お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

HEW 生成ファイル補足説明（H8S、H8/300）

1　はじめに

1．1　本書の記載内容

本書では、アセンブリ言語及び、C/C++言語による組込みプログラムの記述方法を説明しながら、HEW が生成するファイルの補足説明をします。

1．2　対象コンパイラ

本書では、H8S,H8/300 Series C/C++コンパイラパッケージ Ver.5(C/C++コンパイラ Ver.4)、HEW3.0 以降を対象としています。それ以前のバージョンでは使用できない pragma などが登場します。ご了承頂きます様お願い致します。また、HEW、コンパイラ、アセンブラ、及び、最適化リンケージエディタの使用方法につきましては、コンパイラ製品に同梱されている電子マニュアル（PDF ファイル）をご参照頂きます様お願い致します。

1．3　注意事項

実際のメモリ配置やベクタテーブルの内容、マイコンの使用方法につきましては、お使いのマイコンのハードウェアマニュアルやプログラミングマニュアルをご参照下さい。また、本書では C/C++言語の言語仕様などの説明はしておりません。

2　アセンブラでの記述

2．1　リセット関数の作成

下記アセンブリプログラムは、H8S,H8/300 マイコンが動作する必要最低限のプログラムです。パワーオンリセット直後、リセット関数 _PowerON_Reset が配置されている 0x0400 番地にジャンプします（ハードウェアにより自動的に処理されます）。その後、SLEEP で処理を終了します。なお、本来、セクション名は任意に付けられますが、ここでは HEW 生成ファイルに合わせて PResetPRG、$VECT0 としています。関数名も HEW 生成ファイルに合わせて_PowerON_Reset としています。$VECT0 はリセットベクタテーブルで、0 番地に配置します。PResetPRG セクションの配置アドレスはベクタテーブル以外の ROM 領域であれば任意のアドレスに配置できますが、HEW が生成する設定に合わせて、0x0400 番地にしています。

```
    .SECTION PResetPRG,CODE,LOCATE=H'0400
_PowerON_Reset:
    SLEEP

    .SECTION $VECT0,DATA,LOCATE=H'0000
    .DATA.L    _PowerON_Reset

    .END
```

リスト 2-1

２．２　関数呼出し

下記アセンブリプログラムは、パワーオンリセット後、_PowerON_Reset 関数から _main 関数をコールし、戻った後に SLEEP で処理を終了します。また、エントリ関数の入口でスタックポインタの初期設定を行います。

```
    .SECTION PResetPRG,CODE,LOCATE=H'0400
_PowerON_Reset:
    MOV.L      #H'FFEDC0,SP
    JSR         @_main
    SLEEP

    .SECTION P,CODE,LOCATE=H'0800
_main:
    RTS
```

リスト 2-2

関数から戻る場合には、RTS 命令で復帰します。しかし、_PowerON_Reset には RTS 命令を記述しませんでした。リセット関数から”戻る”事はありませんので、RTS は無くて構いません。SLEEP するか、無限ループにしておく必要があります。下記は無限ループの例です。

```
_PowerON_Reset:
    MOV.L      #H'FF EDC0,SP
    JSR         @_main
_loop:
    BRA      _loop
```

リスト 2-3

２．３　割込み関数の作成(1)

割込み関数はアセンブラで下記の様に記述します。下記例では、IRQ0 割込みが発生すると **_INT_IRQ0** 関数がコールされ、SLEEP で処理を終了します。また、IRQ1 による割込みが発生すると、**_INT_IRQ1** 関数がコールされ、SLEEP で処理を終了します。

```
    .SECTION PIntPRG,CODE,LOCATE=H'0400
_INT_IRQ0
    SLEEP

_INT_IRQ1
    SLEEP

    .SECTION    $VECT16,DATA,LOCATE=H'40
    .DATA.L     _INT_IRQ0
    .SECTION    $VECT17,DATA,LOCATE=H'44
    .DATA.L     _INT_IRQ1
```

リスト 2-4

セクション名は任意に付けられますが、HEW 生成ファイルに合わせて PIntPRG としています。PIntPRG セクションはベクタテーブル以外の ROM 領域であれば任意のアドレスに配置できますが、HEW が生成する設定に合わせて、0x0400 番地に配置しています。

２．４　割込み関数の作成(2)

「2.4 割込み関数の作成(1)」は、割込みが発生した後、SLEEP で処理を終了する例でした。割込みによっては、割込みから復帰し、割込み発生前の処理を続ける場合があり、割込み関数から復帰する場合に RTE 命令で復帰する必要があります。

```
_INT_TRAPXX
    RTE
```

リスト 2-5

3　C/C++言語での記述

3．1　リセット関数の作成

「リスト 2-1」のようなプログラムは、C/C++言語では下記のように記述できます。

```
#include <machine.h>                          ...(a)
#pragma stacksize 0x200

#pragma section ResetPRG                      ...(b)
__entry(vect=0) void PowerON_Reset(void)      ...(c)
{
    sleep();                                  ...(d)
}
```

リスト 3-1

PowerON_Reset はリセット関数です。(b)の#pragma section ResetPRG 指定により、PResetPRG セクションに生成されます。HEW が生成する設定では、PResetPRG セクションは 0x0400 番地に配置しています。SLEEP を C/C++言語で記述するためには、組込み関数を使用します(d)。組込み関数は sleep 関数以外に、割込みマスクの設定・参照（set_imask、get_imask）、nop 関数などがあります。組込み関数を使用する場合には<machine.h>を使用するソースファイルの中でインクルードします(a)。PowerON_Reset 関数から”戻る”事はありませんので、関数入口でのレジスタ退避、関数出口でのレジスタ回復は必要ありません。(c)の**__entry** 指定により、レジスタの退避・回復コードを抑止することができます。

3．2　関数呼出し

「リスト 2-4」のようなプログラムは、C/C++言語で下記の様に記述できます。RTS 命令の生成（「２．２　関数呼出し」）はコンパイラが自動的に生成しますので、ユーザが意識する必要はありません。

```
#include <machine.h>

#pragma stacksize 0x200

void sub(void)
{
}

void main(void)
{
    sub();
}

#pragma section ResetPRG
__entry(vect=0) void PowerON_Reset(void)
{
    main();
    sleep();
}
```

リスト 3-2

foo 関数及び、main 関数は P セクションに生成されます。HEW が生成する設定では、P セクションを 0x0800 番地に配置しています。

３．３　割込み関数の作成

「リスト 2-5」のような割込み関数は、C/C++言語では下記の様に記述します。

```
#include <machine.h>

#pragma section IntPRG                          ...(a)

__interrupt(vect=8) void INT_TRAP0(void)   ...(b)
{
        sleep();
}

__interrupt(vect=9) void INT_TRAP1(void)   ...(c)
{
        sleep();
}

__interrupt(vect=XX) void INT_TRAPXX(void) ...(d)
{

}
```

リスト 3-3

INT_TRAP0 関数、**INT_TRAP1** 関数及び、**INT_TRAPXX** 関数は、(a)の#pragma section IntPRG 指定により、PIntPRG セクションに生成されます。HEW が生成する設定では、PIntPRG セクションを 0x00000400 番地に配置してます。INT_TPAPXX 関数は、(d)の__interrupt 指定により、RTE 命令により関数から復帰します（コードはコンパイラが自動的に生成します)。また、割込み関数の中で書き換えられる可能性のあるレジスタは、コンパイラにより退避・回復するようにコード生成されます。**INT_TRAP0** 関数と **INT_TRAP1** 関数は、(b)、(c)の__interrupt 指定により、RTE 命令で関数から復帰するコードが生成されますが、組込み関数 sleep があるため、この例では sleep で処理を終了します。

３．４　未初期化データ領域

下記(a)の変数 a の様に初期値のない外部変数定義は、未初期化データ領域として、デフォルトでは B セクションに生成されます。

```
int a;                              ...(a)
void main(void)
{
    int b = a;                      ...(b)
    a = 1;                          ...(c)
}
```

リスト 3-4

未初期化データ領域は(c)のようにプログラム実行時に書き換えられる可能性があるため、RAM 領域に配置します。また、ANSI 言語仕様により、初期値のない外部変数は 0 でなければなりません。例えば、(b)の参照したとき、変数 a の値は 0 でなければなりません。従って、アプリケーションを実行する際には、下記の様に B セクションを 0 クリアする必要があります。ただし、アプリケーション内で適切に初期化されている場合には、0 クリア処理は無くても構いません。

```
void PowerON_Reset
{
    char* p;

    p=(char*)__sectop("B");
    for(; p<(char*)__secend("B"); p++ )
        *p = 0;
    main();
    sleep();
}
```

リスト 3-5

３．５　定数領域

下記(a)の変数 b の様に、volatile 指定が無く const 指定された、初期値のある外部変数定義は、定数領域として、デフォルトでは C セクションに生成されます。また、(b)のような文字列”string”も定数データ領域としてデフォルトでは C セクションに生成されます。

```
int a;
const int b = 1;                ...(a)
void main(void)
{
    char* str = “string”;       ...(b)
    a = b;
}
```

リスト 3-6

定数領域はプログラム実行中に書き換わることはありませんので、ROM 領域に配置します。
なお、文字列はコンパイラオプションにより、「３．６　初期化データ領域」と同じ扱いにすることができます。

３．６　初期化データ領域

下記(a)の変数 a の様に初期値のある外部変数定義は、初期化データ領域として、デフォルトでは D セクションに生成されます。

```
int a = 1;                          ...(a)
void main(void)
{
    a = 2;                          ...(b)
}
```

リスト 3-7

初期化データ領域は初期値を持つため、初期値データを ROM 領域に持っていなければなりません。また、(b)のようにプログラム実行中に書き換えられる可能性があるため、実際にアクセス（値の格納や取り出し）するデータは RAM 領域になければなりません。このときに使用するのが、最適化リンケージエディタの持つ ROM 化オプションです(「４．４　ROM 化オプション」)。この機能を使用し、例えば D セクションのコピーとして R セクションを生成します。この時、実際のアクセスは R セクションに対して行われます。さらに、最適化リンケージエディタのメモリ配置設定により、D セクションを ROM 領域に、R セクションを RAM 領域に配置します。なお、R セクションの名称は任意に付けることができます。

また、プログラム実行時には、初期値付き変数は既に値を持っている状態でなければなりません。そこで、下記の様に D セクションから R セクションにデータをコピーする必要があります。

```
#include <machine.h>

void PowerON_Reset(void)
{
    char *p, *q;

    p=(char*)__sectop("D");
    q=(char*)__sectop("R");
    for( ; p<(char*)__secend("D"); p++, q++ )
        *q = *p;

    main();
    sleep();
}
```

