ARM® コンパイラ

バージョン **6.01**

ARM C ライブラリ、**C++** ライブラリ、および浮動小数点サポートリファレンスガイド

ARM®

ARM DUI 0809BJ (ID １２１８１４)

ARM コンパイラ
ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートリファレンスガイド

リリース情報

本書には以下の変更が加えられています。

変更履歴

日付	発行	機密保持ステータス	変更点
2014 年 3 月 14 日	A	非機密扱い	ARM コンパイラ v6.00 リリース
2014 年 12 月 15 日	B	非機密扱い	ARM コンパイラ v6.01 リリース

著作権

機密保持ステータス

本書は非機密扱いであり、本書を使用、複製、および開示する権利は、ARM および ARM が本書を提供した当事者との間で締結した契約の条項に基づいたライセンスの制限により異なります。

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

Web アドレス

`http://www.arm.com`

目次
ARM コンパイラ ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートリファレンスガイド

第 1 章
表記規則とフィードバック

以下では、表記規則とフィードバックの方法について説明します。

表記規則

以下の表記規則を使用しています。

`monospace` コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力可能なテキストを示しています。

`mono`space コマンドまたはオプションに使用可能な略語を示します。コマンド名またはオプション名をすべて入力する代わりに、下線部分の文字だけを入力することができます。

`monospace italic`
コマンドまたは関数の引数で、特定の値に置き換えることが可能なものを示しています。

`monospace bold`
サンプルコード以外に使用される言語キーワードを示しています。

italic 重要事項、重要用語、相互参照、引用箇所を斜体で記載しています。

bold メニュー名などのユーザインタフェース要素を太字で記載しています。また、適宜記述リスト内の重要箇所と ARM® プロセッサの信号名にも太字を用いています。

本製品に関するフィードバック

本製品についてのご意見やご提案がございましたら、以下の情報を添えて購入元までお寄せ下さい。

- お名前と会社名
- 製品のシリアル番号
- 製品のリリース情報
- ご使用のプラットフォームの詳細（ハードウェアプラットフォーム、オペレーティングシステムの種類とバージョンなど）
- 問題を再現するサイズの小さな独立したサンプルコード
- 期待した結果と実際の結果の詳しい説明
- 使用したコマンド（コマンドラインオプションを含む）
- 問題を例示するサンプル出力
- ツールのバージョン情報（バージョン番号、ビルド番号を含む）

本書に関するフィードバック

本書に関するご意見につきましては、電子メールを errata@arm.com まで送信して下さい。その際には、以下の内容を記載して下さい。

- タイトル
- 文書番号（ARM DUI 0809BJ）
- オンラインでご覧の場合は、該当するトピック名
- PDF 版の文書をご覧の場合は、問題のあるページ番号
- 問題点の簡潔な説明

また、補足すべき点や改善すべき点についての全般的なご提案もお待ちしております。

ARM では、技術情報記事や *FAQ* の拡充と共に、ドキュメントに対する更新と訂正を ARM Information Center にて定期的に行っております。

その他の情報

- ARM Information Center、http://infocenter.arm.com/help/index.jsp
- ARM Technical Support Knowledge Articles、http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html
- ARM Support and Maintenance、http://www.arm.com/support/services/support-maintenance.php
- ARM 用語集、http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html

第 2 章
C ライブラリと C++ ライブラリ

以下の各トピックでは、C 標準の拡張である、または標準とは何らかの違いがある、標準 C ライブラリ関数および標準 C++ ライブラリ関数について説明します。標準関数の中には、ARM ターゲット変更可能セミホスティング環境と対話するものがあります。以下では、このような関数についても説明します。

2.1 __aeabi_errno_addr()

この関数は、C ライブラリが errno の読み出しまたは書き込みを試行した場合に、C ライブラリの errno 変数のアドレスを取得するために呼び出されます。デフォルトの実装はライブラリによって提供されます。通常、この関数を再実装する必要はありません。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.1.1 構文

```
volatile int *__aeabi_errno_addr(void);
```

2.1.2 関連項目

参照

- *errno*（2-9 ページ）

その他の情報

- *ARM アーキテクチャ用 C ライブラリ ABI*、http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039-/index.html

2.2 alloca()

alloca.h 内で定義されるこの関数は、関数にローカルストレージを割り当てます。*size* バイトのメモリを指すポインタが返されます。デフォルトの実装では、スタック上の 8 バイトアラインメントのメモリブロックが返されます。

alloca() から返されたメモリを free() に渡すことはできません。その代わり、alloca() を呼び出した関数が復帰したときに、そのメモリは自動的に解放されます。

注

alloca() を関数ポインタを介して呼び出すことはできません。alloca() と setjmp() の両方を同じ関数内で使用する場合は注意が必要です。これは、setjmp() と longjmp() の呼び出しの間に alloca() によってメモリを割り当てると、そのメモリは longjmp() への呼び出しによって解放されるためです。

この関数は、多くの C ライブラリで一般的な非標準拡張です。

2.2.1 構文

void *alloca(size_t *size*);

2.2.2 関連項目

参照

- *スレッドセーフな C ライブラリ関数*（2-64 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *ARM C ライブラリとスレッドセーフな関数*（2-17 ページ）
- *C ライブラリを使用しないアプリケーションの作成*（2-41 ページ）

2.3 clock()

time.h に含まれている標準 C ライブラリクロック関数です。

2.3.1 構文

```
clock_t clock(void);
```

2.3.2 使用法

この関数のデフォルト実装では、セミホスト機能が使用されます。

clock_t の単位がデフォルトの 1/100 秒と異なる場合には、コンパイラのコマンドラインから、またはヘッダファイル内で __CLK_TCK を定義する必要があります。この定義で使用されている値が CLK_TCK と CLOCKS_PER_SEC に使用されます。デフォルト値は 1/100 秒を表す 100 です。

注

clock() を再実装する場合は、_clock_init() も再実装する必要があります。

2.3.3 戻り値

符号なし整数が返されます。

2.3.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.4 _clock_init()

rt_misc.h で定義されるこの関数は、clock() の初期化関数です。C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.4.1 構文

```
void _clock_init(void);
```

2.4.2 使用法

これは、実装に固有の方法で再実装できる関数です。ライブラリ初期化コードから呼び出されるため、アプリケーションコードから呼び出す必要はありません。

注

clock() を再実装する場合は、この関数も再実装する必要があります。

_clock_init() が適用される初期化によって、プログラムが開始されたときからの経過時間が clock() から返されるようになります。

_clock_init() を再実装する方法の例として、タイマをゼロに設定することが考えられます。ただし、clock() の実装が、リセットできないシステムタイマに依存している場合は、_clock_init() で起動時（ライブラリ初期化コードからの呼び出し時）に時間を読み取り、その後に clock() で現在のタイマ値から初期化時に読み取った時間を減算します。いずれの場合も、何らかの形式の初期化が _clock_init() に必要です。

2.4.3 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド :

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.5 __default_signal_handler()

rt_misc.h 内で定義されるこの関数は、生成されたシグナルを処理します。デフォルトの動作では、エラーメッセージを出力して終了します。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.5.1 構文

int __default_signal_handler(**int** *signal*, **int** *type*);

2.5.2 使用法

デフォルトのシグナルハンドラはゼロ以外の値を返し、呼び出し元はプログラムを終了するために配列する必要があることを示します。このデフォルトのシグナルハンドラは、以下を定義することによって置き換えられます。

int __default_signal_handler(**int** *signal*, **int** *type*);

インタフェースは __raise() と同じですが、この関数は C シグナル処理メカニズムによってシグナルが処理されなかった場合にのみ呼び出されます。

定義済みシグナルの一覧については、signal.h を参照して下さい。

注

ライブラリによって使用されるシグナルは、本製品の今後のリリースで変更される可能性があります。

2.5.3 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリの signal() 関数と追加の型引数によりサポートされる ISO 準拠のシグナルの実装*（2-113 ページ）

参照

- *__raise()*（2-23 ページ）
- *_ttywrch()*（2-56 ページ）
- *_sys_exit()*（2-46 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリ関数の間接セミホスティング依存関数*（2-38 ページ）

2.6 errno

C ライブラリの errno 変数は、ライブラリの暗黙的なスタティックデータエリア内で定義されます。このエリアは __user_libspace() によって識別されます。errno のアドレスは次の関数から返されます。

```
(*(volatile int *) __aeabi_errno_addr())
```

__aeabi_errno_addr() を定義することで、__user_libspace() によって識別されるデフォルトの場所ではなく、ユーザ定義の場所に errno を配置できます。

2.6.1 戻り値

戻り値は、**int** 型の変数を指すポインタで、現在使用できる errno のインスタンスが格納されています。

2.6.2 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリでのスタティックデータの使用*（2-18 ページ）
- *C ライブラリによる __user_libspace スタティックデータ領域の使用*（2-20 ページ）

参照

- *__aeabi_errno_addr()*（2-4 ページ）

その他の情報

- *Application Binary Interface for the ARM Architecture*、http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036-/index.html

2.7 _findlocale()

rt_locale.h で定義される _findlocale() は、要求されたロケールに対応する一連の連続ロケールデータブロックを検索し、そのロケールを指すポインタを返します。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.7.1 構文

void const *_findlocale(**void const** **index*, **const char** **name*);

各パラメータには以下の意味があります。

index メモリ内で連続しており、終端値（LC_index_end マクロで設定）で終了する、一連のロケールデータブロックを指すポインタ。

name 検索するロケールの名前。

2.7.2 使用法

独自のロケール設定を定義するときに、_findlocale() をオプションヘルパ関数として使用できます。

_get_lc_ctype() などの _get_lc_*() 関数は、アセンブラマクロを使用して作成されたロケール定義を指すポインタを返します。1 つのロケール定義のみを記述する場合、常に同じポインタを返す _get_lc_ctype() の実装を記述できます。ただし、実行時に異なるロケールを使用する場合は、_get_lc_*() 関数が、引数として渡された名前に応じて別のデータブロックを返すことができる必要があります。_findlocale() は、これを実行する簡単な方法です。

2.7.3 戻り値

要求されたデータブロックを指すポインタを返します。

2.7.4 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリでのロケール関数をカスタマイズするためのアセンブラマクロ*（2-62 ページ）
- *C ライブラリのロケールサブシステムのリンク時における選択*（2-63 ページ）
- *C ライブラリのロケールサブシステムの実行時における選択*（2-66 ページ）
- *C ライブラリのロケールデータブロックの定義*（2-67 ページ）

参照

- *lconv 構造体*（2-18 ページ）

2.8 _fisatty()

stdio.h で定義されるこの関数は、指定された stdio ストリームが末端装置と通常のファイルのいずれに接続されるかを決定します。この関数は、ファイルハンドルに基づいて、_sys_istty() 低レベル関数を呼び出します。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.8.1 構文

int _fisatty(FILE *_stream_);

戻り値は、ストリームのデスティネーションを示します。

0 ファイル
1 端末
負 エラー

2.8.2 関連項目

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C および C++ ライブラリでの入出力関数のカスタマイズ*（2-93 ページ）

参照

- *_sys_istty()*（2-48 ページ）

2.9 _get_lconv()

locale.h で定義される _get_lconv() は、標準 C ライブラリ関数の localeconv() と同じ関数を実行しますが、内部のスタティック変数ではなく、ユーザ定義のメモリに結果を返す点が異なります。_get_lconv() は、数値形式に適した値を使用して、lconv 構造体のコンポーネントを設定します。

