NORTi File System

Version 4

ユーザーズガイド

2021 年 5 月 第 12 版

株式会社ミスポ

第12版(本版)で改訂された項目

ページ	内容
7	未実装だった第 2FAT のサポートを特長から削除
10	"NORTi File System Version 4 補足説明書"を"リリースノート"に訂正
12	デバッグ情報なしライブラリの廃止に伴い、その説明や SuperH での例を削除
13	nofshpc.h/.c と noftide.h/.c の廃止や trueide.h/.c の追加に伴う見直し
15	チェックディスク機能の詳しい説明と、fspsubr.c を追加
16	SH7750 でのサンプルプログラムのファイルの例を削除
49	rename()に漏れていたディレクトリ名変更の説明を追加

第11版で改訂された項目

ページ	内容
13,14	「2.3 デバイスドライバ」を全面修正
16	「2.5 サンプルプログラム」付属ファイルの説明や例を修正
17	「2.6 コンフィグレーション」FTP サーバが使用するリソース追記
18	サンプルでは nonecfg.c がコンフィグレーションヘッダをインクルードする役目であると明記
21,22	File System Ver.1 からの移行手順を追記
23,24	File System Ver.2 からの移行手順を追記
25	「2.10 Windows FAT ファイルシステムとの互換性」を追加
73	「ATA ドライバの移植」を「PC カードアクセス関数の移植」に改名
73,74	インターフェース形式毎にどのように関数を実装すべきかの記述を追加。記述漏れの関数の説明を追加

第10版で改訂された項目

ページ	内容
8	ANSI 準拠の API に fgetpos,fsetpos を追加

9	未実装の ANSI 準拠 API から fsetpos,fgetpos を削除
36	2 ギガバイトを超えるファイルの位置決めについての補足を追記
37	2 ギガバイトを超えるファイルの位置取得についての補足を追記
38-39	fsetpos,fgetpos についての記述を追加
62	補足を追加 dformat をサポートする/しないドライバについて
----	「サーバー」→「サーバ」、「割込み」→「割り込み」 「下さい」→「ください」など表記のバラつきを統一

第9版で改訂された項目

ページ	内容
18,52	年の指定は1980年ではなく1900年からの年数であると訂正
52	年の値は80以上の値を指定する必要があることを注記

第8版で改訂された項目

ページ	内容
----	改訂項目を新しい順に並べ替え
9	ドキュメントについてのお願いを追記
12	MR-SHPC は SuperH 専用であることを注記
22,23	disk_ini に、disk_mount 時の全セクタチェック指定を追加
22	複数ドライブでは cycid=0 と指定できない制限について追記
46	readdir で取得できる direntx 構造体のエンディアンの説明を追記
62	ディスクドライバ関数のパラメータ説明漏れを修正

第7版で改訂された項目

ページ	内容
19	誤植を修正(fread を追加)
18,46	誤植を修正(1900 を 1980 に)

第6版で改訂された項目

ページ	内容
11	SH-2 リトルエンディアン用ライブラリを追加
45,47	誤植を修正(fopen から opendir へ)
61	エラーコード EV_FILEFULL/EV_FREESECT を追加

第5版で改訂された項目

ページ	内容
6	チェックディスク機能の追加予定を削除
10	ALL マクロの説明を変更
14	チェックディスク機能の未サポートの記述を削除
34	fflush(NULL)の未サポートの記述を削除

第4版で改訂された項目

ページ	内容
6	CompactFlash のサンプルドライバの付属を削除
20	ATA ドライバに関する記述を削除

第3版で改訂された項目

ページ	内容
6	NFS 対応予定から予定を削除
9	サンプルボード名称を MS7709A から MS7750S に変更 コンパイラ略称例を SHC7 から SHC9 に変更
15	SHC5/6/7 から SHC7/8/9 に変更
17	LFS_UNICODE マクロの未サポートの記述を削除

目　次

第１章 概要

1.1 特長

NORTi File System は、不意な電源断からディスクのデータ破損を極力回避できる機能を盛り込んだ、組み込みシステム専用の DOS 互換ファイルシステムです。μITRON 仕様リアルタイム OS「NORTi」、および、他の NORTi 用ソフトウェアコンポーネントと「NORTi File System」との組み合わせは、組み込みシステム開発に高い効率と自由度とをもたらします。

- DOS 互換 FAT12/16/32 に対応
- ロングファイル名(VFAT)をサポート
- 階層ディレクトリをサポート
- ANSI 準拠のファイル入出力関数、および、POSIX 準拠のディレクトリ操作関数
- FAT のキャッシュとキャッシュの自動保存機能により、高速性と安全性を両立
- チェックディスク(chkdsk)による FAT とディレクトリエントリの修復機能
- CRC でバックアップデータの正当性チェックを行う RAM ディスクドライバ付属
- ATA コマンドを利用した CompactFlash 用のドライバが付属
- 対応プロセッサ/対応コンパイラ別の動作確認済みライブラリで提供
- 全ソースコードが付属
- プロジェクトライセンス制で組み込みロイヤリティフリー

※ CompactFlash および CF はサンディスク社の商標です。
※ CompactFlash 以外のメディアへの対応は、別途、有償にて承ります。
※ ボード上に実装された Flash メモリをディスクとする機能には非対応です。

1.2 ファイルシステムのバージョン

ミスポからは、次の３種類のファイルシステムがリリースされており、本書では、Ver.1、Ver.2、Ver.4 と呼んで、これらを区別します。

Ver.1：NORTi 付属サンプルファイルシステム
Ver.2：HTTPd for NORTi 付属 File System
Ver.4：NORTi File System Version 4（本ファイルシステム）

「File System Ver.1」は、NORTi Version 4、および、NORTi Simulator にサンプルとして標準で付属しており、次の仕様となっています。

- DOS 互換 FAT12/16 に対応
- ロングファイル名(VFAT)非サポート、階層ディレクトリ非サポート
- FAT のキャッシュなし
- RAM ディスクの CRC チェックや、chkdsk による修復機能無し

「File System Ver.2」は、HTTPd for NORTi に標準で付属しており、Ver.1 に対して階層ディレクトリのサポートが追加されています。

1.3　ファイルシステム API 一覧

アプリケーション・インターフェース(API)の一覧を以下に示します。

初期化用の API

file_ini	ファイルシステムの初期化
disk_ini	ディスクドライバの初期化
disk_cache	ディスクキャッシュの設定

ANSI 準拠の API

fopen	ファイルのオープン
fclose	ファイルのクローズ
fgetc	ファイルから１文字読出し
fputc	ファイルへ１文字書込み
fgets	ファイルから１行分の文字列読出し
fputs	ファイルへ１行分の文字列書込み
fread	ファイルからブロック読出し
fwrite	ファイルへブロック書込み
fflush	ファイルのフラッシュ
fseek	ファイル読み書き位置の移動
ftell	現在のファイル読み書き位置を取得
feof	ファイルの終わりを検出
ferror	エラー情報の取得
clearerr	エラー情報のリセット
remove	ファイルの削除
rename	ファイル名やディレクトリ名の変更
fgetpos	ファイル読み書き位置の取得
fsetpos	ファイル読み書き位置の設定

POSIX 準拠の API

mkdir	ディレクトリの作成
rmdir	ディレクトリの削除
opendir	ディレクトリのオープン
closedir	ディレクトリのクローズ
readdir	ディレクトリ情報の読出し

独自拡張の API

fgetattr	ファイルの属性を取得
fsetattr	ファイルの属性を変更
fgetsize	ファイルサイズの取得
fsetsize	ファイルサイズの変更
fgetmtime	ファイルまたはディレクトリの作成時刻を取得
fsetmtime	ファイルまたはディレクトリの作成時刻を変更
returnname	ファイル名を返す
getdiskfree	ディスクの残容量を取得(4GB 未満)
getdiskfreex	ディスクの残容量を取得(4GB 以上)
getdisksize	ディスクの容量を取得
disk_mount	マウント
disk_unmount	アンマウント

dformat	ディスクのフォーマット
qformat	ディスクのクイックフォーマット

未実装の ANSI 準拠 API

fprintf	フォーマットデータをファイルに書込む
fscanf	ファイルからフォーマットデータを読出す
freopen	ファイルの再オープン
rewind	ファイル読み書き位置を先頭に移動
setbuf	入出力バッファの設定
setvbuf	モード指定による入出力バッファの設定

標準ライブラリとの関係

各 API は、C の標準ライブラリと名前が衝突しないよう、実際には、頭に my_ という 3 文字が付いています。それを、nofile.h のマクロ定義によって、my_ の付かない関数名に置き換えています。

```
(例)  FILE *my_fopen(const char *, const char *);
      #define fopen(f,m)   my_fopen(f,m)
```

したがって、nofile.h のインクルードを忘れた場合には、関数名の置き換えが行われず、C の標準ライブラリ関数の方がリンクされます。

nofile.h をインクルードしてある場合でも、未実装の API を記述すると、C の標準ライブラリ関数がリンクされてしまいます。リンクエラーになりませんが、機能しませんので注意してください。

1.4 ファイル名の指定方法と最大長

ディレクトリの情報を含むファイル名をパス名と呼びますが、本ファイルシステムは相対パスに対応しておらず、"d:¥aaaa¥bbbb¥nnnnnnnn.eee"形式の絶対パス(フルパス)での指定だけをサポートしています。

ディレクトリ名の区切りは'¥'ですが、Ｃ言語のコーディングでは'¥'はエスケープ文字なので、実際には'¥'を２つ重ねて"d:¥¥aaaa¥¥bbbb¥¥nnnnnnnn.eee"のように記述してください。重ねた'¥'はパス名の長さには含まれません。

ファイル名の長さに関して、本ファイルシステムでは次のマクロ名と値を定義しています。いずれも文字列の終わりを示す null 文字('¥0')までを含んだ長さなので、実際に指定できる文字数は、これらの値－１となります。

MAX_PATH	260	絶対パスの最大長
MAX_FNAME	255	ファイル名の部分の最大長(.拡張子を含まず)
MAX_DIR	247	ディレクトリ名の部分の最大長
MAX_EXT	255	拡張子の部分の最大長

MAX_PATH の制限により、ルートディレクトリには拡張子を含んで 256 文字のファイル名のファイルを置けますが、ディレクトリ階層が深くなるに従って、扱えるファイル名は短くなります。

第 2 章 導入

2.1 ファイル構成

フォルダ構造

本ファイルシステムの標準的なフォルダ構造は、次のとおりです。

```
NORTi
 ├ NOFILE
 │   ├ DOC ・・・・・・・・・・・・  ドキュメント
 │   ├ DRV ・・・・・・・・・・・・  デバイスドライバ
 │   │   ├ INC
 │   │   └ SRC
 │   ├ INC ・・・・・・・・・・・・  ファイルシステム本体のヘッダファイル
 │   ├ SRC ・・・・・・・・・・・・  ファイルシステム本体のソースファイル
 │   ├ LIB ・・・・・・・・・・・・  ファイルシステム本体のライブラリ
 │   │   └ XXX
 │   │       ├ YYY
 │   │       └ YYY
 │   └ UTL ・・・・・・・・・・・・  ユーティリティ
 │       ├ INC
 │       └ SRC
 └ SMP ・・・・・・・・・・・・・・・  評価ボード用サンプルプログラム
     └ XXX
         ├ BBB
         └ BBB
```

上記の *XXX* は対応 CPU コア略称、*YYY* は対応コンパイラ/バージョン略称、 *BBB* は評価ボード略称です。実際のフォルダ名は、DOC フォルダのリリースノートをご覧ください。

ドキュメント

NOFILE¥DOC フォルダには、次のファイルが収められています。

```
nofile.pdf ・・・・・・・・・・・・・  NORTi File System Version 4 ユーザーズガイド
NOFILE_xxx_yyy_nnn.txt ・・・  リリースノート
```

nofile.pdf(本書)は、各処理系(各プロセッサや各コンパイラ)で共通のマニュアルです。NORTi File System Version 4 を利用するための基本的な事項から各 API の詳細な解説までが記載されています。

NOFILE_*xxx_yyy_nnn*.txt には、対応 CPU/対応コンパイラに依存するフォルダ/ファイル構成等の説明、および、更新履歴が記載されています。ファイル名の *_xxx_yyy* の部分は、対応 CPU/対応コンパイラによって、*_nnn* の部分はバージョンによって異なります。

