H8S,H8/300 シリーズマイコン
評価ボード用 C コンパイラ
ユーザーズマニュアル

目 次

1　C コンパイラの実行

本章では 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。

1.1　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。

ch38［△〈オプション〉．．．］［△〈ファイル名〉［△〈オプション〉．．．］．．］

また、オプションの形式は次のとおりです。

〈オプション〉［=〈サブオプション〉］［，〈サブオプション〉］．．．］

以下、C コンパイラの基本的な操作方法を説明します。

(1) 起動コマンドの入力

```
ch38 (RET)
```

コンパイルは行なわず、コマンドライン形式、オプション一覧を出力します。

(2) プログラムのコンパイル

```
ch38△-cpu=300ha△test.c (RET)
```

C ソースプログラム「test.c」をコンパイルします。

本 C コンパイラは CPU／動作モードを必ず指定する必要があります。

(3) オプション指定方法

```
ch38△-cpu=300ha△-debug△test.c (RET)
ch38△-cpu=300ha△-debug△-speed=register,switch△test.c (RET)
```

オプション cpu、debug、speed の前に、-を付加します。

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。

また、複数のサブオプションを指定するときはカンマ（，）で区切ります。

(4) 複数のプログラムのコンパイル

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

```
ch38△-cpu=300ha△test1.c△test2.c (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）

```
ch38△-cpu=300ha△-indirect△test1.c △test2.c (RET)
```

test1.c、test2.c ともに indirect オプションが有効になります。

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

```
ch38△-cpu=300ha△test1.c△test2.c△-indirect (RET)
```

indirect オプションは test2.c のみ有効になります。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します。

1．2　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 1.1 に示します。下線部（__）は短縮形指定時の文字を示します。

オプション、サブオプションの指定方法は「1.1　C コンパイラの起動方法」を参照してください。

表 1.1　コンパイラオプション一覧

No.	項　目	オプション形式	省略時解釈	関連拡張言語仕様
1	短絶対アドレスの指定	abs8 abs16	なし	#pragma abs8 #pragma abs16 #pragma abs8 section #pragma abs16 section
2	列挙型のサイズ	byteenum	なし	なし
3	switch 文展開方法	case=ifthen｜ table	なし	なし
4	CPU／動作モード(アドレス空間のビット幅)	cpu=2600n｜ 2600a［:<アドレス空間のビット幅>］｜ 2000n｜ 2000a［:<アドレス空間のビット幅>］｜ 300hn｜ 300ha［:<アドレス空間のビット幅>］｜ 300｜ 300l｜ 300reg	なし	なし
5	デバッグ情報	debug nodebug	nodebug	なし

No.	項　目	オプション形式	省略時解釈	関連拡張言語仕様
6	ブロック転送命令	eepmov	なし	なし
7	インクルードファイル	include=<パス名>	なし	なし
8	メモリ間接形式	indirect	なし	#pragma indirect #pragma indirect section
9	リストファイル出力	list nolist	nolist	なし
10	最適化レベル	optimize=0｜1	optimize=1	なし
11	レジスタ変数拡張	regexpansion noregexpansion	regexpansion	なし
12	最適化内容の指定	speed [= register｜ shift｜ loop｜ switch｜ inline｜ struct]	なし	#pragma inline

次に各オプションの意味を示します。

(1) 短絶対アドレスの指定

■形式

abs8

abs16

■説明

静的領域に割り付けるデータを、短絶対アドレッシングモードでアクセスします。

abs8 オプションは、char 型、unsigned char 型および char 型、unsigned char 型の要素、メンバを含む複合型データを 8 ビット絶対アドレス(@aa:8)でアクセスするコードを生成します。

abs16 オプションは、CPU／動作モードが 2600a、2000a、300ha のとき、データを 16 ビット絶対アドレス（@aa:16）でアクセスするコードを生成します。CPU／動作モードが 2600n、2000n、300hn、300 のとき、本オプションの指定は無効です。

abs8 オプションにより、8 ビット絶対アドレスでアクセスされるデータは、セクション名“$ABS8+C セクション名”、“$ABS8+D セクション名”または“$ABS8+B セクション名”に出力されます。また、abs16 オプションにより、16 ビット絶対アドレスでアクセスされるデータは、セクション名“$ABS16+C セクション名”、“$ABS16+D セクション名”または“$ABS16+B セクション名”に出力されます。リンク時には、本オプションにより出力されたセクションを短絶対アドレス領域に割り付ける必要があります。

(2) 列挙型のサイズ

■形式

byteenum

■説明

enum 宣言した列挙型のデータを char 型として扱います。

本オプションが指定された場合で、かつ、enum 宣言した列挙型のメンバの値が全て-128～127 の範囲のとき、列挙型データを char 型として扱います。

本オプションを省略した場合、および、本オプションが指定されても列挙型のメンバの値が 1 つでも-128～127 の範囲外の場合は、列挙型データを int 型として扱います。

(3) switch 文展開方式

■形式

case=ifthen｜table

■説明

switch 文のコード展開方式を指定します。

case=ifthen オプションは、switch 文を if_then 方式で展開します。if_then 方式は、switch 文の評価式の値と case ラベルの値を比較し、一致すれば case ラベルの文へ飛ぶ処理を case ラベルの回数繰り返す展開方式です。この展開方式は、switch 文に含まれる case ラベルの数に比例してオブジェクトコードのサイズが増大します。

case=table オプションは、switch 文をテーブル方式で展開します。テーブル方式は、case ラベルの飛び先をジャンプテーブルに確保し、1 回のジャンプテーブルの参照で switch 文の評価式と一致する case ラベルの文へ飛び越す展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例して定数領域に確保されるジャンプテーブルのサイズが増えますが、実行速度は常に一定です。

本オプションを省略した場合は、オブジェクトサイズの縮小を優先したいずれかの展開方式を C コンパイラが自動的に選択します。

また、本オプション省略時に speed オプションあるいは speed=switch オプションを指定した場合は、実行速度を優先した展開方式を C コンパイラが自動的に選択します。

■例