2.9.1 構文

```
void _get_lconv(struct lconv *lc);
```

2.9.2 使用法

この ISO C ライブラリ拡張では、スタティックデータは一切使用されません。ISO C 標準に厳密に準拠する必要のあるアプリケーションをビルドする場合には、localeconv() を代わりに使用して下さい。

2.9.3 戻り値

既存の lconv 構造体 *lc* に形式設定データが設定されます。

2.9.4 関連項目

参照

- *localeconv()*（2-20 ページ）

2.10 getenv()

stdlib.h に含まれている標準 C ライブラリの getenv() 関数です。指定された環境変数の値を取得します。

2.10.1 構文

char *getenv(**const char** *name);

2.10.2 使用法

デフォルトの実装では、使用できる環境情報がないことを示す NULL が返されます。

getenv() を再実装する場合、入力文字列 *name* を何らかの形式の環境リストで検索する方法で、再実装することを推奨します。一連の環境名と、環境リストを変更するメソッドは、実装によって定義されます。getenv() は他の関数に依存せず、getenv() に依存する関数もありません。

getenv() に密接に関連する関数は _getenv_init() です。_getenv_init() は、定義されていると起動中に呼び出され、ユーザによって再実装された getenv() が自身を初期化できるようにします。

2.10.3 戻り値

一致したリストメンバに関連付けられた文字列を指すポインタが返されます。参照された配列がプログラムによって修正されることはありませんが、getenv() へのその後の呼び出しによって上書きされる場合があります。

2.11 _getenv_init()

rt_misc.h で定義されるこの関数は、ユーザ定義の getenv() が自身を初期化できるようにします。C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.11.1 構文

```
void _getenv_init(void);
```

2.11.2 使用法

この関数が定義されている場合、C ライブラリ初期化コードは、ライブラリが初期化されるとき、つまり main() に入る前にこの関数を呼び出します。

2.12 __heapstats()

stdlib.h で定義されるこの関数は、ストレージ割り当てヒープの状態に関する統計情報を表示します。コンパイラのデフォルトの実装では、存在する未使用ブロックの数と、サイズ範囲の見積もり値に関する情報が返されます。

例 2-1 は、__heapstats() の出力例を示しています。

例 2-1 __heapstats() の出力

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

この出力の 1 行目には、合計バイト数、合計未使用ブロック数、平均サイズが示されています。以降の行には、各ブロックのサイズの見積り値がバイト単位で、範囲として示されています。__heapstats() では、使用ブロック数に関する情報は返されません。

この関数は、出力関数 *dprint()* を呼び出すことによって結果を出力します。この出力関数は fprintf() と同じように機能する必要があります。*dprint()* に渡される最初のパラメータは、提供されているポインタ *param* です。fprintf() 自体を渡すこともできますが、その場合は正しい関数ポインタ型にキャストする必要があります。この型は便宜上 **typedef** として定義されています。これは、__heapprt と呼ばれます。例えば、

```
__heapstats((__heapprt)fprintf, stderr);
```

注

まだ出力を送信していないストリームで fprintf() を呼び出すと、ライブラリは内部で malloc() を呼び出して、そのストリーム用のバッファを作成します。この動作が __heapstats() への呼び出し中に行われると、ヒープが破損することがあります。したがって、何らかの出力が stderr に送信されていることを確認する必要があります。

デフォルトである 1 領域メモリモデルを使用する場合、ヒープメモリは必要に応じて割り当てられます。つまり、ヒープの空き容量は、メモリの割り当てと解放を行うたびに変化します。例えば、以下のシーケンスがあるとします。

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);  // 未使用ヒープのサイズの初期値を出力
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);  // 解放後のヒープのサイズを出力
```

出力は以下のとおりです。

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.12.1 構文

void __heapstats(**int** (*_dprint_)(**void** *_param_, **char const** *format,...), **void** *_param_);

2.13 __heapvalid()

stdlib.h で定義されるこの関数は、ヒープの一貫性チェックを実行します。*verbose* パラメータにゼロ以外の値が設定されている場合は、各未使用ブロックに関する詳細情報が出力され、それ以外の場合にはエラーが出力されます。

この関数は、出力関数 *dprint()* を呼び出すことによって結果を出力します。この出力関数は fprintf() と同じように機能する必要があります。*dprint()* に渡される最初のパラメータは、提供されているポインタ *param* です。fprintf() 自体を渡すこともできますが、その場合は正しい関数ポインタ型にキャストする必要があります。この型は便宜上 **typedef** として定義されています。これは、__heapprt と呼ばれます。例えば、

例 2-2 fprintf() による __heapvalid() の呼び出し

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

注

まだ出力を送信していないストリームで fprintf() を呼び出すと、ライブラリは内部で malloc() を呼び出して、そのストリーム用のバッファを作成します。この動作が __heapvalid() への呼び出し中に行われると、ヒープが破損することがあります。したがって、何らかの出力が stderr に送信されていることを確認する必要があります。ストリームへの書き込みが行われていない場合、例 2-2 のコードを実行すると失敗します。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.13.1 構文

int __heapvalid(**int** (***dprint*)(**void** ***param*, **char const** ***format*,...), **void** ***param*, **int** *verbose*);

2.14 lconv 構造体

locale.h で定義される lconv 構造体は、数値形式情報を保持します。この構造体は、関数 _get_lconv() および localeconv() によって設定されます。

例 2-3 は、locale.h に含まれている lconv の定義を示しています。

例 2-3 lconv 構造体

```
struct lconv {
 char *decimal_point;
    /* The decimal point character used to format non monetary quantities */
 char *thousands_sep;
    /* The character used to separate groups of digits to the left of the */
    /* 使用される文字。    */
 char *grouping;
    /* A string whose elements indicate the size of each group of digits  */
    /* サイズを示す文字列。詳細については、以下を参照して下さい。*/
 char *int_curr_symbol;
    /* The international currency symbol applicable to the current locale.*/
    /* The first three characters contain the alphabetic international    */
    /* currency symbol in accordance with those specified in ISO 4217.   */
    /* 通貨および金額の表記に使用されるコード。4 文字目    */
    /* character (immediately preceding the null character) is the       */
    /* character used to separate the international currency symbol from  */
    /* 文字。                                */
 char *currency_symbol;
    /* 現在のロケールで使用できるローカル通貨記号。      */
 char *mon_decimal_point;
    /* 通貨数量の書式設定に使用される小数点。          */
 char *mon_thousands_sep;
    /* The separator for groups of digits to the left of the decimal point*/
    /* 使用される文字。                          */
 char *mon_grouping;
    /* A string whose elements indicate the size of each group of digits  */
    /* 使用される文字。詳細については、以下を参照して下さい。    */
 char *positive_sign;
    /* The string used to indicate a non negative-valued formatted        */
    /* 文字列。                                */
 char *negative_sign;
    /* The string used to indicate a negative-valued formatted monetary   */
    /* 文字列。                                    */
 char int_frac_digits;
    /* The number of fractional digits (those to the right of the         */
    /* decimal point) to be displayed in an internationally formatted     */
    /* 桁数)。                                */
 char frac_digits;
    /* The number of fractional digits (those to the right of the         */
    /* 小数の桁数（小数点の右側の桁数)。 */
 char p_cs_precedes;
    /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
    /* 値の前に付加する場合は 1、後に付加する場合は 0 を設定します。*/
 char p_sep_by_space;
    /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
    /* separated by a space from the value for a non negative formatted   */
    /* 文字列。                                */
 char n_cs_precedes;
    /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
    /* 値の前に付加する場合は 1、後に付加する場合は 0 を設定します。   */
```

```
 char n_sep_by_space;
    /* Set to 1 or 0 if the currency_symbol respectively is or is not    */
    /* separated by a space from the value for a negative formatted      */
    /* 文字列。                                                          */
 char p_sign_posn;
    /* Set to a value indicating the position of the positive_sign for a */
    /* 位置を示す値を設定します。See below for more details*/
 char n_sign_posn;
    /* Set to a value indicating the position of the negative_sign for a */
    /* 位置を示す値を設定します。*/
};
```

この例では、以下のようになります。

- `grouping` と `mon_grouping` の要素（例 2-3 （2-18 ページ）に示されている）は、以下のように解釈されます。

  `CHAR_MAX` それ以上のグループ化は実行されないことを示しています。

  `0` 残りの桁で前の要素が繰り返されることを示しています。

  `その他` その値が、現在のグループを構成する桁数を示します。次の要素が検査されて、現在のグループの左側にくる次のグループの桁数が決定されます。

- `p_sign_posn` と `n_sign_posn` の値（例 2-3 （2-18 ページ）に示されている）は、以下のように解釈されます。

  `0` 数量と通貨記号が括弧で囲まれます。

  `1` 数量と通貨記号の前に符号文字列が付けられます。

  `2` 数量と通貨記号の後に符号文字列が付けられます。

  `3` 通貨記号の直前に符号文字列が付けられます。

  `4` 通貨記号の直後に符号文字列が付けられます。

2.14.1 関連項目

参照

- *_get_lconv()*（2-12 ページ）
- *localeconv()*（2-20 ページ）

2.15 localeconv()

stdlib.h で定義される localeconv() は、現在のロケールの規則に基づいた数値形式に適した値を使用して、lconv 構造体のコンポーネントを作成して設定します。

2.15.1 構文