2.2　ファイルシステム本体

ヘッダファイル

NOFILE¥INC フォルダには、次のヘッダファイルが収められています。

nofile.h ·····	ファイルシステムのメインヘッダ
nofcfg.h ·····	ファイルシステムのコンフィグレーションヘッダ
nofsys.h ·····	ファイルシステム内部定義ヘッダ
nofutbl1.h ·····	Shift-JIS/Unicode 変換テーブル 1
nofutbl2.h ·····	Shift-JIS/Unicode 変換テーブル 2

nofile.h を、ファイルシステムの API を使用する全てのソースファイルでインクルードしてください。nofile.h には、各 API の関数宣言と、その関数コールに必要な構造体や定数等の定義が記述されています。

nofcfg.h には、ファイルシステムが使う各種リソースの情報が定義されています。ユーザー作成のソースファイルの１つ(通常は、カーネルや TCP/IP スタックのコンフィグレーションを記述してあるソースファイル)でインクルードしてください。nofcfg.h を２ヶ所以上でインクルードすると、二重定義のコンパイルエラーとなります。

nofsys.h、nofutbl1.h、nofutbl2.h は、ファイルシステムの内部定義であり、通常、ユーザー作成のソースファイルからインクルードする必要はありません。

ソースファイル

NOFILE¥SRC フォルダには、ファイルシステム本体のソースファイルが収められています。ファイルシステム本体はライブラリ化されていますので、通常は、これらのソースファイルをコンパイルしてユーザープログラムにリンクする必要はありません。

nofile.c ·····	ファイルシステム本体
nofsj2uc.c ····	S-JIS/UNICODE 変換処理
nofas2uc.c ····	ASCII/UNICODE 変換処理
noftime.c ·····	現在日時取得ダミー関数

noftime.c については、「2.7 現在日時の取得関数の実装」の節も参照してください。

使用されるコンパイラに対応したファイルシステムのライブラリが株式会社ミスポから提供されていない場合には、これらのソースファイルをコンパイルしてユーザープログラムにリンクしてください。(ただし、株式会社ミスポで動作を確認していないコンパイラでの使用は動作保証対象外となります。)

なお、nofile.c は、各 NOF_xxxxx マクロによってコンパイルする部分が選択されるようになっており、機能別にコンパイルを繰り返してライブラリへ結合してあります。こうすることによって、使用しない機能がユーザープログラムへリンクされるのを防いでいます。コンパイルオプションで ALL マクロを定義することで、nofile.c のファイル全体、つまり、全機能をまとめてコンパイルすることもできます。その他、適切なプロセッサコアやエンディアンを指定するコンパイルオプションを付けてください。

ライブラリ

本ファイルシステムはライブラリとして提供されるので、実際に使用する機能のみがユーザープログラムにリンクされ、コードやデータのサイズを自動的に節約することができます。

NOFILE¥LIB フォルダには、対応 CPU コア略称、その下に対応コンパイラ略称のサブフォルダがあり、そこに、ファイルシステム本体のライブラリと、それを生成するためのメイクファイル等が収められています。複数のバージョンのコンパイラに対応している場合には、対応バージョン別にサブフォルダが用意されています。

同じ CPU シリーズでも、複数の CPU コアに対応している場合には、コア別の異なるライブラリ名またはフォルダ名に。ビッグとリトルの両エンディアンに対応している場合は､エンディアン別の異なる次のようなライブラリ名になっています。

f4n????b.lib：ビッグエンディアン用ライブラリ
f4n????l.lib：リトルエンディアン用ライブラリ

ライブラリ名の????の部分は、対応 CPU コアやモード等によって異なります。コンパイラによっては、.lib 以外の拡張子の場合もあります。

2.3 デバイスドライバ

デバイスドライバとしては、ATA コマンドを使用する ATA ドライバ、それを TrueIDE モード専用としたドライバと、RAM ディスクドライバが付属します。

ATA ドライバは、ハードウェアに依存する PC カードアクセス関数を用意することにより、PC カードインターフェースでメディアにアクセスすることができます。CompactFlash, IDE-USB ブリッジ、HDD 等で使える TrueIDE 用ドライバは、ヘッダのカスタマイズのみで、アクセス関数を実装する必要はありません。

RAM ディスクドライバは、通常の RAM をディスク代わりに使えるようにするドライバです。メモリ空間を使用するだけなのでハードウェアには依存しません。

NOFILE¥DRV¥INC¥
- nofata.h ····· ATA ドライバのヘッダ
- nofpccd.h ····· PC カードアクセス関数のヘッダ
- trueide.h ····· TrueIDE 用ドライバのヘッダ

NOFILE¥DRV¥SRC¥
- nofata.c ····· ATA ドライバのソース
- nofgtfrm.c ···· ディスク情報取得関数のソース
- trueide.c ····· TrueIDE 用ドライバのソース
- nofram.c ····· RAM ディスクドライバのソース

ATA ドライバ

nofata.h と nofata.c は、ATA コマンドを使用して PC カードにアクセスする「ATA ドライバ」です。I/O アドレスやレジスタアクセス方法等、使用するハードウェアに合わせて PC カードアクセス関数を実装してください。そのポイントについては第 7 章で説明します。

nofgtfrm.c には、ATA ドライバ nofata.c からコールされるディスク情報取得関数が定義されています。この nofgtfrm.c は、カスタマイズ不要です。

記録メディアへのアクセス

記録メディアへのアクセス方法は、おおむね次の 3 種類に分類されます。

(1) PC カードインターフェース(PCMCIA コントローラ経由)

本格的なコントローラを搭載していて、アドレス空間のマッピングまで行わなければならないものです。設定が面倒ですが、活線挿抜に必要なカード電源の制御まで対応している場合があります。

(2) PC カードインターフェース(簡易型)

ハード的に I/O 空間やアトリビュート空間を割り当てていて、ハード的に決まっている空間をアクセスするだけで良い簡易的な PC カードインターフェースを搭載しているものを指します。ハードウェアの設計者により独自に簡易化されているため、活線挿抜に必要なカード電源の制御までは行えない場合があります。

(3) TrueIDE モード

最も簡単なインターフェースです。(2)よりさらに単純化されていますが、規格上抜き差しを前提としたものではないため、通常は挿入検知はダミー関数として対応しますが、ハードによっては独自に検知方法を追加されている場合があります。

RAM ディスクドライバ

nofram.c は、普通の RAM をディスクに見立てる RAM ディスクドライバです。バッテリーバックアップされた SRAM に RAM ディスク領域を割り当てることで、電源断でもディスクの内容が保持されます。nofram.c は、カスタマイズ不要です。

この RAM ディスクドライバでは、512 バイト当たり 4 バイトの CRC 領域が RAM ディスク領域から割当てられます。例えば、1M バイトの RAM ディスクでは、そのうちの 8K バイトが CRC 領域となります。

RAM ディスクドライバの初期化では RAM ディスク領域のクリアを行っていませんが、配列変数として定義した領域を RAM ディスク領域とすると、C のスタートアップルーチンでゼロクリアされてしまいます。バックアップが必要な場合には、プログラムで使用していない RAM 領域を絶対アドレスで指定してください。

キャッシュを備えたプロセッサで、電源断でも保持されるようにしたい場合は、キャッシュスルー(非キャッシュ)領域のアドレスとしてください。

RAM ディスクドライバの初期化では、全セクタの CRC チェックを行い、異常が検出された場合には、エラーを返します。エラーの場合でも自動的に RAM ディスク領域のクリアは行われませんので、dformat()によって明示的に RAM ディスクのフォーマットを行ってください。

RAM ディスクは簡単に複数ドライブ化できるようにしてあり、一方をワーク用、もう一方をバッテリバックアップされたドライブのように使い分けることが可能です。

2.4 ユーティリティ

NOFILE¥UTL フォルダには、ファイルシステムに関連したアプリケーションのソースファイルが収められています。

NOFILE¥UTL¥INC¥
- nofftpd.h ····· マルチセッション版 FTP サーバのヘッダ
- nofchkd.h ····· チェックディスク機能のヘッダ

NOFILE¥UTL¥SRC¥
- nofftpd.c ····· マルチセッション版 FTP サーバのソース
- nofchkd.c ····· チェックディスク機能のソース
- fspsubr.c ····· ファイルシステムの API の使用例

マルチセッション版 FTP サーバ

nofftpd.h と nofftpd.c は、本ファイルシステムの機能に合わせ、階層ディレクトリやロングファイル名に対応した FTP サーバです。また、同時に複数の FTP セッションを実行でき、GUI ベースの FTP クライアントや、複数の FTP クライアントからの同時接続にも対応しています。NORTi にサンプルとして標準で付属の FTP サーバの代わりに使用してください。

チェックディスク機能

ディスクへデータを書込んでいるタイミングでシステムがダウンすると、FAT の情報とディレクトリエントリの情報の整合がとれなくなる場合があります。このチェックと修復を行うのがチェックディスク機能です。

具体的には、ディレクトリエントリのサイズ情報を超えて FAT チェーンが続く場合は、それを未使用として切り離した上でデータを FOUND***ファイルへ移動します(***は 3 桁の数字)。

ディレクトリエントリのサイズ情報よりも FAT チェーンが短かった場合は、ディレクトリエントリの方を FAT チェーンに合わせて書き換え、データを FOUND***ファイルへ移動します。いずれも元のファイルは削除となるので、不整合は修復されますが、ファイルを復元できる訳ではないことに注意してください。

さらに FAT チェーンの重複のチェックも行っていて、重複していた場合は、先にチェックした方を正常なファイルとし、後からチェックした方を FOUND***ファイルにします。

なお、CompactFlash 等の記録メディアを使用していて、その内部で書き換えが行われている最中の電源断でメディア自体が破損した場合は、チェックディスクによる不整合の修復を行えません。

RAM ディスクでは、電源断からバックアップへの移行にハードウェア上の問題がない限り、チェックディスクによる不整合の修復は有効です。

2.5 サンプルプログラム

NORTi の SMP フォルダに収められているサンプルプログラムのソースは、マクロ定義やリンクするファイルを変更するだけでファイルシステムのサンプルとしても動作するように作られています。そのため、File System の SMP フォルダには、NORTi の SMP フォルダに対して追加となるプロジェクトファイルだけが格納されています。そのまま NORTi の同名フォルダへ上書きコピーしてください。

CompactFlash 使用の有無(CF マクロ)

通常のサンプルプログラムは、RAM 上にドライブ"A:"として RAM ディスクを作成します。CompactFlash ソケットを備えた評価ボード用のサンプルプログラムでは、コンパイル時に CF マクロを 1 に定義することにより、ドライブ"B:"として CompactFlash へもアクセスできるようになります。この場合、CompactFlash がデフォルトのドライブとなりますので、RAM ディスクへは、ドライブ名 "A:"を明示してアクセスしてください。

この CF マクロと下記の NOFILE_VER マクロは、サンプルプログラムのメインのソース net???.c とコンフィグレーション用の nonecfg.c に対して定義してください。メインのソースファイル名の???の部分は、CPU によって異なります。

ファイルシステムのバージョン(NOFILE_VER マクロ)

「1.2 ファイルシステムのバージョン」に記載のとおり、3 種類のファイルシステムがリリースされており、サンプルプログラムでは、次の NOFILE_VER マクロによって各ファイルシステムの非互換の部分に対応できるようになっています。

NOFILE_VER = 4 : 本ファイルシステム NORTi File System Version 4 に対応
NOFILE_VER = 3 : 未使用
NOFILE_VER = 2 : HTTPd for NORTi 付属の File System Version 2 に対応
NOFILE_VER = 1 : NORTi 付属サンプルの File System Version 1 に対応

あくまでもサンプルプログラム側の対応であり、ファイルシステム本体の動作を別バージョンに合わせたりするようなものではありません。

FTP サーバが使用するリソース

サンプルプログラムでは、本ファイルシステムに付属するマルチセッション版 FTP サーバを使用しています。セッション数は NFTP マクロで指定でき、未指定の場合のデフォルトは 1 です。

セッション数を増加させる場合は、セッション毎にタスクが 1 個、TCP 通信端点が 1 個使用されますで、カーネルや TCP/IP スタックのコンフィグレーション値が不足しないよう注意してください。

2.6 コンフィグレーション

コンフィグレーションヘッダのインクルード

ファイルシステム本体のコンフィグレーションを行うには、ファイルシステムが使うリソースの情報が定義されている nofcfg.h を、ユーザープログラムの１つでインクルードしてください。このインクルードの前に、コンフィグレーション用のマクロを定義することで、ユーザーのシステムに合わせたファイルシステムの構築を行うことができます。

