マニュアル

産業用組込み PC EC4A IoT シリーズ用

Linux ディストリビューション

『Algonomix6』について

目　次

はじめに

第１章 概要

第２章 システム構成

第３章 開発環境

第４章 産業用組込みPC EC4A IoT シリーズについて

第５章 システムリカバリ

付録

はじめに

この度は、アルゴシステム製品をお買い上げいただきありがとうございます。

弊社製品を安全かつ正しく使用していただく為に、お使いになる前に本書をお読みいただき、十分に理解していただくようお願い申し上げます。

１）　お願いと注意

本書では、産業用組込み PC EC4A IoT シリーズ（以降 **EC4A シリーズ**）用 Linux ディストリビューション（以降 **Algonomix6**）に特化した部分について説明します。一般的な Linux についての詳細は省略させていただきます。Linux に関する資料および文献は、現在インターネット上や書籍など多数ございます。これらの書籍等と併せて本書をお読みください。

２）　保証について

Algonomix6 の動作は出荷パッケージのバージョンでのみ動作確認しております。Algonomix6 はお客様でソースの改変、ライブラリの追加と変更、プログラム設定の変更等を行うことができますが、これらの変更を行われた場合は動作保証することができません。

第１章　概要

本章では、Algonomix6 の具体的な内容を説明する前に、Algonomix6 の概要について説明します。

１－１　Algonomix6 とは

「Linux」とは、Linux カーネルのみを指す言葉です。しかし、Linux カーネルのみでは、オペレーティングシステム（以下 OS）としての役割を果たすことができません。OS として使うには、Linux カーネルのほかに、以下のような各種ソフトウェアパッケージと併せて使用する必要があります。

- シェル（bash、ash、csh、tcsh、zsh、pdksh、・・・・）
- util-linux（init、getty、login、reset、fdisk、・・・・）
- procps（ps、pstree、top、・・・・）
- GNU coreutils（ls、cat、mkdir、rmdir、cut、chmod、・・・・）
- GNU grep、find、diff
- GNU libc
- 各種基本ライブラリ（ncurses、GDBM、zlib・・・・）
- X Window System

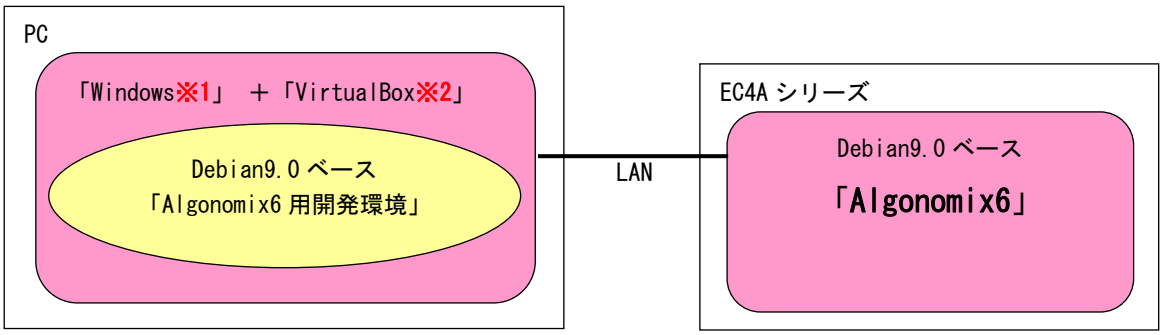
Linux カーネルといくつかの必要なソフトウェアパッケージをまとめて、OS として使えるようにしたものを Linux ディストリビューションといいます。

最初に述べましたとおり、「Linux」という言葉は、本来カーネルを指す言葉です。そのため、「カーネルとしての Linux」と「OS としての Linux」を厳密には区別する必要がありますが、本書では「Linux」とは「OS としての Linux」を指す言葉として使用します。

Algonomix6 は、「Debian 9.0」という Linux ディストリビューションをベースにした、EC4A シリーズ用の Linux ディストリビューションです。Algonomix6 は、「Debian 9.0」に EC4A シリーズ用の独自の I/O ドライバを組込んだものです。

パソコン上に Algonomix6 用の開発環境をインストールすることで、Algonomix6 用のソフトウェアを開発することができます。

Algonomix6 用開発環境イメージを図 1-1-1 に示します。



※注１：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。
※注２：VirtualBox は、米国 Oracle Corporation, Inc.の米国およびその他の国における商標または登録商標です。

図 1-1-1. Algonomix6 の開発環境

開発環境の詳細については、別紙『Algonomix6_開発環境ユーザーズマニュアル』を参照してください。

１－２　Linux の仕組み

Linux のソフトウェア構成を図 1-2-1 に示します。

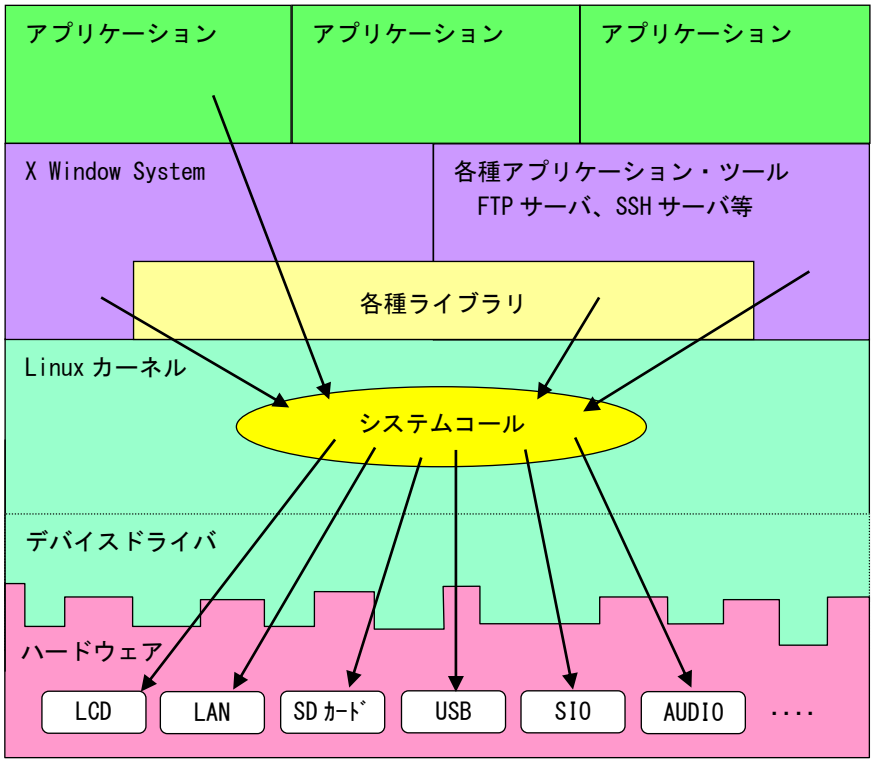


図 1-2-1. Linux ソフトウェア構成図

OS として重要な役割の一つに、**ハードウェアアクセスの複雑さを隠し、統一されたプログラミングインターフェース（システムコールや API と呼ばれる）をアプリケーションに提供する**というものがあります。Linux ではハードウェアを制御する為にドライバに関連付けられた「デバイスファイル」を読み書きすることで制御します。これは UNIX 系 OS の大きな特徴であり、ファイルを扱う感覚でハードウェアを制御することができます。Linux の代表的なシステムコールとして、open、close、read、write 等があります。これらのシステムコールは特別な呼び方をしているわけではなく、関数の呼び出しと同じように呼び出すことができます。

もう一つ OS の重要な役割として、**CPU 時間、メモリ、ネットワーク等のリソースをプログラムやプロセス、スレッドに分配する**というものもあります。これは Linux カーネルが処理しており、アプリケーション作成時に特に意識する必要はありません。図 1-2-1 にあるような X Window System や、SSH サーバや FTP サーバもプロセスの一つです。CPU 時間やメモリなどのリソースには限りがある為、複数のプロセスを同時に実行すると、それぞれのパフォーマンスは落ちます。そのため、必要最低限のプロセスで実行効率のよいプログラムを作成する必要があります。

第２章　システム構成

２－１　Algonomix6 パッケージについて

Algonomix6 であらかじめインストールされているパッケージを表 2-1-1 に示します。ただし Debian 9.0 の基本パッケージとしてインストールされているパッケージは除きます。

表 2-1-1. プリインストールパッケージ一覧

パッケージ名	内容	バージョン
linux-image-4.9.30-rt20-asdatom	Linux Kernel 本体	4.9.30-1
Xfce	デスクトップ環境	
ssh	telnet よりセキュリティの高いシェルクライアントおよびサーバ	1:7.4p1-10
gdbserver	GNU デバッガ	
aptitude	APT 用パッケージ管理ツール	
apache2	Web サーバ	
chromium-browser	オープンソースのウェブブラウザ	
xinput-calibrator	X.org 向けの タッチスクリーンキャリブレーションプログラム	

Algonomix6 では、Debian 9.0 の基本パッケージと本書に表記したパッケージが入った環境でのみ動作を確認しています。そのため、パッケージの追加に制限をかけています。
詳細に関しては、「2-1-1 パッケージについて」を参照してください。

Algonomix6 でインストール済みのパッケージ一覧を表示する為には下記手順を実行します。

① 画面下部中央の  をクリックするか、左上の「アプリケーション」→「ターミナルエミュレータ」をクリックしてください。図 2-1-1 のような、ターミナル画面が表示されます。

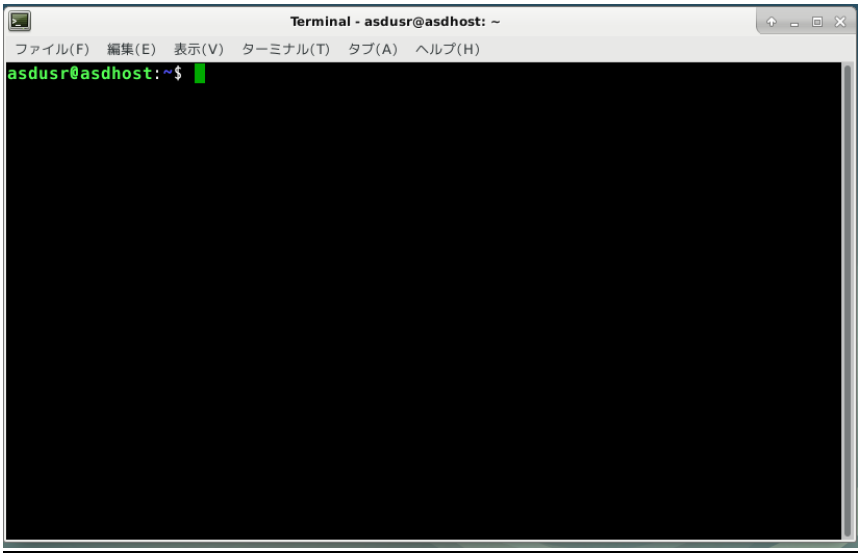


図 2-1-1. ターミナル画面

② 下記のコマンドを実行することで、現在インストールされているすべてのパッケージの名前が一覧で表示されます。

```
$ dpkg -l
```

または

```
$ dpkg --list
```

一覧の表示内容は次のような形式で表示されています。

```
+++-=============-=================-=====-=======================================
ii  adduser       3.113+nmu3ubuntu3  all   add and remove users and groups
```

1　2　3　4　5

1：パッケージのインストール状況
　1 文字目：要望　：(U)不明/(I)インストール/(R)削除/(P)完全削除/(H)維持
　2 文字目：状態　：(N)無/(I)インストール済/(C)設定/(U)展開/(F)設定失敗
　　(H)半インストール/(W)トリガ待ち/(T)トリガ保留
　3 文字目:エラー　:(空欄)無/(H)維持/(R)要再インストール/X=両方(状態,エラーの大文字=異常)

2：パッケージ名

3：パッケージのバージョンおよびリビジョン

4：パッケージが対応しているアーキテクチャ

5：パッケージの 1 行説明

２－１－１　パッケージについて

Algonomix6 では、Debian 9.0 の基本パッケージと本書に表記したパッケージが入った環境でのみ動作を確認しています。

そのため、インターネットからのパッケージの追加に制限をかけています。

以下にパッケージをさらに追加したい場合の制限の解除方法を示しますが、設定を変更してパッケージを追加された場合、動作の保証は致しかねますので、あらかじめご了承頂きますようお願い申し上げます。

Algonomix6 では、インターネットへアクセスするリポジトリを無効にすることでパッケージのインストールに制限をかけています。

制限の解除方法としては、/etc/apt/sources.list を直接編集するか、「Synaptic パッケージマネージャ」を使って間接的に編集することが出来ます。ここでは、後者を紹介します。

また、Debian では、インストールした時点の全パッケージを収納した DVD-ROM 3 枚組が用意されています。ネットワークリポジトリの最新状態にアップデートせずに、インストール時のパッケージが欲しい場合は、弊社までお問い合わせください。

① [アプリケーション]→[設定]→[Synaptic パッケージマネージャ]をクリックします。

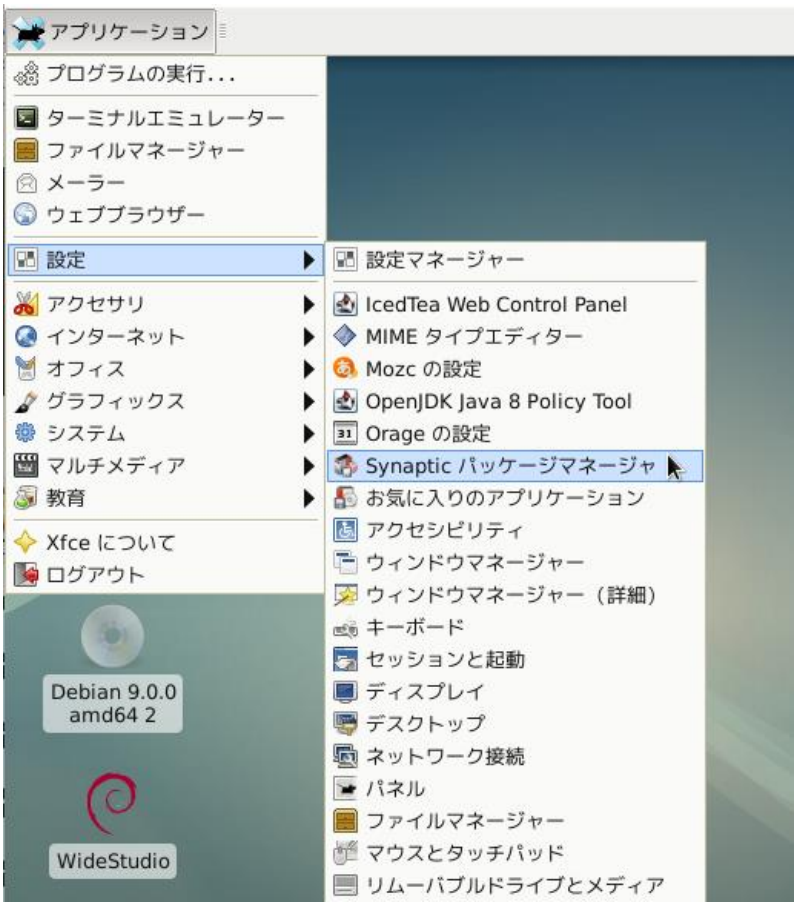


図 2-1-1-1. Sysnaptic パッケージマネージャの起動

② パッケージのインストール等を行うには管理者権限になる必要があります。図 2-1-1-2 のような、認証ウインドウが表示されるので、管理者パスワードを入力して「認証する」をクリックしてください。

デフォルト管理者パスワード　：rootroot

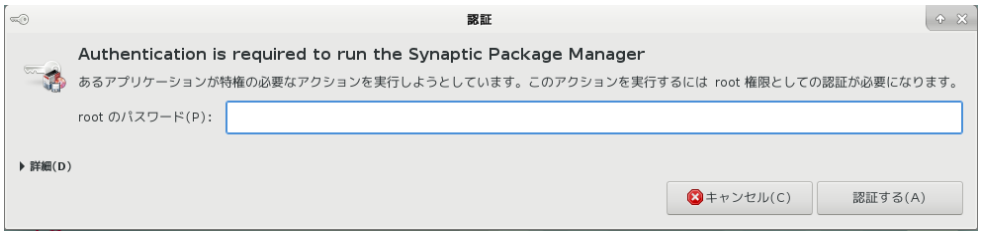


図 2-1-1-2. 管理者権限認証画面

③ 初回起動時は、図 2-1-1-3 のような、紹介画面が表示されます。「Close」をクリックしてください。

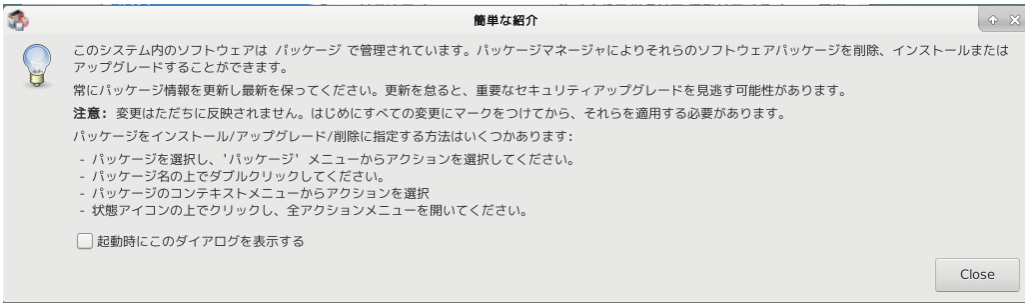


図 2-1-1-3. 紹介画面

④　Synaptic メイン画面を図 2-1-1-4 に示します。

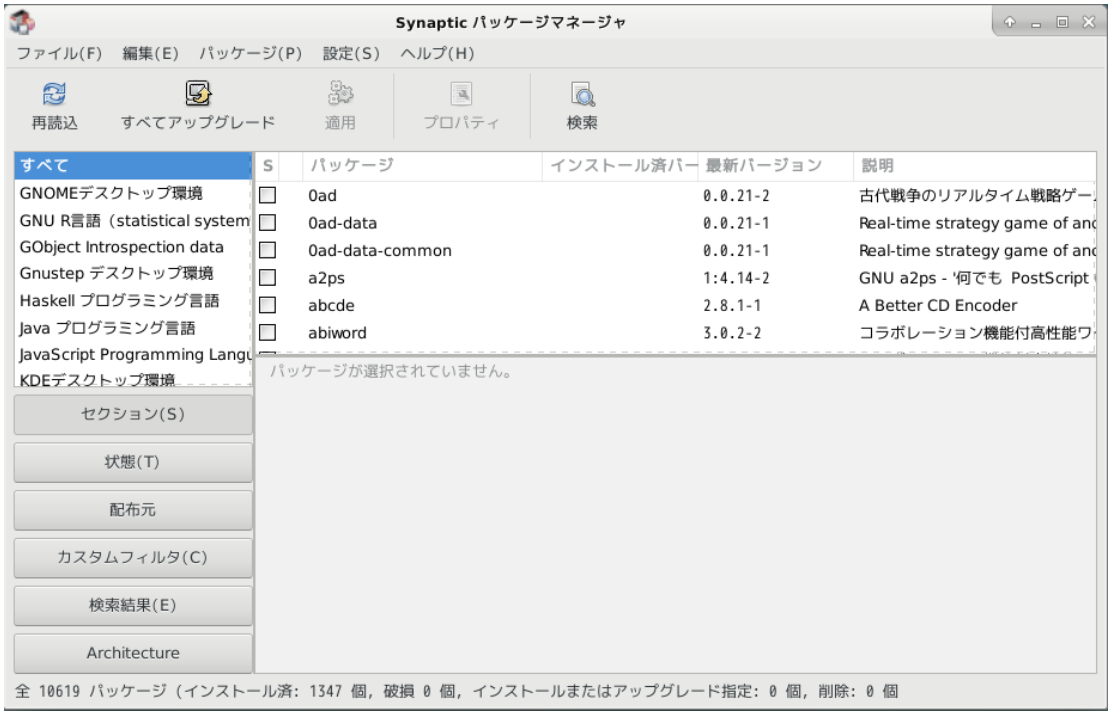


図 2-1-1-4. Synaptic パッケージマネージャメイン画面

⑤　「設定」→「リポジトリ」をクリックしてください。

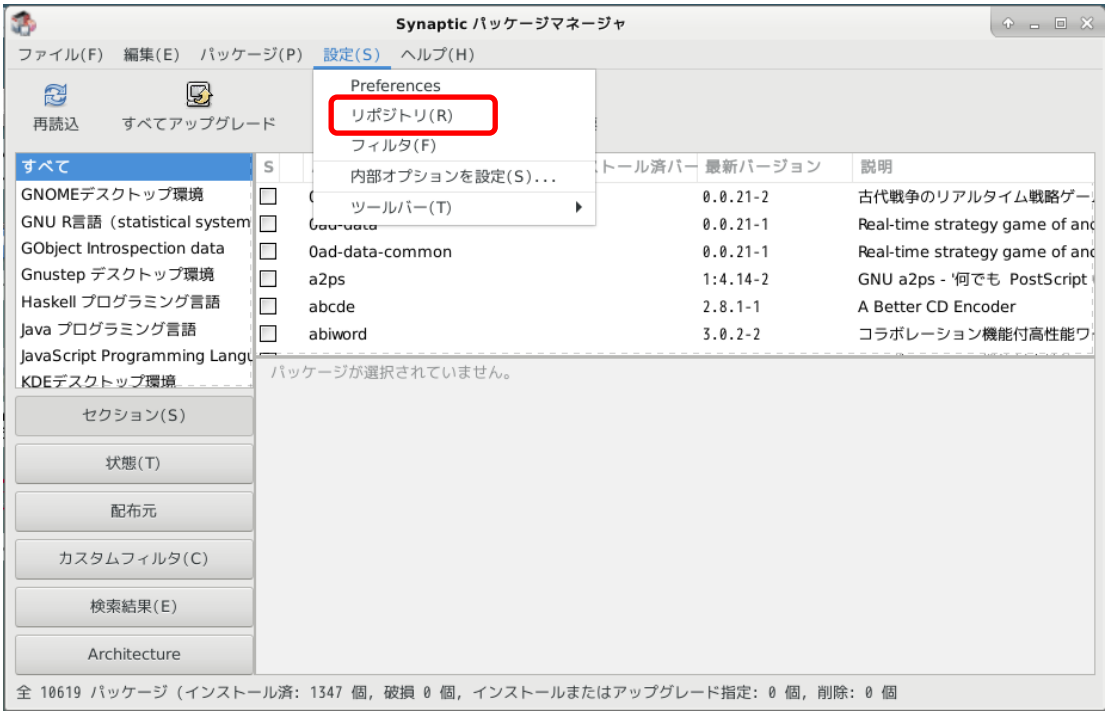


図 2-1-1-5. 設定メニュー

⑥　図 2-1-1-6 のようなリポジトリ設定画面が表示されます。

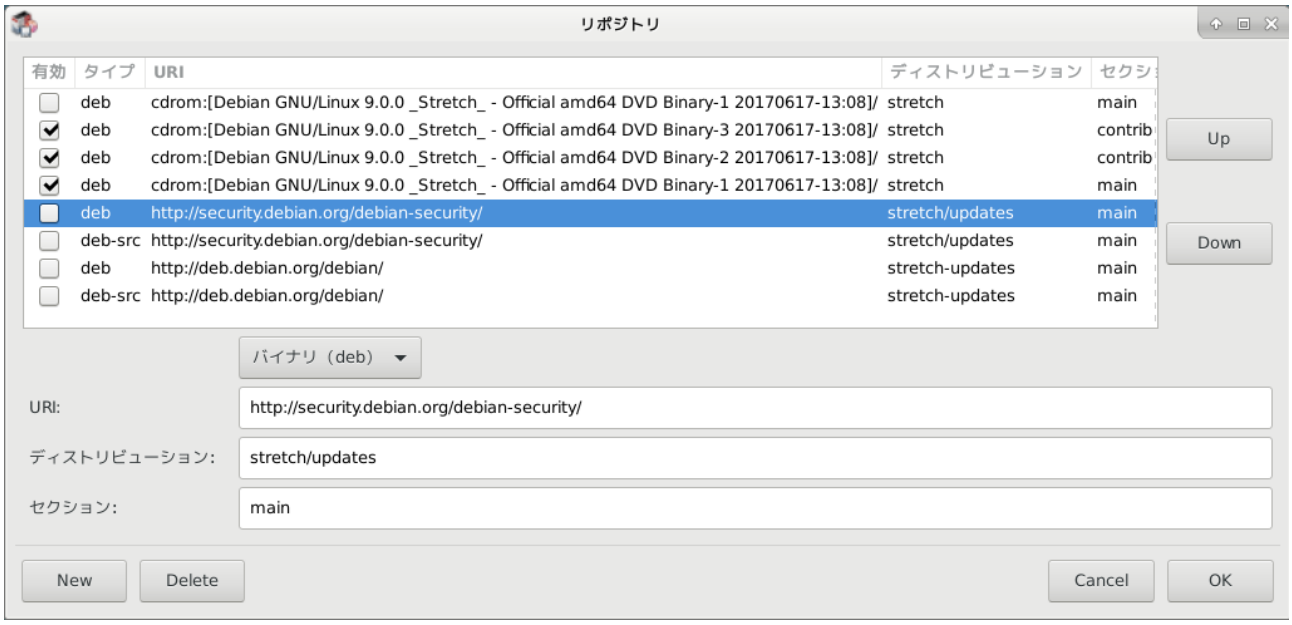


図 2-1-1-6.　リポジトリ画面

デフォルト設定では、DVD-ROM3 枚が有効となっています。
安定版のネットリポジトリは 4 種類登録済みです。
「New」ボタンをクリックして、追加することも可能です。

⑦　設定を更新して、「OK」をクリックすると、図 2-1-1-7 の画面が表示されます。

図 2-1-1-7.　リポジトリ変更確認画面

「再読込」をクリックして、変更を反映させてください。

⑧　再読込されると、図 2-1-1-8 のような、警告画面が表示されます。DVD-ROM リポジトリには、Release ファイルが存在しないため、下記のような警告が表示されます。下記のメッセージについては無視してください。「x」ボタンを押して、閉じてください。

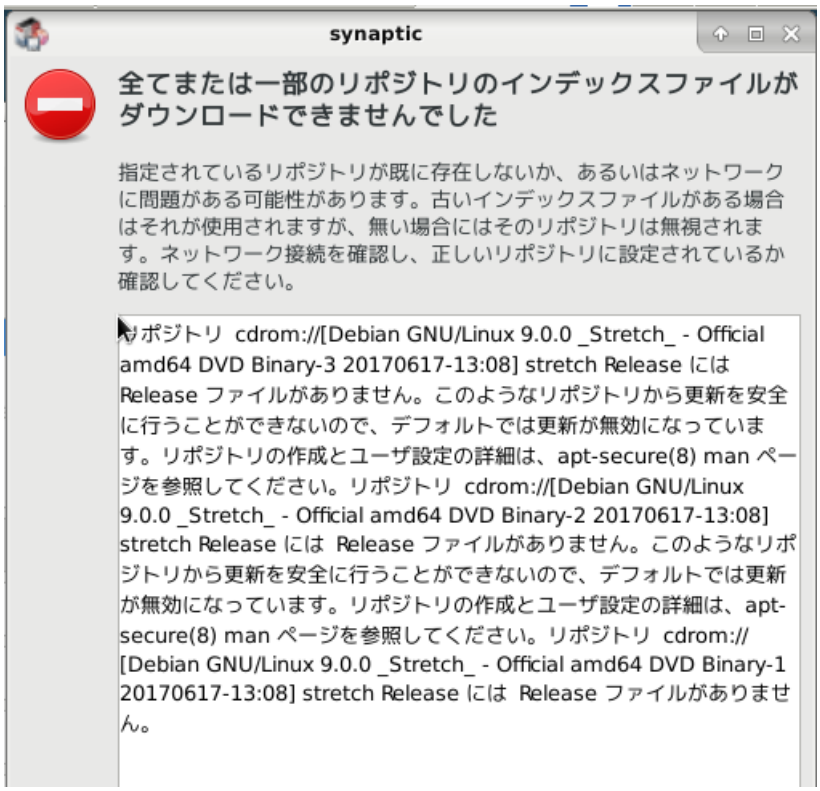


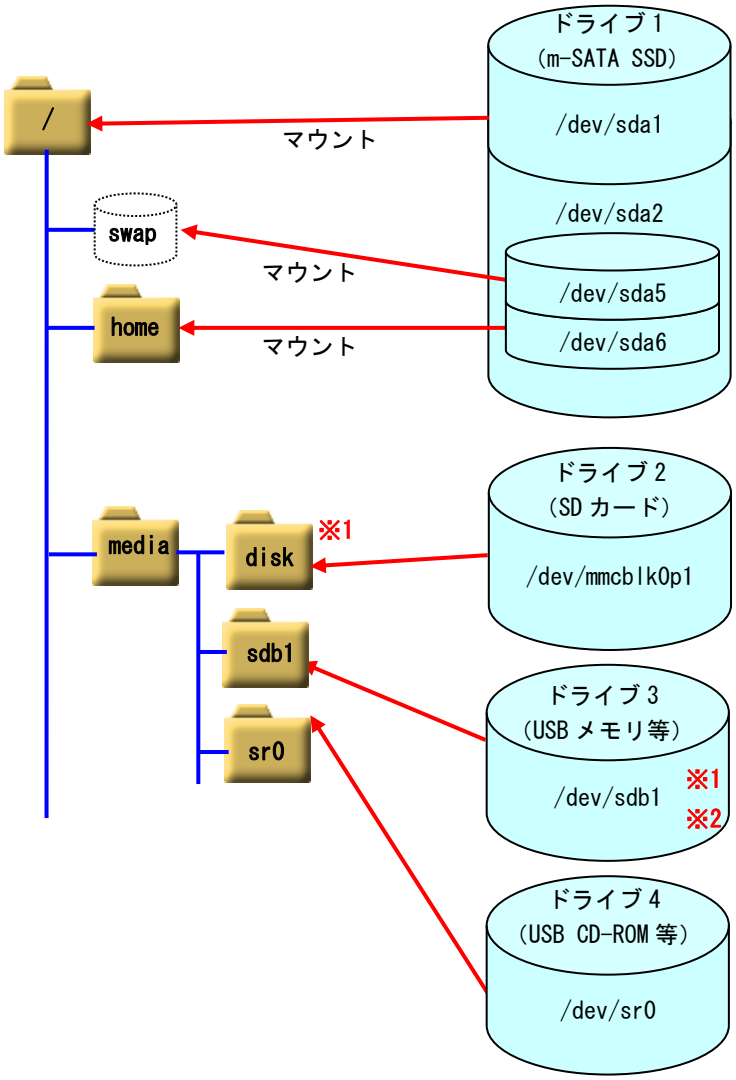
図 2-1-1-8.　警告画面

以上で、リポジトリの変更は完了します。

パッケージの追加については、「Synaptic パッケージマネージャ」を使用するか、「ターミナルエミュレータ」上で、「aptitude」または「apt-get」コマンドを使用してください。

２－２　Algonomix6 のディレクトリ構造

Linux ではルートファイルシステムと呼ばれるツリー構造のファイルシステムを採用しています。ルートディレクトリ（/）以下に、HDD 上に構築されたファイルシステムをマウントすることで、ルートファイルシステムとして利用できるようにしています。Algonomix6 は Debian ベースとなっている為、Debian のディレクトリ構造に従っています。



※注 1：ストレージ機器をマウントするたびに sdb1、sdb1、sdc1 というようにマウントポジションが追加されていきます。

※注 2：USB 機器は、認識順により sdb と sdc が入れ替わる可能性があります。

図 2-2-1. Debian のツリー構造

EC4A シリーズでは、m-SATA SSD 64GByte にルートファイルシステムが構築されています。表 2-2-1 のようにパーティションが切られ、それぞれのディレクトリにマウントされています。

EC4A シリーズのルートファイルシステム構成をリスト 2-2-1 に、ディレクトリ内容を表 2-2-2 に示します。

表 2-2-1. EC4A シリーズの初期パーティション構成(16GByte)

ドライブ名	サイズ	使用容量	空き容量	使用%	マウントポジション
/dev/sda1	19GByte	3.4GByte	15GByte	19%	/
/dev/sda6	36GByte	71MByte	34GByte	1%	/home

リスト 2-2-1. ルートファイルシステムのディレクトリ構成

```
/
+--bin
+--boot
+--dev      --+--shm
+--etc
+--home     --+--asdusr
+--lib
+--lost+found
+--md5sum.txt
+--media
+--mnt
+--opt
+--proc
+--root
+--run      --+--Lock
+--sbin
+--srv
+--sys      --+--fs     --+--cgroup
+--tmp
+--usr      --+--bin
              +--games
              +--include
              +--lib
              +--local
              +--sbin
              +--share
              +--src
+--var      --+--cache
              +--lib
              +--log
              +--tmp
```

表 2-2-2. ルートファイルシステムのディレクトリ内容)

ディレクトリ名	内容	tempfs
/	ルートディレクトリ	
/bin	システム管理者・一般ユーザ共に使用するコマンド群が格納されています。	
/boot	起動時に必要とする設定ファイル群とマップインストーラが配置されています。	
/dev	ハードウェアをコントロールする為のデバイスファイルが格納されています。基本的なデバイスの一覧を以下に示します。 ・ターミナル：/dev/tty*　　例：tty0、tty1 キーボードとプリンタにより構成され、文字を入出力できます。 ・シリアルポート：/dev/ttyS*　　例：ttyS0、ttyS1、ttyS2 シリアル通信することができます。 ・SCSI ディスク：/dev/sd*　　例：sdb1、sdb2、sdc USB メモリ等はこのデバイスになります。sd の後ろにつく文字がディスクを特定し、数字がパーティションを表しています。 **※注：/dev/sda*は m-SATA SSD のデバイスファイルとなりますので、新たに追加した USB メモリ等は sdb 以降になります。**	
/dev/shm	RAM ディスクです。	○
/etc	設定ファイルが格納されています。	
/home	システムに登録された通常ユーザの個人用ディレクトリが入っています。Algonomix6 では、「asdusr」というユーザが登録されています。ftp や ssh ではこのユーザに対してアクセスします。 ユーザ名　：asdusr パスワード：asdusr	
/lib	システム起動時や、/bin や /sbin のコマンドを実行する時に使用される共有ライブラリが格納されています。	
/media	CD-ROM やフロッピーディスクなどの外付けメディア用のディレクトリです。	
/mnt	一時的なファイルシステム用ディレクトリです。	
/opt	オプションのソフトパッケージコピー、インストールファイルが格納されているディレクトリです。	
/proc	バーチャルファイルシステム用の特別なディレクトリです。	
/root	システムの管理者権限を持ったユーザのホームディレクトリです。	
/run	実行プロセス関連データが格納されています。	○
/run/lock	保持ディレクトリです。	○
/sbin	管理用バイナリファイル用のディレクトリです。例えば、reboot、shutdown、lsmod などが格納されています。	
/srv	HTTP、FTP などのサービス用のデータが格納されています。	
/sys	デバイスの情報が格納されているディレクトリです。	
/sys/fs/cgroup	複数のグループをまとめて管理するための機能用のディレクトリです。	○
/tmp	一時ファイルを格納するディレクトリです。起動時に内容が消去されます。	○
/usr/bin	一般的なコマンドが格納されています。	
/usr/include	C 言語で使用する組込みファイルが格納されています。	
/usr/lib	一般的なライブラリファイルが格納されています。	
/usr/local	ユーザで作成されたプログラムを格納する領域です。	
/usr/sbin	システム関連のコマンドが格納されています。	
/usr/share	デフォルト設定ファイル、イメージ、ドキュメント等の共有ファイルが格納されています。	

ディレクトリ名	内容	tempfs
/usr/src	ソースコードが格納されます。	
/var	頻繁に変更されるデータが格納されています。	
/var/cache	アプリケーションのキャッシュデータが格納されています。	
/var/lib	アプリケーションの状態や APT の dpkg が管理されているパッケージ情報が格納されています。	
/var/log	システムやアプリケーションのログが格納されています。	○
/var/tmp	一時ファイルやディレクトリを必要とするプログラムで利用します。 /tmp よりもデータは長く保持されます。	○

２－３　Algonomix6 設定ツール「ASD Config」について

本項では、EC4A シリーズ用の各種設定ツール「ASD Config」について説明します。

ASD Config Menu を起動する方法は以下の 2 通りの方法があります。

１．出荷時状態で起動したとき、自動的に起動されます。

２．コンソールを起動させ、下記のコマンドを実行します。

```
$ sudo AsdConfigMenu
```

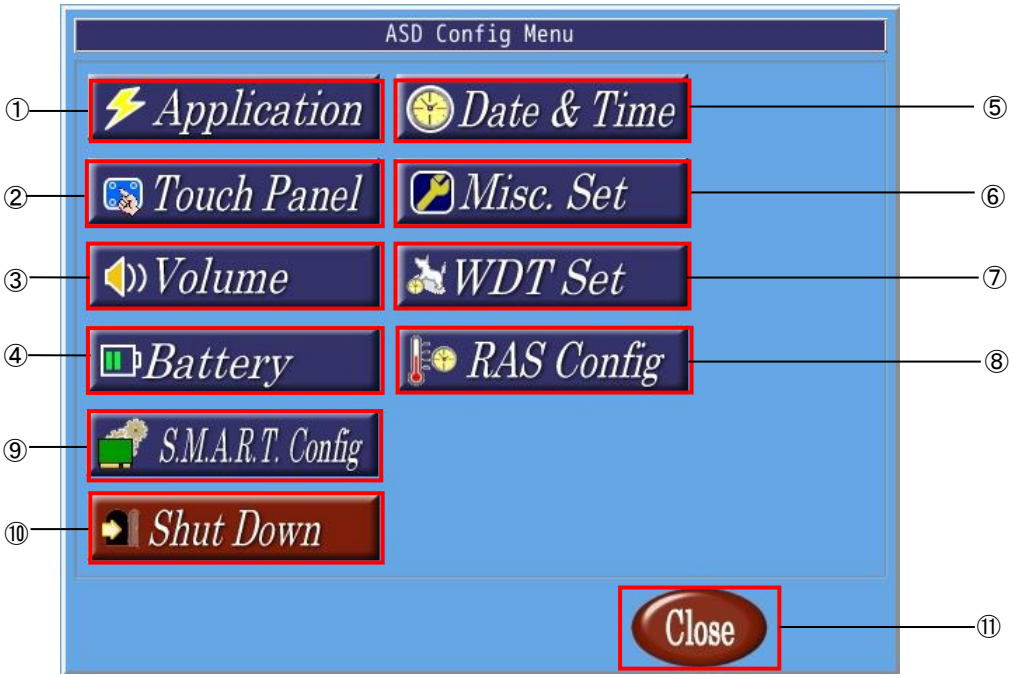


図 2-3-1. ASD Config Menu

ASD Config Menu から起動できる各ツールについて説明します。

① ASD Application Config
ASD Application Config は、ユーザアプリケーションのアップデートを行うためのツールです。
詳細は『2-3-1　ASD Application Config について』で説明します。

② ASD Touch Panel Config
ASD Touch Config は、タッチパネルのキャリブレーションを行うためのツールです。
詳細は『2-3-2　ASD Touch Panel Config について』で説明します。

③ ASD Volume Config
ASD Volume Config は、音声の設定を行うためのツールです。
詳細は『2-3-3　ASD Volume Config について』で説明します。

④　ASD UPS Config
ASD UPS Config は、UPS 機能の設定や RAM バックアップ機能の設定を行うためのツールです。
詳細は、『2-3-4　ASD UPS Config について』で説明します。

⑤　ASD Date Config
ASD Date Config は、時計を設定するためのツールです。
詳細は、『2-3-5　ASD Date Config について』で説明します。

⑥　ASD Misc Setting
ASD Misc Setting は、画面の輝度調節や、本製品の製品情報を確認するためのツールです。
詳細は『2-3-6　ASD Misc Setting について』で説明します。

⑦　ASD WatchdogTimer Config
ASD WatchdogTimer Config は、ハードウェア・ウォッチドッグタイマの設定を行うためのツールです。
詳細は『2-3-7　ASD WatchdogTimer Config について』で説明します。