リスト 3-8

なお、__sectop は__secend 同様セクションアドレス演算子であり、セクションの先頭アドレスを取得できます（やはり関数ではありません)。

３．７　_INITSCT 関数

「リスト 3-5」、「リスト 3-8」では B セクションの 0 クリア及び、D セクションから R セクションへのコピーをユーザプログラムにて記述しました。しかし、標準ライブラリに含まれる_INITSCT 関数を使用することで、初期化ルーチンをユーザが記述する必要がなくなります。_INITSCT 関数は下記の様に使用します。

```
#include <_h_c_lib.h>                                   ...(a)

void PowerON_Reset(void)
{
    _INITSCT();                                         ...(b)

    main();

    sleep();
}

#pragma section $DSEC                                   ...(c)
static const struct {                                   ...(d)
    char *rom_s;
    char *rom_e;
    char *ram_s;
}DTBL[]= {
    {__sectop("D"),__secend("D"), __sectop("R")},
};
#pragma section $BSEC                                   ...(e)
static const struct {                                   ...(f)
    char *b_s;
    char *b_e;
}BTBL[]= {
    {__sectop("B"), __secend("B")},
};
```

リスト 3-9

_INITSCT 関数をコールするだけで、B セクションの 0 クリア及び、D セクションから R セクションへのコピー処理を行います。_INITSCT を使用する場合には、<_h_c_lib.h>をインクルードし(a)、標準ライブラリをリンクして下さい（「４．６　標準ライブラリ」)。構造体変数(d)、(f)は、(c)、(e)の #pragma section 指定により、それぞれ C$DSEC セクション、C$BSEC セクションに生成されます。これらは ROM 領域に配置して下さい。

D セクション以外に初期化データ領域がある場合には、下記の様に DTBL に追加をして下さい。また、ROM 化オプションにより初期化データ領域 D1 のコピーとして、初期化データ領域のコピー領域 R1 セクションを作成して下さい。また、D1 セクションを ROM 領域に、R1 セクションを RAM 領域に配置して下さい。

```
#pragma section 1
int a = 0;                                  ...D1 セクション

#pragma section $DSEC
static const struct {
    char *rom_s;
    char *rom_e;
    char *ram_s;
}DTBL[]= {
    {__sectop("D"),__secend("D"), __sectop("R")},
    {__sectop("D1"),__secend("D1"), __sectop("R1")},
};
```

リスト 3-11

B セクション以外に未初期化データ領域がある場合には、下記の様に BTBL に追加をして下さい。また、B1 セクションを RAM 領域に配置して下さい。

```
#pragma section 1
int a;                                  ...B1 セクション

#pragma section $BSEC
static const struct {
    char *b_s;
    char *b_e;
}BTBL[]= {
    {__sectop("B"), __secend("B")},
    {__sectop("B1"), __secend("B1")},
};
```

リスト 3-11

３．８　グローバルクラスオブジェクト（C++言語使用時）

C++言語を使用して開発する際、グローバルに宣言されたクラスオブジェクト（グローバルクラスオブジェクト）が存在する場合があります。下記ソースプログラムにおいて、(a)、(b)がグローバルクラスオブジェクトです。

```
class A
{
...
};

A       g_A;                            ...(a)
A *     g_pA;
static A s_A;                           ...(b)

void main()
{
    A       a;
    A *     p_a;
    static A s_a;

    g_pA = new A;    delete g_pA;
    l_pA = new A;    delete l_pA;
}
```

リスト 3-12

このクラスがコンストラクタを持っていた場合、そのクラスのメンバにアクセスする前に、既にコンストラクタが呼ばれている状況でなければなりません。例えば、下記 C++プログラムにおいて、(e)を実行する前に(c)が処理され、(d)のメンバ変数 a が 1 に初期化されていなければなりません。つまり、(c)のコンストラクタが呼ばれている必要があります。

```
class A
{
private:
    int a;
public:
    A(void) { a = 1; }                  ...(c)
    int Get(void) { return a; }
};

A g_a;                                  ...(d)

void main()
{
    int a = g_a.Get();                  ...(e)
}
```

リスト 3-13

このコンストラクタコールのために、_CALL_INIT 関数が標準ライブラリとして準備されています。また、同様に、グローバルクラスオブジェクトのデストラクタを呼ぶための関数_CALL_END 関数も準備されています。_CALL_INIT 関数及び、_CALL_END 関数は、<_h_c_lib.h>に宣言されていますので、使用するソースファイル内で、<_h_c_lib.h>をインクルードします(f)。また、アプリケーションの開始前に_CALL_INIT 関数をコールし(g)、アプリケーションの終了後に_CALL_END 関数をコールして下さい(h)。

```
#include <_h_c_lib.h>                           ...(f)

void PowerON_Reset(void)
{
    _INITSCT();
    _CALL_INIT();                               ...(g)

    main();

    _CALL_END();                                ...(h)
    sleep();
}
```

リスト 3-14

また、コンストラクタやデストラクタをコールするための情報が C$INIT セクションに生成されています。このセクションはコンパイラにより自動的に生成されます。最適化リンケージエディタのメモリ配置設定により、C$INIT セクションを ROM 領域に配置して下さい。

３．９　ランタイム関数と標準ライブラリ

使用できるレジスタや CPU 命令は、CPU によって限定されています。例えば、H8/300H マイコンには long 型の加算命令を持っているので long 型の加算を比較的少ない命令で実現できます。対して、H8/300 マイコンでは long 型の加算命令を持っていないため、数命令～数十命令の CPU 命令を組み合わせる事によって処理を行わなければなりません。シフト演算、他の四則演算も同様です。

これらを担当するのがランタイム関数です。一般的に、C/C++言語で開発する際にはランタイム関数が必要です。ランタイム関数は、標準ライブラリに含まれています。標準ライブラリ関数を使用する場合には、標準ライブラリをリンクする必要があります（「４．６　標準ライブラリ」）。

下記例で、long 型の加算を使用したときのコード生成を説明します。

```
long a,b,c;
void main()
{
    c = a + b;
}
```

リスト 3-15

下記は H8/300H 使用時のコード生成です。

```
_main
        MOV.L       @_a,ER0
        MOV.L       @_b,ER1
        ADD.L       ER1,ER0
        MOV.L       ER0,@_c
        RTS
```

リスト 3-16

下記は H8/300 使用時のコード生成です。**$ADDL** がランタイム関数であり、標準ライブラリからコールされるようになります。

```
_main
        PUSH.W      R2
        MOV.W       #_a,R1
        MOV.W       #_b,R2
        MOV.W       #_c,R0
        JSR         @$ADDL$3:16
        POP.W       R2
```

リスト 3-17

３．１０　低水準インターフェースルーチン

C/C++言語で開発をすると、標準入出力ライブラリ（fopen、fprintf、printf など）、メモリ管理ライブラリ（malloc、free、new、delete）といった関数を使用する場合があります。しかし、これらの機能はコンパイラで全て提供している訳ではありません。例えば、標準出力と言っても、その出力先は LCD、LED、HDD、FDD、プリンタ、CD-R/RW など様々です。また、標準入力と言っても、DIP スイッチ、キーボード、マウス、携帯電話のボタン、タッチパネルなど様々です。また、それら機器により当然操作が異なります。従って、標準入出力、メモリ管理ライブラリの処理全てをコンパイラ

側で提供するのは不可能という事になります。そこで標準入出力、メモリ管理ライブラリからは、低水準インターフェースルーチンと呼ばれる関数群を呼ぶようにしています。低水準インターフェースルーチンはユーザにて実装して頂く必要があります。しかしこれにより、入出力機器などのハードウェア操作をユーザが任意に実装する事が可能になります。低水準インターフェースルーチンには open、close、read、write、lseek、sbrk、errno_addr、wait_sem、signal_sem があります。実装方法はコンパイラのユーザーズマニュアルをご参照下さい。なお、errno_addr、wait_sem、siglan_sem は、SHC コンパイラ Ver.7 以降で、かつ、標準ライブラリ構築時に-reent を指定した場合（リエントラントライブラリを指定した場合）に必要になる関数です。また、プログラムの終了処理を行うライブラリ（exit、atexit、abort）もユーザにて実装して頂く必要があります。

4　HEW 生成ファイルの説明

4．1　プロジェクトジェネレータ設定

本書では、プロジェクトジェネレータ（HEW の[ファイル → 新規プロジェクト]メニューで起動）で下記手順に従って生成された生成ファイルについての説明をします。なお、プロジェクトジェネレータ使用方法は製品同梱の電子マニュアル(PDF ファイル)のチュートリアルも併せてご参照下さい。

(1) 新規ワークスペースの作成

ここでは、プロジェクトの種類として「Application」を選択します。

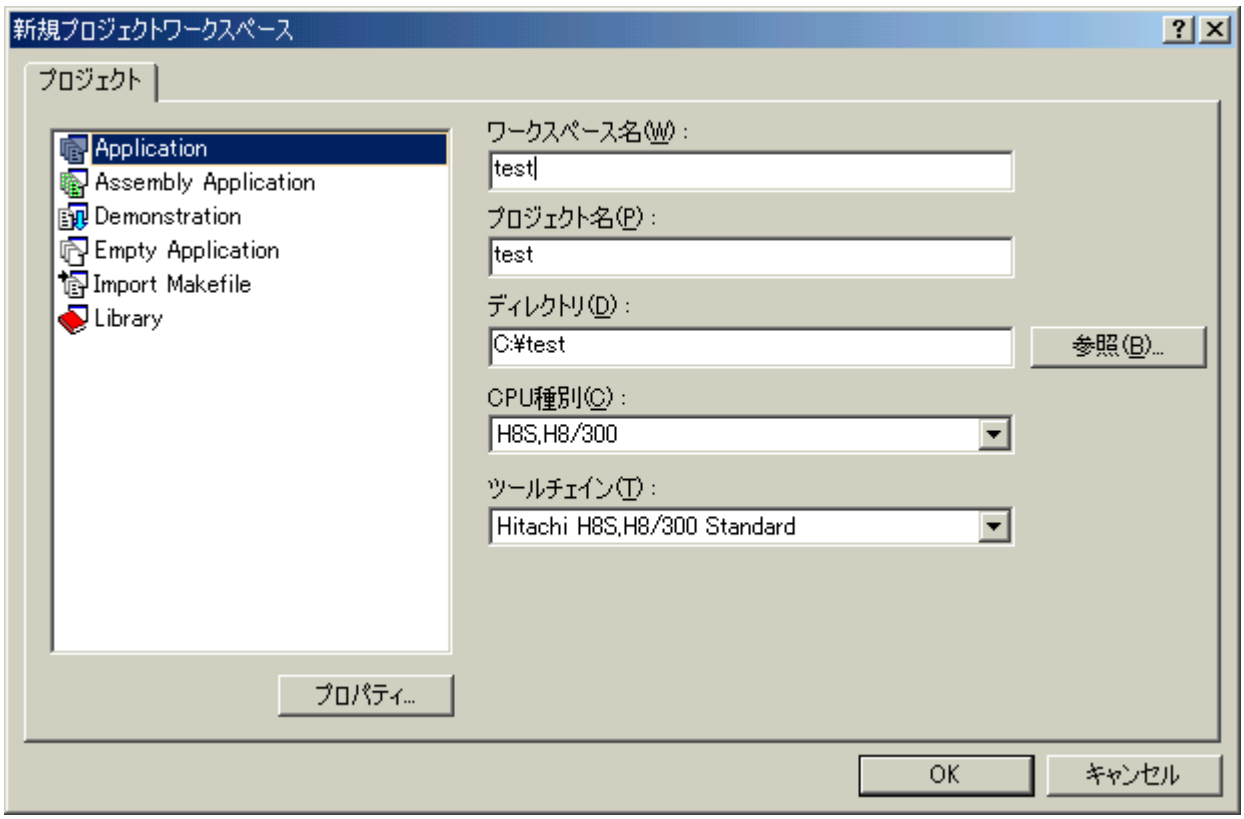

図 4-1

Application 以外のプロジェクトの種類を選択した場合、以降の章で説明するチェックボックスなどの選択項目が選択できない場合があります。

(2) CPU の選択

ここでは、[CPU Series]に「2600」を選択し、[CPU Type]に「2612」を選択します。他の CPU Series（SH-1、SH2-DSP、SH-2E）や、他の CPU Type を選択しても、基本的な生成ファイルは同じです。