```
int a, b;
   :
switch(a){
          case 1:       b=7; break;
          case 2:       b=6; break;
          case 3:       b=5; break;
          case 4:       b=4; break;
          case 5:       b=3; break;
          case 6:       b=2; break;
          default:      b=1; break;
}
```

上記の C ソースプログラムのコード展開例を次に示します。（cpu=300ha の場合）

```
MOV.W     @_a:24,R0          MOV.W     @_a:24,R0
MOV.B     R0H,R0H            SUB.W     #1:16,R0
BNE       Ld:8               CMP.W     #5:16,R0
CMP.B     #1:8,R0L           BHI       Ld:8
BEQ       L1:8               EXTU.L    ER0
CMP.B     #2:8,R0L           ADD.L     ER0,ER0
BEQ       L2:8               ADD.L     ER0,ER0
CMP.B     #3:8,R0L           MOV.L     @(L:24,ER0),ER0
BEQ       L3:8               JMP       @ER0
CMP.B     #4:8,R0L           ：
BEQ       L4:8               L:（ジャンプテーブル）
CMP.B     #5:8,R0L
BEQ       L5:8,R0L
CMP.B     #6:8,R0L
BEQ       L6:8
BRA       Ld:8
```

case=ifthen 時　　**case=table 時**

case 値	if_then 方式		table 方式	
	オブジェクトサイズ	実行サイクル	オブジェクトサイズ	実行サイクル
1	36 バイト	52	40（34+6）バイト	76
6		82		

(4) CPU／動作モード(アドレス空間のビット幅)

■形式

cpu=2600n｜
　2600a［:<アドレス空間のビット幅>］｜
　2000n｜
　2000a［:<アドレス空間のビット幅>］｜
　300hn｜
　300ha［:<アドレス空間のビット幅>］｜
　300｜300l｜300reg

■説明

作成するオブジェクトプログラムの CPU 種別と動作モードを指定します。サブオプションの一覧を表 1.2 に示します。

表 1.2　cpu オプションのサブオプション一覧

No.	サブオプション名	意　　味
1	2600n	H8S/2600 用ノーマルモードのオブジェクトを作成します。
2	2600a［:<アドレス空間のビット幅>］	H8S/2600 用アドバンストモードのオブジェクトを作成します。 <アドレス空間のビット幅>は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。<アドレス空間のビット幅>の省略時解釈は 32 です。
3	2000n	H8S/2000 用ノーマルモードのオブジェクトを作成します。
4	2000a［:<アドレス空間のビット幅>］	H8S/2000 用アドバンストモードのオブジェクトを作成します。 <アドレス空間のビット幅>は、20、24、28、32 のいずれかの数値で、それぞれ 1M バイト、16M バイト、256M バイト、4G バイトのアドレス空間を示します。<アドレス空間のビット幅>の省略時解釈は 32 です。
5	300hn	H8/300H 用ノーマルモードのオブジェクトを作成します。
6	300ha［:<アドレス空間のビット幅>］	H8/300H 用アドバンストモードのオブジェクトを作成します。 <アドレス空間のビット幅>は、20 または 24 の数値で、それぞれ 1M バイト、16M バイトのアドレス空間を示します。<アドレス空間のビット幅>の省略時解釈は 24 です。
7	300	H8/300 のオブジェクトを作成します。
8	300l	H8/300 のオブジェクトを作成します。クロスアセンブラとの互換のために用意しています。
9	300reg	H8/300 のオブジェクトを作成します。旧バージョンとの互換のために用意しています。

(5) デバッグ情報

■形式

debug
nodebug

■説明

debug オプションは、デバッグ時に C ソースプログラムレベルでのデバッグができるようにオブジェクトファイル中にデバッグ情報を出力することを指定します。

オブジェクトファイルがリロケータブルオブジェクトプログラムの時は、直接デバッグ情報が出力されます。

本オプションは、最適化オプションを指定した場合でも有効です。

nodebug オプションは、デバッグ情報をオブジェクトファイル中に出力しないことを指定します。

本オプションの省略時解釈は、nodebug です。

■注意

C ソースレベルのデバッグを行なうためには、アセンブルおよびリンク時にも debug オプションを指定する必要があります。

(6) ブロック転送命令

■形式

eepmov

■説明

構造体の代入文や局所変数で宣言された配列の初期値代入式を、ブロック転送命令 EEPMOV にコード展開します。

本オプションを省略した場合は、構造体の代入文などを MOV 命令または、実行時ルーチンに展開します。

■注意

EEPMOV 命令実行中に NMI 割り込みが発生すると、割り込み処理終了後、次の命令に制御が移るため動作結果が保証されません。本オプション使用時には NMI 割り込みに注意してください。

(7) インクルードファイル

■形式

include=<パス名>

■説明

include オプションは、コンパイルする C ソースプログラムが参照するインクルードファイルの存在するパス名を指定します。

パス名が複数ある場合にはカンマ（，）で区切って指定することができます。

(8) メモリ間接形式

■形式

indirect

■説明

C ソースプログラム内で呼び出す関数を全てメモリ間接（@@aa:8）で呼び出します。

本オプションの指定により、C ソースプログラム中に定義されている関数は、関数本体以外にセクション名“ $INDIRECT+セクション名 ”にメモリ間接呼び出しのためのアドレステーブルが出力されます。

■注意

indirect オプションを指定すると、ソースプログラム内で呼び出している標準ライブラリ関数も全てメモリ間接で呼び出すようにコード生成されます。この場合、当該標準ライブラリ関数のアドレスをアドレステーブルに設定する必要があります。

例：アドレステーブルのアセンブリプログラム