⑧　ASD Ras Config
ASD Ras Config は、CPU 温度の確認や WakeOnRTC 機能の設定を行うためのツールです。
詳細は『2-3-8　ASD Ras Config について』で説明します。

⑨　ASD Smart Config
ASD Smart Config は、メインディスク、サブディスクである、m-SATA の S.M.A.R.T.情報を監視して、寿命に到達すれば警報を上げる機能の設定を行うためのツールです。
詳細は『2-3-9 ASD Smart Config について』を参照してください。

⑩　電源オプション
シャットダウン、再起動を行います。
詳細は『2-3-10　ASD Shutdown Menu について』を参照してください。

⑪　ASD Config Menu の終了
ASD Config Menu を終了します。

２－３－１　ASD Application Config について

ASD Application Config は、USB メモリを用いたアップデートを行うためのツールです。
ASD Application Config を起動するには、メニュー画面から[Application]を選択します。

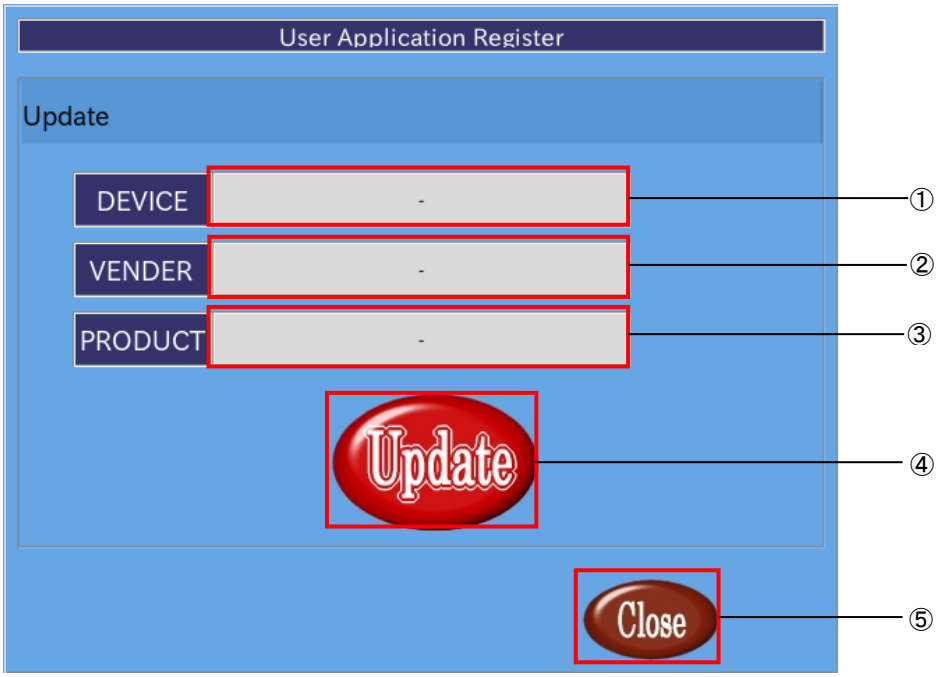


図 2-3-1-1.　アップデート画面

① デバイス名の表示
USB メモリが認識されている場合、[usb-storage]と表示されます。
USB メモリが認識されていなければ、「-」が表示されます。

② ベンダ名の表示
USB メモリが認識されている場合、USB メモリの製造元が表示されます。
USB メモリが認識されていなければ、[-]が表示されます。

③ プロダクト名の表示
USB メモリが認識されている場合、USB メモリの種類が表示されます。
USB メモリが認識されていなければ、[-]が表示されます。

④ アップデート
[Update]ボタンを押下することで、USB メモリ内にある「download.sh」が実行されます。
「download.sh」はシェルスクリプトである必要があります。

⑤ 終了
「Close」ボタンを押下することで、ASD Application Config を終了します。

２－３－２　ASD Touch Panel Config について

ASD Touch Panel Config はタッチパネルのキャリブレーションを行うためのツールです。
ASD Touch Panel Config を起動するには、メニュー画面から[Touch Panel]を選択します。

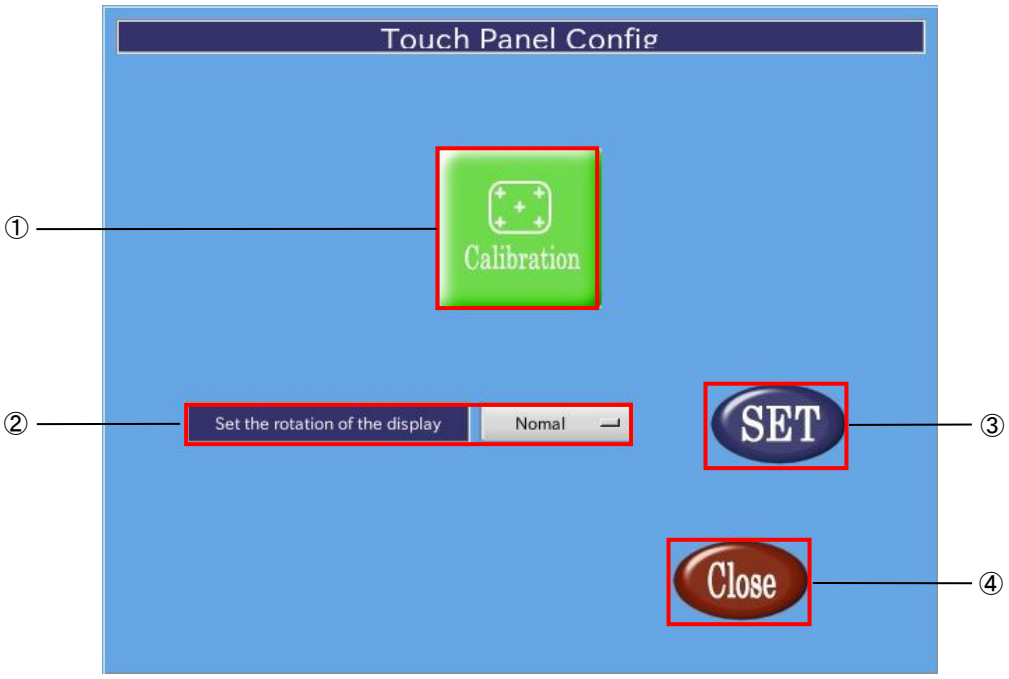


図 2-3-2-1. メニュー画面

① タッチパネルキャリブレーション
　タッチパネルキャリブレーションを開始します。
② ディスプレイとタッチパネルの回転
　ディスプレイとタッチパネルの回転方向を設定します。
③ 設定反映
　②で行った設定を反映し、保存します。
④ 終了
　タッチパネルキャリブレーションを終了します。

●ディスプレイとタッチパネルの回転

②のコンボボックスをクリックすると、図 2-3-2-2 のように、回転方向の設定メニューが表示されます。
画面の向きと設定メニューはそれぞれ表 2-3-2-1 のように対応しています。
任意の向きを選択して③の[SET]ボタンを押すことで、画面が回転します。
回転後、任意の位置をタッチできなくなった場合、①のキャリブレーションを行ってください。

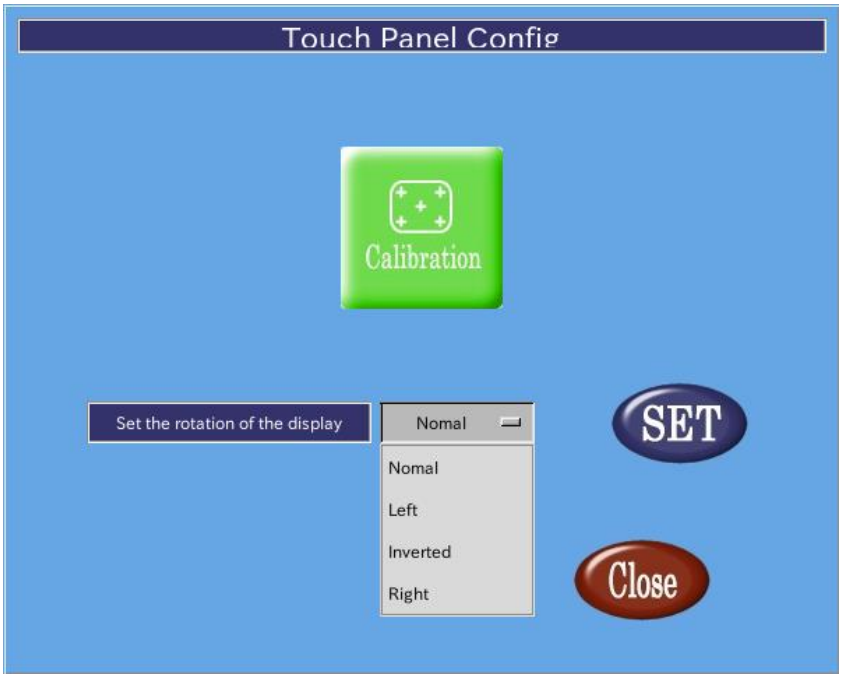


図 2-3-2-2. ディスプレイとタッチパネルの回転

表 2-3-2-1. ディスプレイの向きの設定

名称	内容
Normal	デフォルトの画面の向きです。
Left	反時計回りに 90° 回転します。
Inverted	180° 回転します。
Right	時計回りに 90° 回転します。

２－３－３　ASD Volume Config について

ASD Volume Config は音声ボリュームを設定する為のツールです。
ASD Volume Config は、メニュー画面から「Volume」を選択することで起動します。

●音声ボリュームの設定

[ASD Volume Config]の[Volume]タブを選択することで、各種音声デバイスのボリュームの調整、ミュートの設定を行うことができます。

ASD Volume Config で設定できる音声デバイスを表 2-3-3-1 に示します。

表 2-3-3-1. ASD Volume Config で設定できる音声デバイス

名称	内容
Master	マスターボリュームを設定します
HeadPhone	ヘッドホン出力を設定します ミュートの有無のみ設定可能です
PCM	PCM のボリュームを調整します
Front	フロントスピーカボリュームを調整します
Front Line	フロントラインボリュームを調整します
Front Mic	フロントマイクボリュームを調整します
Front Mic Boost	フロントマイクボリュームを調整します 値を１上げるごとに 10【dB】上昇します
Line	ラインボリュームを調整します
Mic	マイクボリュームを調整します
Mic Boost	マイクボリュームを調整します 値を１上げるごとに 10【dB】上昇します
Capture	録音時のボリュームを調整します
Beep	ビープボリュームを調整します
Speaker	スピーカ出力を設定します

※注：EC4A シリーズでは、Master、HeadPhone、PCM のみ設定が有効です。

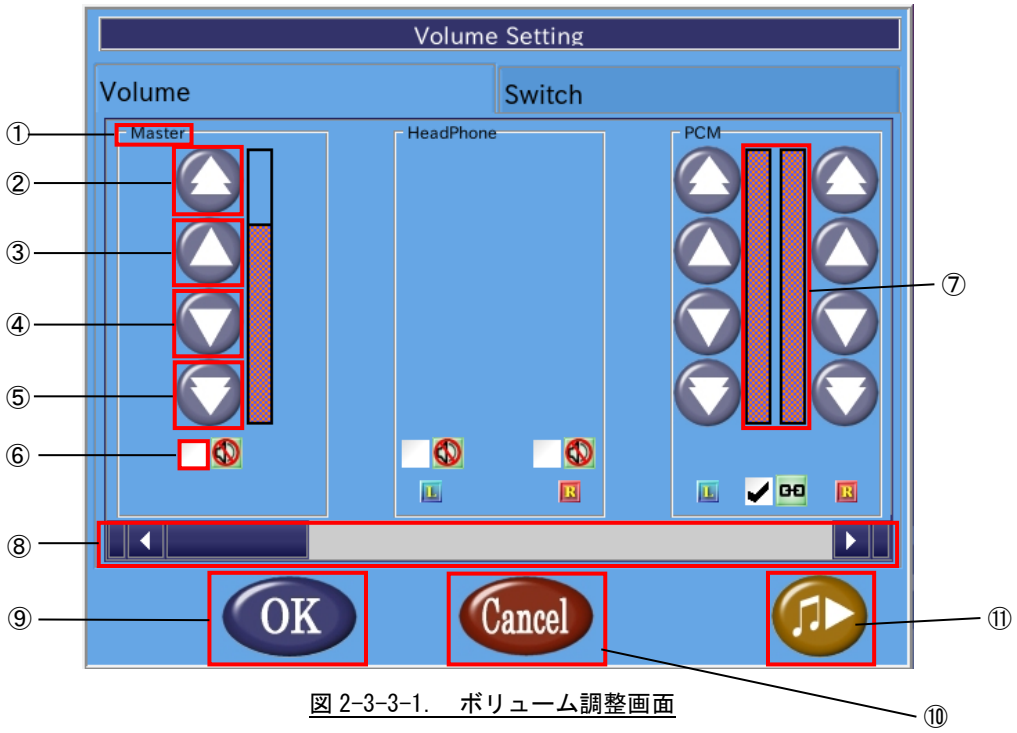


図 2-3-3-1.　ボリューム調整画面

① 音声デバイス名
音声デバイス名を表示します。

② 音量調整（大）
音量を大きく上げます。

③ 音量調整（小）
音量を上げます。

④ 音量調整（小）
音量を下げます。

⑤ 音量調整（大）
音量を大きく下げます。

⑥ ミュート調整
チェックボックスにチェックを入れることでミュート状態になります。チェックボックスを外せばミュートが解除されデバイスが有効になります。

⑦ 音量
現在の音量を表示します。

⑧ スクロールバー
スクロールバーを移動させることで他の音声デバイスを表示します。

⑨ 設定を保存して終了
「OK」ボタンを押下することで、設定を保存して終了します。

⑩ 設定を保存せずに終了
「Cancel」ボタンを押下することで、設定を破棄して終了します。

⑪ サンプル音声
サンプル音声を出力します。

●ボリュームスイッチの変更

[ASD Volume Config]の[Switch]タブを選択することで、IEC958、IEC958 Default の有効・無効の切替え、録音時の音源の切替えを設定できます。

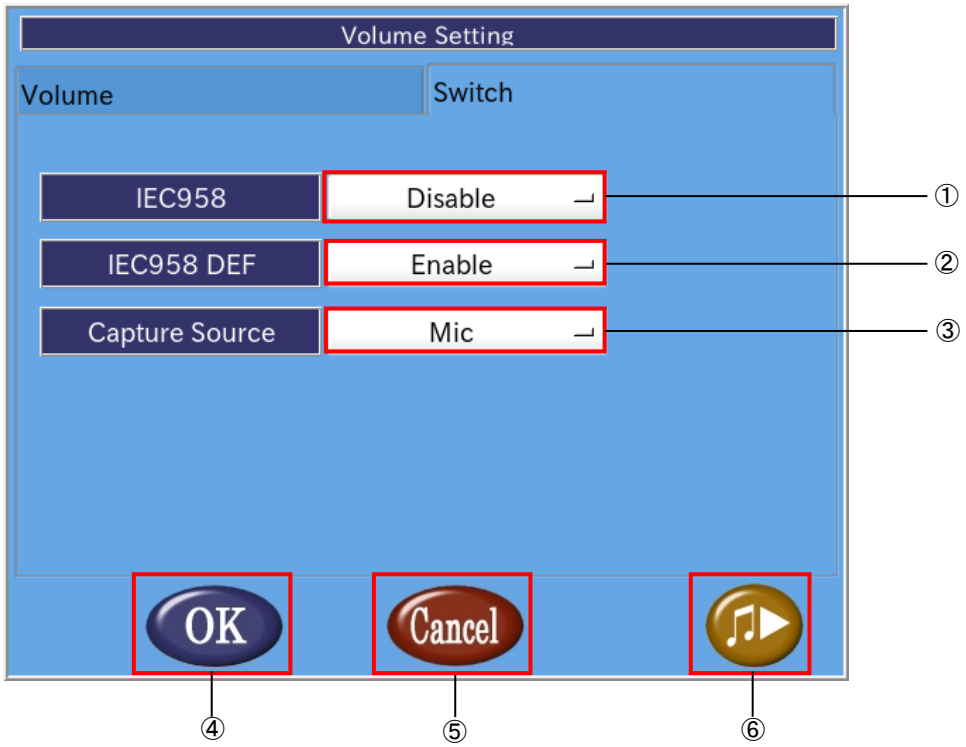


図 2-3-3-2.　ボリュームスイッチ画面

① IEC958
IEC958 プラグインの有効・無効を切替えます。
[Enable]で有効、[Disable]で無効となります。

② IEC958 DEF
デフォルトの IEC958 PCM プラグインの有効・無効を切替えます。
[Enable]で有効、[Disable]で無効となります。
EC4A シリーズでは、デフォルトの PCM プラグインは PCM となります。

③ Capture Source
録音時の音源を切替えます。
マイク、フロントマイク、ライン、フロントラインから選択できます。

※注：EC4A シリーズでは、マイク入力のみ対応しています。

④ 設定を保存して終了
[OK]ボタンを押下することで、設定を保存して終了します。

⑤ 設定を保存せずに終了
[Cancel]ボタンを押下することで、設定を破棄して終了します。

⑥ サンプル音声
サンプル音声を出力します。

２－３－４　ASD UPS Config について

ASD UPS Config は、UPS の状態取得、設定、シャットダウン時のバックアップ機能の設定を行うためのツールです。

ASD UPS Config を起動するには、メニュー画面から[Battery]を選択します。

●バッテリ状態の表示

ASD UPS Config の[Status]タブを選択することで、UPS のバッテリ状態を取得できます。

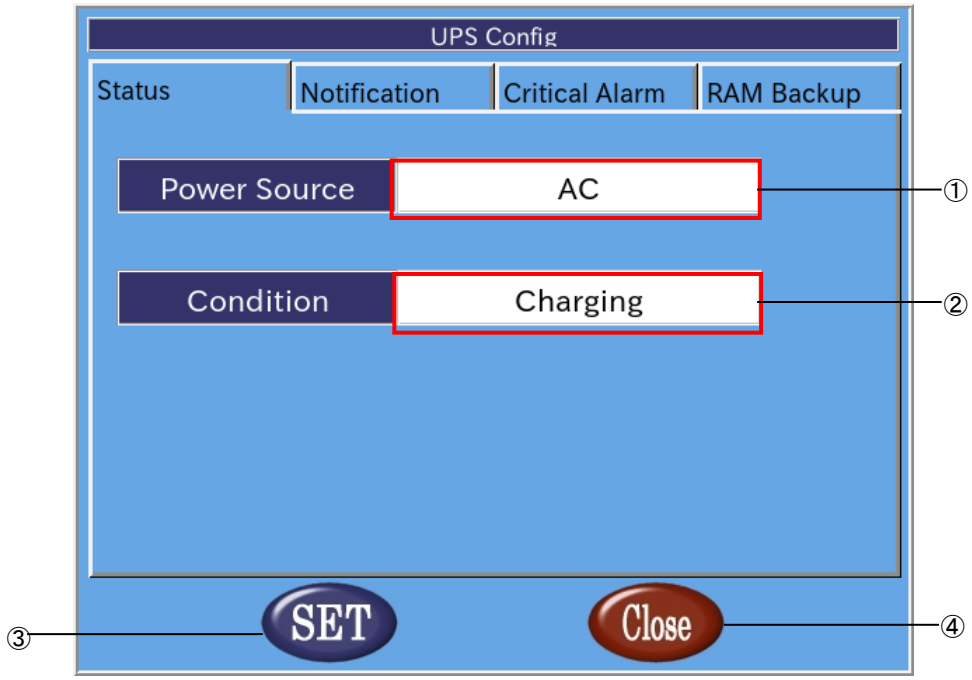


図 2-3-4-1. バッテリ状態の表示

①　電源種別

端末の電源種別を表示します。

表 2-3-4-1. 電源種別

名称	動作
AC	AC 電源で動作中です。
Battery	バッテリ電源で動作中です。

②　状態

バッテリの状態を表示します。

表 2-3-4-2. バッテリの状態

名称	動作
Charging	バッテリは充電中です。
Full Charge	バッテリは満充電状態です。
Discharging	バッテリは放電中です。
Error	バッテリに異常が生じています。

③　設定の反映

「SET」ボタンを押下することで設定を反映します。

④　終了

[Close]ボタンを押下することで ASD UPS Config を終了します。

[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

●バッテリ異常通知設定

ASD UPS Config の[Notification]タブを選択することで、バッテリ異常が発生したときの通知について設定できます。

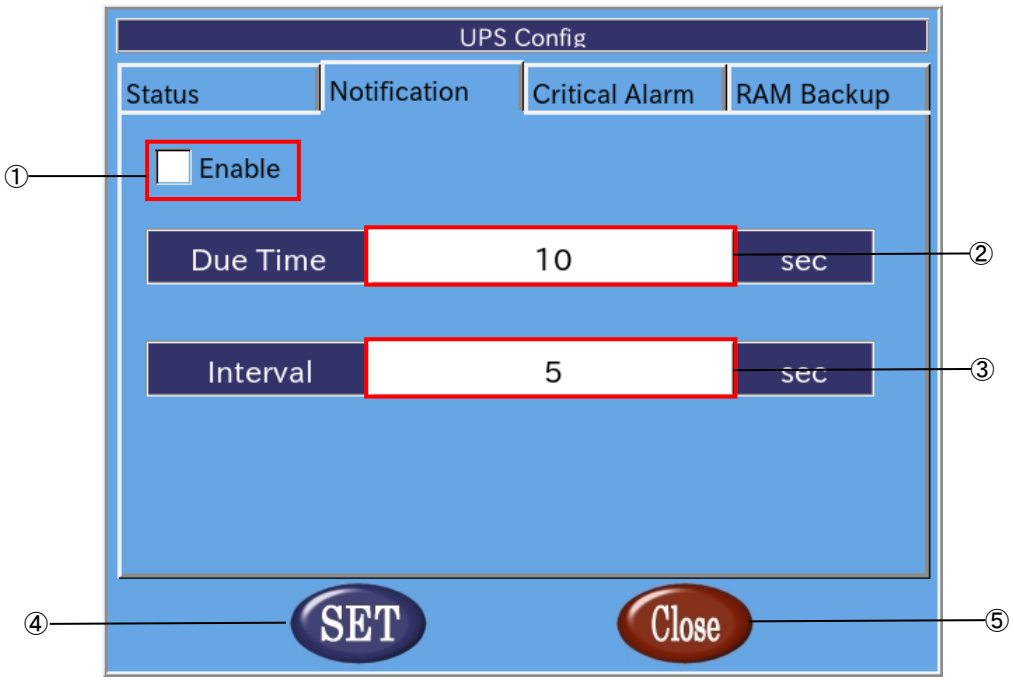


図 2-3-4-2. バッテリ異常通知設定

① バッテリ異常通知有効/無効設定
チェックボックスにチェックを入れることで、バッテリ異常発生時の通知が有効になります。

② 通知開始時間
バッテリ異常が発生してから、通知を開始するまでの時間を設定します（0～120[秒]）。

③ 通知間隔
②の通知後の通知間隔を設定します（5～300[秒]）。

④ 設定の反映
[SET]ボタンを押下することで設定を反映します。

⑤ 終了
[Close]ボタンを押下することで ASD UPS Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

●バッテリ警告設定

ASD UPS Config の[Critical Alarm]タブを選択することで、バッテリ駆動開始後の警告、シャットダウンについて設定できます。

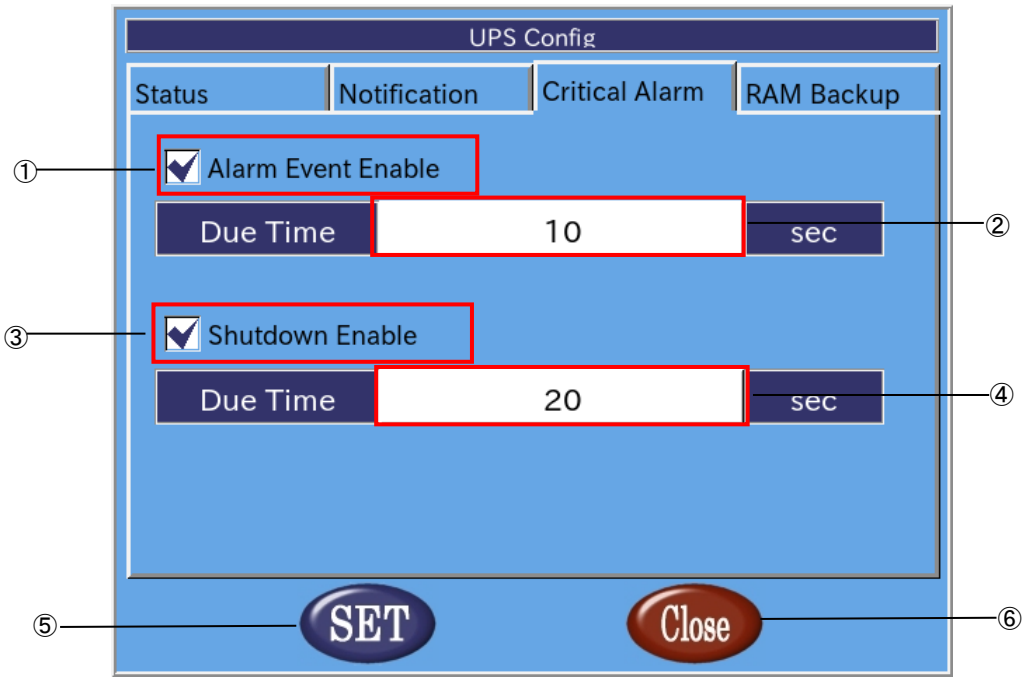


図 2-3-4-3. バッテリ警告設定

① バッテリ警告有効/無効設定
チェックボックスにチェックを入れることで、バッテリ駆動開始後の警告が有効になります。

② 警告開始時間
バッテリ駆動開始から警告を送信するまでの時間の設定します（0～60[秒]）。

③ シャットダウン有効/無効設定
チェックボックスにチェックを入れることで、バッテリ駆動開始後のシャットダウンが有効になります。

④ シャットダウン開始時間
バッテリ駆動開始からシャットダウンするまでの時間を設定します（0～60[秒]）。

⑤ 設定の反映
[SET]ボタンを押下することで設定を反映します。

⑥ 終了
[Close]ボタンを押下することで ASD UPS Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

●RAM バックアップ設定

ASD UPS Config の[RAM Backup]タブを選択することで、仮想 RAM デバイスのシャットダウン時のバックアップ機能について設定できます。

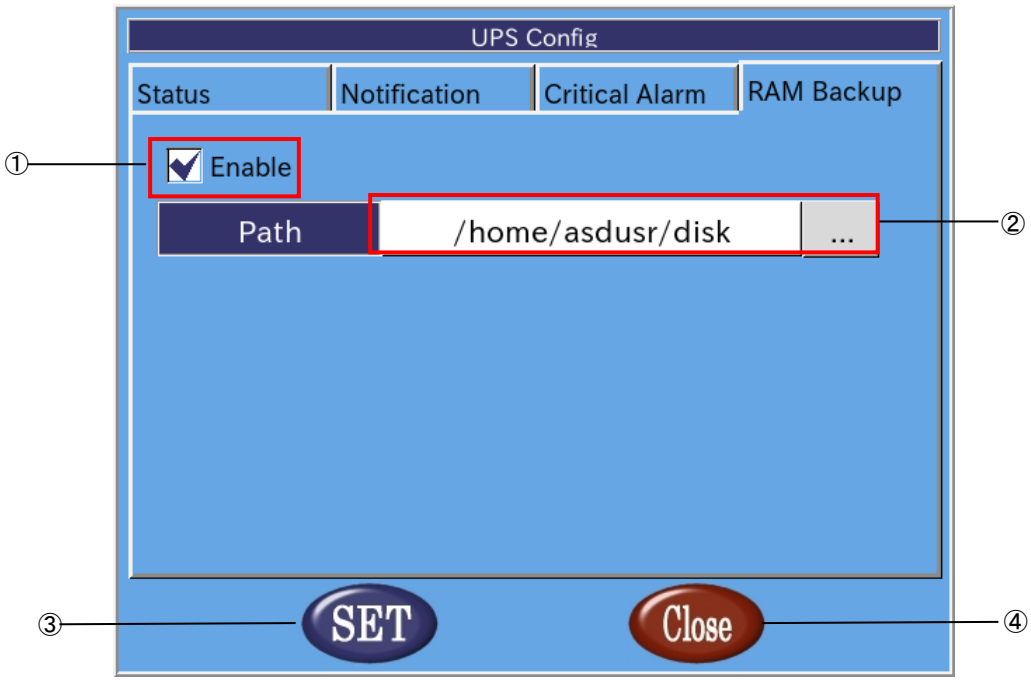


図 2-3-4-4. RAM バックアップ設定

① バックアップ有効/無効設定
チェックボックスにチェックを入れることで、バックアップ機能が有効になります。

② バックアップファイルパス
バックアップファイルの保存先を指定します。
バックアップファイルを外部ストレージに保存する場合、起動時に自動マウントする必要があります。
起動時に自動マウントする方法については、『4-8-3 外部ストレージデバイスの起動時マウントについて』を参照してください。

③ 設定の反映
[SET]ボタンを押下することで設定を反映します。

④ 終了
[Close]ボタンを押下することで ASD UPS Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

２－３－５　ASD Date Config について

ASD Date Config は時計を設定するためのツールです。NTP サーバを設定し、自動的に時計を調整することもできます。

ASD Date Config を起動するには、メニュー画面から[Date & Time]を選択します。

●時計の手動設定

ASD Date Config の[Date & Time]タブを選択することで、手動で時計を設定できます。

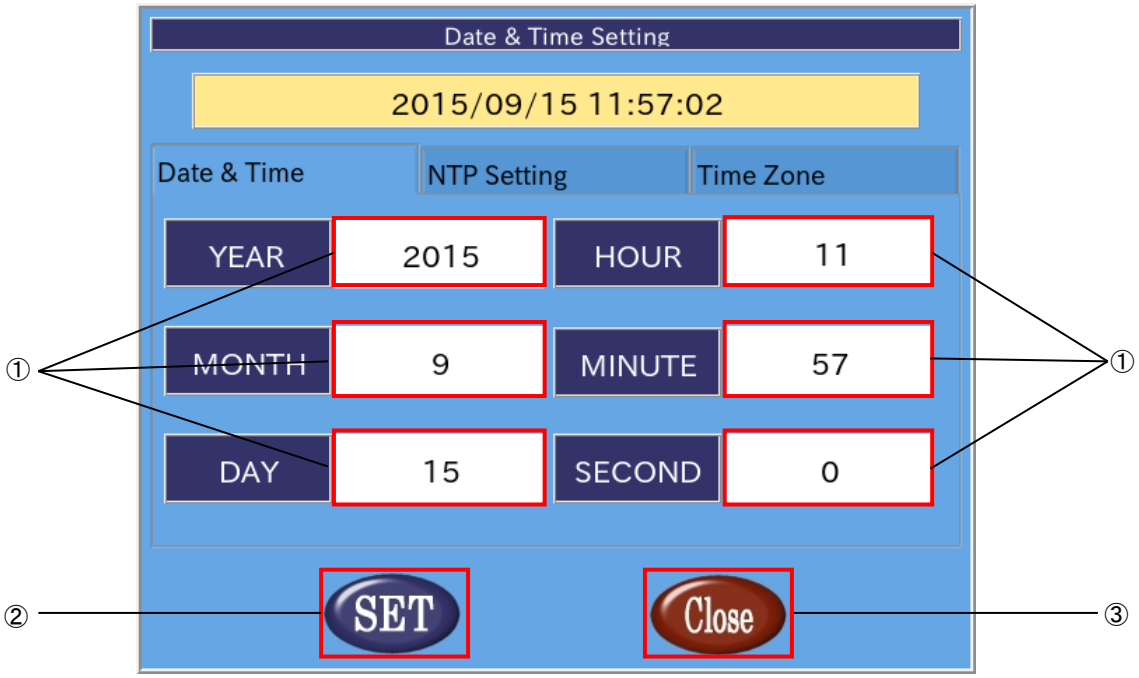


図 2-3-5-1. 時計の設定

① 日付、時刻を設定
白い部分を押下することで、入力画面があらわれ、日付、時間を設定できます。
年、月、日、時、分、秒単位で指定できます。
ntp を有効にした場合、これらの項目は変更できません。

② 設定の反映
「SET」ボタンを押下することで設定を反映します。

③ 終了
[Close]ボタンを押下することで ASD Date Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

●NTP サーバの設定

NTP は、時計を自動的に調整するためのプロトコルです。ネットワークに接続した状態で、NTP を用いると EC4A シリーズの時計が NTP サーバの時計に同期されます。

ASD Date Config の[Network Time Protocol]タブを選択することで、NTP サーバを設定できます。

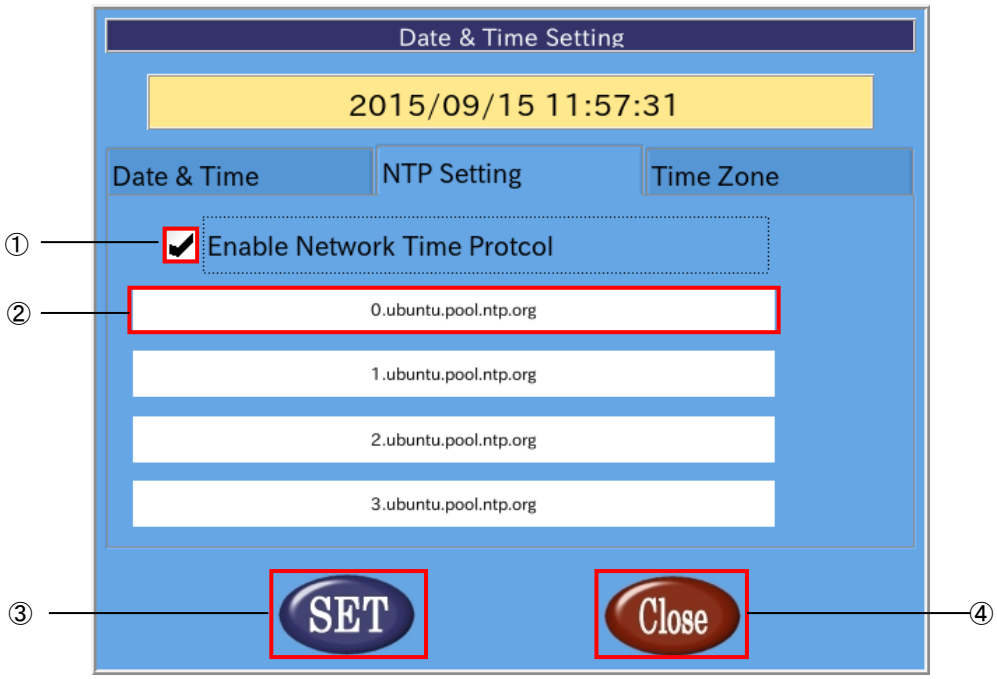


図 2-3-5-2. NTP サーバの設定

① NTP の有効、無効の切替え
チェックボックスにチェックを入れることで NTP を有効にします。
チェックを外すと NTP は無効になります。

② ntp サーバの設定
現在設定されている NTP サーバを表示します。
NTP を有効にした場合、この項目を押下することで入力画面が表示され、NTP サーバを設定できます。
NTP サーバは最大 4 個まで設定できます。

③ 設定の反映
[SET]ボタンを押下することで設定を反映します。

④ 終了
[Close]ボタンを押下することで ASD Date Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、設定は破棄されます。

※注：EC4A シリーズの時計と NTP サーバの時計が大きくずれている場合、NTP デーモンが終了することがあります。その際は、一旦 NTP を無効にし、手動で時刻を設定後、再起動してください。

●タイムゾーンの設定

ASD Date Config の[Time Zone]タブを選択することで、タイムゾーンを設定できます。

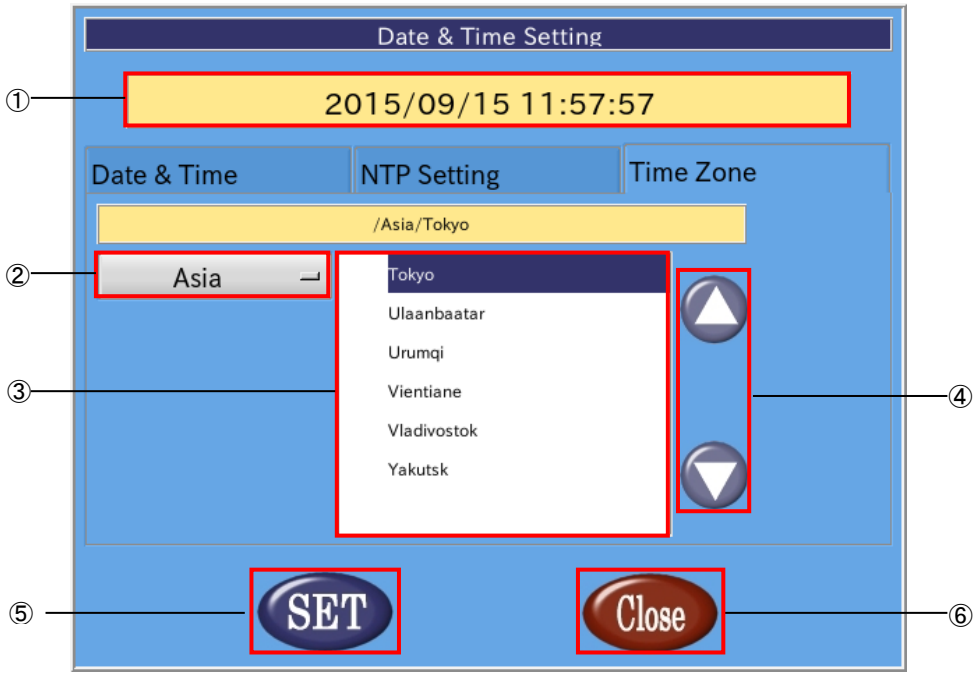


図 2-3-5-3. タイムゾーンの設定

① タイムゾーンの表示
現在設定されているタイムゾーンを表示します。

② 地域の選択
設定するタイムゾーンの地域を選択します。タイムゾーンの都市を東京に設定する場合は「Asia」を選択します。

③ 都市の選択
設定するタイムゾーンの都市を選択します。タイムゾーンの都市を東京に設定する場合は「Tokyo」を選択します。

④ タイムゾーンの都市リストの変更
都市のリストをスクロールします。

⑤ 設定の反映
[SET]ボタンを押下することで設定を反映します。

⑥ 終了
[Close]ボタンを押下することで ASD Date Config を終了します。
[SET]ボタンを押下せずに、[Close]ボタンを押下した場合、変更は破棄されます。

２－３－６　ASD Misc Setting について

ASD Misc Setting は、バックライト、ブザーの設定や製品情報の読み出しを行うツールです。
ASD Misc Setting を起動するには、メニュー画面から「Misc. Set」を選択します。

●バックライトの調節と、タッチパネルのブザーの設定

ASD Misc Setting の[Backlight & Buzzer Setting]タブを選択することで、図 2-3-6-1 の画面になります。
ここでは、バックライトの調節とブザーの設定を行うことができます。

