

ADJ-702-214

日立マイクロコンピュータ開発環境システム
SH シリーズ C++コンパイラ
ユーザーズマニュアル

HS0700CPCS1S

発行年月日	平成 8 年 9 月 第 1 版
発行	株式会社 日立製作所 電子統括営業本部
編集	株式会社日立マイコンシステム 技術情報センタ
©株式会社 日立製作所 1996	

はじめに

本マニュアルは、SH シリーズ C++ コンパイラの使用方法を述べたものです。

SH シリーズ C++ コンパイラをご使用になる前に本マニュアルを良く読んで理解してください。また、後述の関連マニュアルもお読みの上、理解してください。

本 C++ コンパイラは、C++ 言語で記述したソースプログラムを日立 SuperH RISC engine ファミリ (SH1、SH2、SH3) のリロケータブルオブジェクトプログラムまたはアセンブリソースプログラムに変換するソフトウェアシステムです。

本コンパイラシステムの特長

- (1) 機器組み込み用として ROM 化可能なオブジェクトプログラムを生成します。
- (2) オブジェクトプログラムの実行速度向上やサイズ縮小のための最適化オプションをサポートしています。
- (3) C++ ソースレベルデバッグ、ブラウザによる C++ ソース解析を行うためのデバッグ情報出力、ブラウザ情報出力オプションをサポートしています。
- (4) アセンブリソースプログラムまたはリロケータブルオブジェクトプログラムを選択して出力することができます。

本マニュアルは本文 4 章と付録で構成されています。各章の内容を以下に示します。

第 1 章 概要・操作

概要では、コンパイラの機能、プログラムの開発手順について説明します。操作では、コンパイラの起動方法、オプション機能、コンパイルリストの見方について説明します。

第 2 章 C++ プログラミング

C++ プログラミングでは、コンパイラの限界値、オブジェクトプログラムの実行方式などプログラム開発時に考慮すべき事項について説明します。

第 3 章 システム組み込み

システム組み込みでは、本 C++ コンパイラの生成したオブジェクトプログラムをシステムに組み込むために必要なメモリ割り付け方法、ROM 化の方法について説明します。また、C 言語の標準入出力ライブラリ、メモリ管理ライブラリを使用する場合にユーザが作成しなければならない低水準インタフェースルーチンの仕様について説明します。

第4章 エラーメッセージ

コンパイル時に発生するエラーメッセージとエラー内容、実行時に発生する C ライブラリ関数のエラーメッセージとエラー内容を説明します。

本マニュアルは UNIX^{*1} または、PC-9801^{*2} シリーズ、IBM PC^{*3} およびその互換機上で動作する MS-DOS^{*4} に対応するように書かれています。UNIX 上で動作するコンパイラを、以下 UNIX 版と称します。MS-DOS 上で動作するコンパイラを、以下 PC 版と称します。

表記上の注意事項

本マニュアルでコマンド等の指定方法の説明で用いる記号を以下に示します。

< > この記号で囲まれた内容を指定することを示します。

[] 省略してもよい項目を示します。

. . . 直前の項目を 1 回以上指定することを示します。

1 個以上の空白を示します。

(RET) キャリッジリターンキー（リターンキーともいいます）を示します。

| | で区切られた項目を選択できることを示します。

(CNTL) 次の文字を、コントロールキーを押しながら入力することを示します。

【注】 *1 UNIX は、X/Open カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。

*2 PC-9801 は、日本電気株式会社の商標です。

*3 IBM PC は、米国 International Business Machines Corporation の登録商標です。

*4 MS-DOS は、米国マイクロソフト社の登録商標です。

関連マニュアル

本 C++ コンパイラを用いてプログラムを開発するために必要な関連マニュアルを以下に示します。本マニュアルと合わせて参照してください。

- 「SH シリーズクロスアセンブラユーザーズマニュアル」
- 「SH シリーズシミュレータデバッガユーザーズマニュアル」
- 「統合化マネージャユーザーズマニュアル」
- 「SHC++ 対応ブラウザユーザーズマニュアル」
- 「H シリーズリンクージエディタユーザーズマニュアル」
- 「H シリーズライブラリアンユーザーズマニュアル」
- 「E7000 SH7032、SH7034 エミュレータユーザーズマニュアル」
- 「E7000 SH7604 エミュレータユーザーズマニュアル」
- 「E7000 SH7708、SH7702 エミュレータユーザーズマニュアル」

また、SH の実行命令の詳細については、

- 「SH-1 / SH-2 プログラミングマニュアル」
- 「SH7700 シリーズプログラミングマニュアル」

を参照してください。

目次

第1章 概要・操作

1.1	概要.....	3
1.2	プログラムの開発手順.....	4
1.3	C++コンパイラの実行.....	5
1.3.1	C++コンパイラの起動方法.....	5
1.3.2	ファイル名の付け方.....	7
1.3.3	コンパイラオプション.....	8
1.3.4	オプションの組み合わせ.....	15
1.3.5	標準ライブラリとの対応.....	16
1.3.6	コンパイルリストの見方.....	17
1.3.7	コンパイラ環境変数.....	21

第2章 C++プログラミング

2.1	C++コンパイラの限界値.....	25
2.2	C++プログラムの実行方式.....	27
2.2.1	オブジェクトプログラムの構造.....	27
2.2.2	データの内部表現.....	29
2.2.3	Cプログラムとの結合.....	38
2.2.4	アセンブリプログラムとの結合.....	39
2.3	拡張機能.....	48
2.3.1	割り込み関数の使用方法.....	48
2.3.2	組み込み関数の使用方法.....	51
2.3.3	セクション切り替え機能.....	56
2.3.4	単精度浮動小数点ライブラリ.....	57
2.3.5	文字列内の日本語記述.....	59
2.3.6	関数のインライン展開.....	59
2.3.7	アセンブラ埋め込みインライン展開.....	60

2.3.8	2 バイトアドレス変数の指定	62
2.3.9	GBR ベース変数の指定	62
2.3.10	レジスタ退避・回復の制御	63
2.4	プログラム作成上の注意事項	65
2.4.1	コーディング上の注意事項	65
2.4.2	プログラム開発上のトラブル対処方法	67

第3章 システム組み込み

3.1	システム組み込みの概要	71
3.2	メモリ領域の割り付け	72
3.2.1	静的領域の割り付け	72
3.2.2	動的領域の割り付け	76
3.3	実行環境の設定	80
3.4	C ライブラリ関数の実行環境の設定	86

第4章 エラーメッセージ

4.1	C++ コンパイラのエラーメッセージ	105
4.1.1	エラーメッセージ一覧	105
4.2	標準ライブラリのエラーメッセージ	141

《付録》

A.	C++ コンパイラが規定する言語仕様と C ライブラリ関数仕様	147
A.1	言語仕様	147
A.2	C ライブラリ関数仕様	155
A.3	浮動小数点数の仕様	160
B.	引数割り付けの具体例	168
C.	レジスタとスタック領域の使用法	170
D.	終了処理関数の作成例	171
D.1	終了処理の登録と実行 (onexit) ルーチンの作成例	171
D.2	プログラムの終了 (exit) ルーチンの作成例	172
D.3	異常終了 (abort) ルーチンの作成例	174
E.	低水準インタフェースルーチンの作成例	175
F.	ASCII コード表	182
G.	C プログラムを C++ コンパイラでコンパイルするときの注意事項	183
H.	索引	184

H.1	日本語索引.....	184
H.2	英語索引.....	189

図目次

< 概要・操作 >

図 1.1	C++ コンパイラの機能.....	3
図 1.2	プログラムの開発手順.....	4
図 1.3	show = noinclude, noexpansion のソースリスト情報.....	18
図 1.4	show = include, expansion のソースリスト情報.....	18
図 1.5	オブジェクト情報.....	19
図 1.6	統計情報.....	20
図 1.7	コマンド指定情報.....	21

< C++ プログラミング >

図 2.1	スタックフレームの割り付け、解放に関する規則.....	41
図 2.2	引数の割り付け領域.....	45
図 2.3	引数格納用レジスタの割り付け例.....	46
図 2.4	リターン値をメモリに設定する場合のリターン値の設定領域.....	47
図 2.5	割り込み関数によるスタック使用例.....	49

< システム組み込み >

図 3.1	統計情報例.....	72
図 3.2	静的な領域の割り付け例.....	75
図 3.3	関数呼び出しの関係とスタック使用量の例.....	78
図 3.4	プログラムの構成例 (C ライブラリ関数を使用しない場合).....	80
図 3.5	プログラムの構成例 (C ライブラリ関数を使用する場合).....	86
図 3.6	FILE 型データ.....	91

< 付録 >

図 A.1	浮動小数点数の内部表現の構成.....	160
図 C.1	レジスタとスタック領域の使用法.....	170

表目次

< 概要・操作 >

表 1.1	C++ コンパイラで使用する標準のファイル拡張子.....	7
表 1.2	コンパイラオプション一覧	8
表 1.3	define オプションで指定できるマクロ名、名前、定数	11
表 1.4	オプションの組み合わせ.....	15
表 1.5	標準ライブラリとコンパイルオプションの関係	16
表 1.6	コンパイルリストの構成と内容.....	17
表 1.7	環境変数	21

< C++ プログラミング >

表 2.1	C++ コンパイラの限界値	25
表 2.2	メモリ領域の種類とその性質の概要	28
表 2.3	組み込み型の内部表現	30
表 2.4	利用者定義型の内部表現.....	31
表 2.5	ビットフィールドメンバの仕様.....	34
表 2.6	関数呼び出し前後のレジスタ保証規則.....	41
表 2.7	引数割り付け領域の一般規則	45
表 2.8	リターン値の型と設定場所	47
表 2.9	割り込み仕様一覧.....	48
表 2.10	組み込み関数一覧.....	51
表 2.11	単精度浮動小数点ライブラリ関数一覧.....	58
表 2.12	日本語コードのデフォルト設定.....	59
表 2.13	トラブル発生時の対処方法	67

< システム組み込み >

表 3.1	スタックサイズの計算例.....	78
表 3.2	低水準インタフェースルーチンの一覧.....	92

< 付録 >

表 A.1	翻訳の仕様.....	147
表 A.2	環境の仕様.....	147
表 A.3	識別子の仕様	147
表 A.4	文字の仕様.....	148
表 A.5	整数の仕様.....	149
表 A.6	整数型とその値の範囲	149
表 A.7	浮動小数点の仕様.....	150
表 A.8	浮動小数点数の限界値	150
表 A.9	配列とポインタの仕様	151

表 A.10	レジスタの仕様.....	151
表 A.11	クラス、列挙型、ビットフィールドの仕様.....	152
表 A.12	修飾子の仕様.....	152
表 A.13	宣言の仕様.....	153
表 A.14	文の仕様.....	153
表 A.15	プリプロセッサの仕様.....	154
表 A.16	stddef.h の仕様.....	155
表 A.17	assert.h の仕様.....	155
表 A.18	ctype.h の仕様.....	155
表 A.19	真となる文字の集合.....	155
表 A.20	math.h の仕様.....	156
表 A.21	setjmp.h の仕様.....	156
表 A.22	stdio.h の仕様.....	157
表 A.23	無限大および非数の表示形式.....	158
表 A.24	string.h の仕様.....	158
表 A.25	error.h の仕様.....	159
表 A.26	浮動小数点数を表現する値の種類.....	161
表 F.1	ASCII コード一覧表.....	182

1. 概要・操作

第1章 目次

1.1	概要.....	3
1.2	プログラムの開発手順.....	4
1.3	C++コンパイラの実行.....	5
1.3.1	C++コンパイラの起動方法.....	5
1.3.2	ファイル名の付け方.....	7
1.3.3	コンパイラオプション.....	8
1.3.4	オプションの組み合わせ.....	15
1.3.5	標準ライブラリとの対応.....	16
1.3.6	コンパイルリストの見方.....	17
1.3.7	コンパイラ環境変数.....	21

1.1 概要

SH シリーズ C++ コンパイラは、C++ 言語で記述したソースプログラムを、SH シリーズ用リロケータブルオブジェクトプログラムまたはアセンブリソースプログラムに変換するソフトウェアシステムです。本 C++ コンパイラがサポートするマイコンは SH1、SH2 および SH3 です（以下、SH と略します）。

C++ コンパイラの機能を図 1.1 に示します。

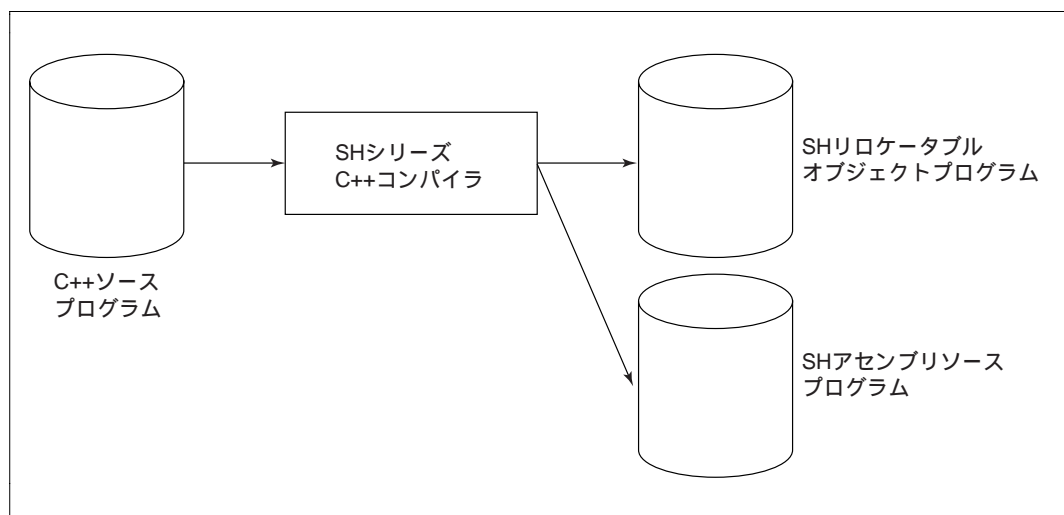


図 1.1 C++ コンパイラの機能

また C++ コンパイラのほかに、標準ライブラリ（C++ 言語で記述されているプログラム内で標準的に利用する C 言語レベル関数群）を提供します。

本システムを用いたプログラムの開発手順を図 1.2 に示します。

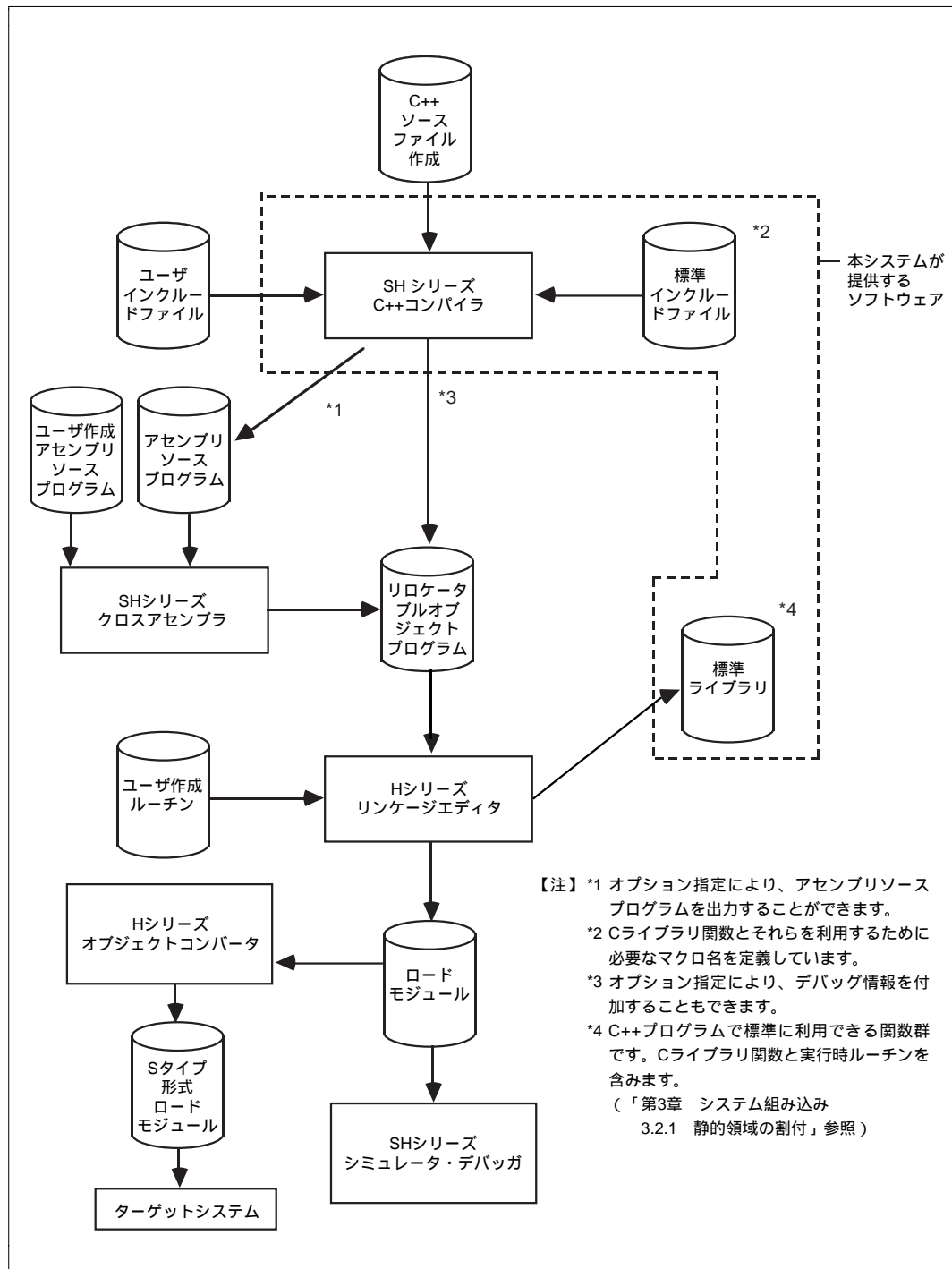


図 1.2 プログラムの開発手順

1.3 C++コンパイラの実行

本節では C++ コンパイラの起動方法、オプションの指定方法、コンパイルリストの見方について解説します。

1.3.1 C++コンパイラの起動方法

C++コンパイラを起動するコマンドラインの形式は次のとおりです。

```
shcpp [ <オプション>... ] [ <ファイル名> [ <オプション>... ] ... ]
```

以下 C++ コンパイラの基本的な操作方法を説明します。ここでは、C++ソースファイル名の拡張子に.cpp を使用しますが、一般的には、C++ソースファイルの拡張子に.cpp の他、.cc や.C あるいは.c が使用されます。

(1) プログラムのコンパイル

C++ソースプログラム「test.cpp」をコンパイルします。

```
shcpp test.cpp (RET)
```

(2) コマンド入力形式、コンパイラオプションの表示

標準出力画面上にコマンドの入力形式、コンパイラオプションの一覧を表示します。

```
shcpp (RET)
```

(3) オプション指定方法

オプション (debug、listfile、show) の前に UNIX 版では -、PC 版では / を付加し、複数のオプションを指定するときはスペース () で区切ります。以下に UNIX 版と PC 版オプション指定方法を示します。

複数のサブオプションを指定するときはコンマ (,) で区切って指定します。

```
shcpp -debug -listfile -show=noobject,expansion test.cpp (RET)
```

PC 版では、さらに括弧 () でくくって指定することもできます。

```
shcpp /debug /listfile /show=(noobject,expansion) test.cpp (RET)
```

(4) 複数 C++ソースプログラムのコンパイル

UNIX 版では複数の C++ソースプログラムを一度にコンパイルできます。

例 1 複数プログラムの指定方法

```
shcpp test1.cpp test2.cpp (RET)
```

例 2 オプションの指定 (C++ソースプログラムすべてに有効なオプション指定例)

```
shcpp -listfile test1.cpp test2.cpp (RET)
```

「test1.cpp」, 「test2.cpp」とも listfile オプションが有効となります。

例 3 オプションの指定 (各ソースプログラムごとに有効なオプション指定例)

```
shcpp test1.cpp test2.cpp -listfile (RET)
```

listfile オプションは「test2.cpp」だけに対して有効になります。C++ソースプログラムごとのオプション指定は、C++ソースプログラム全体に対するオプション指定よりも優先されます。

1.3.2 ファイル名の付け方

ファイル名指定時に拡張子を省略した場合、C++コンパイラは標準のファイル拡張子を付加したファイル名のファイルをコンパイルします。C++コンパイラおよび関連ソフトウェアで使用する標準のファイル拡張子を表 1.1 に示します。なお、ファイル名の付け方の一般的な規則は各ホストマシンに準じています。ご使用になるホストマシンのマニュアルを参照してください。

表 1.1 C++コンパイラで使用する標準のファイル拡張子

No.	拡張子	意味
1	cpp、cc、C	C++言語で記述されたソースプログラムファイル
2	h	インクルードファイル
3	lis、lst	リストファイル ^{*1}
4	obj	リロケータブルオブジェクトプログラムファイル
5	src	アセンブリソースプログラムファイル
6	lib	ライブラリファイル
7	abs	アブソリュートロードモジュールファイル
8	rel	リロケータブルロードモジュールファイル
9	map	リンケージマップリストファイル
10	dtb	デバッグ情報ファイル

【注】 *1 UNIX 版は lis、PC 版は lst を使用します。

1.3.3 コンパイラオプション

コンパイラオプションの形式と短縮形および省略時解釈の一覧を表 1.2 に示します。
下線部 () は短縮形指定時の文字を示します。また、斜体字は省略時解釈を示します。

表 1.2 コンパイラオプション一覧

No.	項目	形式	指定内容
1	CPU 種別	<u>cpu</u> = <i>SH1</i> <i>SH2</i> <i>SH3</i>	SH1 のオブジェクトを生成 SH2 のオブジェクトを生成 SH3 のオブジェクトを生成
2	最適化レベル	<u>optimize</u> = 0 1	最適化なしのオブジェクトを出力 最適化ありのオブジェクトを出力
3	最適化方法の 選択	<i>speed</i> <i>nospeed</i> <i>size</i>	実行速度優先のコードを生成 実行速度、サイズのバランスのとれたコードを生成 サイズ優先のコードを生成
4	デバッグ情報	<u>debug</u> <i>nodebug</i>	出力あり 出力なし
5	ブラウザ情報	<u>browser</u>	出力あり
6	リスト内容と 形式	<u>show</u> = <i>source</i> <i>nosource</i> <i>object</i> <i>noobject</i> <i>statistics</i> <i>nostatistics</i> <i>include</i> <i>noinclude</i> <i>expansion</i> <i>noexpansion</i> <u>width</u> = <数値> <u>length</u> = <数値> 省略時 : (<i>w</i> = 132, <i>l</i> = 66)	ソースリストの有無 オブジェクトリストの有無 統計情報の有無 インクルード展開後リストの有無 マクロ展開後リストの有無 1 行の最大文字数 数値: 0, 80 ~ 132 ページ内の最大行数 数値: 0, 40 ~ 255
7	リストファイル	<u>listfile</u> [= <ファイル名>] <i>nolistfile</i>	出力あり 出力なし
8	オブジェクト ファイル	<u>objectfile</u> = <ファイル名>	出力あり
9	オブジェクト 形式	<u>code</u> = <i>machinecode</i> <i>asmcode</i>	機械語プログラムを出力 アセンブリソースプログラムを出力
10	マクロ名の定義	<u>define</u> = <マクロ名>=<名前> <マクロ名>=<定数> <マクロ名>	<名前>を<マクロ名>として定義 <定数>を<マクロ名>として定義 <マクロ名>を定義したものと仮定
11	インクルード ファイル	<u>include</u> = <パス名>	インクルードファイルの取り込み先パス名を指定 (複数指定可)

No.	項目	形式	指定内容
12	セクション名	<u>section</u> = <u>program</u> = <セクション名> <u>const</u> = <セクション名> <u>data</u> = <セクション名> <u>bss</u> = <セクション名> 省略時 : (<i>p=P, c=C, d=D, b=B</i>)	プログラム領域セクション名指定 定数領域セクション名指定 初期化データ領域セクション名指定 未初期化データ領域セクション名指定
13	ヘルプ メッセージ	<u>help</u>	出力あり
14	ポジションイン ディペンデント	<u>pic</u> = 0 1	ポジションインディペンデントコードを生成しない ポジションインディペンデントコードを生成する
15	プリプロセッサ 展開出力	<u>preprocessor</u>	プリプロセッサ展開後のソースプログラムを出力
16	文字列出力領域	<u>string</u> = <u>const</u> <u>data</u>	文字列を定数セクション (C) へ出力 初期化データセクション (D) へ出力
17	コメントの ネスト	<u>comment</u> = <u>nest</u> <u>nonest</u>	コメント (/* */) のネストを許す コメント (/* */) のネストを許さない
18	文字列内の日本 語コードの選択	<u>euc</u> <u>sjis</u>	euc コードを選択 sjis コードを選択
19	サブコマンド ファイルの選択	<u>subcommand</u> = <ファイル名>	<ファイル名>で指定したファイルからコマンドオプションを取り込む
20	除算の方式	<u>division</u> = <u>cpu</u> <u>peripheral</u> <u>nomask</u>	cpu の除算命令を使用 除算器を使用 (割り込みマスクあり) 除算器を使用 (割り込みマスクなし)
21	メモリのビット 並び順の指定	<u>endian</u> = <u>big</u> <u>little</u>	Big Endian Little Endian
22	インライン展開 の仕様	<u>inline</u> <u>inline</u> = <数値> <u>noinline</u>	インライン展開するかどうかを指定 インライン展開する関数のサイズの限界を指定
23	デフォルトの ヘッダファイル の指定	<u>preinclude</u> = <ファイル名>	指定したファイルの内容をコンパイル単位の先頭に 取り込む
24	MACH、MACL レジスタの保証	<u>macsave</u> = 0 1	関数呼び出しで MACH、MACL レジスタを保証しない 関数呼び出しで MACH、MACL レジスタを保証する

-cpu = SH1 | SH2 | SH3

ターゲット CPU を指定します。選択する cpu により、リンクするライブラリが異なります。詳しくは、「第 1 章 概要・操作 1.3.5 標準ライブラリとの対応」を参照してください。

-optimize = 0 | 1

コンパイラの最適化を行うかどうかを指定します。optimize=0 は最適化を抑止します。optimize=1 は最適化を行います。

-speed, -nospeed

speed 優先の最適化を行います。speed オプションを指定するとプログラムの実行速度は向上しますが、プログラムサイズが増大する場合があります。size 指定がなく、nospeed が有効の場合は、実行速度、サイズのバランスをとった最適化を行います。

-size

オブジェクトサイズ優先の最適化を行います。

-debug

C++ソースレベルデバッグに必要なデバッグ情報を出力します。C++コンパイラは、コンパイルされた C++ソースの存在するディレクトリに cppdtb というディレクトリがなければ作成し、その下にデバッグ情報ファイルを作成します。

-browser

SH C++対応ブラウザに必要なデバッグ情報を出力します。ブラウザを使用するときは必ず本オプションを指定してください。本オプションを指定した場合、debug オプションを同時に指定する必要はありません。C++コンパイラは、コンパイルされた C++ソースの存在するディレクトリに cppdtb というディレクトリがなければ作成し、その下にデバッグ情報ファイルを作成します。

-show = source | nosource | object | noobject | statistics | nostatistics | include | noinclude | expansion | noexpansion | width = <数値> | length = <数値>

show オプションはリストファイルの出力形式を指定します。show オプションは listfile オプション指定時に有効になります。show = width = 0、または show = length = 0 を指定した場合、次のように解釈します。

show = width = 0 改行コードが出力されるまでを 1 行とします。

show = length = 0 最大行数は設定せず、改頁は行いません。

-listfile [= <リストファイル名>], -nolistfile

リストファイルを出力するかどうかを指定します。ファイル名の指定を省略した場合、ソースファイル名と同じファイル名に標準の拡張子 (lis/lst) 付加したファイルを生成します。

-objectfile = <ファイル名>

出力するオブジェクトファイル名を指定します。

-code = machinecode | asmcode

コンパイラが直接マシン語のオブジェクトファイルを出力するか、アセンブリソースファイルを出力するかを指定します。

-define = <マクロ名>=<名前> | <マクロ名>=<定数> | <マクロ名>

本オプションで指定されたマクロ定義をソースプログラムの先頭で有効にします。オプションで指定できるマクロ名の仕様を表 1.3 に示します。

表 1.3 define オプションで指定できるマクロ名、名前、定数

No.	項目	説明
1	マクロ名	英字またはアンダラインで始まり、そのあとに 0 個以上の英字、アンダラインまたは数字が続く文字列です。
2	名前	英字またはアンダラインで始まり、そのとに 0 個以上の英字、アンダラインまたは数字が続く文字列です。
3	定数	1 個以上の数字 (0 ~ 9) の繰り返し、または 1 個以上の数字の繰り返しにピリオドが続き、そのあとに 0 個以上の数字が続く文字列です。

-include = <パス名>

インクルードファイルを検索するディレクトリを指定します。検索方法の詳細は、「付録 A.1 (13) プリプロセッサ」を参照してください。

-section = | program=<セクション名> | const=<セクション名> | data=<セクション名> | bss=<セクション名>

オブジェクトプログラムのセクション名を変更します。本オプション省略時のセクション名は、program セクション P、const セクション C、data セクション D、bss セクション B です。

-help

コンパイラのオプション一覧を表示します。本オプションが指定された場合、他のオプションは無効になります。

-pic = 0 | 1

pic = 1 指定時は、リンク後のプログラムセクションを任意のアドレスに配置して実行できます。データセクションはリンク時に決定したアドレス以外には配置できません。ポジションインディペンデントコードとして実行する場合は、関数のアドレスを初期値として指定することはできません。C++プログラムでは、仮想関数は暗黙的にアドレスを初期値として使用するため、仮想関数を含むプログラムはpic 指定できません。また、メンバ関数へのポインタも同様です。cpu =SH1 指定時は、pic =1 指定を無視します。cpu、pic、endian オプションにより、リンクするライブラリが異なります。詳しくは、「第1章 概要・操作 1.3.5 標準ライブラリとの対応」を参照してください。

[例 1]

```
extern int f();  
int (*fp)() = f;      指定不可
```

[例 2]

```
class A {  
public:  
    virtual int f();    指定不可  
};
```

-preprocessor

プリプロセッサ展開後のソースプログラムを出力します。

-string = *const* | *data*

string = const を指定した場合、同一の文字列のメモリ領域を共有することができます。
string = data を指定した場合、同一の文字列のメモリ領域は別々に割り付けられます。

-comment = nest | *nonest*

コメント `/* */` のネストを許可するかどうかを指定します。以下に例を示します。

[例]

```
/* comment
   int a;  /* nest1 /* nest2 */ */
*/
```

`comment = nest` 指定時は、下線部がネストしたコメントと解釈され、一番外側のコメントが有効になります。

`comment = nonest` 指定時は、`nest2 */`でコメントが終了したと判断し、以降の`*/`がエラーになります。

-euc

C++ プログラム中の文字列内の日本語コードを `euc` コードと解釈します。本オプション省略時はホストマシンによって日本語コードの解釈が異なります。「第 2 章 C++プログラミング 2.3.5 文字列内の日本語記述」を参照してください。

-sjis

C++ プログラム中の文字列内の日本語コードを `sjis` コードと解釈します。本オプション省略時はホストマシンによって日本語コードの解釈が異なります。「第 2 章 C++プログラミング 2.3.5 文字列内の日本語記述」を参照してください。

-subcommand = <ファイル名>

指定されたファイル名の内容をオプションと解釈します。

`subcommand` オプションはコマンドラインの中に複数回指定できます。サブコマンドファイル内にはコマンドラインの引数を空白、改行またはタブで区切ってオプションを指定してください。サブコマンドファイルの内容がコマンドライン引数の `subcommand` 指定位置に展開されます。サブコマンドファイル内に `subcommand` オプションを指定することはできません。

[例]

下記の例は、コマンドライン `shcpp -debug -cpu = sh2 test.cc` と等価になります。

コマンドライン

```
shcpp -sub = test.sub test.cc
```

test.sub の内容

```
-debug
-cpu = sh2
```

-division = *cpu* | *peripheral* | *nomask*

C++ ソース中の整数型除算、剰余算に対する実行時ルーチンを選択します。本オプションは *cpu* オプションの任意のサブオプションと組み合わせが可能ですが *peripheral*、*nomask* を指定したオブジェクトプログラムは SH2 以外では実行できません。

- (1) *cpu* DIV1 命令による実行時ルーチンを選択
- (2) *peripheral* 除算器を用いた実行時ルーチンを選択 (割り込みマスクに 15 を設定)
- (3) *nomask* 除算器を用いた実行時ルーチンを選択 (割り込みマスクは変更なし)

peripheral、*nomask* 指定時は以下の点に注意してください。

- (1) ゼロ除算のチェックおよび *errno* の設定は行いません。
- (2) 除算器動作中に割り込みがかかり、割り込みルーチンで除算器を用いた場合、動作は保証しません。
- (3) オーバフロー割り込みはサポートしていません。
- (4) ゼロ除算、オーバフローなどの演算結果は除算器の仕様に従います。*cpu* サブオプション指定時と異なる場合があります。

-endian = *big* | *little*

本オプションは *cpu* オプションの任意のサブオプションと組み合わせが可能ですが、*little endian* のオブジェクトプログラムは、SH3 以外では実行できません。*cpu*、*pic*、*endian* オプションにより、リンクするライブラリが異なります。詳しくは、「第 1 章 概要・操作 1.3.5 標準ライブラリとの対応」を参照してください。

-inline, -inline = <数値>, -noinline

関数の自動インライン展開をするかしないかを指定します。サブオプションの数値は、インライン展開をする関数の最大サイズを関数のノード数 (宣言部を除く変数、演算子等の語句の総数) で示すものです。

speed オプション指定時のデフォルトは、*inline* = 20 です。*nospeed*、*size* オプション指定時、または *optimize* = 0 オプション指定時のデフォルトは *noinline* です。

-preinclude = <ファイル名>

指定したファイルの内容をコンパイル単位の前頭に取り込みます。

-macsave = 0 | 1

MACH、MACL レジスタを関数の呼び出し前後で保証するかどうかを指定します。
 macsave = 0 は関数の呼び出し前後で MACH、MACL レジスタを保証しません。macsave = 1 は関数の呼び出し前後で MACH、MACL レジスタを保証します。macsave = 1 でコンパイルした関数から macsave = 0 でコンパイルした関数を呼び出すことはできません。逆に macsave = 0 でコンパイルした関数から macsave = 1 でコンパイルした関数を呼び出すことは可能です。

1.3.4 オプションの組み合わせ

コンパイラオプションの組み合わせで、意味上矛盾するオプションやサブオプションを同時に指定した場合、どちらか一方が無効になります。表 1.4 にオプションの組み合わせを示します。

表 1.4 オプションの組み合わせ

No.	オプションの組み合わせ	
	有効となるオプション	無効となるオプション
1	nolist	show
2	code = asmcode *	browser *
3		debug *
4		show = object
5	help	すべてのオプション
6	cpu = sh1	pic = 1
7	optimize = 0	inline
8	nospeed、size	

【注】 * アセンブリソース出力時に browser または debug オプションを指定すると、出力コード内に .LINE 制御命令を埋め込みます。 .LINE 制御命令は、C++言語ソース行情報をデバッガに与えます。これによって、デバッグ時に対応する C++言語ソース行を表示することができます。ただし変数の値に関しては C++言語レベルのデバッグはできません。また、ブラウザによるブラウジングができません。

1.3.5 標準ライブラリとの対応

標準ライブラリには、次の 7 種類があります。cpu オプション、pic オプションおよび endian オプションの組み合わせにより表 1.5 に示すライブラリをリンクしてください。

shclib.lib (SH1 用)

shcnpic.lib (SH2 用ポジション・インディペンデント・コード非対応)

shcpic.lib (SH2 用ポジション・インディペンデント・コード対応)

shc3npb.lib (SH3 用ポジション・インディペンデント・コード非対応、Big Endian)

shc3pb.lib (SH3 用ポジション・インディペンデント・コード対応、Big Endian)

ahc3npl.lib (SH3 用ポジション・インディペンデント・コード非対応、Little Endian)

shc3pl.lib (SH3 用ポジション・インディペンデント・コード対応、Little Endian)

