enterprise Linux v4	

このページは空白です。

目次

はじめに	i
本書の読み方	ii
● マニュアル体系	ii
● 本書の構成について	iii
● 略称	iii
目次	v
第 1 章 概要	1
1.1 提供機能	1
1.2 開発環境	1
1.3 アプリケーションのコンパイル	2
第 2 章 チュートリアル	3
2.1 DIO 制御オープン、クローズ	3
2.2 基本的な DIO の入出力を行う	4
2.3 ウォッチドッグタイマ機能を使用する	7
2.4 リセット・アラーム情報を取得する	10
2.5 割り込み制御を行う	12
2.6 割り込みをトリガーとして DIO やシステム制御を行う	16

このページは空白です。

この章では、RAS 制御を行うアプリケーション開発に際して必要な情報について、説明します。

1.1 提供機能

本システムでは、アプリケーションから DIO 制御機能やシステム制御機能を利用するためのライブラリとして、各種 API を提供しています。

本 API では以下の機能を提供します。

- DIO 制御機能 : DIO 接点に対して入出力を行います。
- WatchDogTimeout 機能 : ウォッチドッグタイマの各種制御を行います。
- アラーム制御機能 : アラームの各種制御を行います。
- リセット・アラーム情報取得機能 : リセット、アラーム情報の取得・クリアを行います。
- 割り込み制御機能 : 接点入力とアラームの割り込みの制御を行います。
- 信号制御機能 : WDTOUT、ALMOUT、RSTOUT の信号の制御を行います。

本書では、主な API の使用方法をチュートリアル形式で説明します。

1.2 開発環境

本 API をアプリケーションで使用するためにはアプリケーション開発環境上に以下のファイルが必要となります。

- ar_dio.h : 定義ファイル
- libar_dio.so : DIO 制御、システム制御用ライブラリ

1.3 アプリケーションのコンパイル

本 API を使用するためには、アプリケーションに DIO 制御ライブラリをリンクする必要があります。

コンパイル時に、以下のようなリンクオプション (-lar_dio) を付けてリンクしてください。

```
gcc コンパイルオプション filename.c -o filename -lar_dio
```

次章からチュートリアルを始めます。

なお、ここで示すサンプルプログラムでは、簡略化のために関数呼び出しの復帰値に対するエラーチェックは省略しています。

また、本書では API の使用方法がすぐに理解できるように、サンプルプログラムを単純化しています。

実際のシステム用にアプリケーションを作成する場合は、各システムの要件にあった設計を行うようにしてください。

第2章 チュートリアル

この章では、チュートリアルに沿って、RAS 制御 API の使用方法について説明します。

2.1 DIO 制御オープン、クローズ

DIO 制御機能を使用するために必ず必要な処理が、定義ファイルのインクルードと、オープン、クローズ処理です。以下のルールに従ってください。

- アプリケーションのソースコードで、定義ファイル `ar_dio.h` をインクルードします。
- DIO ハードウェアにアクセスするために、最初にオープン用関数 `DioOpen()` を呼び出します。
- 処理を終了するときには、クローズ用関数 `DioClose()` を呼び出します。

以下にサンプルプログラムを示します。

```
1 /*
2  *   DIO_API サンプルプログラム (1)
3  *   オープンとクローズ
4  */
5 #include <unistd.h>
6 #include <ar_dio.h>
7
8 int main (int argc, char *argv[])
9 {
10     int fd;
11     unsigned int board_id = 0;           // DIO 制御装置 (DIO ボード) 0 を指定
12     int flag = 0;
13     int rtn;
14
15     // DIO 制御を開始する
16     fd = DioOpen(board_id, flag);
17
18     // DIO 制御処理やシステム制御処理を行う
19
20     // DIO 制御を終了する
21     rtn = DioClose(fd);
22
23     exit(rtn);
24 }
```