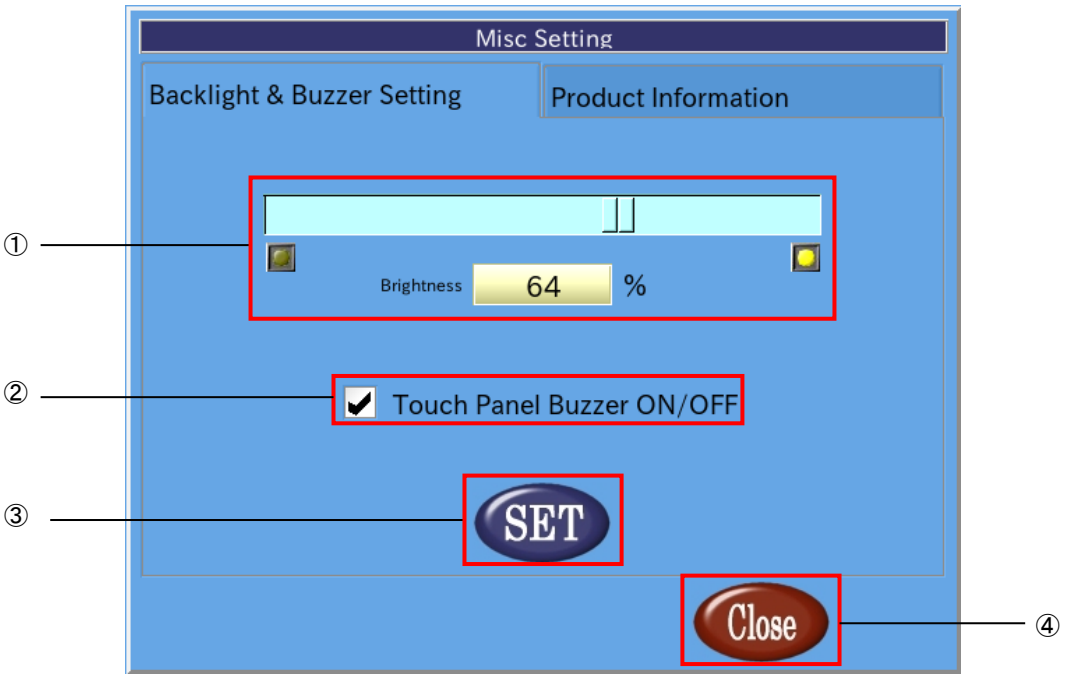


図 2-3-6-1. バックライトとブザーの設定

① バックライトの明るさの調整

スクロールバー上の  をドラッグすることで、バックライトの光量を調節できます。

② タッチパネルブザーの ON/OFF 設定

チェックボックスをチェックすることでタッチパネルをタッチする時にブザーを鳴らすことができます。チェックが外れていればタッチパネルをタッチしてもブザーは鳴りません。

③ 設定の保存

「SET」ボタンを押下することで、現在の設定を保存します。

④ 終了

「Close」ボタンを押下することで ASD Misc Setting を終了します。
「SET」ボタンを押下せずに「Close」ボタンを押下した場合、設定は破棄されます。

●製品情報読み出し

ASD Misc Setting の[Product Information]タブを選択することで、基板情報や FPGA バージョン、Linux カーネルビルドバージョンの情報を読み出すことができます。

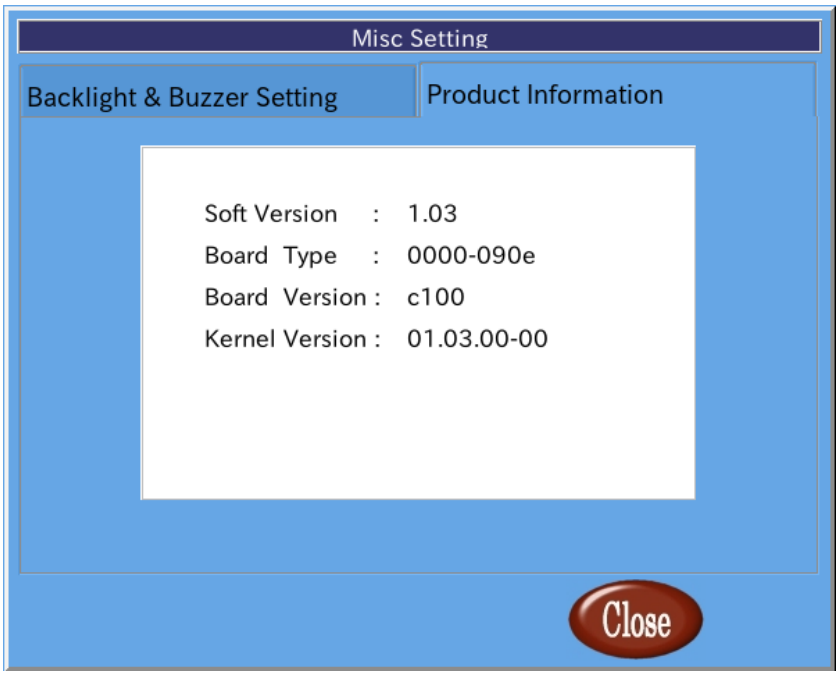


図 2-3-6-2. 製品情報

表 2-3-6-1. 製品情報

項目名	内容
Soft Version	Algonomix の OS バージョン
Board Type	基板型番
Board Version	FPGA のバージョン
Kernel Version	Linux カーネルのビルドバージョン

２－３－７　ASD WatchdogTimer Config について

ASD WatchdogTimer Config は、ハードウェア・ウォッチドッグタイマの設定の取得、
変更を行うためのツールです。
ASD WatchdogTimer Config を起動するには、メニュー画面から[WDT Set]を選択します。

●ハードウェア・ウォッチドッグタイマの設定

ASD WatchdogTimer Config の[Hardware Watchdog Timer]タブを選択することで、ハードウェア・ウォッチドッグタイマを設定できます。

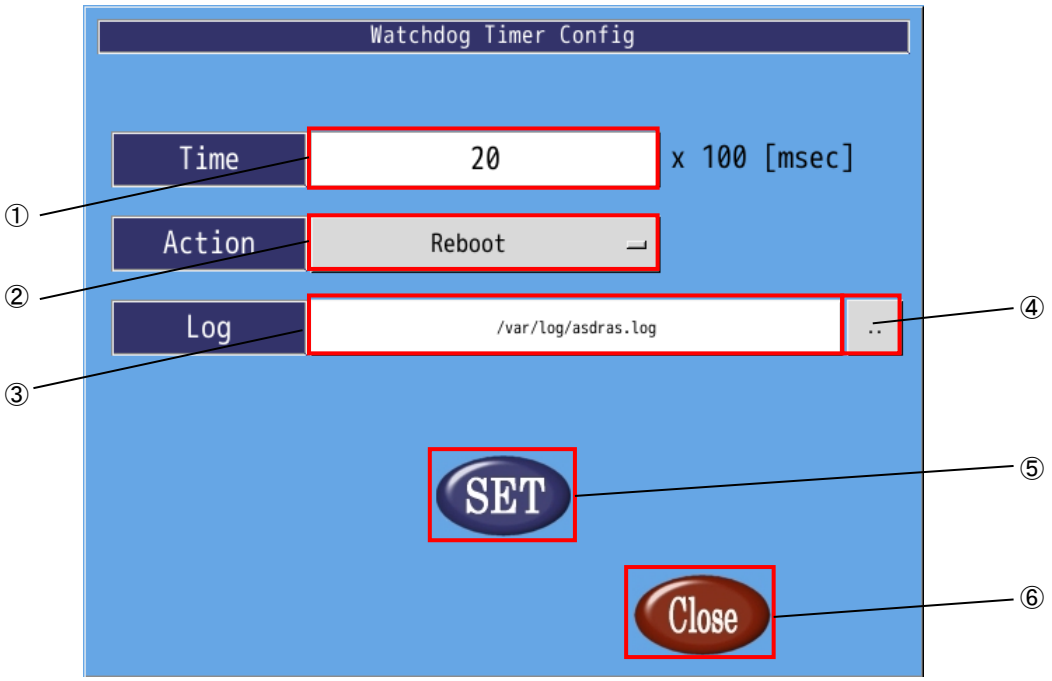


図 2-3-7-1. ハードウェア・ウォッチドッグタイマの設定

①　タイマ時間の設定

ハードウェア・ウォッチドッグタイマがタイムアウトするまでのタイマ時間を設定します。
有効設定値は 1～65535、タイマ時間は設定値×100【msec】になります。
デフォルト値は 20 です。

②　動作の設定

ハードウェア・ウォッチドッグタイマがタイムアウトした際の動作を設定します。
選択できる動作を表 2-3-7-1 に示します。デフォルトでは Reboot が選択されます。

表 2-3-7-1. ハードウェア・ウォッチドッグタイマのタイムアウト時の動作

名称	動作
Shutdown	shutdown コマンドを実行します。
Reboot	reboot コマンドを実行します。
Popup	ポップアップメッセージを表示します。
Event	ユーザアプリケーションにイベント発生を通知します。
PowerOff	強制的に電源を切ります。
Reset	強制的に再起動します。

※注：PowerOff および Reset は、ハードウェア的に電源をオフするため、システムがハングアップしても動作しますが、データが破損する可能性があります。

③　ログファイルパスの設定
ハードウェア・ウォッチドッグタイマが起動時、タイムアウト時に出力するログファイルのパスを指定します。デフォルトでは、/var/log/asdras.log にログを出力します。

④　ファイル選択
ファイル選択ダイアログを表示します。

⑤　設定の反映
[SET]ボタンを押下することで、設定を反映します。

⑥　終了
[Close]ボタンを押下することで、ASD WatchdogTimer Config を終了します。
[SET]ボタンを押下せずに[Close]ボタンを押下した場合、設定は破棄されます。

２－３－８　ASD Ras Config について

ASD Ras Config は、CPU 温度の確認や WakeOnRTC の設定の取得、変更を行うためのツールです。
ASD Ras Config を起動するには、メニュー画面から[Ras Config]を選択します。

●CPU および周辺回路の温度の表示

ASD Ras Config の[Temperature]タブを選択することで、CPU 温度の表示が確認できます。

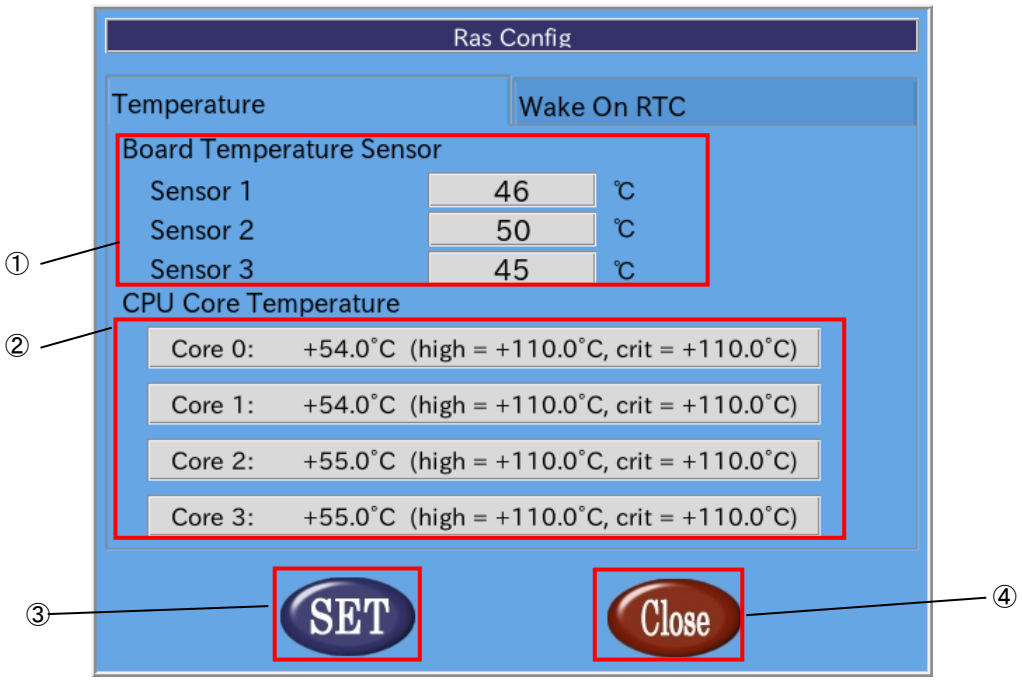


図 2-3-8-1. CPU 温度の表示

①　基板温度

基板内蔵の温度センサの測定値を表示します。

表 2-3-8-1. Board Temperature Sensor

名称	動作
Sensor 1	DRAM 付近の温度センサの測定値を表示します。
Sensor 2	PMIC 付近の温度センサの測定値を表示します。
Sensor 3	UPS バッテリ付近の温度センサの測定値を表示します。

②　CPU 温度

各 CPU の Core の温度を表示します。

③　設定の反映

[SET]ボタンを押下することで、設定を反映します。

④　終了

[Close]ボタンを押下することで、ASD Ras Config を終了します。
[SET]ボタンを押下せずに[Close]ボタンを押下した場合、設定は破棄されます。

●Wake On RTC の設定

ASD Ras Config の[Wake On RTC]タブを選択することで、Wake On RTC 機能の設定ができます。

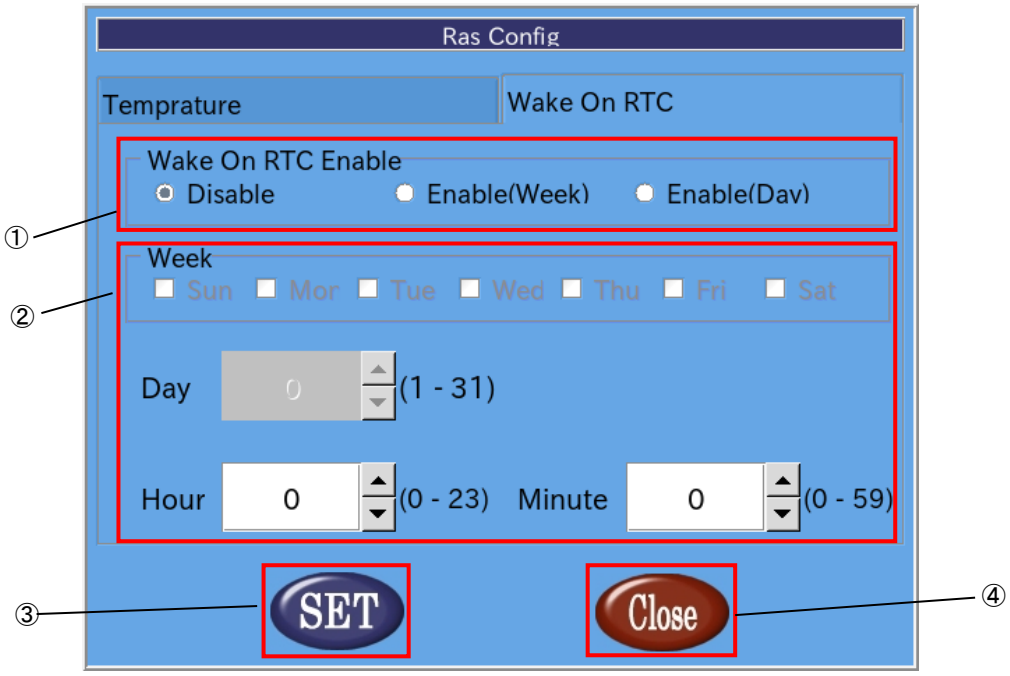


図 2-3-8-2. Wake On RTC の設定

①　Wake On RTC Enable

Wake On Rtc 機能の設定を行います。

表 2-3-8-2. Wake On RTC Enable

名称	動作
Disable	Wake On Rtc Timer 機能を無効にします。
Enable(Week)	曜日指定の Wake On RTC 機能を有効にします。
Enable(Day)	日付指定の Wake On RTC 機能を有効にします。

②　Wake On RTC Timer 設定

Wake On RTC 機能で電源を復帰させる時刻を設定します。

表 2-3-8-3. Wake On RTC Timer

名称	動作
Week	曜日を設定します。 (Wake On RTC Enable の設定で「Enable(Week)」設定時のみ有効)
Day	日を設定します。 (Wake On RTC Enable の設定で「Enable(Day)」設定時のみ有効)
Hour	時を設定します。
Minute	分を設定します。

③　設定の反映

[SET]ボタンを押下することで、設定を反映します。

④　終了

[Close]ボタンを押下することで、ASD Ras Config を終了します。

[SET]ボタンを押下せずに[Close]ボタンを押下した場合、設定は破棄されます。

※注：端末が起動中、もしくは電源未接続の場合は Wake On Rtc Timer は機能しませんので注意してください。

２－３－９　ASD Smart Config について

ASD Smart Config は、m-SATA の S.M.A.R.T.情報監視機能の設定を行えます。

S.M.A.R.T.情報監視機能は、m-SATA の書き込み回数オーバや、BadBlock 検出をトリガに、警告を上げる機能です。

ASD Smart Config を起動するには、メニュー画面から[S.M.A.R.T.　Config]を選択します。

●S.M.A.R.T.監視共有設定

ASD Smart Config の[Common]タブを選択することで、S.M.A.R.T.監視機能の共有設定を行うことが可能です。

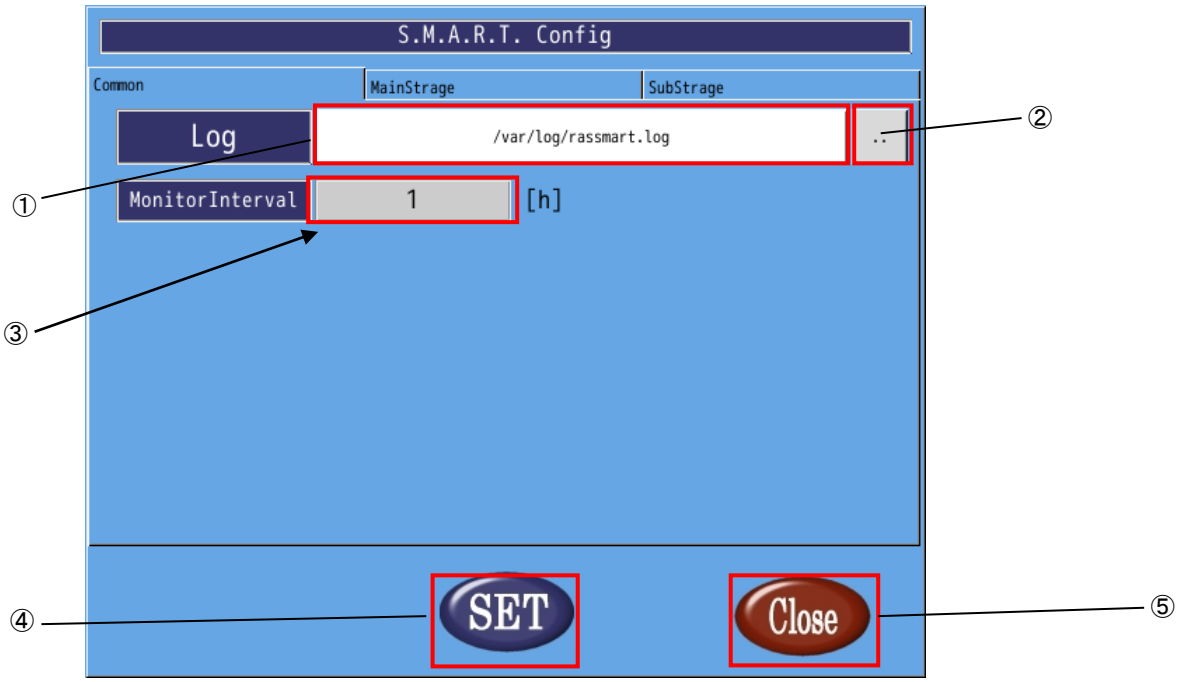


図 2-3-9-1.　S.M.A.R.T.監視共有項目の設定

①　　ログファイルパスの設定

S.M.A.R.T.監視デーモンが、出力するイベント発生ログファイルのパスを指定します。デフォルトでは、/var/log/rassmart.log にログを出力します。

②　　ファイル選択

ファイル選択ダイアログを表示します。

③　　監視インターバル時間

S.M.A.R.T.情報を監視するインターバル時間【H】です。デフォルト設定では、1 時間周期で監視します。

変更する場合は、「/etc/asd_conf/smartd.conf」の「cycletime=1」の項目の数値を変更してください。

設定した数値 x1 時間単位で監視時間を変更できます。

④　　設定の反映

[SET]ボタンを押下することで、設定を反映します。

⑤　終了

[Close]ボタンを押下することで、ASD Smart Config を終了します。

[SET]ボタンを押下せずに[Close]ボタンを押下した場合、変更された設定は破棄されます。

●MainStrage/SubStrage 設定

ASD Smart Config の[MainStrage]タブまたは、[SubStrage] タブを選択することで、それぞれの、S.M.A.R.T. 情報（抜粋）と監視イベント設定を行うことができます。

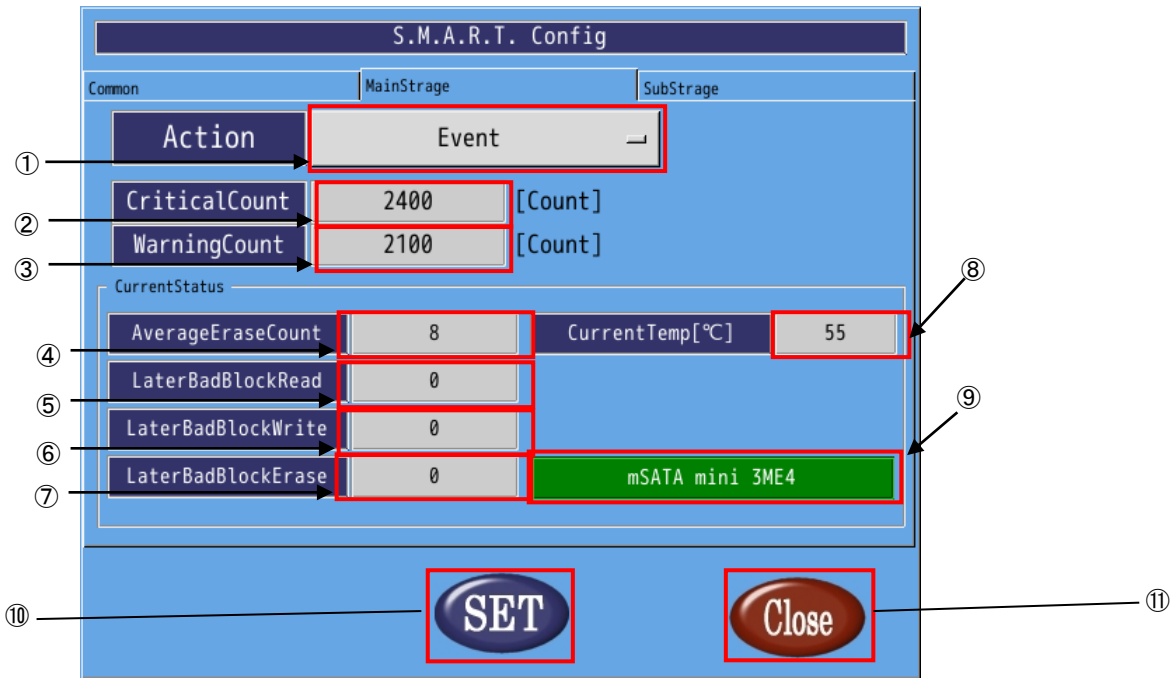


図 2-3-9-2.　S.M.A.R.T.監視 MainStrage/SubStrage の設定

①　動作の設定

S.M.A.R.T.監視デーモンが警報検出した時の動作を設定します。

選択できる動作を表 2-3-9-1 に示します。デフォルトでは Event が選択されます。

表 2-3-9-1.　S.M.A.R.T.警報検出時の動作

名称	動作
None	警報を上げません。
Popup	ポップアップメッセージを表示します。 （デーモン起動方法を変更する必要があります。）
Event	ユーザアプリケーションにイベント発生を通知します。

②　クリティカルイベント発報回数

クリティカルイベントを発報する回数です。④の AverageEraseCount がこの回数を超えた場合に発報します。直ちに、m-SATA を交換する必要があります。

③　ワーニングイベント発報回数

ワーニングイベントを発報する回数です。④の AverageEraseCount がこの回数を超えた場合に発報します。m-SATA 交換を検討する必要があります。

④　AverageEraseCount 値

MainStrage/SubStrage の S.M.A.R.T.情報で、AverageEraseCount の現在の値を表示します。MLC タイプでは 3000 回、iSLC タイプでは 20000 回が寿命になります。

⑤　LaterBadBlock（Read）値

MainStrage/SubStrage の S.M.A.R.T.情報で、BadBlock（Read）の現在の値を表示します。1 カウントアップされれば、クリティカルイベントを発報します。

⑥　LaterBadBlock（Write）値

MainStrage/SubStrage の S.M.A.R.T.情報で、BadBlock（Write）の現在の値を表示します。1 カウント

アップされれば、クリティカルイベントを発報します。

⑦　LaterBadBlock（Erase）値

MainStrage/SubStrage の S.M.A.R.T.情報で、BadBlock（Erase）の現在の値を表示します。1 カウントアップされれば、クリティカルイベントを発報します。

⑧　チップ温度

MainStrage/SubStrage の S.M.A.R.T.情報で、コントローラチップの温度を表示します。表示された温度は更新されません。一度、終了して再表示していただく必要があります。

⑨　m-SATA 名称

MainStrage/SubStrage の認識している m-SATA 名称です。

⑩　設定の反映

[SET]ボタンを押下することで、設定を反映します。

⑪　終了

[Close]ボタンを押下することで、ASD Smart Config を終了します。

[SET]ボタンを押下せずに[Close]ボタンを押下した場合、設定は破棄されます。

２－３－１０　ASD Shutdown Menu について

ASD Shutdown Menu によって、EC4A シリーズの再起動、シャットダウンを行うことができます。
また、起動時のメニューを変更することもできます。
ASD Shutdown Menu を起動するには、メニュー画面から[Shutdown]を選択します。

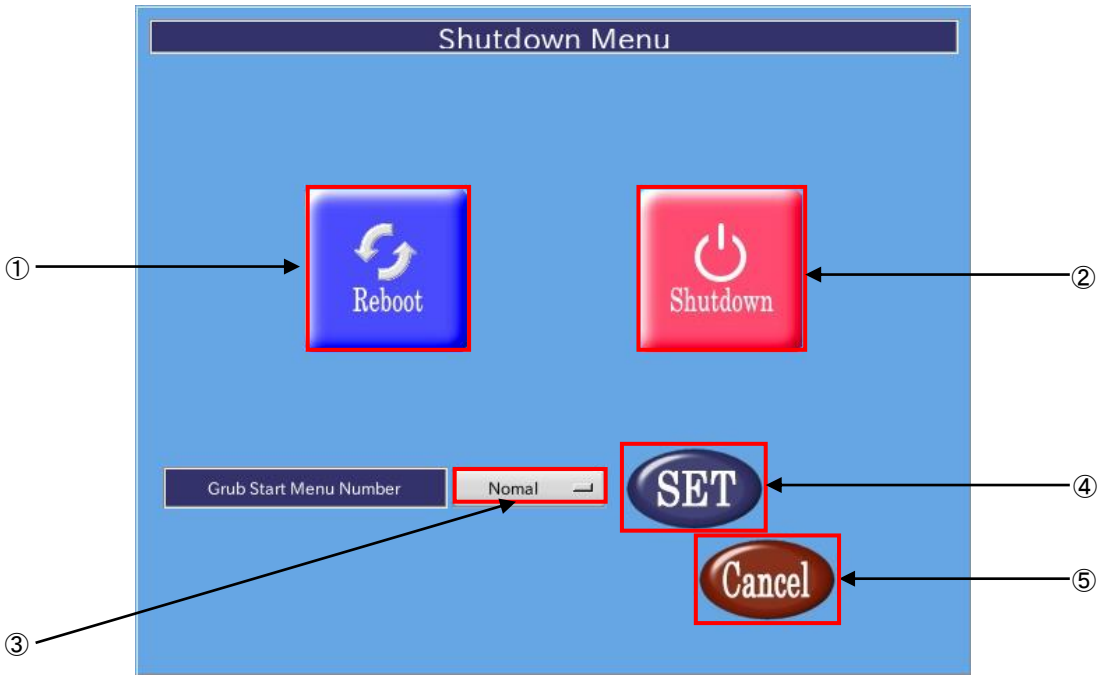


図 2-3-10-1. ASD Shutdown Menu

① 再起動
[Reboot]ボタンを押下することで、EC4A シリーズが再起動します。

② シャットダウン
[Shutdown]ボタンを押下することで、EC4A シリーズがシャットダウンします。

③ Grub Menu の選択
[Grub Start Menu Number]の項目を選択することで、起動モードを変更できます。
選択できるモードを表 2-3-10-1 に示します。

表 2-3-10-1. 起動モード

項目名	内容
Normal	通常モードです。 全てのディレクトリを読書きできます。

④ 設定の反映
[SET]ボタンを押下することで、Grub Start Menu Number で選択した起動モードを反映します。
[SET]ボタンを押下しないと設定が反映されないので注意してください。

⑤ 終了
ASD Shutdown Menu を終了して ASD Config Menu に戻ります。

２－４　有線 LAN の設定について

本稿では、EC4A シリーズにおける有線 LAN の設定方法について説明します。

①　[アプリケーション]→[設定]→[ネットワーク接続]を選択します。

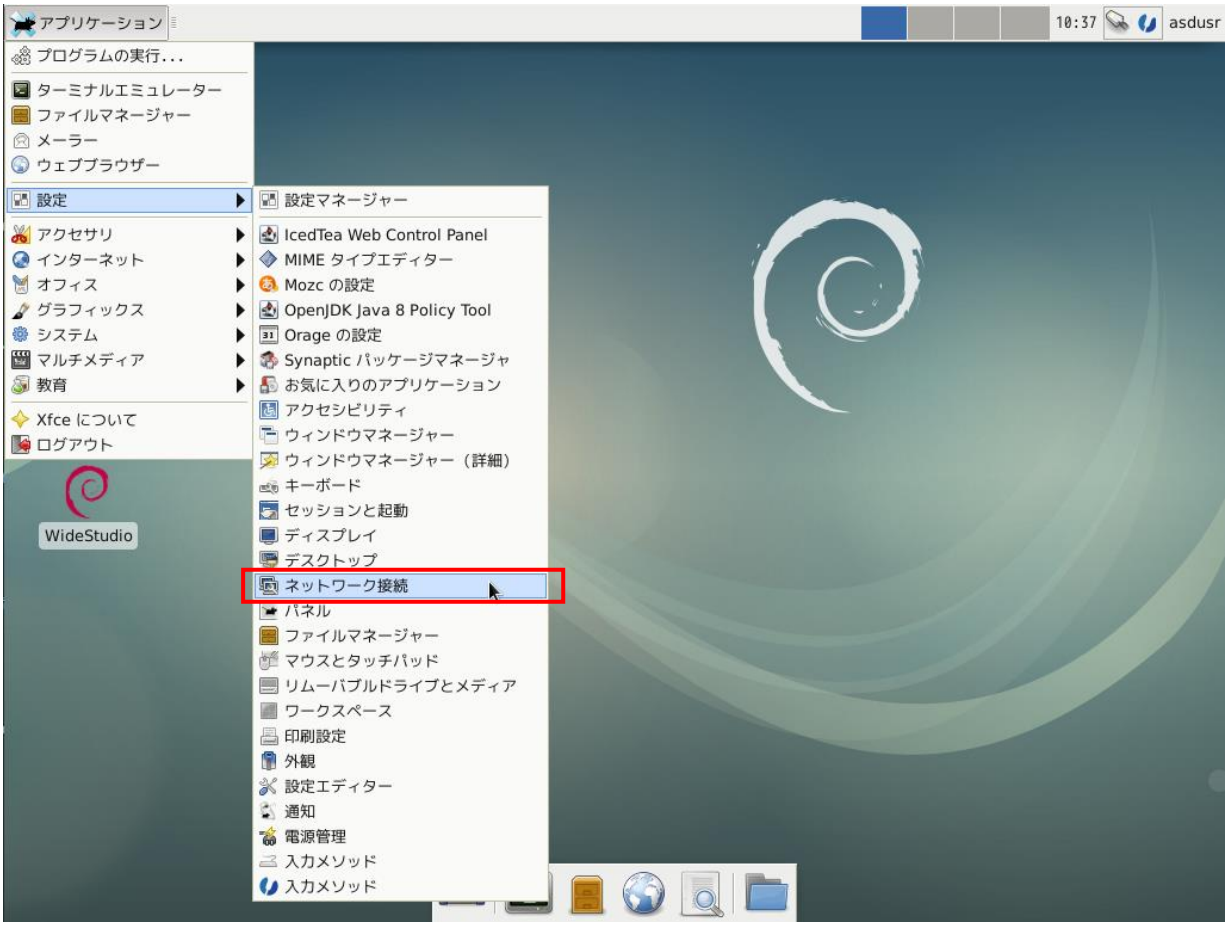


図 2-4-1.　ネットワーク設定画面の起動

② 図 2-4-2 のようなネットワーク設定画面が起動します。認識している LAN の一覧が表示されています。設定したい接続を選択して[編集]ボタンをクリックします。

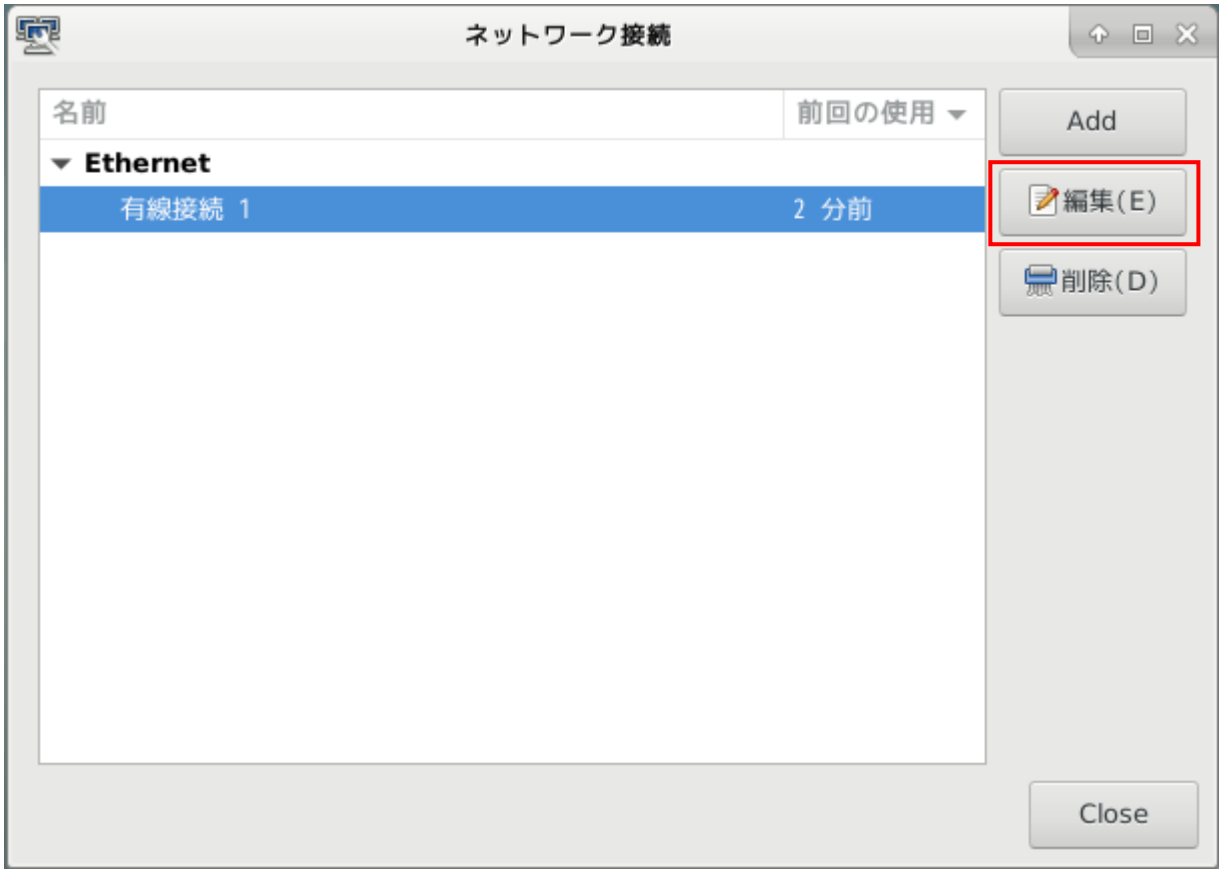


図 2-4-2. ネットワーク設定画面

③　図 2-4-3 のような有線 LAN の設定画面が起動します。
接続しているネットワークの環境に合わせた設定を行ってください。

図 2-4-3. 有線 LAN 設定画面

④　「OK」をクリックすることで、自動的に再接続されます。設定されている IP アドレスを確認する場合は、コンソールウィンドウを起動して下記のコマンドを実行してください。

・現在の設定を見る場合

```
$ ip a
```

IP アドレスの確認方法について、従来の Linux ディストリビューションでは、IP アドレスを確認するために、「ifconfig」というコマンドを使用されていました。
しかし、「ifconfig」を含む「net-tools」というパッケージは、メンテナンスされいないため、数年前に非推奨なパッケージになりました。
Debian9.0 からは「net-tools」パッケージはインストールされておらず、「ifconfig」コマンドも使えません。
今日の Linux ディストリビューションでは、「iproute2」（「ip」コマンドの正式名）が正式採用されています。

「ifconfig」コマンドに対応される「ip」コマンドは以下になります。

	ifconfig	iproute2
各種インターフェースの設定と表示	ifconfig	ip a (ip addr show)
インターフェースの起動	ifconfig eth0 up	ip link set eth0 up
インターフェースの停止	ifconfig eth0 down	ip link set eth0 down

「ip」コマンドは、「ifconfig」よりも、豊富な機能を搭載しています。詳細は、インターネット等で確認してください。

２－５　無線 LAN の設定について

本項では、EC4A シリーズにおける無線 LAN の設定方法について説明します。

●接続方法

①デスクトップ画面右下のネットワークアイコン  をクリックすると、通信可能なアクセスポイントの一覧が表示されます。（通信状況によってネットワークアイコンは変化します。）
通信したいアクセスポイントを選択してください。

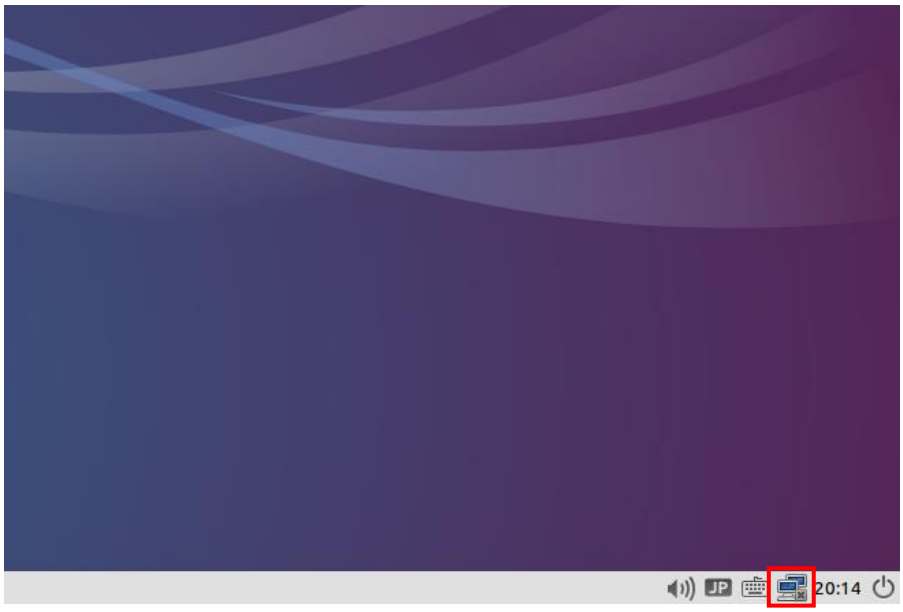


図 2-5-1. ネットワークアイコン

②図 2-5-2 のようなキー入力画面が表示されます。キーを入力し、[接続]を押してください。

図 2-5-2. キー入力画面

③接続に成功すると、図 2-5-3 のように、デスクトップ右下のネットワークアイコンが  から  に変化し、画面右上に接続成功の表示が現れます。（通信状況によってネットワークアイコンは変化します。）