表 1.5 標準ライブラリとコンパイルオプションの関係

endian 指定	endian =big		endian = little	
pic 指定	pic = 0	pic = 1	pic = 0	pic = 1
cpu = sh1	shclib.lib	-	-	-
cpu = sh2	shcnpic.lib	shcpic.lib	-	-
cpu = sh3	shc3npb.lib	shc3pb.lib	shc3npl.lib	shc3pl.lib

1.3.6 コンパイルリストの見方

本節では、コンパイルリストの内容と形式について説明します。

(1) コンパイルリストの構成

コンパイルリストの構成と内容を表 1.6 に示します。

表 1.6 コンパイルリストの構成と内容

No.	リストの構成	内容	オプション指定方法* ¹	オプション省略時
1	ソースリスト情報	ソースプログラムリスト	show=[no]source	出力する
		インクルードファイル、マクロ展開後のソースプログラムのリスト	(show=[no]include) * ² (show=[no]expansion)	出力しない
2	オブジェクト情報	オブジェクトプログラムの機械語、アセンブリコード	show=[no]object	出力する
3	統計情報	エラーの総数、ソースプログラムの行数、セクションサイズ、シンボル数	show=[no]statistics	出力する
4	コマンド指定情報	コマンドで指定されたファイル名とオプション表示		出力する

【注】 *1 全てのオプションは listfile 指定時に有効です。

*2 () 内は show = source 指定時に有効になります。

(2) ソースリスト情報

ソースリスト情報の出力形式には、プリプロセッサ展開前のソースプログラムを出力する形式 (show = noinclude, noexpansion を指定する場合) とプリプロセッサ展開後のソースプログラムを出力する形式 (show = include, expansion を指定する場合) があります。それぞれの出力形式を図 1.3、図 1.4 に示します。また、図 1.4 に相違点を網掛けで示します。

```

***** SOURCE LISTING *****

FILE NAME:m0260.cpp

Seq   File      Line   0-----1-----2-----3-----4-----5--
  1 m0260.cpp    1      #include "header.h"
  4 m0260.cpp    2
  5 m0260.cpp    3      int sum2(void)
  6 m0260.cpp    4      {   int j;
  7 m0260.cpp    5
  8 m0260.cpp    6      #ifdef SMALL
  9 m0260.cpp    7          j=SML_INT;
10 m0260.cpp    8      #else
11 m0260.cpp    9          j=LRG_INT;
12 m0260.cpp   10      #endif
13 m0260.cpp   11
14 m0260.cpp   12          return j; /* continue123456789012345678901234567
(1)   (2)      (3)      ±2345678901234567890  */
15 m0260.cpp   13      (7)   }

```

図 1.3 show = noinclude、noexpansion のソースリスト情報

```

***** SOURCE LISTING *****

FILE NAME:m0260.cpp

Seq   File      Line   0-----1-----2-----3-----4-----5--
  1 m0260.cpp    1      #include "header.h"
  2 header.h     1      #define SML_INT      1      (4)
  3 header.h     2      #define LRG_INT     100
  4 m0260.cpp    2
  5 m0260.cpp    3      int sum2(void)
  6 m0260.cpp    4      {   int j;
  7 m0260.cpp    5
  8 m0260.cpp    6      #ifdef SMALL
  9 m0260.cpp    7      X      j=SML_INT;
10 m0260.cpp    8      (5) #else
11 m0260.cpp    9      E      j=100;
12 m0260.cpp   10      (6) #endif
13 m0260.cpp   11
14 m0260.cpp   12          return j; /* continue123456789012345678901234567
(1)   (2)      (3)      ±2345678901234567890  */
15 m0260.cpp   13      (7)   }

```

- (1) リスト上の行番号
(2) ソースプログラムファイル名またはインクルードファイル名
(3) ソースプログラムまたはインクルードファイル内の行番号
(4) show=include指定時、インクルードファイルの展開のあったソース行
(5) show=expansion指定時、#ifdef文、#elif文等の条件コンパイル文でコンパイル対象とならないソース行
(6) show=expansion指定時、#define文によるマクロ置換のあったソース行
(7) ソースプログラムの1行がコンパイルリストの1行に入りきらず、複数行にまたがって表示されたソース行

図 1.4 show = Include、expansion のソースリスト情報

(3) オブジェクト情報

オブジェクト情報のリスト例を図 1.5 に示します。

```
***** SOURCE LISTING *****
FILE NAME:m0251.cpp

Seq   File      Line  0-----1-----2-----3-----4-----5-
1 m0251.cpp      1      extern int sum(int);
2 m0251.cpp      2
3 m0251.cpp      3      int
4 m0251.cpp      4      sum(int x)
5 m0251.cpp      5      {
6 m0251.cpp      6          int i;
7 m0251.cpp      7          int j;
8 m0251.cpp      8
9 m0251.cpp      9          j=0;
10 m0251.cpp     10          for(i=0; i<=x; i++) {
11 m0251.cpp     11              j+=i;
12 m0251.cpp     12          }
13 m0251.cpp     13          return j;
14 m0251.cpp     14      }
```

```
***** OBJECT LISTING *****
FILE NAME:m0251.cpp
```

SCT (1)	OFFSET (2)	CODE (3)	C LABEL	INSTRUCTION OPERAND (4)	COMMENT (5)
P				<u>; File m0251.cpp ,Line 4</u>	<u>;block</u>
	00000000		_sum:	(6)	<u>;function: sum</u>
					<u>;frame_size=8</u>
	00000000	7FF8		ADD #-8,R15	(7)
				;File m0251.cpp ,Line 5	;block
				;File m0251.cpp ,Line 9	;expression statement
	00000002	E300		MOV #0,R3	
	00000004	2F32		MOV.L R3,@R15	
				;File m0251.cpp ,Line 10	;for
	00000006	E300		MOV #0,R3	
	00000008	1F31		MOV.L R3,@(4,R15)	
	0000000A	A009		BRA L104	
	0000000C	0009		NOP	
	0000000E		L105:		
				;File m0251.cpp ,Line 10	;block
				;File m0251.cpp ,Line 11	;expression statement
	0000000E	52F1		MOV.L @(4,R15),R2	
	00000010	61F3		MOV R15,R1	
	.		.	.	
	.		.	.	

(1) 各セクションのセクション属性 (P、C、D、D_INIT_、D_END_、B)
 (2) 各セクションの先頭からのオフセット
 (3) 各セクションのオフセットアドレスの内容
 (4) 機械語に対応するアセンブリコード
 (5) プログラムに対応するコメント (非最適化時だけ出力、ラベルだけ最適化時も出力)
 (6) プログラムに対応する行情報 (非最適化時だけ出力)
 (7) スタックフレームサイズ (バイト数) (最適化時も出力)

図 1.5 オブジェクト情報

(4) 統計情報

統計情報の出力例を図 1.6 に示します。

```

***** STATISTICS INFORMATION *****

***** ERROR INFORMATION ***** (1)

NUMBER OF ERRORS:      0
NUMBER OF WARNINGS:    0

***** SOURCE LINE INFORMATION ***** (2)

COMPILED SOURCE LINE:  13

***** SECTION SIZE INFORMATION ***** (3)

PROGRAM  SECTION(P):      0x00004A Byte(s)
PROGRAM  SECTION(P_INIT_): 0x000000 Byte(s)
PROGRAM  SECTION(P_END_):  0x000000 Byte(s)
CONSTANT SECTION(C):      0x000000 Byte(s)
CONSTANT SECTION(CINIT_):  0x000000 Byte(s)
CONSTANT SECTION(C_END_):  0x000000 Byte(s)
DATA     SECTION(D):      0x000000 Byte(s)
DATA     SECTION(DINIT_):  0x000000 Byte(s)
DATA     SECTION(D_END_):  0x000000 Byte(s)
BSS      SECTION(BINIT_):  0x000000 Byte(s)
BSS      SECTION(B):       0x000000 Byte(s)
BSS      SECTION(B_END_):  0x000000 Byte(s)

TOTAL PROGRAM SIZE: 0x00004A Byte(s)

***** LABEL INFORMATION ***** (4)

NUMBER OF EXTERNAL REFERENCE SYMBOLS: 0
NUMBER OF EXTERNAL DEFINITION SYMBOLS: 1
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:  4

```

- (1) レベル別メッセージの総数
- (2) ソースファイルのコンパイルした行数
- (3) 各セクションのサイズとその合計
- (4) オブジェクトプログラムの外部参照シンボルの数、外部定義シンボルの数
- (5) 内部ラベルと外部ラベルの合計数

【注】オプションnoobject指定時およびエラーレベル、フェータルレベルのエラーが発生した場合には、セクションサイズ情報(3)とラベル情報(4)を出力しません。
また、オプションcode=asmcode指定時には、セクションサイズ情報(3)は当該セクションの有無を0と1で示すようになります。

図 1.6 統計情報

(5) コマンド指定情報

コンパイラを起動したときのコマンドで指定されたファイル名とオプションを表示します。コマンド指定情報の出力例を図 1.7 に示します。

```
*** COMMAND PARAMETER ***
-listfile test.cpp
```

図 1.7 コマンド指定情報

1.3.7 コンパイラ環境変数

コンパイラで使用する環境変数の使用方法を表 1.7 に示します。

表 1.7 環境変数

No.	環境変数	説明
1	SHC_LIB	コンパイラのロードモジュールおよびシステムインクルードファイルを格納したディレクトリを指定します。
2	SHC_INC	システムインクルードファイル格納ディレクトリを指定します。 ディレクトリはコンマで区切ることによって複数指定可能です。システムインクルードファイルの検索順序は include オプション指定ディレクトリ、SHC_INC 指定ディレクトリ、システムディレクトリ (SHC_LIB) となります。
3	SHC_TMP	コンパイラがテンポラリファイルを作成するディレクトリを指定します。PC 版ではこの環境変数の指定は必須です。UNIX 版ではこの環境変数の指定がない場合、環境変数 TMPDIR が指定されていれば、TMPDIR が示すディレクトリ。SHC_TMP、TMPDIR が指定されていなければ、/usr/tmp にテンポラリファイルを作成します。

2. C++プログラミング

第2章 目次

2.1	C++コンパイラの限界値	25
2.2	C++プログラムの実行方式	27
	2.2.1 オブジェクトプログラムの構造	27
	2.2.2 データの内部表現	29
	2.2.3 Cプログラムとの結合	38
	2.2.4 アセンブリプログラムとの結合	39
2.3	拡張機能	48
	2.3.1 割り込み関数の使用方法	48
	2.3.2 組み込み関数の使用方法	51
	2.3.3 セクション切り替え機能	56
	2.3.4 単精度浮動小数点ライブラリ	57
	2.3.5 文字列内の日本語記述	59
	2.3.6 関数のインライン展開	59
	2.3.7 アセンブラ埋め込みインライン展開	60
	2.3.8 2バイトアドレス変数の指定	62
	2.3.9 GBR ベース変数の指定	62
	2.3.10 レジスタ退避・回復の制御	63
2.4	プログラム作成上の注意事項	65
	2.4.1 コーディング上の注意事項	65
	2.4.2 プログラム開発上のトラブル対処方法	67

2.1 C++コンパイラの限界値

C++コンパイラがコンパイルできるソースプログラムの限界値を表 2.1 に示します。ソースプログラムを作成する場合は、この限界値の範囲内で作成してください。ソースプログラムの編集やコンパイル処理の効率を上げるためには、最大 2000 行程度までのプログラムに分割して、分割コンパイルをすることをお勧めします。

表 2.1 C++コンパイラの限界値

NO.	分類	項目	限界値
1	コンパイラの起動	一度にコンパイルできるソースプログラムの数	制限なし ^{*1}
2		define オプションで指定できるマクロ名の総数	制限なし
3		ファイル名の長さ	128 文字
4	ソースプログラム の行数	1 行の長さ	4096 文字
5		ソースプログラムの行数 (1 ファイル)	65535 行
6	プリプロセッサ	ソースプログラムの行数 (コンパイル単位)	制限なし
7		#include 文によるファイルのネストの深さ	30 レベル
8		#define 文によるマクロ名の総数	制限なし
9		マクロ定義、マクロ呼び出しで指定できる引数の数	63 個
10		マクロ名の再置き換えの数	32 回
11		#if、#ifdef、#ifndef、#else、#elif 文のネストの深さ	32 レベル
12		#if、#elif 文で指定できる演算子と被演算子の合計数	512 個
13	宣言	関数定義の数	512 個
14		内部ラベルの数 ^{*2}	32767 個
15		シンボルテーブルエントリ数 ^{*3}	24576 個
16		基本型を修飾するポインタ型、配列型、関数型の合計数	16 個
17		配列の次元数	6 次元
18	文	構文のネストの深さ	32 レベル
19		繰り返し文 (while 文、do 文、for 文)、選択文 (if 文、switch 文) の組み合わせによる文のネストの深さ	32 レベル
20		1 つの関数内で指定できる goto ラベルの数	511 個
21		switch 文の数	256 個
22		switch 文のネストの深さ	16 レベル
23		case ラベルの数	511 個
24	式	for 文のネストの深さ	16 レベル
25		関数定義、関数呼び出しで指定できる引数の数	63 個
26		1 つの式で指定できる演算子と被演算子の合計数	約 500 個

【注】 *1 ただし、PC 版はコマンドラインの制約により 127 文字までの入力となります。

*2 内部ラベルとは、コンパイラが内部で生成するラベルであり、静的変数の領域を指すアドレス、繰り返し文や選択文で処理の流れが分岐する先のアドレス、case ラベルや goto ラベルのアドレスなどのことです。

*3 シンボルテーブルエントリ数を概算する式を以下に示します。

外部名の数+関数ごとの内部名の数+文字列の数+複文内のクラス・配列の初期値

+複文の数+case ラベルの数+goto ラベルの数

+ (デフォルトコンストラクタ/デストラクタの数)

+ (コピーコンストラクタ/代入演算子の数) + (仮想関数表の数)

() で表しているものはコンパイラで自動生成する外部名です。仮想関数表が生成される場合については、「第 2 章 C++プログラミング 2.2.2 データの内部表現 (2) 利用者定義型」を参照してください。

2.2 C++プログラムの実行方式

本章では、C++コンパイラが生成するオブジェクトプログラムについて説明します。特に、C++プログラムとアセンブリプログラム、C++プログラムとCプログラムを結合する場合に必要なデータの内部表現や、上記の異言語間のインタフェースについて説明しています。

本章で述べる項目は以下のとおりです。

2.2.1 オブジェクトプログラムの構造

C++プログラム、標準ライブラリが使用するメモリ領域の性質について述べます。

2.2.2 データの内部表現

C++プログラムが用いるデータ型のメモリ上での表現について述べます。C++プログラムとハードウェア、アセンブリプログラムの中でデータを相互参照するときに必要です。

2.2.3 Cプログラムとの結合

C++プログラムで使用する変数名や関数名のうち、他のオブジェクトプログラムとの間で相互に参照できる名前の規則について述べます。C++プログラム中でCプログラムの関数の呼び出しを行うときに必要です。

2.2.4 アセンブリプログラムとの結合

C++プログラムで使用する変数名や関数名のうち、他のオブジェクトプログラムとの間で相互に参照できる名前の規則について述べます。また、C++プログラムの関数呼び出しでの引数やリターン値の受け渡し方法、レジスタの使用法に関する規則について述べます。これらの規則は、C++プログラムの関数とアセンブリプログラムのルーチン間で相互に呼び出しや参照を行うときに必要です。

本章では、SHのハードウェアの知識を必要としますので、ハードウェアマニュアルをあわせてお読みください。

2.2.1 オブジェクトプログラムの構造

本節では、C++プログラム、標準ライブラリが使用するメモリ領域の性質について述べます。メモリ領域の性質には、以下の項目があります。

セクション

メモリ領域のうち、C++コンパイラが静的に割り付ける領域は、セクションを構成します。セクションにはセクション名とセクション種別があります。セクション名はコンパイラオプション `section` で変更することができます。

書き込み操作

プログラム実行時における書き込み操作の可/不可を示します。

初期値の有無

プログラム実行開始時の初期値の有無です。

境界調整数

データを割り付けるアドレスに関する制約です。

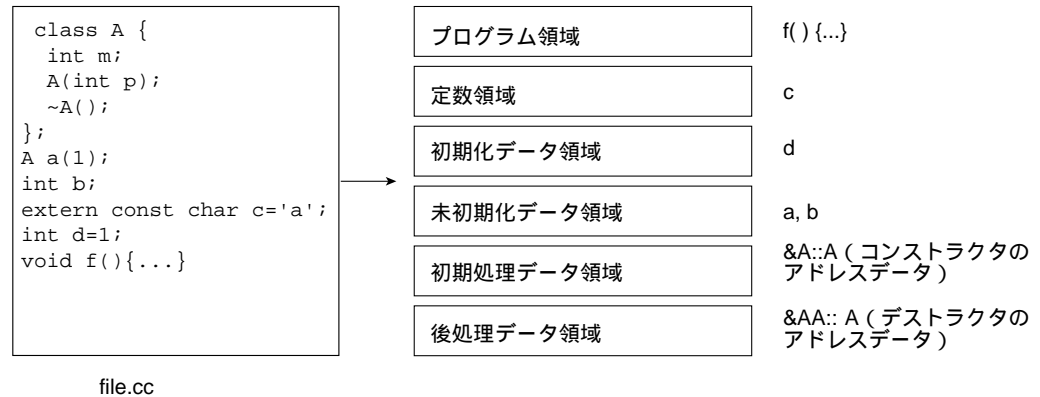
C++ プログラム、標準ライブラリが使用するメモリ領域の種類とその性質の概要を表 2.2 に示します。

表 2.2 メモリ領域の種類とその性質の概要

No.	名称	セクション名*	種別	書き込み	初期値	境界調整数	内容
1	プログラム領域	P	code	不可	有	4byte	機械語を格納する。
2	定数領域	C	data	不可	有	4byte	const 型のデータを格納する。
3	初期化データ領域	D	data	可	有	4byte	コンストラクタによって初期化しない初期値データを格納する。
4	未初期化データ領域	B	data	可	無	4byte	初期値のないデータを格納する。コンストラクタによって初期化されるデータを格納する。
5	初期処理データ領域	D_INIT_	data	不可	有	4byte	グローバルオブジェクトに対して呼び出されるコンストラクタのアドレスを格納する。
6	後処理データ領域	D_END_	data	不可	有	4byte	グローバルオブジェクトに対して呼び出されるデストラクタのアドレスを格納する。
7	スタック領域	-	-	可	無	4byte	プログラムの実行に必要な領域。 「第 3 章 システム組み込み 3.2.2 動的領域の割り付け」参照。
8	ヒープ領域	-	-	可	無	-	ライブラリ関数 (new、malloc、realloc、calloc) で使用する領域。 「第 3 章 システム組み込み 3.2.2 動的領域の割り付け」参照。

【注】 * セクション名はコンパイラオプション section で特定の名前を指定しないときに C++ コンパイラがデフォルトで作成する名前を示します。

[例] C++ プログラムとコンパイラが生成するセクションとの対応をプログラム例を用いて示します。



2.2.2 データの内部表現

本節では、C++言語のデータ型と、データの内部表現の対応について述べます。データの内部表現は、以下の項目から成り立っています。

(a) データのサイズ

データが占有する領域のサイズです。

(b) データの境界調整数

データを割り付けるアドレスに関する制約です。任意のアドレスに割り付ける1バイト境界調整、偶数バイトに割り付ける2バイト境界調整、4の倍数バイトに割り付ける4バイト境界調整があります。

(c) データの範囲

組み込み型の値がとり得る範囲を示します。

(d) データの割り付け例

利用者定義型の要素となるデータの割り付け例を示します。

(1) 組み込み型

C++言語における組み込み型の内部表現を表 2.3 に示します。

表 2.3 組み込み型の内部表現

NO.	データ型	サイズ (バイト)	境界整合数 (バイト)	符号の 有無	最小値	最大値
1	char	1	1	有	-2^7 (-128)	2^7-1 (127)
2	signed char	1	1	有	-2^7 (-128)	2^7-1 (127)
3	unsigned char	1	1	無	0	2^8-1 (255)
4	short	2	2	有	-2^{15} (-32768)	$2^{15}-1$ (32767)
5	unsigned short	2	2	無	0	$2^{16}-1$ (65535)
6	int	4	4	有	-2^{31} (-2147483648)	$2^{31}-1$ (2147483647)
7	unsigned int	4	4	無	0	$2^{32}-1$ (4294967295)
8	long	4	4	有	-2^{31} (-2147483648)	$2^{31}-1$ (2147483647)
9	unsigned long	4	4	無	0	$2^{32}-1$ (4294967295)
10	enum	4	4	有	-2^{31} (-2147483648)	$2^{31}-1$ (2147483647)
11	float	4	4	有	-	+
12	double long double	8	4	有	-	+
13	ポインタ	4	4	無	0	$2^{32}-1$ (4294967295)
14	リファレンス	4	4	無	0	$2^{32}-1$ (4294967295)
15	データメンバ へのポインタ	4	4	有	-2^{31} (-2147483648)	$2^{31}-1$ (2147483647)
15	関数メンバ へのポインタ*1	-	4	-	-	-
16	配列*2	-	-	-	-	-

【注】 *1 関数メンバへのポインタの型は以下のクラスで表現しています。

```
class _PMF{
public:
    short int delta; オブジェクトのオフセット値
    short int index; 対象関数が仮想関数のときの仮想関数表中でのインデックス
    union{
        int(*_deffun)(); 対象関数是非仮想関数のときの関数のアドレス
        short int vt_offset; 対象関数が仮想関数のときの仮想関数表のオブジェ
                               クト中のオフセット
    };
};
```

*2 配列型は、以下の境界調整数、サイズとなります。

No.	データ型	境界調整数 (byte)	サイズ (byte)	データの割り付け例
1	配列型	配列要素の 境界調整数	配列要素の数 × 要素サイズ	int a[10] : 境界調整数 4byte サイズ 40byte


(2) 利用者定義型

本項では、クラス（構造体）型および共用体型の内部表現について説明します。

表 2.4 に利用者定義型の内部表現を示します。

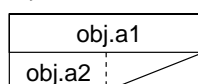
表 2.4 利用者定義型の内部表現

No.	データ型	境界調整数 (byte)	サイズ (byte)	データの割り付け例
1	クラス型*1	クラスメンバ、仮想関数表へのポインタ*2、仮想基底クラスへのポインタ*3のうちの境界調整数の最大値	メンバのサイズの和+仮想関数表へのポインタ数×4byte +仮想基底クラスへのポインタ数×4byte*4	class{ int a,b; : 境界調整数 4byte virtual void (); サイズ 12byte };
2	共用体型	共用体メンバのうち最大値	メンバのサイズの最大値*5	union { : 境界調整数 4byte int a,b;}; サイズ 4byte

以下の例で  は、4バイトを表しています。

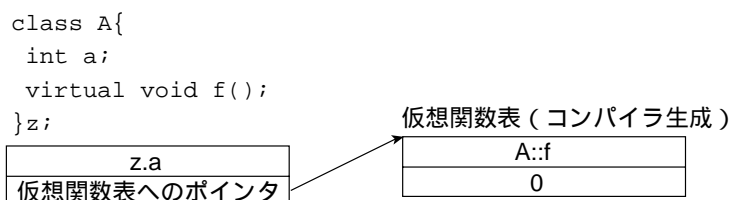
【注】 *1 クラス型の割り付け例

```
class A{
    int a1;
    short a2;
public:
    A();
    int getdata();
}obj;
```

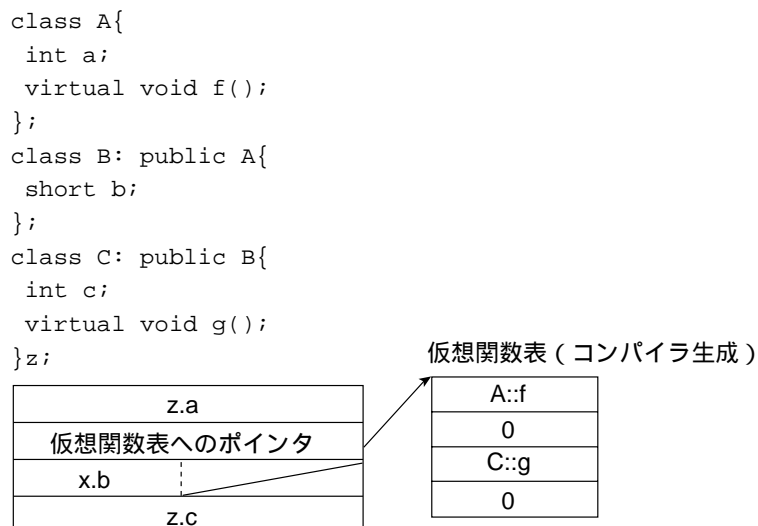


*2 仮想関数表へのポインタとは、（純粹）仮想関数を持つクラスの仮想関数表を指すポインタです。仮想関数表とは、そのクラスで仮想関数宣言された関数へのポインタと基底／派生クラスの派生クラス中でのオブジェクトのオフセット値が格納されているテーブルです。また、仮想関数表へのポインタは、基底クラス列中の第一基底クラス（基底クラス列の先頭のクラス）に（純粹）仮想関数があるときには仮想関数表へのポインタは生成されません。

(a) クラスに仮想関数がある場合のコーディング例とそのメモリ配置

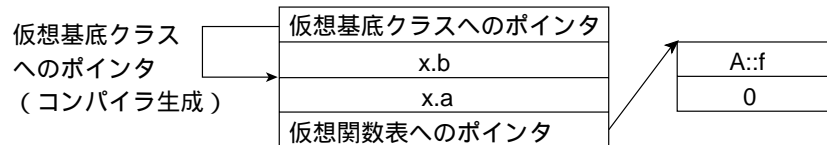


(b) 第一基底クラスに仮想関数がある場合のコーディング例とそのメモリ配置

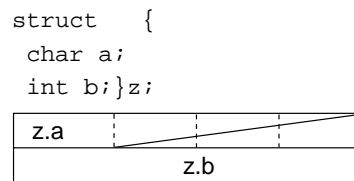


- *3 仮想基底クラスへのポインタは、仮想基底クラス宣言されたクラスごとにクラスオブジェクト内に領域確保されます。

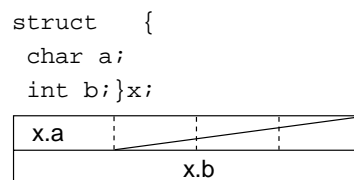
```
class A{
    int a;
    virtual void f();
};
class B: virtual public A{
    int b;
}x;
```



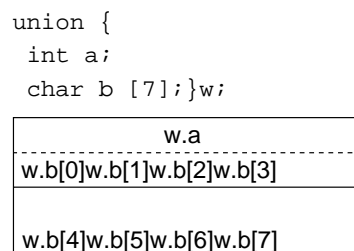
- *4 クラス型の各メンバを割り付ける時、そのメンバのデータ型の境界調整数に合わせるために直前のメンバとの間に空き領域が生じる場合があります。



また、クラスが4バイトの境界調整数を持つ場合で、最後のメンバが1、2、3バイト目で終わっているとき、残りのバイトも含めてクラス型の領域として扱います。



- *5 共用体が4バイトの境界調整数を持つ場合で、メンバのサイズの最大値が4の倍数バイトでないとき、4の倍数になるまで残りのバイトも含めて共用体型の領域として扱います。



(3) ビットフィールド

ビットフィールドは、クラスの中にビット幅を指定して割り付けるメンバです。本項ではビットフィールド特有の割り付け規則について説明します。

(a) ビットフィールドのメンバ

表 2.5 にビットフィールドメンバの仕様を示します。

表 2.5 ビットフィールドメンバの仕様

No.	項目	仕様
1	ビットフィールドで許される型指定子	char, unsigned char, signed char short, unsigned short int, unsigned int long, unsigned long
2	宣言された型に拡張するときの符号の扱い*1	符号なし (unsigned を指定した型) ゼロ拡張*2 符号あり (unsigned を指定しない型) 符号拡張*3

【注】 *1 ビットフィールドのメンバを使用する場合は、ビットフィールドに格納したデータを、宣言した型に拡張して使用します。符号付き (signed) で宣言されたサイズが 1 ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は 0 と -1 だけになります。0 と 1 を表現する場合には、必ず符号なし (unsigned) で宣言してください。

*2 ゼロ拡張：拡張するときに上位のビットにゼロを補います。

*3 符号拡張：拡張するときにビットフィールドデータの最上位ビットを符号として解釈し、データより上位のビット全てに符号ビットを補います。

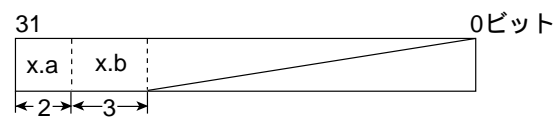
(b) ビットフィールドの割り付け方

ビットフィールドは、以下の5つの規則に従って割り付けます。

(1) ビットフィールドのメンバは領域内で左（上位ビット側）から順に詰め込みます。

[例]

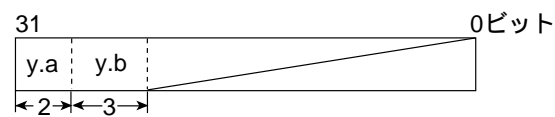
```
struct b1{
    int a:2;
    int b:3;
}x;
```



(2) 同じサイズの型指定子が連続している場合は、可能な限り同じ領域に詰め込みます。

[例]

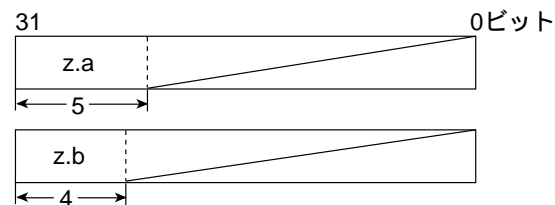
```
struct b1{
    long      a:2;
    unsigned int b:3;
}y;
```



(3) 異なるサイズの型指定子で宣言されたメンバは、次の領域に割り付けます。

[例]

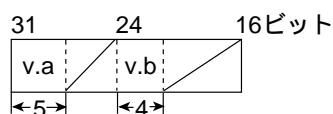
```
struct b1{
    int a:5;
    int b:4;
}z;
```



- (4) 同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。

[例]

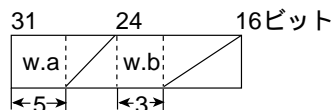
```
struct b2{
    char a:5;
    char b:4;
}v;
```



- (5) ビット幅0のビットフィールドのメンバを指定すると、次のメンバからは、強制的に次の領域に割り付けます。

[例]

```
struct b2{
    char a:5;
    char :0;
    char c:3;
}w;
```



(4) little endian のメモリ割り付け

little endian でのメモリ上のデータ配列は以下の通りです。

- (a) 1 バイトデータ (char, unsigned char, signed char 型)

1 バイトデータの中のビット並び順は、big endian の場合も、little endian の場合も同じです。

- (b) 2 バイトデータ (short, unsigned short 型)

2 バイトデータの中のバイトの並び順は、上位、下位のバイトが逆になります。

[例]

0x100 番地に 2 バイトデータ 0x1234 がある場合 :

big endian : 0x100 番地 : 0x12	little endian : 0x100 番地 : 0x34
0x101 番地 : 0x34	0x101 番地 : 0x12

(c) 4 バイトデータ (int、unsignedint、long、unsigned long、float 型)

4 バイトデータの中のバイト並び順は、big endian と little endian の間で 4 バイトのデータの順序が逆になります。

[例]

0x100 番地に 4 バイトデータ 0x12345678 がある場合：

big endian : 0x100 番地 : 0x12	little endian : 0x100 番地 : 0x78
0x101 番地 : 0x34	0x101 番地 : 0x56
0x102 番地 : 0x56	0x102 番地 : 0x34
0x103 番地 : 0x78	0x103 番地 : 0x12

(d) 利用者定義型データ

利用者定義型データの各メンバの割り付けは big endian のときと同様です。ただし、各メンバのバイトの並び順はそのデータサイズの規則にしたがって反転します。

[例]

0x100 番地に、

```
struct {
    short a;
    int b;
}z = {0x1234, 0x56789abc};
```

がある場合

big endian : 0x100 番地 : 0x12	little endian : 0x100 番地 : 0x34
0x101 番地: 0x34	0x101 番地 : 0x12
0x102 番地 : 空き領域	0x102 番地 : 空き領域
0x103 番地 : 空き領域	0x103 番地 : 空き領域
0x104 番地 : 0x56	0x104 番地 : 0xbc
0x105 番地 : 0x78	0x105 番地 : 0x9a
0x106 番地 : 0x9a	0x106 番地: 0x78
0x107 番地 : 0xbc	0x107 番地: 0x56

(e) ビットフィールド

ビットフィールドの各領域の割り付けも big endian のときと同様です。ただし、各領域のバイト並び順はそのデータサイズの規則に従って反転します。

[例]

0x100 番地に、

```
struct {  
    long a:16;  
    unsigned int b:15;  
    short c:5  
}y={1,1,1};
```

がある場合

big endian : 0x100 番地 : 0x00	little endian: 0x100 番地 : 0x02
0x101 番地 : 0x01	0x101 番地 : 0x00
0x102 番地 : 0x00	0x102 番地 : 0x01
0x103 番地 : 0x02	0x103 番地 : 0x00
0x104 番地 : 0x08	0x104 番地 : 0x00
0x105 番地 : 0x00	0x105 番地 : 0x08
0x106 番地 : 空き領域	0x106 番地 : 空き領域
0x107 番地 : 空き領域	0x107 番地 : 空き領域

2.2.3 C プログラムとの結合

本 C++ コンパイラは、C++ プログラムの中から C の関数を呼び出したり、C のデータを参照するための機能を用意しています。この機能を使えば、既存の C プログラム資産や C ライブラリを有効に活用することができます。C の関数や大域変数を参照する場合は、リンケージ指定子 `extern "C"` を使用します。C++ プログラムから、C の関数を呼び出す例を以下に示します。

[例 1]

```
extern "C" int f(int); // 関数 f が C の関数であることを指定します。  
  
void g( )  
{  
    ...  
    a = f(1); // SH C コンパイラでコンパイルされた関数 f を呼び出します。  
    ...  
}
```

2.2.4 アセンブリプログラムとの結合

本 C++ コンパイラは、SH マイコン固有のレジスタへのアクセス等の機能を組み込み関数としてサポートしています（組み込み関数についての詳細は、「第2章 C++プログラミング 2.3.2 組み込み関数の使用方法」を参照してください）。しかし、MAC 命令による積和演算など C++ 言語で記述できない処理はアセンブリ言語で記述し、C++ 言語と結合する必要があります。

本章では、C++ プログラムとアセンブリプログラムの結合時に注意すべき以下の内容について述べます。

- ・ 外部名の相互参照方法
- ・ 関数呼び出しのインタフェース

2.2.4.1 外部名の相互参照方法

C++ プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。C++ コンパイラは、次のものを外部名として扱います。

- ・ 大域変数であって、かつ static 記憶クラスでないもの
- ・ extern 記憶クラスで宣言されている変数名
- ・ static 記憶クラスを指定されてなく、かつ非メンバ非インライン関数名
- ・ 非インラインメンバ関数名
- ・ 静的データメンバ名

外部名となる変数名をアセンブリプログラムで指定する場合は、C++ プログラム内での名前（最大 250 文字までが有効です）の先頭に下線（`_`）をつけたものになります。

例 1 センブリプログラムの外部名を C++ プログラムで参照する方法

- ・ アセンブリプログラムでは、「`.EXPORT`」制御命令を用いてシンボル名（先頭に下線（`_`）を付与）を外部定義宣言します。
- ・ C++ プログラムでは、シンボル名（先頭に下線（`_`）がない）を「`extern`」宣言します。

アセンブリプログラム（定義する側）

```
.EXPORT  _a, _b
.SECTION D,DATA,ALIGN=4
_a: .DATA.L 1
_b: .DATA.L 1
.END
```

C++ プログラム（参照する側）

```
extern int a,b;

f()
{
    a+=b;
}
```

例2 C++プログラムの外部名をアセンブリプログラムから参照する方法

- ・C++プログラムでは、シンボル名（先頭に下線（_）がない）を外部定義します。
- ・アセンブリプログラムでは、「.IMPORT」制御命令を用いてシンボル名（先頭に下線（_）を付与）を外部参照宣言します。