サンプルプログラムでは nonecfg.c がその役目を担っています。その中でマクロ定義をしてコンフィグレーションの変更を行ってください。

この方法は File System Ver.4 でのみ設けられており、File System Ver.1 や Ver.2 から移行の際には、nofcfg.h のインクルードの追加が必要であることに注意してください。

ファイルシステムが使うリソース

本ファイルシステムでは、OS のリソース(オブジェクト)として、タスクを１個、メールボックスを１個、周期ハンドラを１個使用しています。

nofcfg.h には、ファイルシステムで使用するオブジェクトの個数として、以下のマクロが定義されていますので、システム全体のオブジェクト個数の集計を行う場合に利用してください。

```
#define LFS_NTSK        1       タスク個数
#define LFS_NSEM        0       セマフォ個数
#define LFS_NFLG        0       イベントフラグ個数
#define LFS_NMBX        1       メールボックス個数
#define LFS_NMBF        0       メッセージバッファ個数
#define LFS_NPOR        0       ランデブ用ポート個数
#define LFS_NMPL        0       可変長メモリプール個数
#define LFS_NMPF        0       固定長メモリプール個数
#define LFS_NDTQ        0       データーキュー個数
#define LFS_NMTX        0       ミューテックス個数
#define LFS_NISR        0       割り込みサービスルーチン個数
#define LFS_NCYC        1       周期ハンドラ個数
#define LFS_NALM        0       アラームハンドラ個数
```

ファイルシステムのタスク優先度(LFS_TSKPRI マクロ)

ファイルシステム本体は、ユーザープログラムとは独立したタスクとして実装されています。このファイルシステムのタスク優先度は、LFS_TSKPRI マクロを定義することによって指定できます。LFS_TSKPRI を定義しない場合、あるいは、LFS_TSKPRI が 0 の場合には、最初にファイルシステムの API を発行したタスクの優先度を引き継ぎます。

ファイルシステムのタスク優先度を高く(値では小さく)した場合、ファイル入出力動作は高速となりますが、それより優先度の低いタスクの実行が待たされます。

ディスク書込み遅延時間(LFS_WRTDLY マクロ)

本ファイルシステムでは、高速化のために、ディスクのセクタの割当てを管理している FAT(File Allocation Table)情報を RAM 上にキャッシュすることができます。また、オープンしたファイル毎に入出力バッファを持っていて、fputc()や fwrite()でこのバッファが一杯となるか、fclose()や fflush()が発行されるまではディスクへの書込みを行いません。

このキャッシュや入出力バッファの内容をディスクへ反映するタイミングを、LFS_WRTDLY マクロを定義することによって調整できます。バッファの大きさには無関係に動作します。具体的には、LFS_WRTDLY マクロで指定された時間だけファイルシステムの API が発行されなかった場合に、キャッシュされている FAT の内容や、全ファイルの未書込みデータがディスクへ書込まれ、ディレクトリのファイルサイズ情報も更新されます。

LFS_WRTDLY に小さな値を定義すると、キャッシュや入出力バッファとディスクの内容が直ぐに一致するため、不意なシステムダウンでのファイル破損の可能性が小さくなりますが、システムの負荷が増大します。

LFS_WRTDLY の設定値の単位はシステムクロックの割り込み回数で、1 以上の値を定義してください(ファイルシステムのタスク優先度が最低の場合にのみ、0 も設定可能)。LFS_WRTDLY を定義しない場合は、20/MSEC となります。

LFS_WRTDLY に TMO_FEVR を指定すると、遅延時間を待ってのディスク書込みは行われません。この場合、fclose()あるいは fflush()を行う前のデータは、不意なシステムダウンで失われます。

Shift-JIS/Unicode 変換テーブルの削減(LFS_UNICODE マクロ)

ロングファイル名をサポートする VFAT では、ファイル名が全て Unicode で表現されています。一方、C の文字列は、半角文字が ASCII で、全角文字が Shift-JIS で表現されるため文字コードの変換が必要です。

日本語のファイル名をサポートする必要が無い場合、LFS_UNICODE マクロを 0 に定義することで、ファイルシステムのライブラリに含まれる Shift-JIS/Unicode 変換テーブルが削除され、コードサイズを、約 31KB だけ節約することができます。LFS_UNICODE を定義しない場合、あるいは、LFS_UNICODE が 1 の場合には、ファイル名の Shift-JIS/Unicode 変換が行われます。

コンフィグレーションの例

```
#define LFS_TSKPRI      6         /* ファイルシステムタスク優先度 */
#define LFS_WRTDLY      60/MSEC   /* 書込み遅延時間 */
#define LFS_UNICODE     0         /* 日本語ファイル名を使用しない */
#include "nofcfg.h"               /* コンフィグレーションヘッダ */
```

その他のコンフィグレーション

ファイルシステムのタスク ID、メールボックス ID、周期ハンドラ ID は、空いている大きな ID 番号から自動割当てされます。各 ID 番号を自動割当てでなく固定したい場合には、ファイルシステム初期化関数 file_ini()とディスクドライバ初期化関数 disk_ini()で指定できます。

2.7 現在日時の取得関数の実装

NOFILE¥SRC フォルダにある noftime.c には、現在の日時を取得する関数 fs_get_tm がダミー定義されています。ファイルの作成や更新の際に日付と時刻の情報(タイムスタンプ)を付加する必要がある場合には、fs_get_tm 関数を、ユーザープログラム側に実装してください(ユーザープログラムで fs_get_tm 関数を定義することにより、ファイルシステムのライブラリに含まれるダミーの fs_get_tm 関数はリンクされなくなります)。

ボード上に、RTC(時計 IC)が搭載されている場合には、そのデータを読出して ANSI 互換の tm 構造体に年月日時分秒の情報を返すように実装してください。

```
int fs_get_tm(struct tm *ts)
{
    RTC から年月日時分秒をリード;
    if (異常)
        return 0; /* エラー */
    ts->tm_sec  = 秒(0～59);
    ts->tm_min  = 分(0～59);
    ts->tm_hour = 時(0～23);
    ts->tm_mday = 日(1～31);
    ts->tm_mon  = 月(0～11, 0 が 1 月);
    ts->tm_year = 年(1900 年からの年数);
    ts->tm_wday = 0;
    ts->tm_yday = 0;
    ts->tm_isdst = 0;
    return 1; /* 正常 */
}
```

tm 構造体の tm_wday(曜日)、tm_yday(1 月 1 日からの日数)、tm_isdst(夏時間フラグ)は、設定する必要がありません。

本関数がユーザープログラムに定義されない場合、ダミーの fs_get_tm 関数がコールされ、ファイルのタイムスタンプは、常に“1980 年 1 月 1 日 0 時 0 分 0 秒”となります。本関数がエラーを返した場合にも、この日時となります。ユーザープログラムへエラーは通知されません。

2.8 File System Ver.1 との互換性

File System Ver.1 の API

NORTi Version 4、および、NORTi Simulator 付属サンプルの「File System Ver.1」では、以下の API のみが実装されています。

file_ini, disk_ini, fopen, fclose, fseek, fgetc, fputc,
fgets, fputs, fread, fwrite, ftell, feof, remove, rename, dformat

File System Ver.4 の API は、File System Ver.1 に対して、file_ini と disk_ini を除き完全な上位互換性を保っています。

File System Ver.1 のファイル構成

File System Ver.1 は、次のファイルのみから構成されます。

NETSMP¥INC¥
  nonfile.h ····· ファイルシステムのヘッダファイル
NETSMP¥SRC¥
  nonfile.c ····· ファイルシステムのソースファイル
  nonramd.c ····· RAM ディスクドライバのソースファイル

File System Ver.1 からの移行手順

ヘッダファイルの変更

インクルードするヘッダファイルを下記のように変更してください。
#include “nonfile.h” → #include “nofile.h”

プロジェクトファイルの変更

プロジェクトファイルから nonfile.c を削除して、ビルドの対象外にしてください。ライブラリ指定に本ファイルシステムのライブラリを追加してください。

RAM ディスクドライバの変更

プロジェクトファイルから nonramd.c を削除して、ビルドの対象外にしてください。プロジェクトに nofram.c を追加して、ビルドの対象にしてください。
※nonramd.c をそのまま使うことも可能ですが、初期化時に CRC チェックを行っていないので、バックアップのチェック機能がありません。disk_ini でドライバを初期化する場合は、ゼロクリア済み領域を、RAM ディスク領域として渡してください。

API の差異を修正

disk_ini()の引数の個数が増えています。「第 3 章 ファイルシステム API 解説」を参照して、呼び出し箇所を修正してください。
file_ini()と disk_ini()をコールしていないソースファイルについては、API に互換性がありますので、ファイルシステム内部のデータを直接参照していない限り、"#include nonfile.h"のままとしておいても構いませんが、なるべくインク

ルードするヘッダファイル名は修正してください。

NETSMP¥SRC フォルダの FTP クライアントサンプル nonftpc.c や、TFTP サーバサンプル nontftp.c は、NOFILE_VER=4 と定義してコンパイルすると、nofile.h をインクルードするようになっています。

非互換 API の修正

API のうち、以下の使用方法については互換性がありませんので、代替となる API に修正してください。

API を使用する場面	Ver.1 の API	Ver.4 の API
ディレクトリのオープン	fopen(".", "r")	opendir()
ディレクトリ情報の読出し	fread()	readdir()
ディレクトリのクローズ	fclose()	closedir()

コンフィグレーション項目の設定

「2.6 コンフィグレーション」を参照し、コンフィグレーション用のマクロ定義と、nofcfg.h のインクルードを追加してください。サンプルプログラムでは nonecfg.c というソースでコンフィグレーションの設定を行っています。
このファイルは NOFILE_VER=4 の場合にも対応しています。

インクルードパスの追加

プロジェクトのインクルードパスの設定に下記のディレクトリを追加してください。
C:¥NORTi¥NOFILE¥INC
C:¥NORTi¥NOFILE¥DRV¥INC

2.9 File System Ver.2 との互換性

File System Ver.2 の API

HTTPd for NORTi に付属の「File System Ver.2」では、前ページの Ver.1 の API に加え、以下の API が追加されています。

disk_ini2, disk_mount, disk_unmount, makedir, removedir, dirlist, getdiskfree, alloc_sbuf

上記のうち、disk_mount と disk_unmount 以外の API は、本ファイルシステムと互換性がありません。

File System Ver.2 のファイル構成

File System Ver.2 は、次のファイルから構成されます。

```
NONFILE¥DOC
  nonfile.pdf ···  ユーザーズガイド
NONFILE¥INC¥
  nonfile.h ·····  ファイルシステムのヘッダファイル
  time.h ·····  時刻構造体の定義
NONFILE¥SRC¥
  nonfile.c ·····  ファイルシステムのソースファイル
  get_tm.c ·····  現在日時取得ダミー関数
  nonramd.c ·····  RAM ディスクドライバのソースファイル
```

File System Ver.2 からの移行手順

HTTPd for NORTi で使用している場合に、ファイルシステムを Ver.4 に移行する方法については別項目で説明します。
基本的な移行手順と注意点は、File System Ver.1 から移行の場合と同じです。
disk_ini()については、Ver.2 と Ver.4 は互換ですので修正は不要です。
下記の互換性のない API を使用している箇所は、修正をお願いします。

API を使用する場面	Ver.2 の API	Ver.4 の API
ディスクドライバの初期化	disk_ini2	disk_ini()
ディレクトリの生成	makedir()	mkdir()
ディレクトリの削除	removedir()	rmdir()
ディレクトリのオープン	fopen(".", "r")	opendir()
ディレクトリ情報の読出し	fread()	readdir()
ディレクトリのクローズ	fclose()	closedir()

getdiskfree()によりディスクの空き容量を取得している場合は、引数と戻値が異なりますので修正してください。
alloc_sbuf()に相当する機能はありません。

File System Ver.2 からの移行手順(HTTPD)

HTTPd for NORTi は、ユーザープログラムと共にコンパイルされることを前提にしています。そのコンパイルの際に、NOFILE_VER=4 を定義することにより、Ver.4 対応の API が使用されます。

・コンフィグレーション項目の設定

「2.6 コンフィグレーション」を参照し、コンフィグレーション用のマクロ定義と、nofcfg.h のインクルードを追加してください。
HTTPd に付属するサンプルプログラムでは httpcfg.c というソースでコンフィグレーションの設定を行っています。
このファイルは NOFILE_VER=4 の場合にも対応しています。