```
                .IMPORT   _printf       ;     標準関数の外部参照宣言
                                        ;（__＋標準ライブラリ関数名）
                .EXPORT   $printf       ;     アドレステーブルの外部定義宣言
                                        ;（$＋標準ライブラリ関数名）
                .SECTION  $INDIRECT,DATA,ALIGN=2
  $printf       .DATA.L   _printf       ;アドレステーブルの定義
                .END
```

アドレステーブルを格納できるエリアは、0x0000～0x00FF 番地に制限されています。リンク時には、リンケージマップによりアドレステーブルのセクションが 0x0000～0x00FF 番地の範囲内であることを確認してください。

(9) リストファイル出力

■形式

list
nolist

■説明

list オプションを指定するとリストファイルが作成されます。
nolist オプションを指定すると、リストファイルは作成されません。
本オプションの省略時解釈は nolist です。

(10) 最適化レベル

■形式

optimize=0｜1

■説明

オブジェクトプログラムの最適化レベルを指定します。
optimize=0 オプションは、C コンパイラがオブジェクトプログラムの最適化を行なわないことを示します。
optimize=1 オプションは、C コンパイラが最適化を行なうことを示します。
本オプションの省略時解釈は、optimize=1 とみなします。

■注意

optimize=0 オプションを指定したとき、speed=inline、loop オプションは無効となります。

（11）レジスタ変数拡張

■形式

regexpansion

noregexpansion

■説明

regexpansion オプションは、レジスタ変数を割り付けるレジスタの数を拡張することを指定します。

noregexpansion オプションは、レジスタ変数を割り付けるレジスタの数の拡張を行なわないことを指定します。

レジスタの数を拡張した場合、一般にレジスタに割り付く変数の数が多くなり、変数のアクセススピードが速くなります。

本オプションの省略時解釈は、regexpansion です。

（12）最適化内容の指定

■形式