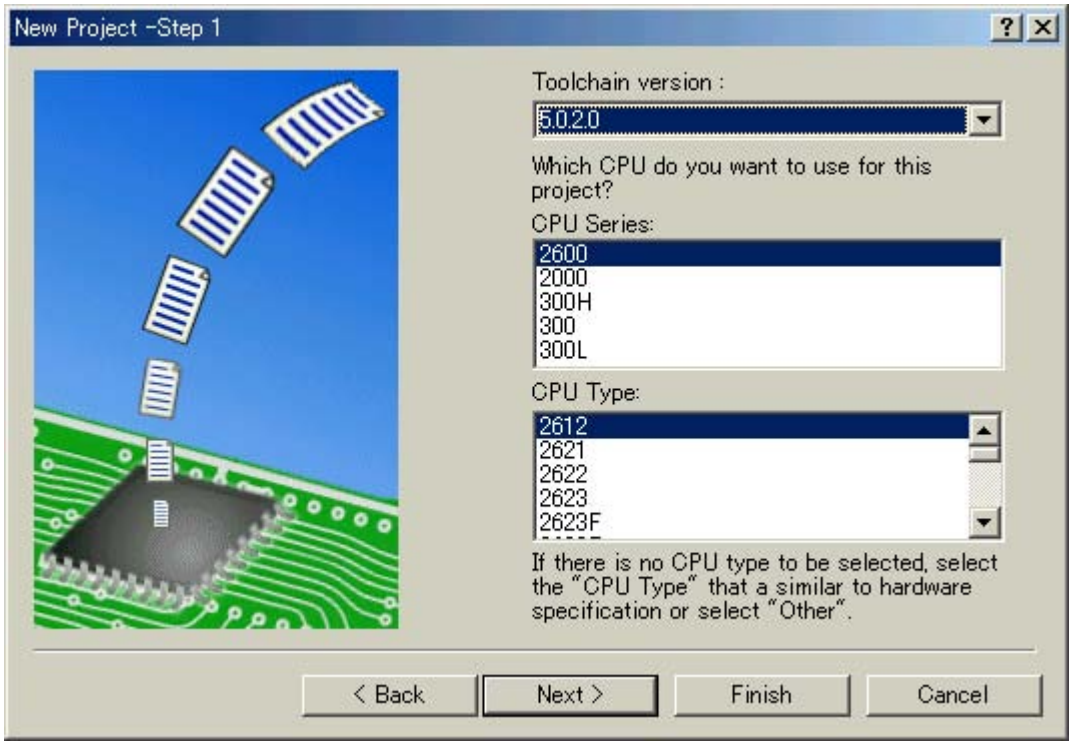

図 4-2

- [CPU Series]の設定は、CPU オプションに反映されます（「４．３　CPU オプション」）。
- [CPU Type]の設定は、iodefine.h ファイルの記述内容や、最適化リンケージエディタのメモリ配置設定に反映されます。

(3) オプション設定

ここでは、特に何も指定を変更せずに先に進みます。

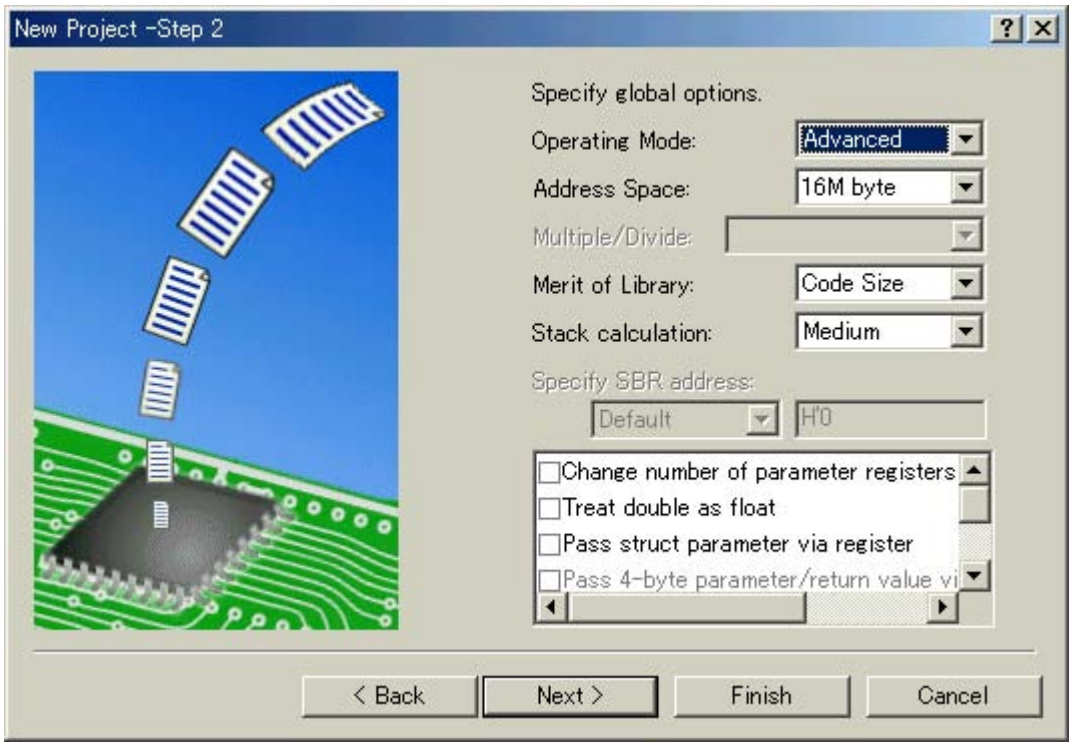

図 4-3

・ このダイアログの設定は、すべて、CPU オプションに反映されます。「(2)CPU の選択」の選択により、選択できる箇所が変わります。

(4) 生成ファイルの設定

ここでは、[Use I/O Library]にチェックを付け、[Number of I/O Streams]に「20」を指定します。また、[Generate Hadware Setup Function]に「C/C++ source file」を指定します。

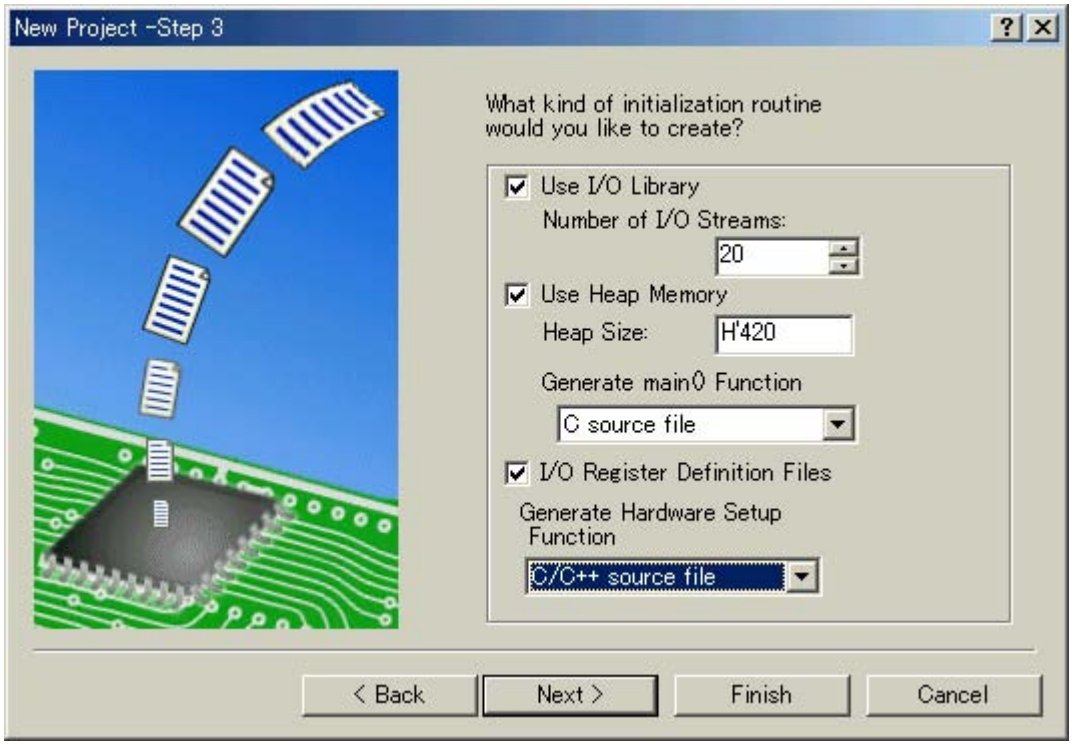

図 4-4

- [Use I/O Library]にチェックを付けると、入出力関連の低水準インターフェースルーチン（open、close、write、read、lseek）のサンプルプログラム及び、標準ライブラリの初期化プログラム（_INIT_IOLIB、_CLOSEALL）が記述された lowlvl.src、lowsrc.c、lowsrc.h が生成されます。もし入出力関連の標準ライブラリを使用しない場合には、このチェックを外すか、ワークスペース構築後に、lowlvl.src、lowsrc.c、lowsrc.h を HEW のプロジェクトから削除して下さい。
- [Number of I/O Stream]の設定は、lowsrc.h に反映されます。
- [Use Heap Memory]にチェックを付けると、メモリ管理関連の低水準インターフェースルーチン（sbrk）のサンプルプログラムが記述された sbrk.h、sbrk.c が生成されます。もしメモリ管理関連の標準ライブラリを使用しない場合には、このチェックを外すか、ワークスペース構築後に、sbrk.h、sbrk.c を HEW のプロジェクトから削除して下さい。
- [Heap Size]の設定は、sbrk.h に反映されます。
- [Generate main() Function]の設定によりメイン関数（C ソースファイルもしくは C++ソースファイル）及び、低水準インターフェースルーチン（exit）が生成されます。
- [I/O Register Definition Files]のチェックを付けると、iodefine.h が生成されます。
- [Generate Hardware Setup Function]の設定により、hwsetup.c や hwsetup.cpp、hwsetup.src が生成されます。