```
struct lconv *localeconv(void);
```

2.15.2 使用法

この構造体の **char** * 型のメンバは文字列です。decimal_point を除き、これらのメンバは空文字列 "" を指すことによって、その値が現在のロケールでは使用できないか、長さがゼロの文字列であることを示すことができます。

char 型のメンバは、負ではない数値を示します。これらのメンバが CHAR_MAX である場合は、その値が現在のロケールで使用できないことを示しています。

この関数は内部のスタティックバッファを使用するため、スレッドセーフではありません。_get_lconv() は代わりに使用できるスレッドセーフな関数です。

2.15.3 戻り値

この関数は、埋め込まれたオブジェクトを指すポインタを返します。この戻り値が指す構造体はプログラムによって修正されませんが、それ以降の localeconv() 関数の呼び出しによって上書きされる場合があります。また、LC_ALL、LC_MONETARY、LC_NUMERIC のいずれかのカテゴリを使用した setlocale() 関数の呼び出しによって、構造体の内容が上書きされる場合があります。

2.15.4 関連項目

参照

- *_get_lconv()*（2-12 ページ）
- *lconv 構造体*（2-18 ページ）
- *setlocale()*（2-36 ページ）

2.16 _membitcpybl()、_membitcpybb()、_membitcpyhl()、_membitcpyhb()、_membitcpywl()、_membitcpywb()、_membitmovebl()、_membitmovebb()、_membitmovehl()、_membitmovehb()、_membitmovewl()、_membitmovewb()

標準 C ライブラリの memcpy() 関数および memmove() 関数と同様、これらの非標準 C ライブラリ関数は、ビット境界で整列されたメモリ操作を実行します。これらは、string.h 内で定義されています。

2.16.1 構文

void _membitcpy[b|h|w][b|l](**void** *_dest_, **const void** *_src_, **int** _dest_offset_, **int** _src_offset_, size_t _nbits_);
void _membitmove[b|h|w][b|l](**void** *_dest_, **const void** *_src_, **int** _dest_offset_, **int** _src_offset_, size_t _nbits_);

2.16.2 使用法

src が指すアドレスの *src_offset* ビット後の（負のオフセットの場合は前の）メモリ位置から、*dest* が指すアドレスの *dest_offset* ビット後の（負のオフセットの場合は前の）メモリ位置へ、*nbits* によって指定された連続するビット数を、（使用される関数に応じて）コピーまたは移動します。

一連のビットを定義するには、順序の形式が必要です。この順序は、各関数のバリアントで次のように定義されます。

- 最後から 2 番目の文字が b の関数（_membitcpybl() など）は、バイト指向です。バイト指向関数では、1 バイト内のすべてのビットが、次の 1 バイト内のビットより前に置かれていると見なされます。
- 最後から 2 番目の文字が h の関数は、ハーフワード指向です。
- 最後から 2 番目の文字が w の関数は、ワード指向です。

それぞれのバイト、ハーフワード、またはワード内のビット順序は、エンディアン方式によって異なります。最後の文字が b の関数（_membitmovewb() など）は、ビット単位のビッグエンディアンです。つまり、各バイト、ハーフワード、またはワードの *最上位ビット*（MSB）（存在する場合）がそのワード内の先頭ビットと見なされ、*最下位ビット*（LSB）が最後のビットと見なされます。最後の文字が l の関数は、ビット単位のリトルエンディアンです。この関数では、LSB が先頭ビット、MSB が最後のビットと見なされます。

memcpy() および memmove() と同様、ビット単位のメモリコピー関数では、コピー元およびコピー先のメモリ領域が重複しないと想定される最も速いタイミングでコピーが実行され、ビット単位のメモリ移動関数では、重複領域内の移動元データがコピーされてから上書きされます。

リトルエンディアンプラットフォームでは、ビット単位のビッグエンディアン関数はそれぞれ動作が異なります。一方、ビット単位のリトルエンディアン関数は同一のビット順序を使用するため、同じ関数を表す同義的なシンボルです。ビッグエンディアンプラットフォームでは、ビット単位のビッグエンディアン関数は事実上すべて同一の関数ですが、ビット単位のリトルエンディアン関数はそれぞれ動作が異なります。

2.17 posix_memalign()

stdlib.h で定義されるこの関数は、メモリの整列割り当てを実行します。これは、POSIX に完全に準拠しています。

2.17.1 構文

int posix_memalign(**void** ***memptr*, size_t *alignment*, size_t *size*);

2.17.2 使用法

この関数は、*size* バイトのメモリを *alignment* の倍数であるアドレスに割り当てます。

alignment の値は、2 の累乗および sizeof(void *) の倍数である必要があります。

posix_memalign() によって割り当てられたメモリは、標準 C ライブラリの free() 関数を使用して解放できます。

2.17.3 戻り値

返されたアドレスは、*memptr* が指す void * 変数に書き込まれます。

関数からの整数の戻り値は、成功時はゼロ、失敗時はエラーコードです。

要求された size および alignment のメモリブロックがない場合、関数は ENOMEM を返し、**memptr* の値は未定義になります。

2.17.4 関連項目

その他の情報

- Open Group Base Specifications、*IEEE Std 1003.1*、http://www.opengroup.org

2.18 __raise()

rt_misc.h で定義されるこの関数は、ランタイム異常を示すシグナルを生成します。C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.18.1 構文

int __raise(**int** *signal*, **int** *type*);

各パラメータには以下の意味があります。

signal シグナルの番号を保持する整数です。

type 一部のシグナルの種類で、シグナルが生成された状況に関する追加情報を表す、整数、文字列定数、または変数です。

2.18.2 使用法

signal() 関数を呼び出すことによって、信号の処理を設定している場合、__raise() は、ユーザが指定したアクションを取ります。つまり、信号を無視するか、ユーザ提供のハンドラ関数を呼び出します。そうでない場合、__raise() は、デフォルトの信号処理動作を提供する __default_signal_handler() を呼び出します。

__raise() 関数は、以下を定義することによって置き換えられます。

int __raise(**int** *signal*, **int** *type*);

この定義によって、C 信号メカニズムとデータを消費する信号ハンドラベクタをバイパスできますが、それ以外には基本的に以下と同じインタフェースが提供されます。

int __default_signal_handler(**int** *signal*, **int** *type*);

ライブラリのデフォルトのシグナルハンドラは __raise() の *type* パラメータを使用して、出力するメッセージを変化させます。

2.18.3 戻り値

__raise() には、以下の 3 つの復帰条件が考えられます。

復帰しない ハンドラによって長距離のジャンプまたは再起動が実行されます。

0 シグナルが処理されたことを示します。

ゼロ以外 呼び出しコードによって戻り値を終了コードに渡す必要があります。デフォルトのライブラリの実装では、__raise() によってゼロ以外の復帰コード *rc* が返されると、_sys_exit(rc) が呼び出されます。

2.18.4 関連項目

概念

『ARM C および C++ ライブラリと浮動小数点サポートの使用』:

- *ARM C ライブラリでのスレッドセーフティ*（2-29 ページ）

参照

- *__default_signal_handler()*（2-8 ページ）

- `_sys_exit()`（2-46 ページ）
- `_ttywrch()`（2-56 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の間接セミホスティング依存関数*（2-38 ページ）
- *C ライブラリの `signal()` 関数と追加の型引数によりサポートされる ISO 準拠のシグナルの実装*（2-113 ページ）

2.19 _rand_r()

stdlib.h で定義されるこの関数は、rand() 関数の再入可能バージョンです。

2.19.1 構文

int __rand_r(**struct _rand_state *** *buffer*);

各パラメータには以下の意味があります。

buffer 乱数ジェネレータの状態を格納する、ユーザ定義のバッファを指すポインタです。

2.19.2 使用法

この関数を使用すると、スレッドローカルストレージ内にある独自のバッファを明示的に指定できます。

2.19.3 関連項目

参照

- *スレッドセーフではない C ライブラリ関数*（2-67 ページ）
- *_srand_r()*（2-38 ページ）

2.20 remove()

stdio.h に含まれている標準 C ライブラリの remove() 関数です。

2.20.1 構文

int remove(**const char** ***filename**);

2.20.2 使用法

この関数のデフォルト実装では、セミホスト機能が使用されます。

remove() によって、*filename* が指す文字列と同じ名前のファイルが削除されます。そのファイルが再度作成されない限り、それ以降にそのファイルを開こうとしても失敗します。ファイルが開いている場合の remove() 関数の動作は実装定義です。

2.20.3 戻り値

処理に成功するとゼロが返され、失敗するとゼロ以外の値が返されます。

2.20.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.21 rename()

stdio.h に含まれている標準 C ライブラリの rename() 関数です。

2.21.1 構文

int rename(**const char** **old*, **const char** **new*);

2.21.2 使用法

この関数のデフォルト実装では、セミホスト機能が使用されます。

rename() によって、*old* が指す文字列と同じ名前のファイルが、それ以降は *new* が指す文字列の名前で認識されます。*old* が指す文字列が名前として付けられているファイルは事実上削除されます。*new* が指す文字列によって識別されるファイルが rename() 関数呼び出しよりも前に存在していた場合の動作は実装定義です。

2.21.3 戻り値

処理に成功するとゼロが返され、失敗するとゼロ以外の値が返されます。ゼロ以外の値が返され、ファイルが既に存在した場合、そのファイルはそれ以降も元の名前で識別されます。

2.21.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.22 __rt_entry

シンボル __rt_entry は、ARM C ライブラリを使用するプログラムの開始位置です。分散ロードされたすべての領域がそれぞれの実行アドレスに再配置された後、制御が __rt_entry に渡されます。

2.22.1 使用法

デフォルトでは、__rt_entry は以下のように実装されます。

1. ヒープとスタックをセットアップします。
2. __rt_lib_init を呼び出して C ライブラリを初期化します。
3. main() を呼び出します。
4. __rt_lib_shutdown を呼び出して C ライブラリを終了します。
5. 終了します。

__rt_entry は、以下のいずれかの関数への呼び出しで終了する必要があります。

exit() atexit() で登録された関数を呼び出し、ライブラリを終了します。

__rt_exit() ライブラリは終了しますが、atexit() 関数は呼び出しません。

_sys_exit() 終了して実行環境に直接戻ります。ライブラリの終了も、atexit() 関数の呼び出しも実行されません。

2.23 __rt_exit()

rt_misc.h で定義されるこの関数は、ライブラリを終了しますが、atexit() で登録された関数は呼び出しません。atexit() で登録された関数は exit() によって呼び出されます。

__rt_exit() は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.23.1 構文

void __rt_exit(**int** *code*);

code は、標準関数では使用されません。

2.23.2 使用法

__rt_lib_shutdown() を呼び出すことによって C ライブラリを終了し、その後 _sys_exit() を呼び出してアプリケーションを終了します。__rt_exit() ではなく、_sys_exit() を再実装します。

2.23.3 戻り値

この関数は値を返しません。

2.24 __rt_fp_status_addr()

rt_fp.h で定義されるこの関数は、デフォルトでは __user_libspace に配置される浮動小数点ステータスワードのアドレスを返します。C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.24.1 構文

```
unsigned *__rt_fp_status_addr(void);
```

2.24.2 使用法

__rt_fp_status_addr() が定義されていない場合は、C ライブラリのデフォルトの実装が使用されます。値は、__rt_lib_init() が _fp_init() を呼び出したときに初期化されます。ステータスワードの定数は、fenv.h 内に記載されています。デフォルトの浮動小数点ステータスは 0 です。

2.24.3 戻り値

浮動小数点ステータスワードのアドレスです。

2.24.4 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *ARM C ライブラリでのスレッドセーフティ*（2-29 ページ）

2.25 __rt_heap_extend()

rt_heap.h で定義されるこの関数は、可能な場合、ヒープに追加するための新しいメモリブロックを返します。

__rt_stackheap_init() を再実装する場合は、この関数を再実装する必要があります。プロトタイプの実装例が rt_memory.s に収録されています。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.25.1 構文

extern size_t __rt_heap_extend(**size_t** *size*, **void** ***block*);

2.25.2 使用法

呼び出し規則は通常の AAPCS に準じます。エントリ時、r0 は追加されるブロックの最小サイズを保持し、r1 はベースアドレスが保存される場所を指すポインタを保持します。

デフォルトの実装には以下の特性があります。

- 以下のいずれかのサイズが返されます。
  - AArch32 状態で要求されたサイズ以上の 8 バイトの倍数
  - AArch64 状態で要求されたサイズ以上の 16 バイトの倍数
  - 要求を満たせない場合は 0
- 返されるベースアドレスは以下で整列されます。
  - AArch32 状態では、8 バイト境界
  - AArch64 状態では、16 バイト境界
- サイズはバイト単位で処理されること。
- この関数には、*ARM アーキテクチャ向けプロシージャコール標準*（AAPCS）の制約条件のみが適用されます。

2.25.3 戻り値

デフォルトの実装では、十分な空きヒープメモリがある場合にヒープが拡張されます。それが不可能な場合、実装されていれば、__user_heap_extend() が呼び出されます。終了処理では、r0 は取得されたブロックのサイズ、または何も取得されなかった場合は 0 を保持し、エントリ時に r1 が指したメモリ位置にブロックのベースアドレスが保持されます。

2.25.4 関連項目

参照

- *__rt_stackheap_init()*（2-35 ページ）
- *__user_heap_extend()*（2-57 ページ）

その他の情報

- *ARM アーキテクチャ向けプロシージャコール標準*、http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html

2.26 __rt_lib_init()

rt_misc.h で定義されるこの関数は、__rt_lib_shutdown() の対となるライブラリ初期化関数です。

2.26.1 構文

extern value_in_regs **struct** __argc_argv __rt_lib_init(**unsigned** *heapbase*, **unsigned** *heaptop*);

各パラメータには以下の意味があります。

heapbase ヒープメモリブロックの開始位置です。

heaptop ヒープメモリブロックの終了位置です。

2.26.2 使用法

この関数は __rt_stackheap_init() の直後に呼び出され、ヒープとして使用される最初のメモリチャンクに渡されます。この関数は標準 ARM C ライブラリ初期化関数であり、再実装することはできません。

2.26.3 戻り値

この関数は、main() に渡される argc と argv を返します。構造体は以下のようにレジスタに返されます。