2.2 基本的な DIO の入出力を行う

ここでは基本的な DIO の入出力の方法を説明します。以下のサンプルプログラムは以下の処理を行っています。

- DIO 制御装置 (DIO ボード) 0 の出力接点 4 と 8 に対して出力を行う (信号のオン・オフを操作する)。
- DIO 制御装置 (DIO ボード) 0 の入力接点 12 から入力を行う (信号状態を読み取る)。
- 以前、オンにした出力接点 4、8 に加えて、出力接点 3、9 の信号をオンにする。

```
1 /*
2 *   DIO_API サンプルプログラム (2)
3 *   基本的な DIO 入出力 (ワードアクセス)
4 */
5 #include <unistd.h>
6 #include <ar_dio.h>
7
8 #define POS3   (1 << (3-1))           // 接点 3 のビット位置
9 #define POS4   (1 << (4-1))           // 接点 4 のビット位置
10 #define POS8   (1 << (8-1))          // 接点 8 のビット位置
11 #define POS9   (1 << (9-1))          // 接点 9 のビット位置
12 #define POS12  (1 << (12-1))        // 接点 12 のビット位置
13
14 int main (int argc, char *argv[])
15 {
16     int fd;
17     unsigned int board_id = 0;        // DIO 制御装置 (DIO ボード) 0 を指定
18     int flag = 0;
19     int rtn;
20     unsigned short set_value = 0;
21     unsigned short rtn_value = 0;
22
23     // DIO 制御を開始する
24     fd = DioOpen(board_id, flag);
25
26     // DIO 出力を行う
27     set_value = (POS4 | POS8);        // 出力接点 4 と 8 の位置のビットオン
28     rtn = DioOutPortW(fd, DIO_W1_16, set_value);
29
30     // DIO 入力を行う
31     rtn = DioInPortW(fd, DIO_W1_16, &rtn_value);
32
33     if (rtn_value & POS12) {          // 入力接点 12 の位置のビットをチェック
34         printf (" 入力接点 12 は ON 状態です。 \n");
```

```

35     } else {
36         printf (" 入力接点 12 は OFF 状態です。 \n" );
37     }
38
39     // 追加の DIO 出力の前に 10 秒待つ
40     sleep(10);
41
42     // DIO 出力を追加して行う
43     set_value |= (POS3 | POS9);           // 出力接点 3 と 9 の位置のビットオン
44     rtn = DioOutPortW(fd, DIO_W1_16, set_value);
45
46     // DIO 制御を終了する
47     rtn = DioClose(fd);
48
49     exit(rtn);
50 }

```

本サンプルプログラムの DIO 出力のメインとなる処理は、以下の部分です。

```

9 #define POS4  (1 << (4-1))           // 接点 4 のビット位置
10 #define POS8  (1 << (8-1))          // 接点 8 のビット位置
    :
27     set_value = (POS4 | POS8);       // 出力接点 4 と 8 の位置のビットオン
28     rtn = DioOutPortW(fd, DIO_W1_16, set_value);

```

ここでは 16 接点分を一度に処理するために、ワード型（16 ビット）のデータを扱う `DioOutPortW()` 関数を使用します。まず、オンにする接点を表すビットを `set_value` に立てます。ビットを立てなかった接点はオフとなることに注意してください。

`POS4` と `POS8` はサンプルプログラムの先頭で定義しています。一番右側のビットが接点 1 となり、以降左側にシフトしていきます。

`DioOutPortW()` 関数の引数の意味は、以下のとおりとなります。

第 1 引数の `fd` には、`DioOpen()` 関数によって返されたファイルディスクリプタを指定します。

第 2 引数は、操作する接点を含むアドレスを指定します。`DIO_W1_16` は接点 1 から 16 までを操作できます。

第 3 引数の `set_value` で、16 接点分のオンとオフを指定します。

次に、DIO 入力の処理を説明します。DIO 入力のメインとなる処理は以下の部分です。

```

12 #define POS12 (1 << (12-1))           // 接点 12 のビット位置
    :
31     rtn = DioInPortW(fd, DIO_W1_16, &rtn_value);
    :
33     if (rtn_value & POS12) {         // 入力接点 12 の位置のビットをチェック
    :
37     }

```

ここでも 16 接点分を一度に処理するために、ワード型（16 ビット）のデータを扱う `DioInPortW`（）関数を使用します。`DioInPortW`（）関数の引数の意味は以下のとおりとなります。

第 1 引数の `fd` には、`DioOpen`（）関数によって返されたファイルディスクリプタを指定します。

第 2 引数は、入力する接点を含むアドレスを指定します。`DIO_W1_16` は接点 1 から 16 までの状態を入力できます。

第 3 引数の `rtn_value` には、関数復帰後に 16 接点分のオンとオフの状態が返ります。`rtn_value` のポインタを指定することに注意してください。