(5) 標準ライブラリの設定

ここでは、特に何も指定を変更せずに先に進みます。

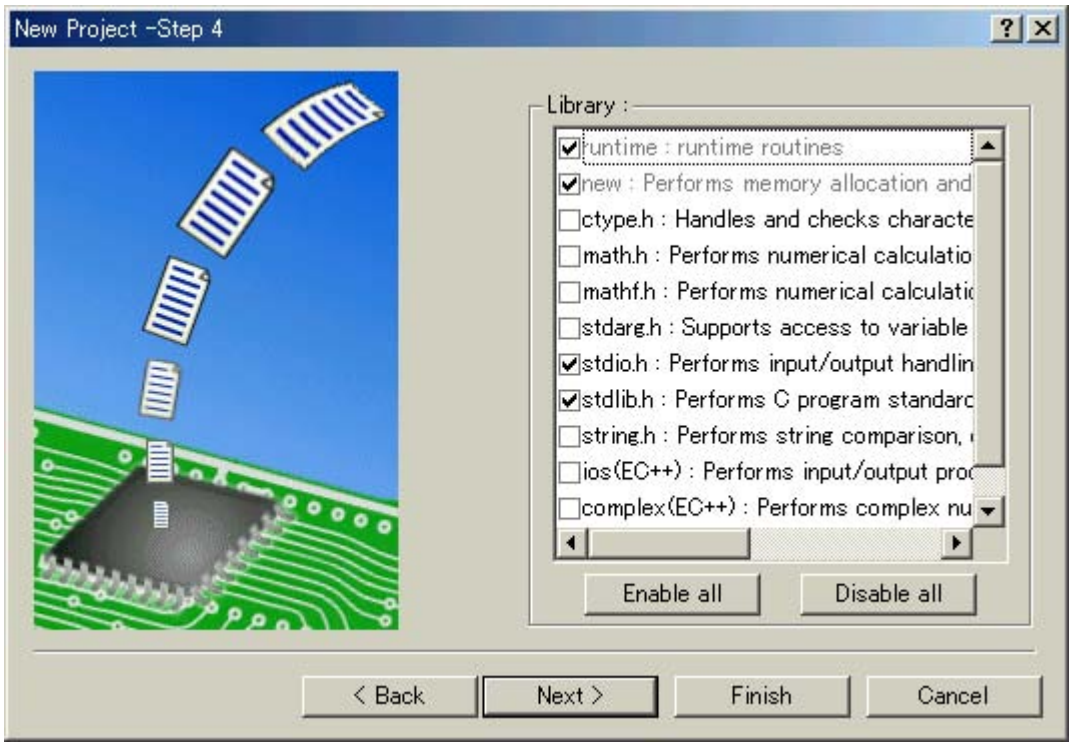

図 4-5

・ このダイアログでの設定は、すべて、標準ライブラリの設定に反映されます。

(6) スタック領域の設定

ここでは、特に何も指定を変更せずに先に進みます。

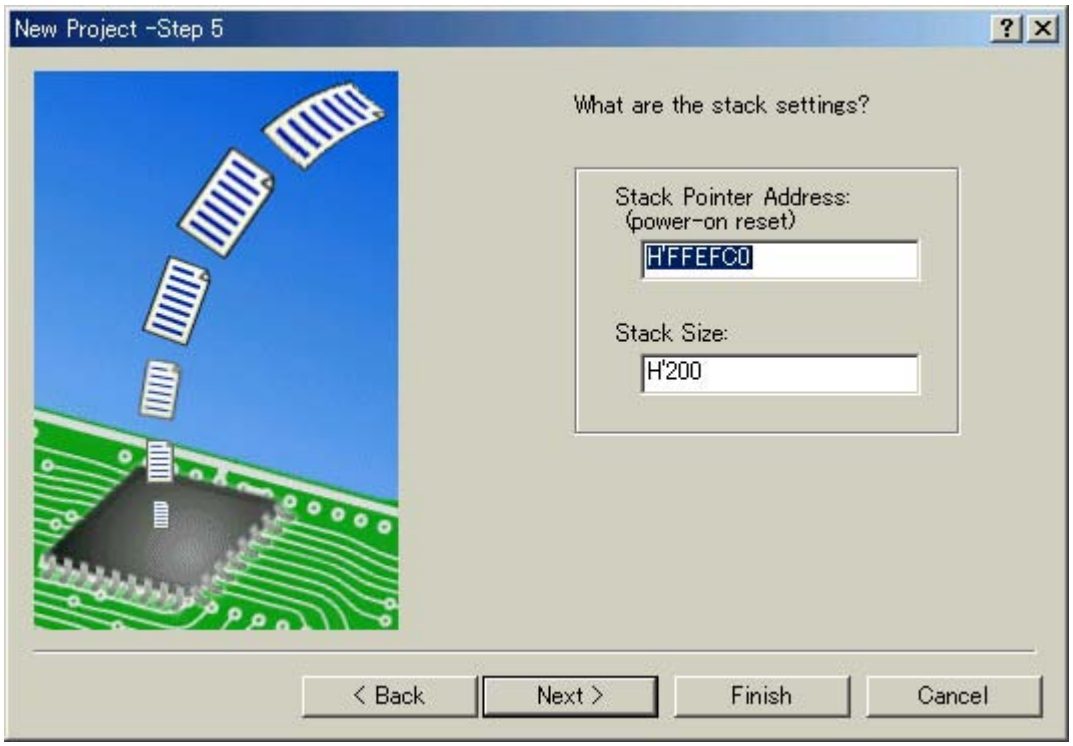

図 4-6

- [Stack Pointer Address]の設定は、最適化リンケージエディタのメモリ配置設定に反映されます。
- [Stack Size]の設定は、stacksct.h に反映されます。
- ただし、「(7) ベクタの設定」で[Vector Definition Files]のチェックを外すと、stacksct.h は生成されません。