```
speed [=register｜
        shift｜
        loop｜
        switch｜
        inline｜
        struct]
```

■説明

コンパイラが生成するオブジェクトに対し、実行速度の高速化を図る最適化を指定します。

speed=register オプションは、CPU／動作モードが 300ha、300hn、300 の時、関数の入口／出口でレジスタを退避／回復するコードとして実行時ルーチンを使用せずに PUSH、POP 命令で展開します。

speed=shift オプションは、シフト演算をより高速なオブジェクトコードで展開します。

speed=loop、speed=switch オプションは、for および switch 文のコード生成の高速化を指定します。

speed=inline オプションは、呼び出す関数をインライン展開することを示します。

speed=struct オプションは、構造体型や double 型の代入文を実行時ルーチンを用いず、直接インライン展開します。

speed のみを指定した場合は、これら全ての実行速度優先の最適化を行ないます。本オプションを省略した場合は、実行速度よりもオブジェクトコードのサイズ縮小を重視したオブジェクトを生成します。

■注意

最適化なし（optimize=0）を指定したとき、speed=loop、inline は無効となります。

2　C プログラムの実行方式

本章では、C プログラムとアセンブリプログラムを結合する場合について詳しく説明します。

なお、本章では、H8S/2600、H8S/2000、H8/300H または H8/300 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。

2.1　アセンブリプログラムとの結合

C 言語はシステムプログラムの記述に適しており、マイコン組み込み用の応用システムのほとんどの処理を C 言語で記述することができます。特に本 C コンパイラでは、アセンブラ埋め込み機能や組み込み関数などをサポートすることにより、全ての機能を C 言語で記述できるようになっています。

しかしながら、ハードウェアのタイミング要求やメモリサイズの制限などのように性能要求が厳しい場合、アセンブリ言語で記述し、C プログラムと結合する必要があります。

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

(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いない関数名

外部名となる変数名、関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（31 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　ア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=2
_a: .DATA.W   1
_b: .DATA.W   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）

```
char  a,b;
```

アセンブリプログラム（参照する側）

```
 .IMPORT   _a,_b
 .SECTION  P,CODE,ALIGN=2
 MOV.B     @_a,R5L
 MOV.B     R5L,@_b
 RTS
 .END
```

(2) 関数呼び出し

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なうときに、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。

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

(a) 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。

(b) スタックフレームの割り付け、解放に関する規則

関数呼び出しが行なわれた時点（JSR または BSR 命令の実行直後）では、スタックポインタはリターンアドレスの領域を指しています。この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、リターンアドレスの領域の解放を呼び出される側の関数で行ないます。これは、通常 RTS 命令を用いて行ないます。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。

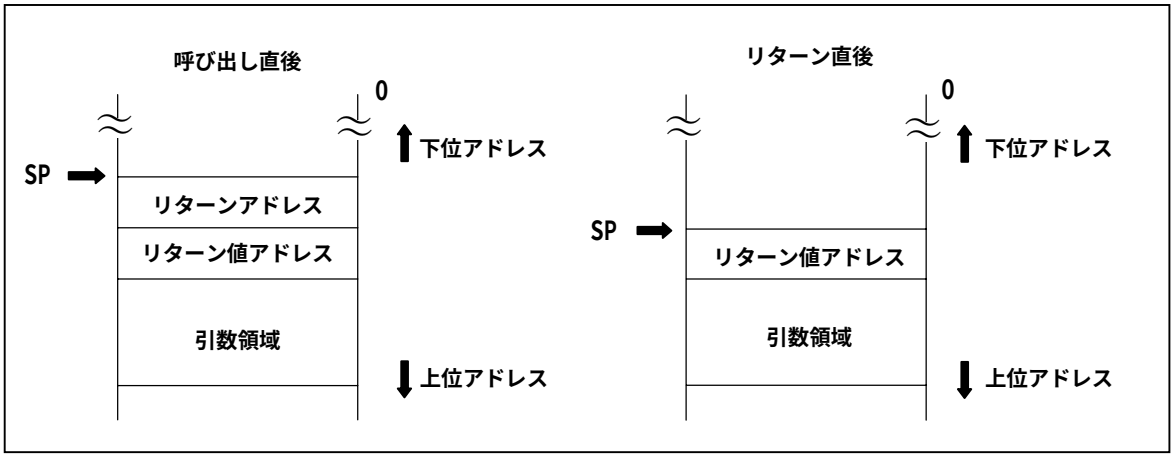

図 2.1　スタックフレームの割り付け、解放に関する規則

(c) レジスタに関する規則

関数呼び出し前後において、値を保証するレジスタと保証しないレジスタがあります。
各 CPU 種類におけるレジスタの保証規則を表 2.1 に示します。

表 2.1　関数呼び出し前後のレジスタ保証規則

No.	項　目	CPU 種類と対象レジスタ		プログラミングにおける留意点
		H8S/2600 用、H8S/2000 用、H8/300H 用	H8/300 用	
1	保証しないレジスタ	ER0、ER1	R0、R1	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、退避せずに使用可能。
2	保証するレジスタ	ER2～ER6	R2～R6	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。

以下、レジスタ保証規則について H8/300H 用アドバンストモードの場合の具体例を示します。

■例 1　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=2
_sub:   JSR        @$sp_regsv$3:24
        SUBS.L     #4,SP
                :
        ADDS.L     #4,SP
        JSR        @$sp_regld$3:24
        RTS
        .END
```