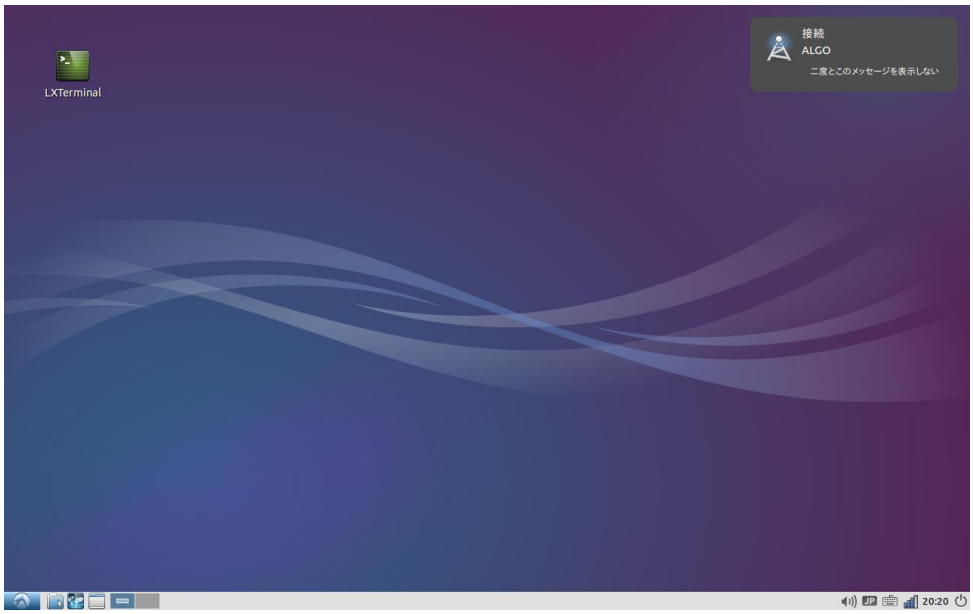


図 2-5-3.　接続成功

●無線 LAN の設定変更

無線 LAN の設定を変更したい場合は、下記の方法で設定してください。

①　[メニューアイコン]→[設定]→[ネットワーク接続]を選択します。

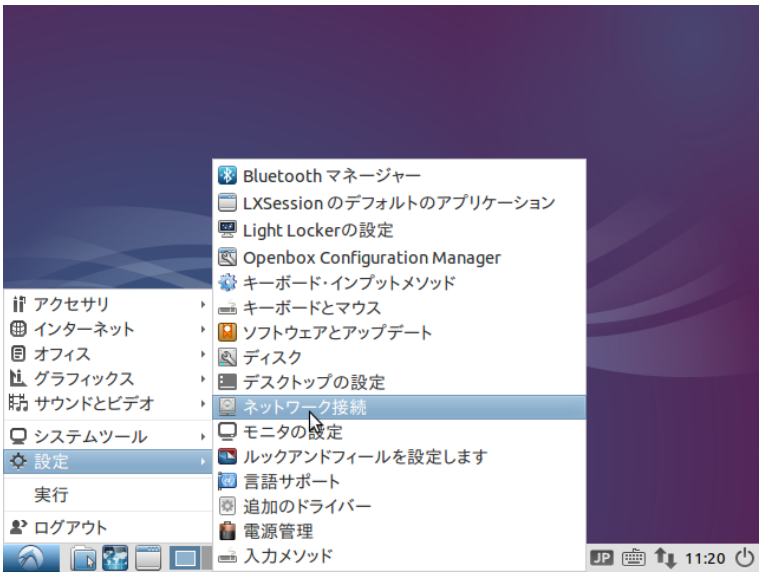


図 2-5-4.　ネットワーク設定画面の起動

②　　図 2-5-5 のようなネットワーク設定画面が起動し、アクセスポイントの一覧が表示されます。設定したいアクセスポイントを選択して[編集]ボタンをクリックします。

図 2-5-5.　ネットワーク設定画面

③　　図 2-5-6 のような設定画面が起動します。接続しているネットワークの環境に合わせた設定を行ってください。

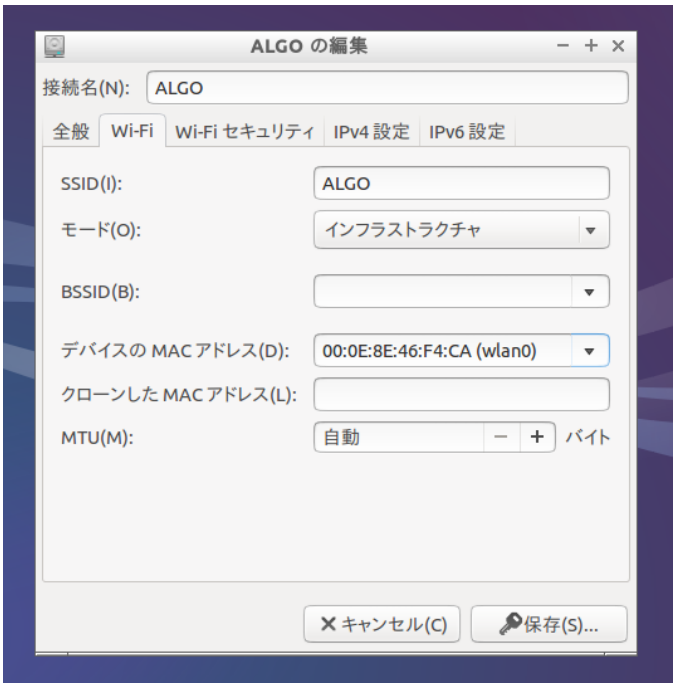


図 2-5-6.　無線 LAN 設定画面

２－６　sysfs ファイルシステム

Linux2.6 カーネルから導入された sysfs ファイルシステムは、proc、devfs、devpty のファイルシステムの統合だと言えます。sysfs ファイルシステムは、システムに接続されているデバイスとバスを、ユーザースペースからアクセスできるファイルシステム内に階層式に列記します。

sysfs ファイルシステムは /sys/ でマウントされ､いくつか異なる方法でシステムに接続されたデバイスを構成する複数のディレクトリを含んでいます。

EC4A シリーズの固有デバイスとして以下のデバイス制御が可能です。

２－６－１　汎用 SW と汎用 LED の制御

汎用 SW(初期化スイッチ)の状態確認と 3 つの汎用 LED の制御ができます。

表 2-6-1-1. sysfs の汎用 SW の入力状態を確認する項目

sysfs ファイル名	データ	内容
/sys/devices/platform/miscio/miscio_sw	0～1 SW1：1bit 目	Read することで汎用 SW の状態を確認することができます。 汎用 SW1 は、電源 ON 直後に 3 秒間長押しすると、ON 状態でラッチします。0 を Write することでラッチを解除します。 それ以外の場合は押下されたときに ON にします。
/sys/devices/platform/miscio/miscio_led	0～7 LED1：1bit 目 LED2：2bit 目 LED3：3bit 目	対応する bit を ON したデータを Write することで 3 つの汎用 LED を点灯することができます。 Read することで、現在の LED の状態を読み出すことができます。 **※注：汎用 SW1 が ON ラッチしている間は LED1 が点滅します。**

２－６－２　基板情報

基板情報を管理します。

表 2-6-2-1. sysfs の基板情報を制御する項目

sysfs ファイル名	データ	内容
/sys/devices/platform/mainboard/boardversion	0XXX	Read することで FPGA バージョンが読み出せます。
/sys/devices/platform/mainboard/buildversion	XXXXXXXX	Read することでカーネルのビルドバージョンが読み出せます。
/sys/devices/platform/mainboard/machcode	0000XXXX	Read することでマシンコードが読み出せます。
/sys/devices/platform/mainboard/dipswitch	X0XXX00000000	Read することで PCI デバイス側のマシンコードが読み出せます。
/sys/devices/platform/asd_sram/size	XXXX	Read することで仮想 RAM のサイズが読み出せます。単位は[kbyte]です。
/sys/devices/platform/asd_ups/condition	X	Read することで UPS 状態が読み出せます。 0：バッテリ充電中 1：バッテリ満充電 2：バッテリ放電中 3：バッテリ異常
/sys/devices/platform/asd_ups/powersource	X	Read することで UPS 電力源が読み出せます。 0：AC 1：バッテリ

２－６－３　温度センサ

基板上の温度センサの情報を取得します。

表 2-6-3-1. sysfs の基板上温度センサの情報を取得する項目

sysfs ファイル名	データ	内容
/sys/bus/i2c/devices/0-004c/hwmon/hwmon2/temp1_input	XX000	Read することで基板上の温度センサ(DRAM 付近)の温度を読み取ります。
/sys/bus/i2c/devices/0-004d/hwmon/hwmon3/temp1_input	XX000	Read することで基板上の温度センサ(PMIC 付近)の温度を読み取ります。
/sys/bus/i2c/devices/0-004e/hwmon/hwmon4/temp1_input	XX000	Read することで基板上の温度センサ(UPS バッテリ付近)の温度を読み取ります。

２－７　データの保護について

２－７－１　データ保護の必要性と方法

Linux ではハードディスクや CF カード、SD カード、USB メモリ等のストレージ上にファイルを Write する際、一端書込みデータをキャッシュ領域に保存して、Write 処理を完了し、OS のアイドル時間に実際のストレージへ書込むことで CPU リソースの有効活用を行っています。

このため､キャッシュ領域にデータがあり､実際のストレージ上に書込まれる前に電源を落としたりした場合、そのファイルまたは書込み途中のセクタが破損する可能性があります。これを防ぐ為に、sync というコマンドがあります。このコマンドを実行することで、キャッシュ内にたまっているデータを実際のストレージ上にすべて書き出すことができます｡ストレージにファイルを書込んだ際は電源を落とす前に sync コマンドを実行してください。

また、SD カードや USB メモリ等の抜き差しが可能なデバイスの取扱いには注意が必要となります。使用中にデバイスが抜かれた場合などは、デバイス内のファイルが破損してしまう場合があります。

また、デバイスにファイルを書込んだ場合は、sync コマンドなどを使用して書込んだ内容が確実にデバイスに書込まれるようにしてください。また、デバイスを抜く際には、必ずデバイスをアンマウントしてから抜くようにしてください。

● sync コマンドの使用

Linux でファイル操作を行った場合、ファイルデータがファイルキャッシュとしてシステムメモリに保存され、実際の SD カードなどのデバイスには反映されていないことがあります。デバイスのファイル操作後、変更されたデータがデバイスに確実に反映されるようにするには、sync コマンドを使用してキャッシュとデバイスのデータの同期をとるようにしてください。

デバイスにファイルコピー後、sync コマンドで同期

```
# cp /home/asdusr/FILE.dat /media/asdusr/デバイス名
# sync
```

● USB メモリの書込み不可/可能の切り替え

USB メモリを書込み不可状態でマウントし、書込み可否の切り替えを行います。書込み不可状態では、ファイルの書込みを行えませんがファイルの読み出しは行うことができます。

USB メモリを書込み不可でマウントします。

```
# mount -o ro /dev/sdb1 /mnt
```

書込み不可でマウントされている USB メモリを書込み可能にします。

```
# mount -o remount,rw /mnt
```

書込み可能でマウントされている USB メモリを書込み不可にします。

```
# mount -o remount,ro /mnt
```

２－８　ソフトウエアキーボードについて

Algonomix6 ではソフトウエアキーボードがあらかじめセットアップされています。
ソフトウエアキーボードは以下の手順で起動することができます。

①　[メニューアイコン]→[ユニバーサル・アクセス]→[Onboard]を選択します。

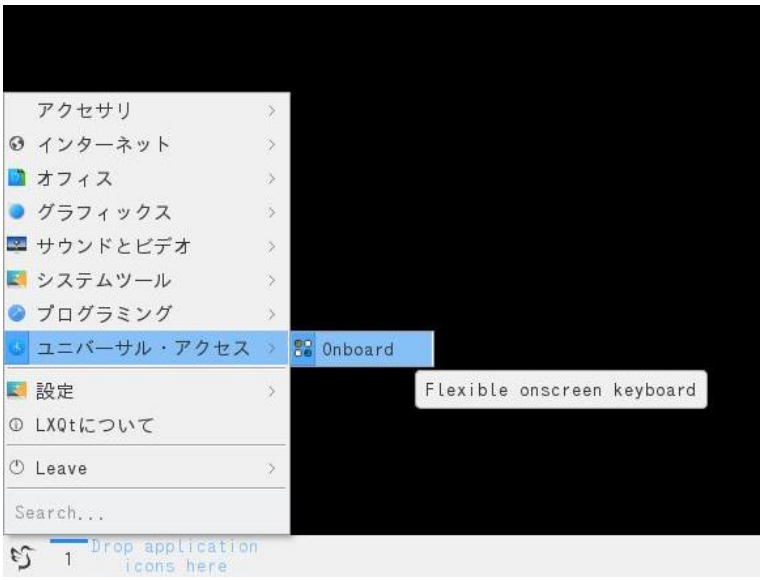


図 2-8-1.　ソフトウエアキーボードの起動

②　ソフトウエアキーボードが表示されます。

図 2-8-2.　ソフトウエアキーボード

本ソフトウエアキーボードはタッチパネルをマウスとして使用できるようにするモードを持ちます。
この機能を使用する場合はマウスカーソルキー(※)をタッチしてください。

第３章　開発環境

Algonomix6 の開発環境について、簡単な説明が記述されています。詳細については、「Algonomix6 開発環境ユーザーズマニュアル」を参照してください。

３－１　クロス開発環境

プログラムを開発する場合に必要となるのが、ソースコードを記述するエディタ、ソースコードをコンパイルするコンパイラ、コンパイルされたプログラムを実行する為の実行環境です。

例えば、Microsoft 社の Windows 上で動作するアプリケーションを開発する場合、エディタでソースを書き、Visual Studio 等のコンパイラでコンパイルを行い、作成された exe ファイルを実行します。これで作成したアプリケーションが Windows 上で実行されます。

Linux の場合でも同じです。Linux マシン上で動作するエディタでソースを書き、gcc でコンパイル後に生成された実行ファイルを実行します。

両者とも、コンパイルと実行を同じパソコン環境上で行うことができます。このような開発方式をセルフ開発といいます。

EC4A シリーズでは、クロス開発方式を採用しています。クロス開発とは、コンパイル環境と実行する環境が異なる方式です。ソースコードの記述やコンパイルはパソコン上で行い、LAN 等で実行ファイルをターゲットに送って実行することになります。(図 3-1-1 参照)。EC4A シリーズはターゲットマシンとして開発された商品である為、セルフコンパイル環境は用意していません。

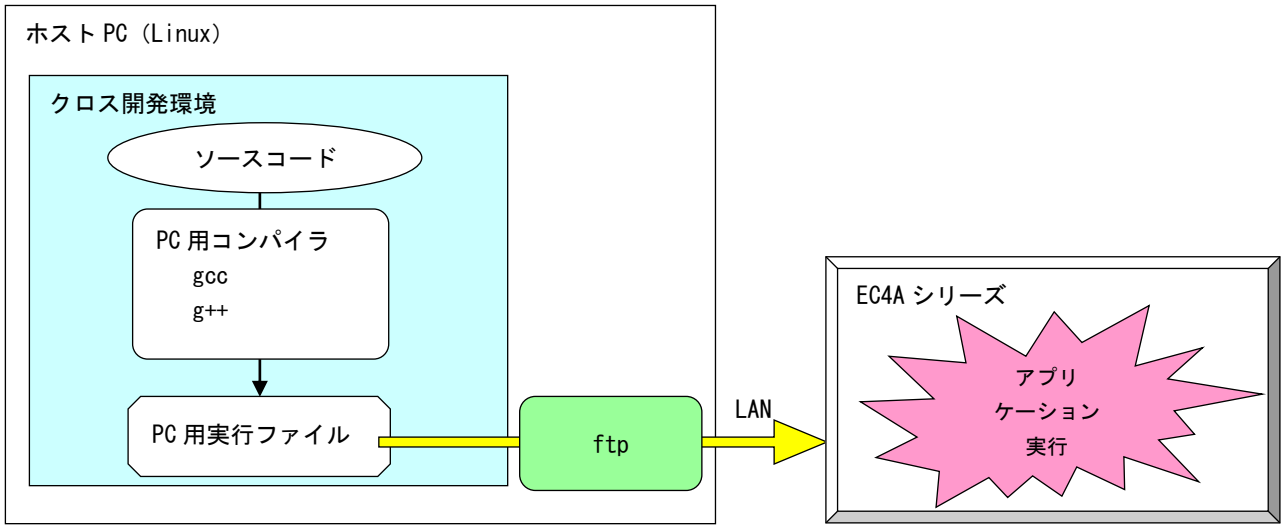


図 3-1-1. クロス開発方式イメージ図

このような開発環境を構築するには、表 3-1-1 に示すような開発環境ツールが必要となります。

表 3-1-1. クロス開発に必要なツール

ツール名	説明
GCC	GNU C コンパイラ
binutils	リンカ、アセンブラ等のソフトウェア開発ツール
GDB	デバッガ
glibc	GNU C ライブラリ

Algonomix6 開発環境は、Debian9.0 ディストリビューションイメージに、Intel CPU 用と ARM CPU 用の２種類の開発環境を組み込んだものになります。

VirtualBox という仮想マシン上で Algonomix6 開発環境を起動すれば、それぞれの CPU 搭載端末用のアプリケーションを開発することが可能です。

第４章　産業用組込み PC EC4A IoT シリーズについて

本章では、EC4A シリーズに実装されているデバイスの使用方法およびアプリケーション作成について説明しています。

Algonomix6 開発環境には、コンソール用と WideStudio 用の 2 種類のサンプルプログラムのソースコードを用意しています。それぞれ表 4-1 にコンソール用サンプル、表 4-2 に WideStudio 用サンプルのディレクトリ名と内容を示します。

コンソール用サンプルプログラムは、コンソール上で「make」コマンドを実行することでコンパイルします。WideStudio 用のサンプルプログラムは、WideStudio を起動して、プロジェクトファイルをオープンしコンパイルします。コンパイル方法は『第 3 章　開発環境』を参照してください。

※注：コンソール用サンプルプログラムは「/usr/local/tools-0010/samples/sampleConsole」に、WideStudio 用サンプルプログラムは「/usr/local/tools-0010/samples/sampleWideStudio」に格納されています。

※注：機種により搭載されているデバイスは異なります。そのため、一部のサンプルでは EC4A シリーズにデバイスが実装されていない為動作しないことがあります。

表 4-1. コンソール用サンプルプログラムのソースコード一覧

ディレクトリ名	内容	機能有無
sample_DIO	汎用入出力制御方法	『4-1　汎用入出力』を参照
sample_SerialPortChange	シリアルポート RS422/R485/RS232C 切り替え	『4-2　シリアルポート』を参照
sample_Serial	シリアルポート制御方法	『4-2　シリアルポート』を参照
sample_TcpIp	ネットワークポート制御方法	『4-3　ネットワークポート』を参照
sample_BeepOnOff	ブザーON/OFF 制御方法	―
sample_Reset	汎用入力 IN0 リセット制御方法	『4-4　RAS 機能』を参照
sample_Interrupt	汎用入力 IN1 割込み制御	『4-4　RAS 機能』を参照
sample_SRAM	バックアップ SRAM 制御方法　(read/write)	『4-4　RAS 機能』を参照
sample_HwWdtKeepAlive	ハードウェア・ウォッチドッグタイマ操作	『4-4　RAS 機能』を参照
sample_HwWdtEventWait	ハードウェア・ウォッチドッグタイマイベント待ち	『4-4　RAS 機能』を参照
sample_SMART	S.M.A.R.T.監視イベント通知待ち	『4-4　RAS 機能』を参照
sample_UPS	UPS 通知待ち	『4-6　UPS 機能』を参照
sample_InputKey	入力されたキーコードの表示	―

表 4-2. WideStudio 用サンプルプログラムのソースコード一覧

ディレクトリ名	内容	機能有無
sample_MultiLang	多言語表示	
sample_10Key	10 キー入力画面	
sample_Launcher	起動ランチャー	
sample_ColorPalette	カラーパレット画面	
sample_Gui	hello サンプル	
sample_FontSet	WideStudio 上でのフォントの変更	
sample_ColorPaletteCustom	カラーパレットの変更	

固有デバイスの説明の前に、Linux の一般的なデバイスドライバアクセスについて説明します。デバイスにアクセスするには「/dev」以下に格納されているデバイスファイルに対しシステムコール（open、close、read、write、ioctl 等）を使用します。

デバイスドライバ操作は、デバイスファイルを「open」関数にてオープンし、「read」関数や「write」関数を使用してデータを読み書きします。デバイスによっては、「ioctl」関数で各種設定を行う場合もあります。また、デバイスによっては、独自のシステムコールが用意される場合もあります。

デバイス毎にどのような設定があるかは、インターネットや書籍で確認してください。ここでは、EC4A シリーズに搭載されたデバイスの仕様について説明します。

EC4A シリーズ本体の外形図を以下に示します。各部名称を表 4-3 に示します。

※注：搭載されているデバイスの種類や実装位置は、機種毎に異なります。
それぞれの機種の詳細はハードウエアマニュアルをご参照ください。

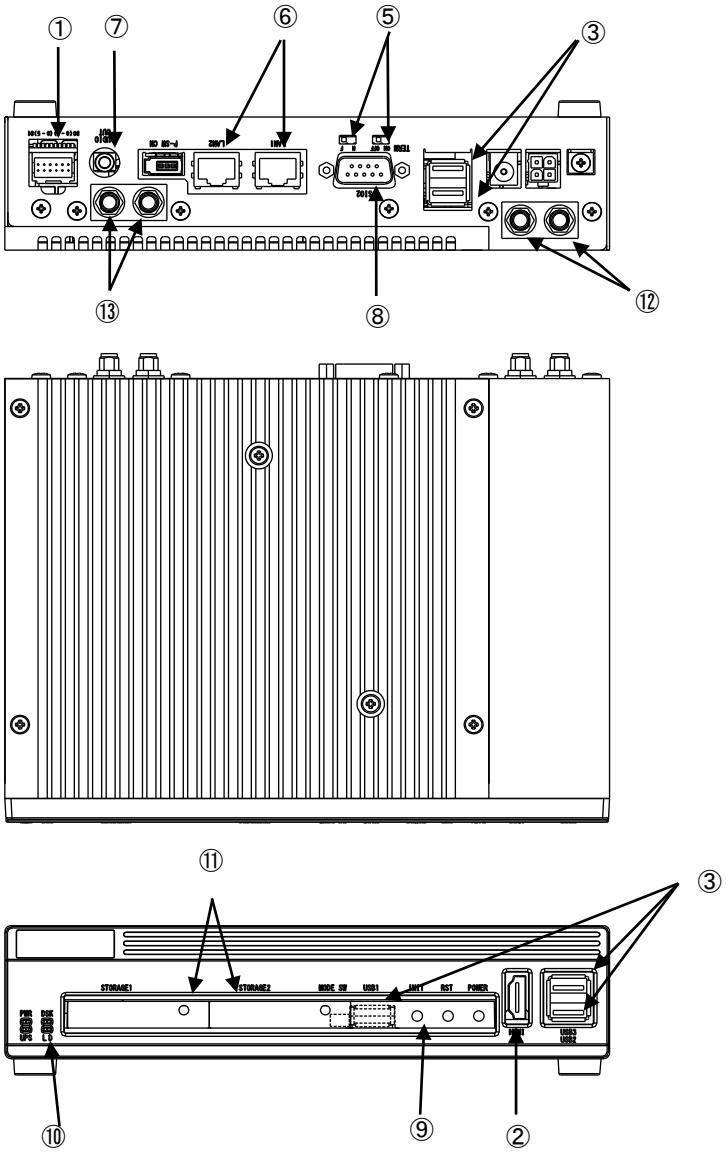


図 4-1. 外形図（EC4A-010CT）

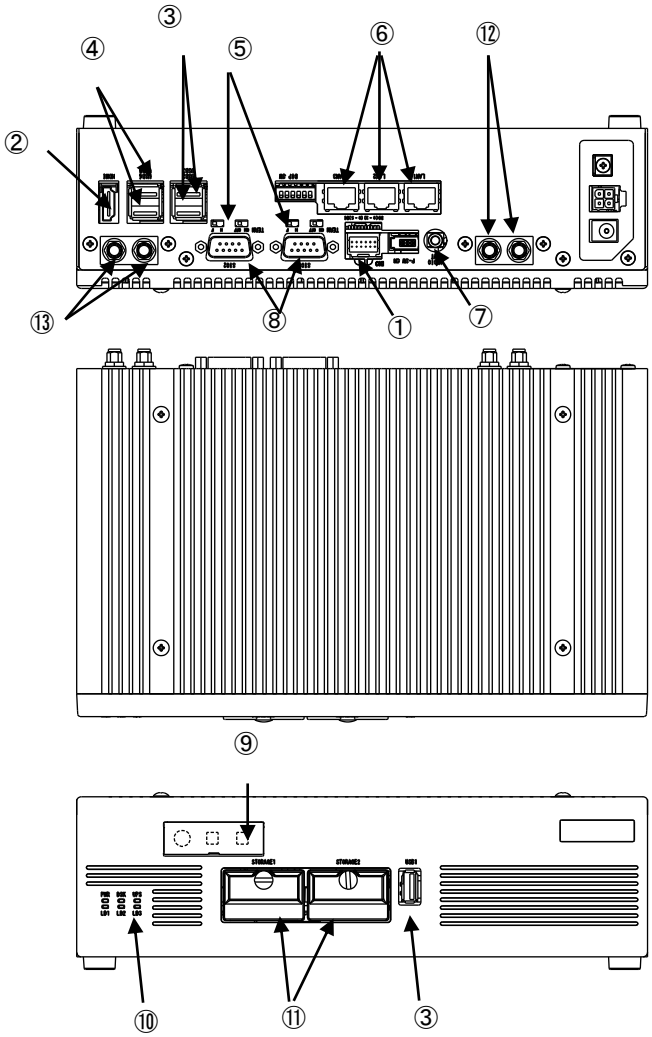


図 4-2. 外形図 (EC4A-110CA)

表 4-3. 各部名称

No.	名称	機能	説明	実装数※1		
				010CT	100 CA/CT	110 CA/CT
①	DIO インターフェース	汎用入出力	汎用の入出力です。 入力 6 点、出力 4 点を制御できます。	1	1	1
②	HDMI インターフェース	グラフィック サウンド	外部モニタに画面、音声を出力できます。	1	1	1
③	USB2.0 インターフェース	USB2.0 ポート	USB1.1/2.0 の機器を接続することができます。	5	3	3
④	USB3.0 インターフェース	USB3.0 ポート	USB1.1/2.0/3.0 の機器を接続することができます。	–	2	2
⑤	シリアル 設定スイッチ	シリアルポート タイプ設定	SIO1、SIO2 のシリアルポートタイプを設定します。 シリアルポート設定スイッチと組込みシステム機能のシリアルコントロール機能を使用することで、SIO1、SIO2 で 232C/422/485 の通信を行うことができます。 設定方法は、「ユーザーズ（ハードウェア）マニュアル」を参照してください。 **※　シリアルポートタイプを切替える場合は、シリアルコントロール機能の設定に合わせて、シリアルポート設定スイッチを設定してください。**	2	2	2
⑥	ネットワーク インターフェース	有線 LAN	ネットワークポートとして使用できます。	2	3	3
⑦	サウンド	サウンド	音声出力が使用できます	1	1	1
⑧	シリアル インターフェース	シリアルポート	シリアル通信が行えます。 SIO1、SIO2 は、シリアルポート設定スイッチと組込みシステム機能のシリアルコントロール機能と併用することで、232C/422/485 の通信を行うことができます。 SIO1: /dev/ttyS4 SIO2: /dev/ttyS5	1	2	2
⑨	初期化スイッチ	スイッチ	電源投入時、初期化スイッチを 3 秒間押すことで、LD1 が点滅します。また、アプリケーションにより電源投入時にスイッチが押されたかどうかを確認できます。	1	1	1

No.	名称	機能	説明	実装数※1		
				010CT	100 CA/CT	110 CA/CT
⑩	汎用 LED	LED	アプリケーションにより 3 点の LED の点灯/消灯を制御できます。 LD1: 電源投入後、SW1 を 3 秒間長押しすると点滅します。 LD2: アプリケーションで制御できます。 LD3: アプリケーションで制御できます。	1	1	1
⑪	mini m-SATA スロット	mini m-SATA SSD	記憶領域として mini m-SATA SSD を使用することができます。 Strage1 (mini m-SATA) : メインストレージ ルートファイルシステムとして使用します。 Strage2 (mini m-SATA) : サブストレージ オプションでデータ領域を追加することができます。	1 (1)	1 (1)	1
ー	2.5 インチ HDD	2.5inch HDD	記憶領域として 2.5inch HDD を使用することができます。	-	-	1
⑫	LTE(外部アンテナ) （オプション）	LTE 通信	LTE 通信を使用することができます。	(2)	(2)	(1)
⑬	無線 LAN(外部アンテナ) （オプション）	無線 LAN	無線ネットワークデバイスとして使用できます。	(2)	(2)	(1)
ー	Bluetooth※2 (オプション)	Bluetooth	Bluetooth 機器を接続することができます。	(1)	(1)	(1)

※1：実装数の()はオプションを表します。

※2：対応するプロファイル一覧を表 4-4 に示します。

表 4-4. Bluetooth 対応プロファイル一覧

プロファイル名	説明
A2DP 1.3	高度オーディオ配信プロファイル
AVRCP 1.5	オーディオ/ビデオ リモート制御プロファイル
DI 1.3	デバイス識別プロファイル
HDP1.0	ヘルスデバイス プロファイル
HID 1.0	ヒューマン インターフェイス デバイス プロファイル
PAN 1.0	パーソナルエリア ネットワーク プロファイル
SPP 1.1	シリアルポート プロファイル
PXP 1.0	近接プロファイル
HTP 1.0	体温計プロファイル
HoG 1.0	マウスやキーボードなどを接続するためのプロファイル
TIP 1.0	タイム プロファイル
CSCP 1.0	回転速度とケイデンス プロファイル
FTP 1.1	ファイル転送プロファイル
OPP 1.1	オブジェクト プッシュ プロファイル
PBAP 1.1	電話帳情報を伝送するためのプロファイル
MAP 1.0	メッセージ アクセス プロファイル
HFP 1.6 (AG &HF)	ハンズフリー プロファイル

４－１　汎用入出力

４－１－１　汎用入出力について

EC4A シリーズでは出力 4 点、入力 6 点を使用することができます。これらの入出力を使用することにより外部 I/O 機器を制御することが可能です。

４－１－２　汎用入出力デバイスドライバについて

入力用のデバイスファイル（/dev/genin）と出力用のデバイスファイル（/dev/genout）に分かれています。キャラクタデバイス方式で入出力の状態を読み書きします。

表 4-1-2-1 に汎用入出力のリファレンスを示します。

表 4-1-2-1. 汎用入出力デバイスリファレンス

GENIN,GENOUT

名前

genin　－　汎用入力6点
genout　－　汎用出力4点

説明

汎用入力6点の状態読み出しと、汎用出力4点の制御を行います。

OPEN

汎用入力デバイス(/dev/genin,/dev/genout)をOpen関数でオープンします。

```
infd = open("/dev/genin", O_RDWR);
outfs = open("/dev/genout", O_RDWR);
```

READ

read関数を用いて汎用入出力の状態をモニタすることができます。

```
char in;
char out;
len = read(infs, &in, 1);
len = read(outfs, &out, 1);
```

汎用入出力デバイスをリードすると1Byteのデータが即リターンされます。
リターンされた値は現在の入出力状態です。

bit	7	6	5	4	3	2	1	0
IN			IN5	IN4	IN3	IN2	IN1	IN0

bit	7	6	5	4	3	2	1	0
OUT					OT3	OT2	OT1	OT0

WRITE

write関数を用いて汎用出力を制御することができます。

```
char out;
out = 1;
len = write(outfd, &out, 1);
```

汎用出力データを1Byteライトすることで対応するビットの出力をON/OFFすることができます。

bit	7	6	5	4	3	2	1	0
OUT					OT3	OT2	OT1	OT0

４－１－３　汎用入出力サンプルプログラム

●WideStudio 用サンプルプログラム
「/usr/local/tools-0010/samples/sampleWideStudio/sample_GenIO」に、汎用入出力を使ったサンプルプログラムが入っています。

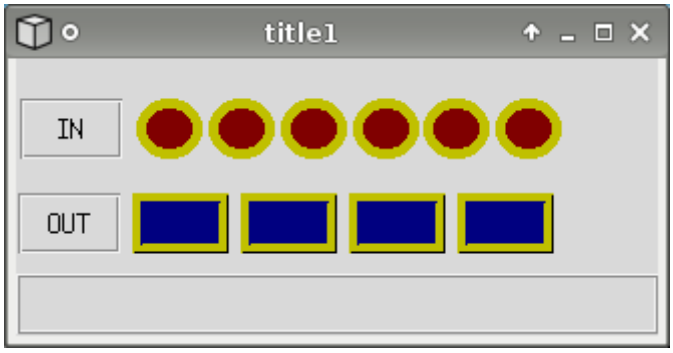


図 4-1-3-1. 汎用入出力デバイス制御サンプルプログラム起動画面

このサンプルプログラムは、スレッド内で汎用入力状態を監視し、入力状態によって IN の色を変化させます。また、OUT のボタンを押下することで、汎用出力が ON/OFF します。

汎用入出力デバイスのオープンとスレッドの生成を記したコードをリスト 4-1-3-1 に示します。
Main_Init 関数は、メインウィンドウの「Initialize」イベントで実行されるように設定されています。この中で、「open」関数を使用し、汎用入力のデバイスファイル「/dev/genin」と汎用出力のデバイスファイル「/dev/genout」をリードライトモードでオープンします。汎用入出力デバイスをリードライトすることで汎用入出力を制御することができます。
また、汎用入力の状態をモニタする為にスレッドを生成しています。

リスト 4-1-3-1. 汎用入出力デバイスのオープンとスレッドの生成（Main_Init.cpp）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "newwin000.h"
#include "IOThread.h"

//---------------------------------------------------------
//Function for the event procedure
```

```
//---------------------------------------------------------
void Main_Init(WSCbase* object){
unsigned char data;

        /*汎用出力ﾃﾞﾊﾞｲｽのｵｰﾌﾟﾝ*/
    out_fd = open("/dev/genout", O_RDWR);        //汎用出力ﾃﾞﾊﾞｲｽｵｰﾌﾟﾝ
    if(out_fd < 0){                              //ｴﾗｰ処理
        Status_Bar->setProperty(WSNlabelString,"OUT Device Open Error");
        return;
    }
    data = 0;
    write(out_fd, &data, 1);                     //初期 0 出力

        /*汎用入力ﾃﾞﾊﾞｲｽのｵｰﾌﾟﾝ*/
    in_fd = open("/dev/genin", O_RDWR);          //汎用入力ﾃﾞﾊﾞｲｽｵｰﾌﾟﾝ
    if(in_fd < 0){                               //ｴﾗｰ処理
        Status_Bar->setProperty(WSNlabelString,"IN Device Open Error");
        close(out_fd);
        return;
    }

        /*ｽﾚｯﾄﾞの生成*/
    ioctrl_thr = 0;
    ioctrl_thr = WSDthread::getNewInstance();           //ｽﾚｯﾄﾞｲﾝｽﾀﾝｽ取得
    ioctrl_thr->setFunction(Ioctrl_Thread);             //ｽﾚｯﾄﾞ本体関数を設定
    ioctrl_thr->setCallbackFunction(Io_callback_func);  //ｺｰﾙﾊﾞｯｸ関数を設定
    ioctrl_thr->createThread((void*)0);                 //ｽﾚｯﾄﾞを生成
}
static WSCfunctionRegister  op("Main_Init",(void*)Main_Init);
```

リスト 4-1-3-2 に汎用入力をモニタするスレッド本体とコールバック関数のソースコードを示します。

汎用入力デバイスを「read」関数を使い、1 バイトリードすることで、現在の汎用入力の状態を読み出すことができます。汎用入力デバイスは、「read」関数を実行されたら、遅延せずに即時に ON/OFF 状態を返します。このサンプルプログラムのソースコードでは、汎用入力の状態が変化したときのみコールバック関数を呼び出し画面の状態を変化させています。

リスト 4-1-3-2. 汎用入力スレッドのソースコード（IOThread.cpp）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <math.h>

#include <WScom.h>
#include <WSCfunctionList.h>
```

```
#include <WSCbase.h>

#include "newwin000.h"
#include "IOThread.h"

int in_fd;
int out_fd;
WSDthread* ioctrl_thr;
unsigned char o_data;

void *Ioctrl_Thread(WSDthread* obj, void *arg)
{
    int len;
    unsigned char data;
    o_data = data = 0;
    int i;
    i = 0;
    for(;;){
        len = read(in_fd, &data, 1);                    /*汎用入力読みだし*/
        data &= 0x3F;
        if(len==1){
            if(o_data != data){
                o_data = data;
                obj->execCallback((void*)data);
            }
        }
    }
    return(NULL);
}
/*ｽﾚｯﾄﾞから通知され、ﾒｲﾝｽﾚｯﾄﾞで実行されるｺｰﾙﾊﾞｯｸ関数*/
void Io_callback_func(WSDthread *, void *val)
{
unsigned char data;

    data = (unsigned char)(unsigned long)val;
    if(data & 0x01){                    /*入力 1*/
        newvarc_000->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
        newvarc_000->setPropertyV(WSNhatchColor,"#800000");
    }
    if(data & 0x02){                    /*入力 2*/
        newvarc_001->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
        newvarc_001->setPropertyV(WSNhatchColor,"#800000");
    }
    if(data & 0x04){                    /*入力 3*/
        newvarc_002->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
```

```
        newvarc_002->setPropertyV(WSNhatchColor,"#800000");
    }
    if(data & 0x08){                    /*入力 4*/
        newvarc_003->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
        newvarc_003->setPropertyV(WSNhatchColor,"#800000");
    }
    if(data & 0x10){                    /*入力 5*/
        newvarc_004->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
        newvarc_004->setPropertyV(WSNhatchColor,"#800000");
    }
    if(data & 0x20){                    /*入力 6*/
        newvarc_005->setPropertyV(WSNhatchColor,"#FF0000");
    }
    else{
        newvarc_005->setPropertyV(WSNhatchColor,"#800000");
    }
}
```

汎用出力を制御するソースコードをリスト 4-1-3-2 に示します。

メインウィンドウに配置したボタンに「Activate」イベントのプロシージャを作成します。その中で、汎用出力デバイスを「read」関数で現在の状態を読込み、新しい状態に変更し「write」関数でデータを更新します。これでボタンが押下されるたびに汎用出力の状態が切り替わります。

リスト 4-1-3-3. 汎用出力 ON/OFF のソースコード (Btn_Click.cpp)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "newwin000.h"
#include "IOThread.h"

int btn[4]={0,0,0,0};
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
```

```
void Btn_Click(WSCbase* object){
    int len;
    unsigned char data;
    long code;

    code = object->getProperty(WSNuserValue);              //押されたボタン番号を取得
    len = read(out_fd, &data, 1);                          //汎用出力の出力値取得
    if(len==1){
        if(btn[code]){
            data &= (~(0x0001 << code));                   //出力 OFF
            object->setProperty(WSNbackColor,"#000080");
            btn[code] = 0;
        }
        else{
            data |= (0x0001 << code);                      //出力 ON
            object->setProperty(WSNbackColor,"#0000FF");
            btn[code] = 1;
        }
        len = write(out_fd, &data, 1);                     //汎用出力更新
    }
}
static WSCfunctionRegister  op("Btn_Click",(void*)Btn_Click);
```

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_DIO」に、汎用入出力を使ったサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。ソースコードをリスト 4-1-3-4 に示します。

まず、「open」関数で「/dev/genout」と「/dev/genin」をリードライトモードで開きます。汎用入出力には設定すべきモードは存在しない為、これでリード／ライトすることができます。

汎用入力デバイスを読み出すとき、1Byte 読み出すことができます。汎用入力の「read」関数は遅延せずに、汎用入力の ON/OFF 状態を即時に返します。サンプルプログラムのソースコードでは、前回値と比較して変化があったときのみコールバック関数を実行しています。

このサンプルプログラムを実行することで、汎用出力を順番に ON していき、現在の汎用入力の値をコンソール上に表示して終了します。

リスト 4-1-3-4. 汎用入出力 ON/OFF のソースコード (main.c)

```
/**
    汎用入出力制御サンプルプログラムのソースコード
**/
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

