M32C シリーズ用 C コンパイラパッケージ V.5.42 Release 00 ガイドブック (第 1 版)

株式会社ルネサス ソリューションズ

2010 年 4 月 1 日

概要

本資料は M32C シリーズ用 C コンパイラパッケージを導入する際の手引きを説明します。C コンパイラパッケージのインストール時、プロジェクトの作成時、等の場合は、このガイドブックをご覧いただきますようお願い申しあげます。

1. C コンパイラパッケージ V.5.42 Release 00 への移行ガイド

C コンパイラパッケージを V.5.20 Release 02 以前から V.5.42 Release 00 に移行する場合の注意事項を説明します。

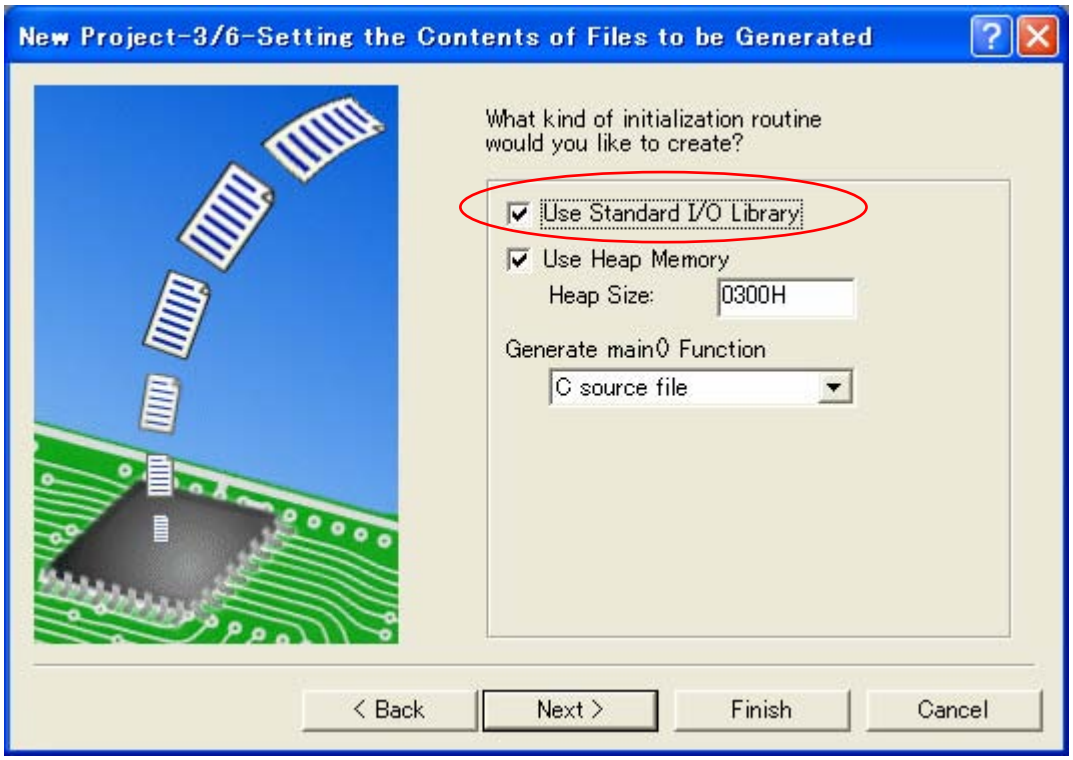
1.1. スタートアッププログラムの変更

V.5.40 Release 00 より、ライブラリ関数 init()の名称は_init()に変更しています。
このため、V.5.20 Release 02 以前のスタートアッププログラムを変更せずにビルドを行った場合、リンクエラー '_init' value is undefined が発生します。

- 発生条件

  次の事項のいずれかひとつに該当している場合に発生します。

  - ◆ V.5.20 Release 02 以前のプロジェクト作成時に[Use Standart I/O Library]を選択している



  - ◆ アセンブリ言語スタートアッププログラム ncrt0.a30 の内容を直接変更している等、init 関数を呼び出している。

- 対処方法

  - ◆ C コンパイラパッケージに付属の ncrt0.a30 を使用している場合
    - 変更前

```
;======================================================================
; Initialize standard I/O
;----------------------------------------------------------------------
.if __STANDARD_IO__ == 1
      .glb    _init
      .call   _init,G
      jsr.a   _init
.endif
```

- 変更後

```
;=====================================================================
; Initialize standard I/O
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
        .glb    __init
        .call   __init,G
        jsr.a   __init
.endif
```

◆ リアルタイム OS に付属の crt0mr.a30 を使用している場合

- 変更前

```
; +-----------------------------------------------------+
; | User Initial Routine ( if there are )               |
; +-----------------------------------------------------+
; Initialize standard I/O
        .GLB    _init
        JSR.A   _init
```

- 変更後

```
; +-----------------------------------------------------+
; | User Initial Routine ( if there are )               |
; +-----------------------------------------------------+
; Initialize standard I/O
        .GLB    __init
        JSR.A   __init
```

1.2. size_t 型、ptrdiff_t 型のビット幅変更

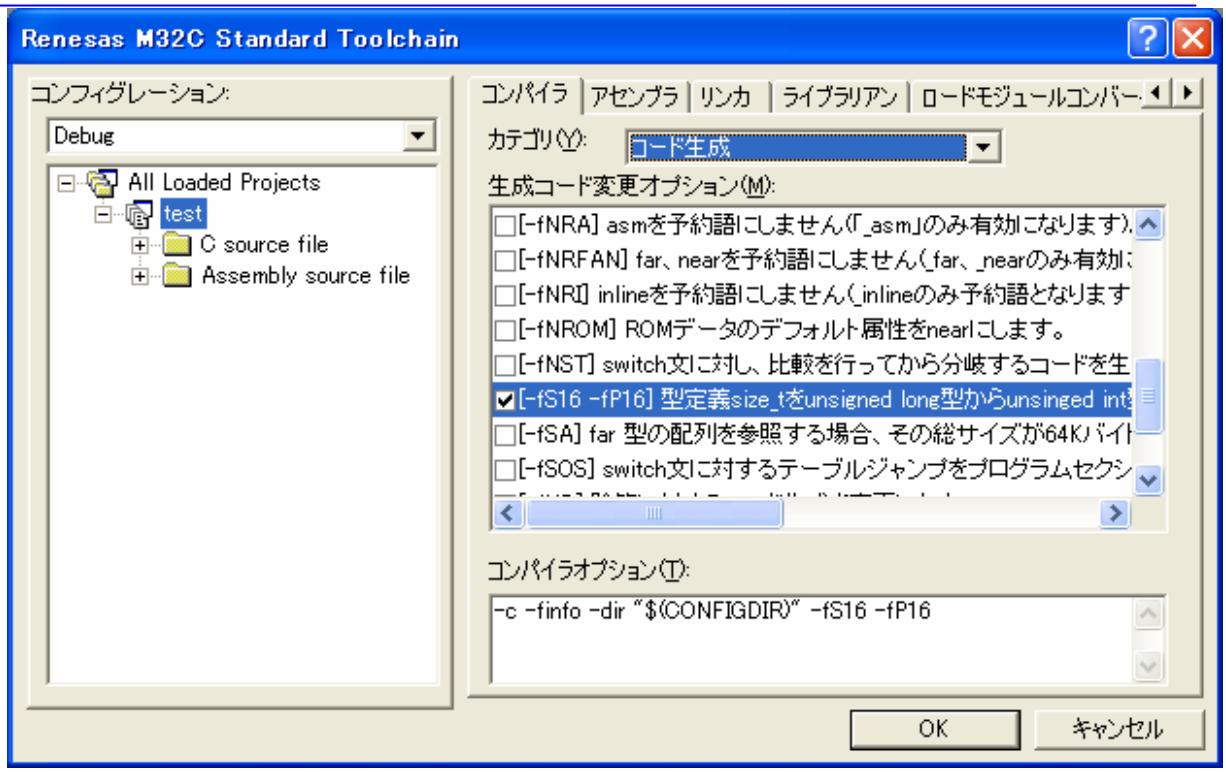
V.5.40 Release 00 から、size_t 型 および ptrdiff_t 型 のビット幅をそれぞれ 16 ビット長から 32 ビット長に変更しています。
V.5.20 Release 02 以前で作成した size_t 型 および ptrdiff_t 型を使用したユーザライブラリを利用する場合等、size_t 型および ptrdiff_t 型を 16 ビット長で使用する場合は、以下の設定を行ってください。

[1] コンパイルオプション-fsizet_16 (-fS16) および-fptrdiff_t (-fP16)を選択する。
[2] リンク時の標準関数ライブラリを変更する。

ターゲットマイコン	選択するライブラリファイル名
M16C/80、/70 シリーズ	nc308_16.lib
M32C/80 シリーズ	nc382_16.lib
M32C/90 シリーズ	nc390_16.lib

- High-performance Embedded Workshop 使用時の設定手順

[1] コンパイルオプション-fsizet_16 (-fS16)および-fptrdiff_t(-fP16)を選択する。
[2] High-performance Embedded Workshop の[ビルド]→[Renesas M32C Standard Toolchain…]→C タブを選択する。
[3] [カテゴリ(Y):]で[コード生成」を選択する。
[4] [生成コード変更オプション(M):]から[-fS16 -fP16]を選択する。



- リアルタイム OS のコンフィグレータにより生成した makefile を使用している場合
  makefile 内の以下の部分を修正してください。
  以下は、ターゲットマイコンが M32C/80 シリーズの場合です。
  - ◆ 変更前

```
# Use the following macro when you use C-libraries for M32C/80 series.
#NEWLIB = -l nc382lib
```

  - ◆ 変更後

```
# Use the following macro when you use C-libraries for M32C/80 series.
#NEWLIB = -l nc382_16.lib
```