} 関数内で使用するレジスタの退避
} 関数本体の処理
（ER0、ER1は保証しないレジスタのため、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  void  sub( );
f( )
{
    sub( );
}
```

■例２　Ｃプログラムのサブルーチン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
 .SECTION   P,CODE,ALIGN=2
          :
 MOV.L      ER1,@(4,SP)
 MOV.L      ER0,ER6
 JSR        @_sub
          :
 .END
```

レジスタER0、ER1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

(d) 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。引数とリターン値の規則は、関数の宣言において、個々の引数とリターン値の型が明示的に宣言されているかどうかによって異なります。引数とリターン値の型を明示的に宣言するには、関数の原型宣言を用います。

以下の解説では、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。

(i) 引数とリターン値に対する一般的な規則

引数の渡し方

引数の値を、必ず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。

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なう場合があります。以下、この型変換の規則について説明します。

●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。

●型の宣言された引数の型変換

原型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。

●型の宣言されていない引数の型変換

原型宣言によって型が宣言されていない引数の型変換は、以下の規則に従って変換します。

・char 型、unsigned char 型の引数は、int 型に変換します。

・float 型の引数は、double 型に変換します。

・上記以外の型は、変換しません。

■例

(1)

```
long  f( );
long  f( )
{    float  x;
     return  x;
}
```

→ リターン値はlong型に型変換します

(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型に変換します

→ 1.0は、対応する引数の型が int型なので、int型に変換します

■注意

C コンパイラでは、原型宣言によって引数の型を宣言していない場合、正しく引数が渡されるように呼び出される側と呼び出す側で同じ型を指定しないと、動作を保証しません。

```
f(x)
float  x;
{
     :
}
main( )
{
     float  x;
     f(x);
}
```

動作を保証しない指定例

```
f(float  x)
{
     :
}
main( )
{
     float  x;
     f(x);
}
```

正しい指定例

動作を保証しない指定例では、関数「f」の引数の原型宣言がないため、関数「main」の側で呼び出すときに引数 x を double 型に変換します。

一方、関数「f」の側では引数を float 型として宣言していますので正しく引数を受け渡しすることはできません。

原型宣言によって引数の型を宣言するか、関数「f」の側の引数宣言を double 型にする必要があります。

正しい指定例は、原型宣言によって引数の型を宣言した例です。

（ii）引数の割り付け領域

引数は、スタック上の引数領域に割り付ける場合と、レジスタに割り付ける場合があります。

オブジェクト種類ごとの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 2.2 にそれぞれ示します。

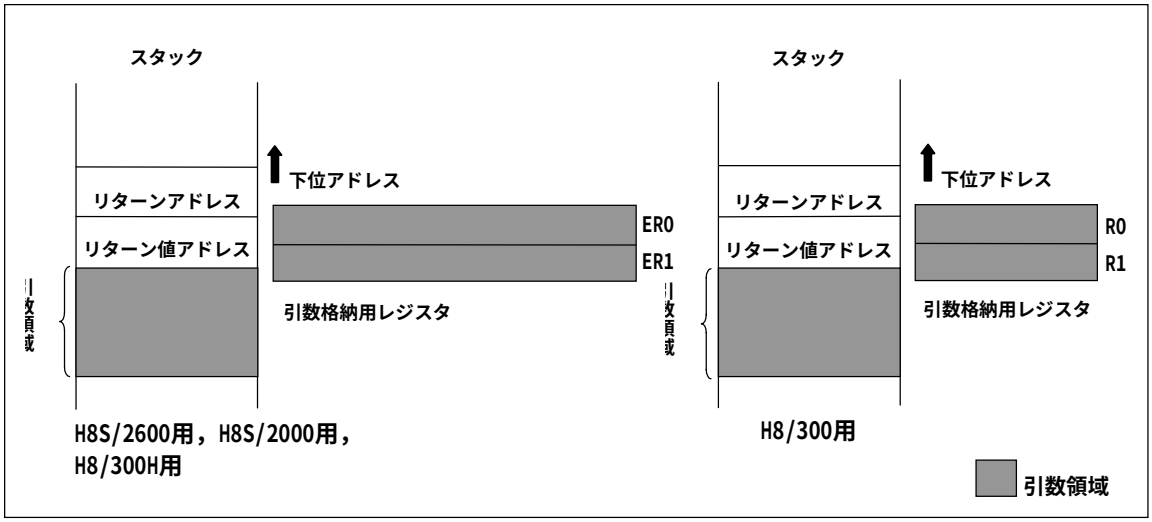

図 2.2　引数の割り付け領域

表 2.2　引数割り付け領域の一般規則

No	CPU 種類	割り付け規則		
		レジスタに割り付ける引数		スタックに割り付ける引数
		引数格納用レジスタ	対象の型	
1	H8S/2600 用、H8S/2000 用、H8/300H 用	ER0、ER1	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、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] 原型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もの　*1 [3] 引数の数が多いため、レジスタに割り付かなかったもの
2	H8/300 用	R0、R1	char、unsigned char、short、unsigned short、int、unsigned int、ポインタ	