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

```
    int             err_no;
    int             outfd;
    int             infd;
    unsigned char   outdata=0x01;
    unsigned char   indata=0x00;
    ssize_t         len;

    /* 汎用出力デバイスのオープン */
    outfd = open("/dev/genout", O_RDWR);        /* 汎用出力ﾃﾞﾊﾞｲｽｵｰﾌﾟﾝ */
    if (outfd == -1){                           /* エラー処理 */
        err_no = errno;
        fprintf(stderr, "out device open: %s¥n", strerror(err_no));
        return(-1);
    }

    /* 汎用入力デバイスのオープン */
    infd = open("/dev/genin", O_RDONLY);        /* 汎用入力ﾃﾞﾊﾞｲｽｵｰﾌﾟﾝ */
    if (infd == -1){                            /* エラー処理 */
        close(infd);
        err_no = errno;
        fprintf(stderr, "out device open: %s¥n", strerror(err_no));
        return (-1);
    }

    /* 汎用出力
        汎用出力の 4 点を 1 点ずつ出力します。
    */
    for (i=0; i<4; i++){
        len = write(outfd, &outdata, 1);        /* 汎用出力更新 */
        if (len == -1){
            err_no = errno;
            fprintf(stderr, "out device write: %s¥n", strerror(err_no));
            /* 汎用出力デバイスのｸﾛｰｽﾞ */
            close(outfd);
            return (-1);
        }
        outdata <<= 1;
        usleep(500 * 1000L);
    }
    outdata = 0x00;
    len = write(outfd, &outdata, 1);            /* 汎用出力すべて OFF */
    if (len == -1){
        err_no = errno;
        fprintf(stderr, "out device write: %s¥n", strerror(err_no));
        /* 汎用出力デバイスのｸﾛｰｽﾞ */
        close(outfd);
        return (-1);
    }

    /* 汎用入力
        汎用入力の 6 点ずつ出力を行います。
```

```
    */
    len = read(infd, &indata, 1);              /* 汎用入力の値を取得 */
    if (len == -1){
        err_no = errno;
        fprintf(stderr, "in device read: %s¥n", strerror(err_no));
        /* 汎用入出力デバイスのｸﾛｰｽﾞ */
        close(outfd);
        close(infd);
        return (-1);
    }
    else if (len == 0){
        fprintf(stderr, "in device read: len zero¥n");
        /* 汎用入出力デバイスのｸﾛｰｽﾞ */
        close(outfd);
        close(infd);
        return (-1);
    }
    else{
        indata &= 0x3F;
        /* IN0 状態 */
        if (indata & 0x01) fprintf(stdout, "IN0: ON¥n");
        else               fprintf(stdout, "IN0: OFF¥n");
        /* IN1 状態 */
        if (indata & 0x02) fprintf(stdout, "IN1: ON¥n");
        else               fprintf(stdout, "IN1: OFF¥n");
        /* IN2 状態 */
        if (indata & 0x04) fprintf(stdout, "IN2: ON¥n");
        else               fprintf(stdout, "IN2: OFF¥n");
        /* IN3 状態 */
        if (indata & 0x08) fprintf(stdout, "IN3: ON¥n");
        else               fprintf(stdout, "IN3: OFF¥n");
        /* IN4 状態 */
        if (indata & 0x10) fprintf(stdout, "IN4: ON¥n");
        else               fprintf(stdout, "IN4: OFF¥n");
        /* IN5 状態 */
        if (indata & 0x20) fprintf(stdout, "IN5: ON¥n");
        else               fprintf(stdout, "IN5: OFF¥n");
    }

    /* 汎用入出力デバイスのｸﾛｰｽﾞ */
    close(outfd);
    close(infd);

    return(0);
}
```

４－２　シリアルポート

４－２－１　シリアルポートについて

EC4A シリーズは、RS232C、RS422、RS485 を切り替えることができるシリアルポートを 2 つ持っており、それぞれをユーザアプリケーションで使用できます。

ポートごとにデバイスファイルが違いますので、表 4-2-1-1 にシリアルタイプ別のデバイスファイル名について示します。

表 4-2-1-1. シリアルタイプ別のデバイスファイル名

ポート番号	デバイスファイル	用途
1	/dev/ttyS4	汎用のシリアルポート 1
2	/dev/ttyS5	汎用のシリアルポート 2

アプリケーションでシリアルポートを使用するには、RS232C、RS422、RS485 のシリアルタイプを切り替えて、それぞれのデバイスファイルをオープンし、Read／Write することで制御します。

シリアルポートのタイプを切り替えるにはシリアルポート切り替えドライバを使用します。シリアルポートの切り替えは、下記に示す 2 種類の方法で行うことができます。詳細は『4-2-2　シリアルポートデバイスドライバについて』を参照してください。

1. デバイスドライバを使用する方法
2. シリアル切り替えコマンドを使用する方法

※注：同一ポートで RS232C、RS422、RS485 同時の使用はできません。
※注：SIO ポート設定スイッチも選択したタイプに設定する必要があります。

４－２－２　シリアルポートデバイスドライバについて

●シリアルポートデバイスドライバについて

Linux では「termios」と呼ばれる、非同期通信ポートを制御する為の汎用ターミナルインターフェースがあります。このインターフェースを使用することで、シリアルポートのボーレートや、CTS/RTS の有効無効等、さまざまな設定が可能となります。「termios」についての詳細はインターネットや書籍等を参照してください。

●RS232C/RS422/RS485 切り替え用デバイスドライバ

各ポートは通信方法制御デバイス（/dev/scictl）を用いて RS232C/RS422/RS485 の切替えを行うことができます。表 4-2-2-1 にデバイスの詳細を示します。

表 4-2-2-1. シリアルポート通信方法制御デバイスリファレンス

SCICTL

名前

シリアルポートの通信方法を制御します。

ヘッダ

```
#include "scictl_ioctl.h"
```

説明

シリアルポートのRS232C/RS422/RS485の通信方法の設定と取得を行うことができます。

OPEN

デバイスファイル（/dev/scictl）を**open**関数でオープンします。

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

IOCTL

通信方法を制御するには**ioctl**関数を用います。

```
error = ioctl(fd, ioctl_type, &conf);
```

- ioctl_type
  - ASD_SCICTL_IOCSCONF
    通信方法を設定します。
  - ASD_SCICTL_IOCGCONF
    現在の通信方法を取得します。
- conf
  通信方法設定データです。以下の構造体になります。

```
struct scictl_conf {
    unsigned short ch;
    unsigned short type;
    unsigned short timer;
};
```

ch	ポート番号 1 : ttyS4 2 : ttyS5
type	ポート通信方法 0 : RS232C 1 : RS422 2 : RS485
timer	RS485用のTXディセーブルタイマ【ミリ秒】 送信完了から、この設定時間の間、再送信がなければTXがディセーブルされます。

●RS232C/RS422/RS485 切り替えコマンド

RS232C/RS422/RS485 切り替えは、Algo Smart Panel 付属のコマンドでも行うことができます。表 4-2-2-2 に詳細を示します。

表 4-2-2-2. シリアルポート通信切り替えコマンドリファレンス

SCICTL_CONF

名前

シリアルポート通信切換えコマンド

書式

sctl_conf　ch type timer

説明

シリアルポートのRS232C/RS422/RS485の通信方法の設定を行います。

引数：

ch	ポート番号 1 : ttyS4 2 : ttyS5
type	ポート通信方法 0 : RS232C 1 : RS422 2 : RS485
timer	RS485用のTXディセーブルタイマ【ミリ秒】 送信完了から、この設定時間の間、再送信がなければ TXがディセーブルされます。

ポート 1 を RS422 に設定する場合は、以下の様にコマンドを実行します。

```
# sctl_conf 1 1 0
ch=1, type=1, timer=0
```

４－２－３　シリアルポートサンプルプログラム

●WideStudio 用サンプルプログラム

「/usr/local/tools-0010/samples/sampleWideStudio/sample_Sio」に、シリアルポートを使ったサンプルプログラムが入っています。

図 4-2-3-1. シリアルポートデバイス制御サンプル画面

このサンプルプログラムは、スレッド内でシリアルポート 0 からの受信待ちを行い、受信した文字をそのまま送信し、同時に中央のテキストフィールドに受信文字列を表示します。

シリアルポートデバイスの切り替えと、オープン、スレッドの生成を記したソースコードをリスト 4-2-3-1 に示します。

シリアルポートデバイスをオープンする前に、シリアルポート切り替えデバイスを「open」関数でオープンし、RS232C として使用するように設定しています。
「open」関数で通信を行いたいポートのデバイスをオープンします。(サンプルではシリアルポート 1 デバイスファイル名「/dev/ttyS4」をオープンしています。)
「O_NOCTTY」は制御端末として使用しないモードでオープンします。「tcgetattr」関数で現状の通信設定(「termios」構造体)を取得します。「termios」構造体のメンバの値を変更することで通信設定を変更します。「tcsetattr」関数で変更した通信設定を反映します。これで通信できる状態になります。
「termios」構造体、ならびにシリアルデバイスの使用方法等の詳細については、インターネットや書籍を参照してください。
このサンプルプログラムの通信設定は 8bit 長、パリティ無し、ストップビット 1bit、ボーレート 38400bps となっています。

リスト 4-2-3-1. シリアルポートデバイスの切り替えとオープンとスレッドの生成 (Main_Init.cpp)

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/types.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "scictl_ioctl.h"

#include "newwin000.h"
#include "ComThread.h"
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
void Main_Init(WSCbase* object){
    struct termios tio;
    int stat;
    int desc;
    int ret;
    struct scictl_conf  conf;

    /*
     * ポート切り替えデバイスをオープン
     *
     */
    desc = open("/dev/scictl", O_RDWR);
    if (desc == -1){
        Status_Bar->setProperty(WSNlabelString,"scictl Open Error");
        return;
    }
```

```
/*
 * ttyS1 を RS232C に変更
 *
 *   conf.ch
 *       ポート番号  1: ttyS4
 *                   2: ttyS5
 *
 *   conf.type
 *       ポートタイプ  0: RS232C
 *                     1: RS422
 *                     2: RS485
 *
 *   conf.timer
 *       RS485 用 TX ディセーブルタイマ[マイクロ秒]
 *       送信完了から設定時間の間、再送信がなければ
 *       TX がディセーブルされます。
 */
conf.ch = 1;
conf.type = 0;
conf.timer = 1000;
ret = ioctl(desc, ASP_SCICTL_IOCSCONF, &conf);
if (ret == -1){
    Status_Bar->setProperty(WSNlabelString,"scictl ioctl Error");
    close(desc);
    return;
}
close(desc);

    /*シリアルポートオープン*/
comm_fd = open("/dev/ttyS4", O_RDWR | O_NOCTTY);        //シリアルポートオープン
if(comm_fd < 0){
    Status_Bar->setProperty(WSNlabelString,"ttyS1 Open Error");
    return;
}

stat = tcgetattr(comm_fd, &tio);                          //現在の通信設定を待避
if(stat < 0){
    Status_Bar->setProperty(WSNlabelString,"Terminal Attribute Get Error");
    close(comm_fd);
    return;
}

//通信設定　(データ長 8bit ストップビット 1bit パリティ無し 制御線無視)
tio.c_cflag &= ~(CSIZE | CSTOPB | PARENB | PARODD | HUPCL);
tio.c_cflag |= CS8 | CLOCAL | CREAD;

//通信設定　(フレームエラー、パリティエラーなし)
tio.c_iflag = IGNPAR;
tio.c_oflag = 0;
tio.c_lflag = 0;
```

```
    tio.c_cc[VINTR] = 0;
    tio.c_cc[VQUIT] = 0;
    tio.c_cc[VERASE] = 0;
    tio.c_cc[VKILL] = 0;
    tio.c_cc[VEOF] = 0;
    tio.c_cc[VTIME] = 50;                          //キャラクタ間タイムアウト時間 50ms
    tio.c_cc[VMIN] = 1;                            //1 文字取得するまでﾌﾞﾛｯｸ
    tio.c_cc[VSWTC] = 0;
    tio.c_cc[VSTART] = 0;
    tio.c_cc[VSTOP] = 0;
    tio.c_cc[VSUSP] = 0;
    tio.c_cc[VEOL] = 0;
    tio.c_cc[VREPRINT] = 0;
    tio.c_cc[VDISCARD] = 0;
    tio.c_cc[VWERASE] = 0;
    tio.c_cc[VLNEXT] = 0;
    tio.c_cc[VEOL2] = 0;

    //通信設定　（ボーレート 38400）
    cfsetospeed(&tio, B38400);
    cfsetispeed(&tio, B38400);

    stat=tcsetattr(comm_fd, TCSAFLUSH, &tio);      //変更した通信設定の反映
    if(stat < 0){
        Status_Bar->setProperty(WSNlabelString,"Terminal Attribute Set Error");
        close(comm_fd);
        return;
    }

        /*スレッドの生成*/
    comctrl_thr = 0;
    comctrl_thr = WSDthread::getNewInstance();             //スレッドインスタンス取得
    comctrl_thr->setFunction(Comctrl_Thread);              //スレッド本体関数を設定
    comctrl_thr->setCallbackFunction(Com_callback_func);   //コールバック関数を設定
    comctrl_thr->createThread((void*)0);                   //スレッドを生成
}
static WSCfunctionRegister  op("Main_Init",(void*)Main_Init);
```

リスト 4-2-3-2 にシリアルポートから 1 バイト受信して、そのまま送信するスレッドのソースコードを示します。

「read」関数で 1 バイト受信するまで待ちます。1 バイト受信されたら、「write」関数を使用して 1 バイト送信し、同時にテキストフィールドに受信した文字を表示します。

リスト 4-2-3-2. シリアルポートデバイスからの受送信のソースコード（ComThread.cpp）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
```

```
#include <termios.h>
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <math.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "newwin000.h"
#include "ComThread.h"

int comm_fd;
WSDthread* comctrl_thr;

void *Comctrl_Thread(WSDthread* obj, void *arg)
{
    int len;
    char data;
    int i;
    i = 0;
    for(;;){
       len = read(comm_fd, &data, 1);              /*1 バイト受信*/
       if(len==1){
           write(comm_fd, &data, 1);               /*1 バイト送信*/
           obj->execCallback((void*)data);
       }
    }
    return(NULL);
}
/*スレッドから通知され、メインスレッドで実行されるコールバック関数*/
void Com_callback_func(WSDthread *, void *val)
{
char data[2];

    data[0] = (char)(unsigned long)val;
    data[1] = 0;
    newtext_001->addString(data);
}
```

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_SerialPortChange」に、シリアルポート切替えを行うサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。ソースコードをリスト 4-2-3-3 に示します。

このサンプルプログラムでは、シリアルポート 1 を RS485 に設定しています。

リスト 4-2-3-3. シリアルポート切替えのソースコード (main.c)

```
/**
    シリアルポート RS422/RS485/RS232C 切り替え制御方法サンプルプログラムのソースコード
**/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>
#include "scictl_ioctl.h"

int main(void)
{
    int                 err_no;
    int                 ret;
    int                 desc;
    struct scictl_conf  conf;
    /*
     * ポート切り替えデバイスをオープン
     *
     */
    desc = open("/dev/scictl", O_RDWR);
    if (desc == -1){
        err_no = errno;
        fprintf(stderr, "scictl open: %s¥n", strerror(err_no));
        return (-1);
    }

    /*
     * ttyS4 を RS485 に変更
     *
     *   conf.ch
     *       ポート番号  1: ttyS4
     *                   2: ttyS5
     *   conf.type
     *       ポートタイプ  0: RS232C
     *                     1: RS422
     *                     2: RS485
     *
     *   conf.timer
     *       RS485 用 TX ディセーブルタイマ[ミリ秒]
```

```
     *       送信完了から設定時間の間、再送信がなければ
     *       TX がディセーブルされます。
     */
    conf.ch = 1;
    conf.type = 2;
    conf.timer = 1000;
    ret = ioctl(desc, ASP_SCICTL_IOCSCONF, &conf);
    if (ret == -1){
        err_no = errno;
        fprintf(stderr, "ioctl failed: %s¥n", strerror(err_no));
        close(desc);
        return (-1);
    }

    close(desc);
    return 0;
}
```

「/usr/local/tools-0010/samples/sampleConsole/sample_Serial」に、シリアルポートで送受信を行うサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。サンプルプログラムのソースコードをリスト 4-2-3-4 に示します。

シリアルポート 1「/dev/ttyS4」を「open」関数でオープンし、「tcsetattr」関数で通信設定を行っています。通信設定は 8bit 長、パリティ無し、ストップビット 1bit、ボーレート 38400bps と設定しています。「read」関数で 100 バイト受信されるまで待ち、コンソール上に受信した文字列を表示し、同時に受信した文字列を「write」関数で送信しています。

リスト 4-2-3-4. シリアルポート送受信を行うソースコード（main.c）

```
/**
    シリアルポート制御サンプルソース
**/

#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char            rbuf[100];
    int             res;
    int             err_no;
    int             comm_fd;
    int             i;
    ssize_t         size;
    struct termios  tio;
```

```
/* シリアルデバイスオープン */
comm_fd = open("/dev/ttyS4", O_RDWR | O_NOCTTY);
if (comm_fd == -1){ /* エラー処理 */
    err_no = errno;
    fprintf(stderr, "ttyS4 open: %s¥n", strerror(err_no));
    return (-1);
}
/* 現在の通信設定を待避 */
res = tcgetattr(comm_fd, &tio);
if(res == -1){
    err_no = errno;
    fprintf(stderr, "ttyS4 tcgetattr_1: %s¥n", strerror(err_no));
    close(comm_fd);
    return (-1);
}

/* 通信設定 (データ長 8bit ストップビット 1bit パリティ無し 制御線無視) */
tio.c_cflag &= ~(CSIZE | CSTOPB | PARENB | PARODD | HUPCL);
tio.c_cflag |= CS8 | CLOCAL | CREAD;

/* 通信設定 (フレームエラー、パリティエラーなし) */
tio.c_iflag = IGNPAR;
tio.c_oflag = 0;
tio.c_lflag = 0;

tio.c_cc[VINTR] = 0;
tio.c_cc[VQUIT] = 0;
tio.c_cc[VERASE] = 0;
tio.c_cc[VKILL] = 0;
tio.c_cc[VEOF] = 0;
tio.c_cc[VTIME] = 250;  /* キャラクタ間タイムアウト時間 250 */
tio.c_cc[VMIN] = 1;     /* 1 文字取得するまでブロック */
tio.c_cc[VSWTC] = 0;
tio.c_cc[VSTART] = 0;
tio.c_cc[VSTOP] = 0;
tio.c_cc[VSUSP] = 0;
tio.c_cc[VEOL] = 0;
tio.c_cc[VREPRINT] = 0;
tio.c_cc[VDISCARD] = 0;
tio.c_cc[VWERASE] = 0;
tio.c_cc[VLNEXT] = 0;
tio.c_cc[VEOL2] = 0;

/* 通信設定 (ボーレート 38400) */
cfsetospeed(&tio, B38400);
cfsetispeed(&tio, B38400);

/* 通信設定変更を反映 */
res = tcsetattr(comm_fd, TCSAFLUSH, &tio);
if (res == -1){
    err_no = errno;
```

```
        fprintf(stderr, "ttyS4 tcgetattr_2: %s¥n", strerror(err_no));
        close(comm_fd);
        return (-1);
    }

    /* シリアルデータ受信処理 */
    while (1){
        memset(rbuf, '¥0', 100);
        /* 100 バイト単位で受信 */
        size = read(comm_fd, &rbuf[0], 100);
        if (size == -1){
            err_no = errno;
            fprintf(stderr, "ttyS4 read: %s¥n", strerror(err_no));
            break;
        }
        else if ( size > 0){
            /* 受信データを 16 進数文字に変換し標準出力 */
            fprintf(stdout, "ttyS4 read data: length[%d] data[", size);
            for (i = 0; i < size; i++)  fprintf(stdout, "0x%02X ", rbuf[i]);
            fprintf(stdout, "]¥n");

            /* 受信したデータ数を送信 */
            size = write(comm_fd, &rbuf[0], size);              /*size バイト送信*/
            if (size == -1){
                err_no = errno;
                fprintf(stderr, "ttyS4 write: %s¥n", strerror(err_no));
                break;
            }
        }
        usleep(10*1000L);
    }
    return(0);
}
```

４－３　ネットワークポート

４－３－１　ネットワークポートについて

ネットワーク通信ではソケットと呼ばれる概念で通信します。ソケットには接続を待つサーバと、サーバに接続にいくクライアントがあります。サーバプログラムがまず起動され、接続を待ちます。次にクライアントプログラムを起動してサーバに接続にいきます。これでネットワーク通信が確立します。

４－３－２　ネットワークソケット用システムコールについて

表 4-3-2-1 にサーバ側のソケットシステムコール、表 4-3-2-2 にクライアント側ソケットシステムコールを示します。また、表 4-3-2-3 にサーバ、クライアント共通のソケット通信用システムコールを示します。

表 4-3-2-1. サーバ側ソケットシステムコール

関数名	説明
socket	ソケットを作成し、対応するファイルディスクリプタを返します。
bind	待ちポート番号を指定します。
listen	カーネルにサーバソケットであることを伝えます。
accept	クライアントが接続してくるまで待ちます。通信が確立したら、接続済みのファイルディスクリプタを返します。

表 4-3-2-2. クライアント側ソケットシステムコール

関数名	説明
socket	ソケットを作成し、対応するファイルディスクリプタを返します。
connect	指定された IP アドレスとポート番号のサーバに接続にいきます。

表 4-3-2-3. ソケット通信用システムコール

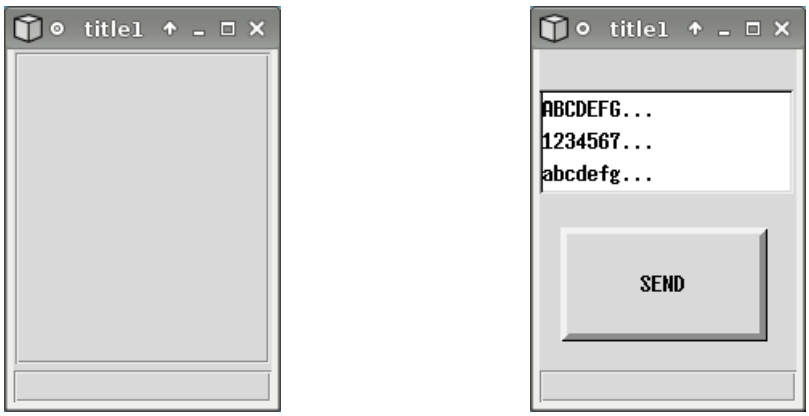
関数名	説明
recv	ソケットからデータを受信します。
send	ソケットからデータを送信します。

それぞれのシステムコール関数の詳細については、書籍やインターネットを参照してください。

これらのシステムコールを使用して、サーバプログラムとクライアントプログラムを作成することができ、ネットワークを利用して離れた場所にある機器と通信を行うことができます。

４－３－３　ネットワークサンプルプログラム

●WideStudio 用サンプルプログラム
「/usr/local/tools-0010/samples/sampleWideStudio/sample_Lan」に、ネットワーク通信を使ったサンプルプログラムが入っています。



（サーバー画面）　　（クライアント画面）

図 4-3-3-1. ネットワークサンプルプログラム画面

このサンプルプログラムは、クライアント画面で選択した文字列を「SEND」ボタンをクリックすることでサーバに送り、サーバ側で、受信した文字列をテキストフィールドに表示します。クライアント側のプログラムを起動するときには、サーバプログラムが起動しているパソコンの IP アドレスを引数として入力します。ここでは、同一のパソコン上で動作させるので、ローカルループバックアドレスを指定しています。

EC4A シリーズ本体にサーバプログラムとクライアントプログラムを送り、コンソールウィンドウを起動してプログラムを実行します。図 4-3-3-1 のような画面が起動します。

```
# ./server_spl &
# ./client_spl 127.0.0.1 &
```

リスト 4-3-3-1 にサーバソケット作成を行うソースコードを示します。
接続待ちを行う、サーバソケットを作成し、実際に待ちを行う為のスレッドを作成します。各関数の引数の意味については、インターネットや書籍を参照してください。

リスト 4-3-3-1. サーバソケット生成（./server/Main_Init.cpp）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <bits/local_lim.h>

#include <WScom.h>
#include <WSCfunctionList.h>
```

```
#include <WSCbase.h>

#include "newwin000.h"
#include "SrvThread.h"

//-----------------------------------------------------------
//Function for the event procedure
//-----------------------------------------------------------
void Main_Init(WSCbase* object){
        /* ｿｹｯﾄ生成 */
    if ((srv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        Status_Bar->setProperty(WSNlabelString,"socket() failed");
        return;
    }

        /* ポート番号指定 */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family      = AF_INET;
    srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    srv_addr.sin_port        = htons(8900);                  //ポート番号指定
    if (bind(srv_sock, (struct sockaddr *) &srv_addr, sizeof(srv_addr)) < 0) {
        Status_Bar->setProperty(WSNlabelString,"bind() failed");
        close(srv_sock);
        return;
    }

        /* カーネル通知 */
    if (listen(srv_sock, 1) < 0) {
        Status_Bar->setProperty(WSNlabelString,"listen() failed");
        close(srv_sock);
        return;
    }
        /*スレッドの生成*/
    srvctrl_thr = 0;
    srvctrl_thr = WSDthread::getNewInstance();              //スレッドインスタンス取得
    srvctrl_thr->setFunction(Srvctrl_Thread);               //スレッド本体関数を設定
    srvctrl_thr->setCallbackFunction(Srv_callback_func);    //コールバック関数を設定
    srvctrl_thr->createThread((void*)0);                    //スレッドを生成
}
static WSCfunctionRegister  op("Main_Init",(void*)Main_Init);
```

リスト 4-3-3-2 に接続待ちとクライアントからのデータ受信待ちを行うスレッドのソースコードを示します。

「accept」関数でクライアントプログラムが接続してくるのを待ちます。正常に通信確立すれば、通信確立済みのソケットのファイルディスクリプタが返されます。通信確立済みのソケットのファイルディスクリプタを使用してデータの送受信を行います。送信は「send」関数を、受信は「recv」関数を使用します。

このプログラムでは、受信した文字列をテキストフィールドに表示します。

リスト 4-3-3-2. クライアント接続待ちおよびデータ受信待ちスレッド（./server/SrvThread.cpp）

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

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <bits/local_lim.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "newwin000.h"
#include "SrvThread.h"

char  tmp[BUFFER];
int flg;
int srv_sock;
int cli_sock;
struct sockaddr_in srv_addr;
struct sockaddr_in cli_addr;
WSDthread* srvctrl_thr;

void *Srvctrl_Thread(WSDthread* obj, void *arg)
{
    int len;

    for(;;){
        /* ｸﾗｲｱﾝﾄ接続待ち */
        len = sizeof(cli_addr);
        if ((cli_sock = accept(srv_sock, (struct sockaddr *) &cli_addr, (socklen_t *)&len)) < 0)
{
            continue;
        }
        flg = 0;

        for(;;){
            /* ﾃﾞｰﾀ受信待ち */
            if(flg == 0){ /*exeCallback 関数が実行されるまで処理しない*/
                len = recv(cli_sock, tmp, BUFFER, 0);
                if (len > 0){
                    flg = 1;
                    obj->execCallback((void *)len); /*正常受信完了*/
                }else{
                    close(cli_sock);                /*通信断*/
                    break;
                }
            }
            usleep(10*1000);    /*10mswait*/
        }
```

```
    }
    return(NULL);
}
/*ｽﾚｯﾄﾞから通知され,ﾒｲﾝｽﾚｯﾄﾞで実行されるｺｰﾙﾊﾞｯｸ関数*/
void Srv_callback_func(WSDthread *, void *val)
{
int len;

    len = (int)val;
    tmp[len] = 0;
    newvlab_000->setProperty(WSNlabelString, tmp);
    flg = 0;
}
```

リスト 4-3-3-3 にクライアントソケット作成を行うソースコードを示します。
「socket」関数でソケットのファイルディスクリプタを生成します。プログラム起動時に引数として、サーバの IP アドレスを指定します。指定した IP アドレスとサーバプログラムで指定したポート番号に「connect」関数で接続します。通信確立したら 0 が、エラーなら-1 が返されます。これで、通信できる状態です。

リスト 4-3-3-3. クライアントソケット生成 (./client/Main_Init.cpp)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <bits/local_lim.h>

#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
#include <WSDappDev.h>

#include "newwin000.h"

int sock;
struct sockaddr_in srv_addr;
//------------------------------------------------------------
//Function for the event procedure
//------------------------------------------------------------
void Main_Init(WSCbase* object){
    char **argv;
    char *srv_ip;

    /* プログラム起動時の引数でサーバの IP アドレスを取得 */
    if(WSGIappDev()->getArgc() < 2){
```

```
        Status_Bar->setProperty(WSNlabelString,"No IP Address");
        return;
    }
    argv = WSGIappDev()->getArgv();
    srv_ip = *(argv + 1);


    /* ｸﾗｲｱﾝﾄｿｹｯﾄ作成*/
    if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        Status_Bar->setProperty(WSNlabelString,"socket() failed");
        return;
    }

    /* ｻｰﾊﾞに接続 */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family      = AF_INET;
    srv_addr.sin_addr.s_addr = inet_addr(srv_ip);       //ターゲットサーバ IP アドレス
    srv_addr.sin_port        = htons(8900);             //ターゲットサーバポート番号
    if (connect(sock, (struct sockaddr *) &srv_addr, sizeof(srv_addr)) < 0) {
        Status_Bar->setProperty(WSNlabelString,"connect failed");
        close(sock);
        return;
    }

    /* list 初期化 */
    newlist_000->delAll();
    newlist_000->addItem("ABCDEFG...");
    newlist_000->addItem("1234567...");
    newlist_000->addItem("abcdefg...");
    newlist_000->updateList();

}
static WSCfunctionRegister  op("Main_Init",(void*)Main_Init);
```

ボタンの「ACTIVATE」プロシージャで、リストから選んだ文字列を送信します。リスト 4-3-3-4 に文字列送信のソースコードを示します。

リストボックスから送信する文字列を選択し、ボタンを押します。「send」関数で該当する文字列を送信します。

リスト 4-3-3-4. クライアントプログラムボタン「ACTIVATE」プロシージャ (./client/Btn_Click.cpp)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <bits/local_lim.h>
```

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>

#include "newwin000.h"

#define  BUFFER  2048

extern int sock;
extern struct sockaddr_in srv_addr;
const char dat[][21]={
    {"ABCDEFGHIJKLMNOPQRST"},
    {"12345678901234567890"},
    {"abcdefghijklmnopqrst"}
};

//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
void Btn_Click(WSCbase* object){
long d;

    d = newlist_000->getSelectedPos();
    if((d < 0)||(d > 2)) return;
    /* データ送信 */
    if (send(sock, &dat[d][0], 20, 0) != 20) {
        Status_Bar->setProperty(WSNlabelString,"send() failed");
    }
}
static WSCfunctionRegister  op("Btn_Click",(void*)Btn_Click);
```

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_TcpIp」に、ネットワーク通信を行うサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。

サーバプログラムとクライアントプログラムの起動手順は以下の通りです。

```
# ./sampleTCPIP_Server &
# ./sampleTCPIP_Client -i 127.0.0.1 &
```

この場合、1 台の EC4A シリーズ上でサーバとクライアントのプログラムが動作し、お互いに通信を行います。

ソースコードをリスト 4-3-3-5 に示します。

サーバソケットを作成し、「accept」関数でクライアントプログラムが接続してくるのを待ちます。正常に通信確立すれば、通信確立済みのソケットのファイルディスクリプタが返されます。通信確立済みのファイルディスクリプタを使用してデータの送受信を行います。送信は「send」関数を、受信は「recv」関数を使用します。

このプログラムでは、受信した文字列を画面に表示します。

リスト 4-3-3-5. サーバソケットのソースコード (main.c)

```
/**
    TCP/IP サーバソケット　サンプルプログラムのソースコード
**/

#include <arpa/inet.h>
```

```
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <unistd.h>

int main(void)
{
    char                rcv_buf[11];
    int                 i;
    int                 srv_sock;
    int                 len;
    int                 res;
    int                 err_no;
    int                 sock;
    struct sockaddr_in  cli_addr;
    struct sockaddr_in  srv_addr;



    /* データ受信処理 */
    while (1){

        /* サーバソケット作成 */
        srv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (srv_sock == -1) {
            err_no = errno;
            fprintf(stderr, "socket failed: %s¥n", strerror(err_no));
            return (-1);
        }

        /* ポート番号指定 */
        memset(&srv_addr, 0, sizeof(srv_addr));

        srv_addr.sin_family      = AF_INET;
        srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        srv_addr.sin_port        = htons(8900);                 //ポート番号指定

        /* ソケットに名前をつける */
        res = bind(srv_sock, (struct sockaddr *) &srv_addr, sizeof(srv_addr));
        if (res == -1) {
            err_no = errno;
            fprintf(stderr, "bind failed: %s¥n", strerror(err_no));
            close(srv_sock);
            return (-1);
        }

        /* カーネル通知 */
```

```
        res = listen(srv_sock, 1);
        if (res == -1) {
            err_no = errno;
            fprintf(stderr, "listen failed: %s¥n", strerror(err_no));
            close(srv_sock);
            return (-1);
        }

        /* クライアント接続待ち */
        len = sizeof(cli_addr);
        srv_sock = accept(srv_sock, (struct sockaddr *)&cli_addr, (socklen_t *)&len);
        if (srv_sock < 0) continue;

        /* クライアントソケット接続待 */
        while (1){
            memset(rcv_buf, '¥0', 11);
            len = recv(srv_sock, &rcv_buf[0], 10, 0);
            if (len <= 0){
                err_no = errno;
                fprintf(stderr, "recv failed: %s¥n", strerror(err_no));
                close(srv_sock);
                break;
            }
            else {
                fprintf(stdout, "recv data: length[%d] [", len);
                for (i = 0; i < len; i++) fprintf(stdout, "0x%02X ", rcv_buf[i]);
                fprintf(stdout, "]¥n");
            }
            usleep(10 * 1000L);
        }
        usleep(10 * 1000L);
    }

    close(sock);
    return (0);
}
```

クライアント側のプログラムは、表 4-3-2-2 に書かれているシステムコールを実行して、接続待ちしているサーバに接続します。

リスト 4-3-3-6 にクライアントソケット作成を行うソースコードを示します。

「socket」関数でクライアント用のソケットのファイルディスクリプタを生成します。プログラム起動時に引数として、サーバの IP アドレスを指定します。指定した IP アドレスとサーバプログラムで指定したポート番号に「connect」関数で接続します。通信が確立したら 0 が返り、通信できる状態になります。エラーなら-1 が返されます。

リスト 4-3-3-6. クライアントソケット (main.c)

```
/**
    TCP/IP クライアントソケット　サンプルプログラムのソースコード
**/

#include <arpa/inet.h>
```

```
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char*               srv_ip=NULL;
    char                snd_buf[] = "ABCDEFGHIJKLMNOPQRST";
    int                 c;
    int                 res;
    int                 err_no;
    int                 sock;
    struct sockaddr_in  srv_addr;

    /*
     * 起動引数取得
     *   -i : IP アドレスを指定します。
     */
    while ((c = getopt(argc, argv, "i:")) != -1){
        switch(c){
            case 'i':
                srv_ip = optarg;
                break;
            default:
                fprintf(stdout, "argument error¥n");
                fprintf(stdout, "  -i : Ip Address : ex) 192.168.0.1¥n");
                return (-1);
        }
    }
    if (srv_ip == NULL){
        fprintf(stdout, "argument error Ip Address : ex) -i 192.168.0.1¥n");
        return (-1);
    }

    /* クライアントソケット作成 */
    if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        fprintf(stderr, "socket failed¥n");
        return (-1);
    }

    /* サーバに接続 */
    memset(&srv_addr, '¥0', sizeof(srv_addr));

    srv_addr.sin_family      = AF_INET;
    srv_addr.sin_addr.s_addr = inet_addr(srv_ip);       /* ﾀｰｹﾞｯﾄｻｰﾊﾞ IP ｱﾄﾞﾚｽ */
    srv_addr.sin_port        = htons(8900);             /* ﾀｰｹﾞｯﾄｻｰﾊﾞﾎﾟｰﾄ番号 8900port */
```

```
    res = connect(sock, (struct sockaddr *)&srv_addr, sizeof(srv_addr));
    if (res == -1){
        err_no = errno;
        fprintf(stderr, "connect failed: %s¥n", strerror(err_no));
        close(sock);
        return (-1);
    }

    /* サーバにデータを送信 */
    res = send(sock, &snd_buf, strlen(snd_buf), 0);
    if (res == -1){
        err_no = errno;
        fprintf(stderr, "send failed: %s¥n", strerror(err_no));
        close(sock);
        return (-1);
    }
    else if (res < strlen(snd_buf)){
        fprintf(stderr, "send failed: send size(%d)¥n", res);
        close(sock);
        return (-1);
    }

    /* データを送信後待機 */
    while (1) usleep(100 * 1000L);

    close(sock);
    return (0);
}
```

４－４　RAS 機能

RAS 機能として以下の様な機能を実装しています。

- 汎用入力 IN0 リセット
- 汎用入力 IN1 割り込み
- ハードウェア・ウォッチドッグタイマ
- バックアップ RAM

次項より各機能についての説明を行います。

４－４－１　汎用入力 IN0 リセットについて

汎用入力 IN0 リセットは、汎用入力の IN0 が ON した場合に本体をリセットする機能です。
デバイスドライバからこの機能の有効・無効を切り替えることができます。

４－４－２　汎用入力 IN0 リセットデバイスドライバについて

汎用入力 IN0 リセットデバイスファイル(/dev/rasin)にアクセスする事により設定状態を読み書きできます。
表 4-4-2-1 に汎用入出力のリファレンスを示します。

表 4-4-2-1. 汎用入力 IN0 デバイスリファレンス

RASIN

名前

汎用入力IN0リセット機能の制御を行います。

インクルード

```
#include "rasin.h"
```

説明

汎用入力IN0リセット機能の有効・無効設定及び設定値の取得を行います。
本設定は、本体再起動後に有効となります。

OPEN

汎用入力IN0リセットデバイス(/dev/rasin)を**open**関数でオープンします。

```
rasinfd  = open("/dev/rasin", O_RDWR);
```

IOCTL

IN0リセットを制御するには**ioctl**関数を使用します。

```
error = ioctl(fd, ioctl_type, &conf);
```

- ioctl_type
  RASIN_IOCSINORST
  　有効・無効を設定します。
  RASIN_IOCGINORST
  　現在の設定値を取得します。

- conf
  0　無効
  1　有効

４－４－３　汎用入力 IN0 リセットサンプル

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_Reset」に、汎用入力 IN0 リセットを使用するサンプルプログラムが入っています。このサンプルプログラムはコンソールアプリケーションとして作成されています。コンソール上で動作させることができます。リスト 4-4-3-1 にソースコードを示します。

リスト 4-4-3-1. 汎用入力 IN0 リセット (main.c)

```
/**
    汎用入力 IN0 リセット制御方法サンプルプログラムのソースコード
**/
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <unistd.h>

#include "rasin.h"

int main(void)
{
    int desc;
    int len;
    int onoff;
    int err_no;

    /*
     * RAS/Input デバイスのオープン
     */
    desc = open("/dev/rasin", O_RDWR);
    if (desc == -1){
        err_no = errno;
        fprintf(stderr, "rasin open: %s¥n", strerror(err_no));
        return (-1);
    }

    /*
     * 現在の設定を取得
     *
     * onoff: 1    有効
     *      : 0    無効
     */
    len = ioctl(desc, RASIN_IOCGINORST, &onoff);
    if (len == -1) {
        err_no = errno;
        fprintf(stderr, "ioctl get INORST faild: %s¥n", strerror(err_no));
        close(desc);
        return (-1);
```

```
    }
    else {
        fprintf(stdout, "ioctl get INORST success: %d¥n", onoff);
    }

    /*
     * INO リセットを有効にする
     *
     * onoff: 1    有効
     *      : 0    無効
     */
    onoff = 1;
    len = ioctl(desc, RASIN_IOCSINORST, &onoff);
    if (len == -1) {
        err_no = errno;
        fprintf(stderr, "ioctl set INORST faild: %s¥n", strerror(err_no));
        close(desc);
        return (-1);
    }
    else {
        fprintf(stdout, "ioctl set INORST success: %d¥n", onoff);
    }
    close(desc);
    return 0;
}
```

４－４－４　汎用入力 IN1 割込みについて

汎用入力 IN1 割込みは、汎用入力の IN1 が ON した場合に割込みを発生させる機能です。デバイスドライバからこの機能の有効・無効を切り替えることができます。ユーザーアプリケーションでは SIGIO シグナルとして通知を受けることができます。

４－４－５　汎用入力 IN1 割込みデバイスドライバについて

汎用入力 IN1 割込みデバイスファイル（/dev/rasin）にアクセスする事により設定状態を読み書きできます。
表 4-4-5-1 に汎用入出力のリファレンスを示します。

表 4-4-5-1. 汎用入力 IN1 割込みデバイスリファレンス

RASIN

名前

汎用入力IN1割込み機能の制御を行います。

インクルード