1.3. 可変割り込みベクタテーブル

V.5.40 Release 00 より、ベクタ番号を指定して割り込み関数を宣言した場合、可変割り込みベクタテーブルを自動的に生成します。
V.5.20 Release 02 以前では、可変割り込みベクターテーブルを自動生成するためにコンパイルオプション "-fmake_vector_table(-fMVT) "、アセンブルオプション"-fMVT"、およびリンクオプション"-fMVT"を選択する必要がありましたが、これらオプションの選択は不要になりました。

なお、V.5.20 Release 02 以前で作成したプロジェクトを移行する場合は、これらのオプションは引き継がれません。このため、ビルド実行時にリンクエラー "Can't generate automatically the variable interrupt vector table." が発生します。
このエラーが発生した場合は、アセンブリ言語スタートアッププログラム sect308.inc を修正してください。

```
[sect308.inc: 428 行目付近]
;-----------------------------------------------------------------
; variable vector section
;-----------------------------------------------------------------
        .section vector,ROMDATA ; variable vector table
        .org VECTOR_ADR
.if     0  ◀──────────────────────────────  .if 0 を挿入にして.lword を無効にする。
.if     __MVT__ == 0
        .lword  dummy_int       ; BRK (software int 0)
        .lword  dummy_int       ;
        (略)
        .lword  dummy_int       ; software int 63
.endif                          ; __MVT__
.endif  ◀──────────────────────────────  .if 0 に対応する.endif を挿入する。
```

1.4. スペシャルページベクタテーブル

V.5.40 Release 00 より、スペシャルページ番号を指定してスペシャルページサブルーチン呼び出しを行う関数を宣言した場合、スペシャルページベクタテーブルを自動的に生成します。
V.5.20 Release 02 以前では、スペシャルベクターテーブルを自動生成するためにコンパイルオプション "-fmake_special_table(-fMST) "、アセンブルオプション"-fMST"、およびリンクオプション"-fMST" を選択する必要がありましたが、これらオプションの選択は不要になりました。

なお、V.5.20 Release 02 以前で作成したプロジェクトを移行する場合は、これらのオプションは引き継がれません。このため、ビルド実行時にリンクエラー "Can't generate automatically the special page vector table." が発生します。このエラーが発生した場合は、アセンブリ言語スタートアッププログラム sect308.inc を修正してください。

```
[sect308.inc: 500 行目付近]
;=================================================================
; fixed vector section
;-----------------------------------------------------------------
        .section        svector,ROMDATA         ; specialpage vector table
.if     0  ◀──────────────────────────────  .if 0 を挿入にして SPECIAL を無効にする。
.if     __MST__ == 0
;=================================================================
; special page defination
;-----------------------------------------------------------------
;       macro is defined in ncrt0.a30
;       Format: SPECIAL number
;
;-----------------------------------------------------------------
;       SPECIAL 255
;       SPECIAL 254
        (略)
;       SPECIAL 19
;       SPECIAL 18
;
.endif  ; __MST__
.endif  ◀──────────────────────────────  .if 0 に対応する.endif を挿入する。
```

2. C 言語スタートアッププログラムについて

C コンパイラパッケージ V.5.40 Release 00 から、C 言語スタートアッププログラムをサポートしました。C 言語スタートアッププログラムは、V.5.20 Release 02 以前のコンパイラでは使用することができません。
なお、C 言語スタートアッププログラムの代わりに、従来のアセンブリ言語スタートアッププログラムも使用できます。

2.1. C 言語スタートアッププログラムのファイル構成

C 言語スタートアップは次の 13 個の C 言語ファイルで構成しています。

(1) resetprg.c
マイコンの初期設定を行います。

(2) initsct.c
各セクションの初期化(ゼロクリア、初期値転送)を行います。

(3) heap.c
ヒープ領域を確保します。

(4) fvector.c
固定ベクタテーブルの定義を行います。

(5) intprg.c
可変ベクタ割り込みのエントリ関数を宣言します。

(6) firm.c / firm_ram.c
OnChipDedebugger 使用時の FoUSB/E8 の firm が使用するプログラム領域及びワークスペース領域をダミーとして確保します。

(7) cregdef.h
マイコン内部レジスタを宣言しています。
このファイルの内容は変更しないでください。

(8) heapdef.h
ヒープ領域を初期化します。

(9) initsct.h
各セクションを初期化する処理(アセンブラマクロ)を記述しています。
このファイルの内容は変更しないでください。

(10) resetprg.h
C 言語スタートアップで使用する各ヘッダファイルをインクルードしています。

(11) stackdef.h
スタックサイズを定義しています。

(12) vector.h
可変ベクタのアドレスを定義しています。

(13) typedef.h
データ型を typedef 宣言しています。

2.2. C 言語スタートアッププログラムの処理

2.2.1. resetprg.c

このファイルの内容は、ご使用のマイコンにより異なります。
このファイルは、C 言語スタートアップで必須のファイルです。

```
#include "resetprg.h"
////////////////////////////////////////
// declare sfr register
#pragma ADDRESS  protect  0AH
#pragma ADDRESS  pmode0   04H
#pragma ADDRESS  _SB__    0400H
_UBYTE protect,pmode0;
_UBYTE _SB__;

#pragma entry start
void start(void);
extern void initsct(void);
extern void _init(void);
void exit(int);

#pragma section program  interrupt                                    → (1)
#pragma inline set_cpu()
void set_cpu(void)                                                    → (2)
{
        _isp_    = &_istack_top;  // set interrupt stack pointer     → (3)
        protect = 0x02;           // change protect mode register    → (4)
        pmode0   = 0x00;          // set processor mode register     → (5)
        protect = 0x00;           // change protect mode register    → (6)
        _flg_    = 0x0080;                // set flag register       → (7)
        _sp_     = &_stack_top;   // set user stack pointer          → (8)
        _sb_     = (char _far *)0x400;    // 400H fixation (Do not change) → (9)
        _asm("   fset     b");
        _sb_     = (char _far *)0x400;
        _asm("   fclr     b");
        _intb_ = (char _far *)VECTOR_ADR; // set variable vector's address → (10)

}
void start(void)
{
        set_cpu();                        // initialize mcu          → (11)
        initsct();                        // initlalize each sections → (12)
#ifdef __HEAP__
        heap_init();             // initialize heap                  → (13)
#endif
#ifdef __STANDARD_IO__
        _init();                 // initialize standard I/O          → (14)
#endif
        _fb_ = 0;                         // initialize FB registe for debugger
        main();                  // call main routine                → (15)

        exit(0);                 // infinite loop
}
void exit(int rc)
{
        while(1);
}
```

(1) マイコンリセット後に実行されるスタート関数 start()は interrupt セクションに配置します。
(2) マイコン初期化関数 set_cpu()本体を宣言します。
(3) 割り込みスタックポインタを初期化します。
(4) プロテクトモードレジスタを"書き込み許可"に設定します。
(5) プロセッサモードレジスタを"シングルチップモード"に設定します。モードを変更する場合は、この式を変更する必要があります。
(6) プロテクトモードレジスタを"書き込み禁止"に設定します。
(7) U フラグを 1 に設定(スタックポインタをユーザスタックに設定)します。
(8) ユーザスタックポインタを初期化します
(9) SB レジスタを 0x400 番地に設定(RAM の先頭アドレスを設定)します。
(10) 可変ベクタアドレスを INTB レジスタに設定します。
(11) マイコン初期化関数を呼び出します。
(12) 各セクションの初期化(ゼロクリア、初期値転送)を行います。
(13) ヒープ領域の初期化を行います。メモリ管理関数を使用する場合は、本関数の呼び出しを有効にする必要があります。
(14) 標準入出力関数の初期化を行います。標準入出力関数を使用する場合は、この関数の呼び出しを有効にする必要があります。
(15) main 関数を呼び出します。

2.2.2. resetprg.h

C 言語スタートアップで使用する各ヘッダファイルをインクルードします。
このファイルは、C 言語スタートアップで必須のファイルです。

2.2.3. initsct.c

このファイルの内容は、ご使用のマイコンにより異なります。
このファイルは、C 言語スタートアップで必須のファイルです。

```
#include "initsct.h"
void initsct(void);

void initsct(void)
{
        sclear("bss_SE","data,align");                          → (1)
        sclear("bss_SO","data,noalign");
        sclear("bss_NE","data,align");
        sclear("bss_NO","data,noalign");
        sclear("bss_FE","data,align");                          → (2)
        sclear("bss_FO","data,noalign");

        /* clear bss for NSD */
        sclear("bss_MON1_E","data,align");
        sclear("bss_MON2_E","data,align");
        sclear("bss_MON3_E","data,align");
        sclear("bss_MON4_E","data,align");
        sclear("bss_MON1_O","data,noalign");
        sclear("bss_MON2_O","data,noalign");
        sclear("bss_MON3_O","data,noalign");
        sclear("bss_MON4_O","data,noalign");

        // when add new sections
        // bss_clear("new section's name");
```

```
        scopy("data_SE","data,align");                          → (3)
        scopy("data_SO","data,noalign");
        scopy("data_NE","data,align");
        scopy("data_NO","data,noalign");

        /* copy data section for NSD */
        scopy("data_MON1_E","data,align");
        scopy("data_MON2_E","data,align");
        scopy("data_MON3_E","data,align");
        scopy("data_MON4_E","data,align");
        scopy("data_MON1_O","data,noalign");
        scopy("data_MON2_O","data,noalign");
        scopy("data_MON3_O","data,noalign");
        scopy("data_MON4_O","data,noalign");
        scopy("data_FE","data,align");                          → (4)
        scopy("data_FO","data,noalign");
}
```

(1) sclear：near 領域の bss セクションをゼロクリアします。
#pragma 拡張機能 #pragma SECTION を用いて bss セクションの名称を変更した場合は、変更後のセクションを追加する必要があります。

```
        sclear( “セクション名_NE” , “data.align” );
        sclear( “セクション名_NO” , “data,noalign” );
```