`DioInPortW`（）関数からの復帰後、`rtn_value` にはオン状態の接点を表すビットが立っています。

入力接点 12 用のビットが立っているかをチェックすることで、入力接点 12 がオンかオフかがわかります。

さらに続けて、`DIO` 出力を行うコードは以下になります。ここでは、以前、オンにした出力接点 4、8 はオフにせず、追加で出力接点 3、9 の信号をオンにしています。

```

8 #define POS3 (1 << (3-1))           // 接点 3 のビット位置
    :
11 #define POS9 (1 << (9-1))         // 接点 9 のビット位置
    :
43     set_value |= (POS3 | POS9);     // 出力接点 3 と 9 の位置のビットオン
44     rtn = DioOutPortW(fd, DIO_W1_16, set_value);

```

第 1 引数、第 2 引数ともに最初に `DioOutPortW`（）関数で指定した値を使用しています。

第 3 引数の `set_value` には、以前、オンにした出力接点をオフにしないように、以前の値と今回指定する値との論理和を代入していることに注意してください。

以上で、基本的な `DIO` の入出力を行うチュートリアルを終了します。

2.3 ウォッチドッグタイマ機能を使用する

ウォッチドッグタイマ機能を使用すると、システムハングなどが発生してアプリケーションが動作できなくなった場合に、システムリセットを行ってシステムを再起動したり、WDTOUT 信号をオンにして外部に異常状態を通知することなどができます。

ウォッチドッグタイマ機能の処理の流れは以下のようになります。通常はデーモンのように常時動作しているプログラムとなります。

- 1 DioWDTStart () 関数でウォッチドッグタイムアウト監視を開始する。
- 2 タイムアウトにならない間隔で定期的に DioWDTContinue () 関数を呼び出し、ウォッチドッグタイマを更新することで、システムが動作していることをDIO 制御装置 (DIO ボード) に通知する。
- 3 アプリケーションを終了するときには、DioWDTStop () 関数でウォッチドッグタイムアウト監視を停止する。

ウォッチドッグタイマが更新されずに、ウォッチドッグタイムアウトが発生した場合、DIO 制御装置 (DIO ボード) はシステムリセットなどアプリケーションで指定した処理を行います。

以下にサンプルプログラムで具体例を説明します。このサンプルプログラムでは以下の処理を行っています。

- サンプルプログラムは起動後にデーモンプログラムとなる。
- 動作中、ウォッチドッグタイムアウト監視を行う。
- ウォッチドッグタイムアウト監視はタイムアウトを5秒に設定し、タイムアウトが発生した場合にはシステムリセットと WDTOUT 信号の発行を行う。
- 1秒間隔でウォッチドッグタイマの更新処理を行う。
- SIGTERM などのシグナルを受けた場合は、ウォッチドッグタイムアウト監視を停止し、プログラムを終了する。

```
1 /*
2 *   DIO_API サンプルプログラム (3)
3 *   ウォッチドッグタイマ制御
4 */
5 #include <unistd.h>
6 #include <signal.h>
7 #include <ar_dio.h>
8
9 int sig_catch = 0;
10
11 void sigHandler (int signum) {
12     sig_catch = 1;
13 }
```

```

14
15 int main (int argc, char *argv[])
16 {
17     unsigned int board_id = 0;           // DIO 制御装置 (DIO ボード) 0 を指定
18     int fd;
19     int flag = 0;
20     int rtn = 0;
21     unsigned int type;
22     unsigned int timer;
23     struct sigaction sa;
24
25     // デーモンプログラムになる
26     if(fork() == 0) {
27         int i;
28         int num_fds = getdtablesize();
29         for (i=0; i<= num_fds; i++) {
30             close(i);
31         }
32         setsid();
33     } else {
34         exit(0);
35     }
36
37     // DIO 制御を開始する
38     fd = DioOpen(board_id, flag);
39
40     // ウォッチドッグタイムアウト監視を開始する
41     timer = 5000;
42     // システムリセットと WDTOUT 信号の発行
43     type = DIO_WDT_RESET_SYS | DIO_WDT_WDTOUT;
44     rtn = DioWDTStart(fd, type, timer);
45
46     // シグナルハンドラを設定する
47     memset(&sa, 0, sizeof(sa));
48     sa.sa_handler = sigHandler;
49     sigaction(SIGHUP, &sa, NULL);
50     sigaction(SIGINT, &sa, NULL);
51     sigaction(SIGQUIT, &sa, NULL);
52     sigaction(SIGTERM, &sa, NULL);
53
54     // 定期的にウォッチドッグタイマを更新する
55     while (1) {
56         sleep(1);
57         if (sig_catch == 1) break;
58         rtn = DioWDTContinue(fd);
59     }
60
61     // ウォッチドッグタイムアウト監視を停止する