```
struct __argc_argv
{  int argc;
   char **argv;
   int r2, r3;  // main() へのエントリ時にレジスタ R2 および R3 に
};          // 格納されているオプションの追加引数。
```

2.27 __rt_lib_shutdown()

rt_misc.h で定義されるこの関数は、__rt_lib_init() の対となるライブラリ終了関数です。

2.27.1 構文

void __rt_lib_shutdown(**void**);

2.27.2 使用法

この関数は、ユーザがこの関数を直接呼び出す必要があるときに備えて提供されています。この関数は標準 ARM C ライブラリ終了関数であり、再実装することはできません。

2.28 __rt_raise()

rt_misc.h で定義されるこの関数は、ランタイム異常を示すシグナルを生成します。C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張機能としてサポートされています。

2.28.1 構文

void __rt_raise(**int** *signal*, **int** *type*);

各項目には以下の意味があります。

signal シグナルの番号を保持する整数です。

type 一部のシグナルの種類で、シグナルが生成された状況に関する追加情報を表す、整数、文字列定数、または変数です。

2.28.2 使用法

この関数を再定義すると、ライブラリのシグナル処理メカニズム全体を置き換えることができます。デフォルトの実装では __raise() が呼び出されます。

__raise() によって以下のいずれかの値が返されます。

復帰しない ハンドラによってロングジャンプまたは再起動が実行され、制御は __rt_raise() に戻されません。

0 シグナルが処理され、__rt_raise() が終了します。

ゼロ以外 デフォルトのライブラリの実装では、__raise() によってゼロ以外の復帰コード *rc* が返されると、_sys_exit(rc) が呼び出されます。

2.28.3 関連項目

参照

- *__raise()*（2-23 ページ）
- *_sys_exit()*（2-46 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリの signal() 関数と追加の型引数によりサポートされる ISO 準拠のシグナルの実装*（2-113 ページ）

2.29 __rt_stackheap_init()

rt_misc.h で定義されるこの関数は、スタックポインタを設定して、初期ヒープとして使用するメモリ領域を返します。この関数はライブラリ初期化コードから呼び出されます。

この関数からの戻り値では、SP がスタック領域の最上部、r0 がヒープ領域のベース、r1 がヒープ領域のリミットを指す必要があります。

ユーザ定義のメモリモデル（つまり、__rt_stackheap_init() と __rt_heap_extend()）には、__user_perproc_libspace 領域からの 16 バイトのストレージが必要に応じて割り当てられます。メモリモデルは、__rt_stackheap_storage() を呼び出して 16 バイト領域を指すポインタを返すことによって、このストレージにアクセスします。

この関数は C ライブラリ標準にはありませんが、ARM C ライブラリでは拡張としてサポートしています。

2.29.1 関連項目

参照

- *__rt_heap_extend()*（2-31 ページ）

2.30 setlocale()

locale.h で定義されるこの関数は、*category* 引数と *locale* 引数によって指定された、適切なロケールを選択します。

2.30.1 構文

char *setlocale(**int** *category*, **const char** **locale*);

2.30.2 使用法

setlocale() 関数は、現在のロケールの一部または全部を、変更または照会するために使用されます。それぞれの値に対応する *category* 引数の影響は、以下のとおりです。

LC_COLLATE　strcoll() の動作に影響を与えます。

LC_CTYPE　文字処理関数の動作に影響を与えます。

LC_MONETARY　localeconv() によって返される通貨形式設定情報に影響を与えます。

LC_NUMERIC　形式指定付き入出力関数と文字列変換関数の小数点文字と、localeconv() によって返される数値形式情報に影響を与えます。

LC_TIME　strftime() の動作に影響を与える可能性があります。現在サポートされているロケールに関しては、このオプションによる影響はありません。

LC_ALL　すべてのロケールカテゴリに影響を与えます。上記のすべてのロケールカテゴリのビット単位論理和が取られます。

locale に値 "C" を指定すると、C 変換の最小限の環境が指定されます。*locale* に空文字列 "" を指定すると、実装定義のネイティブ環境が指定されます。プログラム起動時には、setlocale(LC_ALL, "C") と等価な処理が実行されます。

locale の有効な値は、インポートした __use_*X*_ctype シンボル（__use_iso8859_ctype、__use_sjis_ctype、__use_utf8_ctypte）と、ユーザ定義ロケールによって異なります。

2.30.3 戻り値

locale の文字列へのポインタが渡され、その選択が有効である場合には、新しいロケールに指定されたカテゴリと関連する文字列が返されます。その選択が無効な場合は NULL ポインタが返され、ロケールは変更されません。

locale に NULL ポインタが渡されると、現在のロケールカテゴリに関する文字列が返され、ロケールは変更されません。

category が LC_ALL であり、ロケールを適切に設定した直前の呼び出しで LC_ALL 以外のカテゴリが使用された場合には、混合文字列が返されることがあります。関連カテゴリのその後の呼び出しで使用されたときに返される文字列によって、その部分のプログラムロケールが復元されます。返される文字列はプログラムによって修正されることはありませんが、setlocale() へのその後の呼び出しによって上書きされる場合があります。

2.30.4 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *ISO8859-1 の実装*（2-64 ページ）
- *Shift-JIS および UTF-8 の実装*（2-65 ページ）
- *C ライブラリのロケールデータブロックの定義*（2-67 ページ）

参照

- *lconv 構造体*（2-18 ページ）

2.31 _srand_r()

stdlib.h で定義されるこの関数は、srand() 関数の再入可能バージョンです。

2.31.1 構文

int __srand_r(**struct _rand_state *** *buffer*, **unsigned int** *seed*);

各パラメータには以下の意味があります。

buffer 乱数ジェネレータの状態を格納する、ユーザ定義のバッファを指すポインタです。

seed _rand_r() へのその後の呼び出しによって返される、新しい一連の疑似乱数のシードです。

2.31.2 使用法

この関数を使用すると、スレッドローカルストレージで使用できる独自のバッファを明示的に指定できます。

_srand_r() が同じシード値で繰り返し呼び出される場合、一連の疑似乱数は同じ値が繰り返されます。_rand_r() を呼び出した後で、_srand_r() への呼び出しを同一バッファで行うと、バッファが初期化されていないため、動作は定義されていません。

2.31.3 関連項目

参照

- *スレッドセーフではない C ライブラリ関数*（2-67 ページ）
- *_rand_r()*（2-25 ページ）

2.32 strcasecmp()

string.h で定義されるこの関数は、大文字と小文字を区別しない文字列変換テストを実行します。

2.32.1 構文

extern _ARMABI **int** strcasecmp(const **char** *s1, const **char** *s2);

2.32.2 関連項目

その他の情報

- *『Application Binary Interface (ABI) for the ARM Architecture』*、http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi

2.33 strncasecmp()

string.h で定義されるこの関数は、指定された文字数を超えない、大文字と小文字を区別しない文字列変換テストを実行します。

2.33.1 構文