C++プログラム（定義する側）

```
int a;
```

アセンブリプログラム（参照する側）

```
.IMPORT    _a
.SECTION   P, CODE, ALIGN=2
MOV.L      A_a, R1
MOV.L      @R1, R0
ADD        #1, R0
RTS
MOV.L      R0, @R1
.ALIGN     4
A_a: .DATA.L    _a
.END
```

2.2.4.2 関数の呼び出し

C++プログラムとアセンブリプログラム間で相互に関数呼び出しを行うときに、アセンブリプログラム側で守るべき次の4つの規則について説明します。

- (1) スタックポインタに関する規則
- (2) スタックフレームの割り付け、解放に関する規則
- (3) レジスタに関する規則
- (4) 引数、リターン値の設定、参照に関する規則

(1) スタックポインタに関する規則

スタックポインタの指すアドレスよりも 44444444444444444444444444444444 下位（0番地の方向）のスタック領域に、有効なデータを格納してはいけません。スタックポインタより下位アドレスに格納されたデータは、割り込み処理で破壊される可能性があります。

(2) スタックフレームの割り付け、解放に関する規則

関数呼び出しが行われた時点 (JSR または BSR 命令の実行直後) では、スタックポインタは呼び出した関数側で使用したスタックの最下位アドレスを指しています。このスタックポインタの指している領域より上位アドレスのデータの割り付け、設定は呼び出す側の関数の役目です。

関数のリターン時は、呼び出された関数で確保した領域を解放してから、通常 RTS 命令を用いて呼び出した関数へ返ります。これより上位アドレスの領域 (リターン値アドレスおよび引数の領域) は、呼び出した側の関数で解放します。

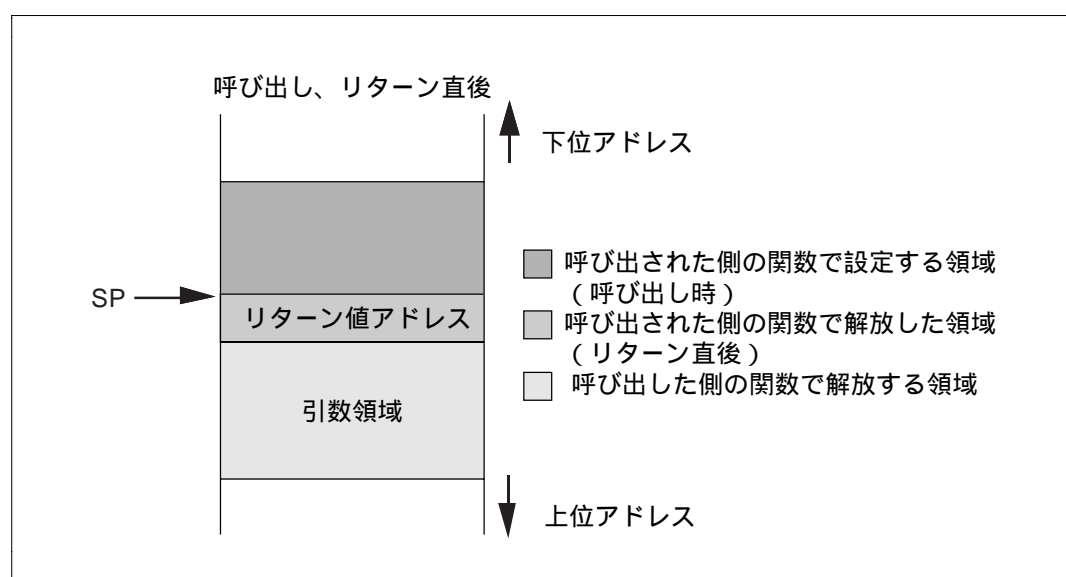


図 2.1 スタックフレームの割り付け、解放に関する規則

(3) レジスタに関する規則

関数呼び出し前後においてレジスタの値が同一であることを保証するレジスタと保証しないレジスタがあります。レジスタの保証規則を表 2.6 に示します。

表 2.6 関数呼び出し前後のレジスタ保証規則

No	項目	対象レジスタ	プログラミングにおける注意点
1	保証しないレジスタ	R0 ~ R7	関数呼び出し時に対象レジスタに有効な値があれば、呼び出し側で値を退避する。呼び出される側の関数では退避せずに使用できる。
2	保証するレジスタ	R8 ~ R15 MACH, MACL, PR	対象レジスタのうち関数内で使用するレジスタの値を退避し、リターン時に回復する。ただし、macsave = 0 オプション指定時は MACH、MACL は保証しないレジスタ。

以下、レジスタの保証規則の具体例を示します。

(a) アセンブリプログラムのサブルーチンを C++プログラムから呼び出す場合

アセンブリプログラム (呼び出される側)

```

        .EXPORT    _sub
        .SECTION   P, CODE, ALIGN=4
_sub:    MOV.L     R14, @-R15
        MOV.L     R13, @-R15
        ADD      #-8, R15

        :

        ADD      #8, R15
        MOV.L    @R15+, R13
        RTS
        MOV.L    @R15+, R14
        .END

```

関数内で使用するレジスタの退避

関数本体の処理
(R0~R7は保証しないレジスタであるので、退避せずに使用可能)

退避したレジスタの回復

C++プログラム (呼び出す側)

```

extern "C" void sub();
f()
{
    sub();
}

```

extern "c"を用いてプロトタイプ宣言を行ってください。

(b) C++プログラムの関数をアセンブリプログラムから呼び出す場合

C++プログラム（呼び出される側）

```
void sub()
{
    :
}
```

アセンブリプログラム（呼び出す側）

```
.IMPORT  _ _sub_ _Fv
.SECTION P, CODE, ALIGN=2

:

STS.L    PR, @-R15
MOV.L    R1, @(1, R15)
MOV      R3, R12
MOV.L    A_sub, R0
JSR      @R0
NOP
LDS.L    @R15+, PR

:

A_sub:   .DATA.L  _ _sub_ _ Fv
.END
```

コンパイラが関数宣言 / 定義から生成した外部名*を
「.IMPORT」制御命令で宣言

関数呼び出しする場合は、PR レジスタ（リターンア
ドレス格納レジスタ）を退避
レジスタR0～R7に有効な値があれば空きレジスタま
たはスタックに退避

関数「sub」の呼び出し

PRレジスタの復帰

【注】* 関数名から生成する外部名は、コンパイラが一定の規則で変換を行っています。コンパイラが生成した外部名を知る必要があるときは、コンパイラオプション `-code = asm` または、`-lis` オプションにてコンパイラが生成する外部名を参照してください。

(4) 引数とリターン値の設定、参照に関する規則

以下、引数とリターン値の設定、参照法について説明します。解説では、まず引数とリターン値に対する一般的な規則について述べたあと、引数の割り付け方とリターン値の設定場所について述べます。

(a) 引数とリターン値に対する一般的な規則

(i) 引数の渡し方

引数の値を、必ずレジスタまたはスタック上の引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出した側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出した側の処理は直接には影響を受けません。

(ii) 型変換の規則

引数を渡す場合、またはリターン値を返す場合、自動的に型変換を行う場合があります。以下、この型変換の規則について説明します。

(ア) 型の宣言された引数の型変換

プロトタイプ宣言によって型が宣言されている引数は、宣言された型に変換します。

(イ) リターン値の型変換

リターン値は、その関数の返す型に変換します。

[例]

(1)

```
long f( );  
long f( )  
{   float x;  
    return x; ← プロトタイプ宣言にしたがってリターン値は long 型に型変換  
}
```

されます。

(2)

```
void p( int, ... );  
f( )  
{   char c;  
    p(1.0, c);  
}
```

→ c は、対応する引数の型宣言がないので、int 型に変換されます。
→ 1.0 は、対応する引数の型が int 型なので、int 型に変換されます。

(b) 引数の割り付け領域

引数は、レジスタに割り付ける場合と、レジスタに割り付けられないときスタック上の引数領域に割り付ける場合があります。引数の割り付け領域を図 2.2 に、引数割り付け領域の一般規則を表 2.7 にそれぞれ示します。

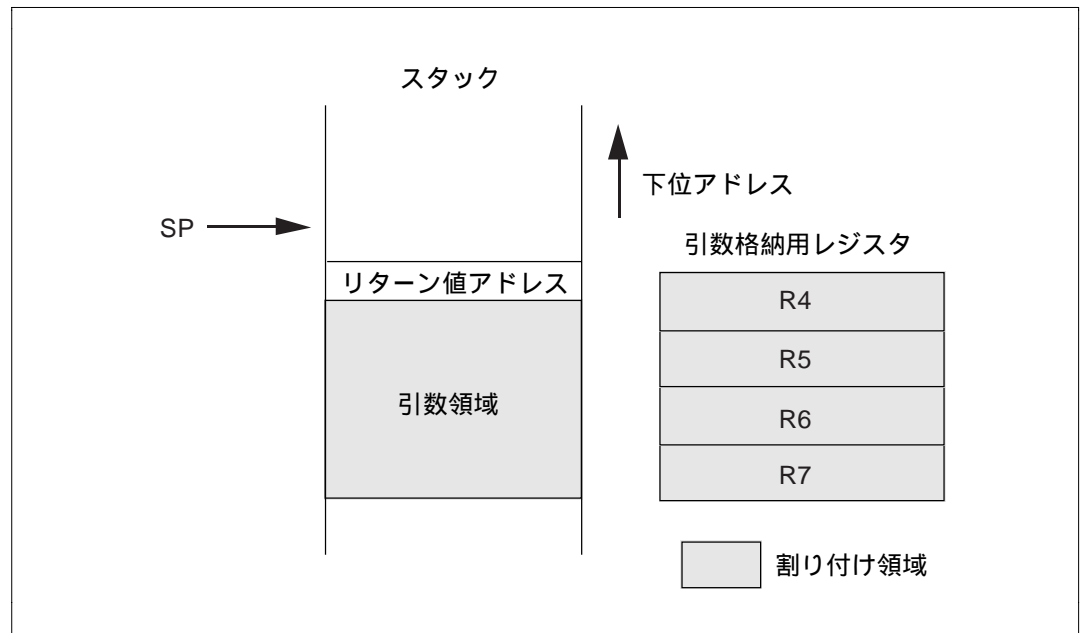


図 2.2 引数の割り付け領域

表 2.7 引数割り付け領域の一般規則

割り付け規則		
レジスタで渡される引数		スタックで渡される引数
引数格納用レジスタ	対象の型	
R4 ~ R7	char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, ポインタ、リファレンス、データメンバへのポインタ	(1) 引数の型がレジスタ渡しの対象の型以外のもの (2) プロトタイプ宣言により可変個の引数を持つ関数として宣言しているもの* (3) 他の引数がすでに R4 ~ R7 に割り付けられている場合。

【注】 * プロトタイプ宣言により可変個の引数をもつ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタックに割り付けます。

【例】

```
int f2(int, int, int, int, ...);
:
f2(a, b, c, x, y, z);    x、y、zはスタックに割り付けます。
```

(c) 引数の割り付け

(i) 引数格納用レジスタへの割り付け

引数格納用レジスタには、ソースプログラムの宣言順に番号の小さいレジスタから割り付けます。引数格納用レジスタの割り付け例を図 2.3 に示します。しかし、非静的メンバ関数は R4 にオブジェクトへのポインタ (this ポインタ) が割り付くので、関数の実引数は R5 から順に割り付けます。

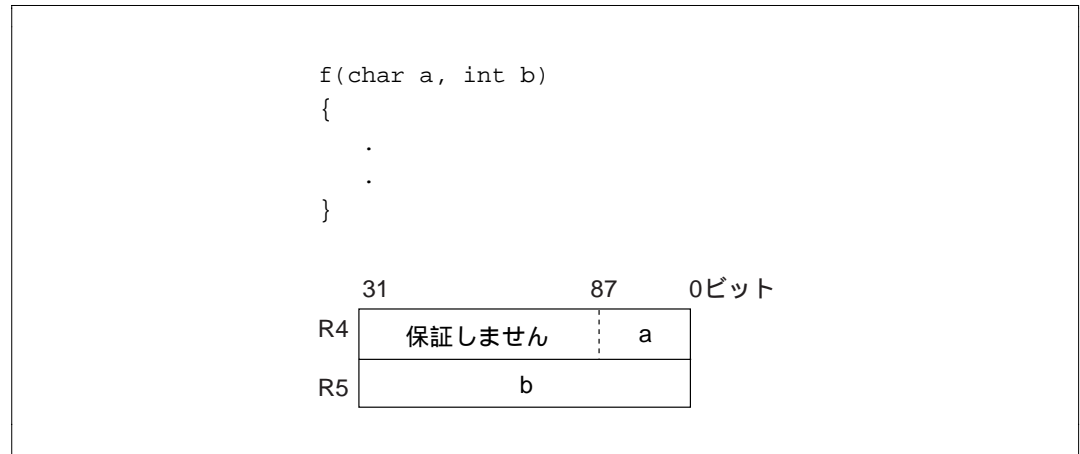


図 2.3 引数格納用レジスタの割り付け例

(ii) スタック上の引数領域への割り付け

スタック上の引数領域には、ソースプログラム上で宣言した順に下位アドレスから割り付けます。

【注】 クラス (構造体、共用体) 型引数に関する注意

クラス型の引数を設定する場合は、その型の本来の境界調整にかかわらず 4 バイト境界に割り付けられ、しかもその領域として 4 の倍数バイトの領域が使用されます。これは、SH のスタックポインタが 4 バイト単位で変化するためです。

「付録 B . 引数割り付けの具体例」に、引数割り付けの具体例がありますので、あわせて参照してください。

(d) リターン値の設定場所

関数のリターン値の型によっては、リターン値をレジスタに設定する場合とメモリに設定場合があります。リターン値の型と設定場所の関係は表 2.8 を参照してください。

関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスをリターン値アドレスに設定してから関数を呼び出します (図 2.4 参照)。関数のリターン値が void 型の場合、リターン値を設定しません。

表 2.8 リターン値の型と設定場所

No	リターン値の型	リターン値の設定場所
1	char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long float ,ポインタ,リファレンス, データメンバへのポインタ	R0 : 32 ビット (char, signed char, unsigned char の上位 3 バイト、short, unsigned short の上位 2 バイトの内容は保証しません)
2	double, long double クラス型、関数メンバへのポインタ	リターン値設定領域 (メモリ)

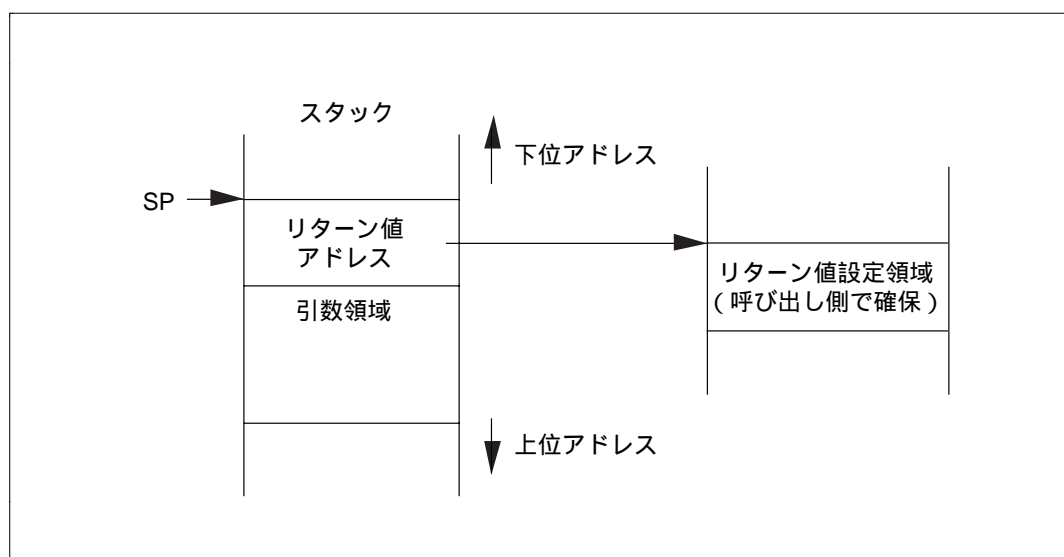


図 2.4 リターン値をメモリに設定する場合のリターン値の設定領域

2.3 拡張機能

本節では、本 C++ コンパイラの以下の拡張機能の使用方法について説明します。

割り込み関数
 組み込み関数
 セクション切り替え機能
 単精度浮動小数点ライブラリ
 文字列内の日本語記述
 関数のインライン展開
 アセンブラ埋め込みインライン展開
 2 バイトアドレス変数指定
 GBR ベース変数指定
 レジスタ退避、回復制御

2.3.1 割り込み関数の使用方法

本 C++ コンパイラでは、プリプロセッサ制御文 (#pragma) を用いて、外部 (ハードウェア) 割り込み関数を C++ プログラムで記述することができます。以下、割り込み関数の作成方法を説明します。SH3 では割り込み時の動作が SH1、SH2 の場合と異なりますので、割り込みハンドラが必要になります。

(1) 記述方法

#pragma interrupt (関数名 [(割り込み仕様)] [, 関数名 [(割り込み仕様)])

割り込み仕様の一覧を表 2.9 に示します。

表 2.9 割り込み仕様一覧

No.	項目	形式	オプション	指定内容
1	スタック 切り替え指定	sp =	<変数> &<変数> <定数>	新しいスタックのアドレスを変数または定数で指定 <変数> : 変数 (オブジェクト型) の値 &<変数> : 変数 (ポインタ型) のアドレス <定数> : 定数値
2	トラップ命令 リターン指定	tn =	<定数>	終了を TRAPA 命令で指定 <定数> : 定数値 (トラップベクタ番号)

(2) 説明

#pragma interrupt を用いて割り込み関数となる関数を宣言します。

#pragma interrupt を用いて宣言した関数は、関数の処理の前後で全レジスタを保証（関数入口／出口で関数内で使用する全レジスタを退避・回復）し、通常 RTE 命令でリターンし、トラップ命令リターンを指定した場合は TRAPA 命令でリターンします。割り込み仕様を指定しない場合は単純な割り込み関数として処理します。また、スタック切り替え指定とトラップ命令リターン指定は重複して指定できます。

[例]

```
extern int STK[100];

int *ptr = STK + 100;

#pragma interrupt( f(sp = ptr, tn = 10))
```

[説明]

(a) スタック切り替え指定

ptr を割り込み関数「f」で使用するスタックポインタとして設定します。

(b) トラップ命令リターン指定

割り込み関数終了時に TRAPA #H'10 でトラップ例外処理を開始します。トラップ例外処理開始時の SP は下図のようになっています。トラップルーチンの側で RTE 命令を使用して PC、SR（ステータスレジスタ）を回復し、割り込み関数から復帰してください。

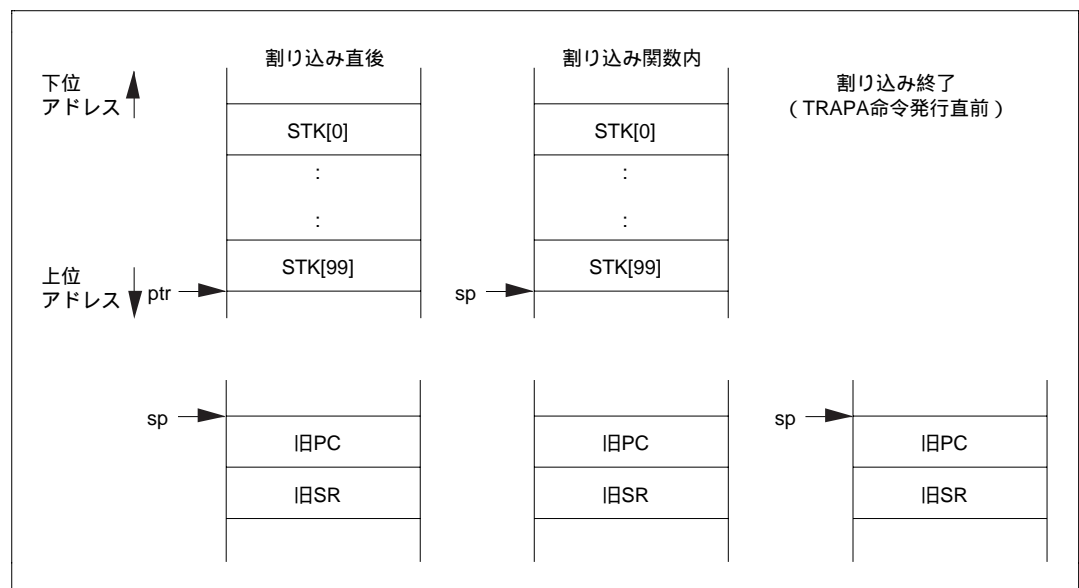


図 2.5 割り込み関数によるスタック使用例

(3) 使用上の注意事項

- (i) 割り込み関数の定義に対して指定できる関数は、非メンバ関数かつ多重定義されていない関数で、非静的な関数だけです。また、static を指定しても extern として処理します。

また、関数の返すデータ型は void のみです。return 文のリターン値を指定することはできません。指定があった場合はエラーを出力します。

[例]

```
#pragma interrupt (f1 (sp = 100), f2)
void f1(){...}..... (a)
int f2(){...}..... (b)
```

[説明]

(a) は正しい宣言になります。

(b) は関数の返すデータ型が void ではないので誤った宣言です。エラーを出力します。

- (ii) 割り込み関数として宣言した関数をプログラムの中で呼び出すことはできません。呼び出しがあった場合はエラーを出力します。ただし、割り込み関数として定義した関数を割り込み関数の宣言のないプログラム内で呼び出した場合は、エラーは出力しません。この場合実行時の動作は保証しません。

[例]

割り込み関数宣言のある場合

```
#pragma interrupt (f1)
void f1(void){...}
int f2(){ f1(); }..... (a)
```

[説明]

関数「f1」は割り込み関数として宣言しているのでプログラム中で呼び出すことはできません。(a) に対してエラーを出力します。

割り込み関数宣言がない場合

```
int f1();
int f2(){ f1(); }..... (b)
```

[説明]

関数「f1」は割り込み関数としての宣言がないので非割り込み関数 int f1 (); としてオブジェクトを生成します。関数「f1」が別コンパイル単位で割り込み関数として宣言された場合、実行時に動作は不定になります。

2.3.2 組み込み関数の使用方法

本 C++ コンパイラでは、SH 固有の命令に対応する組み込み関数を提供しています。

以下、組み込み関数の使用方法を説明します。

(1) 組み込み関数の機能

組み込み関数は以下の機能を記述することができます。

- (i) ステータスレジスタの設定、参照
- (ii) ベクタベースレジスタの設定、参照
- (iii) グローバルベースレジスタを利用した I/O 機能
- (iv) C++ 言語で使用するレジスタ資源と競合しないシステム命令

(2) 説明

組み込み関数を使用する場合は、必ず `<machine.h>` または `<umachine.h>` や `<smachine.h>` をインクルードしてください。

(3) 組み込み関数仕様

組み込み関数の一覧を表 2.10 に示します。

表 2.10 組み込み関数一覧

No.	項目	機能	仕様	説明
1	ステータス レジスタ	ステータス レジスタの設定	<code>void set_cr (int cr)</code>	ステータスレジスタに <code>cr</code> (32 ビット) を設定
2	(SR)	ステータス レジスタの参照	<code>int get_cr (void)</code>	ステータスレジスタを参照
3		割り込みマスクの 設定	<code>void set_imask (int mask)</code>	割り込みマスク (4 ビット) に <code>mask</code> (4 ビット) を設定
4		割り込みマスクの 参照	<code>int get_imask (void)</code>	割り込みマスク (4 ビット) を参照
5	ベクタベース レジスタ	ベクタベース レジスタの設定	<code>void set_vbr (void **base)</code>	VBR に <code>**base</code> (32 ビット) を 設定
6	(VBR)	ベクタベース レジスタの参照	<code>void **get_vbr (void)</code>	VBR の参照

No.	項目	機能	仕様	説明
7	グローバル ベース	GBR の設定	void set_gbr (void *base)	GBR に*base (32 ビット) を 設定
8	レジスタ	GBR の参照	void *get_gbr (void)	GBR を参照
9	(GBR)	GBR ベースの バイト参照	unsigned char gbr_read_byte (int offset)	GBR 相対 offset のバイトデー タ (8 ビット) を参照
10		GBR ベースの ワード参照	unsigned short gbr_read_word (int offset)	GBR 相対 offset のワードデー タ (16 ビット) を参照
11		GBR ベースの ロングワード参照	unsigned long gbr_read_long (int offset)	GBR 相対 offset のロングワー ドデータ (32 ビット) を参照
12		GBR ベースの バイト設定	void gbr_write_byte (int offset, unsigned char data)	GBR 相対 offset の data (8 ビッ ト) を設定
13		GBR ベースの ワード設定	void gbr_write_word (int offset, unsigned short data)	GBR 相対 offset の data (16 ビット) を設定
14		GBR ベースの ロングワード設定	void gbr_write_long (int offset, unsigned long data)	GBR 相対 offset の data (32 ビット) を設定
15		GBR ベースの バイト AND	void gbr_and_byte (int offset, unsigned char mask)	GBR 相対のバイトデータと mask の AND をとり、offset に設定
16		GBR ベースの バイト OR	void gbr_or_byte (int offset, unsigned char mask)	GBR 相対のバイトデータと mask の OR をとり、offset に 設定
17		GBR ベースの バイト XOR	void gbr_xor_byte (int offset, unsigned char mask)	GBR 相対のバイトデータと mask の XOR をとり、offset に設定
18		GBR ベースの バイト TEST	int gbr_tst_byte (int offset, unsigned char mask)	GBR 相対のバイトデータ mask と AND をとり、その値 を 0 と判定し結果を T ビット にセット

No.	項目	機能	仕様	説明
19	その他の	SLEEP 命令	void sleep (void)	SLEEP 命令に展開
20	特殊命令	TAS 命令	int tas (char *addr)	TAS.B @addr に展開
21		TRAPA 命令	int trapa (int trap_no)	TRAPA #trap_no に展開
22		OS システム コールの実現	int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4) trap_no : トラップ番号 code : 機種コード para1 ~ 4 : パラメタ (0 ~ 4 個の可変) type1 ~ 4 : パラメタの型は、汎整数型 または ポインタ型	HI-SH7 (Hitachi Industrial Realtime Operating System SH7000 Series) をはじめ、 各種 OS のシステムコールを 可能にする。trapa_svc を実行 すると、R0 に code、R4 ~ R7 に para1 ~ para4 を設定し、 TRAPA #trap_no 命令を実行する。
23		PREF 命令	void prefetch (void *p) 注意 : prefetch はコンパイラ オプション cpu=sh3 指 定時のみ使用可能	prefetch を実行すると、p の指 す領域 (16 バイト、ただし、 領域は (int) p&0xfffff0) か らの 16 バイトのデータを キャッシュに読み込む。 プログラムの論理的な動作に は全く影響を与えません。

No.	項目	機能	仕様	説明
24	積和演算	MAC.W 命令	<p>int macw (short *ptr1, short *ptr2, unsigned int count)</p> <p>int macwl (short *ptr1, short *ptr2, unsigned int count, unsigned int mask)</p> <p>ptr1 : 積和演算するデータの先頭アドレス</p> <p>ptr2 : 積和演算するデータの先頭アドレス</p> <p>count : 積和演算を実行する回数</p> <p>mask : リングバッファ対応のアドレスマスク</p>	<p>積和演算組み込み関数は、二つのデータテーブルの内容の積和を求める。</p> <p>例</p> <pre>short tbl1[] = {a1,a2,a3,a4}; short tbl2[] = {b1,b2,b3,b4};</pre> <p>この時 macw (tbl1, tbl2, 3) は $a1*b1 + a2*b2 + a3*b3$ を求める。</p> <p>また、リングバッファ機能を用いて、tbl2 を繰り返して演算することができる。繰り返し回数は 2ⁿ 回。</p> <p>例</p> <p>データサイズが 2 バイトでリングバッファマスクを 4 バイト (0xffffffffb または ~0x4) とする</p>
25		MAC.L 命令	<p>int macL (int *ptr1, int *ptr2, unsigned int count)</p> <p>int macLl (int *ptr1, int *ptr2, unsigned int count, unsigned int mask)</p> <p>パラメタの仕様は No.24 と同様。</p> <p>注意 : macL, macLl はコンパイラオプション cpu = sh2, sh3 指定時のみ使用可能。</p>	<p>と、 macwl (tbl1, tbl2, 4, 0xffffffffb) は $a1*b1 + a2*b2 + a3*b1 + a4*b2$ を求める。</p>

(4) 注意事項

「表 2.10 組み込み関数一覧」で使した offset (No.15 ~ 18 を除く)、mask (No.3 を除く) は定数でなければなりません。

また、offset 指定可能範囲は、アクセスサイズがバイトのとき+255 バイト、ワードのとき+510 バイト、ロングワードのとき+1020 バイトまでです。

GBR (グローバルベースレジスタ) 相対のバイト論理演算 (AND、OR、XOR、TEST) で指定できる mask は 0 ~ +255 です。

GBR はコントロールレジスタですので、本 C++ コンパイラでは関数ごとに内容を保証していません。

GBR の設定を変えるときには注意が必要です。

(5) 使用例

```
#include <machine.h>
#define CDATE1 0
#define CDATE2 1
#define CDATE3 2
#define SDATE1 4
#define IDATA1 8
#define IDATA2 12

struct {
    char  cdata1;           //offset 0
    char  cdata2;           //offset 1
    char  cdata3;           //offset 2
    shor  sdata1;           //offset 4
    int   idata1;           //offset 8
    int   idata2;           //offset 12
}table;
void f();

void f()
{
    set_gbr(&table);        //GBR に table の先頭アドレス
                           //を設定
    gbr_write_byte(CDATE2,10); // table.cdata2 に 10 を設定
    gbr_write_long(IDATA2,100); // table.idata2 に 100 を設定
    :
    if(gbr_read_byte(CDATE2)!=10) // table.cdata2 の値を参照
        gbr_and_byte(CDATE2,10); // table.cdata2の値と10のAND
    //をとって table.cdata2 に設定
    gbr_or_byte(CDATE2,0x0F); // table.cdata2の値と0x0fのOR
    : //をとって table.cdata2 に設定
    sleep(); // sleep 命令に展開
}
```

組み込み関数の有効な使用法

- (i) 頻繁にアクセスするオブジェクトをメモリに割り付け、そのオブジェクトの先頭アドレスを GBR に設定する。
- (ii) 論理演算を多用するバイトデータをできるだけクラスの先頭から 128 バイトまでに宣言する。これにより、クラスアクセスに必要な先頭アドレスロードと、論理演算に必要なメモリロード、ストアに対する命令が削減できます。

(6) 積和演算組み込み関数使用上の注意事項

積和演算組み込み関数はパラメタのチェックを行いません。パラメタは次の事に注意してください。

- (i) ptr1、ptr2 の指すテーブルは、それぞれ 2 バイト、4 バイトで境界整合されていなければなりません。
- (ii) macwl、macll の ptr2 の指すテーブルはリングバッファマスク × 2 のサイズで境界整合されていなければなりません。

(7) <machine.h>の分割

SH3 の実行モードに対応し <machine.h> の内容を以下のように分割しました。

- (a) <machine.h> 組み込み関数全体
- (b) <smachine.h> 特権モードでのみ使用可能な組み込み関数
- (c) <umachine.h> (b) 以外の組み込み関数

2.3.3 セクション切り替え機能

#pragma section を用いて、C++ プログラムの中でコンパイラの出力するセクション名を切り替えることができます。関数単位、変数単位で割り付けるアドレスを指定するためには、従来はファイル分割しなければならませんでした。本機能を用いることにより、関数単位、変数単位のファイル分割が不要になります。以下、詳細仕様について説明します。

(1) 記述方法

```
#pragma section 名前 | 数値
    <ソースプログラム>
#pragma section
```

(2) 説明

"#pragma section 名前"または"#pragma section 数値"を用いて、セクション名を指定します。ソースプログラム中の宣言位置以降のセクションが、"P セクション名+名前 (数値)"、"D セクション名+名前 (数値)"、"C セクション名+名前 (数値)"、"B セクション名+名前 (数値)"になります。

"#pragma section"が宣言されると、以降はデフォルトのセクション名になります。

(3) 使用上の注意事項

- (i) #pragma section は関数定義の外に指定してください。
- (ii) 1 ファイルで宣言できるセクション名は最大 64 個です。

(4) セクション切り替え機能の使用例

[例]

```
#pragma section abc
int a;                // a は、セクション Babc に割り付きます。
extern const int c = 1; // c は、セクション Cabc に割り付きます。
f(){                  // f は、セクション Pabc に割り付きます。
    a = c;
}

#pragma section
int b;                // b は、セクション B に割り付きます。
g(){                  // g は、セクション P に割り付きます。
    b = c;
}
```

上記例において、コンパイルオプション section = P = PROG が指定された場合、f はセクション PROGabc、g はセクション PROG にそれぞれ割り付きます。

2.3.4 単精度浮動小数点ライブラリ

ANSI 標準浮動小数点ライブラリ (math.h) のほかに、単精度の浮動小数点ライブラリを使用することができます。本ライブラリは表 2.11 に示す関数群から構成されています。

(1) 記述方法

各関数名は倍精度の ANSI 標準ライブラリの関数名の末尾に「f」を付加したものになっています。パラメタおよびリターンの型が、double 型または double 型へのポインタ型である場合、それぞれ、float 型、float 型へのポインタ型となります。それ以外の仕様は ANSI 標準 C ライブラリと同じです。

(2) 使用上の注意事項

本ライブラリを使用する場合は、必ず `#include <mathf.h>` と `#include <math.h>` を宣言してください。

表 2.11 単精度浮動小数点ライブラリ関数一覧

関数名	説明
<code>float acosf (float x)</code>	逆余弦 $\cos^{-1} x$
<code>float asinf (float x)</code>	逆正弦 $\sin^{-1} x$
<code>float atanf (float x)</code>	逆正接 $\tan^{-1} x$
<code>float atan2f (float y, float x)</code>	除算した結果の値の逆正接 $\tan^{-1} (y/x)$
<code>float cosf (float x)</code>	余弦 $\cos x$
<code>float sinf (float x)</code>	正弦 $\sin x$
<code>float tanf (float x)</code>	正接 $\tan x$
<code>float coshf (float x)</code>	双曲線余弦 $\cosh x$
<code>float sinh (float x)</code>	双曲線正弦 $\sinh x$
<code>float tanhf (float x)</code>	双曲線正接 $\tanh x$
<code>float expf (float x)</code>	指数関数 e^x
<code>float frexpf (float x, int *p)</code>	(0.5, 1.0) の値と 2 のべき乗の積に分解 $\text{result} = \text{frexp} (x, p)$ とすると $x = 2^p \times \text{result}$ (0.5 \leq result < 1.0)
<code>float ldexpf (float x, int i)</code>	2 のべき乗との乗算 $x \times 2^i$
<code>float logf (float x)</code>	自然対数 $\log x$
<code>float log10f (float x)</code>	常用対数 $\log_{10} x$ (底を 10 とする)
<code>float modff (float x, float *p)</code>	$\text{result} = \text{modff} (x, y)$ とすると x を整数部分 $*p$ と小数部分 result に分解
<code>float powf (float x, float y)</code>	べき乗 x^y
<code>float sqrtf (float x)</code>	正の平方根 \sqrt{x}
<code>float ceilf (float x)</code>	x の小数点以下を切り上げて結果とします
<code>float fabsf (float x)</code>	絶対値 $ x $
<code>float floorf (float x)</code>	x の小数点以下を切り捨てて結果とします
<code>float fmodf (float x, float y)</code>	除算した余り $\text{result} = \text{fmodf} (x, y)$ 、 q を商 (整数) とすると $x = q \times y + \text{result}$

2.3.5 文字列内の日本語記述

文字列の中に日本語が記述できます。euc、または sjis オプションで文字コードを選択できます。本オプション省略時の設定は、ホストごとのデフォルトに従います（表 2.12 参照）。