(7) ベクタの設定

ここでは、特に何も指定を変更せずに先に進みます。

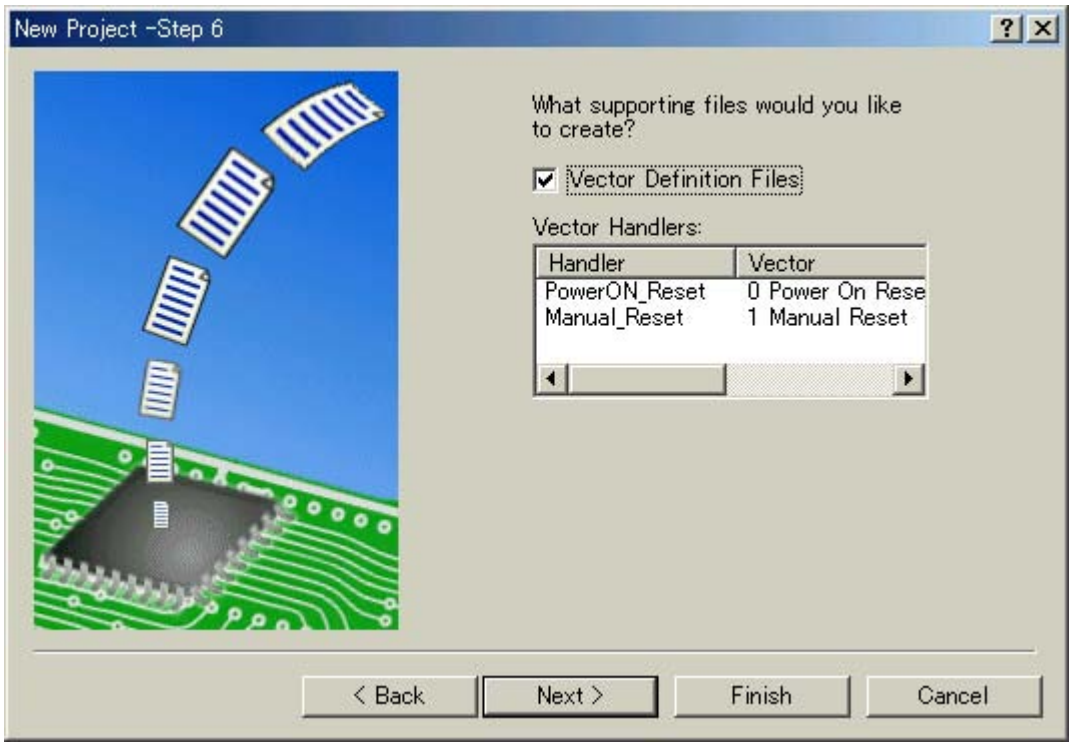

図 4-7

・ [Vector Definition Files]にチェックを付けると、intprg.c、resetprg.c、stacksct.h、が生成されます。

(8) デバッガターゲットの設定

ここでは、特に何も指定せずに先に進みます。

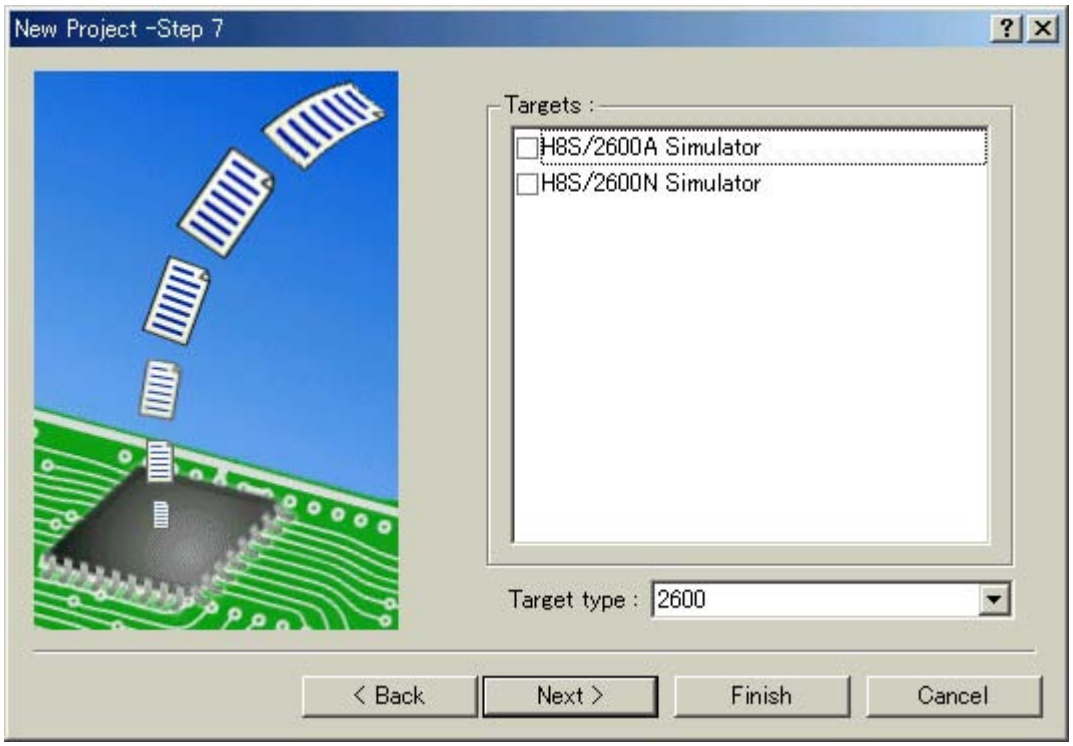

図 4-8

(9) 生成ファイル名の変更

ここでは、特に何も指定を変更せずに先に進みます。

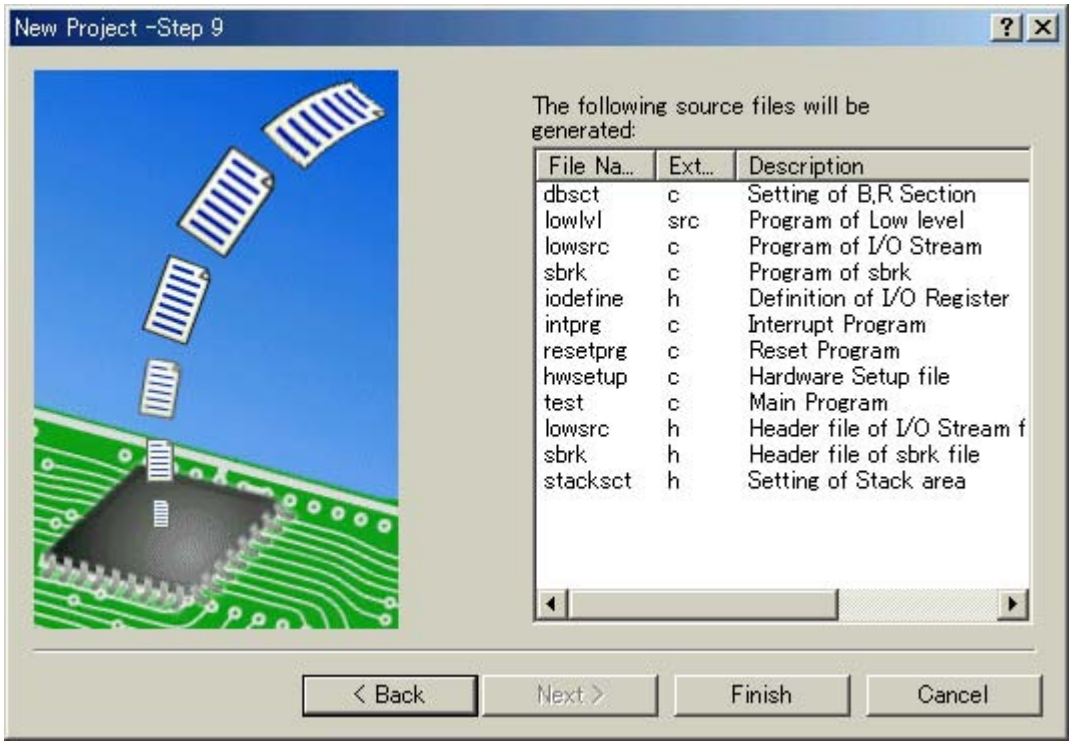

図 4-9

４．２　生成フォルダ及び、生成ファイル一覧

「４．１　プロジェクトジェネレータ設定」によって生成されたフォルダ構成や生成ファイルは下記の様になります（¥xxx はフォルダを意味します）。

C:¥test		ワークスペースフォルダです。 (1)[ディレクトリ]で指定したフォルダ名になります。
	test.hws	ワークスペースファイルです。 (1)「ワークスペース名」で指定した名称になります。
	¥test	プロジェクトフォルダです。 (1)[プロジェクト名]で指定したフォルダ名になります。
	dbsct.c	必ず生成されます。
	hwsetup.c (hwsetup.src 、 hwsetup.cpp)	(4)[Generate Hardware Setup Function]の指定で生成されます。
	intprg.c	(7)[Vector Definition Files]の指定で生成されます。
	iodefine.h	(4)[I/O Register Definition Files]の指定で生成されます。
	lowlvl.src	(4)[Use I/O Library]の指定で生成されます。
	lowsrc.c	(4)[Use I/O Library]の指定で生成されます。
	lowsrc.h	(4)[Use I/O Library]の指定で生成されます。 (4)[Number of I/O Stream]の指定が反映されます。
	resetprg.c	(7)[Vector Definition Files]の指定で生成されます。
	sbrk.c	(4)[Use Heap Memory]の指定で生成されます。
	sbrk.h	(4)[Use Heap Memory]の指定で生成されます。 (4)[Heap Size]の指定が反映されます。
	stacksct.h	(7)[Vector Definition Files]の指定で生成されます。 (6)[Stack Size]の指定が反映されます。
	test.c (test.cpp)	(4)[Generate main() Function]の指定で生成されます。 (1)[プロジェクト名]で指定したファイル名称になります。
	test.hwp	HEW プロジェクトファイルです。 (1)「プロジェクト名」で指定した名称になります。
	¥Debug	Debug コンフィギュレーション選択時、このフォルダに中間ファイルやアブソリュートファイルが生成されます。
	¥Release	Release コンフィギュレーション選択時、このフォルダに中間ファイルやアブソリュートファイルが生成されます。

表 4-1

４．３　CPU オプション

HEW メニュー：[オプション → H8S,H8/300 Standard Toolchain]
→ [CPU]タブを選択
により、下記ダイアログが表示されます。

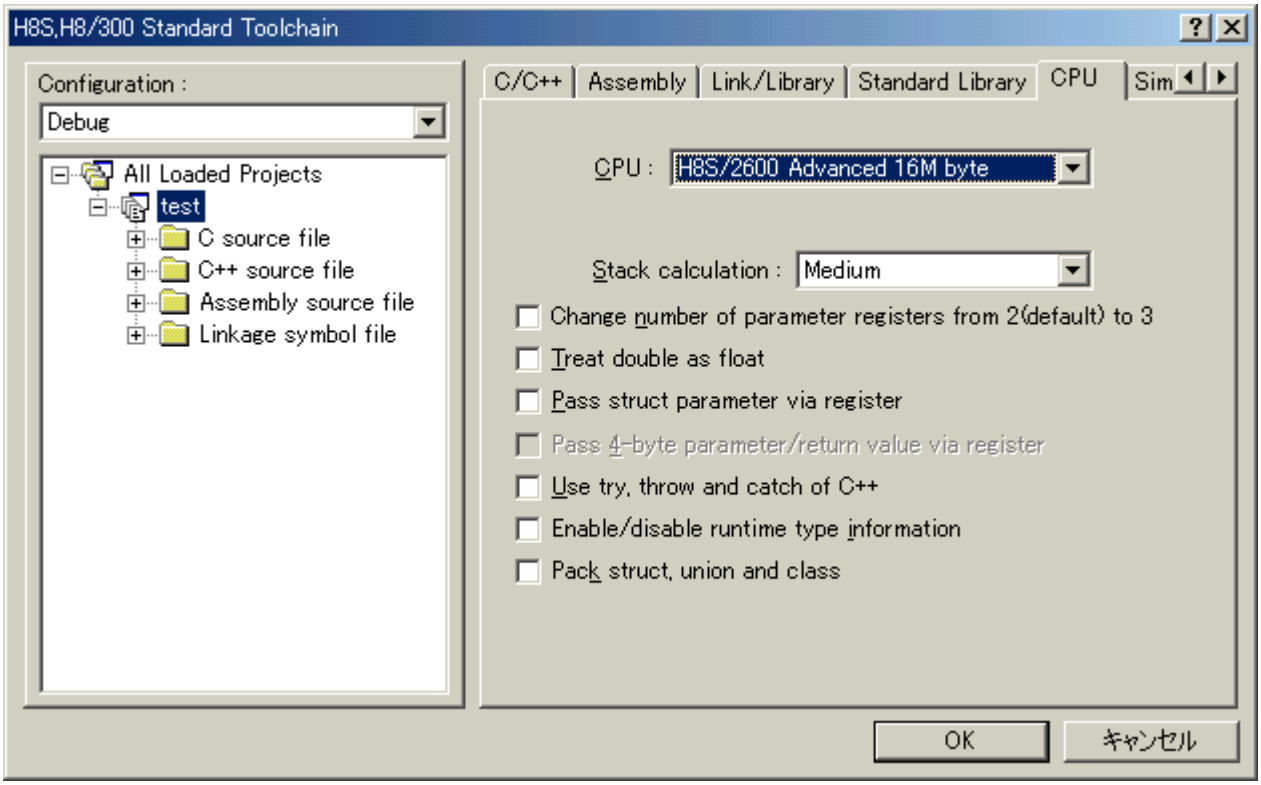

図　4-10

「(2)　CPU の選択」、「(3)　オプション設定」の設定が反映されます。

４．４　ROM 化オプション

HEW メニュー：[オプション → H8S,H8/300 Standard Toolchain]