```
extern _ARMABI int strncasecmp(const char *s1, const char *s2, size_t n);
```

2.33.2 関連項目

その他の情報

- *『Application Binary Interface (ABI) for the ARM Architecture』*、http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi

2.34 strlcat()

string.h で定義されるこの関数は、2 つの文字列を連結します。この関数は、最大 *size*-strlen(*dst*)-1 バイトを、NULL で終わる文字列 *src* から *dst* の末尾に追加します。この関数は、長さだけでなくバッファの完全なサイズを取得し、*size* が 0 より大きい場合には、結果を NULL で終端します。*size* 値には、NULL のためのバイトを含めて下さい。

strlcat() 関数は、長さに制限がない場合に作成される文字列の長さの合計を返します。割り当てられている長さが十分かどうかによって、*実際に*作成される文字列の長さと等しい場合も、等しくない場合もあります。どれくらいの長さが必要かを調べるために、まず strlcat() を呼び出し、長さが不十分な場合は十分な長さを割り当て、再度 strlcat() を呼び出して必要な文字列を作成することができます。

この関数は、多くの C ライブラリで一般的な BSD 由来の拡張です。

2.34.1 構文

extern size_t strlcat(**char** **dst*, **src*, size_t *size*);

2.35 strlcpy()

string.h で定義されるこの関数は、最大 *size*-1 文字を、NULL で終わる文字列 *src* から *dst* にコピーします。この関数は、長さだけでなくバッファの完全なサイズを取得し、*size* が 0 より大きい場合には、結果を NULL で終端します。*size* 値には、NULL のためのバイトを含めて下さい。

strlcpy() 関数は、長さに制限がない場合に作成される文字列の長さの合計を返します。割り当てられている長さが十分かどうかによって、*実際に*コピーされる文字列の長さと等しい場合も、等しくない場合もあります。どれくらいの長さが必要かを調べるために、まず strlcpy() を呼び出し、長さが不十分な場合は十分な長さを割り当て、再度 strlcpy() を呼び出して必要なコピーを行うことができます。

この関数は、多くの C ライブラリで一般的な BSD 由来の拡張です。

2.35.1 構文

extern size_t strlcpy(**char** **dst*, **const char** **src*, size_t *size*);

2.36 _sys_close()

rt_sys.h で定義されるこの関数は、前に _sys_open() で開いたファイルを閉じます。

2.36.1 構文

int _sys_close(FILEHANDLE *fh*);

2.36.2 使用法

何らかの入出力関数が使用される場合、この関数を定義する必要があります。

2.36.3 戻り値

正常に実行されると、0 が返されます。エラーが発生すると、ゼロ以外の値が返されます。

2.36.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.37 _sys_command_string()

rt_sys.h で定義されるこの関数は、現在のアプリケーションを呼び出した環境から、そのアプリケーションを呼び出すコマンドラインを取得します。

2.37.1 構文

char *_sys_command_string(**char** **cmd*, **int** *len*);

各項目には以下の意味があります。

cmd コマンドラインを保存できるバッファを指すポインタです。コマンドラインを *cmd* に保存する必要はありません。

len バッファの長さです。

2.37.2 使用法

この関数がライブラリ起動コードによって呼び出されることで、argv と argc が設定され、main() に渡されます。

注

この関数が呼び出されるときに、C ライブラリが完全に初期化されていると想定しないで下さい。例えば、この関数内から malloc() を呼び出してはいけません。これは、C ライブラリの起動シーケンスによって、ヒープが完全に設定される前にこの関数が呼び出されるからです。

2.37.3 戻り値

この関数は以下のいずれかの値を返す必要があります。

- 正常に実行された場合は、コマンドラインを指すポインタが返されます。このポインタは、*cmd* バッファを指すポインタ（バッファが使用されている場合）、またはコマンドラインが保存される他の場所を指すポインタとなります。
- 実行に失敗した場合は NULL が返されます。

2.37.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.38 _sys_ensure()

この関数は廃止される予定です。この関数が、他のライブラリ関数から呼び出されることはありません。また、標準 I/O（stdio）のターゲットを変更する場合、この関数を再実装する必要はありません。

2.39 _sys_exit()

rt_sys.h で定義されるこの関数は、ライブラリ終了関数です。_sys_exit() は、ライブラリからのすべての終了処理によって最終的に呼び出されます。

2.39.1 構文

void _sys_exit(**int** *return_code*);

2.39.2 使用法

この関数は値を返しません。アプリケーションの終了は、以下のいずれかによって上位レベルでインターセプトできます。

- アプリケーションの一部として、C ライブラリ関数 exit() を実装する。この場合は、atexit() が処理されず、ライブラリは終了されません。
- アプリケーションの一部として、関数 __rt_exit(int n) を実装する。この場合はライブラリは終了されません。ただし、atexit() の処理は、exit() が呼び出された場合、または main() から復帰した場合に実行されます。

2.39.3 戻り値

復帰コードは参考用です。実装によっては、復帰コードを実行環境に渡そうとする場合があります。

2.39.4 関連項目

参照

- *__rt_exit()*（2-29 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.40 _sys_flen()

rt_sys.h で定義されるこの関数は、ファイルの現在の長さを返します。

2.40.1 構文

long _sys_flen(FILEHANDLE *fh*);

2.40.2 使用法

この関数は、_sys_seek() が、ファイルの末尾に対する相対オフセットをファイルの開始位置に対する相対オフセットに変換するために使用されます。

fseek() を使用しない場合は、_sys_flen() を定義する必要はありません。

システム _sys_*() レベルでリターゲットする場合は、ファイルの末尾を基準にしたシークが基礎システムで直接サポートされている場合であっても、_sys_flen() を定義する必要があります。

2.40.3 戻り値

この関数は、ファイルの現在の長さ *fh*、またはエラーを示す負の値を返します。

2.40.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド :

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.41 _sys_istty()

rt_sys.h で定義されるこの関数は、ファイルハンドルによって端末が識別されるかどうかを決定します。

2.41.1 構文

int _sys_istty(FILEHANDLE *fh*);

2.41.2 使用法

ファイルが端末装置に接続されている場合にこの関数が使用されると、デフォルトでバッファリングが行われず（set(v)buf 呼び出しが行われず）、シークが行われません。

2.41.3 戻り値

以下のいずれかの値が返されます。

0 関連付けられた装置がないことを示します。

1 関連付けられた装置があることを示します。

その他 エラーが発生したことを示します。

2.41.4 関連項目

参照

- *_fisatty()*（2-11 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.42 _sys_open()

rt_sys.h で定義されるこの関数は、ファイルを開きます。

2.42.1 構文

FILEHANDLE _sys_open(**const char** **name*, **int** *openmode*);

2.42.2 使用法

_sys_open() 関数は、fopen() および freopen() の実行に必要です。また、これらの 2 つの関数はファイル入出力関数が使用される場合に必要です。

openmode パラメータはビットマップです。これらのビットのほとんどは ISO モード仕様に直接対応しています。ターゲットに依存する拡張も可能ですが、その場合は freopen() も拡張する必要があります。

2.42.3 戻り値

エラー発生時の戻り値は -1 です。

2.42.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド :

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.43 _sys_read()

rt_sys.h で定義されるこの関数は、ファイルの内容をバッファに読み出します。

2.43.1 構文

int _sys_read(FILEHANDLE *fh*, **unsigned char** **buf*, **unsigned** *len*, **int** *mode*);

注

mode パラメータは、ここでは履歴のために使用されます。それ以外に有用な情報は含まれていないため、無視して下さい。

2.43.2 戻り値

以下のいずれかの値が返されます。

- 読みだされなかったバイト数（つまり、読みだされたバイト数は *len - result*）。
- エラー通知。
- EOF インジケータ。通常は、0x80000000 が設定されることで EOF が通知されます。

データの最後のバイトまで読み出しても EOF インジケータは通知されません。データの最後のバイトを超えて読み取りが試行されたときに、EOF インジケータが通知されます。ターゲット依存コードでは、以下の処理を実行できます。

- EOF の前の残りのバイト数のデータと同じ読み取り処理で、EOF インジケータを返す。
- 前の読み取り処理で残りのバイト数のデータが返された後で、EOF インジケータのみを返す。

2.43.3 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.44 _sys_seek()

rt_sys.h で定義されるこの関数は、ファイルポインタをファイルの開始位置からのオフセット *pos* に置きます。

2.44.1 構文

int _sys_seek(FILEHANDLE *fh*, **long** pos);

2.44.2 使用法

この関数は、現在の読み出し位置または書き込み位置を、現在のファイル *fh* の開始位置を基準にした新しい *pos* に設定します。

2.44.3 戻り値

次の値が返されます。

- エラーが発生しなかった場合は負以外の値。
- エラー発生時は負の値。

2.44.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.45 _sys_tmpnam()

rt_sys.h で定義されるこの関数は、一時ファイルのファイル番号 *fileno* を、tmp0001 などの一意のファイル名に変換します。

2.45.1 構文

void _sys_tmpnam(**char** **name*, **int** *fileno*, **unsigned** *maxlength*);

2.45.2 使用法

tmpnam() または tmpfile() が使用される場合は、この関数を定義する必要があります。

2.45.3 戻り値

name に指定されているファイル名を返します。

2.45.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.46 _sys_write()

rt_sys.h で定義されるこの関数は、バッファの内容を、事前に _sys_open() で開かれたファイルに書き込みます。

2.46.1 構文

int _sys_write(FILEHANDLE *fh*, **const unsigned char** **buf*, **unsigned** *len*, **int** *mode*);

注

mode パラメータは、ここでは履歴のために使用されます。それ以外に有用な情報は含まれていないため、無視して下さい。

2.46.2 戻り値

以下のいずれかの値が返されます。

- 書き込まれなかった文字の数を表す正の値（したがってゼロ以外の値が返された場合は、何らかの失敗を意味します）。
- エラーを示す負の値。

2.46.3 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.47 system()

stdlib.h に含まれている標準 C ライブラリの system() 関数です。

2.47.1 構文

int system(**const char** *_string_);

2.47.2 使用法

この関数のデフォルト実装では、セミホスト機能が使用されます。

system() は、*string* が指す文字列をホスト環境に渡し、実装定義の方法によってコマンドプロセッサで実行します。*string* に NULL ポインタを使用することで、コマンドプロセッサの有無を照会できます。

2.47.3 戻り値

引数が NULL ポインタで、コマンドプロセッサを使用できる場合に限り、このシステム関数はゼロ以外の値を返します。

引数が NULL ポインタでない場合には、system() 関数は実装定義の値を返します。

2.47.4 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.48 time()

time.h に含まれている標準 C ライブラリの time() 関数です。

この関数のデフォルト実装では、セミホスト機能が使用されます。

2.48.1 構文

time_t time(time_t *timer);

現在のカレンダータイムの近似値が返されます。

2.48.2 戻り値

暦時間を取得できない場合には、-1 の値が返されます。*timer* が NULL ポインタでない場合には、戻り値が *timer* にも格納されます。

2.48.3 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.49 _ttywrch()

rt_sys.h で定義されるこの関数は、コンソールに文字を書き込みます。コンソール出力はリダイレクトされている場合があります。この関数は、適切なエラー処理手段がない場合の最終的なエラー処理ルーチンとして使用できます。

2.49.1 構文

void __ttywrch(**int** *ch*);

2.49.2 使用法

この関数のデフォルト実装では、セミホスト機能が使用されます。

他の入出力が存在しない場合でも、この関数または __raise() を再定義できます。例えば、この関数によって、非揮発性メモリ内に保存されているログにエラーメッセージを書き込むことができます。

2.49.3 関連項目

参照

- *__raise()*（2-23 ページ）

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *C ライブラリ関数の直接セミホスティング依存関数*（2-37 ページ）

2.50 __user_heap_extend()

rt_misc.h で定義されるこの関数は、初期メモリブロックとは別の、ヒープによって使用されるメモリブロックを返すように定義できます。

定義すると、ヒープ拡張ブロックのサイズとベースアドレスが以下のように返されます。

- AArch32 状態では、ヒープ拡張ブロックは、8 バイトで整列します。
- AArch64 状態では、ヒープ拡張ブロックは、16 バイトで整列します。

2.50.1 構文

extern size_t __user_heap_extend(**int** *var0*, **void** ***base*, **size_t** *requested_size*);

2.50.2 使用法

この関数のデフォルトの実装はありません。この関数を定義する場合は、以下の条件を満たす必要があります。

- 返されるサイズは以下のいずれかであること。
  - AArch32 状態で要求されたサイズ以上の 8 バイトの倍数。
  - AArch64 状態で要求されたサイズ以上の 16 バイトの倍数。
  - 要求を満たせない場合は 0。
- サイズはバイト単位で処理されること。
- この関数には、*ARM アーキテクチャ向けプロシージャコール標準*（AAPCS）の制約条件のみが適用されること。
- 最初の引数はエントリ時に常にゼロとなり、無視できること。ベースはこの引数を保持するレジスタに返されること。
- 返されるベースアドレスは、以下のように整列する必要があります。
  - AArch32 状態では、8 バイト境界。
  - AArch64 状態では、16 バイト境界。

2.50.3 戻り値

この関数は、要求されたサイズまたはそれより大きいブロックへのポインタを **base* に設定して、そのブロックのサイズを返します。条件に一致するブロックを返せない場合は、0 が返されます。この場合、**base* に格納された値は使用されません。

2.50.4 関連項目

その他の情報

- *ARM アーキテクチャ向けプロシージャコール標準*、http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html

2.51 __user_heap_extent()

定義されている場合、この関数はヒープのベースアドレスと最大領域を返します。rt_misc.h を参照して下さい。

2.51.1 構文

extern __value_in_regs **struct** __heap_extent __user_heap_extent(**uintptr_t** *ignore1*, **size_t** *ignore2*);

2.51.2 使用法

この関数のデフォルトの実装はありません。パラメータ *ignore1* および *ignore2* の値は、この関数では使用されません。

2.51.3 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *メモリ割り当て関数の C ライブラリサポート*（2-83 ページ）

2.52 __user_setup_stackheap()

__user_setup_stackheap() は、初期スタックおよびヒープの位置を設定し、返します。この関数を定義した場合、プログラムの起動時に C ライブラリによって呼び出されます。

__user_setup_stackheap() が呼び出されると、SP はアプリケーションへのエントリ時と同じ値になります。有効な値が設定されてから C ライブラリ初期化コードが呼び出されると、この値が保持されます。SP が有効な値でない場合は、スタックの使用前および呼び出し元に戻る前に、__user_setup_stackheap() によってこの値を変更する必要があります。

__user_setup_stackheap() の戻り値は以下のとおりです。

- プログラムでヒープを使用する場合は、ヒープベース。
  - AArch32 状態では、レジスタ R0 にはヒープベースが含まれています。
  - AArch64 状態では、レジスタ X0 にはヒープベースが含まれています。
- SP のスタックベース
- プログラムでヒープおよび 2 領域メモリを使用する場合はヒープリミット。
  - AArch32 状態では、レジスタ R2 にはヒープリミットが含まれています。
  - AArch64 状態では、レジスタ X2 にはヒープリミットが含まれています。

この関数を再実装する場合には、以下のガイドラインに従う必要があります。

- PCS によって要求される SP 以外のレジスタを保存すること。
- ヒープのアラインメントを守ること。
  - AArch32 状態では、ヒープベースが 8 バイトの倍数になるようにして、ヒープ内は常に 8 バイト境界の整列を守る。
  - AArch64 状態では、ヒープベースが 16 バイトの倍数になるようにして、ヒープ内は常に 16 バイト境界の整列を守る。

SP を実行環境から引き継ぎ、ヒープを持たない __user_setup_stackheap() を作成するには、以下のようにします。

- AArch32 状態では、R0 と R2 に ゼロ を設定して、復帰する。
- AArch64 状態では、X0 と X2 に ゼロ を設定して、復帰する。

スタックのサイズに制限はありません。ただし、ヒープ領域がスタックにオーバーラップする場合は、malloc() がオーバーラップしているメモリの検出を試み、新しいメモリ割り当て要求を失敗させます。

注

__user_setup_stackheap() はアセンブラで再実装する必要があります。

2.52.1 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *スタックポインタの初期化とヒープの上下限*（2-88 ページ）
- *__user_initial_stackheap() の従来のサポート*（2-92 ページ）

2.53 __vectab_stack_and_reset

__vectab_stack_and_reset は、sp および pc の初期値をベクタテーブルに配置するためのライブラリセクションです。

__vectab_stack_and_reset を使用するには、ソースコードに main() 関数が必要です。main() 関数がない状態で、スキャッタファイルに __vectab_stack_and_reset セクションを配置すると、以下のようなエラーが生成されます。

エラー : L6236E : セレクタと一致するセクションがありません。FIRST/LAST になるセクションがありません。

通常の起動コードをバイパスする場合（つまり、意図的に main() 関数を使用しない場合）は、__vectab_stack_and_reset を使用せずにベクタテーブルを設定する必要があります。

次のセグメントはスキャッタファイルの一部です。これには、__vectab_stack_and_reset を使用して、ベクタテーブルのアドレス 0x0 および 0x4 の位置に、sp および pc の初期値を配置する方法が示されています。