```

```
62     rtn = DioWDTStop(fd);
63
64     // DIO 制御を終了する
65     rtn = DioClose(fd);
66
67     exit(rtn);
68 }
```

まず、26 行目～35 行目の処理でデーモンプログラムとなります。

次に、ウォッチドッグタイムアウト監視を開始します。以下のコードとなります。

```
41     timer = 5000;
42     // システムリセットと WDTOUT 信号の発行
43     type = DIO_WDT_RESET_SYS | DIO_WDT_WDTOUT;
44     rtn = DioWDTStart(fd, type, timer);
```

第 1 引数の `fd` には、`DioOpen ()` 関数によって返されたファイルディスクリプタを指定します。

第 2 引数の `type` には、タイムアウト時に行う処理を指定します。ここではシステムリセットと `WDTOUT` 信号の発行を指定しています。

第 3 引数の `timer` には、5 秒を指定しています。

47 行目～52 行目の処理で、`SIGHUP`、`SIGINT`、`SIGQUIT`、`SIGTERM` シグナル受信時の処理を登録しています。

シグナル受信時に動作する `sigHandler ()` 関数 (11 行目から 13 行目) では、フラグ `sig_catch` を 1 に設定します。

次に、1 秒間隔で `DioWDTContinue ()` 関数を定期的呼び出し、ウォッチドッグタイマの更新を行います。事前に登録したシグナルを受けた場合、ループを抜けます。

```
55     while (1) {
56         sleep(1);
57         if (sig_catch == 1) break;
58         rtn = DioWDTContinue(fd);
59     }
```

最後に、ループを抜けたあと、ウォッチドッグタイム監視を停止し、プログラムを終了します。

```
62     rtn = DioWDTStop(fd);
```

以上で、ウォッチドッグタイマ機能を使用するチュートリアルを終了します。

2.4 リセット・アラーム情報を取得する

システムリセット・外部リセット要因やアラーム情報を把握する事で、異常発生時の要因やアラームに即した対処をアプリケーションで行う事が出来ます。

ここではリセット情報を取得するサンプルプログラムを説明します。このサンプルプログラムでは以下の処理を行っています。

- システムリセット要因情報を取得する。
- 標準出力にシステムリセット要因情報を表示する。
- 取得したすべてのシステムリセット要因情報をクリアする。

```
1 /*
2 *   DIO_API サンプルプログラム (5)
3 *   リセット要因の取得とクリア
4 */
5 #include <unistd.h>
6 #include <ar_dio.h>
7
8 int main (int argc, char *argv[])
9 {
10     int fd;
11     unsigned int board_id = 0;           // DIO 制御装置 (DIO ボード) 0 を指定
12     int flag = 0;
13     int rtn;
14     int type;
15     unsigned int rtn_status = 0;
16
17     // DIO 制御を開始する
18     fd = DioOpen(board_id, flag);
19
20     // リセット要因を取得する
21     type = DIO_RESET_SYS;                // システムリセット情報を指定
22     rtn = DioGetResetStatus(fd, type, &rtn_status);
23
24     if (rtn_status & DIO_RESET_POWER_SYS) {
25         printf (" 電源オン / オフ状態を検知 \n");
26     }
27     if (rtn_status & DIO_RESET_SYSTEM_SYS) {
28         printf (" システムリセット状態を検知 \n");
29     }
30     if (rtn_status & DIO_RESET_WDT_SYS) {
31         printf (" ウォッチドッグリセット状態を検知 \n");
32     }
```

```
33     if (rtn_status & DIO_RESET_ALARM1_SYS) {
34         printf (" アラーム 1 からのリセット状態を検知 \n");
35     }
36     if (rtn_status & DIO_RESET_ALARM2_SYS) {
37         printf (" アラーム 2 からのリセット状態を検知 \n");
38     }
39     if (rtn_status & DIO_RESET_ALARM3_SYS) {
40         printf (" アラーム 3 からのリセット状態を検知 \n");
41     }
42     if (rtn_status & DIO_RESET_ALARM4_SYS) {
43         printf (" アラーム 4 からのリセット状態を検知 \n");
44     }
45     // 取得された要因をすべてクリアする
46     rtn = DioClearResetStatus(fd, type, rtn_status);
47
48     // DIO 制御を終了する
49     rtn = DioClose(fd);
50
51     exit(rtn);
52 }
```