```
#include "rasin.h"
```

説明

汎用入力IN1割込み機能の有効・無効設定及び設定値の取得を行います。
本設定は、本体再起動後に有効となります。

OPEN

汎用入力IN1割込みデバイス(/dev/rasin)を**open**関数でオープンします。

```
rasinfd  = open("/dev/rasin", O_RDWR);
```

IOCTL

IN1割込みを制御するには**ioctl**関数を使用します。

```
error = ioctl(fd, ioctl_type, &conf);
```

- ioctl_type
  - RASIN_IOCSIN1INT
    有効・無効を設定します。
  - RASIN_IOCGIN1INT
    現在の設定値を取得します。

- conf
  - 0　無効
  - 1　有効

４－４－６　汎用入力 IN1 割込みサンプル

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_Interrupt」に、汎用入力 IN1 割込みを使用するサンプルプログラムが入っています。このサンプルプログラムはコンソールアプリケーションとして作成されています。コンソール上で動作させることができます。リスト 4-4-6-1 にソースコードを示します。

リスト 4-4-6-1. 汎用入力 IN1 割込み（main.c）

```
/**
    汎用入力 IN1 割込み制御サンプルプログラムのソースコード
**/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

#include "rasin.h"

static int  SigOnOff=0;

void sighandler(int signo)
{
    /*
     * SIGIO シグナルなら終了
     */
    if(signo == SIGIO){
        SigOnOff = 1;
    }
}

int main(void)
{
    int         err_no;
    int         desc;
    int         len;
    int         onoff;
    struct sigaction    action;

    /*
     * SIGIO シグナルのハンドラを登録
     */
    memset(&action, 0, sizeof(action));
    action.sa_handler = sighandler;
    action.sa_flags = 0;
    sigaction(SIGIO, &action, NULL);
```

```
    /*
     * RAS/Input デバイスのオープン
     */
    desc = open("/dev/rasin", O_RDWR);
    if (desc == -1){
        err_no = errno;
        fprintf(stderr, "rasin open: %s¥n", strerror(err_no));
        return (-1);
    }

    /*
     * プロセスが SIGIO を受け取れるようにする
     */
    fcntl(desc, F_SETOWN, getpid());
    fcntl(desc, F_SETFL, fcntl(desc, F_GETFL) | FASYNC);

    /*
     * 現在の設定値取得
     *
     * onoff: 1    有効
     *      : 0    無効
     */
    len = ioctl(desc, RASIN_IOCGIN1INT, &onoff);
    if (len == -1) {
        err_no = errno;
        fprintf(stderr, "ioctl get IN1INT faild: %s¥n", strerror(err_no));
        close(desc);
        return (-1);
    }
    else {
        fprintf(stdout, "ioctl get IN1INT success: %d¥n", onoff);
    }

    /*
     * IN1 割込みを有効にする
     *
     * 有効の場合、無効に変更し終了
     * onoff: 1    有効
     *      : 0    無効
     */
    if (onoff != 1) onoff = 1;
    else            onoff = 0;
    len = ioctl(desc, RASIN_IOCSIN1INT, &onoff);
    if (len == -1) {
        err_no = errno;
        fprintf(stderr, "ioctl set IN1INT faild: %s¥n", strerror(err_no));
        close(desc);
        return (-1);
    }
    else {
```

```
        fprintf(stdout, "ioctl set IN1INT success: %d¥n", onoff);
        if (onoff == 0){
            fprintf(stdout, "IN1 Interrupt off¥n");
            return (1);
        }
    }
    /*
     * シグナル (SIGIO) 受信時に標準出力に出力を行います。
     */
    while (1){
        if (SigOnOff == 1){
            printf("IN1INT receive¥n");
            break;
        }
        usleep(10*1000L);
    }
    close(desc);
    return 0;
}
```

４－４－７　ハードウェア・ウォッチドッグタイマ機能について

EC4A シリーズには、ハードウェアによるウォッチドッグタイマ機能が搭載されています。
デバイスドライバ（/dev/rashwwdt）にアクセスすることにより、この機能を使用することができます。
シャットダウン、再起動、ポップアップ通知、ログへの出力は、デーモン（HWWDTD）によって行われています。
図 4-4-7-1 に、デバイス、ドライバ、アプリケーション、設定ツール、デーモンの関係を示します。

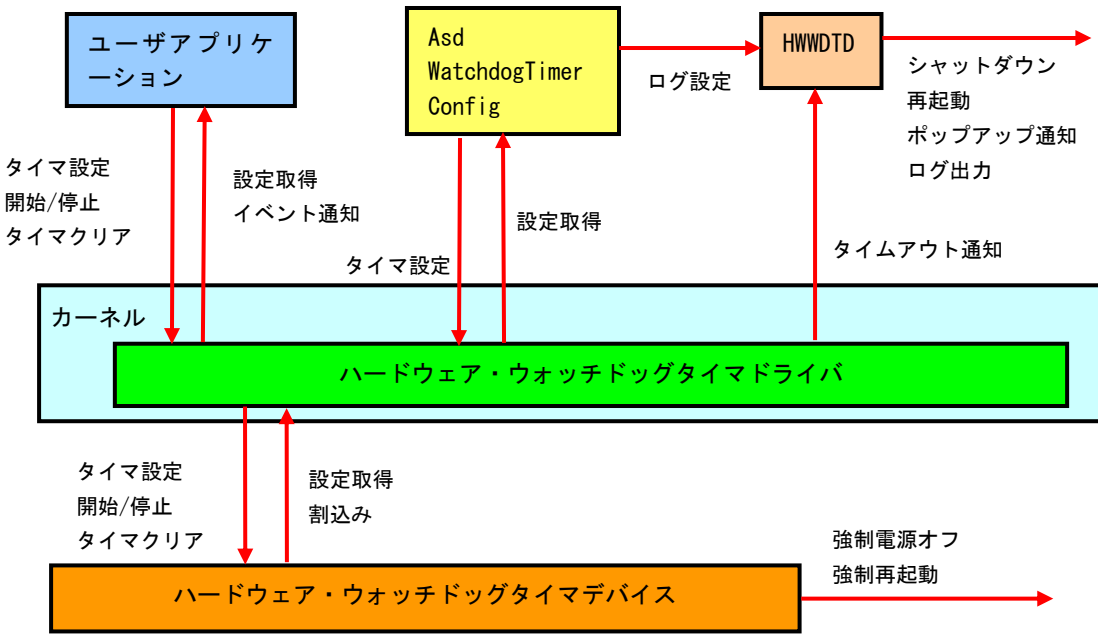


図 4-4-7-1. ハードウェア・ウォッチドッグタイマの構成

ハードウェア・ウォッチドッグタイマでは、ハード的に電源を OFF する機能（**強制電源オフ**）とリセットする機能（**強制再起動**）があります。OS が完全にフリーズした状態でも、電源 OFF やリセットを行うことが出来ます。

ハードウェア・ウォッチドッグタイマで**強制電源オフ**を設定した場合、「POWER ボタン」を長押ししたときと同じ処理になります。EC4A シリーズでは POWER ボタンを 4 秒間長押しすると、強制的に電源 OFF されます。また、Algonomix4 では、POWER ボタンを押されるとシャットダウン処理を実行します。

出荷時設定のままでは、シャットダウン処理中に電源が落ちることになり、m-SATA ディスク破損の原因になる可能性があります。
下記のコマンドを実行して、「/etc/systemd/logind.conf」を編集してください。

```
$ sudo su
パスワード:asdusr
# vi /etc/systemd/logind.conf
```

編集前

```
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.
#
# See logind.conf(5) for details
```

```
[Login]
#NAutoVTs=6
#ReserveVT=6
#KillUserProcesses=no
#KillOnlyUsers=
#KillExcludeUsers=root
Controllers=blkio cpu cpuacct cpuset devices freezer hugetlb memory perf_event net_cls net_prio
ResetControllers=
#InhibitDelayMaxSec=5
#HandlePowerKey=poweroff
#HandleSuspendKey=suspend
#HandleHibernateKey=hibernate
#HandleLidSwitch=suspend
#PowerKeyIgnoreInhibited=no
#SuspendKeyIgnoreInhibited=no
#HibernateKeyIgnoreInhibited=no
#LidSwitchIgnoreInhibited=yes
#IdleAction=ignore
#IdleActionSec=30min
```

編集後

```
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.
#
# See logind.conf(5) for details

[Login]
#NAutoVTs=6
#ReserveVT=6
#KillUserProcesses=no
#KillOnlyUsers=
#KillExcludeUsers=root
Controllers=blkio cpu cpuacct cpuset devices freezer hugetlb memory perf_event net_cls net_prio
ResetControllers=
#InhibitDelayMaxSec=5
HandlePowerKey=ignore
#HandleSuspendKey=suspend
#HandleHibernateKey=hibernate
#HandleLidSwitch=suspend
#PowerKeyIgnoreInhibited=no
#SuspendKeyIgnoreInhibited=no
#HibernateKeyIgnoreInhibited=no
#LidSwitchIgnoreInhibited=yes
#IdleAction=ignore
#IdleActionSec=30min
```

HandlePowerKey の先頭の「#」を外してコメントアウトを解除し、設定を「ignore」に変更することで、logind

に ACPI イベントを無視させます。

この編集後は、POWER ボタンを押してもシャットダウン処理は実行しません。元に戻す場合は、logind.conf ファイルを編集前の状態に戻してください。

４－４－８　ハードウェア・ウォッチドッグタイマデバイスについて

ハードウェア・ウォッチドッグタイマデバイスファイル（/dev/rashwwdt）にアクセスすることにより、ハードウェア・ウォッチドッグタイマ機能を操作します。表 4-4-8-1 に、ハードウェア・ウォッチドッグタイマの詳細を示します。

表 4-4-8-1. ハードウェア・ウォッチドッグタイマデバイスリファレンス

RASHWWDT
名前
rashwwdt
インクルード
#include "asd_rashwwdt.h"
説明
ハードウェア・ウォッチドッグタイマの開始/停止、タイムアウト時の動作の設定、 タイマ時間の設定、イベント通知待ちを行うことが出来ます。
OPEN
デバイスファイル(/dev/rashwwdt)を**open**関数でオープンします。 int fd = open("/dev/rashwwdt", O_RDWR);
READ
使用しません
WRITE
1Byte以上のデータを書込むことで、タイマカウントをクリアします。 char data = `a`; size_t len = write(fd, &data, 1);

IOCTL

ハードウェア・ウォッチドッグタイマデバイスを操作するには**ioctl**関数を使用します。

```
error = ioctl(fd, ctlcode, param);
```

● **ctlcode**

RASHWWDT_IOCTSTART

ハードウェア・ウォッチドッグタイマを開始します。
また、paramで、close時にタイマカウントを停止するか開始、継続するかを指定します。
paramが1の時は、タイマが停止していてもclose時に開始されます。
0: タイマを停止、1: タイマを継続
[param]unsigned long型のポインタ
[error] 0:正常、0以外:異常

RASHWWDT_IOCTSTOP

ハードウェア・ウォッチドッグタイマを停止します。
[param] なし (NULL)
[error] 0:正常、0以外:異常

RASHWWDT_IOCSCONFIG

ハードウェア・ウォッチドッグタイマのタイムアウト時の動作とタイマ時間を設定します。
設定を変更した場合、ハードウェア・ウォッチドッグタイマをオープンしているすべての
アプリケーションに影響を与えます。変更する場合は、他のアプリケーションが
ハードウェア・ウォッチドッグタイマをオープンしていないかどうかを確認してください。
[param] RASSWWDT_CONFIG型変数のポインタ
[error] 0:正常、0以外:異常

RASHWWDT_IOCGCONFIG

ハードウェア・ウォッチドッグタイマの設定を取得します。
[param] RASSWWDT_CONFIG型変数のポインタ
[error] 0:正常、0以外:異常

RASHWWDT_IOCQKEEPALIVE

タイマカウントをクリアします
[param] なし (NULL)
[error] 0：正常、 0以外：異常

RASHWWDT_IOCQWAITEVENT

イベント通知を待ちます。
このコントロールコードでioctlを呼び出すとタイムアウト発生、または強制解除される
まで関数からリターンしません。
[param] なし (NULL)
[error] 1:イベント発生、2:強制解除、その他:異常

RASHWWDT_IOCQWAKEUP

イベント通知待ちを強制解除します。
[param] なし (NULL)
[error] 0:正常、0以外:異常

● **param**

RASHWWDT_CONFIG構造体

ハードウェア・ウォッチドッグタイマの設定を格納します。

```
struct RASHWWDT_CONFIG
{
  unsigned long action;   // タイムアウト時の動作  0：システム停止 1：システム再起動
                                                   2：ポップアップ 3：イベント通知
                                                   4：強制電源OFF  5：強制再起動
  unsigned long time;     // タイマ時間  (time × 100【msec】)
};
```

４－４－９　ハードウェア・ウォッチドッグタイマサンプル

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_HwWdtKeepalive」に、ハードウェア・ウォッチドッグタイマドライバを使用するサンプルプログラムが入っています。リスト 4-4-9-1 にサンプルプログラムのソースコードを示します。

リスト 4-4-9-1. ハードウェア・ウォッチドッグタイマデバイス操作ソースコード（main.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include "asd_rashwwdt.h"

#define HWWDTDEV_FILENAME   "/dev/rashwwdt"

static struct RASHWWDT_CONFIG Config;
static unsigned long Sta_Type;
static volatile int fStart = 0;
static volatile int fStop = 0;
static volatile int fFinish = 0;
static pthread_t Thread_Id;

//------------------------------------------------------------------------
/*
 * キープアライブスレッド
 */
static void *TimerProc(void *arg)
{
    int wdtfd = 0;
    int status = 0;

    /*
     * ハードウェア・ウォッチドッグタイマデバイスのオープン
     */
    wdtfd = open(HWWDTDEV_FILENAME, O_RDWR);
    if(wdtfd <= 0){
        fprintf(stderr, "TimerProc: Device open failed¥n");
        exit(1);
    }
    /*
     * ハードウェア・ウォッチドッグタイマデバイスの設定
     */
    if(ioctl(wdtfd, RASHWWDT_IOCSCONFIG, &Config) != 0){
        close(wdtfd);
        fprintf(stderr, "TimerProc: Device set error¥n");
        exit(1);
```

```
    }
    /*
     * 前回終了状態の取得
     */
    status = ioctl(wdtfd, RASHWWDT_IOCGSTATE, NULL);
    if(status == RASHWWDT_NORMAL_SHUTDOWN){
        fprintf(stdout, "Last Shutdown is normal shutdown¥n");
    }
    if(status == RASHWWDT_FORCE_SHUTDOWN){
        fprintf(stdout, "Last Shutdown is force shutdown¥n");
    }

    /*
     * ハードウェア・ウォッチドッグタイマのスタート
     */
    ioctl(wdtfd, RASHWWDT_IOCTSTART, &Sta_Type);
    printf("TimerProc: Start¥n");

    fFinish = 0;
    while(1){
        if(!fStart){
            break;
        }
        /* KeepAlive */
        if(!fStop){
            ioctl(wdtfd, RASHWWDT_IOCQKEEPALIVE, NULL);
        }
        usleep(100 * 1000);
    }
    /*
     * ハードウェア・ウォッチドッグタイマ停止
     * ハードウェア・ウォッチドッグタイマ破棄
     */
    ioctl(wdtfd, RASHWWDT_IOCTSTOP, NULL);
    close(wdtfd);
    printf("TimerProc: Stop¥n");

    fFinish = 1;

    return 0;
}

//---------------------------------------------------------------------------
static int CreateThread(void)
{
    pthread_attr_t thread_attr;

    pthread_attr_init(&thread_attr);
    pthread_attr_setstacksize(&thread_attr, 0x10000);

    fStart = 1;
```

```
    if(pthread_create(&Thread_Id, &thread_attr, TimerProc, NULL) != 0){
        printf("pthread_create: error¥n");
        return -1;
    }

    return 0;
}

//--------------------------------------------------------------------------
int main(int argc,char** argv)
{
    int c;
    int action;
    int wdt;
    int sta_type;

    wdt = -1;
    action = -1;
    sta_type = -1;

    while((c = getopt(argc, argv, "t:a:s:")) != -1){
        switch(c){
            case 't':
                wdt = atoi(optarg);
                break;
            case 'a':
                action = atoi(optarg);
                break;
            case 's':
                sta_type = atoi(optarg);
                break;
        }
    }
    if((wdt <= 0) || (action < 0) || (action > RASHWWDT_RESET) || (sta_type < 0) || (sta_type
> RASHWWDT_ENDCONTINUE)){
        printf("  t:test time[ *100msec]¥n");
        printf("  a:action 0:Shutdown 1:Reboot 2:Popup 3:Event 4:PowerOff 5:Reset ¥n");
        printf("  s:sta_type 0:endstop 1:continue¥n");
        return -1;
    }

    Config.action = action;
    Config.time = wdt;
    Sta_Type = sta_type;

    CreateThread();

    while(1){
        c = fgetc(stdin);
        if(c == 'q' || c == 'Q'){
            break;
```

```
        }
        if(c == 's' || c == 'S'){
            fStop = 1;
        }
        usleep(100 * 1000);
    }
    /*
     * スレッドを停止
     */
    fStart = 0;
    /*
     * スレッド停止待ち
     */
    while(fFinish != 1){
        usleep(100 * 1000);
    }
    /*
     * スレッド破棄
     */
    pthread_cancel(Thread_Id);

    return 0;
}
```

ハードウェア・ウォッチドッグタイマのタイムアウト時の動作をイベント通知にした場合、ユーザアプリケーションでタイムアウトを取得することができます。

「/usr/local/tools-0010/samples/sampleConsole/sample_HwWdtEventWait」に、タイムアウトのイベント通知を取得するサンプルプログラムが入っています。リスト 4-4-9-2 に、サンプルプログラムのソースコードを示します。

リスト 4-4-9-2.　ハードウェア・ウォッチドッグタイマイベント取得ソースコード（main.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <stdarg.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <bits/local_lim.h>
#include "asd_rashwwdt.h"

#define HWWDTDEV_FILENAME   "/dev/rashwwdt"
#define MAX_HWWDTEVENT      5

//-------------------------------------------------------------------------
```

```
typedef struct {
    int No;
    volatile int fStart;
    volatile int fFinish;
    pthread_t ThreadID;
    int wdtfd;
} HWWDTEVENT_INFO;
//-----------------------------------------------------------------------------
/*
 * タイマイベント処理スレッド
 */
static void *TimerEventProc(void *arg)
{
    HWWDTEVENT_INFO *info = (HWWDTEVENT_INFO *)arg;
    int ret;

    printf("TimerEventProc: Timer%02d: Start¥n", info->No);

    info->fFinish = 0;
    /*
     * イベント通知待ち
     */
    while(1){
        if(!info->fStart){
            break;
        }
        ret = ioctl(info->wdtfd, RASHWWDT_IOCQWAITEVENT, NULL);
        if(ret == RASHWWDT_TIMUP){
            printf("TimerEventProc: TimeOut%02d¥n", info->No);
            break;
        }else{
            printf("TimerEventProc: WakeUp%02d¥n", info->No);
            break;
        }
    }
    info->fFinish = 1;

    printf("TimerEventProc: Timer%02d: Finish¥n", info->No);
    return 0;
}

//-----------------------------------------------------------------------------
static int CreateTimerEventInfo(int No, HWWDTEVENT_INFO *info)
{
    pthread_attr_t thread_attr;

    pthread_attr_init(&thread_attr);
    pthread_attr_setstacksize(&thread_attr, 0x10000);

    info->No = No;
    info->ThreadID = 0;
```

```
    info->fStart = 1;
    info->fFinish = 0;
    info->wdtfd = -1;

    /*
     * ハードウェア・ウォッチドックタイマデバイスのオープン
     */
    info->wdtfd = open(HWWDTDEV_FILENAME, O_RDWR);
    if(info->wdtfd < 0){
        printf("%s: open faild: err=%d¥n", HWWDTDEV_FILENAME, errno);
        return -1;
    }

    /*
     * タイマイベント処理スレッド作成
     */
    if(pthread_create(&info->ThreadID, &thread_attr, TimerEventProc, (void *)info) != 0){
        close(info->wdtfd);
        printf("pthread_create: error¥n");
        return -1;
    }

    return 0;
}

//-----------------------------------------------------------------------
static void DeleteTimer(HWWDTEVENT_INFO *info)
{
    /*
     * スレッド停止
     */
    info->fStart = 0;
    /*
     * タイマ停止
     * タイマイベント待ち解除
     * タイマ破棄
     */
    ioctl(info->wdtfd, RASHWWDT_IOCTSTOP, NULL);
    ioctl(info->wdtfd, RASHWWDT_IOCQWAKEUP, NULL);
    close(info->wdtfd);
    /*
     * スレッド停止待ち
     */
    while(!info->fFinish){
        usleep(10 * 1000);
    }

    pthread_cancel(info->ThreadID);
}

//-----------------------------------------------------------------------
```

```
int main(int argc, char** argv)
{
    int i;
    HWWDTEVENT_INFO info[MAX_HWWDTEVENT];
    int action;
    int wdt;
    int sta_type;
    char c;

    wdt = 20;
    action = RASHWWDT_REBOOT;
    sta_type = RASHWWDT_ENDSTOP;

    for(i = 0; i < MAX_HWWDTEVENT; i++){
        if(CreateTimerEventInfo(i, &info[i]) != 0){
            printf("CreatTimerEvent: NG: %d¥n", i);
            return -1;
        }
    }
    while(1){
        c = fgetc(stdin);
        if(c == 'q' || c == 'Q'){
            break;
        }
    }

    for(i = 0; i < MAX_HWWDTEVENT; i++){
        DeleteTimer(&info[i]);
    }

    return 0;
}
```

４－４－１０　バックアップ RAM 機能について

EC4A シリーズでは 4MByte のバックアップ機能付きの RAM エリアが用意されています。

端末起動時、指定されたファイルからバックアップ RAM 領域に内容が読み出され、端末シャットダウン時に指定されたファイルにバックアップ RAM 領域の内容が書き込まれます。

バックアップ機能の有効/無効、バックアップファイルの保存先は、ASD UPS Config ツールで指定できます。指定方法については、『2-3-4　ASD UPS Config について』を参照してください。

４－４－１１　バックアップ RAM 機能デバイスドライバについて

デバイスファイル（/dev/g5sram0）に対して Read/Write する事によりバックアップ RAM を読み書きできます。また、mmap をつかって、RAM をユーザプログラム内でマッピングし直すことで、直接アクセスすることも可能です。表 4-4-11-1 にデバイスの詳細を示します。

表 4-4-11-1. バックアップ RAM デバイスリファレンス

RASRAM
名前
バックアップRAMの制御を行います。
説明
バックアップRAMは、バックアップ機能付きのRAM領域です。 デバイスドライバからRAMのデータを読み書きできます。
OPEN
バックアップSRAMデバイス(/dev/g5sram0)を**open**関数でオープンします。 rasramfd = open("/dev/g5sram0", O_RDWR);
READ
read関数を用いてバックアップRAMの値を読込みます。
WRITE
write関数を用いてバックアップRAMに値を書込みます。
MMAP
mmap関数を用いてバックアップRAMを連続したメモリ領域としてマッピングし直すことができます。マッピングし直したメモリアドレスへ直接読み書きすることが可能です。
IOCTL
ioctl関数を用いてバックアップRAMの情報を読み出すことができます。 error = ioctl(fd, ctlcode, param); ● **ctlcode** **SRAM_IOCGSIZE** バックアップRAMのサイズを取得します。 [param] unsigned long型変数のポインタ [error] 0:正常、0以外:異常

４－４－１２　バックアップ RAM 制御サンプル

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_SRAM」に、バックアップ付き RAM を read/write システムコールで操作したサンプルプログラムが入っています。このサンプルプログラムはコンソールアプリケーションとして作成されています。コンソール上で動作させることができます。インクリメントデータの読み書きとデクリメントデータの読み書きを行い、データが正常に書き込まれているかを確認しています。リスト 4-4-12-1 にソースコードを示します。

リスト 4-4-12-1. バックアップ RAM 制御 Read/Write 方式（rasram_read.c）

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

#include "asd_rasram.h"

unsigned long rasram_size = 0;

unsigned char *rasram;

int main(void)
{
    int desc;
    int i;
    int len;
    unsigned char d;
    unsigned char *dp;

    desc = open("/dev/g5sram0", O_RDWR);
    if(desc < 0){
        printf("open faild: err=%d¥n", errno);
        return -1;
    }

    if(ioctl(desc, SRAM_IOCGSIZE, &rasram_size) != 0){
        close(desc);
        printf("size check faild.¥n");
        return -1;
    }
    printf("rasram size is %lx.¥n", rasram_size);
    rasram = malloc(rasram_size * sizeof(unsigned char));

    /* inc data */
    d = 1;
    dp = rasram;
```

```
    for(i = 0; i < rasram_size; i++){
        *dp++ = d++;
    }
    lseek(desc, 0, SEEK_SET);
    len = write(desc, rasram, rasram_size);
    if(len != rasram_size){
                printf("inc data write faild: err=%d¥n", errno);
                close(desc);
        free(rasram);
        return -1;
    }
    memset(rasram, 0x00, rasram_size);
    lseek(desc, 0, SEEK_SET);
    len = read(desc, rasram, rasram_size);
    if(len != rasram_size){
                printf("dec data read faild: err=%d¥n", errno);
                close(desc);
        free(rasram);
        return -1;
    }
    d = 1;
    dp = rasram;
    for(i = 0; i < rasram_size; i++){
        if(*dp++ != d++){
                    printf("inc data compare faild¥n");
            close(desc);
            free(rasram);
            return -1;
        }
    }

    /* dec data */
    d = 0xff;
    dp = rasram;
    for(i = 0; i < rasram_size; i++){
        *dp++ = d--;
    }
    lseek(desc, 0, SEEK_SET);
    len = write(desc, rasram, rasram_size);
    if(len != rasram_size){
                printf("dec write faild: err=%d¥n", errno);
                close(desc);
        free(rasram);
        return -1;
    }
    memset(rasram, 0x00, rasram_size);
    lseek(desc, 0, SEEK_SET);
    len = read(desc, rasram, rasram_size);
    if(len != rasram_size){
                printf("dec read faild: err=%d¥n", errno);
                close(desc);
```

```
        free(rasram);
        return -1;
    }
    d = 0xff;
    dp = rasram;
    for(i = 0; i < rasram_size; i++){
        if(*dp++ != d--){
                printf("dec data compare faild¥n");
            close(desc);
            free(rasram);
            return -1;
        }
    }

    close(desc);
    printf("read/write check ok¥n");

    free(rasram);
    return 0;
}
```

「/usr/local/tools-0010/samples/sampleConsole/sample_SRAM」に、バックアップ付き RAM を mmap システムコールで操作したサンプルプログラムが入っています。このサンプルプログラムはコンソールアプリケーションとして作成されています。コンソール上で動作させることができます。mmap 関数でマッピングしたアドレスに対して、バイト、ワード、ロングデータでの読み書きを行い、データが正常に書き込まれているかを確認しています。リスト 4-4-12-2 にソースコードを示します。

リスト 4-4-12-2. バックアップ SRAM 制御 mmap 方式 (rasram_mmap.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>

#include "asd_rasram.h"

unsigned long rasram_size = 0;

int main(void)
{
    int desc;
    int i;
    void *memmap = NULL;

    unsigned char b_dt;
    volatile unsigned char *b_mem;
```

```
    unsigned short w_dt;
    volatile unsigned short *w_mem;
    unsigned long l_dt;
    volatile unsigned long *l_mem;

    desc = open("/dev/g5sram0", O_RDWR);
    if(desc < 0){
        printf("open faild: err=%d¥n", errno);
        return -1;
    }

    if(ioctl(desc, SRAM_IOCGSIZE, &rasram_size) != 0){
        close(desc);
        printf("size check faild.¥n");
        return -1;
    }

    printf("rasram size is %lx.¥n", rasram_size);

    memmap = mmap(0, rasram_size, PROT_READ | PROT_WRITE, MAP_SHARED, desc, 0);
    if (!memmap) {
        printf("memmap mmap faild¥n");
                close(desc);
        return -1;
    }
    fprintf(stderr, "MemMap mmap: %08lx¥n", (unsigned long)memmap);

    /* byte check1 */
    b_dt = 1;
    b_mem = (volatile unsigned char *)memmap;
    for(i = 0; i < rasram_size; i++){
        *b_mem++ = b_dt++;
    }
    b_dt = 1;
    b_mem = (volatile unsigned char *)memmap;
    for(i = 0; i < rasram_size; i++){
        if(*b_mem++ != b_dt++){
                printf("byte check1 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("byte check1 ok¥n");

    /* byte check2 */
    b_dt = 0xff;
    b_mem = (volatile unsigned char *)memmap;
    for(i = 0; i < rasram_size; i++){
        *b_mem++ = b_dt--;
    }
    b_dt = 0xff;
```

```
    b_mem = (volatile unsigned char *)memmap;
    for(i = 0; i < rasram_size; i++){
        if(*b_mem++ != b_dt--){
                printf("byte check2 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("byte check2 ok¥n");

    /* word check1 */
    w_dt = 1;
    w_mem = (volatile unsigned short *)memmap;
    for(i = 0; i < rasram_size/2; i++){
        *w_mem++ = w_dt++;
    }
    w_dt = 1;
    w_mem = (volatile unsigned short *)memmap;
    for(i = 0; i < rasram_size/2; i++){
        if(*w_mem++ != w_dt++){
                printf("word check1 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("word check1 ok¥n");

    /* word check2 */
    w_dt = 0xffff;
    w_mem = (volatile unsigned short *)memmap;
    for(i = 0; i < rasram_size/2; i++){
        *w_mem++ = w_dt--;
    }
    w_dt = 0xffff;
    w_mem = (volatile unsigned short *)memmap;
    for(i = 0; i < rasram_size/2; i++){
        if(*w_mem++ != w_dt--){
                printf("word check2 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("word check2 ok¥n");

    /* long check1 */
    l_dt = 1;
    l_mem = (volatile unsigned long *)memmap;
    for(i = 0; i < rasram_size/4; i++){
        *l_mem++ = l_dt++;
    }
    l_dt = 1;
```

```
    l_mem = (volatile unsigned long *)memmap;
    for(i = 0; i < rasram_size/4; i++){
        if(*l_mem++ != l_dt++){
                printf("long check1 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("long check1 ok¥n");

    /* long check2 */
    l_dt = 0xffffffff;
    l_mem = (volatile unsigned long *)memmap;
    for(i = 0; i < rasram_size/4; i++){
        *l_mem++ = l_dt--;
    }
    l_dt = 0xffffffff;
    l_mem = (volatile unsigned long *)memmap;
    for(i = 0; i < rasram_size/4; i++){
        if(*l_mem++ != l_dt--){
                printf("long check2 compare faild¥n");
                    close(desc);
            return -1;
        }
    }
    printf("long check2 ok¥n");

    close(desc);
    printf("ioctl check finish¥n");
    return 0;
}
```

４－４－１３　Wake On RTC 機能について

4A シリーズには RTC デバイスとして、EPOSON 製 RX-8803LC という RTC チップが搭載されています。このチップには、通常のハードウェアクロック保持機能の他に、設定した時間で端末を起動させる機能を有しています。

WakeOnRTC の時刻設定は「２－３－８　Asd Ras Config について」の項目で説明したとおり、ASDConfig ツールで設定可能です。

また、本項で説明する、C 言語サンプルコードを使っても設定することが出来ます。

WakeOnRTC の設定を行うには、サンプルコード中の「ioctl.cpp」と「rtc_access.h」を追加して、表 4-4-13-1 に記述された関数を使って設定します。

表 4-4-13-1. Wake On RTC 設定関数一覧

関数名	引数	内容
SetHour	時 (0～23)	時を設定します。
GetHour	―	設定されている時を取得します。
SetMin	分 (0～59)	分を設定します。
GetMin	―	設定されている分を取得します。
SetWeekSel	0:曜日 1:日	曜日設定か、日設定を行うのか切り替えます。
GetWeekSel	―	現在の曜日/日の設定を取得します。
SetDate	日 (1～31)	日を設定します。曜日指定と排他となります。
GetDate	―	設定されている日を取得します。
SetWeek	曜日 (bit 指定)	曜日指定します。複数設定可能です。 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 Bit0 土 金 木 水 火 月 日
GetWeek	―	設定されている曜日を取得します。
EnableInt	0：無効 1：有効	WakeONRTC 設定の有効/無効をせっていします。
ClearInt	―	WOR 発生フラグをクリアします。

リスト 4-4-13-1 にソースコードを示します。現在時刻を読み出して、5 分後に WakeOnRTC が動作するように設定しています。WakeOnRTC が実行されるまでにシャットダウンさせておいてください。

リスト 4-4-13-1. WakeOnRTC 設定 (5 分後に起動設定)

```
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

#include "rtc_access.h"

int main(void)
{
    time_t timer;
    struct tm *t_st;
```

```
    /* 現在時刻の取得 */
    time(&timer);

    timer += 300; /*5 分後(300 秒後) に起動する設定*/

   /* 現在時刻を構造体に変換 */
   t_st = localtime(&timer);


    EnableInt(1);           /* RTC 有効 */
    SetHour(t_st->tm_hour); /* 時設定 */
    SetMin(t_st->tm_min);   /* 分設定*/

/*日設定と曜日設定は排他です。必ずどちらかを設定してください。サンプルでは日設定で 5 分後に
起動する設定です。*/
#if 1
/*日設定*/
/*毎月※日に起動する設定にする場合、※の値を設定します。*/
    SetWeekSel(1);          /* 日設定を選択 */
    SetDate(t_st->tm_mday); /*日設定*/
#else
/*曜日設定*/
/*毎週※曜日に設定する場合は、起動したい曜日の bit を ON します。*/
/*bit   bit6 bit5 bit4 bit3 bit2 bit1 bit0 */
/*曜日    土    金    木    水    火    月    日  */
    SetWeekSel(0);  /* 曜日設定を選択 */
    SetWeek(0x2A);  /*月 水 金で起動 */
#endif

    ClearInt();     /* RTC ステータスクリア */

    printf("5 分後に WakeOnRTC が働きます。シャットダウンしておいてください。¥n");

}
```

４－４－１４　S.M.A.R.T.監視機能について

EC4A シリーズには、MainStrage/SubStrage m-SATA の寿命監視機能が搭載されています。書き込み回数が警報値を超えた場合、または、BadBlock が検出されたときイベントが発報されます。

ユーザーアプリケーションは、それぞれのディスクのクリティカル警報とワーニング警報を受け取ることができます。警告の受け取りには、POSIX セマフォを用います。EC4A シリーズで使用できるセマフォを表 4-4-14-1 に示します。

表 4-4-14-1. S.M.A.R.T.用 POSIX セマフォ名

セマフォ名称	用途
/smart1_critical	MainStrage クリティカル警報
/smart1_warning	MainStrage ワーニング警報
/smart2_critical	SubStrage クリティカル警報
/smart2_warning	SubStrage ワーニング警報

セマフォによる警報取得は、Action 設定が Event に設定しているときに行われます。詳細は、『2-3-9 ASD Smart Config について』を参照してください。

４－４－１５　S.M.A.R.T.機能サンプルプログラム

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_SMART」に、S.M.A.R.T.機能を使ったサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。ソースコードをリスト 4-4-15-1 に示します。

サンプルでは、まず「sem_open」関数でセマフォをオープンし、「sem_wait」関数で通知を待ちます。セマフォをクローズするには「sem_close」関数を呼びます。

リスト 4-4-15-1. S.M.A.R.T.通知待ちサンプルソースコード（main.c）

```
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

#include "asd_rassmart.h"

struct smart_t {
    int occured;
    int thread_start;
    const char* message;
    sem_t* semaphore;
    pthread_t thread;
};

static void smart_close(struct smart_t* data);
static void* event_monitor(void* arg);

int main(void)
{
    int key;
    struct smart_t MainStrage_Critical_data = {
```

```
        .occured = 0,
        .thread_start = 0,
        .message = "MainStrage S.M.A.R.T Critical.¥n",
        .semaphore = SEM_FAILED,
    };
    struct smart_t MainStrage_Warning_data = {
        .occured = 0,
        .thread_start = 0,
        .message = "MainStrage S.M.A.R.T Warning.¥n",
        .semaphore = SEM_FAILED,
    };
    struct smart_t SubStrage_Critical_data = {
        .occured = 0,
        .thread_start = 0,
        .message = "SubStrage S.M.A.R.T Critical.¥n",
        .semaphore = SEM_FAILED,
    };
    struct smart_t SubStrage_Warning_data = {
        .occured = 0,
        .thread_start = 0,
        .message = "SubStrage S.M.A.R.T Warning.¥n",
        .semaphore = SEM_FAILED,
    };


    // open semaphore
    MainStrage_Critical_data.semaphore = sem_open(DISK1SMARTCRITICAL_SEMAPHORE_PATH, 0);
    if (MainStrage_Critical_data.semaphore == SEM_FAILED) {
        perror("MainStrage_Critical_data semaphore open failed");
        smart_close(&MainStrage_Critical_data);
        return -1;
    }
    MainStrage_Warning_data.semaphore = sem_open(DISK1SMARTWARNING_SEMAPHORE_PATH, 0);
    if (MainStrage_Warning_data.semaphore == SEM_FAILED) {
        perror("MainStrage_Warning_data semaphore open failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        return -1;
    }
    SubStrage_Critical_data.semaphore = sem_open(DISK2SMARTCRITICAL_SEMAPHORE_PATH, 0);
    if (SubStrage_Critical_data.semaphore == SEM_FAILED) {
        perror("SubStrage_Critical_data semaphore open failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        return -1;
    }
    SubStrage_Warning_data.semaphore = sem_open(DISK2SMARTWARNING_SEMAPHORE_PATH, 0);
    if (SubStrage_Warning_data.semaphore == SEM_FAILED) {
        perror("SubStrage_Warning_data semaphore open failed");
        smart_close(&MainStrage_Critical_data);
```

```
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        smart_close(&SubStrage_Warning_data);
        return -1;
    }

    // create threads
    if (pthread_create(&MainStrage_Critical_data.thread, NULL, &event_monitor,
&MainStrage_Critical_data) != 0) {
        perror("MainStrage_Critical_data thread create failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        smart_close(&SubStrage_Warning_data);
        return -1;
    }
    if (pthread_create(&MainStrage_Warning_data.thread, NULL, &event_monitor,
&MainStrage_Warning_data) != 0) {
        perror("MainStrage_Warning_data thread create failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        smart_close(&SubStrage_Warning_data);
        return -1;
    }
    if (pthread_create(&SubStrage_Critical_data.thread, NULL, &event_monitor,
&SubStrage_Critical_data) != 0) {
        perror("SubStrage_Critical_data thread create failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        smart_close(&SubStrage_Warning_data);
        return -1;
    }
    if (pthread_create(&SubStrage_Warning_data.thread, NULL, &event_monitor,
&SubStrage_Warning_data) != 0) {
        perror("SubStrage_Warning_data thread create failed");
        smart_close(&MainStrage_Critical_data);
        smart_close(&MainStrage_Warning_data);
        smart_close(&SubStrage_Critical_data);
        smart_close(&SubStrage_Warning_data);
        return -1;
    }

    // wait key input
    key = getchar();

    smart_close(&MainStrage_Critical_data);
    smart_close(&MainStrage_Warning_data);
    smart_close(&SubStrage_Critical_data);
    smart_close(&SubStrage_Warning_data);
```