表 2.12 日本語コードのデフォルト設定

ホスト	デフォルト
SPARC	EUC
HP9000/700	シフト JIS
NEWS	環境変数"LANG"の示す文字コード
PC-9801	シフト JIS
IBM-PC	シフト JIS

オブジェクトプログラム内の文字コードは、ソースプログラムの文字コードと同一になります。

文字定数には日本語は指定できません。

2.3.6 関数のインライン展開

コンパイル時にインライン展開する関数名を指定します。

（１）記述方法

```
#pragma inline （関数名,...）
```

（２）説明

#pragma inline で指定した関数名の関数と関数指定子 inline を指定した関数は、その関数を呼び出したところにインライン展開されます。

ただし、以下の場合はインライン展開しません。

- ・ #pragma inline 指定より前に関数の定義がある
- ・ 可変パラメタを持つ関数である
- ・ 関数内でパラメタのアドレスを参照している
- ・ 展開対象関数のアドレスを介して呼び出しを行っている。

(3) 使用上の注意事項

- (a) `#pragma inline` は、関数の本体の定義の前に指定してください。
- (b) `#pragma inline` で指定した関数に対しても外部定義を生成します。(プリプロセッサ展開後の)各ソースプログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に `static` を指定してください。`static` を指定した場合は、外部定義を生成しません。
- (c) `#pragma inline` は、多重定義関数、メンバ関数を指定することはできません。指定できる関数は、多重定義されていない非メンバ関数です。多重定義関数、メンバ関数を `inline` 展開する場合は、関数指定子 `inline` を指定してください。

(4) 使用例

<u>ソースプログラム</u>	<u>展開イメージ</u>
<pre>#pragma inline(func) int func (int a, int b) { return (a+b)/2; } int x; main() { ::x = func(10,20); }</pre>	<pre>int x; main() { int func_result; { int a_1 = 10, b_1 = 20; func_result = (a_1+b_1)/2; } ::x = func_result; }</pre>

2.3.7 アセンブラ埋め込みインライン展開

C++ソースファイル内でアセンブリ言語で記述した関数をインライン展開します。

(1) 記述方法

```
#pragma inline_asm (関数名,...)
```

(2) 説明

アセンブラ埋め込みインライン関数のパラメタは、通常関数呼び出しと同様にレジスタ、あるいはスタックに設定されますので、`inline_asm` 関数から参照することができます。アセンブラ埋め込みインライン関数のリターン値は `R0` に設定してください。

(3) 使用上の注意事項

- (a) `#pragma inline_asm` は、関数の本体の定義の前に指定してください。
- (b) `#pragma inline_asm` で指定した関数に対しても外部定義を生成します。(プリプロセッサ展開後の) 各ソースプログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に `static` を指定してください。`static` を指定した場合は、外部定義を生成しません。
- (c) アセンブラ埋め込みインライン関数中でラベルを使用する場合、必ずローカルラベルを使用してください。
- (d) アセンブラ埋め込みインライン関数中で R8 から R15 のレジスタを使用する場合は、アセンブラ埋め込みインライン関数の先頭と最後でこれらレジスタの退避・回復が必要です。
- (e) アセンブラ埋め込みインライン関数の最後に RTS を記述しないでください。
- (f) 本機能を使用する際は、オブジェクト形式指定オプション `code = asmcode` を用いてコンパイルしてください。

ソースプログラム

```
#pragma inline_asm(rotl)
int rotl (int a)
{
    ROTL R4
    MOV  R4,R0
}
int x;
main()
{
    ::x = 0x55555555;
    ::x = rotl(::x);
}
```

出力結果 (一部)

```
:
-- _main_ _Fv ;function main
;frame size = 8
MOV.L R3,@ - R15
STS.L PR, @ - R15
MOV.L L211,R14
MOV.L L211+4,R3
MOV.L R3,@R14
MOV R3,R4
BRA L210
NOP
L211:
.DATA.L _x
.DATA.L H'55555555
L210:
ROTL R4
MOV R4,R0
.ALIGN 4
MOV.L R4,@R14
LDS.L @R15+,PR
RTS
MOV.L @R15+,R14
.SECTION B,DATA,ALIGN=4
_x: ;static: x
.RES.L END
.END
```

2.3.8 2 バイトアドレス変数の指定

H'00000000 ~ H'0007FFF 番地および H'FFF8000 ~ H'FFFFFFF 番地に配置した変数に対して、アドレスの表現を 2 バイトで済ますことができます。

(1) 記述方法

```
#pragma abs16 ( 識別子,... )
```

(2) 説明

識別子で指定した変数、または関数のアドレスを 2 バイトの値として扱います。これによってプログラムサイズを削減することができます。

(3) 使用上の注意事項

- (a) #pragma abs16 は、自動オブジェクトには指定できません。
- (b) #pragma abs16 で宣言された変数は、必ず H'00000000 ~ H'0007FFF 番地ないしは、H'FFF8000 ~ H'FFFFFFF 番地に配置してください。

2.3.9 GBR ベース変数の指定

変数を GBR レジスタからのオフセットでアクセスすることを指定します。

(1) 記述方法

```
#pragma gbr_base ( 変数名,... )  
#pragma gbr_base1 ( 変数名,... )
```

(2) 説明

#pragma gbr_base で指定した変数は、セクション \$G0 に割り付けられます。#pragma gbr_base1 で指定した変数は、セクション \$G1 に割り付けられます。

#pragma gbr_base は、変数が GBR レジスタの指すアドレスからオフセット 0 ~ 127 バイトにあることを指定します。

#pragma gbr_base1 は、#pragma gbr_base でアクセス可能でない範囲 (GBR レジスタの指すアドレスからオフセット 128 バイト以上) の変数に対して指定できます。GBR レジスタの指すアドレスからのオフセットが、char 型、signed char 型、unsigned char 型の場合は最大 255 バイト、short 型、unsigned short 型の場合は、最大 510 バイト、int 型、unsigned 型、long 型、unsigned long 型、float 型、double 型の場合は最大 1020 バイトであることを指定します。

コンパイラは、これらの指定に基づき、変数の参照、設定に対して、最適な GBR 相対アドレッシングでオブジェクトプログラムを生成します。また、セクション \$G0 内の char 型、signed char 型、unsigned 型のデータに対して、GBR 間接アドレッシングで最適なビッ

ト命令を生成します。

(3) 使用上の注意事項

- (a) セクション\$G0のリンク後のサイズの合計が128バイトを超えた場合は動作を保証しません。また、セクション\$G1内に、上記の`#pragma gbr_base1`の制約で各データ型に示した以上のオフセットを持つデータがある場合、動作を保証しません。
- (b) セクション\$G1は、リンク時にセクション\$G0の128バイト後に必ず配置してください。
- (c) 本機能を使用する場合は、プログラム実行開始時に、GBRレジスタにセクション\$G0の先頭を設定してください。

2.3.10 レジスタ退避・回復の制御

関数のレジスタ退避・回復方法を変更します。

(1) 記述方法

```
#pragma noregsave ( 関数名,... )
#pragma noregalloc ( 関数名,... )
#pragma regsave ( 関数名,... )
```

(2) 説明

- (a) `#pragma noregsave` で指定された関数は、関数の出/入口で保証するレジスタ(表2.6参照)の退避・回復を行いません。
- (b) `#pragma noregalloc` で指定された関数は、関数の出/入口で保証するレジスタの退避・回復を行いません。また、関数呼び出しを越えてR8~R14を割り付けないオブジェクトを生成します。
- (c) `#pragma regsave` で指定された関数は、関数の出/入口で保証するレジスタの退避・回復を行います。また関数呼び出しを越えてR8~R14を割り付けないオブジェクトを生成します。
- (d) `#pragma regsave` と`#pragma noregalloc` は同一関数に対して重複指定できます。このとき、関数の出/入口で保証するレジスタR8~R14を全て退避・回復し、関数呼び出しがあるとき、レジスタ(R8~R14)を割り付けないオブジェクトを生成します。

`#pragma noregsave` が指定された関数は、下記の条件で 사용할 ことができます。

- (i) 他の関数から呼び出されることなく、最初に起動する関数として使用する。
- (ii) `#pragma regsave` を指定した関数から呼び出す。
- (iii) `#pragma regsave` を指定した関数から、さらに`#pragma noregalloc` を介して呼び出す。

(3) 使用上の注意事項

上記以外の方法で `#pragma noregsave` を指定した関数を呼び出した場合の結果は保証されませんので注意が必要です。

(4) 使用例

```
#pragma noregsave(f)
#pragma noregalloc(g)
#pragma regsave(h)
h()
{
    g();
    f(); // #pragma regsave 関数(h)から#pragma noregsave 関数(f)の直後の呼び出し
}

g()
{
    f(); // #pragma regsave 関数(h)から#pragma noregsave 関数(f)の
        // #pragma noregalloc 関数(g)を介した呼び出し
}

f()
{
}
```


2.4 プログラム作成上の注意事項

本章では、C++コンパイラにおけるコーディング上の注意事項と、コンパイルからデバッグまでのプログラム開発上のトラブル対処方法を述べます。

2.4.1 コーディング上の注意事項

(1) C++言語で評価順序を規定していない式

C++言語で評価順序を規定していない式で、評価順序で結果が変わるようなコーディングをした場合、その動作は保証しません。

[例]

```
a[i]=a[++i];
```

代入式の右辺を先に評価するか後に評価するかで、左辺の値が変わります。

```
sub(++i, i);
```

関数の第1引数を先に評価するか後に評価するかで第2引数の値が変わります。

(2) オーバフロー演算、ゼロ除算

オーバフロー演算や浮動小数点のゼロ除算があっても、エラーメッセージを出力しません。ただし、1つの定数または定数どうしの演算でのオーバフロー演算や、定数どうしの演算または整数型のゼロ除算があれば、コンパイル時にエラーメッセージを出力します。

[例]

```
main()
{
    int ia;
    int ib;
    float fa;
    float fb;

    ib=32767;
    fb=3.4e+38f;

    // 定数または定数どうしの演算時はオーバーフロー、ゼロ除算に対する
    // コンパイルエラーメッセージを出力します

    ia=999999999999;    // (W) 定数のオーバーフローを検出します
    fa=3.5e+40f;        // (W) 浮動小数点演算のオーバーフローを検出します
    ia=1/0;             // (E) ゼロ除算を検出します
    fa=1.0/0.0;         // (W) 浮動小数点のゼロ除算を検出します

    // 実行時のオーバーフローに対するエラーメッセージは出力しません

    ib=ib+32767;        // 演算結果のオーバーフローを無視します
    fb=fb+3.4e+38f;     // 浮動小数点演算結果のオーバーフローを無視します
}
```

(3) const 型変数への書き込み

const 型の変数を宣言していても、型変換で const 型でない型に変換して代入した場合や、分割コンパイルしたプログラムの中で、型を統一して扱っていない場合は、const 型変数への書き込みを C++コンパイラでチェックできませんので、注意が必要です。

[例]

```
const char *p;    // ライブラリ関数 strcat の第1引数は char 型への
:                // ポインタ型なので、引数の指す領域が書き換わる
strcat(p, "abc"); // ことがあります。
```

ファイル1

```
const int i;
```

ファイル2

```
extern int i;    // 変数 i は、ファイル2では const 型で宣言していま
:               // せんでファイル2の中で書き込んでもエラーにな
i=10;            // りません。
```

(4) 数学関数ライブラリの精度について

$\text{acos}(x)$ 、 $\text{asin}(x)$ 関数では $x = 1$ で誤差が大きくなりますので注意が必要です。

誤差範囲は以下のとおりです。

$\text{acos}(1.0 - \varepsilon)$ における絶対誤差 倍精度 2^{-39} ($\varepsilon = 2^{-33}$)

単精度 2^{-21} ($\varepsilon = 2^{-19}$)

$\text{asin}(1.0 - \varepsilon)$ における絶対誤差 倍精度 2^{-39} ($\varepsilon = 2^{-28}$)

単精度 2^{-21} ($\varepsilon = 2^{-16}$)

2.4.2 プログラム開発上のトラブル対処方法

C++プログラムの作成からデバッグまでのプログラム開発上で、トラブルが発生したときの対処方法を表 2.13 に示します。

表 2.13 トラブル発生時の対処方法

No	現象	確認内容	対処方法	参照
1	リンク時にエラーNo. 314 cannot found section が出力される	リンケージエディタの start オプションにおいて、コンパイラ出力のセクション名を大文字で指定しているか。	正しいセクション名を指定してください。	「第 2 章 C++プログラミング 2.2.1 オブジェクトプログラムの構造」
2	リンク時にエラーNo. 105 undifined external symbol が出力される。	C++プログラムとアセンブリプログラム間で変数を相互参照している場合、アセンブリプログラム内で下線を付加しているか。	正しい変数名で参照してください。	「第 2 章 C++プログラミング 2.2.3.1 外部名の相互参照方法」
		C++プログラムで C ライブラリ関数を使用していないか。	リンク時に入力ライブラリとして標準ライブラリを指定してください。	標準ライブラリの指定： 「第 1 章 概要・操作 1.3.5 標準ライブラリとの対応」
		未定義参照シンボル名が _ で始まっていないか。 (標準ライブラリ中の実行時ルーチンを使用しています)		ルーチン： 「第 3 章 システム組み込み 3.2.1 (2) 注意」
		C ライブラリ関数の標準入出力ライブラリを使用していないか。	低水準インタフェースルーチン作成してリンクしてください。	「第 3 章 システム組み込み 3.4 (6) 低水準インタフェースルーチン」

No	現象	確認内容	対処方法	参照
3	C++ソースレベルデバッグができない。	コンパイル、リンク時とも debug オプションを指定したか。	コンパイル時とリンク時それぞれに debug オプションを指定してください。	「第1章 概要・操作 1.3.3 コンパイラオプション」
		リンケージエディタの Ver.5.0 以上を使用しているか。	リンケージエディタの Ver.5.0 以上を使用してください。	
4	リンク時に、エラー No. 108 relocation size overflow が出力される。	GBR ベース変数の指定で、指定した変数のオフセットは制限内におさまっているか。	制限を越えるデータに対し、#pragma gbr_base/gbr_base1 宣言を削除してください。	「第2章 C++プログラミング 2.3.9GBR ベース変数の指定」
5	リンク時に、エラー No. 104 duplicate symbol が出力される。	同じ名称の変数または関数を複数のファイル内で外部定義していないか。	名前を変更するかまたは static を指定してください。	「第2章 C++プログラミング 2.3.6 (3) 使用上の注意事項 2.3.7 (3) 使用上の注意事項」
		複数のファイルでインクルードされるヘッダファイル内で変数または関数を外部定義していないか。 (#pragma inline/inline_asm 指定した関数でも同様です)	static を指定してください。	
6	ブラウジングできない。	コンパイル時に browser オプション、リンク時に debug オプションを指定したか。	コンパイル時に browser オプション、リンク時に debug オプションを指定してください。	「第1章 概要・操作 1.3.3 コンパイラオプション」

3. システム組み込み

第3章 目次

3.1	システム組み込みの概要.....	71
3.2	メモリ領域の割り付け.....	72
3.2.1	静的領域の割り付け	72
3.2.2	動的領域の割り付け	76
3.3	実行環境の設定.....	80
3.4	C ライブラリ関数の実行環境の設定	86

3.1 システム組み込みの概要

本節では、SH を応用したシステムに C++ プログラムを組み込む方法を説明します。

C++ プログラムをシステムに組み込むには、以下の準備が必要です。

(1) メモリの割り付け

C++ プログラムの各セクション、スタック領域、ヒープ領域をシステム上の ROM、RAM のメモリ領域に割り当てる必要があります。

(2) C++ プログラム実行環境の設定

C++ プログラムの実行環境を設定する処理には、レジスタの初期設定、メモリ領域の初期化、C++ プログラムの起動があります。これらの機能はアセンブリプログラムで実現する必要があります。

また、入出力等の C ライブラリ関数をご使用になる場合は、実行環境の設定時にライブラリの初期化をする必要があります。

2 節では C++ プログラムのメモリ領域のアドレスを決定する考え方を説明し、実際にアドレスを決定するためのリンケージエディタのコマンドの指定方法について事例を挙げて説明します。

3 節では実行環境設定の項目を説明し、設定プログラムの実例について説明します。

4 節では C ライブラリ関数の初期設定処理と低水準ルーチンの作成方法を説明します。

【注】 入出力 (stdio.h) とメモリ割り付け (stdlib.h) の機能をご使用になる場合は、システムのハードウェア構成にあわせて低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。

3.2 メモリ領域の割り付け

C++コンパイラの出力したオブジェクトプログラムをシステムに組み込むためには、プログラムの使用するメモリ領域のサイズを決定し、それぞれの領域を適切なメモリアドレスに割り付ける必要があります。

C++プログラムが使用するメモリ領域には、C++プログラム中の関数に対応する機械語や外部データ定義や静的データメンバで宣言したデータ領域のように静的に割り付ける領域とスタック領域のように動的に割り付ける領域があります。

以下、各領域の割り付け方を説明します。

3.2.1 静的領域の割り付け

(1) 静的領域の内容

オブジェクトプログラムの各セクション(プログラム領域、定数領域、初期化データ領域、未初期化データ領域、初期処理データ領域、後処理データ領域)は静的領域に割り付けます。

(2) サイズの算出法

静的領域のサイズは、C++コンパイラが生成するオブジェクトプログラムサイズとC++プログラムが使用するライブラリ関数のサイズの合計になります。オブジェクトプログラムをリンクしたあと、リンケージマップリストにライブラリを含めた各セクションのサイズを出力しますので、静的領域のサイズを知ることができます。

リンク前に静的領域のサイズを概算する場合は、コンパイルリストの統計情報にセクションごとのサイズを出力しますので、これに基づいてサイズを算出することができます。

図 3.1 に統計情報の例を示します。

```
***** SECTION SIZE INFORMATION *****
PROGRAM SECTION(P)           :0x00001E Byte(s)
PROGRAM SECTION(P_INIT_)     :0x000000 Byte(s)
PROGRAM SECTION(P_END_)      :0x000000 Byte(s)
CONSTANT SECTION(C)          :0x000000 Byte(s)
CONSTANT SECTION(C_INIT_)    :0x000000 Byte(s)
CONSTANT SECTION(C_END_)     :0x000000 Byte(s)
DATA SECTION(D)              :0x00001C Byte(s)
DATA SECTION(D_INIT_)        :0x000004 Byte(s)
DATA SECTION(D_END_)         :0x000004 Byte(s)
BSS SECTION(B)               :0x000028 Byte(s)
BSS SECTION(B_INIT_)         :0x000000 Byte(s)
BSS SECTION(B_END_)          :0x000000 Byte(s)
TOTAL PROGRAM SIZE           :0x000234 Byte(s)
```

図 3.1 統計情報例

標準ライブラリを使用しない場合は、コンパイル単位ごとに出力する統計情報のセクションごとのサイズの合計が静的領域のサイズになります。

また、標準ライブラリを使用している場合、各セクションのメモリ領域サイズにはライブラリ関数の使用するメモリ領域サイズを加えなければなりません。標準ライブラリ関数には、C 言語仕様で規定した C ライブラリ関数の他に、C++ プログラムを実行する上で必要な算術演算を行うルーチンも含まれています。そのため、C++ ソースプログラム上でライブラリ関数の使用を指定しなくても、必ず標準ライブラリをリンクする必要があります。

C++ コンパイラが提供する標準ライブラリの中には、C 言語仕様で規定した C ライブラリ関数と、C++ プログラムを実行する上で必要な算術演算を行うルーチン（実行時ルーチン）を含みます。実行時ルーチンのサイズも C ライブラリ関数と同じようにメモリ領域サイズに加える必要があります。

C++ プログラムで使用する実行時ルーチンは、C++ コンパイラ出力のアセンブリプログラム（オプション `code = asmcode` 指定）に外部参照シンボルとして出力しますので、そのシンボル名を参照することによって C++ プログラムで使用する実行時ルーチン名を知ることができます。

以下に具体例を示します。

[例]

C++ プログラム例

```
f( int a, int b)
{
    a /= b;
    return a;
}
```

C++ コンパイラ生成のアセンブリプログラム例

```

    .IMPORT      _ _divl$ ; 実行時ルーチンの外部参照宣言
    .EXPORT      _ _f_ _FSiT1
    .SECTION     P, CODE, ALIGN=4
_ _f_ _FSiT1:
    MOV          R5, R1
    MOV.L        A_divl$, R2
    JSR          @R2
    MOV          R4, R0
    RTS
    NOP
A_divl$:        .DATA.L    _ _divl$
    .END
```

上記例では `_divl$` が C++ プログラムで使用する実行時ルーチンになります。

(3) ROM、RAM の割り付け

プログラムをメモリに割り付ける場合は、静的な領域を以下のように ROM と RAM に分けて割り付けます。ただし、P_INIT_、P_END_、C_INIT_、C_END_、B_INIT_、B_END は割り付ける必要はありません。

プログラム領域	(セクション P)	ROM
定数領域	(セクション C)	ROM
未初期化データ領域	(セクション B)	RAM
初期化データ領域	(セクション D)	ROM、RAM (下記 (4) 参照)
初期処理データ領域	(セクション D_INIT_)	ROM
後処理データ領域	(セクション D_END_)	ROM

(4) 初期化データ領域の割り付け

初期化データ領域は、初期値を持ったデータを集めた領域です。この領域にあるデータは値の変更が可能なので、リンク時には ROM 上に置き、プログラムの実行開始時に RAM 上にコピーする必要があります。したがって、初期化データの領域については、ROM 上と RAM 上に、二重に領域を確保しなければなりません。

ただし、初期値を指定した静的変数を変更しないようにプログラムを作成すれば、初期化データの領域は ROM 上に置くだけでよく、RAM 上に割り付ける必要はありません。

(5) メモリ割り付け例とリンク時のアドレス指定方法

アブソリュートロードモジュール作成時にリンケージエディタのオプションまたはサブコマンドで各セクションごとに割り付ける領域のアドレスを指定します。以下、静的領域のメモリ割り付け例とリンク時の指定方法について説明します。

図 3.2 に静的な領域の割り付け例を示します。

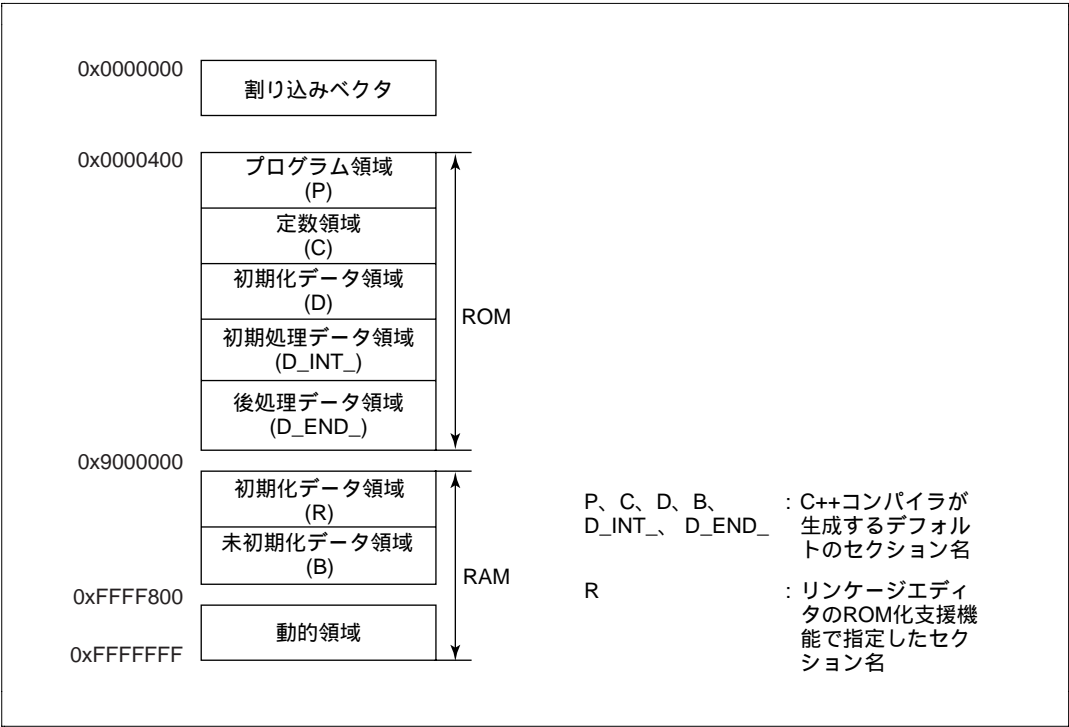


図 3.2 静的な領域の割り付け例

図 3.2 に示すメモリ割り付けを行う場合、リンク時に以下のサブコマンドを指定します。

：

ROM (D,R) (a)

START P,C,D(400),R,B(9000000) (b)

：

[説明]

- (a) セクション名 D と同じ大きさのセクション R を出力ロードモジュールに確保します。また、セクション D に割り付けられたシンボルを参照している場合、セクション R 上のアドレスとなるようリロケーションします。セクション D は ROM 上、セクション R は RAM 上の初期化データセクション名となります。
- (b) セクション P、C、D を内蔵 ROM のアドレス 0x400 から連続した領域に割り付けます。また、セクション R、B を RAM のアドレス 0x9000000 から連続したアドレスに割り付けます。

3.2.2 動的領域の割り付け

(1) 動的領域の内容

C++ プログラムで使用する動的領域には以下の2つがあります。

(a) スタック領域

(b) ヒープ領域 (メモリ割り付けライブラリ関数で使用)

(2) サイズの算出法

(a) スタック領域

C++ プログラムの使用するスタック領域は、関数の呼び出しのたびにスタック上に割り付け、関数のリターン時に解放します。スタック領域のサイズを算出するためには、まず各関数ごとのスタック使用量を算出し、関数の呼び出し関係から実際のスタック使用量を算出します。

(ア) 各関数の使用するスタック領域

各関数の使用するスタック領域は、C++コンパイラ出力のオブジェクトリスト中の frame size から分かります。

以下にオブジェクトリストとスタック上の割り付けの具体例を示し、そのスタック使用量の算出法について説明します。

[例]

次の C++ プログラムに対するオブジェクトリストとスタック使用量の算出法を示します。

```
extern int h(char, int *, double );
int h(char a, register int *b, double c)
{
    char      *d;

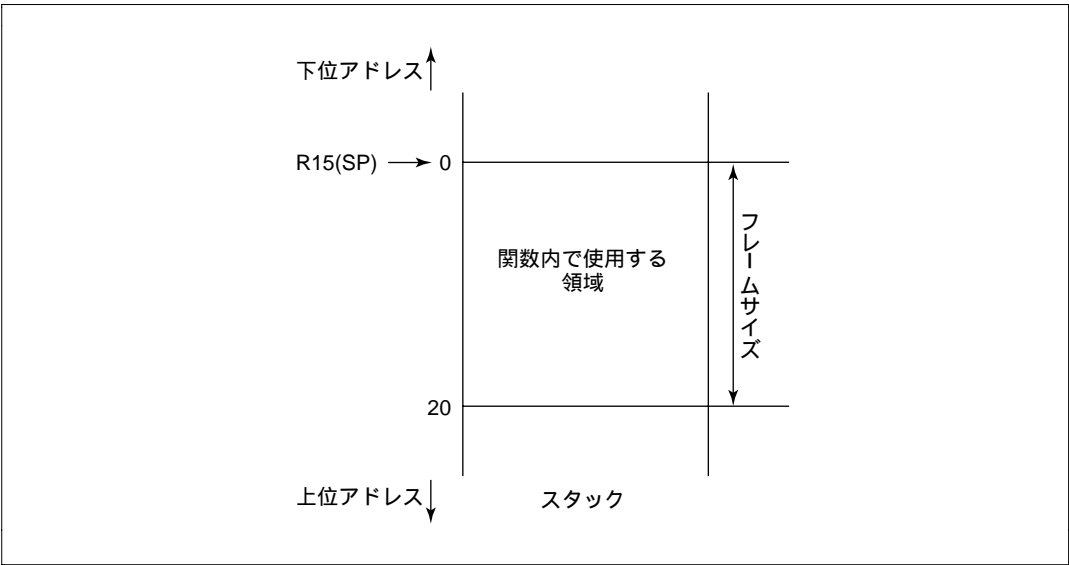
    d= &a;
    h(*d,b,c);
    {
        register int i;

        i= *d;
        return i;
    }
}
```

***** OBJECT LISTING *****

FILE NAME:m0251.cpp

SCT	OFFSET	CODE	C LABEL	INSTRUCTION	OPERAND	COMMENT
P						
	00000000		__h__FcPSid:			; function: h
						; frame size=20
	00000000	2FE6		MOV.L	R14,@-R15	
	00000002	4F22		STS.L	PR,@-R15	
			:			



関数の使用するスタック領域サイズは、フレームサイズの値と同じです。したがって、上記の例で関数 h の使用するスタック領域サイズは、オブジェクトリスト中の項目 COMMENT の frame size の値 20 バイトとなります。

スタック上の引数領域に割り付けられる引数については、「第2章 C++プログラミング 2.2.4.2 (4) 引数とリターン値の設定、参照に関する規則」を参照してください。

(b) スタック使用量の算出法

関数呼び出しの関係から使用するスタック領域のサイズを算出します。

[例]

関数呼び出しの関係と各関数のスタック使用量の例を図 3.3 に示します。

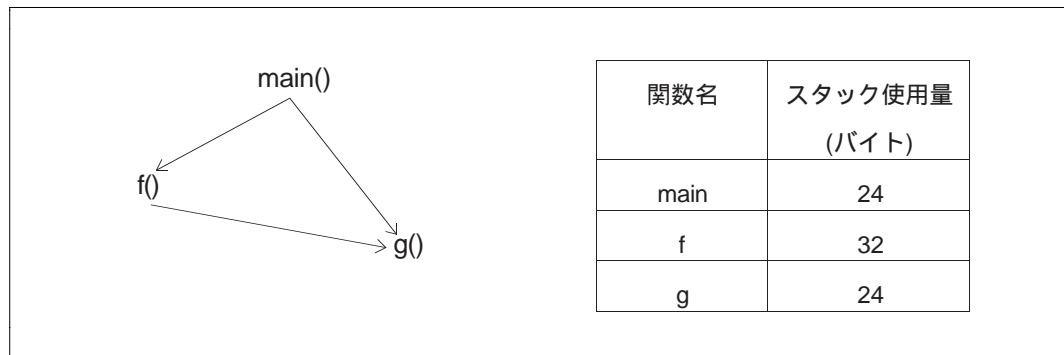


図 3.3 関数呼び出しの関係とスタック使用量の例

この場合、関数「f」を介して関数「g」が呼ばれた時のスタック領域のサイズは、表 3.1 に
よって計算します。

表 3.1 スタックサイズの計算例

呼び出し経路	スタックサイズ計 (バイト)
main (24) f (32) g (24)	80
main (24) g (24)	48

このように、呼び出しレベルの一番深い関数についてスタック領域のサイズを計算し、
その最大値 (この場合 80 バイト) のスタック領域を最低限割り付ける必要があります。

標準ライブラリのライブラリ関数を使用する場合には、ライブラリ関数を含めたスタック領域のサイズを計算する必要があります。ライブラリ関数の使用するスタック領域のサイズについては、製品添付の「標準ライブラリのメモリ・スタック使用量一覧」を参照してください。

【注】 C++プログラムの中で再帰呼び出しを行っている場合は、再帰的に呼び出す回数
の最大値を算出してから、その関数のスタック領域のサイズに再帰的に呼び出す
回数をかけて計算してください。

(c) ヒープ領域

ヒープ領域で使用するメモリ領域のサイズは、C++プログラム内でメモリ管理ライブラリ関数 (`calloc`、`malloc`、`realloc`、`new` 関数) によって割り付ける領域の合計です。ただし、メモリ管理ライブラリ関数は、1 回の呼び出しのたびに管理用の領域として 4 バイト、クラス配列オブジェクトを `new` 関数によって領域を割り付けた場合は、1 回の呼び出しのたびに管理用の領域として 12 バイト使用します。実際に確保する領域サイズにこの管理領域のサイズを加えて計算してください。

入出力ライブラリ関数は、内部処理の中でメモリ管理ライブラリ関数を使用しています。入出力の中で割り付ける領域のサイズは、

$$516 \text{ バイト} \times (\text{同時にオープンするファイルの数の最大値})$$

になります。

【注】メモリ管理ライブラリ関数の `free`、`delete` 関数で解放した領域は、再びメモリ管理ライブラリ関数で領域を確保するときに再利用しますが、割り付けを繰り返すことによって空き領域のサイズの合計は十分でも空き領域が小さな領域に分割しているために、新たに要求した大きなサイズの領域を確保できないという状況が生じることがあります。このような状況を避けるために、以下の注意に従ってヒープ領域を使用してください。

(ア) サイズの大きな領域は、なるべくプログラムの実行開始直後に確保してください。

(イ) 解放して再利用するデータ領域のサイズをなるべく一定にしてください。

(3) 動的領域の割り付け方

動的領域は RAM 上に割り付けます。

スタック領域は、ベクタテーブルにスタック領域の最上位アドレスを `SP` (スタックポインタ) として設定することにより割り付ける場所が決まります。SH3 では割り込み時の動作が SH1、SH2 の場合と異なるので、割り込みハンドラが必要になります。

ヒープ領域は、低水準インタフェースルーチン (`sbrk`) の初期設定で割り付ける場所が決まります。

それぞれ、「第 3 章 システム組み込み 3.3(1)ベクタテーブルの設定 (`VEC_TBL`)」、「第 3 章 システム組み込み 3.4(6)低水準インタフェースルーチン」を参照してください。

3.3 実行環境の設定

本節では、C++プログラムの実行に必要な環境を設定するための処理について説明します。ただし、C++プログラムを実行する環境はユーザシステムごとに異なりますので、使用するシステムの仕様に合わせて実行環境の設定プログラムを作成する必要があります。

ここでは、プログラムの実行環境の最も基本的な構成として、C ライブラリ関数を使用しない場合について説明します。

C ライブラリ関数、低水準の入出力ルーチン、メモリ割り付けルーチンを使用する場合は、「第3章 システム組み込み 3.4 C ライブラリ関数の実行環境の設定」を参照してください。

図 3.4 にプログラムの構成例を示します。

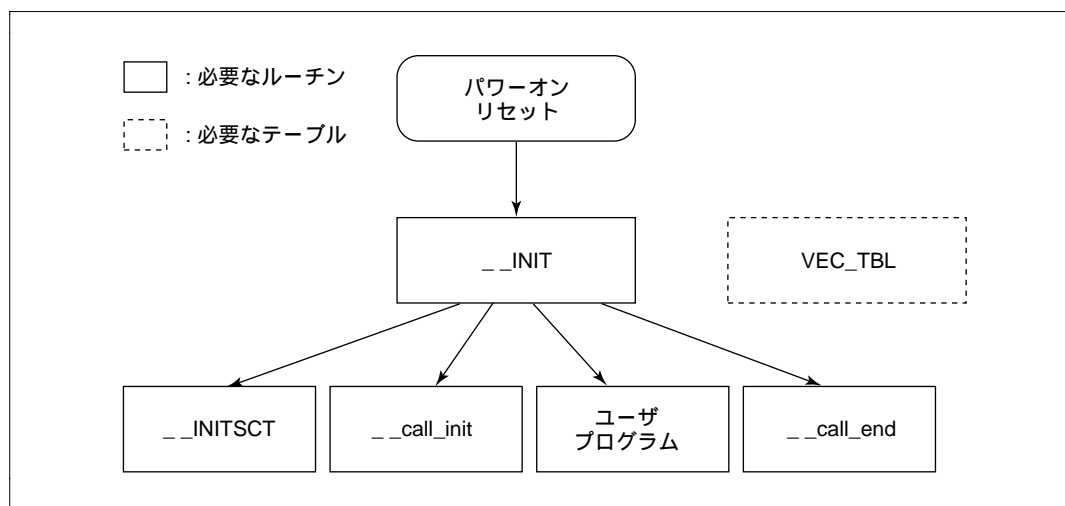


図 3.4 プログラムの構成例 (C ライブラリ関数を使用しない場合)

各構成ルーチンの内容は以下のとおりです。

(1) ベクタテーブルの設定 (VEC_TBL)