システムリセット要因情報を取得するコードは以下のようになります。

```
21     type = DIO_RESET_SYS;           // システムリセット情報を指定
22     rtn = DioGetResetStatus(fd, type, &rtn_status);
```

第 1 引数の fd には、DioOpen () 関数によって返されたファイルディスクリプタを指定します。

第 2 引数の type には、DIO_RESET_SYS を指定して、システムリセット情報の取得を指示します。

第 3 引数の rtn_status には、関数復帰後にシステムリセット要因情報が返ります。rtn_status のポインタを指定することに注意してください。

次に、システムリセット要因情報をチェックして、標準出力に表示します。

ここではシステムリセット要因情報を表示しているだけですが、実際のシステムではそのシステムやアプリケーション動作の要件に沿った処理を行ってください。

```
24     if (rtn_status & DIO_RESET_POWER_SYS) {
25         printf (" 電源オン / オフ状態を検知 \n");
26     }
```

:

最後に、すべてのシステムリセット要因情報をクリアします。

```
46     rtn = DioClearResetStatus(fd, type, rtn_status);
```

第1引数、第2引数ともに `DioGetResetStatus()` 関数で指定した値を使用しています。

第3引数のクリアする要因には、`DioGetResetStatus()` 関数で取得した要因 (`rtn_status`) をそのまま指定しています。

以上で、リセット・アラーム情報を取得するチュートリアルを終了します。

2.5 割り込み制御を行う

入力接点がオンになった場合やアラームなどが発生した場合、割り込みをあげて、アプリケーションに対して通知できます。

この機能を使用するとアプリケーションから入力接点の状態などを定期的に読みに行く必要がなくなり、事象発生時にその事象に則した処理を行うことができます。

割り込み制御の処理の流れは基本的に以下ようになります。

- 1 `DioEnableInterrupt()` 関数により割り込みの発生を検知する設定を行います。
- 2 `select()` 関数により、割り込みの発生を待ち合わせます。
- 3 `DioGetInterrupt()` 関数により、割り込み要因を取得します。

以下にサンプルプログラムで具体例を説明します。このサンプルプログラムでは以下の処理を行っています。

- 全割り込みに対して、割り込み検知の設定をする。
- 割り込みを待ち合わせる。
- 割り込みが発生したら、割り込み要因を取得して、その情報を標準出力に表示する。

```
1 /*
2  *   DIO_API サンプルプログラム (6)
3  *   割り込み制御
4  */
5 #include <sys/select.h>
6 #include <sys/time.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <ar_dio.h>
10
11 int check_interrupt(int);
12
```

```
13 int main (int argc, char *argv[])
14 {
15     unsigned int board_id = 0;           // DIO 制御装置 (DIO ボード) 0 を指定
16     int fd;
17     int flag = 0;
18     int rtn;
19     unsigned int set;
20     fd_set readfds;
21     int max_fds = 0;
22
23     // DIO 制御を開始する
24     fd = DioOpen(board_id, flag);
25     if (fd > max_fds) max_fds = fd;
26
27     // 割り込み検知の設定をする
28     set = DIO_INT_ALL;                   // 全割り込みを検知
29     rtn = DioEnableInterrupt(fd, set);
30
31     FD_ZERO(&readfds);
32     FD_SET(fd, &readfds);
33     // 割り込みを待ち合わせる
34     rtn = select(max_fds + 1, &readfds, NULL, NULL, NULL);
35
36     if (rtn > 0) {
37         if (FD_ISSET(fd, &readfds)) {
38             rtn = check_interrupt(fd);
39         }
40     }
41     // DIO 制御を終了する
42     rtn = DioClose(fd);
43
44     exit(rtn);
45 }
46
47 int check_interrupt(int fd)
48 {
49     int rtn;
50     unsigned int rtn_value;
51
52     // 割り込み要因を取得する
53     rtn = DioGetInterrupt(fd, &rtn_value);
54
55     if (rtn_value & DIO_INT_DIO_RSTIN) {
56         printf ("DIO 部リセット入力からの割り込み \n");
57     }
58     if (rtn_value & DIO_INT_DI1) {
59         printf ("汎用入力 (DI1) からの割り込み \n");
60     }
}
```

```
61     if (rtn_value & DIO_INT_DI2) {
62         printf (" 汎用入力 (DI2) からの割り込み \n");
63     }
64     if (rtn_value & DIO_INT_DI3) {
65         printf (" 汎用入力 (DI3) からの割り込み \n");
66     }
67     if (rtn_value & DIO_INT_DI4) {
68         printf (" 汎用入力 (DI4) からの割り込み \n");
69     }
70     if (rtn_value & DIO_INT_DI5) {
71         printf (" 汎用入力 (DI5) からの割り込み \n");
72     }
73     if (rtn_value & DIO_INT_DI6) {
74         printf (" 汎用入力 (DI6) からの割り込み \n");
75     }
76     if (rtn_value & DIO_INT_DI7) {
77         printf (" 汎用入力 (DI7) からの割り込み \n");
78     }
79     if (rtn_value & DIO_INT_DI8) {
80         printf (" 汎用入力 (DI8) からの割り込み \n");
81     }
82     if (rtn_value & DIO_INT_DI9) {
83         printf (" 汎用入力 (DI9) からの割り込み \n");
84     }
85     if (rtn_value & DIO_INT_DI10) {
86         printf (" 汎用入力 (DI10) からの割り込み \n");
87     }
88     if (rtn_value & DIO_INT_DI11) {
89         printf (" 汎用入力 (DI11) からの割り込み \n");
90     }
91     if (rtn_value & DIO_INT_DI12) {
92         printf (" 汎用入力 (DI12) からの割り込み \n");
93     }
94     if (rtn_value & DIO_INT_DI13) {
95         printf (" 汎用入力 (DI13) からの割り込み \n");
96     }
97     if (rtn_value & DIO_INT_DI14) {
98         printf (" 汎用入力 (DI14) からの割り込み \n");
99     }
100    if (rtn_value & DIO_INT_DI15) {
101        printf (" 汎用入力 (DI15) からの割り込み \n");
102    }
103    if (rtn_value & DIO_INT_DI16) {
104        printf (" 汎用入力 (DI16) からの割り込み \n");
105    }
106    return 0;
107 }
```