例えば、#pragma section bss bss2 で bss2 セクションを追加した場合は、次の行を initsct.c ファイルに追記します。

```
        sclear( “bss2_NE” , “data.align” );
        sclear( “bss2_NO” , “data,noalign” );
```

(2) sclear_f：far 領域の bss セクションをゼロクリアします。
far 修飾子を用いて初期値の無い外部変数を宣言した場合は、このマクロ関数を有効にする必要があります。
(3) scopy: near 領域の data セクションに対して初期値を転送します。
#pragma 拡張機能 #pragma SECTION を用いて data セクション名の名称を変更した場合は、変更後のセクションを追加する必要があります。

```
        sclear( “セクション名_NE” , “data.align” );
        sclear( “セクション名_NO” , “data,noalign” );
```

例えば、#pragma section data data2 で data2 セクションを追加した場合は、次の行を initsct.c ファイルに追記します。

```
        sclear( “data2_NE” , “data.align” );
        sclear( “data2_NO” , “data,noalign” );
```

(4) scopy_f：far 領域の data セクションに初期値を転送します。
far 修飾子を用いて初期値の無い外部変数を宣言した場合は、このマクロ関数を有効にする必要があります。

2.2.4. initsct.h

このファイルは、C 言語スタートアップで必須のファイルです。
ファイルの内容は変更しないでください。

2.2.5. heap.c

このファイルは、malloc 関数などのメモリ管理関数を使用する場合に必要です。

```
#include "typedefine.h"
#include "heapdef.h"
#pragma SECTION  bss     heap                                   → (1)

_UBYTE heap_area[__HEAPSIZE__];                                 → (2)
```

(1) heap 領域を heap_NE セクションに配置します。
ヒープサイズを奇数バイトにした場合は、heap_NO セクションに変更します。
(2) ヒープ領域を__HEAPSIZE__で定義されたサイズ分確保します。

2.2.6. heapdef.h

このファイルは、malloc 関数などのメモリ管理関数を使用する場合に必要です。

```
extern  _UBYTE _far * _mnext;
extern  _UDWORD _msize;
/////////////////////////////////////////
// It's size of heap
// When you want to change size of heap,
// please change this line.
// When you change this line,
// you must modify the value using hex character.

#ifndef __HEAPSIZE__
#define __HEAPSIZE__     0x300
#endif
extern _UBYTE heap_area[__HEAPSIZE__];

#pragma inline heap_init()
void heap_init(void)
{
        _mnext = &heap_area[0];                                    → (1)
        _msize = __HEAPSIZE__;                                     → (2)
}
```

(1) ヒープ管理領域を初期化します。
(2) ヒープサイズを初期化します。

2.2.7. fvector.c

このファイルは、C 言語スタートアップで必須のファイルです。

```
#include "vector.h"
#pragma sectaddress      fvector,ROMDATA Fvectaddr                 → (1)

////////////////////////////////////////////////////////////////////////

#pragma interrupt/v _dummy_int    //udi                            → (2)
#pragma interrupt/v _dummy_int    //over_flow
#pragma interrupt/v _dummy_int    //brki
#pragma interrupt/v _dummy_int    //address_match
#pragma interrupt/v _dummy_int    //single_step
#pragma interrupt/v _dummy_int    //wdt
#pragma interrupt/v _dummy_int    //dbc
#pragma interrupt/v _dummy_int    //nmi
#pragma interrupt/v start                                          → (3)


#pragma interrupt _dummy_int()
void _dummy_int(void){}
```

(1) 固定ベクタテーブルのセクションとアドレスを設定します。
この#pragma 拡張機能は C 言語スタートアップ専用です。
(2) リセット以外の固定ベクタをダミー関数(_dummy_int)で埋めます。
この#pragma 拡張機能は C 言語スタートアップ専用です。
(3) リセット関数を定義します。
マイコンリセット時に実行する関数を固定ベクタに登録します。

2.2.8. intprg.c

このファイルの内容は、ご使用のマイコンにより異なります。

```
// BRK (software int 0)
#pragma interrupt        _brk(vect=0)
void _brk(void){}

// vector 1 reserved
// vector 2 reserved
// vector 3 reserved
// vector 4 reserved
// vector 5 reserved
// vector 6 reserved

// A/D1 (software int 7)
#pragma interrupt        _ad1(vect=7)
void _ad1(void){}

// DMA0 (software int 8)
#pragma interrupt        _dma0(vect=8)                                  → (1)
void _dma0(void){}

// DMA1 (software int 9)
#pragma interrupt        _dma1(vect=9)
void _dma1(void){}

// DMA2 (software int 10)
#pragma interrupt        _dma2(vect=10)
void _dma2(void){}

// DMA3 (software int 11)
#pragma interrupt        _dma3(vect=11)
void _dma3(void){}

// TIMER A0 (software int 12)
#pragma interrupt        _timer_a0(vect=12)
void _timer_a0(void){}

// TIMER A1 (software int 13)
#pragma interrupt        _timer_a1(vect=13)
void _timer_a1(void){}

          :
        (省略)
          :
```

(1) 可変ベクタ割り込み関数を宣言します。
各可変ベクタ割り込み関数に対応した関数を宣言します。ここで宣言された関数は、リンク時に可変ベクタテーブルに反映されます。

2.2.9. firm.c / firm_ram.c

このファイルは、OnChipDebugger を使用する場合に使用します。

```
#include "typedefine.h"
#pragma section bss FirmRam                                    → (1)
_UBYTE _workram[0x4];     // for Firmware's workram            → (2)
```

(1) E8 エミュレータのファームウェアが使用する work ram 領域を FirmRam_NE セクションに確保します。
[5] Work ram 領域を__WORK_RAM__で定義されたサイズ分確保します。

2.2.10. cregdef.h

このファイルは、C 言語スタートアップで必須のファイルです。
ファイルの内容は変更しないでください。

2.2.11. stackdef.h

このファイルは、C 言語スタートアップで必須のファイルです。

```
#ifndef __STACKSIZE__
#pragma STACKSIZE        0x300                                 → (1)
#else
#pragma STACKSIZE        __STACKSIZE__                         → (2)
#endif
#ifndef ISTACKSIZE
#pragma ISTACKSIZE       0x300                                 → (3)
#else
#pragma ISTACKSIZE       __ISTACKSIZE__                        → (4)
#endif
extern _UINT _stack_top,_istack_top;
```

(1) リンク時にスタックサイズを指定していない場合に使用するユーザスタックサイズです。この#pragma 拡張機能により、ユーザスタックのセクション設定とスタックの領域を確保します。
(2) リンク時にスタックサイズを指定している場合に使用するユーザスタックサイズです。この#pragma 拡張機能により、ユーザスタックのセクション設定とスタックの領域を確保します。
(3) この#pragma 拡張機能は C 言語スタートアップ専用です。
(4) リンク時にスタックサイズを指定していない場合に使用する割り込みスタックサイズです。この#pragma 拡張機能により、割り込みスタックのセクション設定とスタックの領域を確保します。
(5) リンク時にスタックサイズを指定している場合に使用する割り込みスタックサイズです。この#pragma 拡張機能により、割り込みスタックのセクション設定とスタックの領域を確保します。
この#pragma 拡張機能は C 言語スタートアップ専用です。

2.2.12. vector.h

このファイルは、C 言語スタートアップで必須のファイルです。

```
#define Fvectaddr      0xffffdc                                → (1)
#ifndef VECTOR_ADR
#define VECTOR_ADR       0x0fffd00                             → (2)
#endif
```

(1) 固定ベクタテーブルの先頭アドレスを設定します。
(2) 可変ベクタテーブルの先頭アドレスを設定します。
可変ベクタテーブルの先頭アドレスを変更する場合は、resetprg.c ファイルの INTB レジスタのアドレス設定も