→ [Link/Library]タブを選択

→ [Category]に”Output”を選択

→ [Show entries for]に”ROM to RAM mapped sections”を選択

により、下記ダイアログが表示されます。

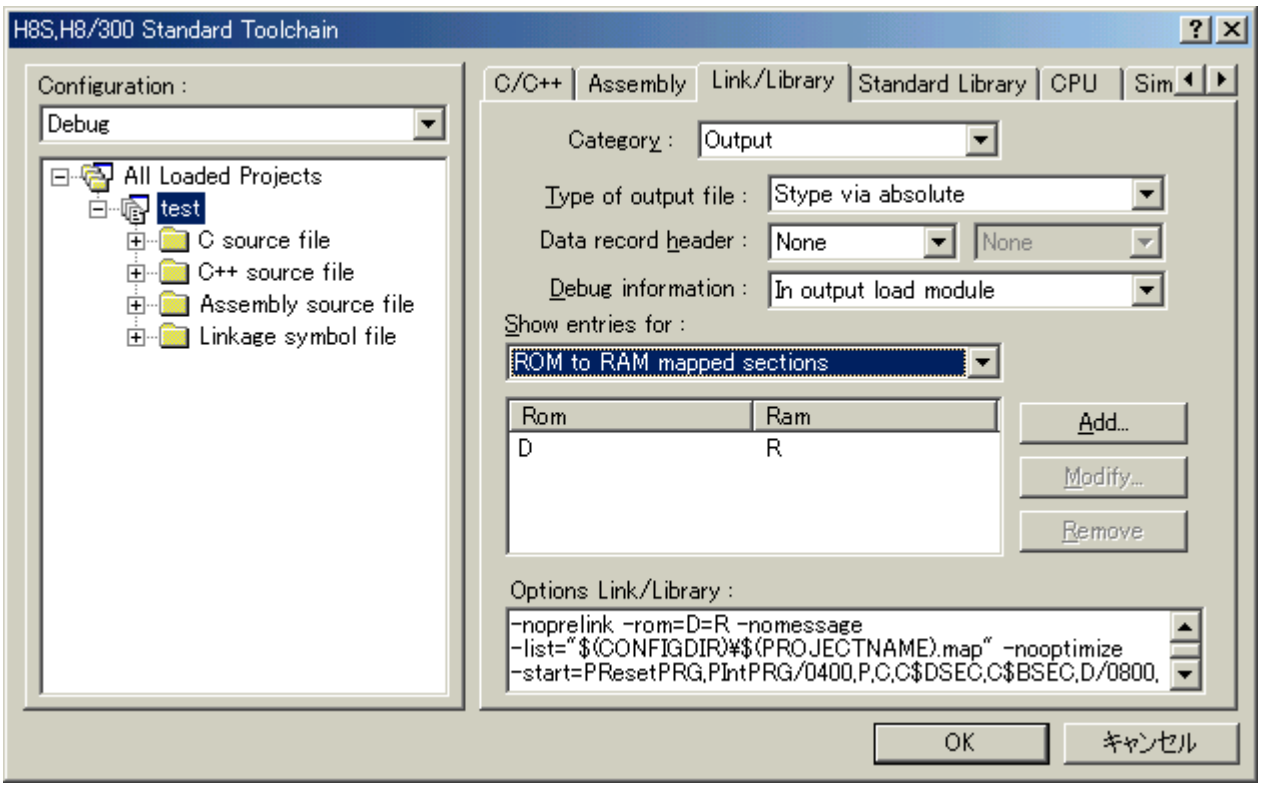

図　4-11

最適化リンケージエディタの ROM 化オプションを使用する場合には、この設定を変更して下さい。

４．５　メモリ配置設定

HEW メニュー：[オプション → H8S,H8/300 Standard Toolchain]

→ [Link/Library]タブを選択

→ [Category]に”Section”を選択

→ [Show entries for]に”Section”を選択

により、下記ダイアログが表示されます。

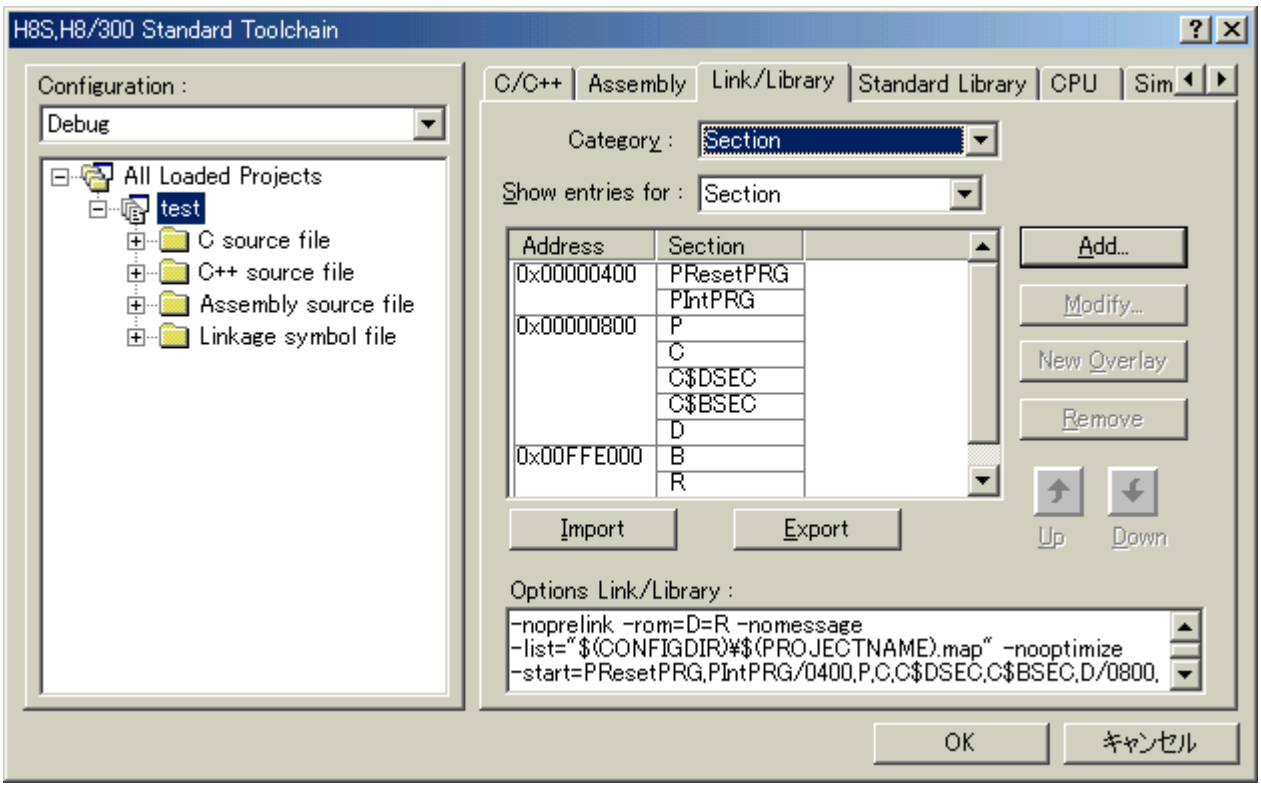

図　4-12

H8S/2612 では、下記の様にメモリ設定されています。メモリ配置設定を行う場合は、適宜この設定を変更して下さい。

アドレス	セクション名
0x00000400	PResetPRG
	PIntPRG
0x00000800	P
	C
	C$BSEC
	C$DSEC
	D
0x00FFE000	B
	R
0x00FFEDC0	S

表　4-2

・ ビルドを行った時、「L1120 (W) Section address is not assigned to “xxx”」と表示される場合には、このダイアログで xxx セクションを指定し忘れている可能性があります。その場合は、この

ダイアログで xxx セクションを所定のアドレスに配置して下さい。

・ ビルドを行った時、「L1100 (W) Cannot find “xxx” specified in option “start”」と表示される場合には、このダイアログで xxx セクションを余分に設定している可能性があります。その場合は、このダイアログから xxx セクションの指定を削除して下さい。なお、HEW が自動生成する設定では C セクションを配置するようにしていますが、生成されるファイルによっては C セクションが存在せず L1100 が出力される場合があります。

４．６　標準ライブラリ

(1) 生成オプション

HEW メニュー：[オプション → H8S,H8/300 Standard Toolchain]

→ [Standard Library]タブを選択

→ [Category]に”Mode”を選択

により、下記ダイアログが表示されます。

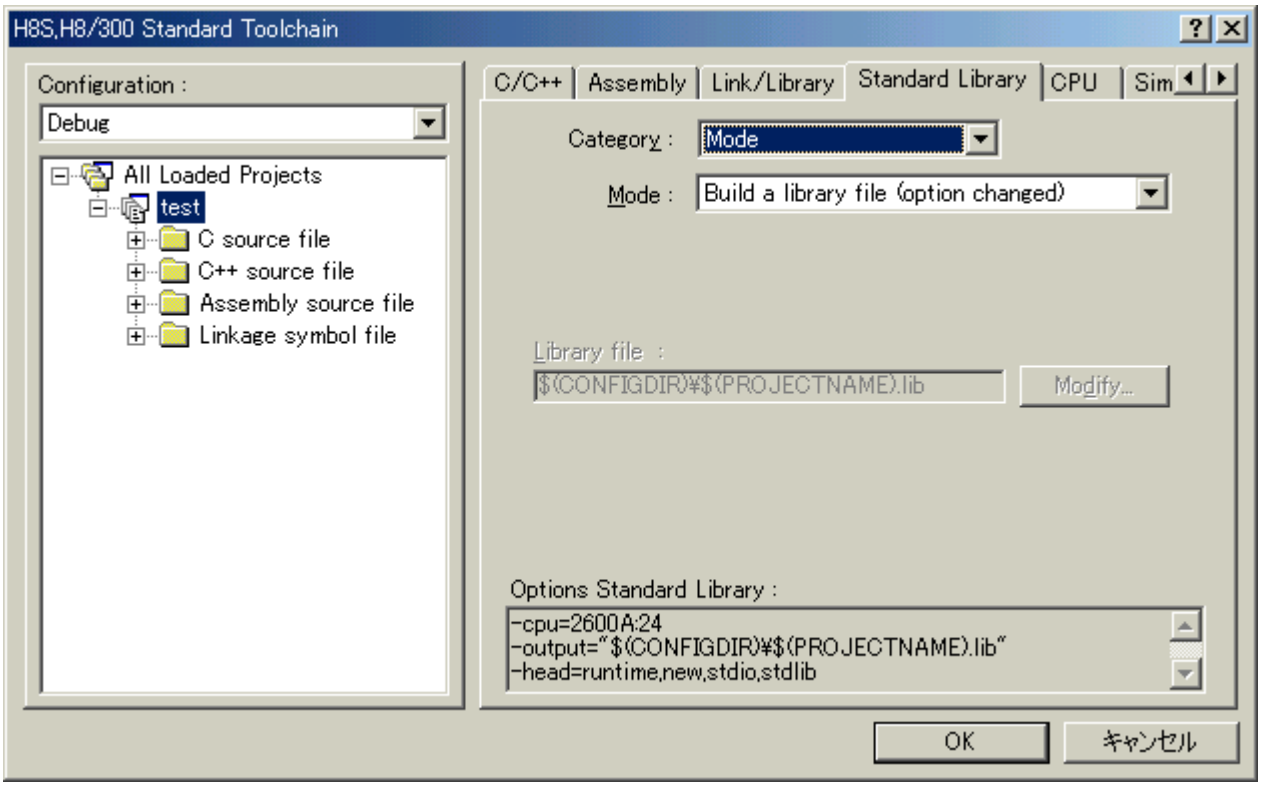

図 4-13

標準ライブラリをリンクする場合には、[Mode]に”Build a library file (option changed)”もしくは、”Build a library file (anytime)”を選択して下さい。