```
    return 0;
}
void smart_close(struct smart_t* p_smart)
{
    if (p_smart == NULL) {
        return;
    }
    if (p_smart->semaphore != SEM_FAILED) {
        sem_close(p_smart->semaphore);
        p_smart->semaphore = SEM_FAILED;
    }
    if (p_smart->thread_start) {
        p_smart->thread_start = 0;
        pthread_kill(p_smart->thread, SIGUSR1);
    }
}
void* event_monitor(void* arg)
{
    struct smart_t* data = (struct smart_t*)arg;

    data->thread_start = 1;

    while (1) {
        sem_wait(data->semaphore);
        printf("%s¥n", data->message);
        usleep(100 * 1000);
    }

    data->thread_start = 0;
    return NULL;
}
```

４－５　タイマ割込み機能

４－５－１　タイマ割込み機能について

EC4A シリーズには、ハードウェアによるタイマ割込み機能が実装されています。タイマ割込み機能を使用することで、指定した時間で周期的に割込みを発生させることができます。タイマドライバによって生成されるタイマデバイスを操作することによって、ユーザアプリケーションからタイマ割込み機能を利用できるようになります。タイマデバイスでは、タイマ設定、タイマイベント通知などの機能を使用することができます。

４－５－２　タイマ割込み機能デバイスについて

タイマドライバはタイマデバイスを生成します。ユーザアプリケーションは、デバイスファイルにアクセスすることによってタイマ機能を操作します。表 4-5-2-1 にタイマデバイスの詳細を示します。

表 4-5-2-1. タイマデバイスリファレンス

タイマデバイス
名前
rastime
ヘッダ
#include "asd_rastime.h"
説明
タイマ時間設定、タイマ開始/停止、タイマイベント待機を行うことが出来ます。
OPEN
デバイスファイル(/dev/rastime)を**open**関数でオープンします。 システム全体で、同時に16までオープンすることが出来ます。 timerfd = open("/dev/rastime", O_RDWR);
READ
使用しません
WRITE
使用しません

IOCTL

タイマデバイスを操作するには**ioctl**関数を使用します。

```
error = ioctl(fd, ctlcode, param);
```

● **ctlcode**

RASTIME_IOCTSTART
タイマを開始します。
[param] なし (NULL)
[error] 0:正常、0以外:異常

RASTIME_IOCTSTOP
タイマを停止します。
[param] なし (NULL)
[error] 0:正常、0以外:異常

RASTIME_IOCSCONFIG
タイマを設定します。
[param] RASTIME_CONFIG型変数のポインタ
[error] 0:正常、0以外:異常

RASTIME_IOCGCONFIG
タイマ設定を取得します。
[param] RASTIME_CONFIG型変数のポインタ
[error] 0:正常、0以外:異常

RASTIME_IOCSTIMERRES
タイマ解像度を設定します。(設定値: 1～65535[msec]、初期値: 10[msec])
システムで使用されている全てのrastimeがこのタイマ解像度で動作します。
変更する場合は、他のrastimeが動作していないかどうか注意してください。
[param] unsigned long型変数のポインタ

RASTIME_IOCGTIMERRES
タイマ解像度を取得します。
[param] unsigned long型変数のポインタ
[error] 0:正常、0以外:異常

RASTIME_IOCQWAITEVENT
タイマイベントを待ちます。
このコントロールコードでioctlを呼び出すとタイマイベント発生、または強制解除されるまで関数からリターンしません。
[param] なし (NULL)
[error] 1:イベント発生、2:強制解除、その他:異常

RASTIME_IOCQWAKEUP
タイマイベント待ちを強制解除します。
[param] なし (NULL)
[error] 0:正常、0以外:異常

● **param**

RASTIME_CONFIG構造体
タイマの設定を格納します。

```
struct RASTIME_CONFIG
{
  unsigned long type;       // タイマタイプ 0:1回で終了, 1:繰返し
  unsigned long duetime;    // タイマ時間 1～ [msec]
};
```

４－５－３　タイマ割込み機能サンプルプログラム

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_Timer」に、タイマデバイスを制御するサンプルプログラムが入っています。リスト 4-5-3-1 にサンプルプログラムのソースコードを示します。タイマ解像度の変更、タイマ設定、タイマ開始/停止、タイマイベント待ちを行うサンプルです。このサンプルプログラムでは 10 個のタイマを同時に使用しています。

リスト 4-5-3-1. タイマデバイス操作ソースコード（main.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <stdarg.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <bits/local_lim.h>
#include "asd_rastime.h"

#define TIMERDEV_FILENAME   "/dev/rastime"
#define MAX_TIMEREVENT      10

//---------------------------------------------------------------------------
typedef struct {
    int No;
    volatile int fStart;
    volatile int fFinish;
    pthread_t ThreadID;
    int TimerDesc;
    struct RASTIME_CONFIG Config;
} TIMEREVENT_INFO;

//---------------------------------------------------------------------------
static void _time_print(const char *format, ...)
{
    struct timeval tv;
    struct tm *tmptr = NULL;
    char txt[256];
    va_list arg;

    va_start(arg, format);
    vsprintf(txt, format, arg);
    va_end(arg);

    gettimeofday(&tv,NULL);
```

```
    tmptr = localtime(&tv.tv_sec);
    printf("%02d:%02d:%02d.%03ld: %s",
        tmptr->tm_hour,             // hour
        tmptr->tm_min,              // min
        tmptr->tm_sec,              // sec
        (tv.tv_usec + 500) / 1000,  // msec
        txt
    );
}

//-----------------------------------------------------------------------------
/*
 * タイマイベント処理スレッド
 */
static void *TimerEventProc(void *arg)
{
    TIMEREVENT_INFO *info = (TIMEREVENT_INFO *)arg;
    int ret;

    printf("TimerEventProc: Timer%02d: Start¥n", info->No);

    info->fFinish = 0;
    while(1){
        ret = ioctl(info->TimerDesc, RASTIME_IOCQWAITEVENT, NULL);
        if(ret != RASTIME_TIMER){
            printf("ioctl: RASTIME_IOCQWAITEVENT: WAKEUP¥n");
            break;
        }
        if(!info->fStart){
            break;
        }
        _time_print("Timer%02d¥n", info->No);
    }
    info->fFinish = 1;

    printf("TimerEventProc: Timer%02d: Finish¥n", info->No);
    return 0;
}

//-----------------------------------------------------------------------------
static int CreateTimerEventInfo(int No, TIMEREVENT_INFO *info)
{
    pthread_attr_t thread_attr;

    pthread_attr_init(&thread_attr);
    pthread_attr_setstacksize(&thread_attr, 0x10000);

    info->No = No;
    info->ThreadID = 0;
    info->fStart = 0;
    info->fFinish = 0;
```

```
    info->TimerDesc = -1;

    /*
     * タイマデバイスのオープン
     */
    info->TimerDesc = open(TIMERDEV_FILENAME, O_RDWR);
    if(info->TimerDesc < 0){
        printf("%s: open faild: err=%d¥n", TIMERDEV_FILENAME, errno);
        return -1;
    }

    /*
     * タイマの設定
     */
    info->Config.type = 1;
    info->Config.duetime = 200 * (No + 1);
    if(ioctl(info->TimerDesc, RASTIME_IOCSCONFIG, &info->Config) != 0){
        close(info->TimerDesc);
        printf("ioctl: RASTIME_IOCSCONFIG: error¥n");
        return -1;
    }

    /*
     * タイマイベント処理スレッド作成
     */
    info->fStart = 1;
    if(pthread_create(&info->ThreadID, &thread_attr, TimerEventProc, (void *)info) != 0){
        close(info->TimerDesc);
        printf("pthread_create: error¥n");
        return -1;
    }

    return 0;
}

//-----------------------------------------------------------------------
static void StartTimer(TIMEREVENT_INFO *info)
{
    /*
     * タイマスタート
     */
    ioctl(info->TimerDesc, RASTIME_IOCTSTART, NULL);
}

//-----------------------------------------------------------------------
static void DeleteTimer(TIMEREVENT_INFO *info)
{
    /*
     * スレッド停止
     */
    info->fStart = 0;
```

```
    /*
     * タイマ停止
     * タイマイベント待ち解除
     * タイマ破棄
     */
    ioctl(info->TimerDesc, RASTIME_IOCTSTOP, NULL);
    ioctl(info->TimerDesc, RASTIME_IOCQWAKEUP, NULL);
    close(info->TimerDesc);

    /*
     * スレッド停止待ち
     */
    while(!info->fFinish){
        usleep(10 * 1000);
    }
    pthread_cancel(info->ThreadID);
}

//---------------------------------------------------------------------------
static int SetTimerResolution(void)
{
    int timerfd;
    unsigned long timer_resolution;

    /*
     * タイマデバイスのオープン
     */
    timerfd = open(TIMERDEV_FILENAME, O_RDWR);
    if(timerfd < 0){
        printf("%s: open faild: err=%d¥n", TIMERDEV_FILENAME, errno);
        return -1;
    }

    /*
     * タイマ解像度取得
     */
    if(ioctl(timerfd, RASTIME_IOCGTIMERRES, &timer_resolution) != 0){
        close(timerfd);
        printf("ioctl: RASTIME_IOCGTIMERRES: error¥n");
        return -1;
    }
    printf("SetTimerResolution: RASTIME_IOCGTIMERRES: timer_resolution=%ld¥n",
timer_resolution);

    /*
     * タイマ解像度変更
     *
     * タイマ解像度が変わります。
     * 他のプロセスで rastime を使用している場合は注意してください。
     */
```

```
    if(timer_resolution == 10){
        close(timerfd);
        return 0;
    }
    timer_resolution = 10;
    if(ioctl(timerfd, RASTIME_IOCSTIMERRES, &timer_resolution) != 0){
        close(timerfd);
        printf("ioctl: RASTIME_IOCSTIMERRES: error¥n");
        return -1;
    }
    printf("SetTimerResolution: RASTIME_IOCSTIMERRES: timer_resolution=%ld¥n",
timer_resolution);

    close(timerfd);
    return 0;
}

//---------------------------------------------------------------------------
int main(void)
{
    int i;
    int c;
    TIMEREVENT_INFO info[MAX_TIMEREVENT];

    if(SetTimerResolution() != 0){
        return -1;
    }

    for(i = 0; i < MAX_TIMEREVENT; i++){
        if(CreateTimerEventInfo(i, &info[i]) != 0){
            printf("CreatTimerEvent: NG: %d¥n", i);
            return -1;
        }
    }
    for(i = 0; i < MAX_TIMEREVENT; i++){
        StartTimer(&info[i]);
    }

    while(1){
        c = fgetc(stdin);
        if(c == 'q' || c == 'Q'){
            break;
        }
    }

    for(i = 0; i < MAX_TIMEREVENT; i++){
        DeleteTimer(&info[i]);
    }

    return 0;
}
```

４－６　UPS 機能

４－６－１　UPS 機能について

EC4A シリーズには、UPS 機能が搭載されています。UPS 機能により、電源断が発生したときに、AC 電源駆動からバッテリ駆動に切り替わり、安全にデータ退避、シャットダウン処理を行うことができます。

ユーザーアプリケーションは、電源異常の通知、電源断の警告を受け取ることができます。通知、警告の受け取りには、POSIX セマフォを用います。EC4A シリーズで使用できるセマフォを表 4-6-1-1 に示します。

表 4-6-1-1. UPS 用 POSIX セマフォ名

セマフォ名称	用途
/ups_notification	電源異常通知
/ups_alarm	電源断警告

電源異常通知、電源断警告は、機能が有効に設定されていなければ発生しません。また、電源異常/電源断が発生してから、実際に通知が行われるまでの時間は設定可能です。詳細は、『2-3-4　ASD UPS Config について』を参照してください。

４－６－２　UPS 機能サンプルプログラム

●コンソール用サンプルプログラム

「/usr/local/tools-0010/samples/sampleConsole/sample_UPS」に、UPS 機能を使ったサンプルプログラムが入っています。コンソールウィンドウを起動して、コンパイルしたコマンドを実行します。ソースコードをリスト 4-6-2-1 に示します。

サンプルでは、まず「sem_open」関数でセマフォをオープンし、「sem_wait」関数で通知を待ちます。セマフォをクローズするには「sem_close」関数を呼びます。

リスト 4-6-2-1. UPS 通知待ちサンプルソースコード（main.c）

```
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

#define NOTIFICATION_SEMAPHORE_PATH "/ups_notification"
#define ALARM_SEMAPHORE_PATH        "/ups_alarm"

struct ups_t {
    int occured;
    int thread_start;
    const char* message;
    sem_t* semaphore;
    pthread_t thread;
};

static void ups_close(struct ups_t* data);
static void* event_monitor(void* arg);

int main(void)
{
    int key;
```

```
    struct ups_t notification_data = {
        .occured = 0,
        .thread_start = 0,
        .message = "UPS error occured.¥n",
        .semaphore = SEM_FAILED,
    };
    struct ups_t alarm_data = {
        .occured = 0,
        .thread_start = 0,
        .message = "Power down detected.¥n",
        .semaphore = SEM_FAILED,
    };

    // open semaphore
    notification_data.semaphore = sem_open(NOTIFICATION_SEMAPHORE_PATH, 0);
    if (notification_data.semaphore == SEM_FAILED) {
        perror("notification semaphore open failed");
        ups_close(&notification_data);
        return -1;
    }
    alarm_data.semaphore = sem_open(ALARM_SEMAPHORE_PATH, 0);
    if (alarm_data.semaphore == SEM_FAILED) {
        perror("alarm semaphore open failed");
        ups_close(&notification_data);
        ups_close(&alarm_data);
        return -1;
    }
    // create threads
    if (pthread_create(&notification_data.thread, NULL, &event_monitor,
&notification_data) != 0) {
        perror("notification thread create failed");
        ups_close(&notification_data);
        ups_close(&alarm_data);
        return -1;
    }
    if (pthread_create(&alarm_data.thread, NULL, &event_monitor, &alarm_data) != 0) {
        perror("alarm thread create failed");
        ups_close(&notification_data);
        ups_close(&alarm_data);
        return -1;
    }
    // wait key input
    key = getchar();

    ups_close(&notification_data);
    ups_close(&alarm_data);

    return 0;
}
void ups_close(struct ups_t* p_ups)
{
```

```
    if (p_ups == NULL) {
        return;
    }
    if (p_ups->semaphore != SEM_FAILED) {
        sem_close(p_ups->semaphore);
        p_ups->semaphore = SEM_FAILED;
    }
    if (p_ups->thread_start) {
        p_ups->thread_start = 0;
        pthread_kill(p_ups->thread, SIGUSR1);
    }
}
void* event_monitor(void* arg)
{
    struct ups_t* data = (struct ups_t*)arg;

    data->thread_start = 1;

    while (1) {
        sem_wait(data->semaphore);
        printf("%s¥n", data->message);
        usleep(100 * 1000);
    }

    data->thread_start = 0;
    return NULL;
}
```

４－７　その他のサンプルプログラム

４－７－１　マルチランゲージ表示サンプルプログラム

●WideStudio 用サンプルプログラム

「/usr/local/tools-0010/samples/sampleWideStudio/sample_MultiLang」に、日英中韓の文字列を同時に表示するプログラムのサンプルプログラムのソースコードが入っています。多言語を同時に表示する為には、文字コードを UTF-8 にする必要があります。その為、サンプルプログラムのソースコードのファイルも UTF-8 形式でないと正常に読み出すことができないのでご注意ください。UTF-8 の文字コードで書かれた文字列を表示するようにして、対応するフォントを用意できれば、多言語プログラムが実現します。

多言語表示のサンプルプログラムを実行した画面を図 4-7-1-1 に示します。

図 4-7-1-1. 多言語表示実行画面

Algonomix4 標準では、中国語と韓国語のフォントが実装されていないため、正しく表示できません。TrueType の中国語、韓国語のフォントを実装すれば改善されます。

４－７－２　テンキーボードサンプル

●WideStudio 用サンプルプログラム

「/usr/local/tools-0010/samples/sampleWideStudio/sample_10Key」に、タッチパネルアプリケーションにおいて数字入力でよく使用されるテンキーキーボードを表示するサンプルプログラムのソースコードが入っています。数値入力をしたいボタンのプロシージャで図のようなテンキー画像を出力する関数を呼び出します。数値を入力して「OK」ボタンを押すことで、その数値が最初に起動したボタン上に表示されます。

テンキーボードのサンプルプログラムを実行した画面を図 4-7-2-1 に示します。

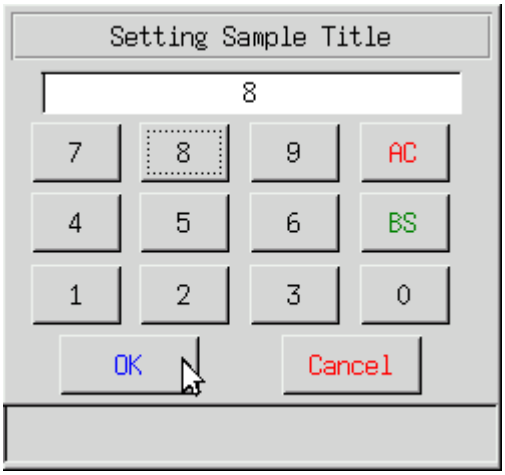


図 4-7-2-1. テンキー入力画面

４－７－３　起動ランチャーサンプルプログラム

●WideStudio 用サンプルプログラム

「/usr/local/tools-0010/samples/sampleWideStudio/sample_Launcher」に、別のアプリケーションを起動するランチャープログラムのサンプルプログラムのソースコードが入っています。それぞれのボタンをクリックすることで、それぞれ対応したアプリケーションが起動します。

起動ランチャーのサンプルプログラムを実行した画面を図 4-7-3-1 に示します。

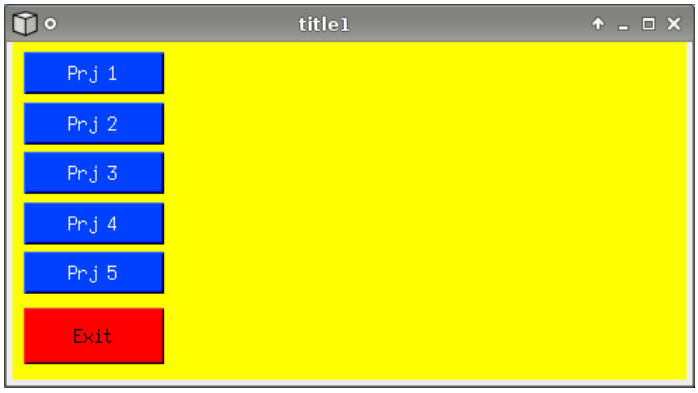


図 4-7-3-1. 起動ランチャー画面

ボタンクリックしたときのプロシージャイベント関数のソースコードをリスト 4-7-3-1 に示します。

リスト 4-7-3-1. 起動ランチャーボタンクリックイベント (Btn_Click.cpp)

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
#include "newwin000.h"
//---------------------------------------------------------
//Function for the event procedure
//---------------------------------------------------------
void Btn_Click(WSCbase* object){
    if(object == btn_prg1){
        system("xeyes &");          /* xeyes ﾌﾟﾛｸﾞﾗﾑを起動 */
    }
    if(object == btn_prg2){
        system("xclock &");         /* xclock ﾌﾟﾛｸﾞﾗﾑを起動 */
    }
    if(object == btn_prg3){
        system("xcalc &");          /* xcalc ﾌﾟﾛｸﾞﾗﾑを起動 */
    }
    if(object == btn_prg4){
        system("xlogo &");          /* xlogo ﾌﾟﾛｸﾞﾗﾑを起動 */
    }
    if(object == btn_prg5){
        system("chromium browser &");       /* chromium browser ﾌﾟﾛｸﾞﾗﾑを起動 */
    }
    if(object == btn_exit){
        exit(0);                /* 終了 */
    }
}
static WSCfunctionRegister  op("Btn_Click",(void*)Btn_Click);
```

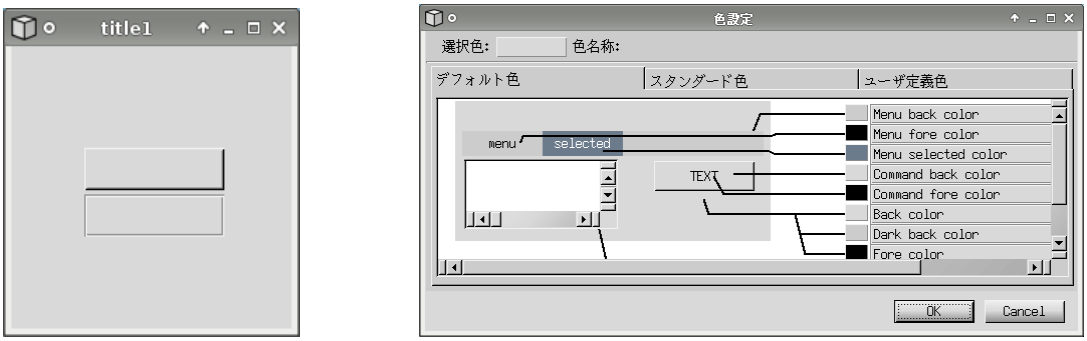
ボタンがクリックされると、object にボタンのポインタが渡されます。ここでは、押されたボタン毎に「system」という関数の引数で渡された文字列のコマンドを実行します。コマンドの後ろに「&」が記述されるとコマンドの終了まで待ちません。「&」を記述しないでコマンドを実行することで、そのコマンドが終了されるまで「system」関数から返りません。

４－７－４　色選択サンプルプログラム

●WideStudio 用サンプルプログラム

「/usr/local/tools-0010/samples/sampleWideStudio/sample_ColorPalette」に、カラーパレットを表示するサンプルプログラムのソースコードが入っています。ボタンを押すと、ボタンのプロシージャでカラーパレットが呼び出されます。カラーパレットで色を選択し、「OK」ボタンを押すと、ボタンの背景色が選択した色になり、色名称がラベルに表示されます。

色選択サンプルプログラムを実行した画面を図 4-7-4-1 に示します。



（起動画面）　　（カラーパレット）

図 4-7-4-1. 色選択サンプルプログラム

４－８　ストレージデバイスについて

ここでは、外部ストレージデバイスの使用方法について説明します。

４－８－１　外部ストレージデバイスの使用方法

外部ストレージデバイスはブロックデバイスを使用して通常のディスクと同様に操作することができます。
Algonomix6 の初期設定では、外部ストレージデバイスは自動的にマウントされます。手動でマウントする必要がある場合に、本項目を参照してください。

●外部ストレージデバイスのマウント
外部ストレージデバイスのブロックデバイスをマウントします。(/dev/sdb と認識した場合)

例：外部ストレージデバイスのパーティション 1 をマウント

```
# mount /dev/sdb1 /mnt
```

●外部ストレージデバイスのファイルへのアクセス
マウント以後は､マウントしたディレクトリから外部ストレージデバイス内のファイルにアクセスができます。ファイル操作方法は、ルートファイルシステム上のファイルと同様です。

例：外部ストレージデバイス内ファイル一覧表示

```
# cd /mnt
# ls
file1     file2     file3
```

例：外部ストレージデバイスへのファイルコピー

```
# cp /home/asdusr/FILE /mnt
```

●外部ストレージデバイスのアンマウント
外部ストレージデバイスを使用し終わったらブロックデバイスをアンマウントします。外部ストレージデバイスを抜く前に必ずアンマウントを行ってください。

例：外部ストレージデバイスのアンマウント

```
# umount /mnt
```

※注：外部ストレージデバイスのデバイス名(/dev/sdb など)については『4-8-2　ストレージデバイス名の割り振りについて』を参照してください。

４－８－２　ストレージデバイス名の割り振りについて

各ストレージデバイスは Linux 上で使用するためにデバイス名が以下のような名前で割り振られます。

/dev/sdx　　：　ストレージデバイス全体のブロックデバイス
x は a, b, c, d･･･と認識順に増えていきます
/dev/sdx*　　：　ストレージデバイス上パーティションのブロックデバイス
*はパーティション毎に 1, 2, 3･･･と増えていきます。
（例：　/dev/sdb1, /dev/sdb2）

各ストレージデバイスは以下のルールに従い、デバイス名にアルファベットの若い文字から割り振られます。

(1) OS 起動時に各ストレージデバイススロットにデバイスが挿入されていた場合、接続されているデバイスが表 4-8-2-1 の優先順位に従ってデバイス名が割り振られます。（接続されていない場合はスキップします。）
(2) OS 起動後､新たに USB メモリなどを挿入した場合､それらの USB メモリにデバイス名が割り振られます。

EC4A シリーズでの認識優先度を表 4-8-2-1 に示します。

表 4-8-2-1. ストレージデバイス認識優先度

優先度	スロット名
①	m-SATA メインストレージ
②	m-SATA サブストレージ
③	USB スロット 0～3

※注：USB スロットや SSD スロットに複数のストレージを接続したときは、それぞれにデバイス名が若いアルファベット文字から割り振られます。

例 1:mSATA メインストレージのみ挿入して OS 起動したとき

mSATA メインストレージが sda として認識されます。その後挿入された USB メモリは sdb、sdc･･･として認識されます。

表 4-8-2-2. デバイス名割り振り

スロット名	デバイス名
m-SATA メインストレージ	/dev/sda
USB スロット 0～3	/dev/sdb、/dev/sdc･･･

例 2:mSATA メインストレージと m-SATA サブストレージを挿入して OS 起動したとき

mSATA メインストレージが sda として、m-SATA サブストレージが sdb として認識されます。その後挿入された USB メモリは sdc、sdd･･･として認識されます。

表 4-8-2-3. デバイス名割り振り

スロット名	デバイス名
m-SATA メインストレージ	/dev/sda
m-SATA サブストレージ	/dev/sdb
USB スロット 0～3	/dev/sdc、/dev/sdd･･･

４－８－３　外部ストレージデバイスの起動時マウントについて

ここでは、外部ストレージデバイス(SD カードなど)を、起動時に任意のディレクトリにマウントする方法について記述します。

Linux では、ストレージのマウント情報は/etc/fstab というファイルに記述されます。/etc/fstab を編集することで、起動時にデバイスをマウントすることができます。

以下に、SD カードを/home/asdusr/sd ディレクトリにマウントする例を示します。マウント先のディレクトリは、mkdir コマンドで事前に作成する必要があります。

リスト 4-8-3-1. SD カードの起動時マウント設定例 (/etc/fstab)

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda2 during installation
UUID=0b45b6fb-1646-4d70-bd5b-6e221decd535 /               ext4    noatime,errors=remount-ro 0
0
# /boot was on /dev/sda1 during installation
UUID=372c52f7-6bea-474c-98f3-101637bb047e /boot           ext4    noatime,defaults          0
0
# /home was on /dev/sda3 during installation
UUID=41e73ccd-2280-4e91-8018-0644747ec214 /home           ext4    noatime,defaults          0
0
# /usr/local was on /dev/sda4 during installation
UUID=8a2f8345-1607-4ef0-8a5e-ad951606bacc /usr/local      ext4    noatime,defaults          0
0

tmpfs   /tmp        tmpfs   defaults,size=192m  0   0
tmpfs   /var/log    tmpfs   defaults,size=64m   0   0
/dev/mmcblk0p1  /home/asdusr/sd vfat    defaults,nofail 0   0
```

この行を追加

４－９　LTE について

LTE 機能を使用して LTE 通信を行うことができます。
例として Docomo の SP モードを使用して LTE 通信を行う場合、以下のように設定を行います。

① コンソールを起動させ、下記のコマンドを実行します。

```
# wvdialconf /etc/wvdial.conf
```

② OS を再起動してください。

③ [メニューアイコン]→[設定]→[ネットワーク接続]を選択します。

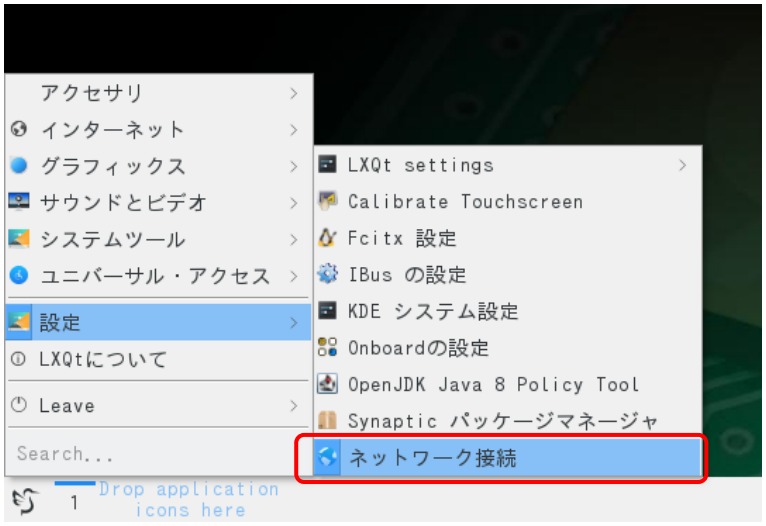


図 4-9-1. ネットワーク設定画面の起動

④ 図 4-9-2 のようなネットワーク設定画面が起動します。
「Add」ボタンをクリックしてください。

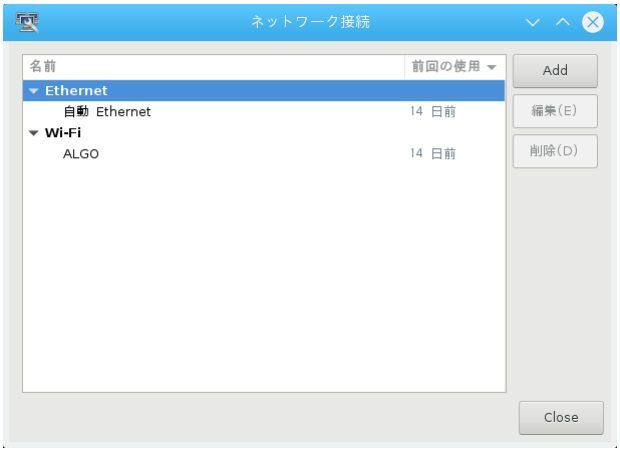


図 4-9-2. ネットワーク設定画面

⑤　図 4-9-3 のような接続種類を選択するダイアログが表示されます。
「モバイルブロードバンド」を選択して「作成」ボタンをクリックしてください。

図 4-9-3.　ネットワーク種類の選択

⑥　モバイルブロードバンド接続のセットアップ画面が開きます。
ご使用の SIM カードの資料を参照し、設定をしてください。

図 4-9-4.　接続のセットアップ

※　設定中、数画面が表示されますが、使用する SIM カードにより画面は異なります。
最終的に図 4-9-6 のような画面が表示されれば完了です。

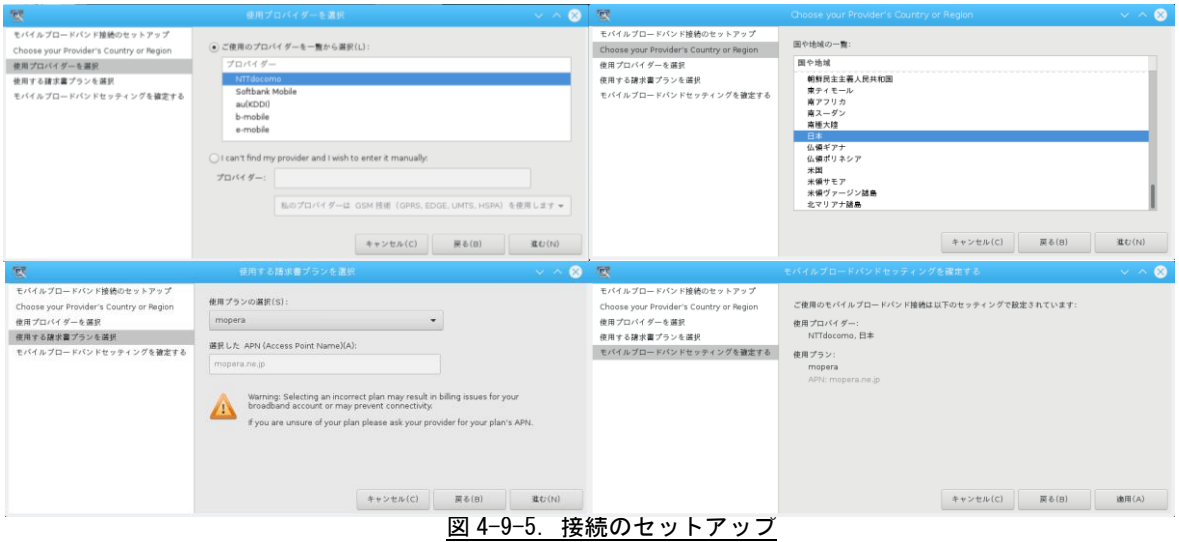


図 4-9-5.　接続のセットアップ

⑦　通信設定の編集画面が開きます。
この画面もご使用の SIM カードの内容に従い設定をしてください。

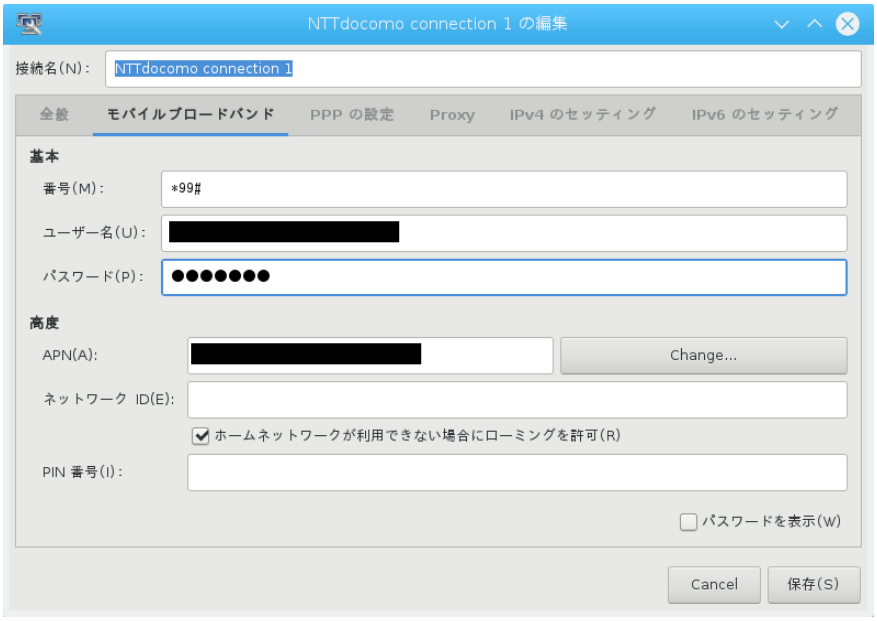


図 4-9-6.　ネットワーク設定の編集

⑧　設定が完了するとタスクバーの右下に設定した項目が追加されます。
この項目をクリックすると通信が開始されます。

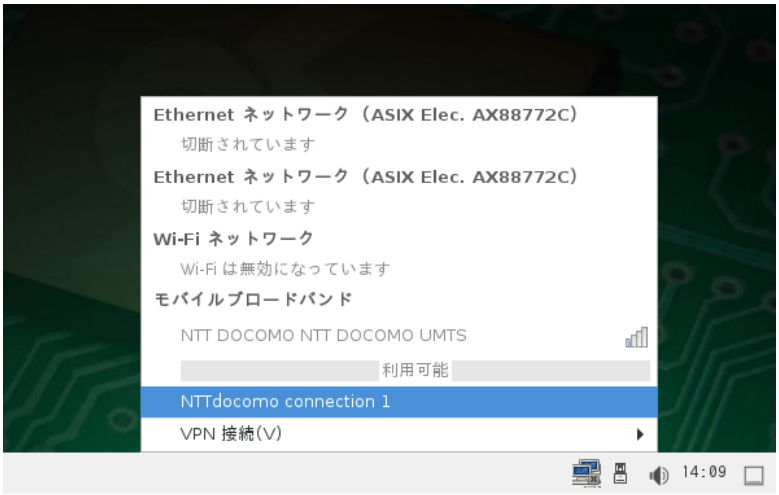


図 4-9-7.　通信の開始

注）　起動している間は LTE 通信したままとなります。
この場合、通信料が発生し続けるのでご注意ください。

第５章　システムリカバリ

本章では、「EC4A 用 Algonomix6 64 ビット版 リカバリ DVD」を使用したシステムのリカバリとバックアップについて説明します。

５－１　リカバリ DVD について

EC4A シリーズ本体は、システムのリカバリを行うことができます。リカバリで行える処理は以下のとおりです。

- システムの復旧（バックアップデータ）
- システムのバックアップ

リカバリを実行するにはリカバリ DVD 以外に以下のものを用意する必要があります。

- 4GByte 以上の USB メモリ（リカバリ USB 用）
- 8GByte 程度の空き容量がある USB 接続可能なストレージメディア（USB メモリなど）
  （バックアップイメージ保存用）

リカバリ DVD を実行する場合、BIOS 設定が必要となります。また、リカバリ DVD での作業終了後は BIOS の設定を元に戻す必要があります。

- リカバリ実行の流れ

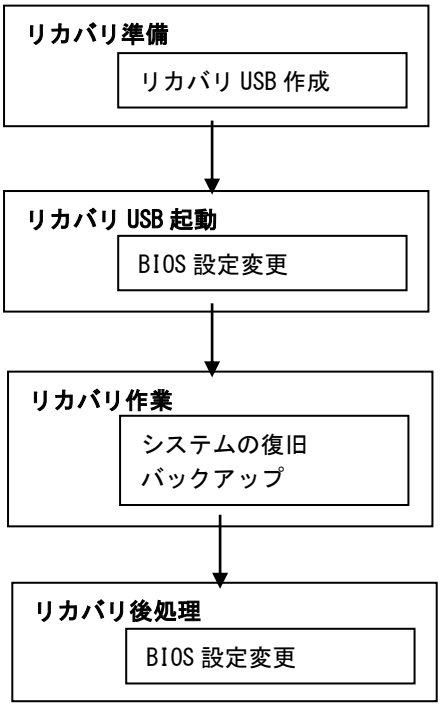


図 5-1-1. リカバリ DVD 使用の流れ

５－１－１　リカバリ準備

リカバリを実行する前に、リカバリシステムを起動するための USB メモリを作成します。
4GByte 以上のサイズの USB メモリをあらかじめ用意してください。

※注：　ここで用意した USB メモリの中身は全て消去されます。
あらかじめバックアップをとるなどしておいてください。

● リカバリ USB 作成手順

① Windows が動作する PC にリカバリ DVD を挿入します。
② 用意した USB メモリを、手順①の PC に接続します。
③ リカバリ DVD の以下のファイルを実行します。
[リカバリ DVD]¥Recovery¥RecoveryUSBImage.exe

④ 圧縮ファイルの解凍が始まります。
PC 上の任意の場所に解凍してください。
解凍が完了するまでお待ちください。
解凍が完了すると「RecoveryUSBImage.ddi」というファイルが展開されます。
⑤ リカバリ DVD の以下のファイルを実行します。
[リカバリ DVD]¥Recovery¥DDwin¥DDwin.exe
※注：　WindowsVista 以降の OS をご使用の場合は管理者権限で起動する必要があります。

⑥ DD for Windows というツールが起動します。
⑦ 「対象ディスク」の項目に手順②で接続した USB メモリが表示されていることを確認してください。
「ディスク選択」ボタンを押して接続した USB メモリを選択してください。