・インクルードパスの変更

プロジェクトに設定されているインクルードパスのうち、ファイルシステムに関連するものを下記のように変更・追加してください。

C:¥NORTi¥NONFILE¥INC → C:¥NORTi¥NOFILE¥INC
C:¥NORTi¥NOFILE¥DRV¥INC

2.10 Windows FAT ファイルシステムとの互換性

ファイル名として使用可能な文字や、長いファイル名(LFN)、短いファイル名(SFN)についての決まりはMicrosoft Windowsに従います。

WindowsのFAT仕様については、正確にはWindows 98の仕様とWindows NT系の仕様(Windows XPを含む)がありますが、本ファイルシステムはWindows 98仕様で実装されています。作成されたファイルは、WindowsXP等のNT系OSでも問題なくアクセスすることができます。

SFNではファイル名に小文字が含まれていることを禁止しているため、それらの文字が含まれる場合は、文字数としてはSFNの8.3形式に収まっていても、SFN上では大文字に変換し、LFNを生成します。(LFNに小文字のファイル名が入ります) また、LFNからSFNを作成するときに、名前が衝突しないように、ファイル名の一部を～(チルダ)と複数桁の数字に置き換えます。

NT系Windowsでは、小文字が含まれていても、すべてが小文字のSFNの場合は、LFNのディレクトリエントリを生成しないように修正が加えられています。[1]

NT系Windowsにも名前の衝突防止を考慮したSFNの生成が同様に実装されていますが、こちらはアルゴリズムが改訂されていてWindows 98と同じファイル名にはなりません。またそのアルゴリズムについてもWindows NT系のものは公開されていません。

これらの事情から、本ファイルシステムでは、正式に公開されているWindows 98仕様で実装しています。

[1] NT系ではシステムファイルに小文字だけのファイルが多く、それによるディレクトリエントリの浪費を嫌っての変更と言われています。

第３章　ファイルシステム API 解説

file_ini - ファイルシステムの初期化

形 式　int file_ini(T_FILE *f, int nfile, ID tskid, ID mbxid);

引 数　f　　T_FILE 構造体配列へのポインタ
nfile　同時にオープンするファイル数
tskid　ファイルシステムで使用するタスクの ID
mbxid　ファイルシステムで使用するメールボックスの ID

解 説　ファイルシステムの初期化を行います。全てのファイルシステム API の発行に先だって、1 回だけ file_ini()をタスクからコールしてください。

f には T_FILE 構造体配列へのポインタを指定してください。nfile には同時にオープンできるファイル数を指定してください。つまり T_FILE 構造体配列の要素数と同じ値を指定してください。

ファイルシステムの初期化ではタスクを 1 個生成します。タスク ID を明示したい場合には、その ID 番号を tskid に指定してください。tskid が 0 の場合は、自動的に ID が割り当てられます。

ファイルシステムの初期化ではメールボックスも 1 個生成します。メールボックス ID を明示したい場合には、その ID 番号を mbxid に指定してください。mbxid が 0 の場合は、自動的に ID が割り当てられます

戻 値　正常終了の場合は E_OK を返します。
エラーの場合は NORTi のシステムコールのエラーコードを返します。主なエラーは、タスク生成やメールボックス生成に伴うメモリ不足や ID 割り当ての失敗です。

例　disk_cache()の例を参照してください。

補 足　File System Ver.1 と Ver.2 では、file_ini()の第４引数がメールボックス ID ではなくセマフォ ID です。本ファイルシステムからファイル入出力を独立したタスクで行うようになってセマフォによる排他制御が不要となり、その代わりにタスク間通信用のメールボックスが使用されるようになりました。第４引数の意味は異なりますが、本ファイルシステムへ移行のためには、0 のまま修正不要です。

disk_ini - ディスクドライバの初期化

形 式　int disk_ini(T_DISK *d, const char *drv, DISK_FP func, UW addr, UW param, DISK_CALLBACK callback, ID cycid, int opt);

引 数

d	T_DISK 構造体へのポインタ
drv	ドライブ名("A:", "B:", …)
func	ドライバ関数へのポインタ
addr	ドライバ依存のパラメータ(アドレス等)
param	ドライバ依存のパラメータ(サイズ等)
callback	状態変化通知用コールバック関数へのポインタ
cycid	周期ハンドラ ID
opt	オプション指定

解 説　ディスクにアクセスするためのデバイスドライバを初期化します。複数のドライブがある場合には、引数を変えて disk_ini()をドライブ数分だけコールしてください。

ドライバの作業領域である T_DISK 構造体をユーザープログラム定義し、d に、そのポインタを指定してください。複数のドライブがある場合、それぞれ異なる T_DISK 構造体を指定してください。

drv にはドライブレター文字列を指定してください。"A:", "B:", …のように"A:"から開始して連続している必要はありません。任意のアルファベットが使用できます。大文字と小文字の区別はありません。

func にはドライバとする関数へのポインタを指定してください。付属の RAM ディスクドライバ(nofram.c)の場合は ramdisk、付属の CompactFlash 用の ATA ドライバ(nofata.c)の場合は flash_ATA となります。

addr と param はドライバにより指定方法が異なります。付属の RAM ディスクドライバでは、RAM ディスクとする領域の先頭アドレスとサイズを指定してください。付属の ATA ドライバでは未使用なので、0, 0 を指定してください。addr と param の内容はファイルシステム本体では関知していないので、ドライバとの取り決めで自由に使用できます。

ディスクのアクセスや挿抜などによる状態変化で、callback に指定したユーザー定義の関数が、ドライバの周期ハンドラからコールされます。このコールバック関数は「第 7 章 状態変化通知用コールバック関数」を参照して作成してください。この関数でアクセスランプを点滅させたり、ディスクが抜かれた場合の警告を出したりすることができます。コールバック関数を使用しない場合は NULL を指定してください。付属の RAM ディスクドライバでは未使用なので、NULL となります。

コールバック関数を用いる場合は、ドライバが周期ハンドラを生成します。周期ハンドラ ID を明示したい場合には、その ID 番号を cycid に指定してください。cycid が 0 の場合は、自動的に ID が割り当てられます。複数の CF/HDD ドライブが存在し、disk_ini を複数回コールする場合は、明示的に周期ハンドラ ID を指定して各ドライブを初期化してください。本バージョンでは、1 個の周期ハンドラで複数ドライブに対応する機能には対応しておりません。

付属の RAM ディスクドライバでは未使用なので、0 を指定してください。

opt には、下記の指定が可能です。

Bit1: disk_mount 時の全セクタチェック指定

0 を指定した場合、disk_mount 時に全セクタを読み出せるかのチェックを行います。1 を指定した場合、このチェックを省略します。

全セクタの読み出しには時間がかかりますので、大容量メディアを使用される場合は、1 を指定してチェックを省略することをお勧めします。

Bit0:　取り外し可能メディア指定

ディスクが取り外し可能ならば 1 を､取り外し不可能なら 0 を指定してください。

0 を指定したときは、disk_ini 呼び出し時に disk_mount まで自動的におこないます。1 を指定したときは、挿入を検知して、disk_mount をコールしていただく必要があります。

戻 値　**正常終了の場合は E_OK を返します。**
エラーの場合はドライバの戻値を返します。

例　disk_cache()の例を参照してください。

補 足　File System Ver.1 では、disk_ini()の第６引数以降 **callback, cycid, opt** がありません。File System Ver.1 は RAM ディスクにのみ対応していますので、本ファイルシステムへ移行のためには、第６引数以降には、NULL, 0, 0 を指定してください。

File System Ver.2 では第７引数(**cycid**)の用途が異なりますが、未使用(0)でしたので、本ファイルシステムへ移行のためには、0 のまま修正不要です。

disk_cache - ディスクキャッシュの設定

形 式　int disk_cache(T_DISK *d, void *buf, int ncache);

引 数　d　　T_DISK 構造体へのポインタ
buf　　キャッシュバッファへのポインタ
ncache　キャッシュの個数

解 説　ファイルシステムを高速化するためには、頻繁にアクセスされるディスクの FAT 領域を RAM 上にキャッシュすることが望ましいです。そのためのバッファを割り当て、キャッシュの使用を有効にます。

d には、先に disk_ini の第 1 引数 d として指定した T_DISK 構造体へのポインタを指定してください。

ユーザープログラムにキャッシュバッファ領域を T_CACHE 構造体配列として定義し、その領域へのポインタを、buf に指定してください。

ncache には T_CACHE 構造体配列の要素数を指定してください。最大個数は 255 で、255 を超える数を指定した場合、255 個に強制的に補正されます。推奨値は 5 から 10 です。

T_CACHE 構造体 1 つのサイズは、520 バイトです。T_CACHE 型を使用せずに任意の領域をキャッシュバッファとして buf に指定することも可能です。その場合、キャッシュバッファはロングワード境界のアドレスで開始し、520 バイトの整数倍のサイズとしてください。ncache にはキャッシュバッファ領域の総サイズを 520 で割った数を指定してください。

複数のドライブがある場合には、それぞれ異なる領域をキャッシュバッファとして指定してください。キャッシュバッファの領域や個数を途中で変更することはできません。なお、RAM ディスクの場合は、ディスクキャッシュの効果がありませんので、disk_cache()の実行は行わない方が良いです。

戻 値　正常終了の場合は E_OK を返します。エラー(キャッシュバッファサイズ不足)の場合は-1 を返します。

例　ファイルシステムとRAMディスクドライバとCompactFlash用のATAドライバを初期化し、CompactFlashへだけディスクキャッシュを割り当てます。

```
#define NFILE         6                /* 同時オープンするファイル数 */
#define NCACHE    5            /* ディスクキャッシュ個数 */
#define RAMDISK       0xAC200000       /* RAMディスク領域のアドレス */
#define RAMDISKSZ  0x100000            /* RAMディスク領域のサイズ */
T_FILE file[NFILE];
T_DISK disk[2];
T_CACHE cache[NCACHE];

void test(void)
{
    int r;

    /* ファイルシステム初期化 */
    r = file_ini(file, NFILE, 0, 0);
    if (r != E_OK) {
        /* error */
    }

    /* RAMディスクドライバ初期化 */
    r = disk_ini(&disk[0], "A:", ramdisk, RAMDISK, RAMDISKSZ, NULL, 0, 0);
    if (r != E_OK) {
        /* error */
    }

    /* CompactFlash用ATAドライバ初期化 */
    r = disk_ini(&disk[1], "B:", flash_ATA, 0, 0, NULL, 0, 0);
    if (r != E_OK) {
        /* error */
    }

    /* CompactFlashにディスクキャッシュを割り当て */
    r = disk_cache(&disk[1], cache, NCACHE);
    if (r != E_OK) {
        /* error */
    }
}
```

例では２つの T_DISK 構造体を配列として定義していますが、配列として連続している必要はなく、RAMディスクとCompactFlashとで別々の名前の構造体を定義することでも構いません。

RAM ディスク領域が、プログラムで使用している領域と重ならないように注意してください。キャッシュを内蔵したプロセッサでRAMディスク領域をバックアップする場合には、キャッシュスルー(非キャッシュ)領域のアドレスを指定してください。

fopen - ファイルのオープン

形 式　FILE *fopen(const char *path, const char *mode);

引 数　path　　パス名(フルパス指定)
　　mode　　ファイルアクセスモード

解 説　**path** には、"d:¥aaaa¥bbbb¥nnnnnnnn.eee"形式のドライブ名やルートからのディレクトリ名を含むフルパスでファイル名を指定してください。**path** の最大長は MAX_PATH で規定されます。ディレクトリ名の区切りは'¥'ですが、C言語のコーディングでは'¥'はエスケープ文字なので、実際には'¥'を２つ重ねて"d:¥¥aaaa¥¥bbbb¥¥nnnnnnnn.eee"のように記述してください。

"d:"はドライブ名で、"A:"、"B:"、"C:"等を指定してください。ドライブ名("d:"または"d:¥")を省略した場合は、最初に disk_ini()で初期化されたディスクが選択されます。**path** の長さは、省略したドライブ名を補った形式で計算されます。本ファイルシステムはカレントディレクトリの管理を行っていないので、相対パスによる指定はできません。

mode には、アクセスの種類を文字列で指定してください。

"r"　読出しモードでオープンします。指定されたファイルが存在しない場合はエラーになります。
"w"　書込みモードで空のファイルを開きます。指定されたファイルが既に存在する場合は、そのファイルの内容を破棄します。
"a"　追加書込み用にオープンします。指定されたファイルの末尾から書込みます。指定されたファイルが存在しない場合は、新規に作成します。
"r+"　読出しと書込みの両方のモードで開きます。指定されたファイルが存在しない場合はエラーになります。
"w+"　読出しと書込みの両方のモードで空のファイルを開きます。指定されたファイルが既に存在する場合は、そのファイルの内容を破棄します。
"a+"　読出しと追加の両方のモードで開きます。指定されたファイルが存在しない場合は、新規に作成します。