```
;; 最大 256 の例外（256*4 バイト == 0x400）
VECTORS 0x0 0x400
{
  ; 先頭の 2 エントリはライブラリが提供
  ; 残りのエントリはユーザが exceptions.c で定義

  * (:gdef:__vectab_stack_and_reset, +FIRST)
  * (exceptions_area)
}

CODE 0x400 FIXED
{
  * (+RO)
}
```

2.53.1 関連項目

概念

『リンカの使用』:

- *スキャッタロードの概要*（8-3 ページ）

2.54 wcscasecmp()

wchar.h で定義されるこの関数は、ワイドキャラクタに対して、大文字と小文字を区別しない文字列変換テストを実行します。これは、ライブラリの GNU 拡張機能です。POSIX では標準化されていません。

2.54.1 構文

int wcscasecmp(**const wchar_t** * __restrict *s1*, **const wchar_t** * __restrict *s2*);

2.55 wcsncasecmp()

wchar.h で定義されるこの関数は、指定された文字数を超えない、大文字と小文字を区別しない文字列変換テストを実行します。これは、ライブラリの GNU 拡張機能です。POSIX では標準化されていません。

2.55.1 構文

int wcsncasecmp(**const wchar_t** * __restrict *s1*, **const wchar_t** * __restrict *s2*, size_t *n*);

2.56 wcstombs()

wchar.h で定義されるこの関数には、POSIX によって指定された拡張機能があり、ISO C 標準の規定に従って動作します。つまり、*s* が NULL ポインタである場合、wcstombs() は *n* の値にかかわらず配列全体を変換するために必要な長さを返しますが、値は保存されません。

2.56.1 構文

size_t wcstombs(**char** *s, **const** wchar_t *pwcs, size_t *n*);

2.57 スレッドセーフな C ライブラリ関数

以下の表は、スレッドセーフな C ライブラリ関数の一覧です。

表 2-1 スレッドセーフな関数

関数	説明
calloc()、 free()、 malloc()、 realloc()	`_mutex_*` 関数が実装されている場合、ヒープ関数はスレッドセーフです。 すべてのスレッドで 1 つのヒープが共有されます。同時アクセスが発生する場合は、データの破損を防ぐためにミューテックスが使用されます。ヒープに関するロック処理は、個々のヒープ実装自体によって固有の方法で行われます。ユーザが独自にアロケータを指定する場合は、そのアロケータでも独自のロック処理を行う必要があります。これにより、単一のミューテックスを使用してヒープ全体を保護する（粗粒ロック）のではなく、必要に応じて細粒ロックを行うことができます。
alloca()	`alloca()` は、スタック上のメモリを割り当てるのでスレッドセーフです。
abort()、 raise()、 signal()、 fenv.h	ARM 信号処理関数と浮動小数点例外トラップはスレッドセーフです。 シグナルハンドラおよび浮動小数点トラップの設定はプロセス全体を通じてグローバルです。また、これらの設定は、ロックによって保護されています。複数のスレッドによって `signal()` 関数または `fenv.h` 関数が同時に呼び出されても、データは破損しません。ただし、関数呼び出しの影響は、呼び出し側スレッドだけではなく、すべてのスレッドに及ぶので注意して下さい。 **注** ARM コンパイラツールチェーンでは、AArch64 ターゲット用の浮動小数点例外トラップはサポートされません。

表 2-1 スレッドセーフな関数 （続き）

関数	説明
clearerr()、fclose()、 feof()、ferror()、fflush()、 fgetc()、fgetpos()、fgets()、 fopen()、fputc()、fputs()、 fread()、freopen()、fseek()、 fsetpos()、ftell()、fwrite()、 getc()、getchar()、gets()、 perror()、putc()、putchar()、 puts()、rewind()、setbuf()、 setvbuf()、tmpfile()、tmpnam()、 ungetc()	_mutex_* 関数が実装されている場合、stdio ライブラリはスレッドセーフです。 個々のストリームはロックにより保護されているので、2 つのスレッドは互いに割り込むことなく、それぞれが stdio ストリームを開いて利用できます。 2 つのスレッドによって同じストリームの読み出しまたは書き込みが実行される場合は、fgetc() および fputc() レベルでロックすることによりデータ破損を回避できます。ただし、各スレッドによって出力される文字がインターリーブされ、混乱する可能性があります。 **注** tmpnam() には、スタティックバッファも格納されますが、使用されるのは引数が NULL の場合だけです。使用する tmpnam() を確実にスレッドセーフにするには、独自のバッファ領域を指定します。
fprintf()、rintf()、 vfprintf()、vprintf()、fscanf()、 scanf()	これらの関数を使用する場合、以下のようになります。 • 標準的な C の printf() および scanf() 関数は stdio を使用するため、スレッドセーフです。 • 標準 C の printf() 関数は、マルチスレッドプログラムで呼び出された場合、ロケール設定の変更の影響を受けます。
clock()	clock() には、プログラム起動時に一度書き込まれると、その後は読み出し専用となるスタティックデータが格納されます。したがって、ライブラリ初期化時にまだ別の他のスレッドが実行されていなければ、clock() はスレッドセーフです。
errno	errno はスレッドセーフです。 各スレッドでは、__user_perthread_libspace ブロックに独自の errno が格納されます。そのため、各スレッドは、errno 設定関数を個別に呼び出してから、他のスレッドからの割り込みを受けることなく errno をチェックできます。
atexit()	atexit() に保持されている終了関数のリストは、プロセス内でグローバルです。また、ロックによって保護されています。 最悪の場合、複数のスレッドが atexit() を呼び出すと、呼び出される終了関数の順序は保証されません。

表 2-1 スレッドセーフな関数 （続き）

関数	説明
abs()、acos()、asin()、atan()、 atan2()、atof()、atol()、atoi()、 bsearch()、ceil()、cos()、 cosh()、difftime()、div()、 exp()、fabs()、floor()、 fmod()、frexp()、labs()、 ldexp()、ldiv()、log()、 log10()、memchr()、memcmp()、 memcpy()、memmove()、memset()、 mktime()、modf()、pow()、 qsort()、sin()、sinh()、 sqrt()、strcat()、strchr()、 strcmp()、strcpy()、strcspn()、 strlcat()、strlcpy()、strlen()、 strncat()、strncmp()、strncpy()、 strpbrk()、strrchr()、strspn()、 strstr()、strxfrm()、tan()、tanh()	これらの関数は本質的にスレッドセーフです。
longjmp()、setjmp()	`setjmp()` および `longjmp()` ではデータが `__user_libspace` に格納されますが、スレッドセーフな `__alloca_*` 関数が呼び出されます。
remove()、rename()、time()	これらの関数は、ARM デバッグ環境と通信する割り込みを使用します。通常、実際のアプリケーションではこれらの関数を再実装する必要があります。
snprintf()、sprintf()、 vsnprintf()、vsprintf()、sscanf()、 isalnum()、isalpha()、iscntrl()、 isdigit()、isgraph()、islower()、 isprint()、ispunct()、isspace()、 isupper()、isxdigit()、tolower()、 toupper()、strcoll()、strtod()、 strtol()、strtoul()、strftime()	これらの関数を使用した場合、文字列ベース関数はロケール設定を読み出します。通常、これらの関数はスレッドセーフです。ただし、セッションの途中でロケールを変更する場合は、これらの関数に影響が及ばないようにする必要があります。 `sprintf()` や `sscanf()` などの文字列ベース関数は、`stdio` ライブラリを使用しません。
stdin、stdout、stderr	これらの関数は、スレッドセーフです。

2.57.1 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *ARM C ライブラリでのスレッドセーフティ*（2-29 ページ）

2.58 スレッドセーフではない C ライブラリ関数

以下の表は、スレッドセーフではない C ライブラリ関数の一覧です。

表 2-2 スレッドセーフではない関数

関数	説明
asctime()、 localtime()、 strtok()	これらの関数は、いずれもスレッドセーフではありません。各関数では、スタティックバッファが使用されます。スタティックバッファは、関数の呼び出しとその戻り値の使用の間に別のスレッドによって上書きされる可能性があります。 ARM では、再入可能バージョンとして、_asctime_r()、_localtime_r()、および _strtok_r() が用意されています。安全性を確保するためにこれらの関数を使用することを推奨します。 **注** これらの再入可能関数は、追加のパラメータを取ります。_asctime_r() は、追加のパラメータとして、出力文字列の書き込み先バッファを指すポインタを取ります。_localtime_r() は、追加のパラメータとして、結果の書き込み先 struct tm を指すポインタを取ります。_strtok_r() は、追加のパラメータとして、次のトークンを指す **char** ポインタを指すポインタを取ります。
exit()	マルチスレッドプログラムでは、exit() を呼び出さないで下さい（すべてのスレッドを停止する基礎となる _sys_exit() の実装を指定していても呼び出さないで下さい）。 これを守らないと、_sys_exit() が呼び出される*前に* exit() がクリアされ、他のスレッドに対する割り込みが発生します。
gamma()、 lgamma()、 lgammaf()、 lgammal()	これらの拡張 mathlib 関数はグローバル変数 _signgam を使用するため、スレッドセーフではありません。 **注** gamma() は廃止される予定です。
mbrlen()、 mbsrtowcs()、 mbrtowc()、 wcrtomb()、 wcsrtombs()	C89 マルチバイト変換関数（stdlib.h に定義）は、すべてのスレッドにロックなしで共有される内部静的状態を保持しているため、スレッドセーフではありません。このような関数として、mblen() および mbtowc() があります。 ただし、拡張再起動可能バージョン（wchar.h に定義）である mbrtowc() および wcrtomb() は、独自に作成した mbstate_t オブジェクトを指すポインタを渡す場合にはスレッドセーフです。マルチバイト文字列を処理する場合にスレッドセーフティを確保するには、これらの関数を非 NULL mbstate_t * パラメータと一緒に排他的に使用する必要があります。

表 2-2 スレッドセーフではない関数 （続き）

関数	説明
rand()、srand()	これらの関数は、保護されていないグローバルな内部状態を維持します。つまり、rand() への呼び出しはスレッドセーフではありません。 ARM では、以下のいずれかの方法を推奨します。 • ARM が提供する再入可能バージョンの _rand_r() および _srand_r() を使用する。これらの関数では、C ライブラリ内のスタティックデータではなく、ユーザ指定のバッファが使用されます。 • 一度に 1 つのスレッドのみが rand() を呼び出すように独自のロックを使用する。例えば、コードを変更せずにロックを適用するには、$Sub$$rand() を定義します。 • 乱数を生成する必要があるスレッドが 1 つのみになるように調整する。 • 複数の独立したインスタンスを持つことができる、独自の乱数ジェネレータを用意する。 **注** _rand_r() および _srand_r() はいずれも、追加のパラメータとして、乱数ジェネレータの状態が格納されるバッファを指すポインタを取ります。

表 2-2 スレッドセーフではない関数 （続き）

関数	説明
setlocale()、localeconv()	setlocale() は、ロケールの設定および読み出しに使用されます。ロケール設定はすべてのスレッドを通じてグローバルであり、ロックによって保護されていません。ロケール設定を変更するために 2 つのスレッドによって setlocale() が呼び出された場合、または、1 つのスレッドが設定を読み出し、別のスレッドが設定を変更した場合、データが破損する可能性があります。また、strtod() や sprintf() など、その他の多くの関数も、現在のロケール設定を読み出します。したがって、1 つのスレッドが他のスレッドと同時に setlocale() を呼び出すと、予期しない結果になる場合があります。 複数のスレッドによる設定の同時*読み出し*は、単純な処理の場合、および、これらの設定を同時に変更するスレッドが存在しない場合はスレッドセーフです。ただし、返される結果が複雑で内部的に中間バッファが必要な場合は、再入可能バージョンの setlocale() を使用しない限り、予期しない結果になる可能性があります。 ARM では以下のいずれかの方法を推奨します。 • 使用するロケールを 1 つに決定し、setlocale() を一度だけ呼び出して初期化する。これは、プログラムでスレッドを追加する前に行って下さい。これにより、複数のスレッドが相互に干渉することなく同じロケール設定を同時に読み出すことができます。 • ARM が提供する再入可能バージョンの _setlocale_r() を使用する。この関数は、C ライブラリ内のメモリを使用する代わりに、定数文字列を指すポインタ、または、スレッドローカルストレージとして使用できるユーザ定義のバッファに格納される文字列を指すポインタを返します。バッファの長さは、少なくとも _SETLOCALE_R_BUFSIZE バイトである必要があります（末尾の NUL のスペースを含む）。 **注** • _setlocale_r() は、ロケール設定の変更のために同時にアクセスされる場合があり、完全なスレッドセーフではないことに注意して下さい。このようなアクセスが行われた場合、ロックでは保護されません。 • localeconv() はスレッドセーフではありません。この関数の代わりに、ユーザ指定バッファを指すポインタを使用して、ARM 関数 _get_lconv() を呼び出して下さい。

2.58.1 関連項目

概念

- *ARM C ライブラリでのスレッドセーフティ*（2-29 ページ）

参照

- *_rand_r()*（2-25 ページ）
- *_srand_r()*（2-38 ページ）

第 3 章
浮動小数点のサポート

以下の各トピックでは、ARM における浮動小数点計算のサポートについて説明します。

3.1 _clearfp()

float.h で定義されるこの関数は、Microsoft 製品との互換性を維持するために提供されています。

_clearfp() 関数は、5 つの例外のスティッキーフラグのすべてをクリアし、それらの前の値を返します。_controlfp() の引数に使用されるマクロ（例えば、_EM_INVALID および _EM_ZERODIVIDE）を使用して、返された結果のビットをテストできます。

_clearfp() の関数プロトタイプは次のとおりです。