【注】　*1　原型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,...);
       :
       f2(x,y,z);    →  y,z はスタックに割り付けます。
       :
```

（iii）引数の割り付け

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、LSB 側の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。

```
f(char  a, int  b)
{
     :
}
```

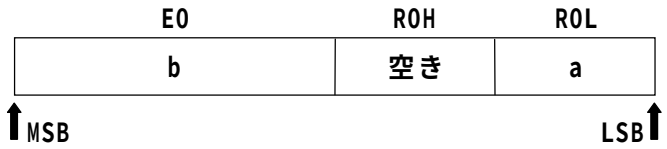

図 2.3　引数格納用レジスタの割り付け例（H8/300H 用）

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指定した順に下位アドレスから割り付けます。

■注意

構造体型、共用体型引数に関する注意

構造体型、共用体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 2 バイト境界に割り付け、しかもその領域として偶数バイトの領域を使用します。これは、H8S,H8/300 シリーズのスタックポインタが 2 バイト単位で変化するためです。

（iV）リターン値の設定場所

関数のリターン値の型により、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は表 2.3 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の領域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 2.3　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	
		H8S/2600 用、H8S/2000 用、H8/3用	H8/300 用
1	char、 unsigned char	レジスタ（R0L）	レジスタ（R0L）
2	short、 unsigned short、 int、 unsigned int	レジスタ（R0）	レジスタ（R0）
3	ポインタ	レジスタ ノーマルモード：（R0） アドバンストモード：（ER0）	レジスタ（R0）
4	long、 unsigned long、 float	レジスタ（ER0）	リターン値設定領域（メモリ）
5	double、 long double、 構造体、共用体	リターン値設定領域（メモリ）	リターン値設定領域（メモリ）

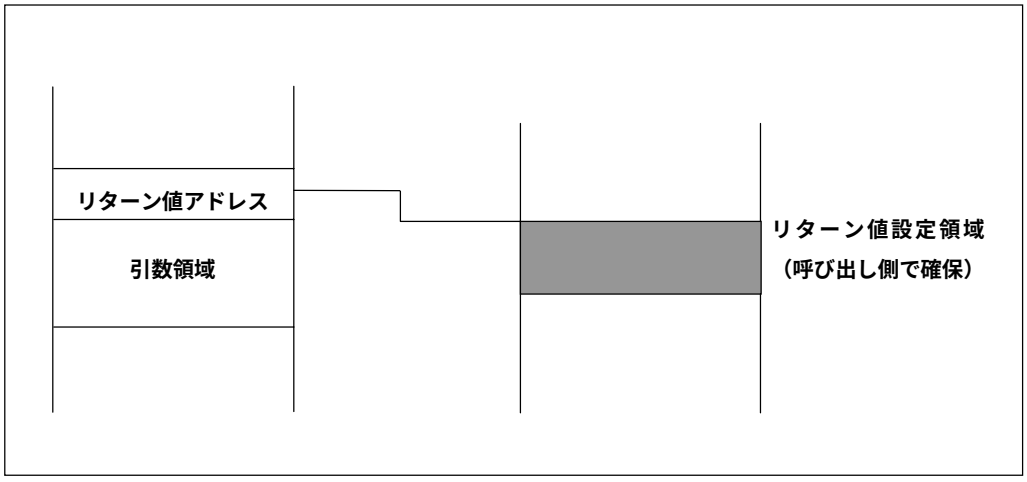

図 2.4　リターン値をメモリに設定する場合の設定領域

付録

Cライブラリ関数のスタック使用量一覧

No.	ヘッダ名	関数名	スタックサイズ(バイト)					備 考
			H8/300	H8/300H ﾉｰﾏﾙﾓｰﾄﾞ	H8/300H ｱﾄﾞﾊﾞﾝｽﾄﾓｰﾄﾞ	H8S ﾉｰﾏﾙﾓｰﾄﾞ	H8S ｱﾄﾞﾊﾞﾝｽﾄﾓｰﾄﾞ	
1	stddef.h	offsetof	—	—	—	—	—	マクロ
2	assert.h	assert	—	—	—	—	—	マクロ
3	ctype.h	isalnum	4	4	8	4	8	
		isalpha	4	4	8	4	8	
		iscntrl	4	4	8	4	8	
		isdigit	4	4	8	4	8	
		isgraph	4	4	8	4	8	
		islower	4	4	8	4	8	
		isprint	4	4	8	4	8	
		ispunct	4	4	8	4	8	
		isspace	4	4	8	4	8	
		isupper	4	4	8	4	8	
		isxdigit	4	4	8	4	8	
		tolower	2	2	4	2	4	
		toupper	2	2	4	2	4	
4	math.h	frexp	54	54	60	44	56	
		ldexp	32	30	52	30	40	
		modf	38	46	52	36	48	
		ceil	102	110	156	100	128	
		fabs	30	28	36	28	36	
		floor	70	78	104	68	88	
		fmod	92	110	120	88	104	
5	setjmp.h	setjmp	2	2	4	2	4	
		longjmp	2	2	4	2	4	
6	stdarg.h	va_start	—	—	—	—	—	マクロ
		va_arg	—	—	—	—	—	マクロ
		va_end	2	2	4	2	4	
7	stdio.h	fclose	28	48	56	28	44	

		fflush	12	22	24	10	12	
		fopen	42	54	88	40	72	
		freopen	44	66	92	40	72	
		setbuf	4	4	8	4	8	