上記のアクセスの種類に加え、一般的なファイルシステムは、変換モードとして"b"(バイナリモード)か"t"(テキストモード)かを指定できますが、本ファイルシステムはテキストモードをサポートしていません。すなわち、ファイル入出力時に CL+LF⇔LF 変換は行われません。"b"を省略した場合には、バイナリモードと見なされます。"r+b"と"rb+"のように、"b"は先頭以外のどちらへでも記述できます。"rt"や"wt"のようにテキストモードでオープンしようとするとエラーとなります。

戻 値　**オープンしたファイルを管理する構造体へのポインタ(ファイルポインタ)を返します。以降のファイル操作では、このポインタでファイルを指定してください。エラーの場合は NULL が返ります。**

fclose - ファイルのクローズ

形 式　int fclose(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルをクローズします。

戻 値　正常終了の場合は 0 を返します。
エラーの場合は EOF(-1)を返します。

例　ファイルを"w"モードでオープンし、１バイトのデータを書込み、ファイルをクローズします。

```
void test(void)
{
    FILE *fp;

    if ((fp = fopen("A:¥¥test.txt", "w")) != NULL) {
        fputc('Z', fp);
        fclose(fp);
    }
}
```

fgetc - ファイルから１文字読出し

形 式　int fgetc(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルから１バイトのデータを読出します。

戻 値　読出した１バイトのデータを符号拡張せずに int へ変換した値を返します。
ファイルの終わり、またはエラーの場合は EOF(-1)を返します。

例　ファイルを"r"モードでオープンし、10 バイトのデータを読出してバッファに格納します。途中でファイルの終わりを検出した場合は、読出しを中止します。

```
void test(void)
{
    FILE *fp;
    int n, c;
    char buf[10];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;  /* error */
    for (n = 0; n < 10; n++) {
        c = fgetc(fp);
        if (c == EOF)
            break;
        buf[n] = c;
    }
    fclose(fp);
}
```

補 足　実際には、fgetc 関数をループ処理して複数回発行するよりも、fread 関数を使用した方が高速に読出すことができます。

fputc - ファイルへ１文字書込み

形 式　int fputc(int c, FILE *fp);

引 数　c　　書込む文字
　　fp　　ファイルポインタ

解 説　fp で指定されたファイルへ char 型に変換した c を書込みます。

戻 値　正常終了の場合は c を返します。エラーの場合は EOF(-1)を返します。

例　　ファイルを"w"モードでオープンし、10 バイトの文字列を書込みます。

```
void test(void)
{
    FILE *fp;
    int n, c;
    char *str = "0123456789";

    if ((fp = fopen("A:¥¥test.txt", "w")) == NULL)
        return;  /* error */
    for (n = 0; n < 10; n++) {
        c = fputc(str[n], fp);
        if (c == EOF)  /* error */
            break;
    }
    fclose(fp);
}
```

補 足　実際には、fputc 関数をループ処理して複数回発行するよりも、fwrite 関数を使用した方が高速に書込むことができます。

fgets - ファイルから１行分の文字列読出し

形 式　char *fgets(char *buf, int n, FILE *fp);

引 数　buf　　読出した文字列を受け取るバッファ
n　　バッファのサイズ
fp　　ファイルポインタ

解 説　fp で指定されたファイルから文字列を読出し、buf で指定されたバッファへ格納します。改行文字('¥n')を読出すまで、または、読出した文字数が n - 1 に等しくなるまで、あるいは、ファイルの終端に達するまで読出します。改行文字も buf へ格納されます。buf へ格納した文字列の最後に null 文字('¥0')が付加されます。

戻 値　正常終了の場合は buf を返します。ファイルの終わりに達し、かつ buf に１文字も格納されていない場合は NULL を返します。エラーが発生した場合も NULL を返します。

例　ファイルを"r"モードでオープンし、文字列を 1 行読出して、バッファに格納します。

```
void test(void)
{
    FILE *fp;
    char *p;
    char buf[80];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;  /* error */
    p = fgets(buf, sizeof(buf), fp);
    if(p == NULL) {
        /* error */
    }
    fclose(fp);
}
```

補 足　この例のように auto 変数にバッファを定義する場合には、タスクのスタックオーバーフローに注意してください。

fputs - ファイルへ１行分の文字列書込み

形 式　int fputs(const char *buf, FILE *fp);

引 数　buf　　書込む文字列が格納されているバッファ
　　fp　　ファイルポインタ

解 説　fp で指定されたファイルへ buf に格納されている文字列を書込みます。文字列を終端する null 文字('¥0')は、ファイルへ書込まれません。

戻 値　正常に文字列を書込めた場合は、非負の値(本ファイルシステムでは 0)を返します。書込めなかった場合は EOF(-1)を返します。

例　ファイルを"w"モードでオープンし、"abcdefg¥n"という文字列を書込みます。

```
void test(void)
{
    FILE *fp;
    int c;

    if (fp = fopen("A:¥¥test.txt", "w")) == NULL)
        return;  /* error */
    c = fputs("abcdefg¥n", fp);
    if (c == EOF) {
        /* error */
    }
    fclose(fp);
}
```

fread - ファイルからブロック読出し

形 式　size_t fread(void *buf, size_t size, size_t n, FILE *fp);

引 数　buf　　データを格納するバッファのアドレス
size　　１ブロックのバイト数
n　　読出すブロックの数
fp　　ファイルポインタ

解 説　fp で指定されたファイルから size×n バイトのデータを読出し、buf へ格納します。

戻 値　読出したブロック数を返します。正常終了の場合は n と等しい値を返します。ファイルの終わり、またはエラーの場合は n より小さい値を返します。size または n に 0 を指定した場合は何も読出さずに 0 を返します。

例　ファイルを"r"モードでオープンし、100 バイトのデータを読出しバッファに格納します。

```
void test(void)
{
    FILE *fp;
    unsigned long n;
    char buf[100];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;  /* error */
    n = fread(buf, 1, 100, fp);
    if (n < 100) {
        /* error */
    }
    fclose(fp);
}
```

補 足　size_t 型は stdio.h か stdef.h に定義されていて、コンパイラによって、long の場合と short の場合があります。総バイト数が **size**×**n** のため、size_t が short の処理系でも、long サイズのデータを一度に扱うことが可能です。また、**size** = 1 と指定すれば、**n** と戻値をブロック数でなく総バイト数と見なすことができます。

fwrite - ファイルへブロック書込み

形 式　size_t fwrite(const void *buf, size_t size, size_t n, FILE *fp);

引 数　buf　　書込みデータバッファのアドレス
size　　１ブロックのバイト数
n　　書込むブロックの数
fp　　ファイルポインタ

解 説　fp で指定されたファイルへ、buf に格納されている size×n バイトのデータを書込みます。

戻 値　書込んだブロック数を返します。正常終了の場合は n と等しい値を返します。エラーの場合は n より小さい値を返します。

例　ファイルを"w"モードでオープンし、バッファのデータを 100 バイト書込みます。

```
void test(void)
{
    FILE *fp;
    unsigned long n;
    char buf[100];

    if ((fp = fopen("A:¥¥ test.txt", "w")) == NULL)
        return;  /* error */
    n = fwrite(buf, 1, 100, fp);
    if (n < 100) {
        /* error */
    }
    fclose(fp);
}
```

補 足　fread の補足を参照してください。

fflush - ファイルのフラッシュ

形 式　int fflush(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルの、バッファされているデータをディスクへ書込みます。fflush()の実行後も、指定されたファイルはオープンのままで、引き続き読み書きが行えます。

戻 値　正常終了の場合は 0 を返します。
エラーの場合は EOF(-1)を返します。

例　ファイルを"w"モードでオープンし、100 バイトのデータを、1 バイトずつフラッシュしながら書込みます。

```
void test(void)
{
    FILE *fp;
    int i, c;

    if ((fp = fopen("A:¥¥test.txt", "w")) == NULL)
        return;  /* error */
    for (i = 0; i < 100; i++) {
        c = read_some_data(); /* 何らかのデータ入力関数 */
        fputc(c, fp);
        fflush(fp);
    }
    fclose(fp);
}
```

補 足　本ファイルシステムでは、コンフィグレーションで設定される書込み遅延時間を待って fflush()と同じ動作が自動的に繰り返されますが、その機能を使用しない場合や、より明示的にファイルのフラッシュを行いたい場合には、fflush()によって FAT キャッシュや入出力バッファの内容とディスクの内容を一致させ、不意なシステムダウンに備えてください。

fseek - ファイル読み書き位置の移動

形 式　int fseek(FILE *fp, long offset, int origin);

引 数　fp　　ファイルポインタ
offset　基準点からの移動バイト数
origin　基準点の種類

解 説　fp で指定されたファイルの読み書き位置を、origin で指定された基準点から offset で指定されたバイト数だけ移動します。

origin には、次のマクロを指定してください。

SEEK_SET	ファイルの先頭
SEEK_CUR	現在位置
SEEK_END	ファイルの終わり

SEEK_SET 指定時はファイルの先頭から offset バイト先の位置へ移動します。この場合、offset には 0 または正の値を指定してください。SEEK_CUR 指定時はファイルの現在の読み書き位置から offset バイト先の位置へ移動します。この場合、offset で指定する値が正ならばファイルの後尾へ、負ならばファイルの先頭へ向かって移動します。SEEK_END 指定時はファイルの終わりからの移動なので、offset には 0 または負の値を指定してください。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ファイルを"w"モードでオープンし、ファイルポインタをファイルの先頭から 100 バイト目に移動します。

```
void test(void)
{
    FILE *fp;
    int r;

    if ((fp = fopen("A:¥¥test.txt", "w")) == NULL)
        return;  /* error */
    r = fseek(fp, 100, SEEK_SET);
    if (r != 0) {
        /* error */
    }
    fclose(fp);
}
```

補 足　FAT32 では 4 ギガバイトまでのファイルを操作できますが、fseek の offset パラメータは long 型のため、2 ギガまでの値しか指定できません。2GB を超えるファイルの位置決めには fsetpos 関数を使用してください。

ftell - 現在のファイル読み書き位置を取得

形 式　long ftell(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルの現在の読み書き位置を返します。値はファイル先頭からのバイト数です。

戻 値　正常終了で現在位置を返します。エラーの場合は-1 を返します。

例　ファイルを"r"モードでオープンし、オープン直後のファイルポインタの位置を調べます。さらに 100 バイト読出し後のファイルポインタの位置を調べます。

```
void test(void)
{
    FILE *fp;
    long pos;
    char buf[100];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;             /* error */
    pos = ftell(fp);        /* pos には 0 が入ります */
    fread(buf, 1, 100, fp);
    pos = ftell(fp);        /* pos には 100 が入ります */
    fclose(fp);
}
```

補 足　FAT32 では 4 ギガバイトまでのファイルを操作できますが、ftell 関数の型は long 型のため、2 ギガまでの値しか表現できません。2 ギガを超える位置は負の数となってしまい、エラーと見分けが付きません。2 ギガを超える位置が返される可能性がある場合は、fgetpos 関数を使用してください。

fsetpos - ファイル読み書き位置の移動

形 式　int fsetpos(FILE *fp, const fpos_t *pos);

引 数　fp　　ファイルポインタ
pos　　読み書き位置の格納アドレス

解 説　fp で指定されたファイルの読み書き位置を、*pos で指定された値に設定します。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ファイルを"w"モードでオープンし、ファイルポインタをファイルの先頭から 100 バイト目に移動します。

```
void test(void)
{
    FILE *fp;
    fpos_t pos;

    if ((fp = fopen("A:¥¥test.txt", "w")) == NULL)
        return;  /* error */

    pos = 100;
    r = fsetpos(fp, &pos);
    if (r != 0) {
        /* error */
    }
    fclose(fp);
}
```