パワーオンリセットでレジスタの初期設定プログラム (__INIT) が起動され、またスタックポインタ (SP) に値が設定されるように、ベクタテーブルを設定します。SH3 では割り込み時の動作が SH1、SH2 の場合と異なるので割り込みハンドラが必要になります。

(2) 初期設定 (__INIT)

レジスタの初期設定を行ったあと、初期設定ルーチンを順次呼び出します。

(3) セクションの初期化 (__INITSC)

初期値が設定されていない静的変数領域 (未初期化データ領域) をゼロで初期化します。また、初期化データ領域の初期値を ROM 上から RAM 上にコピーします。

(4) グローバルオブジェクト初期処理 (__call_init)

グローバルに宣言されたクラスオブジェクトに対してコンストラクタを呼び出します。

(5) グローバルオブジェクト後処理 (__call_end)

main 関数処理の実行後、グローバルなクラスオブジェクトに対してデストラクタを呼び出します。

以下、この構成に従って各処理の実現方法について解説します。

(1) ベクタテーブルの設定 (VEC_TBL)

パワーオンリセットで、レジスタの初期設定を行う関数「__INIT」が呼び出されるようにするためには、ベクタテーブルの0番地に関数「__INIT」の先頭アドレスを設定します。また、スタックポインタ (SP) を設定するためには4番地にスタック領域の最上位アドレスを設定します。SH3では割り込み時の動作がSH1、SH2の場合と異なるので割り込みハンドラが必要になります。

また、ユーザシステムで割り込み処理を使用する場合は、割り込みベクタの設定も本ルーチンで行います。以下にそのコーディング例を示します。

[例]

```
.SECTION VECT,DATA,LOCATE=H'0000
                                ; セクション制御命令で「VECT セクション」を0番地に配置
.IMPORT __ __INIT
.IMPORT __IRQ0
.DATA.L __ __INIT              ; 「__INIT」の先頭アドレスを0x0~0x3番地に配置
.DATA.L (a)                   ; スタックポインタの値を0x4~0x7番地に配置
                                ; (a): スタック領域の最上位アドレス
.ORG H'00000100
.DATA.L __IRQ0                ; 「IRQ0」の先頭アドレスを0x100~0x103番地に配置
.END
```

(2) 初期設定 (__INIT)

ここでは、レジスタの初期設定を行い、初期設定ルーチンを順次呼び出したあと、main関数を呼び出します。

以下にコーディング例を示します。

[例]

```
extern "C" {
    extern void _INITSCT(void);
    extern void main(void);
}

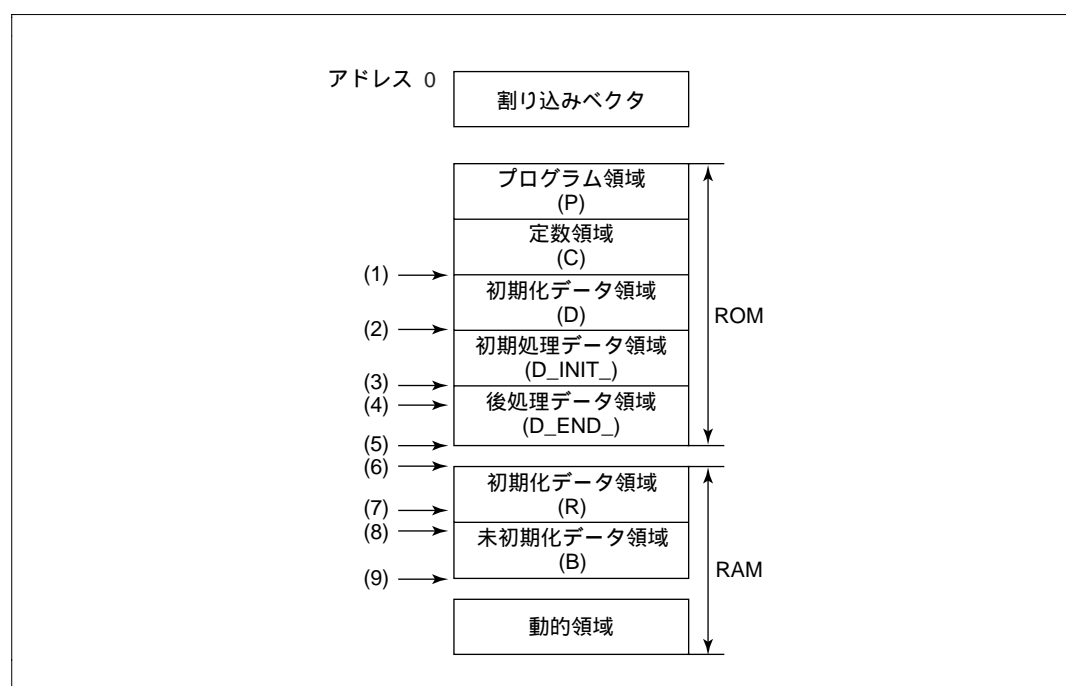
extern "C" void __INIT()
{
    _INITSCT();
    // セクションの初期化ルーチン「_INITSCT」の呼び出し
    _call_init();
    // グローバルオブジェクトのコンストラクタ処理ルーチン
    // 「_call_init」の呼び出し
    main();
    // メインルーチン「_main」の呼び出し
    _call_end();
    // グローバルオブジェクトのデストラクタ処理ルーチン
    // 「_call_end」の呼び出し
    for( ; ; )
        ;
    // main 関数終了後、無限ループしてリセットを待つ
}
```

(3) セクションの初期化 (_ _ _ _ _ _ _ _ _ _)

C++プログラムの実行環境を設定するために、未初期化データ領域をゼロで初期化することとROM上にある初期化データをRAM上にコピーすることが必要です。

「 _ _ _ _ _ _ _ _ _ _ 」の処理を行うためには、次のアドレスを知る必要があります。

- ・ 初期化データ領域のROM上の先頭アドレス (1)
- ・ 初期処理データ領域のROM上の先頭アドレス (2)、最終アドレス (3)
- ・ 後処理データ領域のROM先頭アドレス (4)、最終アドレス (5)
- ・ 初期化データ領域のRAM上の先頭アドレス (6)、最終アドレス (7)
- ・ 未初期化データ領域の先頭アドレス (8)、最終アドレス (9)



これらのアドレスを知るためには、次のアセンブリプログラムを作成、リンクしてください。

```

        .SECTION D,DATA,ALIGN=4
        .SECTION D_INIT_,DATA,ALIGN=4
        .SECTION D_END_,DATA,ALIGN=4
        .SECTION R,DATA,ALIGN=4
        .SECTION B,DATA,ALIGN=4
        .SECTION C,DATA,ALIGN=4
__ _D_ROM      .DATA.L  (STARTOF D)
;セクションDの先頭アドレス          (1)
__ _PRE_BGN    .DATA.L  (STARTOF D_INIT_)
;セクションD_INIT_の先頭アドレス    (2)
__ _PRE_END    .DATA.L  (STARTOF D_INIT_) + (SIZEOF D_INIT_)
;セクションD_INIT_の最終アドレス    (3)
__ _POST_BGN   .DATA.L  (STARTOF D_END_)
;セクションD_END_の先頭アドレス      (4)
__ _POST_END   .DATA.L  (STARTOF D_END_) + (SIZEOF D_END_)
;セクションD_END_の最終アドレス      (5)
__ _D_BGN      .DATA.L  (STARTOF R)
;セクションRの先頭アドレス          (6)
__ _D_END      .DATA.L  (STARTOF R) + (SIZEOF R)
;セクションRの最終アドレス          (7)
__ _B_BGN      .DATA.L  (STARTOF B)
;セクションBの先頭アドレス          (8)
__ _B_END      .DATA.L  (STARTOF B) + (SIZEOF B)
;セクションBの最終アドレス          (9)

        .EXPORT __ _D_ROM
        .EXPORT __ _PRE_BGN
        .EXPORT __ _PRE_END
        .EXPORT __ _POST_BGN
        .EXPORT __ _POST_END
        .EXPORT __ _D_BGN
        .EXPORT __ _D_END
        .EXPORT __ _B_BGN
        .EXPORT __ _B_END
        .END

```

【注】 1 セクション名B、Dはコンパイラオプション section で指定した未初期化データ領域、初期化データ領域のセクション名を指定してください。B、D はデフォルトのセクション名です。

2 セクション名Rは、リンク時にROM化支援オプションROMで指定したRAM上のセクション名を指定してください。Rはデフォルトのセクション名です。

上記の準備をすれば、セクションの初期化ルーチン、グローバルオブジェクト初期/後処理ルーチンはC++言語で記述することができます。以下にプログラム例を示します。

セクション初期化ルーチンの例

```
extern int *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;

extern "C" void _INITSCT( )
{
    int *p, *q;

    //未初期化データ領域をゼロで初期化

    for( p = _B_BGN; p < _B_END; p++)
        *p = 0;

    //初期化データをROM上からRAM上へコピー

    for(p = _D_BGN, q = _D_ROM; p < _D_END; p++, q++)
        *p = *q;
}
```

【注】 セクションのサイズが4の倍数バイトでない場合は、p、qの宣言をchar*にする必要があります。

グローバルオブジェクトの初期 / 後処理ルーチンの例

```
extern void (**_PRE_BGN) ( ), (**_PRE_END) ( );
extern void (**_POST_BGN) ( ), (**_POST_END) ( );
extern void _call_init ( );
extern void _call_end ( );

void _call_init ( )
{
    void (**ppf) ( );

    for ( ppf = _PRE_BGN; ppf < _PRE_END; ppf++)
        (*ppf) ( ); //グローバルオブジェクトのコンストラクタの呼び出し
}

void _call_end ( )
{
    void (**ppf) ( );

    for ( ppf = _POST_BGN; ppf < _POST_END; ppf++)
        (*ppf) ( ); //グローバルオブジェクトのデストラクタの呼び出し
}
```

3.4 C ライブラリ関数の実行環境の設定

C ライブラリ関数を使用する場合は、C++ プログラムの実行環境の設定として C ライブラリ関数の初期化をする必要があります。特に入出力(`stdio.h`)とメモリ割り付け(`stdlib.h`)の機能を使用する場合や、プログラム終了処理を行う C ライブラリ関数を使用する場合は、システムごとに低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。

本節では、C ライブラリ関数使用時の C++ プログラムの実行環境の設定方法について説明します。

図 3.5 にプログラムの構成例を示します。

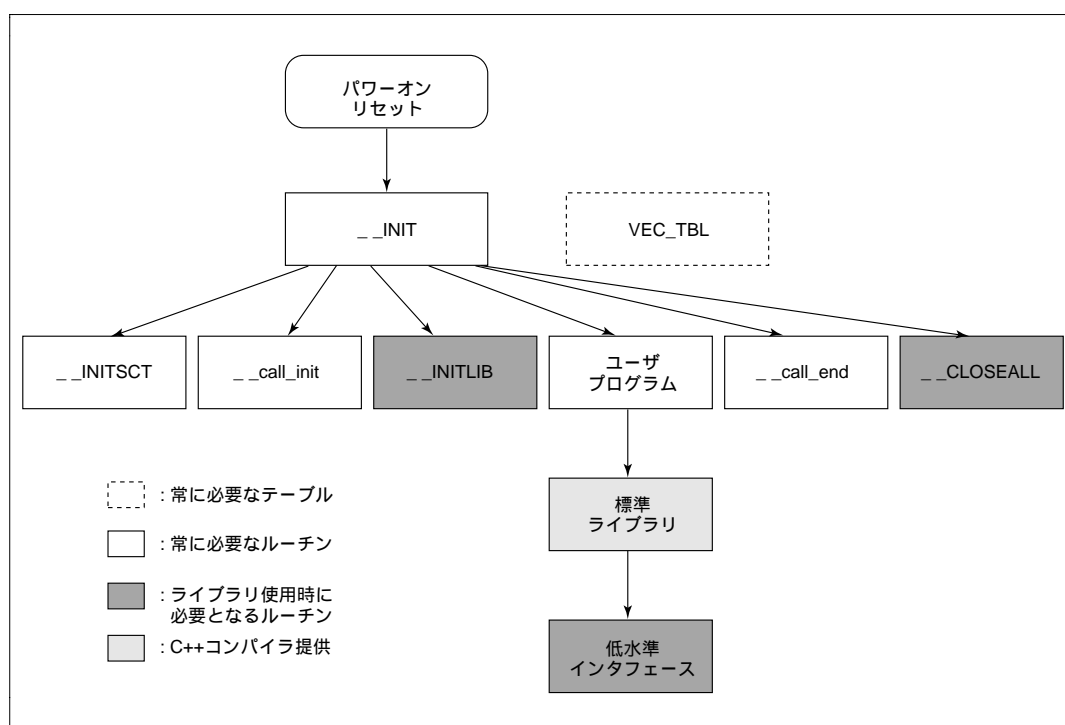


図 3.5 プログラムの構成例 (C ライブラリ関数を使用する場合)

プログラムの終了処理を行うCライブラリ関数 `exit`、`onexit`、`abort` 関数を使用する場合は、ユーザシステムに合わせてこれらの関数を作成する必要があります。具体的なプログラム例を「付録D. 終了処理関数の作成例」を参照してください。なお、Cライブラリ関数 `assert` マクロを使用する場合、`abort` 関数は必ず作成する必要があります。

以下にライブラリ使用時に必要な各構成ルーチンの内容を示します。

(1) ベクタテーブルの設定 (`VEC_TBL`)

パワーオンリセットでレジスタの初期設定プログラム (`__INIT`) が起動され、またスタックポインタ (`SP`) に値が設定させるように、ベクタテーブルを設定します。SH3では割り込み時の動作がSH1、SH2の場合と異なるので割り込みハンドラが必要になります。

(2) 初期設定 (`__INIT`)

レジスタの初期設定を行ったあと、初期設定ルーチンを順次呼び出します。

(3) セクションの初期化 (`__INITSCT`)

初期値が設定されていない静的変数領域 (未初期化データ領域) をゼロで初期化します。また、初期化データ領域の初期値をROM上からRAM上にコピーします。

(4) グローバルオブジェクト初期処理 (`__call_init`)

グローバルに宣言されたクラスオブジェクトに対してコンストラクタを呼び出します。

(5) グローバルオブジェクト後処理 (`__call_end`)

`main` 関数処理の実行後、破壊されるグローバルなクラスオブジェクトに対してデストラクタを呼び出します。

(6) Cライブラリの初期設定 (`__INITLIB`)

Cライブラリ関数の中で、初期設定の必要なものについて、初期設定を行います。特に、標準入出力を行う準備をします。

(7) ファイルのクローズ (`__CLOSEALL`)

オープンしているファイルをすべてクローズします。

(8) 低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリを使用する場合に必要なライブラリ関数とユーザシステムとの間のインタフェースルーチンです。

以下、この構成に従って各処理の実現方法について解説します。

(1) ベクタテーブルの設定 (`VEC_TBL`)

ベクタテーブルはCライブラリ関数を使用しない場合と同じです。「第3章 システム組み込み 3.3 実行環境の設定」を参照してください。

(2) 初期設定 (__INIT)

C ライブラリ関数を使用する場合には、ここでライブラリの初期設定を行う「__INITLIB」とファイルのクローズ処理を行う「__CLOSEALL」を呼び出します。以下に「__INIT」のコーディング例を示します。SH3 では割り込み時の動作がSH1、SH2の場合と異なるので割り込みハンドラが必要になります。

[例]

```
extern "C" {
    extern void __INITSCT(void);
    extern void __INITLIB(void);
    extern void main(void);
    extern void __CLOSEALL(void);
}
extern void __call_init(void);
extern void __call_end(void);

extern "C" void __INIT(void)
{
    __INITSCT();           //セクションの初期化ルーチン「__INITSCT」の呼び出し
    __INITLIB();           //ライブラリの初期化ルーチン「__INITLIB」の呼び出し
    __call_init();         //グローバルオブジェクトのコンストラクタ処理ルーチン
                           //「__call_init」の呼び出し
    main();                //メインルーチン「_main」の呼び出し
    __call_end();          //グローバルオブジェクトのデストラクタ処理ルーチン
                           //「__call_end」の呼び出し
    __CLOSEALL();          //ファイルのクローズルーチン「__CLOSEALL」の呼び出し

    for( ; ; )             //main関数終了後、無限ループしてリセットを待つ
        ;
}
```

(3) セクションの初期化 (__INITSCT)

セクションの初期化はC ライブラリ関数を使用しない場合と同じです。「第3章 システム組み込み 3.3 実行環境の設定」を参照してください。

(4) グローバルオブジェクト初期処理 (__call_init)

グローバルオブジェクトの初期処理は、C ライブラリ関数を使用しない場合と同じです。「第3章 システム組み込み 3.3 実行環境の設定」を参照してください。

(5) グローバルオブジェクト後処理 (__call_end)

グローバルオブジェクトの後処理は、C ライブラリ関数を使用しない場合と同じです。「第3章 システム組み込み 3.3 実行環境の設定」を参照してください。

(6) C ライブラリ関数の初期設定 (_ _INITLIB)

C ライブラリ関数の中には、初期設定が必要な関数があります。それらの関数を使用する場合、使用する前に定められた初期設定を行わなければなりません。本項では、プログラム起動ルーチン中の「_ _INITLIB」の中で初期設定を行う場合を例に説明します。

実際に使用する機能に合わせた必要最低限の初期設定を行うために、以下の指針を参考にしてください。

- (1) ライブラリのエラー状態を示す「errno」の初期設定はすべてのライブラリ関数共通に必要です。
- (2) <stdio.h> の各関数と assert マクロを使用する場合、標準入出力の初期設定が必要です。作成した低水準インタフェースルーチンの中で初期設定が必要な場合、低水準インタフェースルーチンに合わせた初期設定が必要です。
- (3) rand 関数、strtok 関数を使用する場合、標準入出力以外の初期設定が必要です。

ライブラリの初期設定を行うプログラム例を以下に示します。

[例]

```
#include <stdlib.h>
extern "C" {
    extern void _INIT_LOWLEVEL(void) ;
    extern void _INIT_IOLIB(void) ;
    extern void _INIT_OTHERLIB(void) ;
}

extern "C" void _INITLIB(void)
    /* アセンブリルーチンのシンボル名から下線を一つ削除 */
{
    errno=0;                /* ライブラリ共通の初期設定 */

    _INIT_LOWLEVEL( ) ;
    /* 低水準インタフェースの初期設定ルーチンの呼び出し */
    _INIT_IOLIB( ) ;
    /* 標準入出力の初期設定ルーチンの呼び出し */
    _INIT_OTHERLIB( ) ;
    /* 標準入出力以外の初期設定ルーチンの呼び出し */
}
```

以下、標準入出力の初期設定ルーチン(_INIT_IOLIB)、標準入出力以外の初期設定ルーチン(_INIT_OTHERLIB)の作成例を示します。低水準インタフェースルーチンの初期設定ルーチン(_INIT_LOWLEVEL)は、ユーザ作成の低水準インタフェースルーチンの仕様に合わせて作成してください。

(a) 標準入出力の初期設定ルーチン (_INIT_IOLIB) の作成例

標準入出力の初期設定では、ファイルを参照するために必要な FILE 型データ (図 3.6) の初期設定と標準入出力ファイルのオープンを行います。FILE 型データの初期設定は、必ず標準入出力ファイルのオープンの前に行ってください。

標準入出力の初期設定を行うプログラム例を以下に示します。

[例]

```
#include <stdio.h>
extern "C" void _INIT_IOLIB(void)
{
    FILE *fp ;
    /* FILE 型データの初期設定 */
    for (fp=_iob; fp<_iob+_NFILE; fp++){
        fp -> _bufptr=NULL ; // バッファポインタのクリア
        fp -> _bufcnt=0 ;    // バッファカウンタのクリア
        fp -> _buflen=0 ;    // バッファ長のクリア
        fp -> _bufbase=NULL ; // ベースポインタのクリア
        fp -> _ioflag1=0 ;   // I/O フラグのクリア
        fp -> _ioflag2=0 ;
        fp -> _iofd=0 ;
    }

    // 標準入出力ファイルのオープン

    if (freopen( "stdin"*1 , "r", stdin)==NULL) // 標準入力ファイルのオープン
        stdin->_ioflag1=0xff ;                // ファイルアクセスの禁止*2
        stdin->_ioflag1 |= _IOUNBUF ;           // データのバッファリング無*3

    if (freopen( "stdout"*1 , "w", stdout)==NULL) // 標準出力ファイルのオープン
        stdout->_ioflag1=0xff ;
        stdout->_ioflag1 |= _IOUNBUF ;

    if (freopen( "stderr"*1 , "w", stderr)==NULL) // 標準エラーファイルのオープン
        stderr->_ioflag1=0xff ;
        stderr->_ioflag1 |= _IOUNBUF ;
}
```

【注】 *1 標準入出力ファイルのファイル名を指定します。この名前は、低水準インタフェースルーチン「open」で使用します。

*2 ファイルのオープンが失敗した場合、ファイルアクセス禁止のフラグを立てます。

*3 コンソール等の対話的な装置の場合、バッファリングを行わないためのフラグを立てます。

```
// ファイル型データのC++言語での宣言
#define _NFILE 20
struct _iobuf{
    unsigned char *_bufptr;    // バッファへのポインタ
    long _bufcnt;             // バッファカウンタ
    unsigned char *_bufbase;   // バッファベースポインタ
    long _buflen;             // バッファ長
    char _ioflag1;            // I/O フラグ
    char _ioflag2;            // I/O フラグ
    char _iofd;               // I/O フラグ
}_iob[_NFILE];
```

図 3.6 FILE 型データ

(b) 標準入出力以外の初期設定ルーチン (_INIT_OTHERLIB) の作成例

標準入出力以外で初期設定が必要な C ライブラリ関数 (rand 関数、strtok 関数) の初期設定プログラム例を以下に示します。

[例]

```
#include <stddef.h>

extern char *_slptr ;
extern "C" void srand(unsigned int) ;

extern "C" void _INIT_OTHERLIB(void)
{
    srand(1) ;    // rand 関数を使用する場合の初期値の設定
    _slptr=NULL ; // strtok 関数で使用するポインタの初期設定
}
```

(7) ファイルのクローズ (_ _CLOSEALL)

通常ファイルへの出力は、メモリ領域上のバッファにためておき、バッファがいっぱいになったときに実際の外部記憶装置への書き出しを行います。したがってファイルのクローズを行わないと、ファイルへの出力内容が外部記憶装置へ書き出されないことがあります。

機組組み込み用のプログラムの場合、通常プログラムが終了することはありません。しかし、プログラムの誤りなどにより main 関数が終了する場合、オープンしているファイルはすべてクローズしなければなりません。

本処理は、main 関数終了時にオープンしているファイルのクローズを行います。

ファイルのクローズを行うプログラム例を以下に示します。

[例]

```
#include <stdio.h>

extern "C" void _CLOSEALL(void)
    // アセンブリルーチンのシンボル名から下線を一つ削除
{
    int i;

    for (i=0; i<_NFILE; i++)
        // ファイルがオープンしているかどうかのチェック

        if(_iob[i]._ioflag1 & ( _IOREAD | _IOWRITE | _IORW))
            // オープンしているファイルのクローズ

            fclose(&_iob[i]) ;
}
```

(8) 低水準インタフェースルーチン

標準入力、メモリ管理ライブラリを C++プログラムで使用する場合は、低水準インタフェースルーチンを作成しなければなりません。表 3.2 に C ライブラリ関数で使用している低水準インタフェースルーチンの一覧を示します。

表 3.2 低水準インタフェースルーチンの一覧

項番	名称	機能
1	open	ファイルのオープン
2	close	ファイルのクローズ
3	read	ファイルからの読み込み
4	write	ファイルへの書き出し
5	lseek	ファイルの読み込み / 書き出し位置の設定
6	sbrk	メモリ領域の確保

各 C ライブラリ関数に対して必要な低水準インタフェースルーチンについては、製品添付の「ソフトウェア添付資料」の中の「標準ライブラリのメモリスタック使用量一覧」を参照してください。

低水準インタフェースルーチンで必要な初期化は、プログラム起動時に行う必要があります。これは、「(4) C ライブラリ関数の初期設定 (__INITLIB) 」の中の「_INIT_LOWLEVEL」という関数の中で行ってください。

以下、低水準入出力の基本的な考え方を説明したあと、各インタフェースルーチンの仕様を説明します。また、SH シミュレータ・デバッガ上で実行する低水準インタフェースルーチン例を「付録 E. 低水準インタフェースルーチンの作成例」に示しますので、あわせて参照してください。

(a) 入出力の考え方

標準入出力ライブラリでは、ファイルを FILE 型のデータによって管理しますが、低水準インタフェースルーチンでは、実際のファイルと 1 対 1 に対応する正の整数を与え、これによって管理します。この整数をファイル番号といいます。

open ルーチンでは、与えられたファイル名に対してファイル番号を与えます。open ルーチンでは、この番号によってファイルの入出力ができるように、以下の情報を設定する必要があります。

- (1) ファイルのデバイスの種類(コンソール、プリンタ、ディスクファイル等)。コンソールやプリンタ等の特殊なデバイスに対しては、特別なファイル名をシステムで決めるにおいて open ルーチンで判定する必要があります。
- (2) ファイルのバッファリングをする場合はバッファの先頭位置、サイズ等の情報。
- (3) ディスクファイルならば、ファイルの先頭から次に読み込み / 書き出しを行う位置までのバイトオフセット。

open ルーチンで設定した情報に基づいて、以後、入出力 (read, write ルーチン)、読み込み / 書き出し位置の設定 (lseek ルーチン) を行います。

close ルーチンでは、出力ファイルのバッファリングを行っている場合はバッファの内容を実際のファイルに書き出し、open ルーチンで設定したデータの領域が再使用できるようにしてください。

(b) 低水準インタフェースルーチンの仕様

本項では、低水準インタフェースルーチンを作成するための仕様を説明します。以下、各ルーチンごとに、ルーチンを呼び出す際のインタフェースとその動作および実現上の注意事項を示します。

各ルーチンのインタフェースは以下の形式で示します。なお、低水準インタフェースルーチンは、必ず C リンケージでプロトタイプ宣言してください。

[凡例]

(ルーチン名)				
機能	(ルーチンの機能概要を示します。)			
インタフェース	(ルーチンのC++の関数としての宣言方法を示します。)			
引数	No.	名前	型	意味
	1	引数の名前です。	引数の型を示します。	(引数として渡される値の意味を示します。)
	:	:	:	:
	:	:	:	:
リターン値	型	(リターン値の型を示します。)		
	正常	(正常に終了した場合のリターン値の意味を示します。)		
	異常	(エラーが生じた場合のリターン値を示します。)		

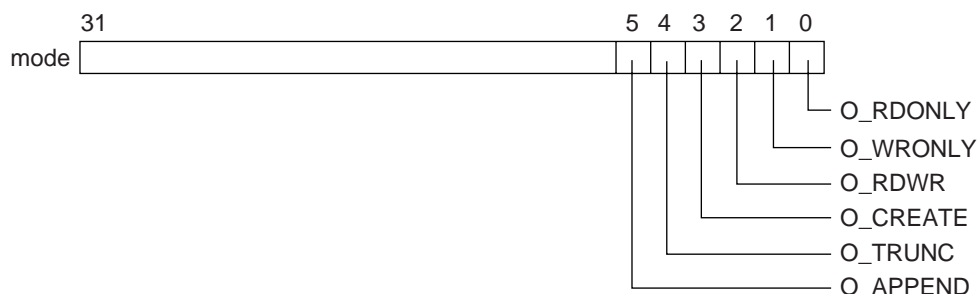
(a) open ルーチン				
機能	ファイルをオープンします。			
インタフェース	extern "C" int open (char *name, int mode);			
引数	No.	名前	型	意味
	1	name	char型を指すポインタ	ファイルのファイル名を指す文字列
	2	mode	int	ファイルをオープンするときの処理の指定
リターン値	型	int		
	正常	オープンしたファイルのファイル番号		
	異常	-1		

[説明]

第1引数として渡されたファイル名に対応するファイル进行操作するための準備をします。

open ルーチンでは、後で読み込み / 書き出しを行うために、ファイルの種類（コンソール、プリンタ、ディスクファイル等）を決定しなければなりません。ファイルの種類は、以後 open ルーチンで返したファイル番号を用いて読み込み / 書き出しを行うたびに参照する必要があります。

第2引数の mode は、ファイルをオープンするときの処理の指定です。このデータの各ビットの意味について以下に示します。



O_RDONLY (0ビット)	このビットが1のとき、ファイルを読み込み専用にオープン
O_WRONLY (1ビット)	このビットが1のとき、ファイルを書き出し専用にオープン
O_RDWR (2ビット)	このビットが1のとき、ファイルを読み込み、書き出し両用にオープン
O_CREATE (3ビット)	このビットが1のとき、ファイル名で示すファイルが存在しない場合にファイルを新規に作成
O_TRUNC (4ビット)	このビットが1のとき、ファイル名で示すファイルが存在する場合にファイルの内容を捨て、ファイルのサイズを0に更新
O_APPEND (5ビット)	次の読み込み / 書き出しを行うファイル内の位置を設定 ビットが0のとき：ファイルの先頭に設定 ビットが1のとき：ファイルの最後に設定

mode 示したファイルの処理の指定と実際のファイルの性質が矛盾する場合はエラーにしてください。

正常にファイルがオープンできた場合は、以後の read、write、lseek、close ルーチンで使用されるファイル番号（正の整数）を返してください。ファイル番号と実際のファイルの対応は、低水準インタフェースルーチンで管理する必要があります。オープンに失敗した場合は-1を返してください。

(b) close ルーチン				
機能	ファイルをクローズします。			
インタフェース	extern "C" int close (int fileno);			
引数	No.	名前	型	意味
	1	fileno	int	クローズするファイル番号
リターン値	型	int		
	正常	0		
	異常	-1		

[説明]

open ルーチンで得られたファイル番号が引数として渡されます。

open ルーチンで設定したファイル管理情報を再び使用できるように解放してください。

また、低水準インタフェースルーチン内で出力ファイルのバッファリングを行っている場合は、バッファの内容を実際のファイルに書き出してください。

ファイルを正常にクローズできた場合は0、失敗した場合は-1を返してください。

(c) read ルーチン				
機能	ファイルからデータの読み込みを行います。			
インタフェース	extern "C" int read (int fileno, char *buf, unsigned int count);			
引数	No.	名前	型	意味
	1	fileno	int	読み込み対象となるファイル番号
	2	buf	char型を指すポインタ	読み込んだデータを設定する領域
	3	count	unsigned int	読み込むバイト数
リターン値	型	int		
	正常	実際に読み込まれたバイト数		
	異常	-1		

〔 説明 〕

第 1 引数 (fileno) で示すファイルから、第 2 引数 (buf) の指す領域へデータを読み込みます。読み込むデータのバイト数は第 3 引数 (count) で示します。

ファイルが終了した場合、count で示されたバイト数以下のバイト数しか読み込むことができません。

ファイルの読み込み / 書き出しの位置は、読み込んだバイト数だけ先に進みます。

正常に読み込みができた場合は、実際に読み込んだバイト数を返してください。読み込みに失敗した場合は-1を返してください。

(d) writeルーチン				
機能	ファイルからデータの書き出しを行います。			
インタフェース	extern "C" int write (int fileno, char *buf, unsigned int count);			
引数	No.	名前	型	意味
	1	fileno	int	書き出し対象となるファイル番号
	2	buf	char型を指すポインタ	書き出すデータの領域
	3	count	unsigned int	書き出すバイト数
リターン値	型	int		
	正常	実際に書き出されたバイト数		
	異常	-1		

[説明]

第 2 引数 (buf) の指す領域から、第 1 引数 (fileno) の示すファイルにデータを書き出します。書き込むデータのバイト数は第 3 引数 (count) で示します。

ファイルを書き出そうとしているデバイス (ディスク等) が満杯の時は、count で示されたバイト数以下のバイト数しか書き出すことができません。実際に書き出すことのできたバイト数が何度か連続して 0 バイトの場合、ディスクが満杯であると判断してエラー (-1) を返すように実現することをお勧めします。

ファイルの読み込み / 書き出しの位置は、書き出したバイト数だけ先に進みます。

正常に書き出しができた場合は、実際に書き出したバイト数を返してください。書き出しに失敗した場合は -1 を返してください。

(e) lseekルーチン				
機能	ファイルの読み込み / 書き出し位置を設定します。			
インタフェース	extern "C" long lseek (int fileno, long offset, int base);			
引数	No.	名前	型	意味
	1	fileno	int	対象となるファイル番号
	2	offset	long	読み込み / 書き出し位置を示すオフセット (バイト単位)
	3	base	int	オフセットの起点
リターン値	型	int		
	正常	新しいファイルの読み込み / 書き出しの位置の先頭からのオフセット (バイト単位)		
	異常	-1		

[説明]

ファイルの読み込み / 書き出しを行うファイル内の位置を、バイト単位で設定します。
新しいファイル内の位置は、第 3 引数 (base) によって、以下の方法で計算し設定してください。

(1) base が 0 のとき

ファイルの先頭から offset バイトの位置に設定します。

(2) base が 1 のとき

現在の位置に offset バイトを加えた位置に設定します。

(3) base が 2 のとき

ファイルのサイズに offset バイトを加えた位置に設定します。

ファイルがコンソールやプリンタ等の対話的なデバイスの場合や、新しいオフセットの値が負になったり、(1)、(2) のときファイルのサイズを越える場合はエラーにします。

正しくファイル位置を設定できた場合は、新しい読み込み / 書き出し位置のファイルの先頭からのオフセットを、そうでない場合は -1 を返してください。

(f) sbrkルーチン				
機能	メモリ領域を割り付けます。			
インタフェース	extern "C" char *sbrk (unsigned long size);			
引数	No.	名前	型	意味
	1	size	unsigned long	割り付けるデータのサイズ (バイト数)
リターン値	型	char型を指すポインタ		
	正常	割り付けた領域の先頭アドレス		
	異常	(char*) -1		

[説明]

メモリ領域を割り付けるサイズが引数として渡されます。

連続して sbrk ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。

割り付けるメモリ領域が不足した場合はエラーにしてください。

正常に割り付けができた場合は、割り付けた領域の先頭アドレスを、失敗した場合は「(char *) -1」を返してください。

4. エラーメッセージ

第4章 目次

4.1	C++コンパイラのエラーメッセージ.....	105
	4.1.1 エラーメッセージ一覧.....	105
4.2	標準ライブラリのエラーメッセージ.....	141

4.1 C++コンパイラのエラーメッセージ

4.1.1 エラーメッセージ一覧

本章では、コンパイラの出力するエラーメッセージとエラー内容を説明します。

エラー番号 (エラーレベル) エラーメッセージ

エラーレベルは、エラーの重要度にしたい4種に分類されます。

エラーレベル (W) ウォーニング
(E) エラー
(F) フェータル
(-) インターナル

0100 (W) Function not optimized

関数サイズが大きすぎ、最適化がかかりませんでした。

1000 (W) Illegal pointer assignment

ポインタ型どうしの代入で、それぞれのポインタ型の指す型が異なります。

1001 (W) Illegal comparison in "演算子"

2項演算子 == または != の被演算子が、一方がポインタ型で他方が値 0 以外の汎整数型を指しています。

1002 (W) Illegal pointer for "演算子"

2項演算子 ==、!=、>、<、>= または <= の被演算子が、同じ型へのポインタ型を指していません。

1005 (W) Undefined escape sequence

文字定数または文字列の中で、文法上定義していない拡張表記 (逆スラッシュとそれに続く文字) を用いています。

1007 (W) Long character constant

文字定数の長さが2文字以上になっています。

1008 (W) Identifier too long

識別子の長さが250文字を超えています。

1010 (W) Character constant too long

文字定数の長さが4文字を超えています。

1012 (W) Floating point constant overflow

浮動小数点定数の値が値の範囲を超えています。符号にしたがって+ または- に対応する内部表現の値を仮定します。

1013 (W) Integer constant overflow

整数定数の値が unsigned long 型のとり得る値の範囲を超えています。オーバーフローした上位ビットを無視した値を仮定します。

1014 (W) Escape sequence overflow

文字定数あるいは文字列の中でのビットパターンを示す拡張表記の値が 255 を超えています。下位1バイトの値を有効とします。

1015 (W) Floating point constant underflow

浮動小数点定数の値の絶対値が表現できる最小値よりも小さな値となっています。定数の値を 0.0 と仮定します。

1016 (W) Argument mismatch

原型宣言の中の引数と関数呼び出しの対応する引数の型がポインタ型で、それぞれの指す型が異なっています。関数呼び出しの引数のポインタの内部表現をそのまま設定します。

1017 (W) Return type mismatch

関数の返す型とリターン文の式の型がポインタ型で、それぞれの指す型が異なっています。リターン文の式のポインタの内部表現をそのまま設定します。

1019 (W) Illegal constant expression

定数式において関係演算子 <、>、<= または >= の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