		setvbuf	16	6	30	6	18	
		fprintf	660	626	734	594	698	
		fscanf	464	426	518	410	496	
		printf	658	624	718	592	694	
		scanf	462	424	502	408	492	
		sprintf	660	626	734	594	698	
		sscanf	464	426	518	410	496	
		vfprintf	662	642	738	602	702	
		vprintf	660	626	734	594	698	
		vsprintf	662	642	738	602	702	
		fgetc	52	66	140	52	100	
		fgets	68	88	160	70	132	
		fputc	64	84	156	56	104	
		fputs	76	90	180	62	120	
		getc	50	64	124	50	96	
		getchar	48	62	124	48	96	
		gets	56	86	148	64	114	
		putc	64	84	156	56	104	
7	stdio.h	putchar	70	100	156	60	110	
		puts	76	106	180	66	124	
		ungetc	54	78	132	50	96	
		fread	78	86	152	70	124	
		fwrite	76	78	148	62	120	
		fseek	36	22	32	22	32	
		ftell	16	22	24	10	12	
		rewind	10	6	26	6	14	
		clearerr	2	2	4	2	4	
		feof	4	4	8	4	8	
		ferror	4	4	8	4	8	
		perror	672	648	758	604	718	
8	stdlib.h	atof	426	392	466	376	444	
		atoi	70	38	82	38	66	

		atol	68	38	82	38	66	
		strtod	418	384	438	368	428	
		strtol	60	34	74	34	58	
		rand	18	4	8	4	8	
		srand	2	6	8	6	8	
		calloc	42	72	108	44	80	
		free	12	22	24	14	20	
		malloc	36	50	84	34	68	
		realloc	52	72	116	52	100	
		bsearch	22	24	48	20	48	
		qsort	32	28	64	28	64	再帰的関数
		abs	4	2	4	2	4	
		div	130	134	168	134	168	
		labs	12	6	8	6	8	
		ldiv	146	144	172	144	172	
9	string.h	memcpy	14	22	28	14	28	
		strcpy	12	6	24	6	16	
		strncpy	14	22	28	14	28	
		strcat	12	6	24	6	16	
		strncat	14	22	28	14	28	
		memcmp	14	6	28	6	24	
		strcmp	6	6	24	6	12	
		strncmp	14	6	28	6	24	
		memchr	14	6	28	6	20	
		strchr	4	4	8	4	8	
		strcspn	12	22	24	10	24	
		strpbrk	12	22	24	10	24	
		strrchr	18	24	48	12	28	
		strspn	12	22	24	10	24	
		strstr	26	28	52	20	48	
		strtok	24	44	48	24	16	
		memset	16	22	30	14	26	
		strerror	6	2	24	2	16	
		strlen	6	2	24	2	12	
		memmove	14	22	28	14	28	

実行時ルーチンスタック使用量一覧

No.	ルーチン名	スタックサイズ(バイト)				
		H8/300	H8/300H ﾉｰﾏﾙﾓｰﾄﾞ	H8/300H ｱﾄﾞﾊﾞﾝｽﾄﾓｰﾄﾞ	H8S/2600 ﾉｰﾏﾙﾓｰﾄﾞ	H8S/2600 ｱﾄﾞﾊﾞﾝｽﾄﾓｰﾄﾞ
1	$ADDD$3	40	28	30	30	32
2	$ADDF$3	20	18	20	18	20
3	$ADDL$3	6	—	—	—	—
4	$ANDL$3	6	—	—	—	—
5	$BFINC$3	6	6	8	6	8
6	$BFINI$3	8	8	10	10	12
7	$BFSC$3	4	4	6	4	6
8	$BFSI$3	4	4	6	4	6
9	$BFUC$3	4	4	6	4	6
10	$BFUI$3	4	4	6	4	6
11	$CDVC$3	12	—	—	—	—
12	$CDVI$3	14	—	—	—	—
13	$CDVUI$3	10	—	—	—	—
14	$CMDC$3	12	—	—	—	—
15	$CMDI$3	14	—	—	—	—