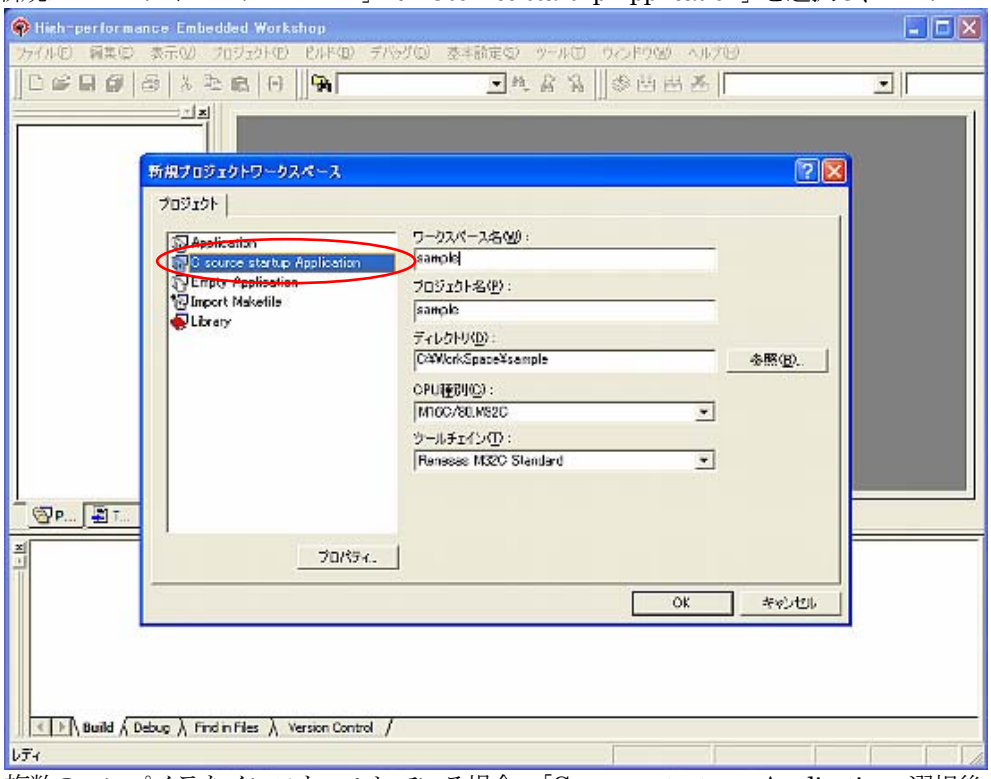
変更してください。

2.2.13. typedefine.h

このファイルは、C 言語スタートアップで必須のファイルです。
ファイルの内容は変更しないでください。

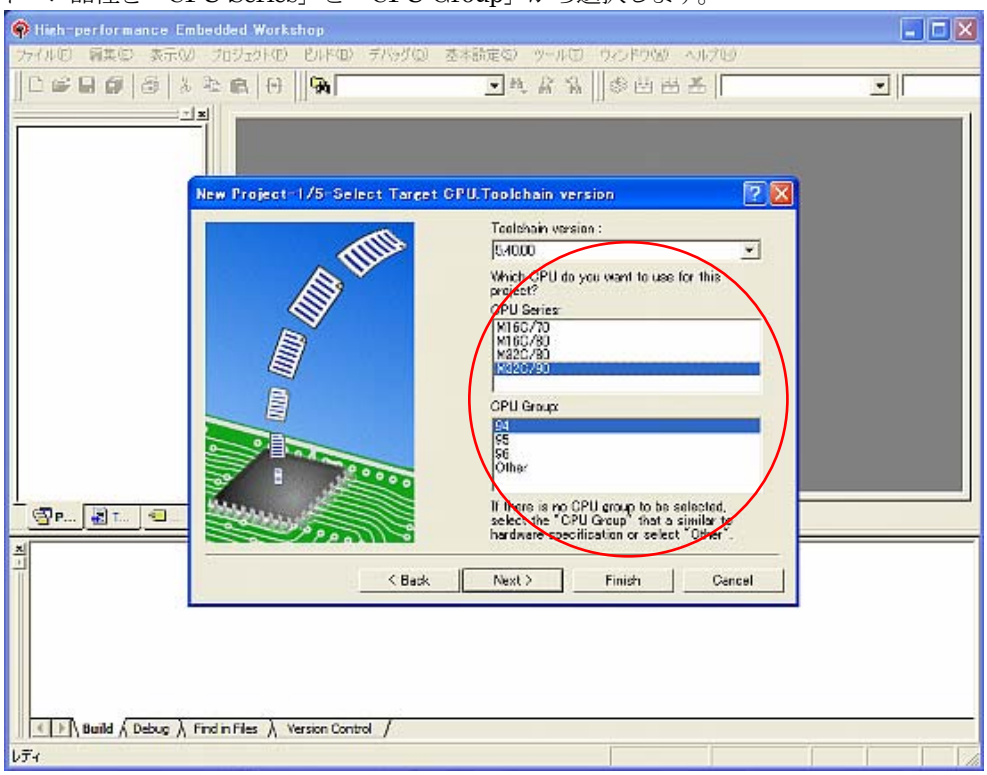
2.3. High-performance Embedded Workshop で C 言語スタートアッププログラムを使用する場合

(1) 「新規プロジェクトワークスペース」で「Csource startup Application」を選択し、ワークスペースを作成します。



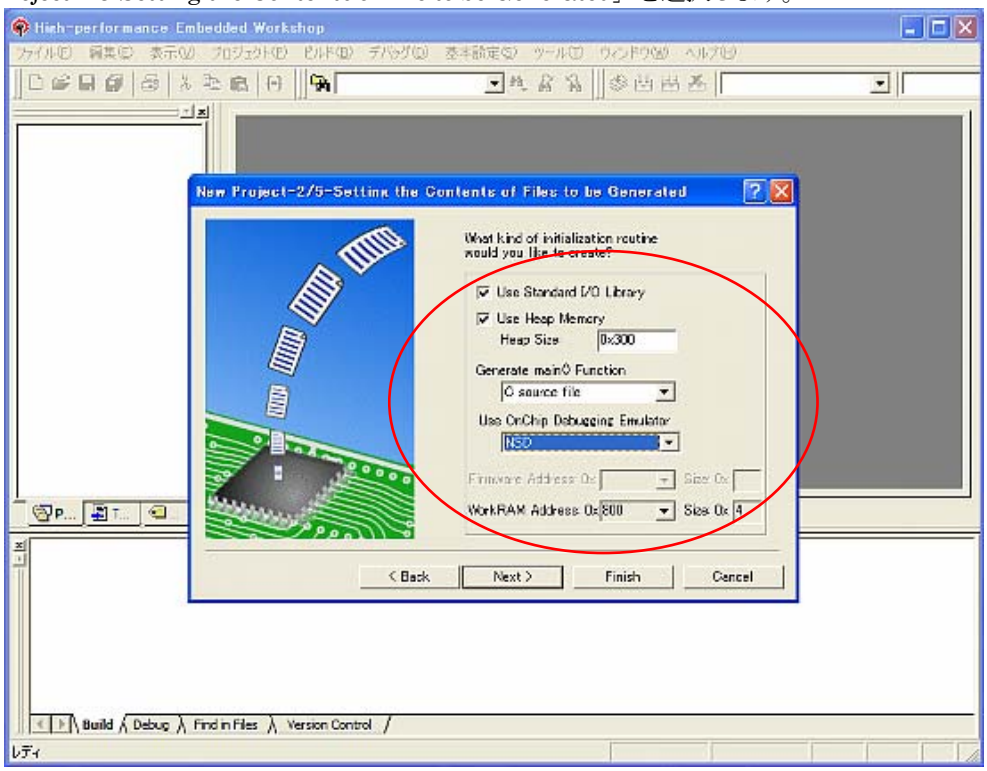
複数のコンパイラをインストールしている場合、「C source startup Application」選択後、「CPU 種別」で他マイコンを選択した場合、「C source startup Application」へのフォーカスが「Application」に移動して C ソーススタートアップの選択が無効になります。
この場合は、再度「C source startup Application」を選択してください。

(2) マイコン品種を「CPU Series」と「CPU Group」から選択します。



この選択により、対応する sfr ヘッダファイルがワークスペースへコピーされます。また、可変ベクタテーブル(intprg.c)が登録されます。

(3) 標準関数ライブラリとメモリ管理関数ライブラリを使用する場合は、「Use Standard I/O Library(UART1)」および「Use Heap Memory」を選択します。「OnChip Debugging Emulator」を使用する場合は、「New Project-2/5-Setting the Contents of File to be Generated」を選択します。



[1] 標準関数ライブラリを使用する場合

(1) チェックすることにより、resetprg.c 中の_init()呼び出しが有効になります。

(2) また、ファイル device.c と init.c がプロジェクトに登録されます。

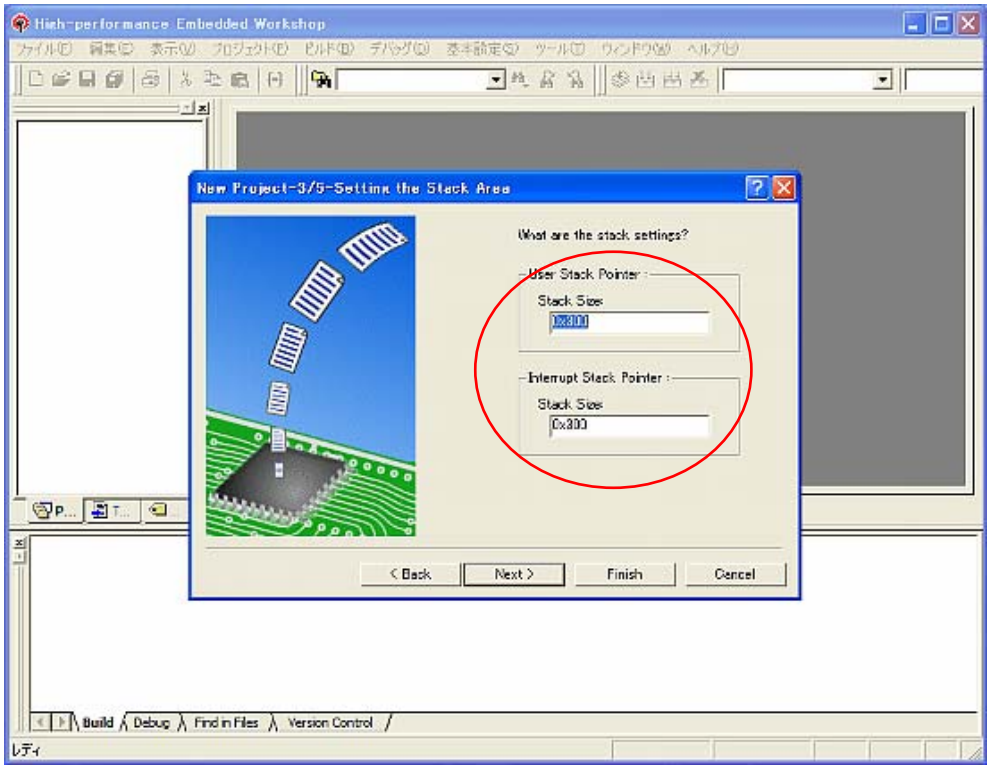
(3) メモリ管理関数を使用する場合に、チェックします。

(4) チェックすることにより、resetprg.c 中の heap_init()呼び出しが有効になります。
(5) また、heapdef.h ,heap.c がプロジェクトに登録されます。

[2] OnChip Debugging Emulator を使用する場合

選択可能なデバッガは、「FoUSB」と「NSD」です。ただし、選択するマイコン品種により、いずれか一方、もしくは両方選択できない場合があります。
この選択により、firm.c が登録されます。
標準入出力関数ライブラリを選択した状態で「OnChip Debugging Emulator」を選択した場合、(UART1)の表示が(UART0)に変わります。これは、標準入出力関数および OnChip Debugging Emulator が共に UART1 を使用するため、標準入出力側を UART0 へ変更することを意味しています。