補 足　fpos_t は一般的には stdio.h で定義されていますが、本ファイルシステム用に nofile.h において unsigned long で再定義してあり、4GB までのファイル位置を操作できます。fpos_t の実装は ANSI で規定されておらず、コンパイラによって異なるので、fsetpos/fgetpos を使用したソースを移植する際は互換性に注意してください。

fgetpos - 現在のファイル読み書き位置を取得

形 式　int fgetpos(FILE *fp, fpos_t *pos);

引 数　fp　　ファイルポインタ
*pos　読み書き位置の格納アドレス

解 説　fp で指定されたファイルの現在の読み書き位置を取得し、*pos に返します。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ファイルを"r"モードでオープンし、オープン直後のファイルポインタの位置を調べます。さらに 100 バイト読出し後のファイルポインタの位置を調べます。

```
void test(void)
{
    FILE *fp;
    Int r;
    fpos_t pos;
    char buf[100];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;             /* error */
    r = fgetpos(fp, &pos);  /* pos には 0 が入ります */
    if (r != 0) {
        /* error */
    }
    fread(buf, 1, 100, fp);
    r = fgetpos(fp, &pos);  /* pos には 100 が入ります */
    if (r != 0) {
        /* error */
    }
    fclose(fp);
}
```

補 足　fpos_t は一般的には stdio.h で定義されていますが、本ファイルシステム用に nofile.h において unsigned long で再定義してあり、4GB までのファイル位置を操作できます。fpos_t の実装は ANSI で規定されておらず、コンパイラによって異なるので、fsetpos/fgetpos を使用したソースを移植する際は互換性に注意してください。

feof - ファイルの終わりを検出

形 式　int feof(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルの読み書き位置がファイルの終わりか否かを調べます。

戻 値　ファイルの終わりでは無い場合は 0 を返します。ファイルの終わりを検出した場合は非 0 の値(本ファイルシステムでは 1)を返します。

例　ファイルを"r"モードでオープンし、データを 100 バイト読出し、ファイルの終わりを調べます。

```
void test(void)
{
    FILE *fp;
    char buf[100];

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;  /* error */
    fread(buf, 1, 100, fp);
    if (feof(fp)) {
        /* ファイルの終わり */
    }
    fclose(fp);
}
```

ferror - エラー情報の取得

形 式　int ferror(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルの入出力で発生したエラーの詳細なエラーコードを返します。ファイルシステム内部で保持されているエラーコードは、clearerr()をコールするまではクリアされません。多重にエラーが発生している場合には、新しい方のエラーコードを返します。

戻 値　エラーが発生していない場合は 0 を返します。エラーが発生していた場合は 0 以外の値を返します。具体的な値は、第 4 章“エラーコード一覧”を参照してください。

例　fwrite()の戻値が指定サイズより小さい場合に、エラーコードを参照します。

```
void test(void)
{
    FILE *fp;
    char buf[100];
    int size, err;

    if ((fp = fopen("A:¥¥test.txt", "w")) != NULL) {
        size = fwrite(buf, 1, 100, fp);
        if (size < 100) {
            err = ferror(fp);
            if (err == EV_DISKFULL) {
                /* ディスクフルで書込み中断 */
            }
        }
        fclose(fp);
    }
}
```

補 足　ferror 関数自体は ANSI 準拠ですが、エラーコードは、本ファイルシステム独自のものです。ANSI 準拠のコーディングを行うためには、0(正常)か 0 以外(エラー)かの判断のみを行ってください。

clearerr - エラー情報のリセット

形 式　void clearerr(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　ファイルシステム内部に保持されているエラーコードをクリアします。

戻 値　なし

例　fwrite()の戻値が指定サイズより小さい場合に、エラーコードを参照 し、エラーをクリアします。

```
void test(void)
{
    FILE *fp;
    char buf[100];
    int size, err;

    if ((fp = fopen("A:¥¥test.txt", "w")) != NULL) {
        size = fwrite(buf, 1, 100, fp);
        if (size < 100) {
            err = ferror(fp);
            if (err == EV_DISKFULL) {
                /* ディスクフルで書込み中断 */
                clearerr(fp);
            }
        }
        fclose(fp);
    }
}
```

remove - ファイルの削除

形 式　int remove(const char *path);

引 数　path　　削除するファイルのパス名(フルパス指定)

解 説　path で指定されたファイルを削除します。path は、fopen()と同じようにフルパスで指定してください。フルパス指定の詳細については fopen()の解説を参照してください。

path にワイルドカード('*'や'?')を使うことはできません。オープンしているファイルを path に指定するとエラーになります。path で指定されたファイルが存在しない場合もエラーになります。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ファイルを削除します。

```
void test(void)
{
    int r;

    r = remove("A:¥¥test.txt");
    if (r == -1) {
        /* error */
    }
}
```

rename - ファイル名やディレクトリ名の変更

形 式　int rename(const char *oldname, const char *newname);

引 数　oldname 旧ファイル名またはディレクトリ名(フルパス指定)
newname 新ファイル名またはディレクトリ名(パス指定なし)

解 説　oldname で指定されたファイルまたはディレクトリの名前を newname で指定された名前に変更します。

oldname は、fopen()の path と同じようにフルパスで指定してください。存在しないファイルやオープン中のファイル、または存在しないディレクトリを指定するとエラーになります。

newname はファイル名またはディレクトリ名のみを指定してください。パスを付けるとエラーになります。すでに存在するファイル名またはディレクトリ名を指定するとエラーとなります。

oldname と newname にワイルドカードを使うことはできません。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ディレクトリ"A:¥dir"の下にある"old.c"というファイルを"new.c"というファイル名に変更します。

```
void test(void)
{
    int r;

    r = rename("A:¥¥dir¥¥old.c", "new.c");
    if (r != 0) {
        /* error */
    }
}
```

補 足　他のファイルシステムに比較しての制限として、newname にパスを指定して別のディレクトリへファイルを移動することはできません。

mkdir - ディレクトリの作成

形 式　int mkdir(const char *dirname);

引 数　dirname 作成するディレクトリ名(フルパス名)

解 説　dirname で指定されたディレクトリを作成します。dirname は fopen()で解説したパス名から最後のファイル名を取り除いたものです。

最下層以外のディレクトリが、呼び出し時に存在していない場合、本関数はエラーになります。たとえば 3 階層目のディレクトリを作成する場合、すでに

A:¥AAAAA¥BBBBB

というディレクトリが存在する状態で

mkdir("A:¥¥AAAAA¥¥BBBBB¥¥CCCCC");

のように指定してください。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ディレクトリ"A:¥dir"を作成します。

```
void test(void)
{
    int r;

    r = mkdir("A:¥¥dir");
    if (r == -1) {
        /* error */
    }
}
```

rmdir - ディレクトリの削除

形 式　int rmdir(const char *dirname);

引 数　dirname 削除するディレクトリ名(フルパス名)

解 説　dirname で指定されたディレクトリを削除します。dirname は fopen()で解説したパス名から最後のファイル名を取り除いたものです。

ディレクトリの削除はディレクトリ内が空でない場合、エラーになります。そのため、多階層のディレクトリを削除する場合は下層側から順に削除しなければなりません。ルートディレクトリは指定できません。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ディレクトリ"A:¥dir"を削除します。

```
void test(void)
{
    int r;

    r = rmdir("A:¥¥dir");
    if (r == -1) {
        /* error */
    }
}
```

opendir - ディレクトリのオープン

形 式　DIR *opendir(const char *dirname);

引 数　dirname オープンするディレクトリ名(フルパス指定)

解 説　dirname で指定されたディレクトリをオープンします。オープン後は readdir()でディレクトリ情報を読出すことができます。

存在しないディレクトリ名を指定するとエラーになります。

戻 値　正常にオープンできた場合には、ディレクトリを管理する構造体へのポインタ(ディレクトリポインタ)を返します。以後、ディレクトリに対する操作はこのポインタを指定してください。エラーが発生した場合は NULL を返します。

例　readdir()の例を参照してください。

closedir - ディレクトリのクローズ

形 式　int closedir(DIR* dp);

引 数　dp　　ディレクトリポインタ

解 説　dp で指定されたディレクトリをクローズします。opendir()の戻値を dp に指定してください。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　readdir()の例を参照してください。

readdir - ディレクトリ情報の読出し

形 式　struct dirent *readdir(DIR *dp);

引 数　dp　　ディレクトリポインタ

解 説　dp で指定されたディレクトリの情報(ディレクトリエントリ)を、１つずつ順に読出します。

POSIX 準拠の dirent 構造体は、次の形式となっています。POSIX の場合の NAME_MAX の値は 128 ですが、本ファイルシステムでは 256 に拡張してあります。

```
struct dirent
{
    long d_ino;                 /* inode 番号(未使用:不定) */
    unsigned short d_namlen;    /* ファイル名の長さ(null 文字含まず) */
    char d_name[NAME_MAX+1];    /* ファイル名(null 文字'¥0'で終端) */
};
```

戻 値　正常に読出せた場合は、dirent 構造体へのポインタを返します。最後の情報に達していた場合やエラーの場合は、NULL を返します。

例　ディレクトリ"A:¥dir"をオープンしてこのディレクトリ下にあるディレクトリとファイルの情報を取得し、ディレクトリ名またはファイル名を出力します。

```
void test(void)
{
    DIR *dp;
    struct dirent *p;

    if ((dp = opendir("A:¥¥ dir ")) == NULL)
        return;  /* error */
    for (;;) {
        p = readdir(dp);
        if (p == NULL)
            break;
        puts(p->d_name);
    }
    closedir(dp);
}
```

補 足　本ファイルシステムの独自拡張として、dirent 構造体を direntx 構造体へキャスト(型変換)することによって、より詳細なディレクトリエントリ情報を取得することが可能です。独自の direntx 構造体は、次の形式となっています。

```
struct direntx
{
    T_DIRENTRY *direntry;       /* ディレクトリエントリ詳細情報 */
    unsigned short d_namlen;    /* ファイル名の長さ(null 文字含まず) */
    char d_name[NAME_MAX+1];    /* ファイル名(null 文字'¥0'で終端) */
};
```

direntx 構造体の direntry メンバーでポイントされる独自の T_DIRENTRY 構造体は、次の形式となっています。

```
typedef struct t_direntry {
    unsigned char name[8+3];    ファイル名、ディレクトリ名
    char attr;                  属性
    char rsv;                   予約(値 0x00)
    unsigned char mktimems;     作成時刻(10ms 単位)
    unsigned short mktime;      作成時刻
    unsigned short mkdate;      作成日付
    unsigned short access;      アクセス日付
    unsigned short up_clusno;   上位 FAT エントリ No.
    unsigned short time;        更新時刻
    unsigned short date;        更新日時
    unsigned short clusno;      下位 FAT エントリ No
    unsigned long size;         ファイルサイズ(バイト単位)
} T_DIRENTRY;
```

※上記の構造体は、ディスク上にリトルエンディアン並びで格納されている値そのものです。ビッグエンディアンの CPU で参照する際は、エンディアン変換が必要です。

attr のビット構成

7	6	5	4	3	2	1	0

ビット 0：読出し専用ファイル　　ビット 3：ディスクドライブボリュームラベル
ビット 1：隠しファイル　　ビット 4：ディレクトリ
ビット 2：システムファイル　　ビット 5～7：未使用

mktime、time のビット構成

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

時：０～２３　　分：０～５９　　秒：０～２９

注)秒は 2 秒単位です。例えば、40 秒の場合は 20 が格納されます。

mkdate、date、access のビット構成

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

年：０～１２７　　月：１～１２　　日：１～３１

注)年は 1980 を起点とします。例えば、2004 年では 24 が格納されます(2004-1980=24)。

例　ディレクトリ"A:¥dir"をオープンしてこのディレクトリ下にあるディレクトリとファイルの情報を取得し、ファイル名とファイルサイズを出力します。

```
void test(void)
{
    DIR *dp;
    struct direntx *p;
    char s[11];

    if ((dp = opendir("A:¥¥ dir ")) == NULL)
        return;  /* error */
    for (;;) {
        p = (struct direntx *)readdir(dp);
        if (p == NULL)
            break;
        print(p->d_name); print(" ");
        ltod(s, p->direntry->size, 10);
        puts(s);
    }
    closedir(dp);
}
```

fgetattr - ファイルの属性を取得

形 式　int fgetattr(const char *path);

引 数　path　　パス名(フルパス指定)

解 説　path で指定されたファイルの属性値を取得します。path は、fopen()と同じようにフルパスで指定してください。ファイルの属性値は fsetattr()の解説を参照してください。本ファイルシステムでは、ディレクトリの属性は取得できません。