(2) 対象ヘッダファイル

HEW メニュー：[オプション → H8S,H8/300 Standard Toolchain]

→ [Standard Library]タブを選択

→ [Category]に”Standard Library”を選択

により、下記ダイアログが表示されます。

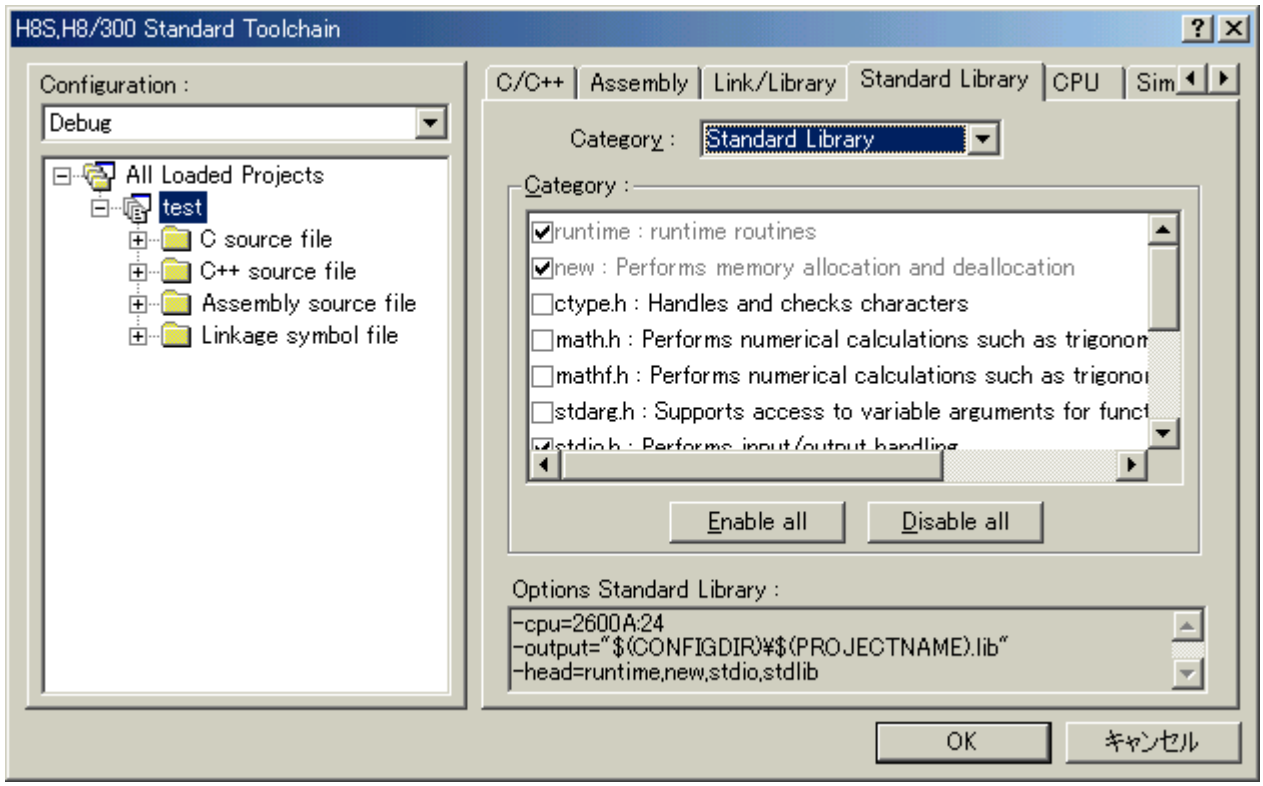

図 4-14

必要な標準ライブラリをヘッダファイル単位でチェックして下さい。

４．７　コンフィギュレーション

コンパイラ、アセンブラ、最適化リンケージエディタのオプションは、コンフィギュレーションという単位でプロジェクトに保存されます。コンフィギュレーションを切り替える事で、これらのオプションの呼び出し、切り替えができるようになります。

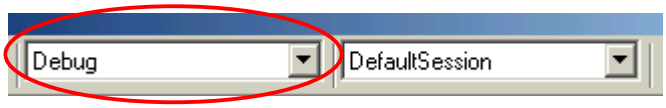

図　4-15

Debug、Release 及び、デバッグセッション用のコンフィギュレーションが生成されます。ただし、デバッグセッション用のコンフィギュレーションは、プロジェクトジェネレータの設定で「(8)　デバッガターゲットの設定」を設定した時に生成されます。デバッガターゲットに依存してコンパイラなどの設定が変わる場合は、デバッグセッション用のコンフィギュレーションを使用してください。
HEW が生成したコンフィギュレーションは、全て同じ設定になっています。ただし、Release コンフィギュレーションだけは、デバッグ情報を出力しない設定になっています。デバッグ情報は、デバッガでデバッグする際に必要となる付加的な情報であるため、Debug と Release コンフィギュレーションとで出力されるコードが変わる事はありません。
コンパイラなどの設定を変更する場合は、現在使用中のコンフィギュレーションが対象となります。例えば Debug コンフィギュレーションのみ設定を変更するような事があると、Debug とそれ以外のコンフィギュレーションとで、異なるコードが生成される事になりますので、注意が必要です。オプションを変更する画面で、下記の様に複数のコンフィギュレーションも設定対象とする事ができます。これにより、複数のコンフィギュレーションに対して纏めてオプションを変更する事ができます。

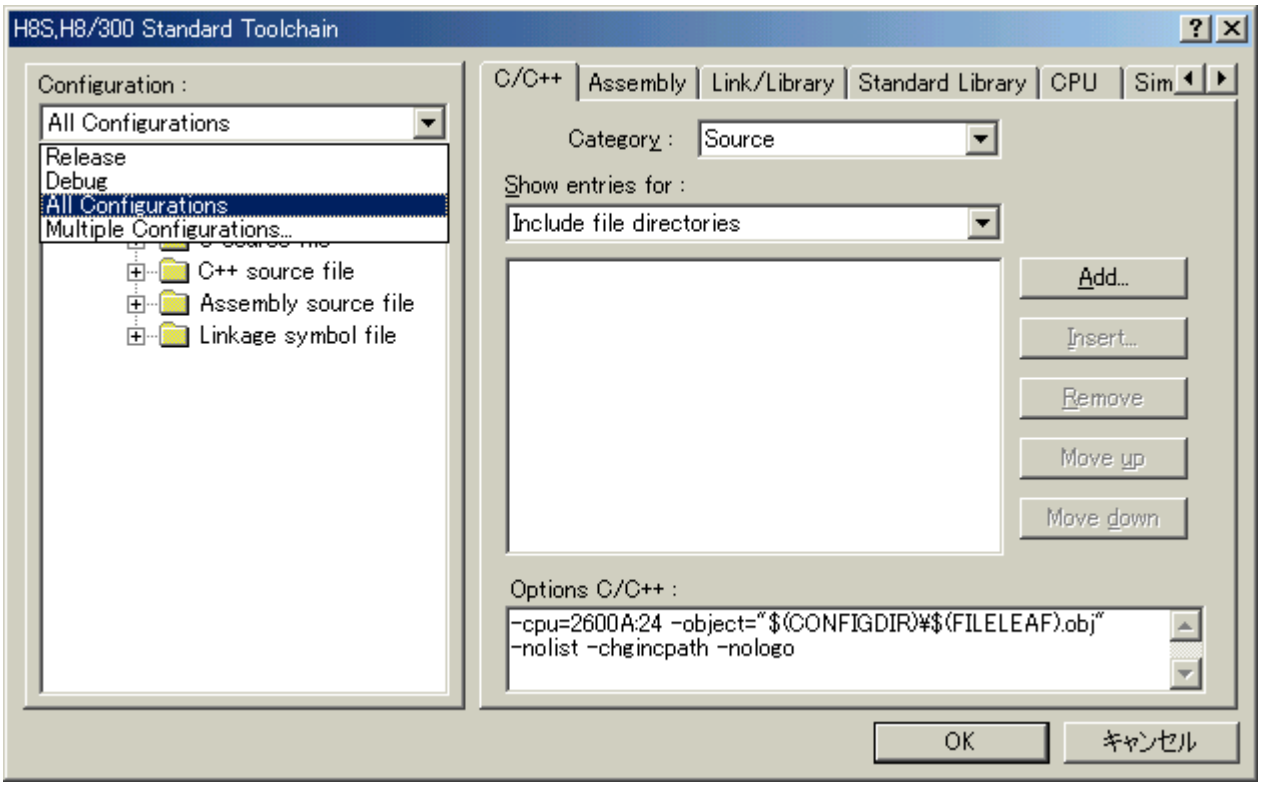

図　4-16

４．８　構成の編集

HEW メニュー：[プロジェクト → 構成の編集]にて、HEW が生成した設定などを変更する事ができます。

(1) I/O Register

[I/O Register]タブの[I/O Register Definition Files (Overwrite)]にチェックを付けて[OK]を押すと、iodefine.h を生成する事ができます。プロジェクトジェネレータ設定の「(4)　生成ファイルの設定」の[I/O Register Definition Files]にチェックしなかった場合や、iodefine.h を復元したい場合に使用してください。