16	$CMDUI$3	10	—	—	—	—
17	$CMLI$3	12	—	—	—	—
18	$CMPD$3	10	24	24	24	24
19	$CMPF$3	10	6	8	6	8
20	$CMPL$3	10	—	—	—	—
21	$CTOL$3	6	—	—	—	—
22	$DIVC$3	6	—	—	—	—
23	$DIVD$3	48	30	32	30	32
24	$DIVF$3	32	22	24	22	24
25	$DIVI$3	8	—	—	—	—
26	$DIVL$3	18	10	14	10	14
27	$DIVUI$3	4	—	—	—	—
28	$DIVUL$3	16	6	8	6	8
29	$DIVUX$3	6	—	—	—	—
30	$DIVXSB$3	6	—	—	—	—

31	$DIVXSW$3	10	—	—	—	—
32	$DIVXUW$3	10	—	—	—	—
33	$DSLC$3	6	6	8	6	8
34	$DSLI$3	6	6	8	6	8
35	$DSLL$3	—	8	10	10	12
36	$DSRC$3	6	6	8	6	8
37	$DSRI$3	6	6	8	6	8
38	$DSRL$3	—	8	10	10	12
39	$DSRUC$3	6	6	8	6	8
40	$DSRUI$3	6	6	8	6	8
41	$DSRUL$3	—	8	10	10	12
42	$DTOF$3	16	14	16	14	16
43	$DTOI$3	18	—	—	—	—
44	$DTOL$3	14	10	12	10	12
45	EQD3	12	26	28	26	28
46	EQF3	12	8	12	8	12
47	fp_regld3	2	2	4	—	—
48	fp_regsv3	10	18	20	—	—
49	$FTOD$3	10	10	12	10	12
50	$FTOI$3	14	—	—	—	—
51	$FTOL$3	10	6	8	6	8
52	GED3	12	26	28	26	28
53	GEF3	12	8	12	8	12
54	GTD3	12	26	28	26	28
55	GTF3	12	8	12	8	12
56	$ITOD$3	6	6	8	6	8
57	$ITOF$3	8	4	6	4	6
58	$ITOL$3	4	—	—	—	—
59	LED3	12	26	28	26	28
60	LEF3	12	8	12	8	12
61	LTD3	12	26	28	26	28
62	LTF3	12	8	12	8	12
63	$LTOD$3	10	10	12	10	12
64	$LTOF$3	8	8	10	8	10
65	$MODL$3	18	—	—	—	—
66	$MODUL$3	16	—	—	—	—

67	$MULD$3	48	54	56	54	56
68	$MULF$3	32	18	20	18	20
69	$MULI$3	8	—	—	—	—
70	$MULL$3	14	10	12	10	12
71	$MULXSB$3	4	—	—	—	—
72	$MULXSW$3	10	—	—	—	—
73	$MULXUW$3	10	—	—	—	—
74	$MV8$3	4	6	8	6	8
75	MVN3	12	14	18	14	20
76	NED3	12	26	28	26	28
77	NEF3	12	8	12	8	12
78	$NEGD$3	4	6	8	6	8
79	$NEGF$3	4	—	—	—	—
80	$NEGL$3	6	—	—	—	—
81	ORL3	6	—	—	—	—
82	$PODD$3	46	36	42	40	44
83	$POID$3	46	36	42	40	44
84	$PODF$3	26	—	—	—	—
85	$POIF$3	26	—	—	—	—
86	$PODL$3	8	—	—	—	—
87	$POIL$3	8	—	—	—	—
88	$PRDD$3	46	34	42	36	44
89	$PRDF$3	26	24	28	24	28
90	$PRDL$3	4	—	—	—	—
91	$PRID$3	46	34	42	36	44
92	$PRIF$3	26	24	28	24	28
93	$PRIL$3	4	—	—	—	—
94	SLI3	6	—	—	—	—
95	SLL3	8	—	—	—	—
96	sp_regld3	2	2	4	—	—
97	sp_regsv3	12	22	24	—	—
98	SRI3	6	—	—	—	—
99	SRL3	8	—	—	—	—
100	$SRUI$3	6	—	—	—	—
101	$SRUL$3	8	—	—	—	—
102	$SUBD$3	40	46	50	48	52

103	$SUBF$3	20	18	20	18	20
104	$SUBL$3	6	—	—	—	—
105	SWI3	10	—	—	—	—
106	$ULTOD$3	10	10	12	10	12
107	$ULTOF$3	8	6	8	6	8
108	$UTOD$3	6	6	8	6	8
109	$UTOF$3	6	2	4	2	4
110	$XORL$3	6	—	—	—	—
111	_INITSCT	10	16	18	16	18