まず、各割り込みの発生をアプリケーションで検知するか、検知しないかの設定を行います。以下のようなコードになります。

```
28     set = DIO_INT_ALL;           // 全割り込みを検知
29     rtn = DioEnableInterrupt(fd, set);
```

第1引数の fd には、DioOpen () 関数によって返されたファイルディスクリプタを指定します。

第2引数の set には、DIO_INT_ALL を設定して、全割り込みの検知を指定します。

次に、select () 関数によって割り込みの発生を待ち合わせます。

```
31     FD_ZERO(&readfds);
32     FD_SET(fd, &readfds);
33     // 割り込みを待ち合わせる
34     rtn = select(max_fds + 1, &readfds, NULL, NULL, NULL);
```

割り込みを受けて、select () 関数から復帰したら、check_interrupt () 関数を呼び出します。

```
36     if (rtn > 0) {
37         if (FD_ISSET(fd, &readfds)) {
38             rtn = check_interrupt(fd);
39         }
40     }
```

check_interrupt () 関数ではまず、割り込み要因を読み込みます。

```
53     rtn = DioGetInterrupt(fd, &rtn_value);
```

第1引数の fd には、DioOpen () 関数によって返されたファイルディスクリプタを指定します。

第2引数の rtn_value には、関数復帰後に割り込み要因情報が返ります。rtn_value のポインタを指定することに注意してください。

最後に、読み込んだ割り込み要因の情報を標準出力に表示します。

```
55     if (rtn_value & DIO_INT_ALARM) {
56         printf (" アラームからの割り込み \n");
57     }
```

:

ここでは割り込み要因情報を表示しているだけですが、実際のシステムでは割り込みを契機として、次の処理へと続いていくはずです。

そのシステムやアプリケーション動作の要件に沿った処理を行ってください。

以上で、割り込み制御を行うチュートリアルを終了します。

2.6 割り込みをトリガーとして DIO やシステム制御を行う

最後に、割り込みをトリガーとして、DIO 制御やシステム制御を行うプログラム例を示します。

ここではもう細かな説明は行いません。本 API の使用方法の理解に役立ててください。

このサンプルプログラムでは以下の処理を行っています。

- サンプルプログラムは起動後にデーモンプログラムとなる。
- 全アラーム信号 1、2、3、4 にアラームが発生した場合、割り込みを発生させるように設定する。
- 割り込み検知の設定をする。
- SIGTERM などのシグナルを受けた場合は、出力接点をすべてオフとし、プログラムを終了する。
- 割り込みを待ち合わせる。
- 割り込みが発生したら、割り込み要因を取得する。