```
unsigned _clearfp(void);
```

注

この関数には、例外をサポートする浮動小数点モデルが必要です。ARM コンパイラ 6 では、これはデフォルトで有効になっています。-ffast-math などの armclang コマンドラインオプションによって無効になります。

3.1.1 関連項目

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *Microsoft 製品との互換性を維持するための浮動小数点関数*（4-5 ページ）

参照

- *_controlfp()*（3-3 ページ）
- *_statusfp()*（3-14 ページ）

3.2 _controlfp()

注

ARM コンパイラツールチェーンでは、AArch64 ターゲット用の浮動小数点例外トラップはサポートされません。

float.h で定義されるこの関数は、Microsoft 製品との互換性を維持するために提供されています。この関数を使用して、例外のトラップと丸めモードを制御できます。

_controlfp() の関数プロトタイプは次のとおりです。

unsigned int _controlfp(**unsigned int** new, **unsigned int** mask);

注

この関数には、例外をサポートする浮動小数点モデルが必要です。ARM コンパイラ 6 では、これはデフォルトで有効になっています。-ffast-math などの armclang コマンドラインオプションによって無効になります。

また、_controlfp() は、マスクを使用して制御ワードを修正し、修正するビットを切り離すことができます。mask のビットにゼロが設定されている場合、それに対応する制御ワードのビットは変更されません。mask のビットにゼロ以外の値が設定されている場合は、それに対応する制御ワードのビットに、new の対応するビットの値が設定されます。戻り値は、制御ワードの前の状態となります。

注

これは、マスクのワードにゼロを設定し、フラグのワードに 1 を設定することでビットを切り替えられる、__ieee_status() または __fp_status() の動作とは異なります。

表 3-1 は、_controlfp() への引数を作成する場合に使用できるマクロを示しています。

表 3-1 _controlfp の引数に使用されるマクロ

マクロ	説明
_MCW_EM	すべての例外ビットを保持するマスクが生成されます。
_EM_INVALID	無効演算例外を示すビットが生成されます。
_EM_ZERODIVIDE	ゼロによる除算例外を示すビットが生成されます。
_EM_OVERFLOW	オーバーフロー例外を示すビットが生成されます。
_EM_UNDERFLOW	アンダーフロー例外を示すビットが生成されます。
_EM_INEXACT	不正確結果例外を示すビットが生成されます。
_MCW_RC	丸めモードフィールドのマスクが生成されます。
_RC_CHOP	ゼロへの丸めを示す丸めモードの値が生成されます。

表 3-1 _controlfp の引数に使用されるマクロ （続き）

マクロ	説明
_RC_UP	切り上げを示す丸めモードの値が生成されます。
_RC_DOWN	切り下げを示す丸めモードの値が生成されます。
_RC_NEAR	近似値への丸めを示す丸めモードの値が生成されます。

注

これらのマクロの値は、将来の ARM 製品のバージョンにおいても同じであるという保証はありません。将来のリリースで値が変更されてもコードが機能するように、値ではなくマクロを使用して下さい。

例えば、丸めモードを切り下げに設定するには、以下の関数を呼び出します。

```
_controlfp(_RC_DOWN, _MCW_RC);
```

無効演算例外のトラップを有効にし、他のすべての例外のトラップを無効にするには、以下のように記述します。

```
_controlfp(_EM_INVALID, _MCW_EM);
```

不正確結果例外のトラップを無効にするには、以下のように記述します。

```
_controlfp(0, _EM_INEXACT);
```

3.2.1 関連項目

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *Microsoft 製品との互換性を維持するための浮動小数点関数*（4-5 ページ）

参照

- *__ieee_status()*（3-8 ページ）
- *__fp_status()*（3-5 ページ）
- *_clearfp()*（3-2 ページ）
- *_statusfp()*（3-14 ページ）

3.3 __fp_status()

注

ARM コンパイラツールチェーンでは、AArch64 ターゲット用の浮動小数点例外トラップはサポートされません。

旧バージョンの ARM ライブラリの中には、浮動小数点環境でステータスワードを操作する __fp_status() という名前の関数が実装されていたものがあります。この関数は、__ieee_status() と同じですが、旧式のステータスワードのレイアウトを使用します。

__fp_status() の関数プロトタイプは次のとおりです。

unsigned int __fp_status(**unsigned int** mask, **unsigned int** flags);

注

この関数には、例外をサポートする浮動小数点モデルが必要です。ARM コンパイラ 6 では、これはデフォルトで有効になっています。-ffast-math などの armclang コマンドラインオプションによって無効になります。

図 3-1 は、__fp_status() を使用したときのステータスワードのレイアウトを示しています。

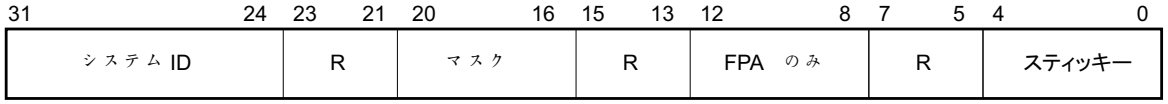

図 3-1 浮動小数点ステータスワードのレイアウト

図 3-1 は以下のフィールドを示しています。

- ビット 0 ～ 4（それぞれ 0x1 ～ 0x10 の値）は、各例外のスティッキーフラグまたは累積フラグです。例外のスティッキーフラグには、その例外が発生し、トラップされなかった場合に 1 が設定されます。スティッキーフラグはシステムによってクリアされることはなく、ユーザによってのみクリアされます。例外はこれらのビットに以下のようにマップされます。
  - ビット 0（0x01）は無効演算例外です。
  - ビット 1（0x02）はゼロによる除算例外です。
  - ビット 2（0x04）はオーバーフロー例外です。
  - ビット 3（0x08）はアンダーフロー例外です。
  - ビット 4（0x10）は不正確結果例外です。
- ビット 8 ～ 12（0x100 ～ 0x1000 の値）は、*浮動小数点アーキテクチャ*（FPA）のさまざまな機能を制御します。FPA は廃止され、ARM Compilation Tools ではサポートされていません。これらのビットへの書き込みはすべて無視されます。
- ビット 16 ～ 20（0x10000 ～ 0x100000 の値）は例外マスクです。これらのビットは例外がトラップされるかどうかを制御します。これらのビットのいずれかに 1 が設定されると、対応する例外がトラップされます。これらのビットのいずれかに 0 が設定されると、対応する例外によってそのスティッキーフラグが設定され、正しそうな結果が返されます。

- ビット 24 〜 31 に保持されるシステム ID は変更できません。ソフトウェア浮動小数点の場合は 0x40 が設定され、ハードウェア浮動小数点の場合は 0x80 以上の値が設定され、ハードウェア浮動小数点環境がエミュレータによってシミュレートされる場合は 0 または 1 が設定されます。
- R でマークされているビットは予約ビットです。__fp_status() 呼び出しによってこれらのビットに書き込むことはできません。これらのビットに何らかの値が保持されていても無視する必要があります。

丸めモードを __fp_status() 呼び出しで変更することはできません。

__fp_status() 呼び出しの定義に加え、stdlib.h ではその引数に使用される次の定数も定義されています。

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

例えば、無効演算例外をトラップして、他のすべての例外のトラップを無効にするには、次の入力パラメータを使用して __fp_status() を呼び出します。

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
        _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

不正確結果例外のトラップを無効にするには、以下のように記述します。