(4) スタックサイズを設定します。

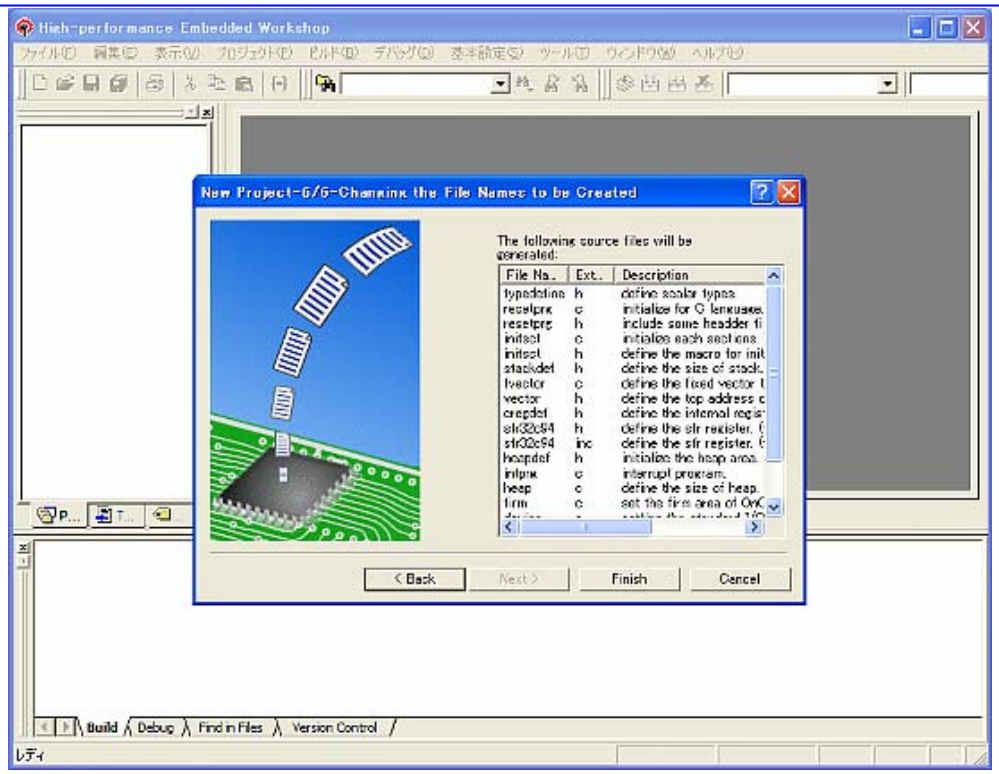


[1] ユーザスタックサイズの設定
stackdef.h が登録されます
[2] ユーザスタックサイズの設定
stackdef.h が登録されます

プロジェクト作成後、スタックサイズ及び HEAP サイズを変更する場合は、コンパイルオプションで以下の項目を変更してください。

```
#define Fvectaddr      0xffffdc                                    → (1)
#ifndef VECTOR_ADR
#define VECTOR_ADR       0x0fffd00                                 → (2)
#endif
```

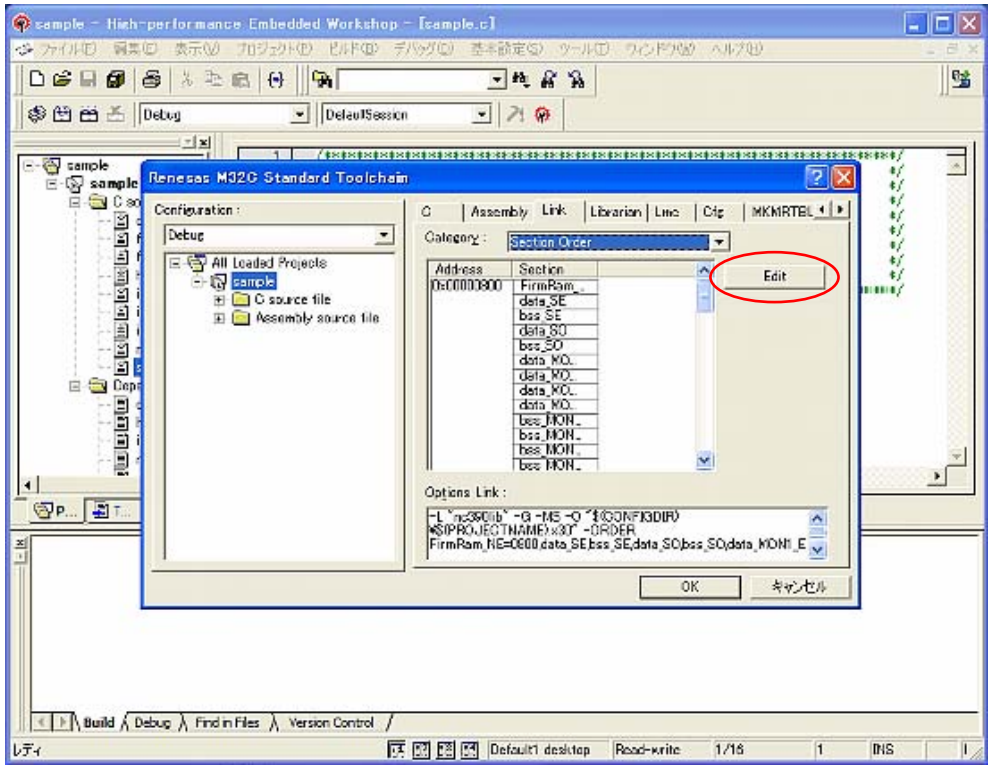
(5) 登録ファイルを確認します。



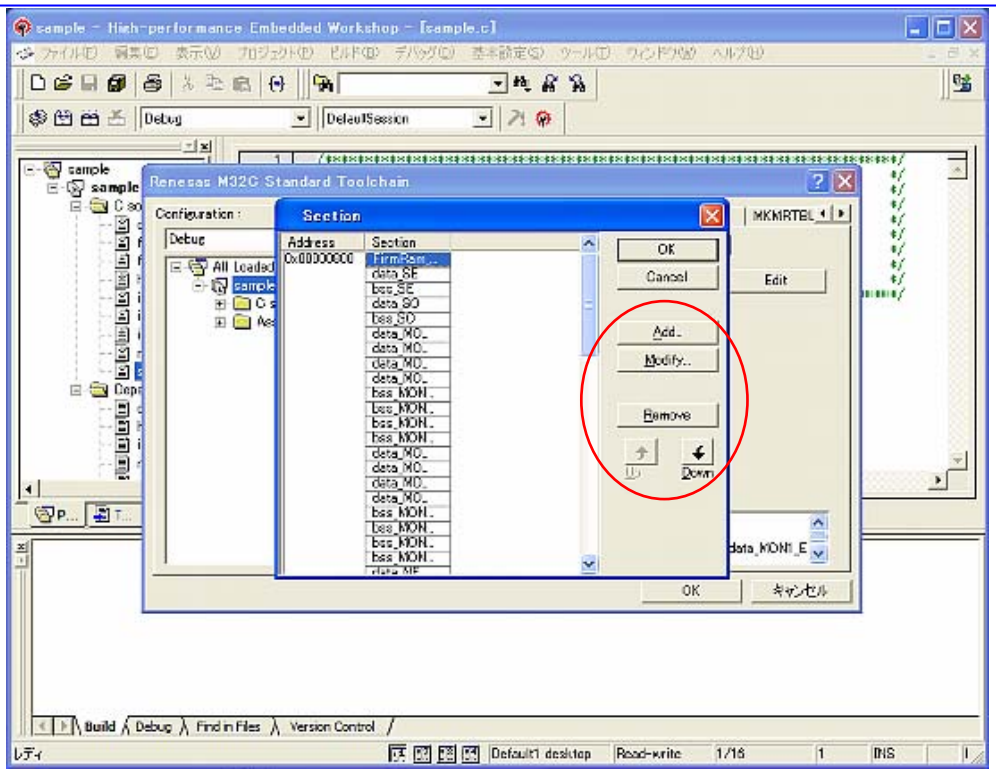
ここで、登録されるファイルを一覧で確認できます。
ただし、マイコン品種ごとに登録される sfr ヘッダ(C 言語用ヘッダ、アセンブラ用ヘッダ)は、ワークスペースへコピーのみです。

(6) セクションオーダー

「Renesas M32C Standard Toolchain」→「Link」の「Category」で各セクションの配置およびアドレスを確認することができます。

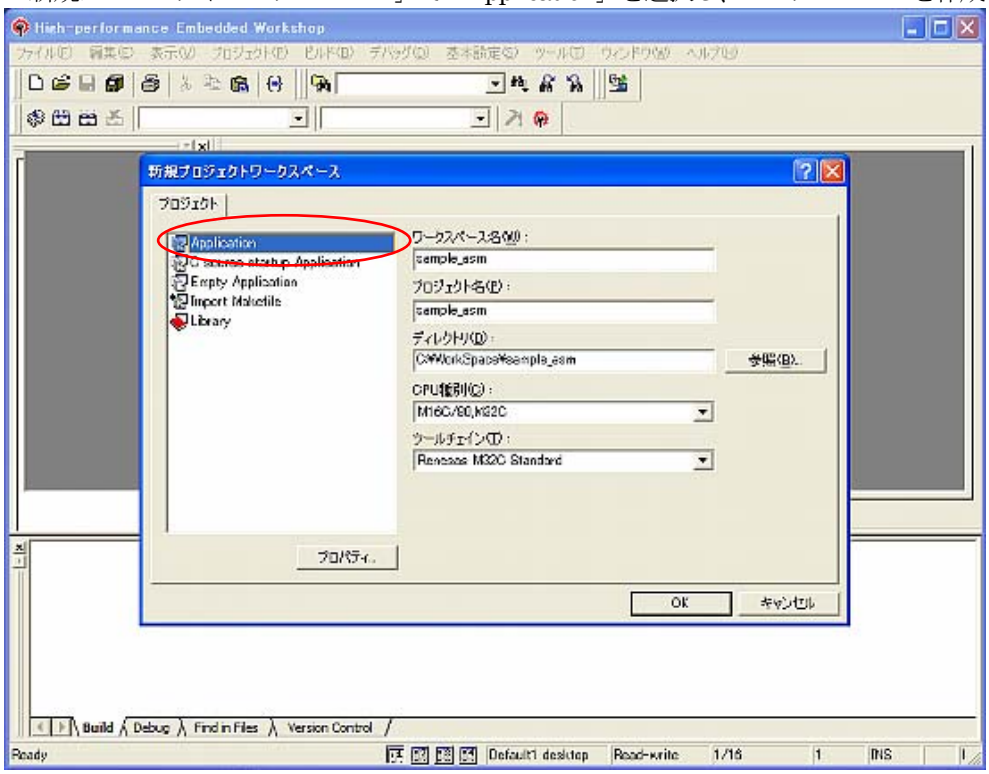


#pragma SECTION により新規にセクションを追加した等の場合は、「Edit」を選択して、「Section Window」をオープンし、セクションの配置等を変更します。