割り込みを受けたとき、割り込み種別によって以下の処理を行います。

- 割り込みがアラーム発生によりあがった場合、アラーム情報を読み込み、アラームの重大度により管理者コールを行う。なお、この例ではアラーム信号 1 のアラームは非重大、それ以外は重大と想定しています。
- DIO 部リモートリセット入力による割り込みの場合、DIO 制御装置 (DIO ボード) が初期化されてしまい以前の設定が無効となるので、割り込み検知の設定を再設定する。
- 入力接点 1 からの割り込みの場合、出力接点 4 と 8 をオンにする。
- 入力接点 2 からの割り込みの場合、出力接点 4 と 8 をオフにする。
- 入力接点 3 からの割り込みの場合、割り込み検知の設定を解除し、プログラムを終了する。
- 入力接点 4 からの割り込みの場合、信号制御で外部リセット出力を行う。

```
1 /*
2  *   DIO_API サンプルプログラム (7)
3  *   割り込みをトリガーとした DIO 制御やシステム制御
4  */
5 #include <unistd.h>
6 #include <signal.h>
7 #include <sys/select.h>
8 #include <sys/time.h>
9 #include <sys/types.h>
10 #include <ar_dio.h>
11
12 int check_interrupt(int, unsigned short *);
13 int read_alarm(int);
14 int enable_interrupt(int);
15 int dio_on(int, unsigned short *);
16 int dio_off(int, unsigned short *);
17 int disable_interrupt(int);
18 int signal_cntl(int);
19 void call_administrator(void);
20
21 #define POS4   (1 << (4-1))           // 接点 4 のビット位置
22 #define POS8   (1 << (8-1))           // 接点 8 のビット位置
23
24 int alarm1_count = 0;
25 int break_loop = 0;
26 int sig_catch = 0;
27
28 void sigHandler (int signum) {
29     sig_catch = 1;
30 }
31
32 int main (int argc, char *argv[])
33 {
34     unsigned int board_id = 0;        // DIO 制御装置 (DIO ボード) 0 を指定
35     int fd;
36     int flag = 0;
37     int rtn;
38     unsigned int set;
39     unsigned short set_value = 0;
40     struct sigaction sa;
41     fd_set readfds;
42     int max_fds = 0;
43
44     // デモンプログラムになる
45     if(fork() == 0) {
46         int i;
47         int num_fds = getdtablesize();
```

```

48         for (i=0; i<= num_fds; i++) {
49             close(i);
50         }
51         setsid();
52     } else {
53         exit(0);
54     }
55     // DIO 制御を開始する
56     fd = DioOpen(board_id, flag);
57     if (fd > max_fds) max_fds = fd;
58
59     // アラーム設定を行う
60     rtn = DioAlarmSet(fd, 1, DIO_ALM_INTERRUPT);
61     rtn = DioAlarmSet(fd, 2, DIO_ALM_INTERRUPT);
62     rtn = DioAlarmSet(fd, 3, DIO_ALM_INTERRUPT);
63     rtn = DioAlarmSet(fd, 4, DIO_ALM_INTERRUPT);
64
65     // 割り込み検知の設定をする
66     set = DIO_INT_SYS_ALL | DIO_INT_DI1 | DIO_INT_DI2 | DIO_INT_DI3 |
DIO_INT_DI4;
67     rtn = DioEnableInterrupt(fd, set);
68
69     // シグナルハンドラを設定する
70     memset(&sa, 0, sizeof(sa));
71     sa.sa_handler = sigHandler;
72     sigaction(SIGHUP, &sa, NULL);
73     sigaction(SIGINT, &sa, NULL);
74     sigaction(SIGQUIT, &sa, NULL);
75     sigaction(SIGTERM, &sa, NULL);
76
77     while (1) {
78         FD_ZERO(&readfds);
79         FD_SET(fd, &readfds);
80
81         // 割り込みを待ち合わせる
82         rtn = select(max_fds + 1, &readfds, NULL, NULL, NULL);
83
84         if (rtn > 0) {
85             if (FD_ISSET(fd, &readfds)) {
86                 rtn = check_interrupt(fd, &set_value);
87             }
88         }
89         if ((break_loop == 1) || (sig_catch == 1)) break;
90     }
91
92     // 出力接点をすべてオフとする
93     set_value = 0;
94     rtn = DioOutPortW(fd, DIO_W1_16, set_value);