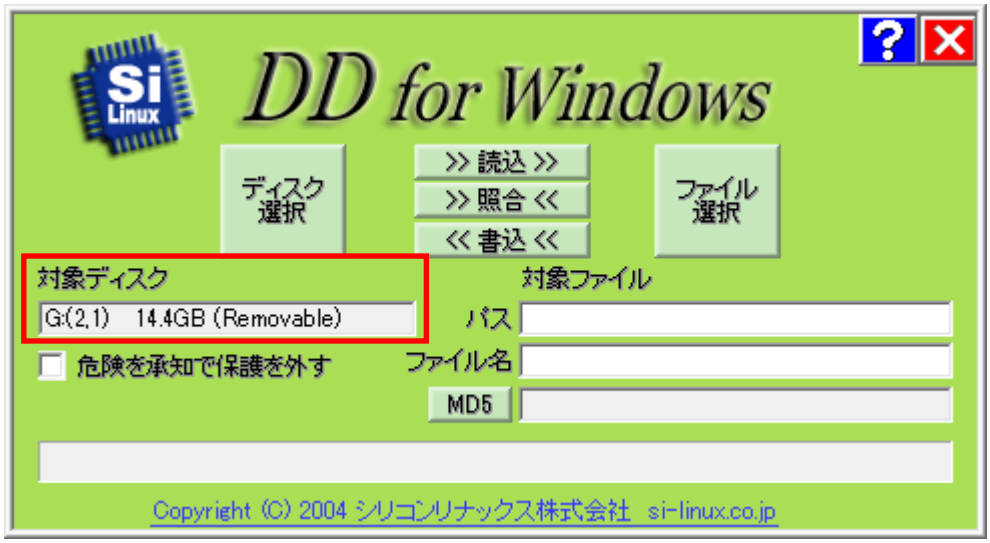


図 5-1-1-1. DD for Windows

⑧　「ファイル選択」ボタンを押してください。
ファイル選択画面が開くので、手順④で解凍した「RecoveryUSBImage.ddi」を選択してください。

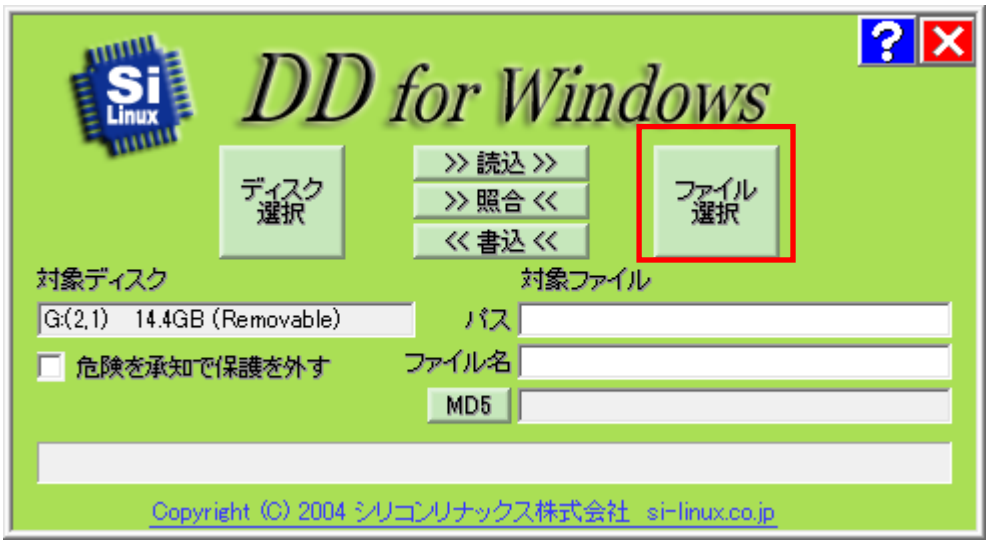


図 5-1-1-2.　ファイル選択

⑨　「対象ファイル」の項目に RecoveryUSBImage.ddi が表示されたことを確認してください。

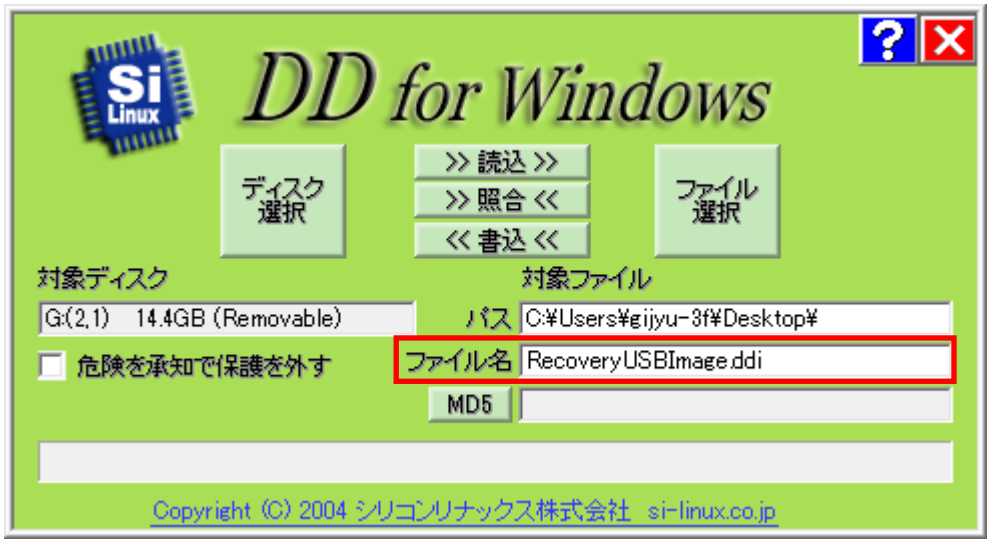


図 5-1-1-3.　対象ファイル

⑩　「<<書込<<」ボタンを押してください。
USB メモリへ書き込みが始まります。
書き込みが完了するまでお待ちください。

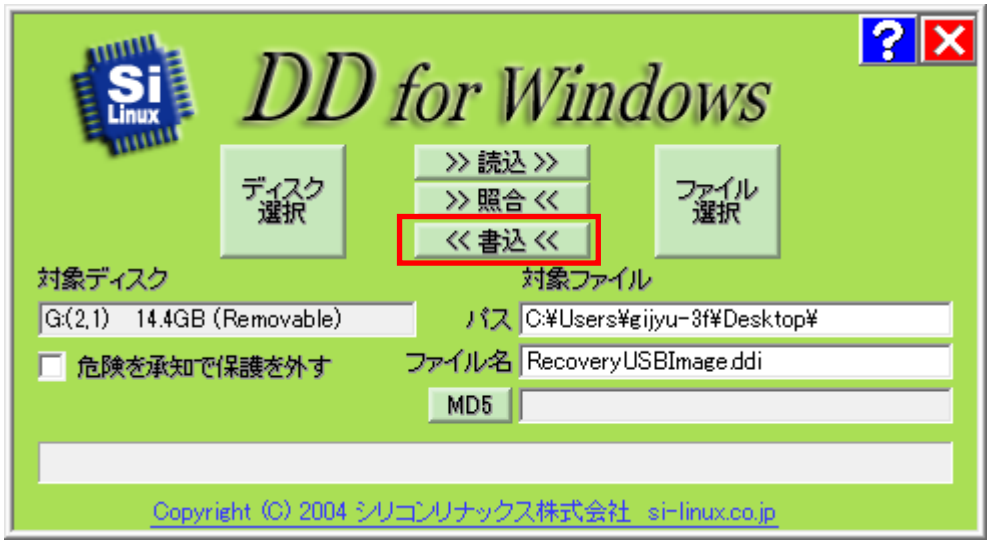


図 5-1-1-4.　書き込み開始

以上でリカバリ USB の作成は完了です。
リカバリ USB は一度作成すれば次回以降も使用することができます。

５－１－２　リカバリ USB 起動

リカバリ USB を起動させる前に、本体に接続されている LAN ケーブル、ストレージ（USB メモリ、SD カードなど）を取り外してください。サブストレージ（mSATA2）を接続している場合は、サブストレージを取り外してください。

リカバリ USB の起動にはリカバリ USB の他に以下のものを用意する必要があります。

- USB キーボード
- USB マウス
- 8GByte 程度の空き容量がある USB 接続可能なストレージメディア(USB メモリなど)

● リカバリ USB 起動手順

① LAN ケーブルが接続されている場合は、LAN ケーブルを取り外してください。
② リカバリ USB を本体に接続します。
③ USB キーボード、USB マウスを接続します。
④ 電源を入れます。BIOS 起動画面が表示されたところで[F2]キーを押し、BIOS 設定画面を表示させます。

⑤ BIOS 設定画面が表示されたら、[Boot]メニューを選択します。(図 5-1-2-1)
⑥ [USB HDD](接続した USB メモリ)を[ATA HDD0](m-SATA メインストレージ)よりも上に設定します。

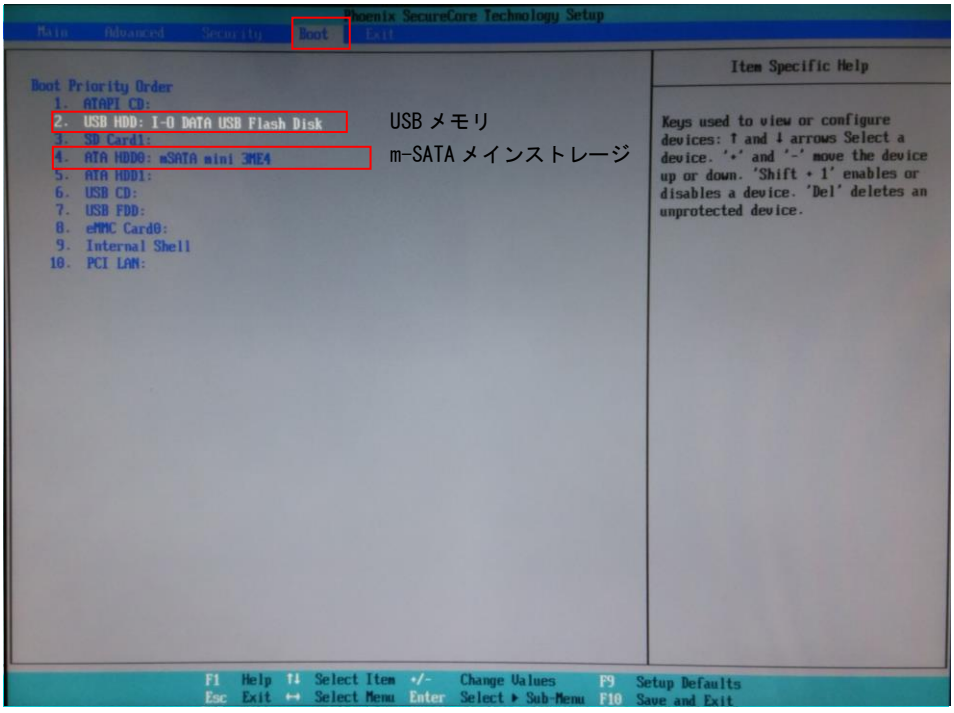


図 5-1-2-1. Boot デバイスの選択

⑦　[Exit]メニューを選択します。
⑧　[Exit Saving Changes]を実行し、設定を保存して終了します。

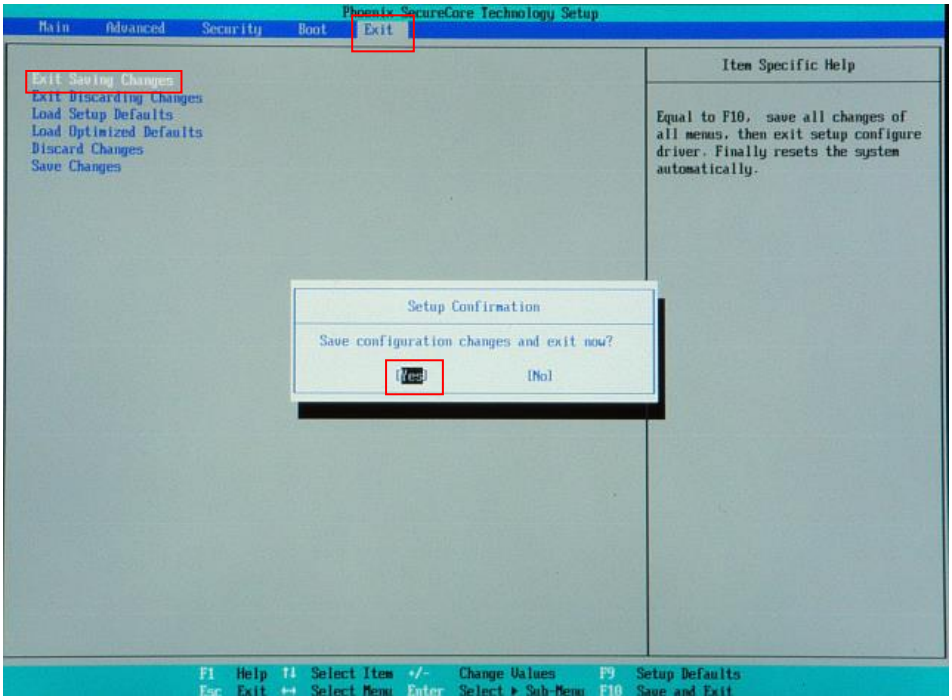


図 5-1-2-2. BIOS 設定 保存と終了

⑨　再起動し、正常にリカバリ USB から起動すると図 5-1-2-3 のリカバリメイン画面が表示されます。

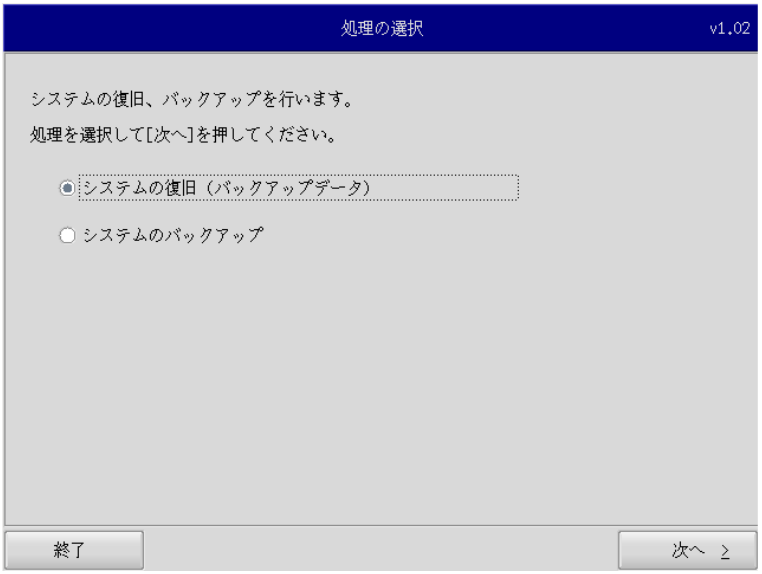


図 5-1-2-3. リカバリメイン画面

５－１－３　リカバリ作業

リカバリメイン画面から処理を選んでリカバリ作業を行います。

- システムの復旧（バックアップデータ）
- システムのバックアップ

リカバリ作業の詳細は、「5-2 システムの復旧（バックアップデータ）」、「5-3 システムのバックアップ」を参照してください。

５－１－４　リカバリ後処理

リカバリ作業が終わったら、通常使用のために BIOS 設定を戻します。

● リカバリ後処理手順

① 電源を入れ、BIOS 起動画面が表示されたところで[F2]キーを押し、BIOS 設定画面を表示させます。
② BIOS 設定画面が表示されたら、[Boot]メニューを選択します。(図 5-1-4-1)
③ [ATA HDD0](m-SATA メインストレージ)を[USB HDD](接続した USB メモリ)よりも上に設定します。

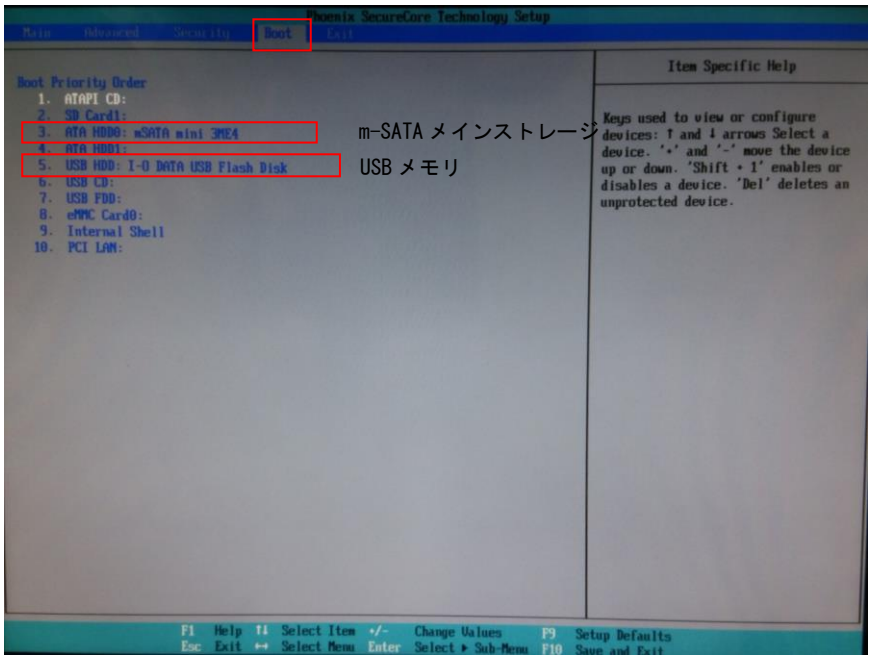


図 5-1-4-1. Boot デバイスの再設定

④ [Exit]メニューを選択します。
⑤ [Exit Saving Changes]を実行し、設定を保存して終了します。

５－２　システムの復旧（バックアップデータ）

工場出荷イメージをメインストレージ（mSATA1）に書込むことで、システムを工場出荷状態に復旧することができます。

また、「システムのバックアップ」で作成したバックアップファイルを使用して、メインストレージ（mSATA1）をバックアップファイルの状態に復旧させることができます。

※　システムを工場出荷状態へ復旧するとメインストレージにあるデータはすべて消えてしまいます。必要なデータがある場合は、復旧作業を行う前に保存してください。

※　バックアップファイルは、必ず対象となる本体で作成されたものを使用してください。他の本体のバックアップファイルでは動作しないので注意してください。

※　バックアップデータで復旧を行うとメインストレージのデータは、バックアップファイルの状態に戻ります。必要なデータがある場合は、復旧作業を行う前に保存してください。

●システムの復旧（バックアップデータ）の手順

① LAN ケーブルが接続されている場合は、LAN ケーブルを取り外してください。
また、工場出荷状態への復旧を行う場合、8GByte 程度の空き容量がある USB 接続可能なストレージメディア(USB メモリなど)にリカバリ DVD 内の工場出荷時イメージファイルをコピーしておいてください。
工場出荷時イメージファイルはリカバリ DVD の以下のフォルダに格納されている xxxxx.img ファイルです。
[リカバリ DVD]¥Recovery¥Image¥

② USB メモリ、SD カードなどのストレージメディアが接続されている場合は、ストレージメディアを取り外してください。

③ 「5-1-2　リカバリ USB 起動」を参考にリカバリ USB を起動させます。

④ リカバリメイン画面（図 5-1-2-3）で[システムの復旧（バックアップデータ）]を選択し、[次へ]ボタンを押します。

⑤　メディアの選択画面(図 5-2-1)が表示されます。コピー先となるメディアを選択し、「次へ」ボタンを押します。

図 5-2-1. メディア選択画面

⑥　メディアとパーティション選択画面（図 5-2-2）が表示されます。あらかじめ用意しておいたストレージメディアを本体に接続し、[メディア情報更新]ボタンを押してください。ストレージメディアのパーティションを選択し、[次へ]ボタンを押します。

ストレージメディアの認識には少し時間がかかります。ストレージメディアを接続してすぐに[メディア情報更新]ボタンを押すと、目的のメディア情報が現れないことがあります。この場合は、1 分程度待って再度、[メディア情報更新]ボタンを押してみてください。

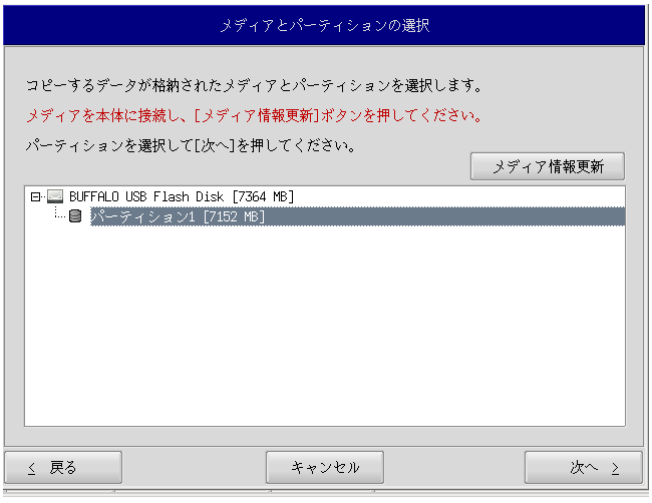


図 5-2-2. メディアとパーティション選択画面

⑦　フォルダ選択画面（図 5-2-3）が表示されます。［参照］ボタンを押します。

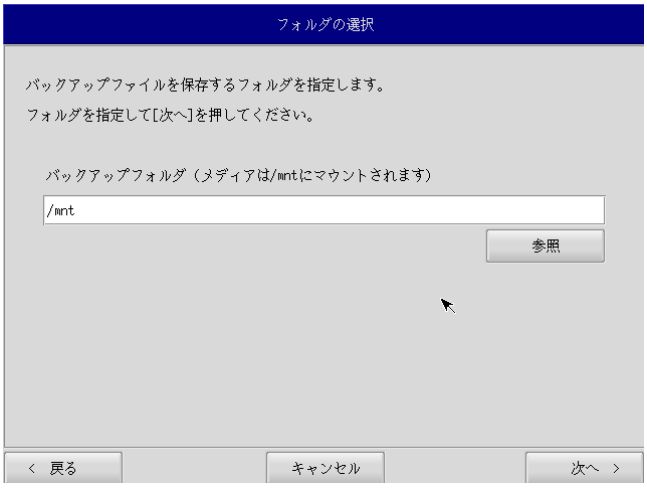


図 5-2-3. フォルダ選択画面

⑧　ファイル参照画面（図 5-2-4）が表示されます。接続したストレージメディアは、/mnt にマウントされていますので、/mnt 以下から目的のファイルを探してください。[OK]を押すとファイル選択画面にもどります。

※　USB メモリの¥backup¥xxxxxx.img というバックアップファイルを指定する場合
/mnt/backup/xxxxxx.img
を指定します。

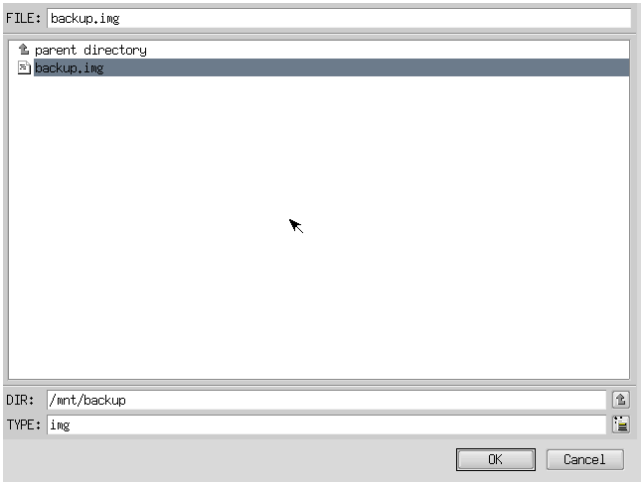


図 5-2-4. ファイル参照画面

⑨　ファイル参照画面（図 5-2-4）で指定したバックアップファイルが入力されていることを確認します。［次へ］ボタンを押します。

⑩　コンペア処理の選択画面(図 5-2-5)が表示されます。
データ書き込み時のコンペア処理の有無を選択します。

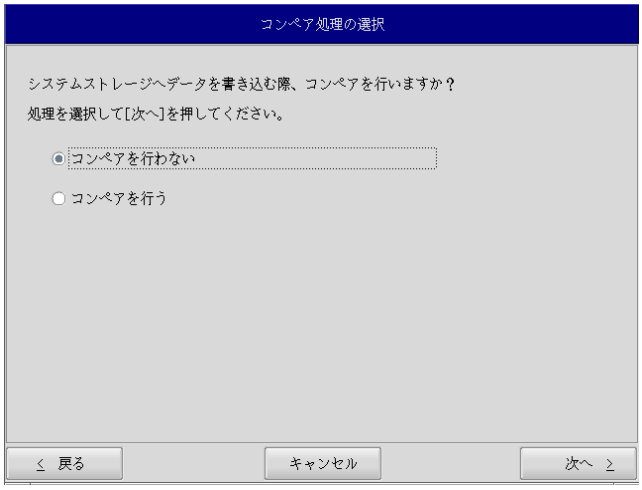


図 5-2-5. コンペア処理選択画面

⑪　確認画面(図 5-2-6) が表示されます。メディア、パーティション、バックアップファイルを確認します。[次へ]ボタンを押します。

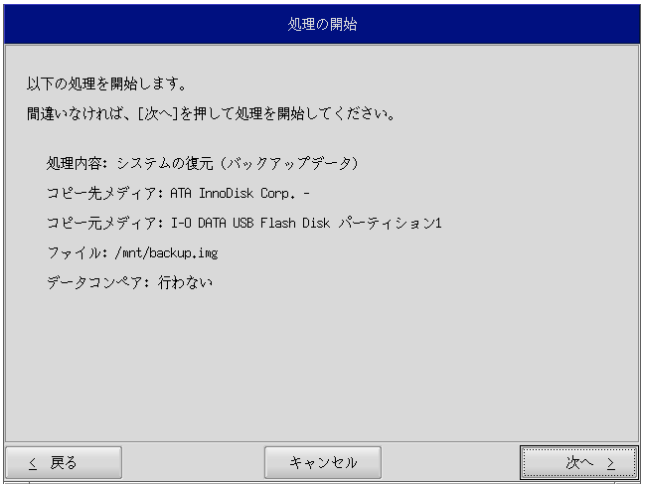


図 5-2-6. 確認画面

⑫　実行中画面（図 5-2-7）が表示され、処理が開始されます。実行中はリカバリ USB メモリ、保存メディアを外さないでください。また、電源を落とさないようにしてください。

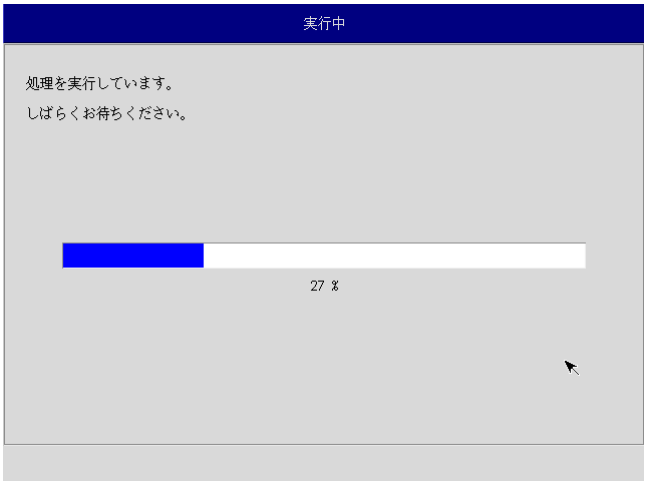


図 5-2-7. 実行中画面

⑬　終了画面（図 5-2-8）が表示されるとバックアップファイルの書き込みは完了です。[終了]ボタンを押して電源を落とし、リカバリ USB メモリ、保存メディアを外します。

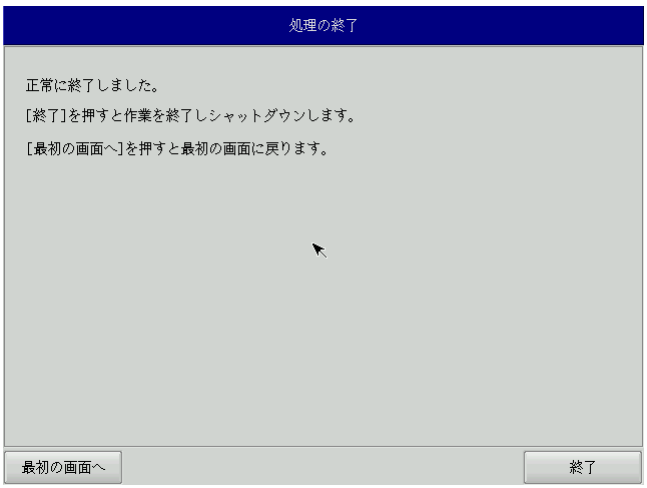


図 5-2-8. 終了画面

⑭　電源を入れ、BIOS 起動画面が表示されたところで[F2]キーを押し、BIOS 設定画面を表示させます。
⑮　デスクトップが表示されて正常に起動すれば、システム復旧は完了です。

※　システムを工場出荷状態へ復旧する場合、一度目の起動時にシステム再起動を求められる場合があります。この場合は指示に従い再起動してください。

５－３　システムのバックアップ

メインストレージ（mSATA1）の状態をファイルに保存します。バックアップファイル保存のために外部メモリとして、USB メモリ、SD カードなどが必要となります。外部メモリの空き容量は、バックアップファイルの保存に十分な空き容量が必要となります。安全のためメインストレージの容量以上の空き容量がある外部メモリを用意してください。

バックアップソフトにフォーマット機能はありません。外部メモリはあらかじめ Windows、Linux などでフォーマットしてください。対応しているファイルシステムは、NTFS、EXT2、EXT3 となります。

※　バックアップファイルのサイズが 4GByte を超える可能性があるため、FAT、FAT32 ファイルシステムは使用しないでください。

※　バックアップファイルのサイズは、システムの状態によって変化しますので注意してください。

※　作成されたバックアップファイルは、バックアップ作業を行った本体でのみ動作します。同じ型の本体であっても、他の本体では動作しませんので注意してください。

●システムのバックアップの手順

① LAN ケーブルが接続されている場合は、LAN ケーブルを取り外してください。
② USB メモリ、SD カードなどのストレージメディアが接続されている場合は、ストレージメディアを取り外してください。
③ 「5-1-2　リカバリ USB 起動」を参考にリカバリ USB を起動させます。
④ リカバリメイン画面（図 5-1-2-3）で[システムのバックアップ]を選択し、[次へ]ボタンを押します。
⑤ メディアの選択画面(図 5-3-1)が表示されます。コピー先となるメディアを選択し、[次へ]ボタンを押します。

図 5-3-1. メディア選択画面

⑥　メディアとパーティション選択画面（図 5-3-2）が表示されます。本体にメディアを接続し、[メディア情報更新]ボタンを押してください。バックアップファイルを保存するメディアのパーティションを選択し、[次へ]ボタンを押します。

メディアの認識には少し時間がかかります。メディアを接続してすぐに[メディア情報更新]ボタンを押すと、目的のメディア情報が現れないことがあります。この場合は、1 分程度待って再度、[メディア情報更新]ボタンを押してみてください。

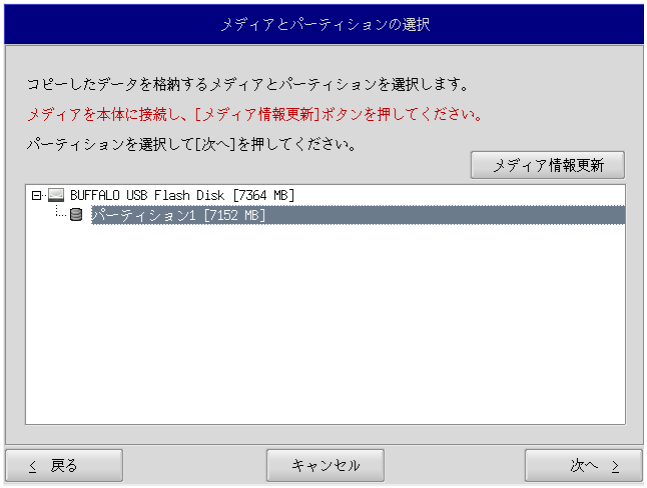


図 5-3-2. メディアとパーティション選択画面

⑦　フォルダ選択画面（図 5-3-3）が表示されます。［参照]ボタンを押します。

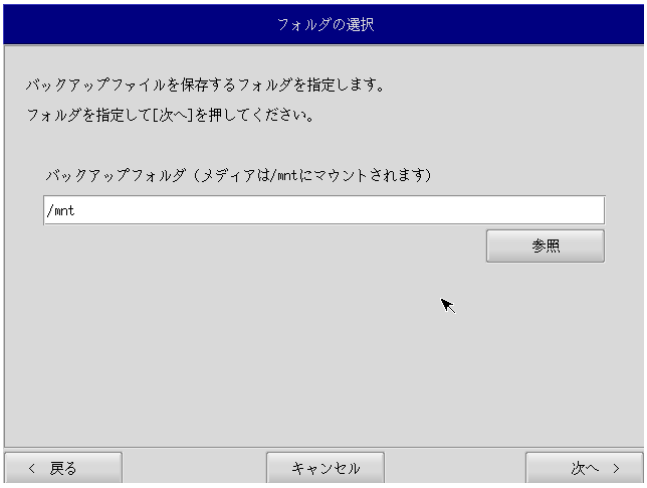


図 5-3-3. フォルダ選択画面

⑧　フォルダ参照画面（図 5-3-4）が表示されます。②で接続したパーティションは、/mnt にマウントされますので、/mnt 以下のフォルダを選択してください。[OK]を押すとフォルダ選択画面にもどります。

※　USB メモリに backup というフォルダがあり、このフォルダに保存する場合
/mnt/backup
を指定します。

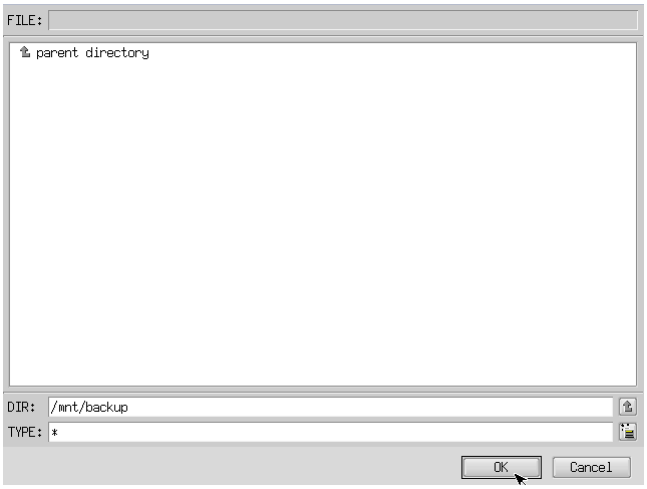


図 5-3-4.　フォルダ参照画面

⑨　フォルダ選択画面（図 5-3-3）で指定したバックアップフォルダが入力されていることを確認します。[次へ]ボタンを押します。

⑩　コンペア処理の選択画面(図 5-3-5)が表示されます。
データ書き込み時のコンペア処理の有無を選択します。

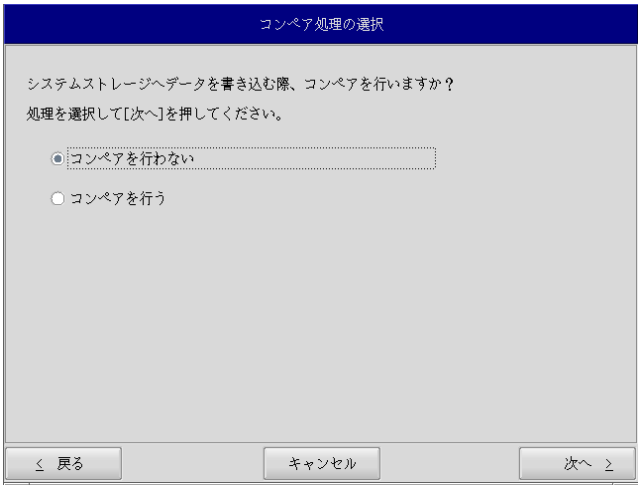


図 5-3-5.　コンペア処理選択画面

⑪　確認画面（図 5-3-6）が表示されます。メディア、パーティション、保存ファイルを確認します。[次へ]ボタンを押します。

※　保存ファイル名は、現在時刻から自動生成されます。

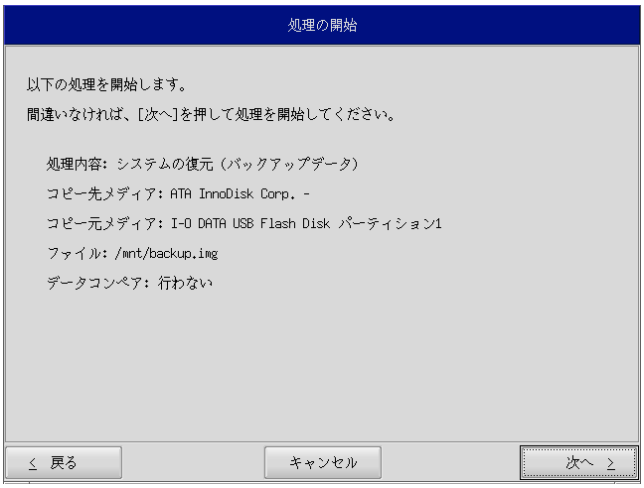


図 5-3-6. 確認画面

⑫　実行中画面（図 5-3-7）が表示され、処理が開始されます。実行中はリカバリ USB メモリ、保存メディアを外さないでください。また、電源を落とさないようにしてください。

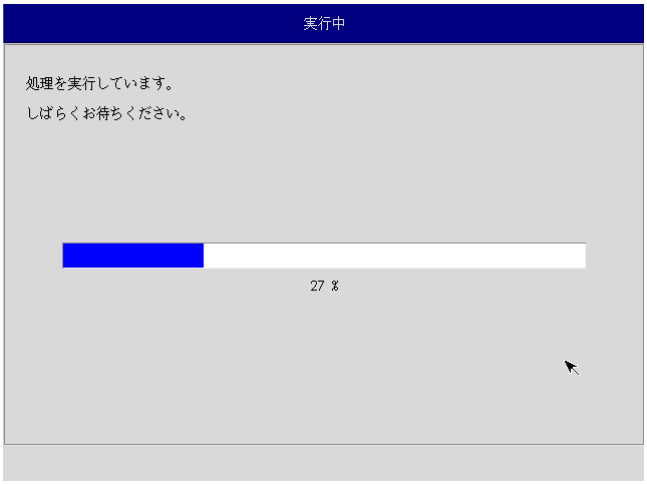


図 5-3-7. 実行中画面

⑬　終了画面（図 5-3-8）が表示されるとバックアップ作業は完了です。[終了]ボタンを押して電源を落としてください。

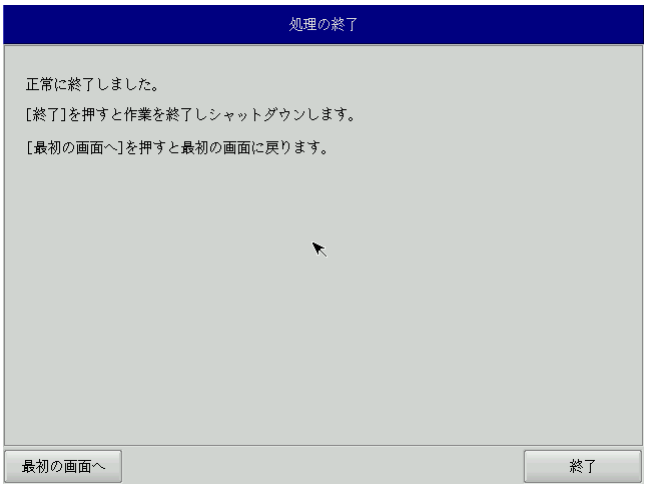


図 5-3-8.　終了画面

⑭　電源を入れ、BIOS 起動画面が表示されたところで[F2]キーを押し、BIOS 設定画面を表示させます。
⑮　デスクトップが表示されて正常に起動すれば、システム復旧は完了です。

※　システムを工場出荷状態へ復旧する場合、一度目の起動時にシステム再起動を求められる場合があります。この場合は指示に従い再起動してください。

付録

Ａ－1　参考文献

●「ふつうのLinuxプログラミング Linuxの仕組みから学べるGCCプログラミングの王道」
著者　青木 峰郎
発行所　ソフトバンク パブリッシング
発行年　2005年

●「How Linux Works Linuxの仕組み」
著者　Brian Ward
訳　吉川 典秀
発行所　毎日コミュニケーションズ
発行年　2006年

●「Linuxデバイスドライバ 第3版」
著者　Jonathan Corbet
Alessandro Rubini
Greg Kroah-hartman
訳　山崎 康宏
山崎 邦子
長原 宏治
長原 陽子
発行所　オライリー・ジャパン
発行年　2005年

このマニュアルについて

(1)本書の内容の一部又は全部を当社からの事前の承諾を得ることなく、無断で複写、複製、掲載することは固くお断りします。
(2)本書の内容に関しては、製品改良のためお断りなく、仕様などを変更することがありますのでご了承下さい。
(3)本書の内容に関しては万全を期しておりますが、万一ご不審な点や誤りなどお気付きのことがございましたらお手数ですが巻末記載の弊社までご連絡下さい。その際、巻末記載の書籍番号も併せてお知らせ下さい。

７７Ｇ０５００７Ｆ　２０１９年　１月　第６版
７７Ｇ０５００７Ａ　２０１８年　７月　第１版

株式会社アルゴシステム

本社
〒５８７－００２１　大阪府堺市美原区小平尾６５６番地
ＴＥＬ（０７２）３６２－５０６７
ＦＡＸ（０７２）３６２－４８５６

ホームページ　http://www.algosystem.co.jp