2.4. High-performance Embedded Workshop でアセンブリ言語スタートアッププログラムを使用する場合

「新規プロジェクトワークスペース」で「Application」を選択し、ワークスペースを作成します。



3. TM→High-performance Embedded Workshop V.4 移行の手引き

本資料は、TM V.2.xx、V.3.xx で作成したプロジェクトを High-performance Embedded Workshop V.4 環境へ移行するための情報を説明します。
なお、本移行の手引きについては、今後 High-performance Embedded Workshop のバージョンアップ等に伴い、記載内容が異なる事が考えられます。
最新の情報につきましては、ルネサス開発環境 Home Page 内の FAQ サイトをご覧ください。

3.1. 概要

TM V.2.xx、V.3.xx で作成したプロジェクトを High-performance Embedded Workshop V.4 環境へ移行するには、High-performance Embedded Workshop の「Import Makefile」機能を使用します。「Import Makefile」は、指定された makefile に書かれているソースファイルやオプション情報からプロジェクトを作成する機能です。
TM のプロジェクトファイルは、GNU make から実行可能である makefile フォーマットで作成されています。「Import Makefile」にて、TM のプロジェクトファイルを makefile として指定することで、TM のプロジェクトを High-performance Embedded Workshop のプロジェクトへ変換することができます。「Import Makefile」では、TM のプロジェクトファイル以外に hmake、nmake、gmake 用の makefile フォーマットのファイルを High-performance Embedded Workshop のプロジェクトに変換することができます。

3.2. 変換手順

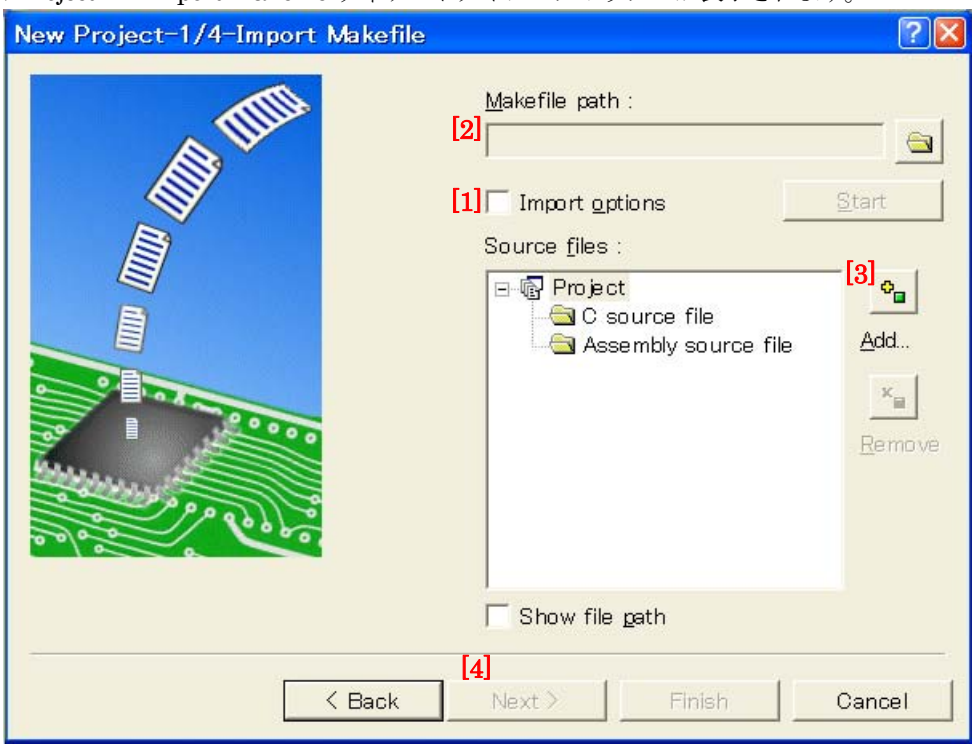
以下に、TM のプロジェクトを High-performance Embedded Workshop のプロジェクトへ移行する手順を示します。
(1) メニュー［ファイル］→［新規ワークスペース］をクリックします。
(2) 新規プロジェクトワークスペースダイアログボックスが表示されます。

[1] CPU 種別を選択します。TM のプロジェクトで使用していた CPU 種別を選択してください。
[2] ツールチェインを選択します。ツールチェイン名とクロスツール名の対応は以下の通りです。TM のプロジェクトで使用していたツールチェイン（クロスツール）を選択してください。

ツールチェイン名	クロスツール名
Renesas M16C Standard	NC30WA
Renesas R8C Standard	NC8C
Renesas M32C Standard	NC308WA
Renesas M32R Standard	CC32R

[3] プロジェクトタイプから Import Makefile を選択します。
[4] ディレクトリを指定します。
[5] ワークスペース名を指定します。ワークスペース名を指定すると自動的に（ワークスペースと同名の）プロジェクト名が指定されます。
[6] OK ボタンをクリックします。

(3) New Project-1/4-Import Makefile ウィザードダイアログボックスが表示されます。



[1] 「Import options」をチェックします。この項目をチェックすると、ビルド（コンパイラ、アセンブラ etc）オプション情報が High-performance Embedded Workshop プロジェクトへ移行されます。チェックをはずすと、オプション情報は無視されます（High-performance Embedded Workshop プロジェクトへ移行されません）。
[2] Makefile path に TM のプロジェクトファイル（拡張子が tmk）を指定します。Makefile path にファイルが指定されるとすぐに指定ファイルの解析作業が行われます。解析が終了すると解析したソースファイルが Source files ツリーに表示されます。「Start」ボタンはクリックすると、再度、指定ファイルの解析作業が行われます。
[3] 解析結果（Source files ツリー）に誤りがある場合は、Add…、Remove ボタンから Source files ツリーを編集してください。
[4] Next ボタンをクリックします。

(4) 以降はダイアログボックスの指示に従って作業を進めてください。

3.3. 注意事項

3.3.1. 移行できる情報、できない情報

TM のプロジェクトを High-performance Embedded Workshop 環境へ移行する場合、TM のプロジェクトの全構成を移行できるわけではありません。移行できる情報は、以下の通りです。

- アセンブラソースファイルパス
- C 言語ソースファイルパス
- アセンブラオプション
- C コンパイラオプション
- リンカオプション（リンク順序を除く）

その他の情報は High-performance Embedded Workshop 環境へ移行することができません。移行できない情報は、「Import Makefile」の処理終了後、これ以降に示す注意事項の通りに High-performance Embedded Workshop プロジェクトを編集してください。

3.3.2. クロスツール

「Import Makefile」では、クロスツールのバージョンを High-performance Embedded Workshop プロジェクトへ移行することができません。よって、TM のプロジェクトで使用していたクロスツールのバージョンにかかわらず、High-performance Embedded Workshop プロジェクト移行後に使用可能なクロスツールのバージョンは以下のものになります。

NC30WA	:	V.5.20 Release1 以降
NC8C	:	V.5.30 Release1
NC308WA	:	V.5.20 Release1 以降
CC32R	:	V.4.20 Release1 以降

3.3.3. High-performance Embedded Workshop のバージョン

TM のプロジェクトを High-performance Embedded Workshop 環境へ移行する場合、移行先の High-performance Embedded Workshop のバージョンにより移行できる情報が異なります。High-performance Embedded Workshop のバージョンによる移行可能情報は以下の通りです。

		High-performance Embedded Workshop	
		V.3	V.4
NC30WA	V.5.20 Release1	△	△
	V.5.30 Release1	△	△
	V.5.30 Release 02	---	○
	V.5.40 Release00	---	○
NC8C	V.5.30 Release1	△	△
NC308WA	V.5.20 Release1	△	△
	V.5.20 Release 02	---	▲
	V.5.40 Release00	---	○
CC32R	V.4.20 Release1	△	△
	V.4.20 Release1A	△	△
	V.4.30 Release 00	▲	▲
	V.5.00 Release 00	---	○

○ : アセンブラソースファイル、C 言語ソースファイルおよびアセンブラ、C コンパイラ、リンカのオプション移行可能
▲ : アセンブラソースファイル、C 言語ソースファイルおよびアセンブラ、C コンパイラのオプション移行可能
△ : アセンブラソースファイル、C 言語ソースファイルのみ移行可能