1020 (W) Illegal constant expression of "-"

定数式において2項演算子-の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

1021 (W) Register saving pragma conflict with interrupt function "関数名"

"関数名"で示す割り込み関数に対するレジスタ退避・回復を制御する`#pragma` が不適切です。`#pragma` 指定を無視します。

1200 (W) Division by floating point zero

定数式の中で浮動小数点数 0.0 を除数とする割り算を行っています。符号にしたがって、
+ または - に対応する内部表現の値を仮定します。

1201 (W) Ineffective floating point operation

定数式の中で `-`、`0.0/0.0` 等の無効演算を行っています。無効演算の結果を表す非数に対応する内部表現の値を仮定します。

1300 (W) Command parameter specified twice

同じコンパイラオプションを 2 度以上指定しています。同じコンパイラオプションの中で最後に指定したものを有効とします。

1400 (W) Function "関数名" in `#pragma inline` is not expanded

`#pragma inline` で指定した関数がインライン展開されませんでした。コンパイル処理を継続します。

2000 (E) Illegal preprocessor keyword

プリプロセッサ文で、誤ったキーワードを使用しています。

2001 (E) Illegal preprocessor syntax

プリプロセッサ文またはマクロ呼び出しの指定方法に誤りがあります。

2002 (E) Missing " , "

引数のある`#define` 文で引数の並びを区切るコンマ (,) が抜けています。

2003 (E) Missing ")"

名前が`#define` 文で定義されているかどうかを判定する `defined` 式で名前の次の右括弧 () が抜けています。

2004 (E) Missing ">"

`#include` 文のファイル名の指定でファイル名の次の `>` がありません。

2005 (E) Cannot open include file "ファイル名"

#include 文で指定したファイル名のファイルがオープンできません。

2006 (E) Multiple #defines

#define 文で同じマクロ名を再定義しています。

2008 (E) Processor directive #elif mismatches

#elif 文に対応する #if 文、#ifdef 文、#ifndef 文、#elif 文がありません。

2009 (E) Processor directive #else mismatches

#else 文に対応する #if 文、#ifdef 文、#ifndef 文がありません。

2010 (E) Macro parameters mismatch

マクロ呼び出しの引数の数がマクロ定義の引数の数と異なります。

2011 (E) Line too long

マクロ展開後のソースプログラムの行が限界値を超えています。

2012 (E) Keyword as a macro name

プリプロセッサで規定しているキーワードを #define 文または、#undef 文のマクロ名として定義しています。

2013 (E) Processor directive #endif mismatches

#endif 文に対応する #if、#ifdef、#ifndef 文がありません。

2014 (E) Missing #endif

#if 文、#ifdef 文、#ifndef 文に対応する #endif 文がないままでファイルが終了しました。

2016 (E) Preprocessor constant expression too complex

#if、#elif 文で指定した定数式の演算子と被演算子の合計が限界値を超えています。

2017 (E) Missing "

#include 文のファイル名の指定で、ファイル名の次に " がありません。

2018 (E) Illegal #line

#line 文で指定した行数が限界値を超えています。

2019 (E) File name too long

ファイル名の長さが 128 文字を超えています。

2020 (E) System identifier "名前" redefined

実行時ルーチンと同名のシンボルを定義しています。

2100 (E) Multiple storage classes

宣言の中で 2 つ以上の記憶クラス指定子を指定しています。

2101 (E) Address of register

レジスタ記憶クラスを持つ変数に対して、単項演算子&を適用しています。

2102 (E) Illegal combination

型指定子の組み合わせが誤っています。

2103 (E) Bad self reference structure

クラス、構造体、共用体のメンバの型を親のクラス、構造体または共用体と同じ型で宣言しています。

2104 (E) Illegal bit field width

ビットフィールド幅を示す定数式が整数型ではありません。あるいはビットフィールド幅として負の整数を指定しています。

2105 (E) Incomplete tag used in declaration

クラス、構造体または共用体で仮宣言されたタグ名または、未宣言のタグ名を typedef 宣言、ポインタを指す型あるいは関数の返す型以外の宣言で使用しています。

2106 (E) Extern variable initialized

複文内で extern 記憶クラスを指定した宣言に対して初期値を指定しています。

2107 (E) Array of function

要素の型が関数型となる配列型を指定しています。

2108 (E) Function returning array

リターン値の型が配列型となる関数型を指定しています。

2109 (E) Illegal function declaration

複文内の関数型の変数の宣言において、extern 以外の記憶クラスを指定しています。

2110 (E) Illegal storage class

外部定義の中で記憶クラスとして auto または register を指定しています。

2111 (E) Function as a member

構造体または共用体のメンバの型に関数型を指定しています。

2112 (E) Illegal bit field

ビットフィールドに整数型以外の型を指定しています。

2113 (E) Bit field too wide

ビットフィールド幅が型指定子で指定したサイズ (8 ビット、16 ビット、32 ビット) を超えています。

2114 (E) Multiple variable declarations

変数名を同じ有効範囲の中で重複して宣言しています。

2115 (E) Multiple tag declarations

クラス、構造体、共用体、列挙型のタグ名を同じ有効範囲の中で重複して宣言しています。

2117 (E) Empty source program

ソースプログラム内に外部定義が含まれていません。

2118 (E) Prototype mismatch

関数の型が以前になされている宣言で指定した型と一致しません。

2119 (E) Not a parameter name

関数の引数宣言列にない識別子に対して引数宣言を行っています。

2120 (E) Illegal parameter storage class

関数の引数宣言で register 以外の記憶クラスを指定しています。

2121 (E) Illegal tag name

クラス、構造体、共用体または列挙型とタグ名の組み合わせが、以前に宣言した型とタグ名の組み合わせと異なっています。

2122 (E) Bit field width 0

メンバ名を指定しているビットフィールドの幅が0になっています。

2123 (E) Undefined tag name

列挙型の宣言で未定義のタグ名を使用しています。

2124 (E) Illegal enum value

列挙型のメンバに整数でない定数式を指定しています。

2125 (E) Function returning function

リターン値の型が関数型となる関数型を指定しています。

2126 (E) Illegal array size

配列の要素数の値が1以上 2147483647 以下の整数値以外の値を指定しています。

2127 (E) Missing array size

配列の要素数の指定がありません。

2128 (E) Illegal pointer declaration for "*"

ポインタ型の宣言を示す * の直後に const、volatile 以外の型指定子を指定しています。

2129 (E) Illegal initializer type

変数の初期値指定において初期値の型が変数に代入可能な型ではありません。

2130 (E) Initializer should be constant

クラス型、構造体型、共用体型、配列型の変数の初期値、または静的に割り付けられる変数の初期値に定数式でないものを指定しています。

2131 (E) No type or storage class

外部データ定義において記憶クラスまたは型の指定がありません。

2132 (E) No parameter name

関数の引数宣言列が空であるにもかかわらず引数の宣言を行っています。

2133 (E) Multiple parameter declarations

(マクロ) 関数定義の引数宣言列の中で同一名の引数を重複して宣言しているか、または引数宣言が関数宣言子の中と外の 2 箇所で行われています。

2134 (E) Initializer for parameter

引数の宣言において初期値を指定しています。

2135 (E) Multiple initialization

同一の変数に対して、初期化を重複して行っています。

2136 (E) Type mismatch

`extern` あるいは `static` 記憶クラスを持つ変数あるいは関数を 2 度以上宣言しており、その型が一致していません。

2137 (E) Null declaration for parameter

関数の引数宣言で識別子を指定していません。

2138 (E) Too many initializers

構造体、共用体または配列の初期値指定において、構造体のメンバ数または配列の要素数より多く初期値の数を指定しています。あるいは、共用体の最初のメンバがスカラ型のときに 2 個以上の初期値を指定しています。

2139 (E) No parameter type

関数宣言の引数宣言に型指定がありません。

2140 (E) Illegal bit field

共用体にビットフィールドを指定しています。

2141 (E) Struct has no member name

構造体のメンバ名が指定されていません。

2142 (E) Illegal void type

`void` 型の指定方法に誤りがあります。 `void` 型を指定できるのは以下の 3 つの場合です。

- (1) ポインタの指す先の型として指定する場合。
- (2) 関数の返す型として指定する場合。
- (3) 原型宣言の関数が引数を持たないことを明示的に指定する場合。

2143 (E) Illegal static function

ソースファイル内に定義のない `static` 記憶クラスを持つ関数宣言があります。

2144 (E) Type mismatch

`extern` 記憶クラスを持つ同じ名前の変数あるいは関数の型が一致していません。

2200 (E) Index not integer

配列の添字の式が整数型ではありません。

2201 (E) Cannot convert parameter

関数呼び出しの引数を対応する原型宣言の引数の型に変換できません。

2202 (E) Mismatch number of parameters

関数呼び出しにおける引数の数が原型宣言の引数の数と一致しません。

2203 (E) Illegal member reference for "."

演算子 `.` の左側の式の型がクラス型、構造体型、共用体型ではありません。

2204 (E) Illegal member reference for "->"

演算子 `->` の左側の式の型がクラス型、構造体型または共用体型へのポインタではありません。

2205 (E) Undefined member name

クラス型、構造体、共用体への参照で宣言していないメンバ名を使用しています。

2206 (E) Modifiable lvalue required for "演算子"

前置または後置演算子 `++`、`--` を代入可能な左辺値（配列型、`const` 型を除く左辺値）でない式に使用しています。

2207 (E) Scalar required for "!"

単項演算子 `!` をスカラ型でない式に使用しています。

2208 (E) Pointer required for "***"

単項演算子 `*` をポインタ型でない式か、または `void` 型へのポインタ型の式に使用しています。

2209 (E) Arithmetic type required for "演算子"

単項演算子 + または - を算術型でない式に使用しています。

2210 (E) Integer required for "~"

単項演算子 ~ を汎整数型でない式に使用しています。

2211 (E) Illegal sizeof

sizeof 演算子をビットフィールドの指定のあるメンバ、関数型、void 型またはサイズの指定していない配列に使用しています。

2212 (E) Illegal cast

キャスト演算子で指定している型が配列型、クラス型、構造体型または共用体型です。あるいはキャスト演算子の被演算子が void 型、構造体型か共用体型で型変換できません。

2213 (E) Arithmetic type required for "演算子"

2 項演算子 *, /、*= または /= を算術型でない式に適用しています。

2214 (E) Integer required for "演算子"

2 項演算子 <<, >>, &, |, ^, %, <<=, >>=, &=, |=, ^= または %= を汎整数型でない式に適用しています。

2215 (E) Illegal type for "+"

2 項演算子 + の被演算子の型の組み合わせが許されていません。

2216 (E) Illegal type for parameter

関数呼び出しの引数の型に void 型を指定しています。

2217 (E) Illegal type for "-"

2 項演算子 - の被演算子の型の組み合わせが許されていません。

2218 (E) Scalar required

条件演算子 ?: の第 1 被演算子の型がスカラ型ではありません。

2219 (E) Type not compatible with "? :"

条件演算子 ?: の第 2 被演算子と第 3 被演算子の型が合っていない。

2220 (E) Modifiable lvalue required for "演算子"

代入演算子 `=`、`*=`、`/=`、`%=`、`+=`、`-=`、`<<=`、`>>=`、`&=`、`^=` または `|=` の左辺の式に代入可能な左辺値（配列型、`const` 型を除く左辺値）以外の式を指定しています。

2221 (E) Illegal type for "演算子"

後置演算子 `++` または `--` の被演算子にスカラ型以外の型、関数型または `void` 型へのポインタ型を指定しています。

2222 (E) Type not compatible for "="

代入演算子 `=` の両辺の式の型が合っていません。

2223 (E) Incomplete tag used in expression

クラス、構造体または共用体で仮宣言されたタグ名を式中使用しています。

2224 (E) Illegal type for assign

代入演算子 `+=` または `-=` の両辺の型が正しくありません。

2225 (E) Undeclared name

宣言していない名前を式の中で用いています。

2226 (E) Scalar required for "演算子"

2項演算子 `&&` または `||` をスカラ型でない式に適用しています。

2227 (E) Illegal type for equality

等値演算子 `==` または `!=` の被演算子の型の組み合わせが許されていません。

2228 (E) Illegal type for comparison

関係演算子 `>`、`<`、`>=` または `<=` の被演算子の型の組み合わせが許されていません。

2230 (E) Illegal function call

関数呼び出しにおいて、関数型あるいは関数型へのポインタ型でない式を用いています。

2231 (E) Address of bit field

単項演算子 `&` をビットフィールドに適用しています。

2232 (E) Illegal type for "演算子"

前置演算子 ++, または -- の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。

2233 (E) Illegal array reference

配列型、関数型または void 型を除くポインタ型以外の式を配列として使用しています。

2234 (E) Illegal typedef name reference

typedef 宣言された名前を式の中で変数として使用しています。

2235 (E) Illegal cast

ポインタを浮動小数点型にキャストしています。

2236 (E) Illegal cast in constant

定数式でポインタ型を char 型または short 型にキャストしています。

2237 (E) Illegal constant expression

定数式の中でポインタ型の定数を整数型へキャストした結果に対して演算を行っています。

2238 (E) Lvalue or function type required for "&"

単項演算子 & を左辺値あるいは関数型以外の式に適用しています。

2300 (E) Case not in switch

case ラベルを switch 文以外に指定しています。

2301 (E) Default not in switch

default ラベルを switch 文以外に指定しています。

2302 (E) Multiple labels

1 つの関数内にラベル名を重複して定義しています。

2303 (E) Illegal continue

continue 文を while 文、for 文または do 文以外に指定しています。

2304 (E) Illegal break

break 文を while 文、for 文、do 文または switch 文以外に指定しています。

2305 (E) Void function returns value

void 型を返す関数の中の return 文でリターン値を指定しています。

2306 (E) Case label not constant

case ラベルの式が汎整数型の定数式ではありません。

2307 (E) Multiple case labels

同一の値を持つ case ラベルを 1 つの switch 文の中に重複して指定しています。

2308 (E) Multiple default labels

default ラベルを 1 つの switch 文の中に重複して指定しています。

2309 (E) No label for goto

goto 文で指定した行き先のラベルがありません。

2310 (E) Scalar required

while 文、for 文または do 文の制御式（文の実行を判定する式）がスカラ型ではありません。

2311 (E) Integer required

switch 文の制御式（文の実行を判定する式）が汎整数型ではありません。

2312 (E) Missing (

if 文、while 文、for 文、do 文または switch 文の制御式（文の実行を判定する式）の左括弧「(」がありません。

2313 (E) Missing ;

do 文の最後のセミコロンの「;」がありません。

2314 (E) Scalar required

if 文の制御式（文の実行を判定する式）がスカラ型ではありません。

2316 (E) Illegal type for return value

return 文の式の型を関数の返す型に変換することができません。

2400 (E) Illegal character "文字"

不正な印字文字があります。

2401 (E) Incomplete character constant

文字定数の途中で改行があります。

2402 (E) Incomplete string

文字列の途中で改行があります。

2403 (E) EOF in comment

コメントの途中でファイルが終了しました。

2404 (E) Illegal character code "文字コード"

不正な文字コードがあります。

2405 (E) Null character constant

文字定数の中に文字を指定していません。すなわち `''` という形式の文字定数を指定しています。

2406 (E) Out of float

浮動小数点定数の有効桁数が 17 桁を超えています。

2407 (E) Incomplete logical line

空でないソースファイルの最後の文字に、バックスラッシュ (`\`) またはバックスラッシュのあとに改行文字 (`\ (RET)`) を指定しています。

2408 (E) Comment nest too deep

コメントのネストが限界値を超えています。限界値は 255 レベルです。

2500 (E) Illegal token

語句の並びが文法に合っていません。

2501 (E) Division by zero

定数式中で整数型データのゼロ除算が行われました。

2600 (E) 文字列

`nolist` オプションが指定されていなければ、`#error` の文字列で指定されたエラーメッセージをリストファイルに表示します。

2650 (E) Invalid pointer reference

指定されたアドレス値が境界調整数と一致しません。

2700 (E) Function "関数名" in #pragma interrupt already declared

割り込み関数宣言#pragma interrupt で指定した関数が、すでに通常の関数として宣言されています。

2701 (E) Multiple interrupt for one function

1 つの関数に対して割り込み関数宣言#pragma interrupt を重複して宣言しています。

2702 (E) Multipe #pragma interrupt options

同種の割り込み仕様が重複して指定されています。

2703 (E) Illegal #pragma interrupt declaration

割り込み関数宣言#pragma interrupt の仕様の指定が異なります。

2704 (E) Illegal reference to interrupt function

割り込み関数を不正に参照しています。

2705 (E) Illegal parameter in interrupt funtion

割り込み関数で使用する引数の型が一致していません。

2706 (E) Missing parameter declaration in interrupt function

割り込み関数のオプション指定で使用する変数の宣言がありません。

2707 (E) Parameter out of range in interrupt function

割り込み関数のパラメタ tn の値が 256 を超えています。

2709 (E) Illegal section name declaration

#pragma section 指定に誤りがあります。

2710 (E) Section name too long

指定したセクション名の長さが 31 文字を超えています。

2711 (E) Section name table overflow

指定したセクションの数が 1 ファイルで 64 個を超えています。

2712 (E) GBR based displacement overflow

#pragma gbr_base で宣言した変数の領域がオーバーフローしました。

2713 (E) Illegal #pragma interrupt function type

#pragma interrupt 指定した関数の型が不正です。

2800 (E) Illegal parameter number in in-line function

組み込み関数で使用する引数の数が一致しません。

2801 (E) Illegal parameter type in in-line function

組み込み関数で引数の型が一致しません。

2802 (E) Parameter out of range in in-line function

組み込み関数で引数の大きさが指定可能範囲を超えています。

2803 (E) Invalid offset value in in-line function

組み込み関数で引数の指定が不適当です。

2804 (E) Illegal in-line function

組み込み関数 trapa_svc の指定に誤りがあります。

2805 (E) Function "関数名" in #pragma inline/inline_asm already declared

"関数名"で示す関数の本体が、#pragma 指定よりも前にあります。

2806 (E) Multiple #pragma for one function

1 つの関数に対して複数の矛盾した #pragma 指定をしています。

2807 (E) Illegal #pragma inline/inline_asm declaration

#pragma inline または #pragma inline_asm の指定方法に誤りがあります。

2808 (E) Illegal option for #pragma inline_asm

#pragma inline_asm の指定があるにもかかわらず、-code=machinecode オプションを指定しています。

2809 (E) Illegal option for #pragma inline/inline_asm function type

#pragma inline または #pragma inline_asm を指定した識別子の種類が誤っています。

2810 (E) Global variable "変数名" in #pragma gbr_base/gbr_base1 already declared
"変数名" で示す変数の定義が#pragma 指定よりも前にあります。

2811 (E) Multiple #pragma for one global variable
変数に対して複数の矛盾する#pragma が指定されています。

2812 (E) Illegal #pragma gbr_base/gbr_base1 declaration
#pragma gbr_base、#pragma gbr_base1 の指定方法が誤っています。

2813 (E) Illegal #pragma gbr_base/gbr_base1 global variable type
#pragma gbr_base、#pragma gbr_base1 を指定した識別子の種類が誤っています。

2814 (E) Function "関数名" in #pragma noregsave/norealloc/regsave already declared
"関数名" で示す関数の本体が、#pragma 指定よりも前にあります。

2815 (E) Illegal #pragma noregsave/noregalloc/regsave declaration
#pragma noregsave、#pragma noregalloc、#pragma regsave の指定方法が誤っています。

2816 (E) Illegal #pragma noregsave/noregalloc/regsave function type
#pragma noregsave、#pragma noregalloc、#pragma regsave を指定した識別子の種類が誤っています。

2817 (E) Symbol "識別子" in #pragma abs16 already declared
"識別子" で示す名前の宣言が、#pragma 指定よりも前にあります。

2818 (E) Multiple #pragma for one symbol
同一の識別子に対して、複数の矛盾した #pragma が指定されています。

2819 (E) Illegal #pragma abs16 declared
#pragma abs16 の指定方法が誤っています。

2820 (E) Illegal #pragma abs16 symbol type
#pragma abs16 を指定した識別子の種類が誤っています。

3000 (F) Statement nest too deep
if 文、while 文、for 文、do 文および switch 文のネストが限界値を超えています。限界値は、32 レベルです。

3001 (F) Block nest too deep

複文のネストが限界値を超えています。限界値は、32 レベルです。

3002 (F) #if nest too deep

条件コンパイル (#if、#ifdef、#ifndef、#elif、#else) のネストが限界値を超えています。限界値は、32 レベルです。

3006 (F) Too many parameters

関数の宣言または呼び出しにおいて引数の数が限界値を超えています。限界値は、63 個です。

3007 (F) Too many macro parameters

マクロの定義または呼び出しにおいて、引数の数が限界値を超えています。限界値は、63 個です。

3008 (F) Line too long

マクロ展開後の 1 行の長さが限界値を超えています。限界値は、4096 文字です。

3009 (F) String literal too long

文字列の長さが 512 文字を超えています。文字列の長さは、連続して指定した文字列を連結した後のバイト数です。ここでいう文字列の長さとは、ソースプログラム上の長さではなく文字列のデータに含まれるバイト数で、拡張表記も 1 文字に数えます。

3010 (F) Processor directive #include nest too deep

#include 文によるファイルの取り込みのネストが限界値を超えています。限界値は、30 レベルです。

3011 (F) Macro expansion nest too deep

#define 文によるマクロ名の再置換が限界値を超えています。限界値は、32 個です。

3012 (F) Too many function definitions

関数定義の数が限界値を超えています。限界値は、512 個です。

3013 (F) Too many switches

switch 文の数が限界値を超えています。限界値は、256 個です。

3014 (F) For nest too deep

for 文のネストが限界値を超えています。限界値は、16 レベルです。

3015 (F) Symbol table overflow

コンパイラが生成するシンボルの数が限界値を超えています。限界値は、24576 個です。

3016 (F) Internal label overflow

コンパイラが生成する内部ラベルの数が限界値を超えています。限界値は、32767 個です。

3017 (F) Too many case labels

1 つの switch 文の中の case ラベルの数が限界値を超えています。限界値は、511 個です。

3018 (F) Too many goto labels

1 つの関数の中で定義している goto ラベルの数が限界値を超えています。限界値は、511 個です。

3019 (F) Cannot open source file "ファイル名"

ソースファイルをオープンすることができません。

3020 (F) Source file input error "ファイル名"

ソースファイルまたはインクルードファイルを読み込むことができません。

3021 (F) Memory overflow

コンパイラが内部で使用するメモリ領域を割り当てることができません。

3022 (F) Switch nest too deep

switch 文のネストが限界値を超えています。限界値は、16 レベルです。

3023 (F) Type nest too deep

基本型を修飾する型（ポインタ型、配列型、関数型）の数が16個を超えています。

3024 (F) Array dimension too deep

配列の次元数が6次元を超えています。

3025 (F) Source file not found

コマンドラインの中にソースファイル名の指定がありません。

3026 (F) Expression too complex

式が複雑すぎます。

3027 (F) Source file too complex

プログラムの文のネストが深いあるいは、式が複雑すぎます。

3028 (F) Source line number overflow

ソース行番号が限界値を超えています。限界値は、UNIX 版の場合 32767 行、PC 版の場合 65535 行です。

3030 (F) Too many compound statements

複文の数が限界値を超えました。

3031 (F) Data size overflow

配列または構造体の大きさが、2147483647 バイトを超えています。

3033 (F) Symbol table overflow

デバッグ情報のためのシンボルの数が、8162 個を超えています。

3100 (F) Misaligned pointer access

境界整合が正しくないポインタを用いて参照または設定をしようとしています。

3201 (F) Object size overflow

オブジェクトサイズが 4G バイトを超えています。

3202 (F) Too many source lines for debug

ソース行数が多すぎて、デバッグのための情報が出力できません。

3203 (F) Assembly source line too long

出力するアセンブリソースの 1 行が長すぎます。

3204 (F) Illegal stack access

関数内で使用するスタックのサイズ（局所変数領域、レジスタ退避領域その他関数呼び出しのためのパラメタプッシュ領域等含む）または、その関数呼び出しのためのパラメタ領域が2Gバイトを超えています。

3300 (F) Cannot open internal file

以下、4つの場合のいずれかでエラーが起こっている可能性があります。

- (1) コンパイラが内部で生成する中間ファイルをオープンすることができません。
- (2) 中間ファイルと同じ名前のファイルが既に存在しています。
- (3) リストファイル仕様のパス名が128文字を超えています。
- (4) コンパイラが内部で使用するファイルをオープンすることができません。

3301 (F) Cannot close internal file

コンパイラが内部で生成する中間ファイルをクローズすることができません。コンパイラのインストール手順に誤りがないことを確認してください。

3302 (F) Cannot input internal file

コンパイラが内部で生成する中間ファイルを読み込むことができません。コンパイラのインストール手順に誤りがないことを確認してください。

3303 (F) Cannot output internal file

コンパイラが内部で生成する中間ファイルに書き込むことができません。

3304 (F) Cannot delete internal file

コンパイラが内部で生成する中間ファイルを削除することができません。

3305 (F) Invalid command parameter "オプション名"

コンパイラオプションの指定方法が誤っています。

3306 (F) Interrupt in compilation

コンパイル処理中に標準入力端末から (CNTL) C コマンドによる割り込みを検出しました。

3307 (F) Compiler version mismatch

コンパイラを構成するファイル間のバージョンが一致していません。

3320 (F) Command parameter buffer overflow

コマンドラインの指定が 256 文字を超えています。

3321 (F) Illegal environment variable

以下のどちらかの場合でエラーが起っています。

(1) 環境変数 SHC_LIB が設定されていません。

(2) 環境変数 SHC_LIB の設定でファイル名の規約に反した指定をしているか、パス名の長さが 118 文字を超えています。

4000 - 4999 (-) Internal error

コンパイラの内部処理で何らかの障害が生じました。本コンパイラをお求めになった営業所あるいは代理店にエラーの発生状況をご連絡ください。

5001 (W) Linkage specification ignored for 'static' functions and objects

リンケージ指定の { } の中で、static と宣言された関数やオブジェクトの extern リンケージ指令は無視されます。

6001 (E) Overloaded function "関数名" cannot specify #pragma function

多重定義関数を #pragma 指定できません。

6002 (E) Preprocessing numerical token is not floating constant value or integer constant value

前処理数値字句が浮動小数点定数、整数定数ではありません。

6101 (E) Illegal storage class 'auto'

auto 記憶クラスは、ブロック内か、仮引数内のオブジェクト宣言に指定しなければなりません。

6102 (E) Illegal storage class 'register'

register 記憶クラスは、ブロック内か、仮引数内のオブジェクト宣言に指定しなければなりません。

6103 (E) 'static' keyword must be applied to objects, functions and anonymous unions

static 記憶クラスは、オブジェクト名、関数名または名前なし共用体以外に指定できません。

- 6104 (E) Function declaration "関数名" cannot have 'static' storage in a block
ブロック内で static 関数を宣言することはできません。
- 6105 (E) Static linkage specifications for "名前" must agree
名前に対する全てのリンケ - ジ指定が一致していません。
- 6106 (E) Illegal storage class 'extern'
extern 記憶クラスは、オブジェクト名、関数名または名前なし共用体以外に指定できません。
- 6108 (E) Inline member function "関数名" must be declared before it is called
メンバ関数を呼んだ後でその関数を inline と宣言することはできません。
- 6109 (E) Illegal function specifier 'virtual'
Virtual 指定子は、クラス定義の非静的クラス・メンバ関数の宣言以外に指定できません。
- 6110 (E) Illegal 'typedef' declaration in function definition "名前"
typedef 指定子は、関数定義に指定できません。
- 6111 (E) 'typedef' "名前" cannot re-define other types in the same scope
同ースコープで宣言された型の名前を、別の型を参照するように再定義することはできません。
- 6112 (E) Cannot specify 'typedef' "名前" after 'enum'
typedef で定義された enum 型名を enum 接頭辞のあとで使用することはできません。
- 6114 (E) Integral type data is not assigned to an enumeration
整数型あるいは整数型への昇格の結果を列挙型オブジェクトに代入することはできません。
- 6116 (E) Undefined linkage-specification "文字列"
リンケージ指定に未定義の文字列が指定されました。
- 6117 (E) Multiple linkage-specification "名前"
関数やオブジェクトが複数のリンケージ指定を持つ時のリンケージ指定文字列が一致していません。

- 6118 (E) 'static' function "関数名" with linkage-specification
リンケージ指定された関数に static 指定の関数宣言があります。
- 6119 (E) Multiple 'C' linkage overloaded function "関数名"
多重定義関数の集合の中で C リンケージを持つものが複数あります。
- 6120 (E) Illegal 'void &' type
void 型へのリファレンスは指定できません。
- 6123 (E) Cannot specify a reference to &, to bit-fields, to array or to pointer type
リファレンス型、ビットフィールド、配列またはポインタ型に対してのリファレンス型は指定できません。
- 6124 (E) Initial value required for declaration & "名前"
リファレンス型宣言に初期値の指定がありません。
- 6128 (E) New type defined in a return type of a function or in a parameter
戻り値や仮引数型の中で、型を定義しています。
- 6129 (E) Non-static members or 'auto' variables cannot be used as default parameter
デフォルト引数に非静的メンバが使われています。
- 6130 (E) Default parameter re-defined in function "名前"
デフォルト引数は、後の宣言で再定義することができません。
- 6131 (E) Non-default parameters found following default parameters
デフォルト引数の後にデフォルト引数以外の引数が存在します。
- 6133 (E) Overloaded operators cannot have default parameters
多重定義演算子はデフォルト引数を持てません。
- 6139 (E) Overloaded functions "関数名" have the same type of parameters except & type
関数の引数の型がリファレンス型の違いだけなので多重定義できません。

6140 (E) Functions "関数名" have the same type of parameters except const/volatile type

関数の引数の型が const/volatile 型の違いだけなので多重定義できません。

6141 (E) Functions "関数名" have the same type of parameters except return type

関数の戻り値が型の違いだけなので多重定義できません。

6142 (E) Cannot overload function "関数名"

メンバ関数が static であるかどうかの違いだけなので多重定義できません。

6143 (E) Operator overloaded function "関数名" does not have correct parameters

演算子関数は、以下のいずれかでなければなりません。

(1) メンバ関数である。

(2) クラスまたは列挙の引数を 1 つ以上持つ。

(3) クラスまたは列挙へのリファレンスの引数を 1 つ以上持つ。

6144 (E) Operator overloaded function "=" is not a nonstatic member function

operator=() が非静的メンバ関数ではありません。

6145 (E) Operator overloaded function "(" is not a nonstatic member function

operator()() が非静的メンバ関数ではありません。

6146 (E) Operator overloaded function "[" is not a nonstatic member function

operator[]() が非静的メンバ関数ではありません。

6147 (E) Operator overloaded function "->" returns illegal type value

operator->()がクラスへのポインタ、または、operator->()を定義しているクラスのオブジェクト、あるいはリファレンスを返していません。

6148 (E) Operator overloaded function "->" is not a nonstatic member function

operator->()が非静的メンバ関数ではありません。

6149 (E) The second parameter of the postfix operator function should have type int

後置の operator++()または operator--()の第 2 引数が int 型ではありません。

6150 (E) 'const' object should have an initial value

const 型のオブジェクトは外部リンケージを持たないので初期値が必要です。

6151 (E) 'friend' should be specified in a class declaration

friend 指定子がクラス宣言外で使われています。

6152 (E) 'friend' keyword specified twice

friend 指定子が2度指定されています。

6153 (E) 'inline' keyword specified twice

inline 指定子が2度指定されています。

6154 (E) 'virtual' keyword specified twice

virtual 指定子が2度指定されています。

6155 (E) 'inline' must be specified for functions

inline 指定子が関数以外に指定されています。

6156 (E) Parameters cannot have 'inline' specifier

引数に inline 指定子が指定されています。

6157 (E) Cannot specify 'inline' and 'extern' together

inline と extern 指定子が同時に指定されています。

6158 (E) Object "名前" initialized with { } format

{ } 形式で初期化できるオブジェクトは、次のいずれかです。

(1) 配列

(2) 非公開/限定公開のメンバ、基底クラス、コンストラクタ、仮想関数のいずれも持たないクラス

6159 (E) 'typedef' cannot specify 'friend'

typedef と friend が合わせて指定されています。

6162 (E) Cannot declare 'typedef' name qualified by a class

typedef 宣言の名前をクラスで限定することはできません。

6163 (E) Missing the number of operator function parameters

多重定義演算子の引数の数が間違っています。

6165 (E) Operator member function cannot specify 'static'

多重定義演算子は静的メンバ関数になれません。

6166 (E) Operator function or conversion function has function type

多重定義演算子、変換関数が関数型ではありません。

6167 (E) Conversion function should be a nonstatic member function

変換関数が非静的メンバ関数ではありません。

6169 (E) A destructor should be a function type

デストラクタが関数型ではありません。

6170 (E) Non-member functions cannot specify 'const' or 'volatile'

メンバ関数以外では、関数自身に `const` または `volatile` 型修飾できません。

6171 (E) Declarations qualified by a class cannot specify storage class specifier

クラスで限定された宣言に記憶クラスは指定できません。

6172 (E) Cannot declare parameter declarations qualified by a class

引数宣言にクラスで限定された名前を指定しています。

6203 (E) Ambiguous name"名前"

関数名、オブジェクト名、または、型名が曖昧です。

6209 (E) Array "名前" does not have dimension size

配列が非静的メンバの型として使われていますが、全ての次元数が指定されていません。

6210 (E) Member declarator be omitted

クラスのメンバ宣言子を省略することはできません。

6211 (E) Non-virtual function "関数名" cannot specify "=0"

純粋指定子 `=0` 指定できません。純粋指定子は仮想関数の宣言でのみ使用できます。

6212 (E) Member "名前" cannot have the same name of that class

静的データメンバ、列挙子、名前なしの共用体メンバあるいは入れ子型がクラスと同じ名前を持っています。

6213 (E) Static member function cannot access "名前"

静的メンバ関数ができるのは、静的メンバ、列挙子、入れ子型だけです。

6219 (E) Static data member "名前" cannot exist in a local class

局所クラスでは、静的データメンバを宣言できません。

6221 (E) Union "名前" cannot have virtual functions

共用体は、仮想関数を持つことができません。

6222 (E) Union "名前" cannot have base classes

共用体は、基底クラスを持つことができません。

6223 (E) Union "名前" cannot be a base class

共用体を基底クラスとして使用することはできません。

6224 (E) Union member cannot have a constructor, destructor or class with a user defined operator =()

コンストラクタ、デストラクタまたはユーザ定義の代入演算子を持つクラスオブジェクトが共用体のメンバに存在します。

6225 (E) Union "名前" cannot have static data members

static データメンバが共用体のメンバに存在します。

6227 (E) Anonymous union must specify 'static'

static 指定のない大域的な名前無し共用体宣言が存在します。

6228 (E) Anonymous union cannot have private and protected members

名前無し共用体に private や protected メンバが存在します。

6229 (E) Anonymous union cannot have member functions

名前無し共用体に、関数メンバが存在します。

6233 (E) Nested class declaration cannot access "名前"

入れ子クラスの宣言では、それを囲むクラスからのオブジェクト、静的メンバ、列挙子以外を使用できません。

6234 (E) "名前" cannot be specified in a nested class declaration

局所クラス宣言内では、型名、静的変数、extern 変数、extern 関数、列挙子以外使用できません。

6236 (E) Member function "名前" should be defined in the local class declaration

局所クラス宣言内のメンバ関数をクラス内で定義していません。

6240 (E) Member "名前" defined more than once

クラスメンバが2度宣言されています。

6241 (E) Undeclared member "名前"