```

```
95
96 // DIO 制御を終了する
97 rtn = DioClose(fd);
98 exit(rtn);
99 }
100
101 int check_interrupt(int fd, unsigned short *set_valuep)
102 {
103     int rtn;
104     unsigned int rtn_value;
105
106     // 割り込み要因を取得する
107     rtn = DioGetInterrupt(fd, &rtn_value);
108
109     // アラーム情報を取得し、アラームに則した処理を行う
110     if (rtn_value & DIO_INT_ALARM) {
111         rtn = read_alarm(fd);
112     }
113     // 割り込み検知の設定を再設定する
114     if (rtn_value & DIO_INT_DIO_RSTIN) {
115         rtn = enable_interrupt(fd);
116     }
117     // 出力接点 4 と 8 をオンにする
118     if (rtn_value & DIO_INT_DI1) {
119         rtn = dio_on(fd, set_valuep);
120     }
121     // 出力接点 4 と 8 をオフにする
122     if (rtn_value & DIO_INT_DI2) {
123         rtn = dio_off(fd, set_valuep);
124     }
125     // 割り込み検知の設定を解除する
126     if (rtn_value & DIO_INT_DI3) {
127         rtn = disable_interrupt(fd);
128     }
129     // 信号制御を行う
130     if (rtn_value & DIO_INT_DI4) {
131         rtn = signal_cntl(fd);
132     }
133     return rtn;
134 }
135
136 int read_alarm(int fd)
137 {
138     int rtn;
139     unsigned int rtn_status = 0;
140
141     // アラームの発生したアラーム信号の識別番号を取得する
142     rtn = DioGetAlarmStatus(fd, &rtn_status);
```

```

143
144     if (rtn_status & DIO_ALARM1) {
145         // アラーム 1 は 3 回の発生で管理者コールを行う
146         alarm1_count++;
147         if (alarm1_count >= 3) {
148             call_administrator();
149             alarm1_count = 0;
150         }
151     }
152     if (rtn_status & (DIO_ALARM2 | DIO_ALARM3 | DIO_ALARM4)) {
153         // アラーム 2、3、4 は発生したらすぐに管理者コールを行う
154         call_administrator();
155     }
156
157     // 取得されたアラーム情報をすべてクリアする
158     rtn = DioClearAlarmStatus(fd, rtn_status);
159     return rtn;
160 }
161
162 int enable_interrupt(int fd)
163 {
164     int rtn, set;
165
166     // 割り込み検知の設定を再設定する
167     set = DIO_INT_SYS_ALL | DIO_INT_DI1 | DIO_INT_DI2 | DIO_INT_DI3 |
DIO_INT_DI4;
168     rtn = DioEnableInterrupt(fd, set);
169     return rtn;
170 }
171
172 int dio_on(int fd, unsigned short *set_valuep)
173 {
174     int rtn;
175
176     // 出力接点 4 と 8 をオンにする
177     *set_valuep |= (POS4 | POS8);
178     rtn = DioOutPortW(fd, DIO_W1_16, *set_valuep);
179     return rtn;
180 }
181
182 int dio_off(int fd, unsigned short *set_valuep)
183 {
184     int rtn;
185
186     // 出力接点 4 と 8 をオフにする
187     *set_valuep &= ~(POS4 | POS8);
188     rtn = DioOutPortW(fd, DIO_W1_16, *set_valuep);
189     return rtn;

```

```
190 }
191
192 int disable_interrupt(int fd)
193 {
194     int rtn, set;
195
196     // 割り込み検知の設定を解除する
197     set = DIO_INT_SYS_ALL | DIO_INT_DI1 | DIO_INT_DI2 | DIO_INT_DI3 |
DIO_INT_DI4;
198     rtn = DioDisableInterrupt(fd, set);
199     break_loop = 1;
200     return rtn;
201 }
202
203 int signal_cntl(int fd)
204 {
205     int rtn;
206
207     // 外部リセット出力を行う
208     rtn = DioSignalCntl(fd, DIO_SIG_RSTOUT, DIO_ASSERT);
209     return rtn;
210 }
211
212 void call_administrator()
213 {
214     // 管理者に通知する処理を行う
215 }
```

このページは空白です。

**エンベデッドコンピュータ AR2000 シリーズ
フロントオペレーションモデル
RAS 制御 チュートリアル (Linux 編)
P3XU-E571-02Z0**

発行日 2007年5月

発行責任 株式会社 PFU

- 本書の内容は、改善のため事前連絡なしに変更することがあります。
- 本書に記載されたデータの使用に起因する、第三者の特許権およびその他の権利の侵害については、当社はその責を負いません。
- 無断転載を禁じます。