```
__fp_status(_fpsr_IXE, 0);
```

アンダーフロースティッキーフラグをクリアするには、以下のように記述します。

```
__fp_status(_fpsr_UFC, 0);
```

3.3.1 関連項目

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *ARM 浮動小数点環境の制御*（4-4 ページ）

参照

- *__ieee_status()*（3-8 ページ）

3.4 gamma()、gamma_r()

これらの関数はいずれもガンマ関数の対数を計算します。これらは、lgamma および lgamma_r と同じです。

double gamma(**double** x); **double** gamma_r(**double** x, **int** *);

注

これらの関数はガンマ関数と呼ばれていますが、実際はガンマ関数そのものではなく、ガンマ関数の対数を計算するものです。

注

これらの関数は廃止される予定です。

3.4.1 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *mathlib での標準以外の関数*（4-26 ページ）

3.5 __ieee_status()

注

ARM コンパイラツールチェーンでは、AArch64 ターゲット用の浮動小数点例外トラップはサポートされません。

ARM コンパイラツールチェインでは、浮動小数点環境のステータスワードへのインタフェースがサポートされています。このインタフェースは、関数 __ieee_status() によって提供されます。一般的に VFP のステータスワードを変更するにはこの関数を使用するのが最も効率的です。__ieee_status() は fenv.h で定義されています。

__ieee_status() の関数プロトタイプは次のとおりです。

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```

注

この関数には、例外をサポートする浮動小数点モデルが必要です。ARM コンパイラ 6 では、これはデフォルトで有効になっています。-ffast-math などの armclang コマンドラインオプションによって無効になります。

__ieee_status() は、パラメータに基づいてステータスワードの書き込み可能部分を修正し、そのワード全体の前の値を返します。

書き込み可能ビットを変更するには、それらの値を以下のように設定します。

```
new = (old & ~mask) ^ flags;
```

対応するマスクとフラグのビットに基づき、ステータスワードの各ビットに対しては 4 つの異なる演算を実行できます。詳細については、「表 3-2」を参照して下さい。

表 3-2 ステータスワードビットの修正

マスクのビット	フラグのビット	効果
0	0	変更なし
0	1	切り替え
1	0	0 を設定
1	1	1 を設定

図 3-2 は、__ieee_status() を使用したときのステータスワードのレイアウトを示しています。

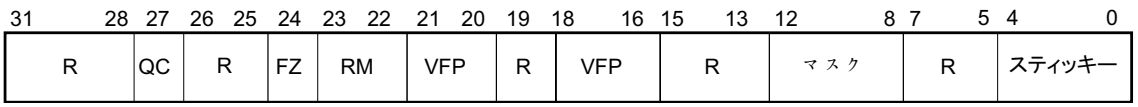

図 3-2 IEEE のステータスワードのレイアウト

図 3-2 （3-8 ページ）は以下のフィールドを示しています。

- ビット 0 〜 4（それぞれ 0x1 〜 0x10 の値）は、各例外のスティッキーフラグまたは累積フラグです。例外のスティッキーフラグには、その例外が発生し、トラップされなかった場合に 1 が設定されます。スティッキーフラグはシステムによってクリアされることはなく、ユーザによってのみクリアされます。例外はこれらのビットに以下のようにマップされます。
  - ビット 0（0x01）は無効演算例外です。
  - ビット 1（0x02）はゼロによる除算例外です。
  - ビット 2（0x04）はオーバーフロー例外です。
  - ビット 3（0x08）はアンダーフロー例外です。
  - ビット 4（0x10）は不正確結果例外です。
- ビット 8 〜 12（0x100 〜 0x1000 の値）は例外マスクです。これらのビットは例外がトラップされるかどうかを制御します。これらのビットのいずれかに 1 が設定されると、対応する例外がトラップされます。これらのビットのいずれかに 0 が設定されると、対応する例外によってそのスティッキーフラグが設定され、正しそうな結果が返されます。
- ビット 16 〜 18 とビット 20 〜 21 は、VFP ベクタ機能を制御するために VFP ハードウェアによって使用されます。__ieee_status() の呼び出しによってこれらのビットを変更することはできません。
- ビット 22 とビット 23 は、丸めモードを制御します。詳細については、「表 3-3」を参照して下さい。

表 3-3 丸めモードの制御

ビット	丸めモード
00	近似値への丸め
01	切り上げ
10	切り下げ
11	ゼロへの丸め

注

ARM コンパイラ 6 では、デフォルトで関連ライブラリが選択されます。armclang コマンドラインオプション -ffast-math は、他のバリアントを選択します。

- ビット 24 を設定すると、FZ（ゼロクリア）モードが有効にされます。非正規数の処理は複雑で浮動小数点システムの処理速度が低下するため、FZ モードでは処理速度を上げるために非正規数はすべてゼロに強制設定されます。このビットを設定することで精度は落ちますが、処理速度は上がります。

注

— IEEE ステータスワードの FZ ビットは、fplib バリアントのいずれでもサポートされていません。つまり、ゼロクリアとゼロクリア以外との間の切り替えを、*実行*時に fplib のバリアントのいずれかで実行することはできません。ただし、ゼロクリアまたはゼロクリア以外の設定は、ビルド用に選択するライブラリにより、結果的にコンパイル時に設定できます。

— 一部の関数は、ハードウェアで提供されていません。これらは、ソフトウェア浮動小数点ライブラリにのみ存在します。このため、これらの関数では、ハードウェア VFP アーキテクチャ用にコンパイルする場合であっても、FZ モードがサポートされません。したがって、FZ モードを動的に変更した場合、浮動小数点ライブラリの動作が関数によって異なります。

- ビット 27 は、整数のアドバンスト SIMD サチュレート演算でサチュレーションが発生したことを示します。このビットには、__ieee_status() 呼び出しによってアクセスできます。
- R でマークされているビットは予約ビットです。__ieee_status() 呼び出しによってこれらのビットに書き込むことはできません。これらのビットに何らかの値が保持されていても無視する必要があります。

__ieee_status() 呼び出しの定義に加え、fenv.h ではその引数に使用される次の定数も定義されています。

```
#define FE_IEEE_FLUSHZERO        (0x01000000)
#define FE_IEEE_ROUND_TONEAREST  (0x00000000)
#define FE_IEEE_ROUND_UPWARD     (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD   (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO (0x00C00000)
#define FE_IEEE_ROUND_MASK       (0x00C00000)
#define FE_IEEE_MASK_INVALID     (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO   (0x00000200)
#define FE_IEEE_MASK_OVERFLOW    (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW   (0x00000800)
#define FE_IEEE_MASK_INEXACT     (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT  (0x00001F00)
#define FE_IEEE_INVALID          (0x00000001)
#define FE_IEEE_DIVBYZERO        (0x00000002)
#define FE_IEEE_OVERFLOW         (0x00000004)
#define FE_IEEE_UNDERFLOW        (0x00000008)
#define FE_IEEE_INEXACT          (0x00000010)
#define FE_IEEE_ALL_EXCEPT       (0x0000001F)
```

例えば、丸めモードを切り下げに設定するには、以下の呼び出しを行います。

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

無効演算例外のトラップを有効にし、他のすべての例外のトラップを無効にするには、以下のように記述します。

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

不正確結果例外のトラップを無効にするには、以下のように記述します。

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

アンダーフロースティッキーフラグをクリアするには、以下のように記述します。

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

3.5.1 関連項目

概念

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *IEEE 754 浮動小数点演算で発生する例外*（4-37 ページ）
- *C および C++ ライブラリの命名規則*（2-123 ページ）

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド:

- *ARM 浮動小数点環境の制御*（4-4 ページ）

参照

- `__fp_status()`（3-5 ページ）

3.6 j0()、j1()、jn()（第 1 種のベッセル関数）

これらの関数は、第 1 種の Bessel 関数を計算します。j0 と j1 は、それぞれ 0 次と 1 次の関数を計算します。jn は *n* 次の関数を計算します。

double j0(**double** x); **double** j1(**double** x); **double** jn(**int** n, **double** x);

x の絶対値が $\pi \times 2^{52}$ を超える場合、これらの関数は ERANGE エラーを返し、結果が無効であることを示します。

注

これらの関数は廃止される予定です。

3.6.1 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *mathlib での標準以外の関数*（4-26 ページ）

3.7 significand()（数値の小数部）

この関数は、*x* の小数部を 1.0 ～ 2.0（2.0 は含まれない）の数値として返します。

double significand(**double** x);

注

この関数は廃止される予定です。

3.7.1 関連項目

参照

『ARM C および C++ ライブラリと浮動小数点サポートの使用』:

- *mathlib での標準以外の関数*（4-26 ページ）

3.8 _statusfp()

float.h で定義されるこの関数は、Microsoft 製品との互換性を維持するために提供されています。この関数は、例外のスティッキーフラグの現在の値を返します。_controlfp() の引数に使用されるマクロ（例えば、_EM_INVALID および _EM_ZERODIVIDE）を使用して、返された結果のビットをテストできます。

_statusfp() の関数プロトタイプは次のとおりです。

```
unsigned _statusfp(void);
```

注

この関数には、例外をサポートする浮動小数点モデルが必要です。ARM コンパイラ 6 では、これはデフォルトで有効になっています。-ffast-math などの armclang コマンドラインオプションによって無効になります。

3.8.1 関連項目

タスク

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *Microsoft 製品との互換性を維持するための浮動小数点関数*（4-5 ページ）

参照

- *_controlfp()*（3-3 ページ）
- *_clearfp()*（3-2 ページ）

3.9 y0()、y1()、yn()（第 2 種のベッセル関数）

これらの関数は、第 2 種のベッセル関数を計算します。y0 と y1 は、それぞれ 0 次と 1 次の関数を計算します。yn は *n* 次の関数を計算します。

double y0(**double** x); **double** y1(**double** x); **double** yn(**int**, **double**);

x が正の値であり、$\pi \times 2^{52}$ を超える場合、これらの関数は ERANGE エラーを返し、結果が無効であることを示します。

注

これらの関数は廃止される予定です。

3.9.1 関連項目

参照

ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド：

- *mathlib での標準以外の関数*（4-26 ページ）