クラス宣言中に未定義のメンバ関数をクラス外でメンバ関数として定義しています。

6242 (E) "名前" declared multiple

クラス再評価時に名前が確定しません。

6243 (E) Undeclared base class "名前"

基底クラスに指定したクラス名が定義されていません。

6244 (E) Base class "名前" defined more than once in the derived class declaration

派生クラスの直接基底クラスが2度以上指定されています。

6246 (E) Virtual base class "名前" caasted to the derived class

仮想基底クラスを派生クラスへキャストすることはできません。

6248 (E) Returning type mismatch in virtual function "関数名"

基底クラスの仮想関数と派生クラスの仮想関数の戻り型が異なります。

6250 (E) Illegal 'static' virtual function "関数名"

仮想関数を static 指定できません。

6251 (E) Cannot create abstract class object "名前"

抽象クラスのオブジェクトを作成することはできません。

6254 (E) Implicit pure virtual function "関数名" call

純粋仮想関数は明示的に限定して呼び出さなければなりません。

- 6256 (E) "名前" is not a member of the class
指定されたクラスのメンバが定義されていません。
- 6263 (E) "名前" should be defined as a class or a struct
class または、struct 指定子で宣言した名前が、異なる指定子で定義されています。
- 6267 (E) Illegal declaration "名前" in type-specifier
クラス、列挙、typedef 名は、型指定並びの中で宣言できません。
- 6268 (E) "関数名" cannot have parameters and return value
変換関数に引数型や戻り型を指定できません。
- 6269 (E) Ambiguous user defined conversion
利用者定義変換が曖昧です。
- 6270 (E) Destructor "関数名" cannot have parameters and return value
デストラクタに引数や戻り型を指定できません。
- 6271 (E) Constructor and destructor cannot have 'const', 'volatile' or 'static' keywords
コンストラクタ、デストラクタに const、volatile、static を指定できません。
- 6272 (E) Constructor and destructor "関数名" do not have addresses
コンストラクタ、デストラクタのアドレスを求めることはできません。
- 6274 (E) The first parameter type of operator new() should be 'size_t'
クラスの operator new() 関数の第 1 引数は、整数型の size_t でなければなりません。
- 6275 (E) The return type of operator new() should be 'void *'
クラスの operator new() 関数の戻り型は、void * 型でなければなりません。
- 6276 (E) The first parameter type of operator delete() should be 'void *'
クラスまたは大域の operator delete() 関数の第 1 引数は、void * でなければなりません。
- 6277 (E) The return type of operator delete() should be 'void'
クラスの ::operator delete() の戻り型は void 型でなければなりません。

6278 (E) Operator delete() function cannot be overloaded

operator delete() は多重定義できません。

6280 (E) The second parameter type of operator delete() should be 'size_t'

クラスの operator delete() 関数の第2引数は、整数型の size_t でなければなりません。

6281 (E) Illegal virtual operator new() or delete() function

operator new()、operator delete() は、仮想関数指定できません。

6282 (E) Default constructor required

初期設定子ならびに対応する初期値がそろっていないか、初期値がない場合のデフォルトコンストラクタが定義されていません。

6283 (E) Class "名前" requires a constructor

仮想基底クラスの初期設定に誤りがあります。最派生クラスのコンストラクタが仮想基底クラスのメンバ初期設定子を指定していなければ、その仮想基底クラスはデフォルトコンストラクタを持つか、コンストラクタを持たないかのどちらかでなければなりません。

6284 (E) Undefined class member "名前"

クラスメンバの初期設定に誤りがあります。クラスメンバのオブジェクトは、コンストラクタを持たないか、デフォルトコンストラクタを持つか、あるいは、クラスメンバを宣言しているクラスのコンストラクタの中でクラスメンバを初期化しなければなりません。

6285 (E) Multiple implicit conversion

式の値に暗黙のうちに適用される利用者定義変換が複数存在します。

6286 (E) 'public' copy constructor "関数名" required

コピーコンストラクタを private メンバとして宣言しているのでアクセスできません。

6287 (E) Illegal nonstatic 'const' array initialization

非静的 const 配列のメンバをコンストラクタで初期設定することはできません。

6288 (E) Constructors cannot initialize indirect base classes or derived members

間接基底クラスや派生メンバをコンストラクタで初期設定することはできません。

6289 (E) Cannot generate default assignment operator

デフォルトの代入演算子は生成されません。クラスが `const` メンバ、リファレンスメンバ、非公開の `operator=()` を持つクラスのメンバ、あるいは、非公開の `operator=()` を持つ基底クラスのメンバを持っています。

6290 (E) Cannot generate default copy constructor "名前"

デフォルトのコピーコンストラクタは生成されません。クラスが名前なしクラスか、すでにクラスのメンバまたは基底クラスのメンバが非公開コピーコンストラクタを持っています。

デフォルトのコピーコンストラクタは生成されません。クラスのメンバまたは基底クラスのメンバが非公開コピーコンストラクタを持っています。

6291 (E) Cannot assign "名前"

基底クラスのオブジェクトを派生クラスのオブジェクトに代入することはできません。

6292 (E) Cannot access private member "名前"

非公開メンバを参照することはできません。

6293 (E) Cannot access protected member "名前"

限定公開メンバは参照することはできません。

6294 (E) Illegal access declaration "名前"

アクセス宣言において、基底クラスのメンバで宣言したアクセス指定を変更することはできません。

6296 (E) Cannot change the access control of overloaded function "関数名"

アクセス指定が同じでない多重定義関数をアクセス宣言で調整することはできません。

6297 (E) Cannot change the access control of redefined member "名前"

派生クラスにおいてメンバが再定義された場合、基底クラスのメンバアクセスを派生クラスで調整することはできません。

6298 (E) 'friend' declaration syntax error

フレンド宣言の中で、クラスを定義しようとしています。

6303 (E) Cannot access base class "クラス名"

その基底クラスにアクセスできません。

6304 (E) Cannot define 'friend' data members

データメンバを friend 宣言している。

6305 (E) Illegal pure virtual function specifier

純粋仮想関数指定に =0 以外を指定している。

6306 (E) Cannot access class "名前" member

アクセスできないクラスメンバを指定しています。

6307 (E) Cannot access base class member "名前"

基底クラスメンバをアクセスできません。

6308 (E) Constructors cannot have return type

コンストラクタにリターン型を指定している。

6310 (E) Cannot define class member "名前"

別クラスの中で別クラスのメンバを宣言しています。

6311 (E) Illegal constructor/destructor declaration

コンストラクタ/デストラクタの宣言に誤りがあります。

6313 (E) Cannot declare nonstatic data member "名前"

非静的データメンバをクラス外部で再宣言することはできません。

6314 (E) Illegal data member initializer

データメンバの初期値指定の方法が正しくありません。

6401 (E) 'this' should be referred to in a nonstatic member function

キーワード this は、非静的メンバ関数以外で参照することはできません。

6402 (E) "名前" does not exist in this file scope

:: 演算子に続く識別子がファイルスコープに存在しません。

6404 (E) "名前" is not a member

:: 演算子に続く名前がそのクラスのメンバではありません。

- 6407 (E) The type of the second operand of '?' is different from the third operand
?: 演算子の第2オペランドと第3オペランドの型が異なるクラスの場合は、共通の基底クラスを持たなければなりません。
- 6408 (E) Base class objects cannot be assigned to derived class objects
基底クラスのオブジェクトは、派生クラスのオブジェクトに代入できません。
- 6409 (E) '?' operands should have integral types
クラス・オブジェクト、リファレンスは使用できない。
- 6417 (E) Illegal type conversion
関数呼び出し形式の明示的型変換において、対応するコンストラクが宣言されていません。
- 6421 (E) Invalid 'new' operand
new 演算子のオペランドが正しくありません。
- 6425 (E) An array cannot be initialized in a 'new' operator
new 演算子のオペランドで指定された配列を、初期設定子を使って初期化することはできません。
- 6428 (E) Invalid 'delete' operand
delete 演算子のオペランドが正しくありません。
- 6430 (E) Cannot convert an object or data to a class object
コンストラクタや変換演算子が正しく宣言されていないため、オブジェクトや値をクラスオブジェクトに変換できません。
- 6431 (E) Cannot convert a virtual base class to a derived class
仮想基底クラスを派生クラスにキャストすることはできません。
- 6432 (E) Illegal explicit type conversion
メンバへのポインタを別のメンバへのポインタ型へ変換することはできません。
- 6433 (E) '->*' operands type mismatch
->* 演算子のオペランドの型が合致しません。

6434 (E) The second operand of '->*' and '.*' operator should be a pointer to member of a class

->* 演算子、.* 演算子の第 2 オペランドはクラスのメンバへのポインタ型でなければなりません。

6435 (E) '.*' operands type mismatch

.* 演算子のオペランドの型が合致しません。

6437 (E) Unary '&' operand require a qualified name

単項演算子 & のオペランドで限定した名前がクラスメンバ名ではありません。

6441 (E) 'typedef' "名前" used as a constructor or a destructor

typedef 宣言されたクラスの typedef 名をコンストラクタやデストラクタの名前として使用できません。

6442 (E) Cannot refer to undefined class

未定義のクラスは式中などで使用できません。

6446 (E) Ambiguous overloaded function call

多重定義関数呼び出しの関数が曖昧です。

6449 (E) Ambiguous function name referred to

多重定義関数のアドレスを示す関数名の式が曖昧です。

6450 (E) Illegal function pointer conversion

関数へのポインタ型を他の型に標準変換できません。

6451 (E) Undeclared function call

該当する関数が宣言されていません。

6452 (E) Pointer-to-Member cannot be used as an operand of a function call

メンバ関数へのポインタ型が関数呼び出し演算子以外のオペランドに使用できません。

6503 (E) Statement is required in selection statements or iteration statements

'if'、'switch' 文の中には 1 つ以上の文を含まなくてはなりません。または、反復文の文は宣言であってなりません。

6504 (E) Declaration with initialization is not executed

明示的あるいは暗黙の初期設定子を持った宣言を飛び越えている。

6508 (E) Illegal conditional expression

'while'、'do'、'for' 文の条件式は、算術型、ポインタ型または、算術型かポインタ型への型変換が存在するクラス型でなければなりません。

6513 (E) Illegal jump

不当なジャンプです。

6514 (E) "xxxx" is not supported

本バージョンでサポートしていない機能を使用しています。

7001 (E) Too many identifiers

識別子の数が制限値を超えています。

4.2 標準ライブラリのエラーメッセージ

ライブラリ関数の中には、ライブラリ関数を実行中にエラーが発生した場合、標準ライブラリのヘッダファイル<stdio.h>で定義しているマクロ `errno` にエラー番号を設定するものがあります。エラー番号には、対応するエラーメッセージが定義されており、エラーメッセージを出力することができます。エラーメッセージを出力するプログラム例を以下に示します。

[例]

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

main ( )
{
    FILE *fp;

    fp=fopen ( "file", "w" );
    fp=NULL;

    fclose ( fp );                      /* error occurred */

    printf ( "%s\n", strerror ( errno ) ); /* print error message */
}
```

[説明]

- (1) `fclose` 関数に値 `NULL` のファイルポインタを実引数として渡しているため、エラーとなります。このとき `errno` に対応するエラー番号が設定されます。
- (2) `strerror` 関数は、エラー番号を実引数として渡すと、対応するエラーメッセージの文字列のポインタを返します。`printf` 関数の文字列出力指定によりエラーメッセージを出力します。

・標準ライブラリエラーメッセージ一覧

エラー番号	エラーメッセージ / 説明	エラー番号を設定する関数
1100 (ERANGE)	Data out of range オーバーフローが発生しました。	atan, cos, sin, tan, cosh, sinh, tanh, exp, fabs, frexp, ldexp, modf, ceil, floor, strtol, atoi, fscanf, scanf, sscanf, atol
1101 (EDOM)	Data out of domain 数学関数の引数に対する結果の値が定義 されません。	acos, asin, atan2, log, log10, sqrt, fmod, pow
1102 (EDIV)	Division by zero ゼロによる除算を行っています。	divbs, divws, divls, divbu, divwu, divlu
1104 (ESTRN)	Too long string 文字列の長さが512文字を超えています。	strtol, strtod, atof, atoi, atol
1106 (PTRERR)	Invalid file pointer ファイルポインタの値に NULL ポインタ 定数を指定しています。	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf , scanf, sprintf, sscanf, vsprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 (ECBASE)	Invalid radix 基数の指定が誤っています。	strtol, atol, atoi
1202 (ETLN)	Number too long 数値を表現する文字列の長さが17桁を 超えています。	strtod, fscanf, scanf, sscanf, atof
1204 (EEXP)	Exponent too large 指数部の桁数が3桁を超えています。	strtod, fscanf, scanf, sscanf, atof

1206 (EEXPN)	Normalized exponent too large 文字列を一度 IEEE 規格 10 進形式に正規化したとき指数部の桁数が 3 桁をこえています。	strtod, fscanf, sscanf, atof
1210 (EFLOATO)	Overflow out of float float 型の 10 進数値が、float 型の範囲を超えています (オーバーフロー)。	strtod, fscanf, scanf, sscanf, atof
1220 (EFLOATU)	Underflow out of float float 型の 10 進数値が、float 型の範囲を超えています (アンダフロー)。	strtod, fscanf, scanf, sscanf, atof
1250 (EDBLO)	Overflow out of double double 型の 10 進数値が、double 型の範囲を超えています (オーバーフロー)。	strtod, fscanf, scanf, sscanf, atof
1260 (EDBLU)	Underflow out of double double 型の 10 進数値が、double 型の範囲を超えています (アンダフロー)。	strtod, fscanf, scanf, sscanf, atof
1270 (ELDBLO)	Overflow out of long double long double 型の 10 進数値が、long double 型の範囲を超えています (オーバーフロー)。	fscanf, scanf, fscanf
1280 (ELDBLU)	overflow out of long double long double 型の 10 進数値が、long double 型の範囲を超えています (アンダフロー)。	fscanf, scanf, sscanf
1300 (NOTOPN)	File not open ファイルがオープンされていません。	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen

4. エラーメッセージ

1302 (EBADF)	Bad file number 入力専用ファイルに対して出力関数、 あるいは出力専用ファイルに対して入 力関数を発行しています。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 (ECSPEC)	Error in format 書式付き入出力関数で指定している書式 が誤っています。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror

付録

付録 目次

A.	C++コンパイラが規定する言語仕様とCライブラリ関数仕様.....	147
	A.1 言語仕様.....	147
	A.2 Cライブラリ関数仕様.....	155
	A.3 浮動小数点数の仕様.....	160
B.	引数割り付けの具体例.....	168
C.	レジスタとスタック領域の使用法.....	170
D.	終了処理関数の作成例.....	171
	D.1 終了処理の登録と実行（onexit）ルーチンの作成例.....	171
	D.2 プログラムの終了（exit）ルーチンの作成例.....	172
	D.3 異常終了（abort）ルーチンの作成例.....	174
E.	低水準インタフェースルーチンの作成例.....	175
F.	ASCIIコード表.....	182
G.	CプログラムをC++コンパイラでコンパイルするときの注意事項.....	183
H.	索引.....	184
	H.1 日本語索引.....	184
	H.2 英語索引.....	189

付録 A. C++コンパイラが規定する言語仕様と C ライブラリ関数仕様

A.1 言語仕様

(1) 翻訳

表 A.1 翻訳の仕様

項番	項目	本コンパイラの仕様
1	エラー検出時のエラー情報	「第4章 エラーメッセージ」を参照。

(2) 環境

表 A.2 環境の仕様

項番	項目	本コンパイラの仕様
1	main 関数への実引数の意味	規定しません。
2	対話的入出力装置の構成	規定しません。

(3) 識別子

表 A.3 識別子の仕様

項番	項目	本コンパイラの仕様
1	外部結合とならない識別子（内部名）の有効文字数	外部 / 内部名ともに 250 文字までが有効です。しかし、関数や静的メンバ等は、識別子
2	外部結合となる識別子（外部名）の有効文字数	にクラス名、関数の仮引数情報を付加したものを外部 / 内部名とするので注意してください。
3	外部結合となる識別子（外部名）の大文字小文字の区別	大文字小文字を区別します。

【注】 250 文字目までが同じで、251 文字目以降が異なっている 2 つの識別子は、同じ識別子とみなされます。

(a) longabcde...ab; (250 文字が a、251 文字目が b)

(b) longabcde...ac; (250 文字が a、251 文字目が c)

(a) と (b) の 2 つの識別子は、250 文字目までが一致しているので、同じ識別子とみなします。

(4) 文字

表 A.4 文字の仕様

項番	項目	本コンパイラの仕様
1	ソース文字集合および実行環境文字集合の要素	ソース文字集合、実行環境文字集合ともに ASCII 文字集合です。 ただし、ソースプログラムのコメント内と文字列内にはホスト環境の日本語コードを記述できます。
2	多バイト文字のコード化で使用するシフト状態	シフト状態はサポートしていません。
3	プログラム実行時の文字集合の文字のビット数	ビット数は 8 ビットです。
4	文字定数内、文字列内のソース文字集合と実行環境文字集合の文字との対応付け	同じ ASCII 文字に対応します。
5	言語で規定していない文字や拡張表記を含む整数文字定数の値	言語で規定する以外の文字、拡張表記はサポートしていません。
6	2 文字以上の文字を含む文字定数または 2 文字以上の多バイト文字を含む広角文字定数の値	文字定数は上位 1 文字を有効とします。広角文字定数はサポートしていません。また、1 文字より多く指定した場合はウォーニングエラーを出力します。
7	多バイト文字を広角文字に変換するために使用される locale の仕様	locale はサポートしていません。
8	単なる char 型が signed char 型、unsigned char 型のどちらと同じ値の範囲を持つか	signed char 型と同じ値の範囲を持ちます。ただし、char と signed char は異なる型としてコンパイル処理を行います。

(5) 整数

表 A.5 整数の仕様

項番	項目	本コンパイラの仕様
1	整数型の表現方法とその値	表 A.6 に示します。 (負の数は2の補数で表現します)
2	整数の値がより短いサイズの符号付き整数型あるいは、符号付き char 型で表現できない値に変換されたときの値(結果の値が変換先の型で表現できない場合)	整数の値の下位2バイトあるいは下位1バイトが変換後の値となります。
3	符号付き整数に対するビットごとの演算の結果	符号付きの値とみなします。
4	整数除算における余りの符号	被除数の符号と同符号になります。
5	負の値を持つ符号付き汎整数型の右シフトの結果	符号ビットを保持します。

表 A.6 整数型とその値の範囲

項番	型	値の範囲	データサイズ
1	char	- 128 ~ 127	1 バイト
2	signed char	- 128 ~ 127	1 バイト
3	unsigned char	0 ~ 255	1 バイト
4	(signed) short (int)	- 32768 ~ 32767	2 バイト
5	unsigned short (int)	0 ~ 65535	2 バイト
6	(signed) int	- 2147483648 ~ 214783647	4 バイト
7	unsigned (int)	0 ~ 4294967295	4 バイト
8	(signed) long (int)	- 2147483648 ~ 2147483647	4 バイト
9	unsigned long (int)	0 ~ 4294967295	4 バイト

【注】 括弧内の型指定子は省略可能です。また、型ならび順序は規定されません。

(6) 浮動小数点数

表 A.7 浮動小数点数の仕様

項番	項目	本コンパイラの仕様
1	浮動小数点型の表現方法とその値	浮動小数点数型には、float 型、double 型、long
2	整数を本来の値に正確に表現することができない浮動小数点数に変換したときの切り捨て方向	double 型があります。浮動小数点数型の内部表現や変換仕様、演算仕様等の性質は、「A.3 浮動小数点数の仕様」で説明します。表 A.8
3	浮動小数点数をより狭い浮動小数点数に変換したときの切り捨てまたは丸めの方法	に、浮動小数点数型の表現可能な値の限界値を示します。

表 A.8 浮動小数点数の限界値

項番	項目	限界値	
		10 進数表現*1	16 進数表現
1	float 型の最大値	3.4028235677973364e+38f (3.4028234663852886e+38f)	7f7fffff
2	float 型の正の最小値	7.0064923216240862E.46f (1.4012984643248171E.45f)	00000001
3	double 型、long double 型の最大値	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffffff
4	double 型、long double 型の正の最小値	4.9406564584124655E.324 (4.9406564584124654E.324)	0000000000000001

【注】 *1 10 進表現の限界値は 0 また無限大にならない限界値です。また、() 内は理論値を表示します。

(7) 配列とポインタ

表 A.9 配列とポインタの仕様

項番	項目	本コンパイラの仕様
1	配列の大きさの最大値を保持するために必要な整数の型 (size_t)	unsigned long 型
2	ポインタ型から整数型への変換 (ポインタ型のサイズ 整数型のサイズ)	ポインタ型の下位バイトの値になります。
3	ポインタ型から整数型への変換 (ポインタ型のサイズ < 整数型のサイズ)	符号拡張します。
4	整数型からポインタ型への変換 (整数型のサイズ ポインタ型のサイズ)	整数型の下位バイトの値になります。
5	整数型からポインタ型への変換 (整数型のサイズ < ポインタ型のサイズ)	符号拡張します。
6	同じ配列内のメンバへのポインタ間の差を保持するために必要な整数の型 (ptrdiff_t)	int 型

(8) レジスタ

表 A.10 レジスタの仕様

項番	項目	本コンパイラの仕様
1	レジスタに割り付けることができるレジスタ変数の最大数	7 個
2	レジスタに割り付けることができるレジスタ変数の型	char、signed char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、ポインタ、リファレンス、データメンバへのポインタ

(9) クラス、列挙型、ビットフィールド

表 A.11 クラス、列挙型、ビットフィールドの仕様

項番	項目	本コンパイラの仕様
1	異なる型のメンバでアクセスされる共用型メンバの参照	参照はできますが、値は保証しません。
2	クラスメンバの境界調整	クラスメンバ中のデータサイズの最大値が境界調整数になります。また、派生クラスや仮想関数を持つクラスは、4 バイト調整になります。表 A.6 「整数型とその値の範囲」を参照してください。 ^{*1}
3	単なる int 型のビットフィールドの符号	signed int 型とします。
4	int 型のサイズ内のビットフィールドの割り付け順序	上位ビットから割り付けます。 ^{*2}
5	int 型のサイズ内にビットフィールドが割り付けられるとき、次に割り付けるビットフィールドのサイズが int 型内の残っているサイズを越えたときの割り付け方	次の int 型の領域に割り付けます。 ^{*2}
6	ビットフィールドで許される型指定子	char、signed char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long 型
7	列挙型の値を表現する整数値	int 型

【注】 *1 クラスメンバの割り付け方の詳細については「第 2 章 C++プログラミング 2.2.2(2) 利用者定義型」を参照してください。

*2 ビットフィールドの割り付け方の詳細については「第 2 章 C++プログラミング 2.2.2 (3) ビットフィールド」を参照してください。

(10) 修飾子

表 A.12 修飾子の仕様

項番	項目	本コンパイラの仕様
1	volatile データへのアクセスの種類	規定しません。

(11) 宣言

表 A.13 宣言の仕様

項番	項目	本コンパイラの仕様
1	基本型を修飾する型（ポインタ型、配列型、関数型、リファレンス型）	16 個まで指定できます。

(a) 基本型を修飾する型の数の数え方の例を以下に示します。

(i) `int a;` `a` は `int` 型（基本型）であり、基本型を修飾する宣言子の数は 0 個です。

(ii) `char *f();` `f` は `char` 型（基本型）へのポインタ型を返す関数型であり、基本型を修飾する宣言子の数は 2 個です。

(12) 文

表 A.14 文の仕様

項番	項目	本コンパイラの仕様
1	1 つの <code>switch</code> 文中で宣言できる <code>case</code> ラベルの数	511 個まで指定できます。

(13) プリプロセッサ

表 A.15 プリプロセッサの仕様

項番	項目	本コンパイラの仕様
1	条件コンパイルの定数式内の単一文字の文字定数と実行環境文字集合の対応	プリプロセッサ文の文字定数と実行環境文字集合は一致します。
2	インクルードファイルの読み込み方法	「<」、「>」で囲まれたファイルは include オプションで指定されたディレクトリから読み込みます。 複数ディレクトリを指定した場合は指定した順番に検索します。 ファイルが見つからない場合、環境変数 SHC_INC が指定するディレクトリ、システムディレクトリ (SHC_LIB) の順序で各ディレクトリを検索します。
3	二重引用符で囲まれたインクルードファイルのサポート有無	サポートします。インクルードファイルを現ディレクトリから読み込みます。現ディレクトリにない場合は、前項 2 の規則に従ってファイルを読み込みます。
4	#define 文の実引数の文字列が空白文字のとき展開された後の文字列の空白文字	空白文字列は、空白文字 1 文字として展開されます。
5	#pragma 文の動作	#pragma interrupt #pragma section #pragma inline #pragma inline_asm #pragma abs16 #pragma gbr_base #pragma gbr_base1 #pragma noregsave #pragma noregalloc #pragma regsave をサポートしています。 ^{*1}
6	__DATE__, __TIME__ の値	コンパイル開始時ホストマシンのタイムに基づく値が設定されます。

【注】 *1 #pragma の仕様については「第 2 章 C++プログラミング 2.3 拡張機能」を参照してください。

A.2 C ライブラリ関数仕様

C 言語仕様で規定されていない処理系定義の C ライブラリ関数仕様を以下に示します。

(1) stddef.h

表 A.16 stddef.h の仕様

項番	項目	本コンパイラの仕様
1	マクロ NULL の値	値 0 です。
2	ptrdiff_t の内容	int 型

(2) assert.h

表 A.17 assert.h の仕様

項番	項目	本コンパイラの仕様
1	assert 関数が出力する情報と終了動作	出力情報の形式を (a) に示します。情報を出力した後 abort 関数を呼び出して終了します。

(a) assert (式) において、式の値が 0 のとき以下のメッセージを出力します。

Assertion failed : <式> File <ファイル名>,Line <行番号>

(3) ctype.h

表 A.18 ctype.h の仕様

項番	項目	本コンパイラの仕様
1	isalnum 関数、isalpha 関数、iscntrl 関数、islower 関数、isprint 関数、isupper 関数で検査される文字集合	unsigned char 型で表現できる文字集合です。検査の結果真となる文字を表 A.19 に示します。

表 A.19 真となる文字の集合

項番	関数名	真となる文字
1	isalnum	'0' ~ '9'、'A' ~ 'Z'、'a' ~ 'z'
2	isalpha	'A' ~ 'Z'、'a' ~ 'z'
3	iscntrl	'¥X00' ~ '¥X1f'、'¥X7f'
4	islower	'a' ~ 'z'
5	isprint	'¥X20' ~ '¥X7E'
6	isupper	'A' ~ 'Z'

(4) math.h

表 A.20 math.h の仕様

項番	項目	本コンパイラの仕様
1	数学関数の入力パラメタ値が範囲を越えたときの数学関数が返す値	非数を返します。非数の形式については「A.3 浮動小数点の仕様」を参照ください。
2	数学関数でアンダフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmod 関数で第 2 実引数が 0 の場合の動作	非数を返し、定義域エラーとなります。

math.h には、ライブラリのエラー番号の値を示すマクロ ENUM、ERANGE が定義されています。

(5) setjmp.h

表 A.21 setjmp.h の仕様

項番	項目	本コンパイラの仕様
1	setjmp 関数の呼び出しが許されるプログラムの文脈	setjmp () または、ver = setjmp () の形式で、単独の文や、if 文、while 文、do 文、for 文の条件を示す式あるいは、switch 文、return 文の式に指定したときに保証されます。

(6) stdio.h

表 A.22 stdio.h の仕様

項番	項目	本コンパイラの仕様
1	入力テキストの最終の行が終了を示す改行文字を必要とするかどうか	規定しません。低水準インタフェースルーチンの仕様によります。
2	改行文字の直前に書き出された空白文字は、読み込み時に読み込まれるかどうか	
3	バイナリファイルに書かれたデータに付加されるヌル文字の数	
4	追加モード時のファイル位置指定子の初期値	
5	テキストファイルへの出力によってそれ以降のファイルのデータが失われるかどうか	
6	ファイルのバッファリングの仕様	
7	長さ 0 のファイルが存在するかどうか	
8	正当なファイル名の構成規則	
9	同時に同じファイルをオープンできるかどうか	
10	fprintf 関数における%p 書式変換の出力形式	16 進数出力となります。
11	fscanf 関数における%p 書式変換の出力形式 fscanf 関数での変換文字「-」の意味	16 進数出力となります。 先頭、最後あるいは「^」の直後でない場合、直前の文字と直後の範囲を示します。
12	fgetpos, ftell 関数で設定される errno の値	fgetpos 関数はサポートしていません。 ftell 関数については規定しません。低水準インタフェースルーチンの仕様によります。
13	perror 関数が生成するメッセージの出力形式	メッセージの出力形式を (a) に示します。
14	calloc、malloc、realloc 関数でサイズが 0 の時の動作	0 バイトの領域を割り付けます。

(a) perror 関数の出力形式は、

< 文字列 > : < error に設定したエラー番号に対応するエラーメッセージ >
となります。

(b) printf 関数、fprintf 関数等で浮動小数点数の無限大および非数を表示するときの形式を表 A.23 に示します。

表 A.23 無限大および非数の表示形式

項番	項目	本コンパイラの仕様
1	正の無限大	+++++
2	負の無限大	-----
3	非数	*****

(7) string.h

表 A.24 string.h の仕様

項番	項目	本コンパイラの仕様
1	memcmp 関数、strcmp 関数、strncmp 関数の 処理において返される値の符号	符号付きとして扱います。
2	strerror 関数が返すエラーメッセージの内容	「第4章 エラーメッセージ 4.2 標準ライ ブラリのエラーメッセージ」を参照してくだ さい。

(8) errno.h

表 A.25 error.h の仕様

項番	項目	本コンパイラの仕様
1	errno	int 型変数、ライブラリ関数においてエラーが発生したときにエラー番号が設定される。
2	ERANGE	「第4章 エラーメッセージ 4.2 標準ライブラリのエラーメッセージ一覧」を参照ください。
3	EDOM	
4	EDIV	
5	ESTRN	
6	PTRERR	
7	ECBASE	
8	ETLN	
9	EEXP	
10	EEXPN	
11	EFLOATO	
12	EFLOATU	
13	EDBLO	
14	EDBLU	
15	ELDBLO	
16	ELDBLU	
17	NOTOPN	
18	EBADF	
19	ECSPEC	

A.3 浮動小数点数の仕様

(1) 浮動小数点数の内部表現

本コンパイラで扱う浮動小数点数の内部表現は、IEEE の標準形式に従っています。
ここでは、IEEE 形式の浮動小数点数の内部表現の概要について述べます。

(a) 内部表現の形式

float 型は IEEE 単精度形式 (32 ビット)、double 型と long double 型は IEEE の倍精度 (64 ビット) で表現します。

(b) 内部表現の構成

float 型、double 型および long double 型の内部表現の構成を図 A.1 に示します。

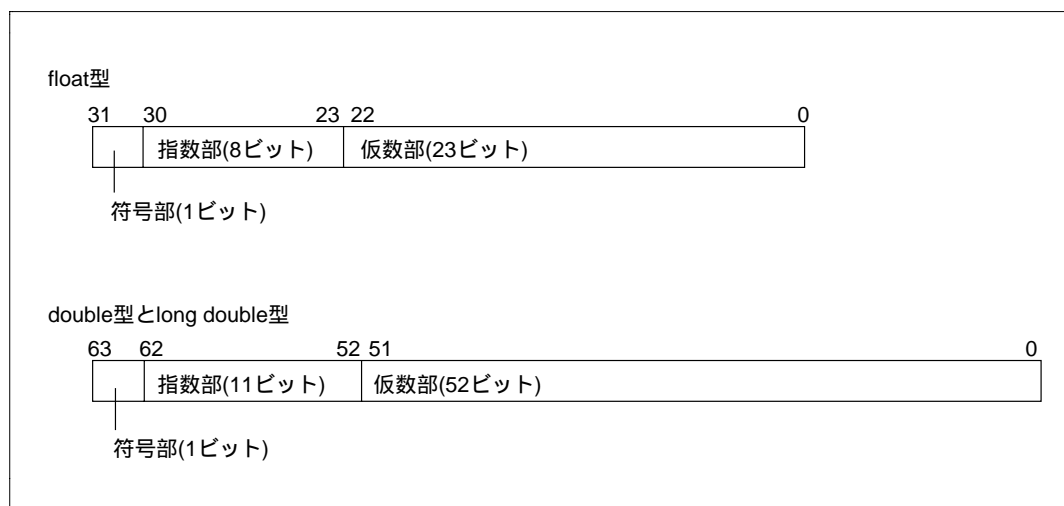


図 A.1 浮動小数点数の内部表現の構成

内部表現の各構成要素の意味を以下に示します。

(i) 符号部

浮動小数点数の符号を示します。0 のとき正、1 のとき負を示します。

(ii) 指数部

浮動小数点数の指数を 2 のべき乗で示します。

(iii) 仮数部

浮動小数点数の有効数字に対応するデータです。

(c) 表現する値の種類

浮動小数点数は、通常の実数値のほかに、無限大等の値も表現することができます。浮動小数点数が表現する値の種類を以下に示します。

(i) 正規化数

指数部が0または全ビット1ではない場合です。通常の実数値を表現します。

(ii) 非正規化数

指数部が0で、仮数部が0でない場合です。絶対値の小さな実数値を表現します。

(iii) ゼロ

指数部および仮数部が0の場合です。値0.0を表現します。

(iv) 無限大

指数部が全ビット1で仮数部が0の場合です。無限大を表現します。

(v) 非数

指数部が全ビット1で仮数部が0でない場合です。「0.0/0.0」、「 / 」、「 - 」等、結果が数値または無限大に対応しない演算の結果として得られます。

【注】非正規化数は、正規化数で表現できない範囲の絶対値の小さな浮動小数点数を表現しますが、正規化数に比較して有効桁数が少なくなっています。したがって、演算の結果、あるいは途中結果が非正規化数となる場合、結果の有効桁数は保証されませんので注意してください。

表 A.26 浮動小数点数を表現する値の種類

指数部 仮数部	0	0でも全ビット1でもない	全ビット1
0	0	正規化数	無限大
0以外	非正規化数		非数

(2) float 型

float型の内部表現は、1 ビットの符号部、8 ビットの指数部、23 ビットの仮数部からなります。

(i) 正規化数

符号部は、0（正）または1（負）で、値の符号を示します。

指数部は、 $1 \sim 254 (2^8 - 2)$ の値をとります。実際の指数は、この値から 127 を引いた値で、その範囲は $-126 \sim 127$ です。

仮数部は、 $0 \sim 2^{23} - 1$ の値をとります。実際の仮数は、 2^{23} のビットを 1 と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$$(-1)^{(\text{符号部})} \times 2^{(\text{指数部}) - 127} \times (1 + (\text{仮数部}) \times 2^{-23})$$

となります。

[例]

[illegible]

符号： -

指数： $10000000_{(2)} - 127 = 1$ $_{(2)}$ は 2 進数を表します。

仮数: $1.11_{(2)} = 1.75$

值 : $-1.75 \times 2^1 = -3.5$

(ii) 非正規化数

符号部は0（正）または1（負）で値の符号を示します。

指数部は0で、実際の引数は - 126 になります。

仮数部は、 $1 \sim 2^{23} - 1$ で、実際の仮数は、 2^{23} のビットを0と仮定し、その直後に小数点があるものとして解釈します。

非正規化数を表現する値は、

$$(-1)^{(\text{符号部})} \times 2^{(\text{指数部}) - 126} \times ((\text{仮数部}) \times 2^{-23})$$

〔例〕

31 30 23 22 0

0 0 0 0 0 0 0 0 0 1 1 0

符号： +

指数： $0_{(2)} - 126 = -126$

仮数： $0.11_{(2)} = 0.75$ $_{(2)}$ は 2 進数を表します。

值 : 0.75×2^{-126}

(iii) ゼロ

符号部は0（正）または1（負）で、それぞれ+0.0、-0.0を示します。

指数部、仮数部はともに0です。

+0.0、-0.0 は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては「A.3 (4) 浮動小数点演算の仕様」を参照してください。

(iv) 無限大

符号部は0（正）または1（負）で、それぞれ+、-を示します。

指数部は $255 (2^8 - 1)$ です。

仮数部は0です。

(v) 非数

指数部は $255 (2^8 - 1)$ です。

仮数部は 0 以外の値です。

【注】 非数の符号、および仮数フィールドの(0以外の)値については規定していません。

(3) double 型と long double 型

double 型と long double 型の内部表現は、1 ビットの符号部、11 ビットの指数部、52 ビットの仮数部からなります。

(i) 正規化数

符号部は0（正）または1（負）で、値の符号を示します。

指数部は $1 \sim 2046 (2^{11} - 2)$ の値をとります。実際の指数は、この値から 1023 を引いた値で、その範囲は $-1022 \sim 1023$ です。

仮数部は $0 \sim 2^{52} - 1$ の値となります。実際の仮数は、 2^{52} のビットを 1 と仮定し、その後、小数点があるものとして解釈します。

正規化数の表現する値は、

$$(-1)^{(\text{符号部})} \times 2^{(\text{指数部}) - 1023} \times (1 + (\text{仮数部}) \times 2^{-52})$$

となります。

〔例〕

[illegible]

符号： +

指数：1111111111₍₂₎ - 1023 = 1 (2) は2進数を表します。

仮数: $1.111_{(2)} = 1.875$

值 : $1.875 \times 2^0 = 1.875$

符号部は0（正）または1（負）で、値の符号を示します。

指数部は0で、実際の指数は - 1022 になります。

仮数部は、 $1 \sim 2^{52} - 1$ で実際の仮数は、 2^{52} のビットを 0 と仮定し、その直後に小数点があるものとして解釈します。

となります。

[illegible]

值 : $0.875 \times 2^{-1022} = 1.875$

符号部が0（正）または1（負）で、それぞれ+0.0、-0.0を示します。
指数部、仮数部は、ともに0です。
+0.0、-0.0は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては「A.3（4）浮動小数点演算の仕様」を参照してください。

仮数部は0です。

【注】 非数の符号、および仮数部の（0以外の）値については規定していません。

(4) 浮動小数点演算の仕様

本項では、C++言語の機能として実現されている浮動小数点数の四則演算、およびコンパイル時やライブラリの処理で生じる浮動小数点数の10進表現と内部表現の間の変換の仕様について解説します。

(a) 四則演算の仕様

(i) 結果の値の丸め方

浮動小数点数の四則演算の結果の正確な値が、内部表現の仮数の有効数字を越えた場合は、以下の規則に従って丸めを行います。

- (ア) 結果の値は、その値を近似する2つの浮動小数点数の内部表現のうち、近い方に向かって丸められます。
- (イ) 結果の値が、その値を近似する2つの浮動小数点数のちょうど中央になる場合は、仮数の最後の桁が0となる方向に丸められます。

(ii) オーバフロー、アンダフロー、無効演算の時の処置

実行時のオーバフロー、アンダフロー、無効演算に対しては、以下の処置を行います。

- (1) オーバフローの場合は、結果の符号に従って正または負の無限大になります。
- (2) アンダフローの場合は、結果の符号に従って正または負のゼロになります。
- (3) 無効演算は、符号が逆の無限大を加算した場合、符号が同じ無限大を減算した場合、ゼロと無限大を乗算した場合、ゼロをゼロで、あるいは無限大を無限大で除算した場合に生じます。これらの場合、結果は非数になります。
- (4) 浮動小数点数から整数へ変換したときにオーバフローが生じた場合、結果の値は保障されません。
- (5) 上記のいずれの場合も、エラーの発生を示す変数 `errno` に対応するエラーの番号を設定します。この番号については、「第4章 エラーメッセージ 4.2 Cライブラリのエラーメッセージ」を参照してください。
エラーチェックが必要な場合は、この `errno` の値によってエラーの発生を判定してください。

【注】 定数式に関しては、コンパイル時に演算を行います。この時にオーバフロー、アンダフロー、無効演算を検出した場合は、ウォーニングレベルのエラーになります。

(iii) 特殊値の演算に関する注意事項

以下、特殊な値（ゼロ、無限大、非数）の演算に関する注意事項を述べます。

- (1) 正のゼロと負のゼロの和は正のゼロとなります。
- (2) 同符号のゼロの差は正のゼロになります。
- (3) 被演算子の一方あるいは両方に非数を含む演算の結果は、常に非数になります。
- (4) 比較演算においては、正のゼロと負のゼロは等しいものとして扱います。
- (5) 被演算子の一方あるいは両方が非数であるような比較演算、等値演算の結果は、「!=」については常に真、その他は常に偽となります。

(b) 10 進表現と内部表現の間の変換

本項ではソースプログラム上の浮動小数点数と内部表現の間の変換、あるいは C ライブラリ関数による ASCII 文字列による浮動小数点数の 10 進表現と内部表現の間の変換の仕様について解説します。

- (i) 10 進表現から内部表現に変換する場合、まず 10 進表現を 10 進表現の正規形に変換します。10 進表現の正規形は、「 $\pm M \times 10^{\pm N}$ 」の形式で、M、N の範囲は以下の通りです。

(1) float 型の正規形

0 M $10^9 - 1$

0 N 99

(2) double 型、long double 型の正規形

0 M $10^{17} - 1$

0 N 999

正規形に変換できない 10 進表現については、オーバフロー、またはアンダフローになります。また、10 進表現が、正規形よりも、多くの有効数字を含んでいる場合は下位の桁は切り捨てます。これらの場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数 `errno` に設定します。

また、正規形に変換するためには、もとの 10 進表現の ASCII 文字列としての長さが 511 文字以下でなければなりません。そうでない場合、コンパイル時にはエラーになり、実行時には対応するエラーの番号を変数 `errno` に設定します。

内部表現から 10 進表現に変換する場合には、一度 10 進表現の正規形に変換してから、指定した書式に従って ASCII 文字列に変換します。

(ii) 10 進表現の正規形と内部表現の間の変換

10 進表現の正規形と内部表現の間の変換は、指数が大きい時や小さい時には、処理の効率上、誤差を回避することができません。以下に、正確な変換ができる範囲とその範囲外の場合の誤差の限界値について解説します。

(イ) 正確な変換ができる範囲

以下に示す指数の範囲の浮動小数点数については、「(a)(i) 結果の値の丸め方」に示す丸めが正確に行われます。この範囲ではオーバーフロー、アンダフローは生じません。

(1) float 型の場合: $0 \leq M \leq 10^9 - 1$, $0 \leq N \leq 13$

(2) double 型、long double 型の場合: $0 \leq M \leq 10^{17} - 1$, $0 \leq N \leq 27$

(ロ) 誤差の限界値

(イ) で示す範囲に入っていない値を変換する場合の誤差と正確な丸めを行った時の誤差の差は、有効数字の最小位桁の 0.47 倍を超えません。

また、(イ) で示した範囲を超えている場合、変換の際にオーバーフローやアンダフローが生じる場合があります。この場合、コンパイル時にはウォーニングレベルのエラーとなり、実行時には対応するエラーの番号を変数 `errno` に設定します。

付録 B. 引数割り付けの具体例

例 1. レジスタ渡しの対象の型である引数は、宣言順にレジスタ R4 ~ R7 に割り付けます。

<code>int f(char, short, int, float);</code>	R 4	符号拡張	1
<code>:</code>	R 5	符号拡張	2
<code>f(1, 2, 3, 4.0);</code>	R 6	3	
	R 7	4.0	

例 2. レジスタに割り付けることができなかった引数は、スタックに割り付けます。また、引数の型が (unsigned/signed) char 型、または、(unsigned) short 型でスタック上の引数領域に割り付く場合、4 バイトに拡張して割り付けます。

<code>int f(int, short, long, float, char);</code>	R 4	1
<code>:</code>	R 5	符号拡張 2
<code>f(1, 2, 3, 4.0, 5);</code>	R 6	3
	R 7	4.0

引数領域 (スタック)	符号拡張	5
	下位アドレス	
	上位アドレス	

例 3. レジスタに割り付けられない型の引数は、スタックに割り付けます。

<code>struct s{int x,y;}a;</code>	R 4	1
<code>int f(int, struct s, int);</code>	R 5	3
<code>:</code>		
<code>f(1, a, 3);</code>		

引数領域 (スタック)	a.x	下位アドレス
	a.y	
		上位アドレス

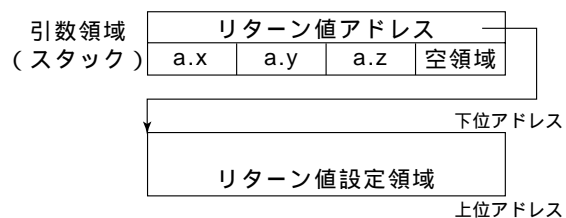
例 4. プロトタイプ宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。

<code>int f(double, int, int...)</code>	R 4	2
<code>:</code>		
<code>f(1.0, 2, 3, 4)</code>		

引数領域 (スタック)	1.0	下位アドレス
	3	
	4	
		上位アドレス

例 5. 関数の返す型が 4 バイトをこえる場合またはクラスの場合、引数領域の直前にリターン値アドレスを設定します。また、クラスのサイズが 4 の倍数バイトでないと、空領域が生じます。