戻 値　正常終了の場合は属性値を返します。エラーの場合は-1 を返します。

fsetattr - ファイルの属性を変更

形 式　int fsetattr(const char *path, int attr);

引 数　path　　パス名(フルパス指定)
attr　　属性値

解 説　path で指定されたファイルの属性値を attr で指定された値に変更します。
ファイルの属性値は１バイトで、各ビットの意味は次のようになります。

bit0	読出し専用ファイル	ATTR_READONLY
bit1	隠しファイル	ATTR_HIDDEN
bit2	システムファイル	ATTR_SYSTEM
bit3	ディスクドライブボリュームラベル	ATTR_VOLUME
bit4	ディレクトリ	ATTR_DIRECTORY
bit5	アーカイブ	ATTR_ARCHIVE
bit6	未使用	
bit7	未使用	

上記以外に全ビットが 0 の ATTR_NORMAL が定義されています。本ファイルシステムでは、ディレクトリの属性は変更できません。ATTR_VOLUME や ATTR_DIRECTORY は定義されていますが、指定するとエラーとなります。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ファイルの属性値を取得し、読出し専用ビット(bit0)を ON に変更します。

```
void test(void)
{
    int r, attr;

    attr = fgetattr("A:¥¥test.txt");
    r = fsetattr("a:¥¥test.txt", (attr | ATTR_READONLY));
    if (r == -1) {
        /* error */
    }
}
```

fgetsize - ファイルサイズの取得

形 式　int fgetsize(FILE *fp, unsigned long *size);

引 数　fp　　ファイルポインタ
size　ファイルサイズ格納先のポインタ

解 説　fp で指定されたファイルのサイズを取得します。取得したファイルサイズは、バイト単位で、size で指定された変数へ格納されます。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ファイルのファイルサイズを取得します。

```
void test(void)
{
    FILE *fp;
    unsigned long size;
    int r;

    if ( (fp = fopen("A:¥¥test.txt", "r")) != NULL) {
        r = fgetsize(fp, &size);
        if (r == 0) {
            /* size に"A:¥¥test.txt"のファイルサイズが格納されます */
        } else {
        /* error */
        }
        fclose(fp);
    }
}
```

fsetsize - ファイルサイズの変更

形 式　int fsetsize(FILE *fp, unsigned long size);

引 数　fp　　ファイルポインタ
size　変更後のファイルサイズ

解 説　fp で指定されたファイルのサイズを、size で指定されたサイズへ変更します。

現在のファイルサイズより大きいサイズを指定した場合は、追加となるデータとして、null 文字('¥0')が書込まれます。小さいサイズを指定した場合は、切り捨てられる部分のデータが失われます。"r"モードでオープンしている場合はエラーを返します。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ファイルを空にします。

```
void test(void)
{
    FILE *fp;
    int r;

    if ((fp = fopen("A:¥¥test.txt", "r+")) != NULL) {
        r = fsetsize(fp, 0);
        if (r != 0) {
            /* error */
        }
        fclose(fp);
    }
}
```

fgetmtime - ファイルまたはディレクトリの作成時刻を取得

形 式　int fgetmtime(const char *path, struct tm *mtime);

引 数　path　　ファイルまたはディレクトリ名(フルパス指定)
mtime　時刻情報格納先へのポインタ

解 説　path で指定されたファイルやディレクトリの作成時刻(タイムスタンプ)を mtime で指定された変数へ格納します。path は fopen()と同様にフルパスで指定してください。

tm 構造体については、fsetmtime()の解説を参照してください。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ファイルのタイムスタンプを取得します。

```
void test(void)
{
    struct tm ts;
    int r;

    r = fgetmtime("A:¥¥test.txt", &ts);
    if (r == 0) {
        /* ts に"A:¥¥test.txt"のタイムスタンプが格納されます*/
    } else {
        /* error */
    }
}
```

補 足　Unix 系のファイルが持つ時刻としては、mtime：ファイルが作成/修正された(Modified)時刻、atime：最後にアクセスされた(Accessed)時刻、ctime：属性が変更された(Changed)時刻の３つが存在します。本ファイルシステムでは、このうち、mtime のみの取得と変更をサポートしています。

fsetmtime - ファイルまたはディレクトリの作成時刻を変更

形 式　int fsetmtime(const char *path, struct tm *mtime);

引 数　path　　ファイルまたはディレクトリ名(フルパス指定)
mtime　時刻情報へのポインタ

解 説　path で指定されたファイルまたはディレクトリの作成時刻(タイムスタンプ)を変更します。時刻情報を tm 構造体へ代入し、そのポインタを mtime に指定してください。

月は、1 月を 0 として計算されます。2 月は 1、3 月は 2、…、12 月は 11 という指定になりますので注意してください。

年は 1900 年を 0 として計算されます。ディスク上には 1980 年以降の値しか保存できないので、必ず 80 以上(1980 年以降)を指定してください。

秒の値は偶数秒に丸められます。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　タイムスタンプを「2004 年 8 月 9 日 10 時 20 分 30 秒」にセットします。

```
void test(void)
{
    struct tm ts;

    ts.tm_sec = 30;             /* 30 秒 */
    ts.tm_min = 20;             /* 20 分 */
    ts.tm_hour = 10;            /* 10 時 */
    ts.tm_mday = 9;             /* 9 日 */
    ts.tm_mon = 8 - 1;          /* 8 月(1 月を 0 とした値) */
    ts.tm_year = 2004 - 1900;   /* 2004 年(1900 年を 0 とした値) */
    ts.tm_wday = 0;             /* 未使用 */
    ts.tm_yday = 0;             /* 未使用 */
    ts.tm_isdst = 0;            /* 未使用 */
    fsetmtime ("A:¥¥test.txt", &ts);
}
```

returnname - ファイル名を返す

形 式　const char *returnname(FILE *fp);

引 数　fp　　ファイルポインタ

解 説　fp で指定されたファイルのパス前を、"d:¥aaaa¥bbbb¥nnnnnnnn.eee"形式のフルパスで返します。

戻 値　パス名が格納されている領域へのポインタを返します。

例　　オープンしてあるファイルのタイムスタンプを取得します。

```
void test(void)
{
    FILE *fp;
    struct tm ts;

    if ((fp = fopen("A:¥¥test.txt", "r")) == NULL)
        return;  /* error */
    fgetmtime(returnname(fp), &ts); /* fp をパス名に変換 */
    fclose(fp);
}
```

補 足　ファイルポインタでなくパス名でファイルを指定するタイプの API を、オープン中のファイルに対しても簡単に利用できるように用意されている独自の API です。

getdiskfree - ディスクの残容量を取得(4GB 未満)

形 式　int getdiskfree(const char *drv, unsigned long *size);

引 数　drv　　ドライブ名("A:", "B:", …)
size　　取得した残容量の格納先へのポインタ

解 説　drv で指定されたドライブの残容量を求め、size で指定された先へ結果を格納します。容量の単位はバイトです。結果が 32 ビットで表現できる最大値を超えた場合(4GB 以上の場合)は、0xffffffff を size で指定された先へ格納します。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ドライブ"A:"のディスク残容量を求めます。

```
void test(void)
{
    unsigned long size;
    int r;

    if ((r = getdiskfree("A:", &size)) != 0) {
        /* error */
    }
}
```

補 足　サイズが４GB を超える場合は getdiskfreex()を使用してください。

getdiskfreex - ディスクの残容量を取得(4GB 以上)

形 式　int getdiskfreex(const char *drv, unsigned long *h_size, unsigned long *l_size);

引 数　drv　　ドライブ名("A:", "B:", …)
h_size　残容量上位 32bit の格納先へのポインタ
l_size　残容量下位 32bit の格納先へのポインタ

解 説　drv で指定されたドライブの残容量を求め、h_size、l_size で指定された先へ結果を格納します。容量の単位はバイトです。結果の上位 32bit は h_size で指定された先へ格納され、下位 32bit は l_size で指定された先へ格納されます。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ドライブ"A:"の残容量を調べます。

```
void test(void)
{
    unsigned long h_free, l_free;
    int r;

    r = getdiskfreex ("A:", &h_free, &l_free);
    if (r != 0) {
        /* error */
    }
    if (h_free > 0) {
        /* 残容量は 4GB 以上です */
    }
}
```

getdisksize - ディスクの容量を取得

形 式　int getdisksize(const char *drv, unsigned long *size);

引 数　drv　　ドライブ名("A:", "B:", …)
size　　ディスク容量の格納先へのポインタ

解 説　drv で指定されたドライブの容量を求め、size で指定された先へ取得したディスク容量を格納します。容量の単位はメガバイト(MB)です。

戻 値　正常終了の場合は 0 を返します。エラーの場合は非 0 の値(本ファイルシステムでは-1)を返します。

例　ドライブ"A:"の総容量を調べます。

```
void test(void)
{
    unsigned long total;
    int r;

    r = getdisksize("A:",  &total);
    if (r != 0) {
        /* error */
    }
}
```

disk_mount - マウント

形 式　int disk_mount(T_DISK *d);

引 数　d　　T_DISK 構造体へのポインタ

解 説　ディスクドライブを使用可能にします。d には disk_ini()の第 1 引数 d で指定した T_DISK 構造体へのポインタを指定してください。disk_ini()で取り外し可能(リムーバブル)を指定しなかった場合には、マウント処理を行う必要はありません 。(行っても害はありません)

戻 値　正常終了の場合は E_OK を返します。エラーの場合はドライバの戻値を返します。

例　　disk_unmount()の例を参照してください。

disk_unmount - アンマウント

形 式　int disk_unmount(T_DISK *d);

引 数　d　　T_DISK 構造体へのポインタ

解 説　ディスクドライブの使用を終了します。d には disk_ini()の第 1 引数 d で指定した T_DISK 構造体へのポインタを指定してください。

戻 値　正常終了の場合は E_OK を返します。エラーの場合はドライバの戻値を返します。

例　ドライブ"A:"をリムーバブルなディスクとして初期化し、マウント、アンマウントします。

```
#define NCACHE 5
T_DISK disk[1];
T_CACHE cache[NCACHE];

int test(void)
{
    int r;

    r = disk_ini(disk, "A:", flash_ATA, 0, 0, NULL, 0, 1);
    if (r != E_OK) {
        return -1;  /* error */
    }
    disk_cache(disk, cache, NCACHE);

    r = disk_mount(disk);
    if (r != E_OK) {
        return -1;  /* error */
    }

    /* この間でファイル入出力 */

    r = disk_unmount(disk);
    if (r != E_OK) {
        return -1;  /* error */
    }
}
```

dformat - ディスクのフォーマット

形 式　int dformat(const char *drv, const long param);

引 数　drv　　ドライブ名("A:", "B:", …)
param　ディスクドライバ依存のパラメータ

解 説　指定されたディスクを完全に初期化します。

drv にはフォーマットするドライブのドライブレター文字列を指定してください。

param にはドライバによって決められたパラメータを指定してください。RAM ディスクの場合は未使用で、0 を指定してください。

本関数を使用する場合はドライバがフォーマット機能をサポートしている必要があります。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

補 足　付属の ATA ドライバは本機能をサポートしていません。

RAM ディスクドライバは本機能をサポートしています。

qformat - ディスクのクイックフォーマット

形 式　int qformat(const char *drv);

引 数　drv　　ドライブ名("A:", "B:", …)

解 説　drv で指定されたディスクのファイルを全て削除します。

すでに FAT ファイルシステムでフォーマットされているディスクのみに使用できます。ディスクの BPB(BIOS Parameter Block)は書き換えません。

戻 値　正常終了の場合は 0 を返します。エラーの場合は-1 を返します。

例　ドライブ"A:"に対してクイックフォーマットを行います。

```
void test(void)
{
    int r;

    r = qformat("A:");
    if (r == -1) {
        /* error */
    }
}
```