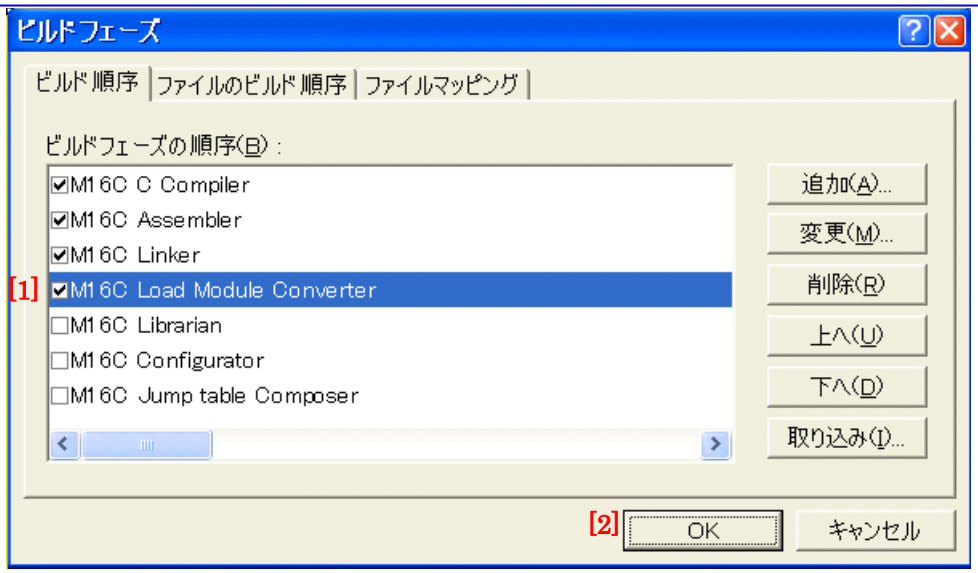
3.3.4. 作成されるプロジェクトワークスペース

TM のプロジェクトを HEW 環境へ移行して作成したプロジェクトワークスペースは、makefile の内容そのままであるため、HEW 上で新規に作成したプロジェクトワークスペースとはコンフィグレーション（オブジェクト出力ディレクトリ）が異なります。
コンパイラやアセンブラ、リンカの出力ディレクトリ・ファイル名の指定を以下に変更することで、コンフィグレーションを有効化できます。

出力ディレクトリ（コンパイラ・アセンブラ）	:	$(CONFIGDIR)
出力ファイル名（リンカ）	:	$(CONFIGDIR)¥$(PROJECTNAME).x30

3.3.5. ロードモジュールコンバータ

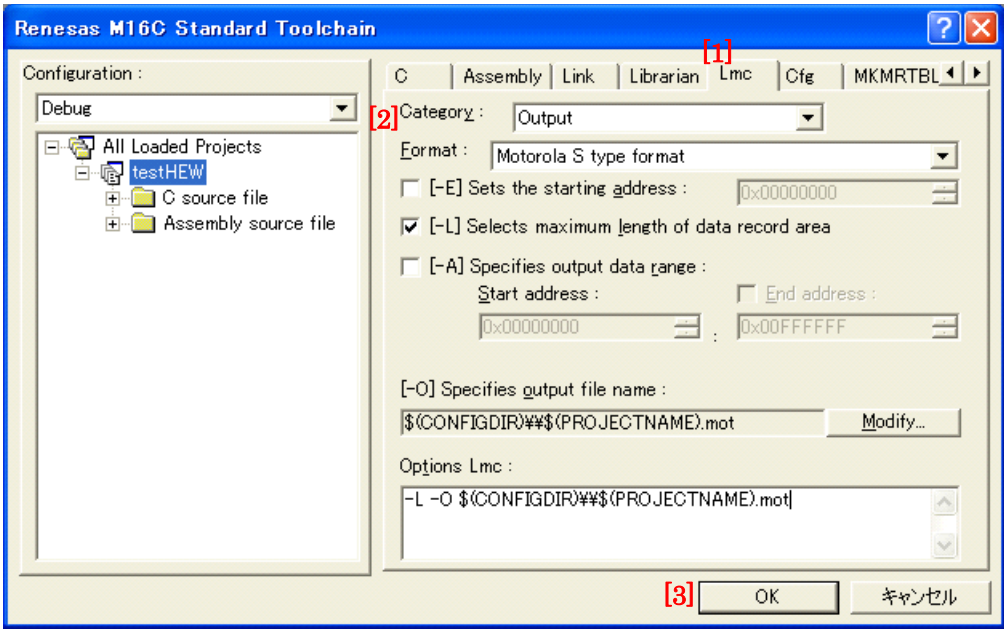
「Import Makefile」では、ロードモジュールコンバータの情報（コマンド実行、オプション情報）を High-performance Embedded Workshop プロジェクトへ移行することができません。TM のプロジェクトでロードモジュールコンバータを使用していた場合は、「Import Makefile」の処理終了後、以下の手順でロードモジュールコンバータの設定を行ってください。
(1) メニュー［ビルド］→［ビルドフェーズ］をクリックします。
(2) ビルドフェーズダイアログボックスが表示されます。



[1] Mxxx Load Module Converter をチェックします。
[2] OK ボタンをクリックします。

(3) メニュー［ビルド］→［Renesas Mxxx Standard Toolchain…］をクリックします。

(4) Renesas Mxxx Standard Toolchain ダイアログボックスが表示されます。



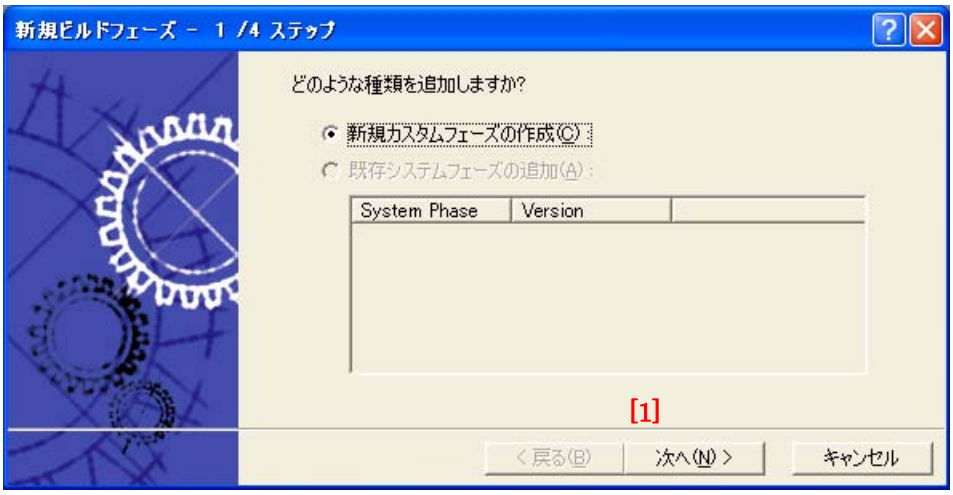
[1] Lmc タブをクリックします。
[2] Category を変更してオプションを指定します。
[3] OK ボタンをクリックします。

3.3.6. 外部ツール

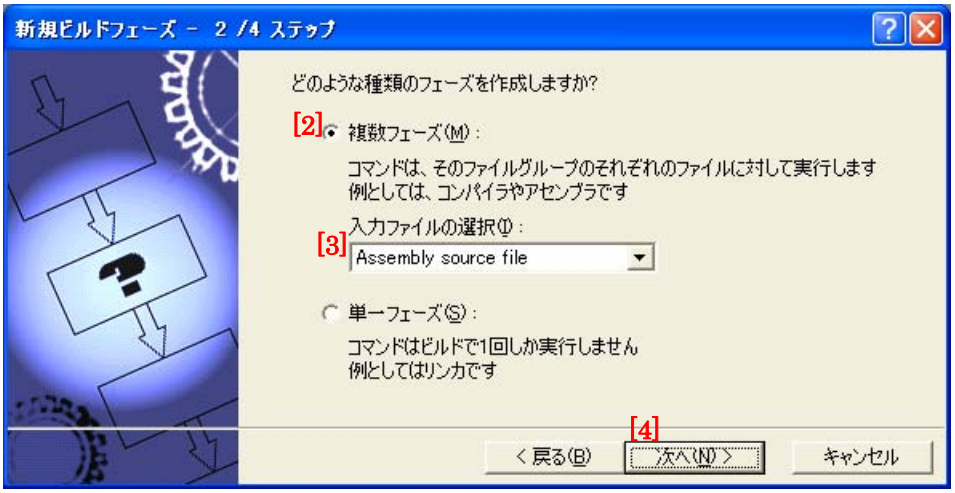
「Import Makefile」では、アセンブラ、C コンパイラ、リンカ以外のツールの情報（コマンド実行、オプション情報、依存関係）を High-performance Embedded Workshop プロジェクトへ移行することができません。TM のプロジェクトで、アセンブラ、C コンパイラ、リンカおよびロードモジュールコンバータ以外のツールを使用していた場合は、High-performance Embedded Workshop のカスタムビルドフェーズを作成していただく必要があります。カスタムビルドフェーズは、標準のビルド実行（アセンブラ、C コンパイラ、リンカ）の前後または途中に外部ツールを実行するための独自のビルドフェーズです。
カスタムビルドフェーズ作成手順についての詳細は、High-performance Embedded Workshop4 ユーザーズマニュアル「3.2 カスタムビルドフェーズを作成する」をご覧ください。
ここでは、例としてクロスツールにバンドルされている xrf30 を登録する方法を示します。