```
struct s{char x,y,z;}a;  
double f(struct s);  
:  
f(a);
```



付録 C. レジスタとスタック領域の使用法

C++コンパイラのレジスタ、スタック領域の使用法を示します。

関数内でのレジスタ、スタック領域はすべてC++コンパイラが操作しますので、ユーザーが特にこの領域の使用方法に留意する必要はありません。

レジスタとスタック領域の使用法を図 C.1 に示します。

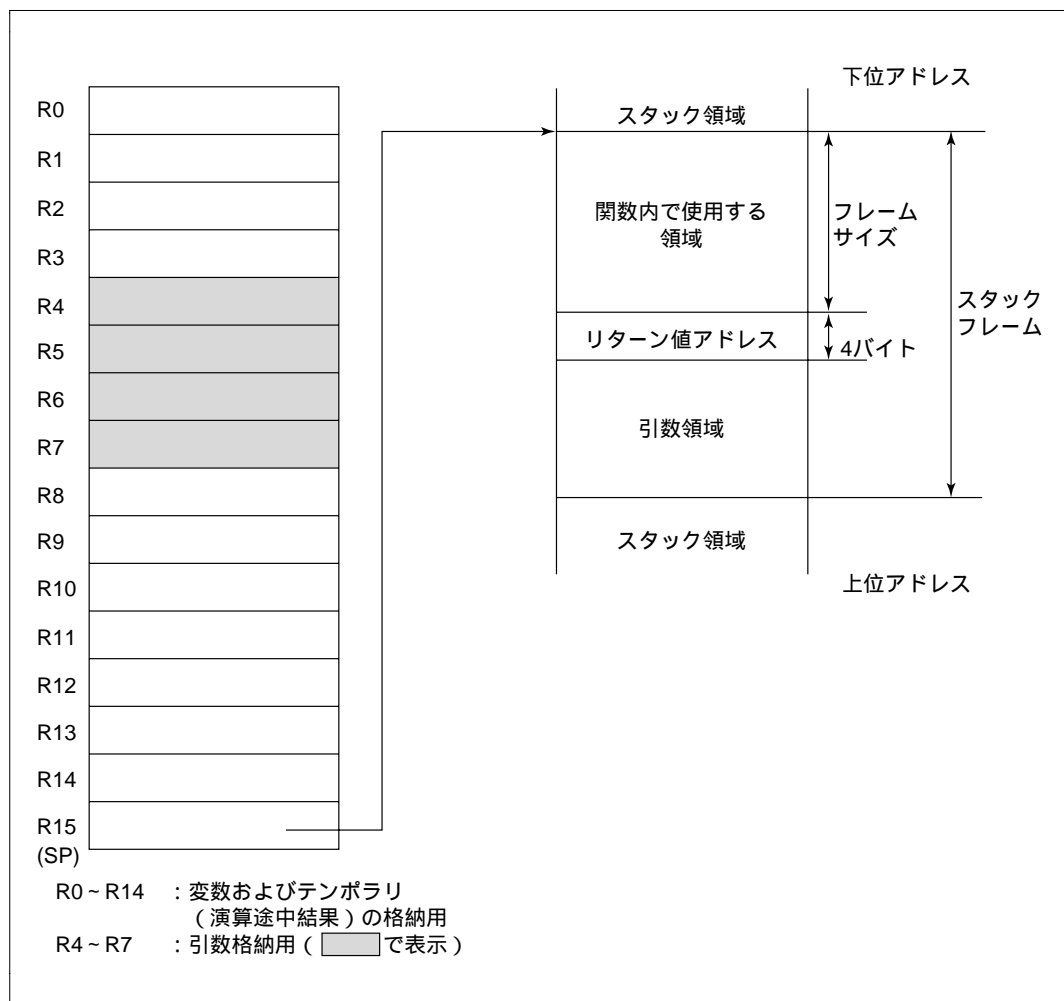


図 C.1 レジスタとスタック領域の使用法

付録 D. 終了処理関数の作成例

D.1 終了処理の登録と実行 (onexit) ルーチンの作成例

終了処理の登録を行うライブラリ onexit 関数の作成例を示します。

onexit 関数では、引数として渡された関数のアドレスを終了処理テーブルに登録します。登録された関数の個数が限界値（ここでは、登録できる関数の個数を 32 個とします）を越えた場合、あるいは、同じ関数が二度以上登録された場合はリターン値として NULL を返します。そうでなければ NULL 以外の値（この場合は、関数を登録したアドレス）を返します。

以下にプログラム例を示します。

[例]

```
#include <stdlib.h>
typedef void *onexit_t;

int _onexit_count = 0;
onexit_t (*_onexit_buf[32]) (void);

extern "C" onexit_t onexit (onexit_t (*) (void));

extern "C" onexit_t onexit (onexit_t (*f) (void))
{
    int i;

    for ( i = 0; i < _onexit_count; i++ )
        if ( _onexit_buf[i] == f )          //既に登録されていないかチェック
            return NULL;
    if ( _onexit_count == 32 )              //登録数の限界値チェック
        return NULL;
    else {
        _onexit_count++;
        _onexit_buf[ _onexit_count ] = f; //関数のアドレスを登録
        return &_onexit_buf[_onexit_count];
    }
}
```

D.2 プログラムの終了 (exit) ルーチンの作成例

プログラムの終了処理を行うライブラリ exit 関数の作成例を示します。プログラムの終了処理は、ユーザシステムによって異なりますので、以下のプログラム例を参考にユーザシステムの仕様に従った終了処理を作成してください。

exit 関数は、引数として渡されたプログラムの終了コードに従って C++ プログラムの終了処理を行い、プログラム起動時の環境に戻ります。ここでは、終了コードを外部変数に設定して、main 関数を呼び出す直前に setjmp 関数で退避した環境に戻ることによって実現します。

以下にプログラム例を示します。

[例]

```
#include <setjmp.h>
#include <stddef.h>

typedef void *onexit_t;
extern int _onexit_count;
extern onexit_t ( *_onexit_buf[32] ) ( void );

extern jmp_buf _init_env;
extern int _exit_code;

extern "C" {
    extern void _CLOSEALL ( );
    extern void exit ( int );
}

extern "C" void exit ( int code )
{
    ::_exit_code = code;          //_exit_code にリターンコードを設定
    for ( int i = _onexit_count-1; i > 0; i-- )
        ( *_onexit_buf[i] ) ( ); //onexit 関数で登録した関数を順次実行

    _CLOSEALL ( );               //オープンした関数をすべてクローズ

    longjmp ( _init_env, 1 );    //setjmp で退避した環境にリターン
}
```

【注】 上記関数で、プログラム実行前の環境に戻るためには、次の関数「callmain」を作成し、初期化ルーチン「init」から関数「main」を呼び出す代わりに関数「callmain」を呼び出してください。

```
#include <setjmp.h>
jmp_buf _init_env;
int      _exit_code;
extern "C" main();

void callmain()
{
    //setjmpを用いて現在の環境を退避し、main関数を呼び出します。
    //exit関数からのリターン時には処理を終了します。

    if(!setjmp(_init_env))
        _exit_code = main();
}
```

D.3 異常終了 (abort) ルーチンの作成例

異常終了の場合は、ご使用になっているユーザシステムの仕様に従ってプログラムを異常終了させる処理を行ってください。

以下、標準出力装置にメッセージを出力したあと、ファイルをクローズしてから無限ループしてリセットを待つプログラム例を示します。

[例]

```
#include <stdio.h>

extern "C" void abort ( ) ;
extern "C" void _CLOSEALL ( ) ;

extern "C" void abort ( )
{
    printf ( "program is abort !!¥n" ) ;           //メッセージの出力
    _CLOSEALL ( ) ;                               //ファイルのクローズ
    while ( 1 ) ;                                  //無限ループ
}
```

付録 E. 低水準インタフェースルーチンの作成例

```

/*****
//
//                               lowsrc.c:
// - - - - -
//          SHシリーズ シミュレータ・デバッガ インタフェースルーチン
//          - 標準入出力(stdin, stdout, stderr)だけをサポートしています -
/*****
#include <string.h>

// ファイル番号

#define STDIN  0                // 標準入力      (コンソール)
#define STDOUT 1                // 標準出力      (コンソール)
#define STDERR 2                // 標準エラー出力(コンソール)

#define FLMIN  0                // 最小のファイル番号
#define FLMAX  3                // ファイル数の最大値

// ファイルのフラグ

#define O_RDONLY 0x0001        // 読み込み専用
#define O_WRONLY 0x0002        // 書き込み専用
#define O_RDWR  0x0004        // 読み書き両用

// 特殊文字コード

#define CR 0x0d                // 復帰
#define LF 0x0a                // 改行

// sbrkで管理する領域サイズ

#define HEAPSIZE 1024

/*****
// 参照関数の宣言：
// シミュレータ・デバッガでコンソールへの文字入出力を行うアセンブリプログラムの参照
/*****

extern "C" void charput(char);    // 一文字入力処理
extern "C" char charget(void);    // 一文字出力処理

```

```
/* **** */
// 静的変数の定義：
// 低水準インタフェースルーチンで使用する静的変数の定義
/* **** */

char flmod[FLMAX]; // オープンしたファイルのモード設定場所

static union {
    long dummy; // 4バイト境界にするためのダミー
    char heap[HEAPSIZE]; // sbrk で管理する領域の宣言
} heap_area;

static char *brk=(char*)&heap_area; // sbrk で割り付けた領域の最終アドレス

/* **** */
// open：ファイルのオープン
// リターン値：ファイル番号（成功）
// -1（失敗）
/* **** */
extern "C" int open(char *name, // ファイル名
    int mode) // ファイルのモード
{ // ファイル名に従ってモードをチェックし、ファイル番号を返す

    if(strcmp(name,"stdin")==0){ // 標準入力ファイル
        if((mode&O_RDONLY)==0)
            return -1;
        flmod[STDIN]=mode;
        return STDIN;
    }

    else if(strcmp(name,"stdout")==0){ // 標準出力ファイル
        if((mode&O_WRONLY)==0)
            return -1;
        flmod[STDOUT]=mode;
        return STDOUT;
    }

    else if(strcmp(name,"stderr")==0){ // 標準エラー出力ファイル
        if((mode&O_WRONLY)==0)
            return -1;
        flmod[STDERR]=mode;
        return STDERR;
    }

    else
        return -1; // エラー
}
```

```

/*****
//  close: ファイルのクローズ
//      リターン値: 0          (成功)
//              -1          (失敗)
/*****
extern "C" int close(int fileno)      // ファイル番号
{
    if(fileno<FLMIN || FLMAX<fileno)  // ファイル番号の範囲チェック
        return -1;

    flmod[fileno]=0;                  // ファイルのモードリセット
    return 0;
}

/*****
//  read: データの読み込み
//      リターン値: 実際に読み込んだ文字数 (成功)
//              -1          (失敗)
/*****
extern "C" int read(int fileno,      // ファイル番号
    char *buf,                      // 転送先バッファアドレス
    unsigned int count)              // 読み込み文字数
{
    unsigned int i;

    // ファイル名に従ってモードをチェックし、一文字ずつ入力してバッファに格納

    if(flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR){
        for(i=count; i>0; i--){
            *buf=charget();
            if(*buf==CR)                // 改行文字の置き換え
                *buf=LF;
            buf++;
        }
        return count;
    }
    else
        return -1;
}

```



```

/*****
//  write: データの書き出し
//      リターン値: 実際に書き出した文字数 (成功)
//      -1 (失敗)
*****/
extern "C" int write(int  fileno,      // ファイル番号
                    char *buf,        // 転送元バッファアドレス
                    unsigned int  count) // 書き出し文字数
{
    unsigned int  i;
    char c;

    // ファイル名に従ってモードをチェックし、一文字ずつ出力

    if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
        for(i=count; i>0; i--){
            c=*buf++;
            charput(c);
        }
        return count;
    }
    else
        return -1;
}

/*****
//  lseek: ファイルの読み込み / 書き出し位置の設定
//      リターン値: 読み込み / 書き出し位置のファイル先頭からのオフセット (成功)
//      -1 (失敗)
//      (コンソール入出力では、lseekはサポートしていません)
*****/
extern "C" long lseek(int  fileno,      // ファイル番号
                     long offset,      // 読み込み / 書き出し位置
                     int  base)        // オフセットの起点
{
    return -1;
}
```

```
/* **** */
// sbrk: データの書き出し
//      リターン値: 割り付けた領域の先頭アドレス (成功)
//      -1 (失敗)
/* **** */
extern "C" char *sbrk(unsigned long size) // 割り付ける領域のサイズ
{
    char *p ;

    if (brk+size>heap_area.heap+HEAPSIZE) // 空き領域のチェック
        return (char *)-1 ;

    p=brk ; // 領域の割り付け
    brk += size ; // 最終アドレスの更新
    return p ;
}
```

```

;-----
;          lowlvl.src
;-----
;          SH SERIES SIMULATOR DEBUGGER INTERFACE ROUTINE
;          一文字入出力を行います
;-----
          .EXPORT      _charput
          .EXPORT      _charget
SIM_IO:   .EQU          H'0080          ;TRAP_ADDRESSの指定
          .SECTION     P, CODE, ALIGN=4
;-----
;          _charput:一文字出力
;          C++ program interface : extern "C" void charput(char);
;-----
          _charput:
MOV.L     A_DATA, r0          ; データの設定
MOV.B     R4, @R0
MOV.L     A_PARM, R1          ; パラメタブロックアドレスの設定
MOV.L     R0, @R1             ; データバッファの先頭アドレス設定
MOV.L     A_FNO, R0           ; ファイル番号の設定
MOV.B     @R0, R0
MOV.B     R0, @(1, R1)
MOV.L     F_putc, R0          ; 機能コードの設定
MOV.L     N_IO, R2
JSR       @R2
NOP
RTS
NOP
;-----
;          _charget:一文字入力
;          C++ program interface : extern "C" void charget(void);
;-----
          _charget:
MOV.L     A_PARM, R1          ; パラメタブロックアドレスの設定
MOV.L     A_DATA, R0          ; データバッファの先頭アドレス設定
MOV.L     R0, @R1
MOV.L     A_FNO, R0           ; ファイル番号の設定
MOV.B     @R0, R0
MOV.B     R0, @(1, R1)
MOV.L     F_getc, R0          ; 機能コードの設定
MOV.L     N_IO, R2
JSR       @R2

```

```

NOP
MOV.L      A_PARM,R1          ;データの参照
MOV.L      @R1,R0
MOV.B      @R0,R0
RTS
NOP

.ALIGN      4
A_DATA:    .DATA.L DATA      ;データバッファの先頭のアドレス
A_PARM:    .DATA.L PARM        ;パラメタブロックアドレス
A_FNO:     .DATA.L FILENO     ;ファイル番号領域のアドレス
F_putc:    .DATA.l H'01220000 ;putcの機能番号
F_getc:    .DATA.l H'01210000 ;getcの機能番号
N_IO:      .DATA.l SIM_IO     ;トラップアドレス

;-----
;      bufferの定義
;-----

.SECTION    B,DATA,ALIGN = 4
PARM:      .RES.L 1            ;パラメタブロック領域
FILENO:    .RES.B 1            ;ファイル番号領域
DATA:      .RES.B 1            ;データ代入領域
.END
```

付録 F. ASCII コード表

表 F.1 ASCII コード一覧表

パリティビット					b8								
					b7								
					b6								
					b5								
b4	b3	b2	b1	MSB LSB	0	1	2	3	4	5	6	7	
				0	NUL	DC ₀	SP	0	@	P	TM	p	
				1	SOM	X-ON	!	1	A	Q	a	q	
				2	EOA	TAPE	®	2	B	R	b	r	
				3	EOM	X-OFF	#	3	C	S	c	s	
				4	EOT	TAPE	\$	4	D	T	d	t	
				5	WRU	ERROR	%	5	E	U	e	u	
				6	RU	SYNC	&	6	F	V	e	v	
				7	BELL	LEM	TM	7	G	W	g	w	
				8	FE ₀	CAN	(8	H	X	h	x	
				9	TAB	S ₁)	9	I	Y	i	y	
				A	LF	EOF	*	:	J	Z	j	z	
				B	VT	ESC	+	;	K	[k	{	
				C	FF	S ₄	'	<	L	¥	l		
				D	CR	S ₅	-	=	M]	m	}	
				E	S ₀	S ₆	.	>	N	^	n	~	
				F	S ₁	S ₇	/	?	O	-	o	RUB OUT	

付録 G. C プログラムを C++ コンパイラでコンパイルするときの注意事項

(1) 関数のプロトタイプ宣言

関数を使用する前にプロトタイプ宣言が必要です。そのときには、仮引数の型も必ず宣言してください。

[例]

```
extern void func1();  
void g()  
{  
    func1(1); //エラー  
}
```



```
extern void func1(int);  
void g()  
{  
    func1(1); //OK  
}
```

(2) const オブジェクトのリンケージ

const オブジェクトのリンケージは、C では、外部的であるのに対し C++ では、内部的です。また、const オブジェクトは初期値を必要とします。

[例]

```
const cvalue1; //エラー  
  
const cvalue2 = 1; //内部的
```



```
const cvalue1 = 0;  
//初期値を与える  
  
extern const cvalue2 = 1;  
//Cの時と等価なふるまい
```

(3) void*からの代入

C++ では、void* は、明示的なキャストを用いないと他のオブジェクト型へのポインタ(関数へのポインタ、メンバへのポインタ除く)へ代入できません。

[例]

```
void func(void *ptrv,  
          int *ptri)  
{  
    ptri = ptrv; //エラー  
}
```



```
void func(void *ptrv,  
          int *ptri)  
{  
    ptri = (int*)ptrv; //OK  
}
```

付録 H. 索引

H.1 日本語索引

ア行

アセンブラ埋め込みインライン展開	60
アセンブリプログラムとの結合	39
後処理ルーチンの例	85
アンダフロー	165
インクルードファイル	7
インクルードファイルの読み込み方法	154
インターナルレベルメッセージ	105
ウォーニングレベルメッセージ	105
エラーメッセージ	105
エラーメッセージ一覧	105
エラーレベルメッセージ	105
オブジェクト情報	17,19
オブジェクトプログラムの構造	27
オプション	8
オプション一覧	8
オプション指定方法	5
オプションの組み合わせ	15
オーバフロー	65,165

カ行

外部名	39
外部名の相互参照方法	39
拡張機能	48
仮数部	160
仮想関数表	32
仮想基底クラスへのポインタ	33
環境の仕様	147
環境変数	21
関数のインライン展開	59

関数の呼び出し	40
起動方法	5
境界調整数	28
共用体型	31
組み込み型	30
組み込み型の内部表現	30
組み込み関数	51
組み込み関数の使用方法	51
グローバルオブジェクトの後処理	81,85
グローバルオブジェクトの初期処理	81,85
グローバルベースレジスタ	51,55
クラス	31
クラス、列挙型、ビットフィールドの仕様	152
限界値	25
言語仕様	147
コーディング上の注意事項	65
コマンド指定情報	21
コンパイルリストの見方	17
コンパイラのエラーメッセージ	105
コンパイラの環境変数	21
コンパイラの限界値	25
サ行	
サブコマンドファイル	9,13
識別子の仕様	147
指数部	160
システム組み込みの概要	71
実行環境の設定	80
実行時ルーチン	73
自動インライン展開	9,13
C プログラムとの結合	38
修飾子の仕様	152
初期化データ領域	28,74
初期処理データ領域	74,83
除算器	13
シンボルテーブルエントリ数	25,26
スタック切り替え指定	48,49

スタックフレーム.....	41
スタックポインタ.....	40
スタック領域.....	27,28,76,78
スタック領域の使用法.....	76
ステータスレジスタ.....	49,51
正規化数.....	161
整数の仕様.....	148
整数型とその値の範囲.....	149
静的領域の割り付け.....	72
積和演算.....	54,56
セクション.....	27,28,56,72
セクション切り替え機能.....	56
セクション初期化ルーチンの例.....	85
セクションの初期化.....	83,88
セクション名.....	11
ゼロ拡張.....	34
宣言の仕様.....	153
ソースリスト情報.....	17
タ行	
単精度浮動小数点ライブラリ.....	57
低水準インタフェースルーチン.....	87,88,92
低水準インタフェースルーチンの作成例.....	175
定数領域.....	28,74
データの内部表現.....	29
デバッグ情報.....	7,10,68
統計情報.....	8,17,20
動的領域.....	76
動的領域の割り付け.....	76
トラップ命令リターン指定.....	48,49
トラブル発生時の対処方法.....	67
ナ行	
内部表現.....	29,160
内部ラベル.....	25,26
2 バイトアドレス変数の指定.....	62

日本語	9,14,59
ノード	14
八行	
配列型	31
配列とポインタの仕様	151
引数	43
引数の型変換	44
引数の割り付け領域	45
引数割り付けの具体例	168
非正規化数	161
非数	161
ビットフィールド	34,38,152
ヒープ領域	28,76
評価順序	65
標準インクルードファイル	4
標準ライブラリとの対応	16
標準ライブラリのエラーメッセージ	141
ファイル拡張子	7
ファイル名の付け方	7
フェータルレベルメッセージ	105
符号拡張	34
符号部	160
浮動小数点数の限界値	150
浮動小数点数の仕様	150,160
ブラウザ情報	8,10,68
プリプロセッサの仕様	154
フレームサイズ	77
プログラム開発上のトラブル対処方法	67
プログラム作成上の注意事項	65
プログラムの構成例	80,86
プログラムの実行方式	27
プログラム領域	28
文の仕様	153
ベクタテーブルの設定	80,81,87
ベクタベースレジスタ	51
ポジションインディペンデントコード	9,12

翻訳の仕様.....	147
マ行	
マクロ名の定義.....	8,11
丸め.....	150,165
未初期化データ領域	28
無限大	161
無効演算	165
メッセージレベル.....	105
メモリ領域の割り付け	72
文字の仕様.....	148
文字列内の日本語記述	9,13,59
文字列の共有	9,12
ヤ行	
予約語	92
ラ行	
ライブラリ.....	4,16
ライブラリ関数仕様	155
リスト	8,11,17
リターンアドレス格納レジスタ.....	43
リターン値.....	41
リターン値の設定場所	46
リトルエンディアン	9,14,36
利用者定義型	31
レジスタ	41
レジスタとスタック領域の使用法	170
レジスタ退避・回復の制御	63
レジスタの仕様.....	151
レジスタ保証規則.....	41
列挙型	152

ワ行

割り込み関数	48
割り込み関数の使用方法.....	48

H.2 英語索引

A

abort ルーチン（異常終了関数）	174
abs16（pragma 指定）	62
ASCII コード	182
asmcode（サブオプション）	8,11
assert.h（標準ヘッダファイル）	155

B

browser（オプション）	8,10
bss（サブオプション）	9,11
big（サブオプション）	9,14
big endian	36

C

char 型	30,36
close ルーチン（低水準インタフェースルーチン）	97,177
code（オプション）	8,11
comment（オプション）	9,13
const（サブオプション）	9,11,12
const 型	66
cpu（オプション）	8,10
cpu（サブオプション）	9,14
ctype.h（標準ヘッダファイル）	155
C++ コンパイラの起動方法	5
C++ コンパイラの限界値	25
C++ コンパイラの実行	5
C++ プログラムの実行方式	27
C ライブラリ関数仕様	155

C ライブラリ関数のエラーメッセージ	141
C ライブラリ関数の実行環境の設定	86
C ライブラリの初期設定	89

D

data (サブオプション)	9,11,12
debug (オプション)	8,10
define (オプション)	8,11
division (オプション)	9,14
double 型	30

E

endian (オプション)	9,14
enum 型	30
errno	89,159
errno.h (標準ヘッダファイル)	159
euc (オプション)	9,13,59
exit ルーチン (プログラム終了関数)	172
expansion (サブオプション)	8,10

F

FILE 型	91
float 型	30,37

G

GBR (グローバルベースレジスタ)	52,55
gbr_base (pragma 指定)	62
gbr_base1 (pragma 指定)	62
GBR ベース変数の指定	62

H

help (オプション)	9,12
--------------------	------

I

IEEE	160
include (オプション)	8,11
include (サブオプション)	8,10
inline (オプション)	9,14
inline (pragma 指定)	59
inline_asm (pragma 指定)	60
int 型	30,37
interrupt (pragma 指定)	48

J

K

L

length (サブオプション)	8,10
listfile (オプション)	8,11
little (サブオプション)	9,14
little endian	14,36
long 型	30,37
long double 型	30
lseek ルーチン (低水準インタフェースルーチン)	100,178

M

machinecode (サブオプション)	8,11
machine.h (標準ヘッダファイル)	51,56
macsave (オプション)	9,15,41
math.h (標準ヘッダファイル)	57,156
mathf.h (標準ヘッダファイル)	58

N

O

object (サブオプション)	8,10
objectfile (オプション)	8,11
onexit ルーチン (終了処理関数)	171
open ルーチン (低水準インタフェースルーチン)	95,176
optimize (オプション)	8,10

P

peripheral (サブオプション)	9,14
pic (オプション)	9,12
PR レジスタ	43
pragma	48,154
preinclude (オプション)	9,14
preprocessor (オプション)	9,12
program (サブオプション)	9,11
ptrdiff_t 型	151,155

Q

R

RAM.....	74,75
read ルーチン (低水準インタフェースルーチン)	98,177
regsave (pragma 指定)	63
ROM.....	74,75
ROM (リンケージエディタのサブコマンド)	75

S

sbrk ルーチン (低水準インタフェースルーチン)	101,179
section (オプション)	9,11
section (pragma 指定)	56
setjmp.h (標準ヘッダファイル)	156
SH1 (サブオプション)	8,10
SH2 (サブオプション)	8,10
SH3 (サブオプション)	8,10

SHC_INC	21
SHC_LIB.....	21
SHC_TMP	21
short 型.....	30,36
show (オプション)	8,10
size (オプション)	8,10
sjis (オプション)	9,13,59
smachine.h (標準ヘッダファイル)	51,56
source (サブオプション)	8,10
SP (スタックポインタ)	41,45,47,49,79,81
sp (スタック切り替え指定)	48
speed (オプション)	8,10
SR (ステータスレジスタ)	49,51
start (リンケージエディタのサブコマンド)	75
statistics (サブオプション)	8,10
stddef.h (標準ヘッダファイル)	155
stdio.h (標準ヘッダファイル)	157
string (オプション)	9,12
string.h (標準ヘッダファイル)	158
subcommand (オプション)	9,13
T	
tn (トラップ命令リターン指定)	48
TRAPA 命令	48,49,53
U	
umachine.h (標準ヘッダファイル)	51,56
unsigned	30,37
V	
VBR (ベクタベースレジスタ)	51
VEC_TBL (ベクタテーブル)	81,87
volatile 型	152

W

width (サブオプション)	8,10
write ルーチン (低水準インタフェースルーチン)	99,178

X

Y

Z

—

__call_end.....	80,85,88
__call_init.....	81,85,88
__CLOSEALL.....	91,92
__DATE__.....	154
__INIT	81,82,88
__INITLIB.....	89
__INITSCT	83,88
__INIT_IOLIB.....	90
__INIT_LOWLEVEL.....	89
__INIT_OTHERLIB	91
__TIME__.....	154

記号

.LINE.....	15
\$G0.....	62
\$G1.....	62