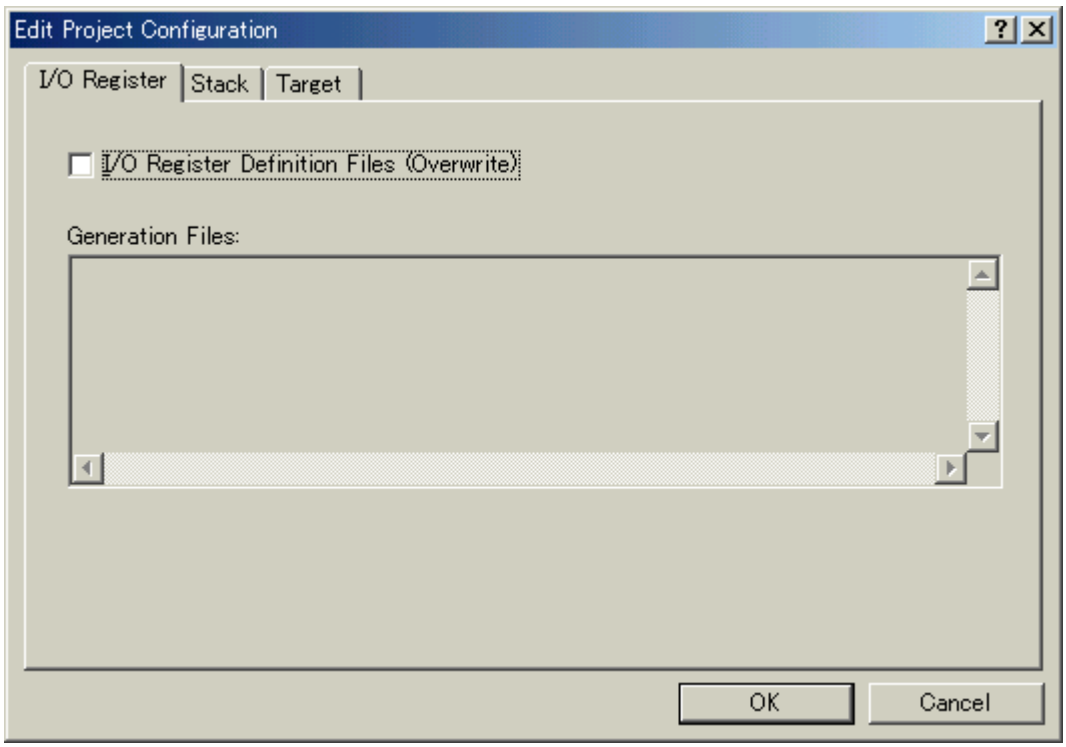

図　4-17

(2) Stack

[Stack]タブの設定により、スタック領域の再設定をする事ができます。プロジェクトジェネレータ設定の「(6)　スタック領域の設定」の設定を変更する場合に使用してください。この設定は、最適化リンケージエディタのメモリ配置設定や、stacksct.h に再設定されます。ただし、最適化リンケージエディタのメモリ配置設定で S セクションのアドレスをユーザが変更したり、stacksct.h の #pragma stacksize の記述をユーザが変更した場合には、この機能は使用できなくなります。

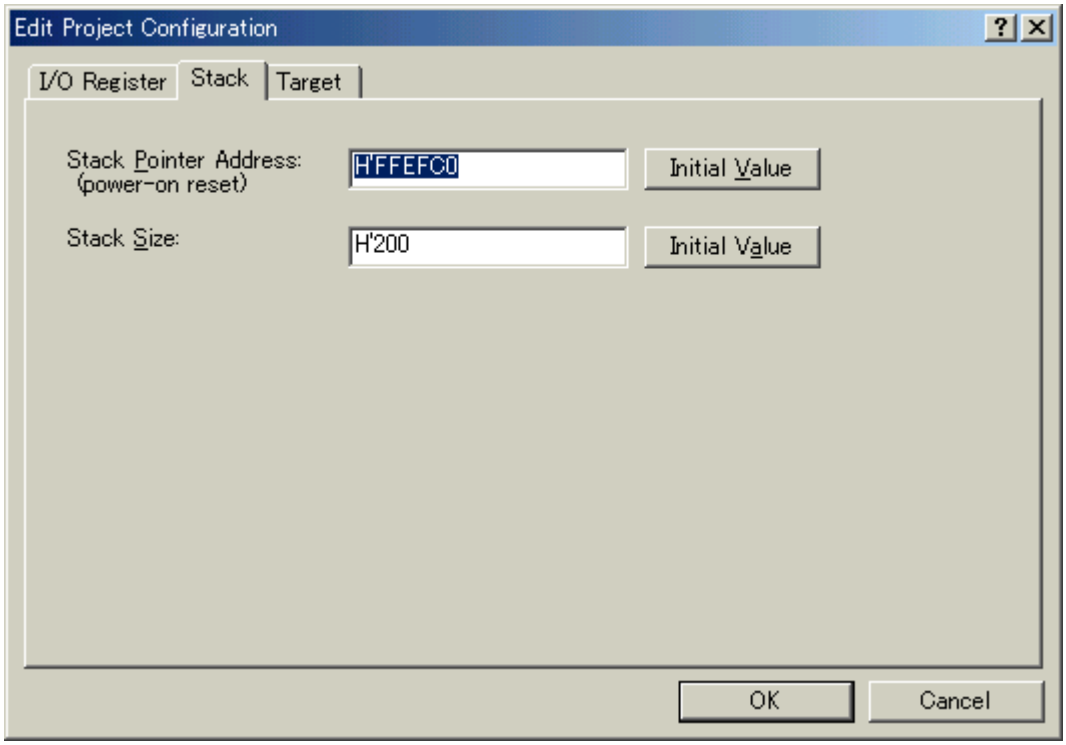

図 4-18

(3) Target

[Target]タブの設定により、デバッガターゲットの再設定を行う事ができます。プロジェクトジェネレータの「(8) デバッガターゲットの設定」に設定をしなかった場合などに使用してください。

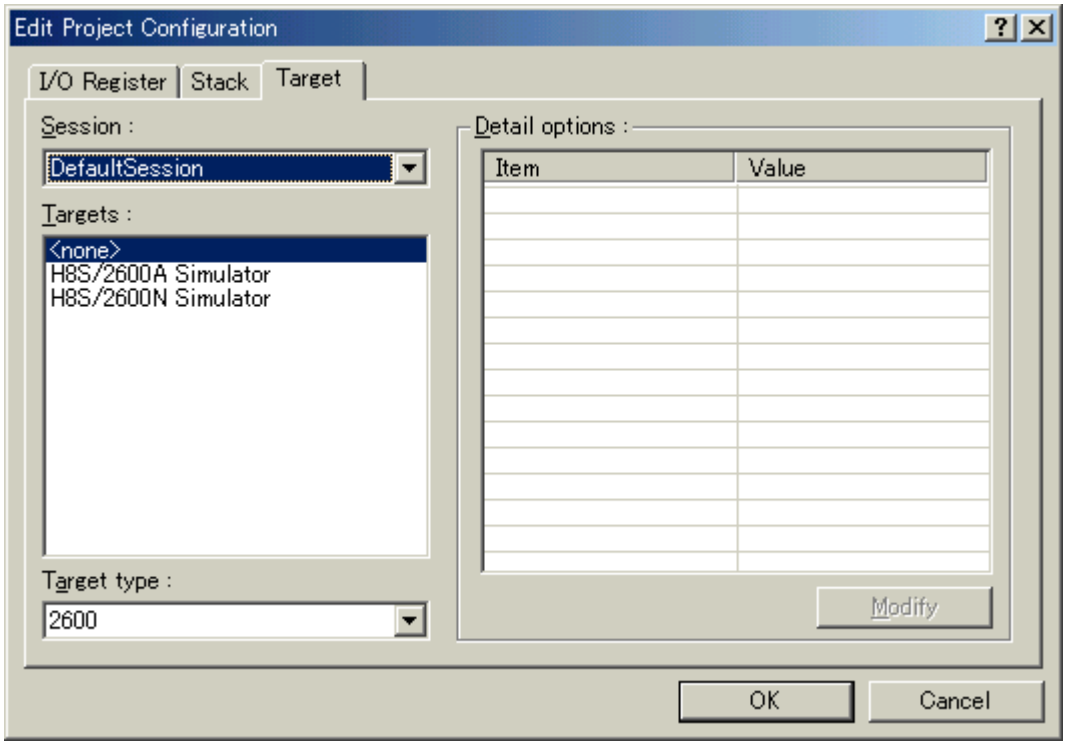

図 4-19

４．９ 低水準インターフェースルーチン

(1) メモリ管理関連

sbrk.c には、メモリ管理ライブラリの低水準インターフェースルーチンとして、sbrk 関数が記述されています。ユーザシステムに合わせて適宜修正して下さい。malloc や new が呼ばれると、sbrk 関数が呼ばれます。メモリ管理ライブラリを使用しない場合は、このファイルをプロジェクトから削除する事ができます。

sbrk.h には、ヒープ領域の全体サイズを指定する HEAPSIZE マクロが定義されています。ヒープ領域の全体サイズを変更する場合には、HEAPSIZE マクロを修正して下さい。

(2) 入出力関連

lowlvl.src、lowsrc.c、lowsrc.h には、入出力関連の低水準インターフェースルーチンが記述されています。ユーザシステムに合わせて適宜修正して下さい。

lowsrc.c には、ファイルハンドラ（入出力ストリーム）の初期化ルーチンとして、_INIT_IOLIB、_CLOSEALL 関数が定義されています。_INIT_IOLIB 関数では、ファイルハンドラの初期化と標準入力(stdin)、標準出力(stdout)、標準エラー出力(stderr)用のファイルのオープンを行っています。標準入力、標準出力、標準エラー出力を使用しない場合は、これらのオープン処理を削除して下さい。

_INIT_IOLIB 関数以外の場所では、ファイルハンドラの操作は行わないで下さい。ファイルハンド

ラのメンバ変数 _bufptr、_bufcnt、_bufbase、_buflen は、ファイルオープン後に setbuf 関数または setvbuf 関数を使用して設定して下さい。_CLOSE 関数では、閉じていないファイルを全て閉じる処理が記述されています。

lowsrc.h には、ファイルハンドラの数を指定する IOSTREAM が定義されています。ファイルハンドラの数を変更する場合には、IOSTREAM マクロを修正して下さい。なお、HEW が生成した lowsrc.c では、_INIT_IOLIB 関数内で、上記の通り 3 つのファイルハンドラをオープンしています。従って、これらのオープン処理が有効になっている状況では、ユーザが使用できるファイルハンドラの数は、（IOSTREAM - 3）となります。

lowsrc.c には、入出力ライブラリの低水準インターフェースルーチンとして、open、close、read、write、lseek 関数が定義されています。

- open 関数では、ファイルオープン要求が標準入力／標準出力／標準エラー出力かの判別と、ファイルモードのチェックを行っています。これ以外の入力に対してはエラーとして –1 を返します。
- close 関数では、ファイル番号の範囲チェックとファイルモードのクリアを行います。ファイル番号が範囲エラーの場合は、エラーとして –1 を返します。
- read 関数では、ファイルモードのチェックを行った後、実際に文字を取得する charget 関数を要求された文字数分コールします。エラーの場合は、–1 を返します。
- write 関数では、ファイルモードのチェックを行った後、実際に文字を出力する charput 関数を要求された文字数分コールします。エラーの場合は、–1 を返します。
- HEW が生成する lseek 関数では、何も処理をしていません。

lowlvl.src には、read 関数、write 関数から呼ばれる、charget 関数、charput 関数が定義されています。これらの定義は、シミュレータデバッガ上でのみ動作可能なように実装されています。従って、実際のターゲットでは動作できませんのでご注意下さい。

(3) 終了処理

<プロジェクト名>.c には、終了処理ルーチンとして abort 関数が定義されています。ユーザシステムに合わせて適宜修正して下さい。終了処理が必要ない場合には、abort 関数を削除する事ができます。

以上