第４章　エラーコード一覧

本ファイルシステムのエラーコードは次のようになります。

エラーコード名	エラー コード値	エラー内容
EV_FNAME	-97	指定ファイル名、指定ディレクトリ名が異常
EV_FSAME	-98	同一ファイル名、同一ディレクトリ名が存在する
EV_NOFILE	-99	指定ファイルが存在しない
EV_NODIR	-100	指定ディレクトリが存在しない
EV_FMODE	-101	指定モードが異常
EV_NOOPEN	-102	ファイルがオープンされていない
EV_OPENED	-103	指定ファイルが既にオープンされている
EV_DISKRD	-104	デバイスリードエラー
EV_DISKWR	-105	デバイスライトエラー
EV_NFILE	-106	可能な同時オープンファイル数を超えた
EV_DISKFULL	-107	ディスクフル
EV_DRVNAME	-108	指定ドライブ名が異常
EV_FPAR	-109	指定パラメータが異常
EV_EOF	-110	ファイルの終端に達した
EV_DIRENT	-111	可能な同時オープンディレクトリ数を超えた
EV_FNOSPT	-112	サポートされていない
EV_DISKINI	-113	初期化(disk_ini)されていない
EV_FILEINI	-114	初期化(fsys_ini)されていない
EV_UNMOUNT	-115	マウントされていない
EV_NOEMPTY	-116	空でないディレクトリを削除しようとした
EV_DRVINI	-117	デバイス初期化エラー
EV_MOUNT	-118	デバイスマウントエラー
EV_CACHESZ	-119	キャッシュサイズ不足
EV_FATS	-120	FAT エラー
EV_FILEFULL	-121	2GB 以上のファイルを作成しようとした
EV_FREESECT	-122	空き容量に設定されている値が異常
EV_FERR	-128	その他

第５章　ドライバ・インターフェース

5.1　ディスクドライバ関数

ディスクドライバは､本ファイルシステムにデバイスのread/writeまたはformatなどの機能を提供します。ディスクドライバのエントリは１ドライブにつき１個の関数です。ディスクドライバの初期化 disk_ini()の引数 **func** にこの関数名を指定することにより、ファイルシステムとディスクドライバが関連付けられます。

ディスクドライバ関数はディスクによって異なり、そのの名前は自由ですが、本書では **diskdrive** として説明します｡ディスクドライバ関数は､次の形式をしています。

```
ER diskdrive(T_DISK *d, FN fncd, UW sectno, VP parm, UH snum);
```

T_DISK 構造体へのポインタ **d** へは､ディスクドライバの初期化 disk_ini()の引数 **d** で指定された値が渡ります。**fncd** へは、ディスクドライバへのコマンドが指定されます。**sectno** は、読み書きするセクタの番号です。**param** はコマンドによって意味が変わります｡**snum** は連続した複数セクタの読み書きを指示する場合のセクタ数を指定します。

ディスクドライバ関数の戻値はエラーコードで、正常終了の場合は E_OK が返ります。

5.2　コマンド一覧

ディスクドライバのコマンドの体系はATAコマンドをベースにしています。ファイルシステム本体からディスクドライバへ渡されるコマンドには次のものがあります。

TFN_READ_SECTOR	指定セクタを読出し(param = 入力バッファ)
TFN_WRITE_SECTOR	指定セクタへ書込み(param = 出力バッファ)
TFN_IDENTIFY	パラメータ情報の取得
TFN_FORMAT	フォーマット(param = dformat()の第２引数)
TFN_HARDWARE_INIT	ハードウェアの初期化
TFN_HARDWARE_RESET	ハードウェアのリセット
TFN_MOUNT	マウント
TFN_UNMOUNT	アンマウント
TFN_MEDIACHK	挿入チェック

5.3 ディスクドライバの例

```
ER diskdrive(T_DISK *d, FN fncd, UW sectno, VP parm)
{
  ER ercd;
  switch (fncd) {
  case TFN_READ_SECTOR:
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_READING );
       ercd = read_sector(d, sectno, (UB **)parm);
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_STOPPED );
       return ercd;
  case TFN_WRITE_SECTOR:
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_WRITING );
       ercd = write_sector(d, sectno, (UB *)parm);
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_STOPPED );
       return ercd;
  case TFN_IDENTIFY:
       ercd = identify(d);
       return ercd;
  case TFN_FORMAT:
       ercd = format(d);
       return ercd;
  case TFN_HARDWARE_INIT:
       ercd = hardware_init(d);
       return ercd;
  case TFN_HARDWARE_RESET:
       return E_OK;
  case TFN_MOUNT:
       ercd = mount(d);
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_MOUNTED );
       return ercd;
  case TFN_UNMOUNT:
       ercd = unmount(d);
       if ( d->callback != (DISK_CALLBACK)NULL )
           d->callback( d, TFN_DISK_UNMOUNTED );
       return ercd;
  case TFN_MEDIACHK:
       ercd = ref_drv(d);
       return ercd;
  default:
       return E_NOSPT;
  }
}
```

第６章　状態変化通知用コールバック関数

6.1　概要

状態変化通知用コールバック関数は本ファイルシステムがユーザーにディスクの状態を通知するための関数です。本関数を使用すれば、アクセス中にメディアをロックする､またはアクセスランプをつけることなどが可能になります｡また、コンパクトフラッシュの挿入や抜取の検出時も本関数がコールされます。挿抜の検出時に、本コールバック関数の中から NORTi のシステムコールを使用して任意のタスクに挿抜の検出を通知することが可能です。ただし、挿抜の検出は周期起動ハンドラで行っているので、呼び出せるシステムコールには制限があります。

6.2　機能一覧

状態変化通知用コールバック関数の機能一覧は次のようになります。

TFN_DISK_READING	読出し開始の通知
TFN_DISK_WRITING	書込み開始の通知
TFN_DISK_STOPPED	読出しまたは書込み終了の通知
TFN_DISK_DETECTED	メディア挿入の通知
TFN_DISK_REMOVED	メディア抜取の通知
TFN_DISK_MOUNTED	マウントの通知
TFN_DISK_UNMOUNTED	アンマウントの通知
TFN_DISK_ERROR	エラーの通知

6.3 コールバック関数の例

コールバック関数のエントリは 1 ドライブにつき 1 個です。関数 **disk_ini** の引数 **callback** にエントリのポインタを指定してください。関数名は自由ですが、本書ではコールバック関数のエントリ名を disk_callback として、その例を示します。例では、各タイミングで書込み中や読出し中を示すＬＥＤを点灯しています。

```
#define LED_RD     0x8000
#define LED_WR     0x4000
#define LED_DET        0x2000
#define LED_MOUNT  0x0100

void led_set(unsigned short c);
void led_clr(unsigned short c);

ER  disk_callback(T_DISK *d, FN fncd)
{
  switch(fncd) {
  case TFN_DISK_READING:
       led_set(LED_RD);
       return E_OK;
  case TFN_DISK_WRITING:
       led_set(LED_WR);
       return E_OK;
  case TFN_DISK_STOPPED:
       led_clr(LED_RD|LED_WR);
       return E_OK;
  case TFN_DISK_DETECTED:
       led_set(LED_DET);
       return E_OK;
  case TFN_DISK_REMOVED:
       led_clr(LED_DET);
       return E_OK;
  case TFN_DISK_MOUNTED:
       led_set(LED_MOUNT);
       return E_OK;
  case TFN_DISK_UNMOUNTED:
       led_clr(LED_MOUNT);
       return E_OK;
  case TFN_DISK_ERROR:
       return E_OK;
  default:
       return E_NOSPT;
  }
}
```

第７章　PC カードアクセス関数の実装

本ファイルシステムに付属する ATA ドライバが、コンパクトフラッシュ等のレジスタをアクセスする為に呼び出す関数を、PC カードアクセス関数と呼びます。これらはハードウェア依存であり、ハードにあわせての移植作業が必要です。本章では、そのポイントを関数ごとに記述します。

なお ATA ドライバは ATA(IDE)の各レジスタにバイトアクセス可能で、データレジスタはワードアクセス可能であることを前提に実装されています。これらを満たさない場合は nofata.c の変更が必要になります。DMA と割り込みは使用していないので、それに関する移植作業は考慮する必要がありません。

ER PCCard_init(void)

解 説　PC カード挿入の確認後、ドライバの関数 mount()から呼ばれます。
正式な PCMCIA(PC カード)手順の場合は、カードの種別を確認し、適切な VCC(3.3V または 5V)を与えます。安定待ちのために最後に一定時間のウェイトを行います。
PC カード挿入を確認できない場合は異常終了します。
PC カード挿入と同時に、自動的に電源が ON になるようなハードウェアの場合、本関数はウェイトのみの実装で構いません。
PC カード準拠でも常時電源 ON である簡易型接続や、TrueIDE での接続の場合は、すでに電源 ON ですので正常終了だけで構いません。

ER PCCard_check(void)

解 説　PC カード挿入チェック関数です。
PC カードが挿入されている場合は E_OK を、挿入されていない場合は E_OBJ を返してください。
もともと活線挿抜に対応していない TrueIDE や、簡易型の PC カードの場合は挿抜が検知できない場合が多いので、E_OK を返してください。

ER PCCard_end(void)

解 説　PC カード電源(VCC)を OFF にします。電源を ON/OFF する機能がない場合は、何もしないでリターンさせてください。

void PCCard_open(void)

解 説　PC カードの挿入の確認後、関数 mount()から PCCard_init()の後に呼ばれます。
PC カードの各空間(コモン、アトリビュート、I/O)をそれぞれ別々のアドレスに割り当てる為の設定を、PC カードコントローラに対して行う必要がある場合は、ここで行います。
設定を行わなくても、PC カードの各空間をアクセスできるようなハードウェアの場合は、その設定は不要です。
上記はいずれも PC カード仕様なので、ATA レジスタを I/O 空間でアクセスする為に、CompactFlash を I/O モードに設定する必要があります。
カードコンフィグレーションレジスタに Index=1 を書込んで、ContiguousI/O モードに設定してください。(PCCard_atr_writeb 関数を使用)
TrueIDE の場合はこの処理は不要です。

UB PCCard_atr_readb(UW addr)

解 説　PC カードのアトリビュート空間を読み出してカードのタプル情報を取得する場合に実装してください。
アトリビュート空間のベースアドレスにオフセット(addr)を加算したアドレスからデータをバイトアクセスで読出し、戻値として返します。
ドライバではタプル情報の活用をしていないので、省略可能です。TrueIDE の場合は、そもそも不要です。

void PCCard_atr_writeb(UW addr, UB data)

解 説　PC カードの、アトリビュート空間のベースアドレスにオフセット(addr)を加算したアドレスへデータをバイトアクセスで書込みます。
CompactFlash カードを I/O カードモードに設定するために、アトリビュート空間内のカードコンフィグレーションレジスタへの書き込みが必要ですが、その為だけに使用されます。PC カード用カードアクセス関数のソース内でしか使用しないので、直接アドレスを指定してコンフィグレーションレジスタへの書き込みを記述するならば、この関数は不要です。
TrueIDE の場合はアトリビュート空間が無いので、そもそも不要です。

UB PCCard_io_readb(UW addr)

解 説　PC カードでは、I/O 空間のベースアドレスにオフセット(addr)を加算したアドレスからデータをバイトアクセスで読出し、戻値として返します。
TrueIDE の場合は、TrueIDE のレジスタ群のベースアドレスに対し、オフセット(addr)を加算したアドレスからデータをバイトアクセスで読み出し、戻値として返します。データレジスタ以外のレジスタを読み出す為に使用します。

void PCCard_io_writeb(UW addr, UB data)

解 説　PC カードでは、I/O 空間のベースアドレスにオフセット(addr)を加算したアドレスへデータ(data)をバイトアクセスで書込みます。
TrueIDE の場合は、TrueIDE のレジスタ群のベースアドレスに対し、オフセットを加算したアドレスへデータをバイトアクセスで書き込みします。データレジスタ以外のレジスタへ書き込む為に使用します。

UH PCCard_io_readw(UW addr)

解 説　PC カードでは、I/O 空間のベースアドレスにオフセット(addr)を加算したアドレスからデータをワードアクセスで読み出し、戻値として返します。
TrueIDE の場合は、TrueIDE のレジスタ群のベースアドレスに対し、オフセットを加算したアドレスからデータをワードアクセスで読み出し、戻値として返します。データレジスタを読み出す為に使用します。

void PCCard_io_writew(UW addr, UH data)

解 説　PC カードでは、I/O 空間のベースアドレスにオフセット(addr)を加算したアドレスへデータ(data)をワードアクセスで書き込みます。
TrueIDE の場合は、TrueIDE のレジスタ群のベースアドレスに対し、オフセット(addr)を加算したアドレスへデータ(data)をワードアクセスで書き込みます。データレジスタへ書き込む為に使用します。

NORTi File System ユーザーズガイド

株式会社ミスポ http://www.mispo.co.jp/
〒222-0033 神奈川県横浜市港北区新横浜 3-20-8 BENEX S-3 12F
一般的なお問い合せ sales@mispo.co.jp
技術サポートご依頼 norti@mispo.co.jp