(1) メニュー［ビルド］→［ビルドフェーズ］をクリックします。
(2) ビルドフェーズダイアログボックスが表示されます。追加ボタンをクリックします。

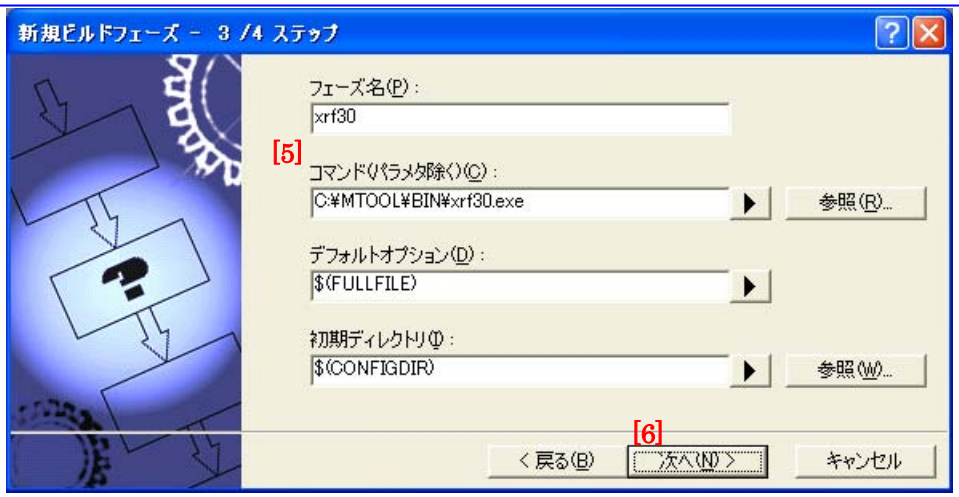
(3) 新規ビルドフェーズダイアログボックスが表示されます。ウィザードに従ってツールを登録します。



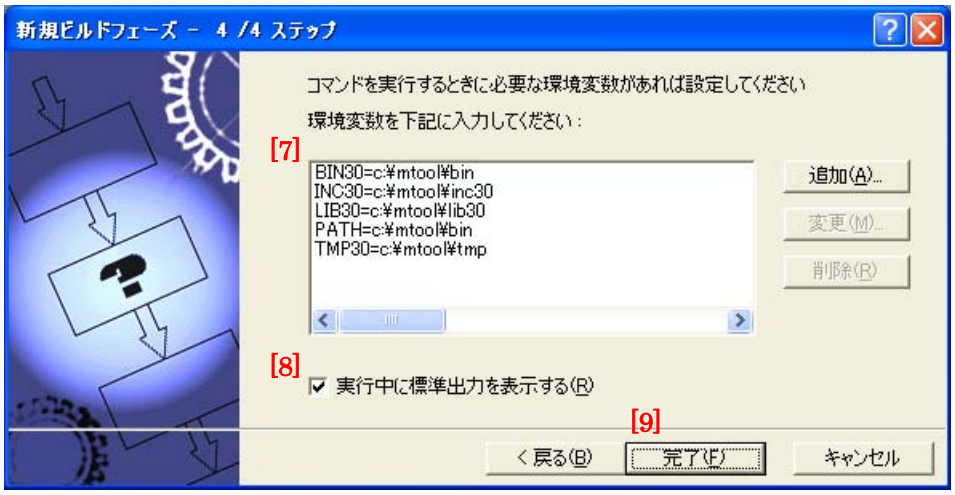
[1] ［1/4 ステップ］次へボタンをクリックします。



[2] ［2/4 ステップ］複数フェーズを選択します。
[3] ［2/4 ステップ］入力ファイルの選択から Assembly source file を選択します。
[4] ［2/4 ステップ］次へボタンをクリックします。



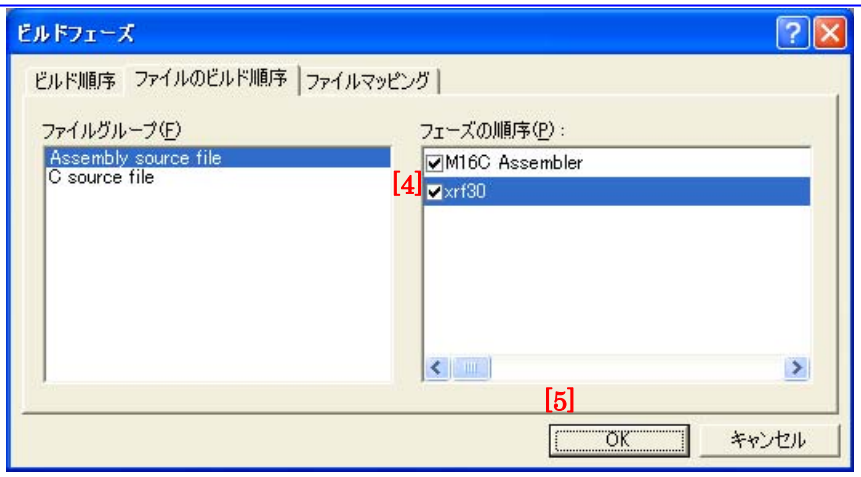
[5] ［3/4 ステップ］フェーズ名に xrf30 を、コマンドに xrf30 のフルパスを指定します。
[6] ［3/4 ステップ］次へボタンをクリックします。



[7] ［4/4 ステップ］環境変数を指定します。
[8] ［4/4 ステップ］実行中に標準出力を表示するをチェックします。
[9] ［4/4 ステップ］完了ボタンをクリックします。

(4) ビルドフェーズダイアログボックスに戻ります。
登録したビルドフェーズ（xrf30）がリストの最後に追加されます。

[1] xrf30 を選択します。
[2] 上へボタンクリックします。Assembler の下まで xrf30 を移動します。
[3] ファイルのビルド順序タブをクリックします。



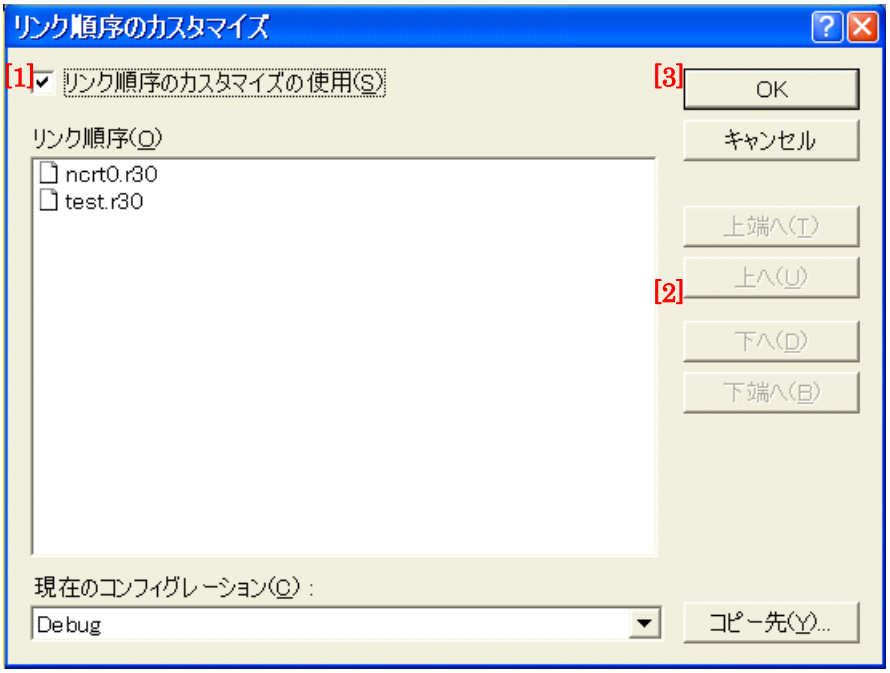
[4] xrf30 をチェックします。
[5] OK ボタンをクリックします。

(5) メニュー［ビルド］→［xrf30］をクリックします。
(6) xrf30 Options ダイアログボックスが表示されますので、必要に応じてオプションなどを設定してください。この設定を行うと、ビルド時のアセンブラ実行後（リンカの実行前）に xrf30 がすべてのアセンブラソースファイルに対し実行されるようになります。

3.3.7. リンク順序

「Import Makefile」では、リンク順序の情報を High-performance Embedded Workshop プロジェクトへ移行することができません。リンク順序は、ファイル名のアルファベット順になります。リンク順序を変更する場合は、以下の手順で設定してください。

(1) メニュー［ビルド］→［リンク順の指定］をクリックします。
(2) リンク順序のカスタマイズダイアログボックスが表示されます。



[1] リンク順序のカスタマイズの使用をチェックします。
[2] リンク順序リストからファイルを選択して、上へまたは下へボタンをクリックします。
[3] OK ボタンをクリックします。

3.3.8. スタートアッププログラムの先頭リンク

「Import Makefile」では、リンク順序の情報をHigh-performance Embedded Workshopプロジェクトへ移行することができません。リンク順序は、ファイル名のアルファベット順になります。このためスタートアップが先頭にリンクされないことがあります。スタートアッププログラムを先頭にリンクするには、「3.3.7. リンク順序」の項の設定を行ってください。

4. V.5.41 Release 00 で追加した機能

4.1. Call Walker 対応

V.5.41 Release 00 より、STK ビューワに加え、Call Walker を使用することにより、アプリケーションプログラム中で使用するスタックサイズを算出することができます。
Call Walker をご使用の際は、Call Walker の「Help」メニューの「Help Topics」をクリックすることにより表示される「Call Walker Help」をご参照ください。

4.1.1. Call Walker の起動方法

- 起動
  次の 2 種類の方法により、Call Walker を起動することができます。
  [1] High-performance Embedded Workshop から Call Walker を起動する場合
  High-performance Embedded Workshop の「ツール」メニュー内の「Renesas Call Walker」をクリックします。
  [2] Windows スタートメニューから Call Walker を起動する場合
  Windowsスタートメニューの「すべてのプログラム」[1] の中にある「Renesas」メニューの「M32C Series C Compiler V.5.42 Release 00」メニュー内の「Call Walker」をクリックします。
- 終了
  Call Walker の「File」メニューの「Exit」をクリックします。

4.1.2. Call Walker の入力ファイルの作成

Call Walker の入力ファイルは、.sni ファイル作成ツール gensni を使用して作成します。
Call Walker の入力ファイルの作成は、アブソリュートモジュールファイル(x30)のビルド方法により異なります。
(1) High-performance Embedded Workshop 上でビルドする場合
ビルド時に gensni が自動的に実行されます。
(2) コマンドプロンプト(または DOS プロンプト)上でコンパイル、アセンブル、リンクする場合
gensni をコマンドプロンプト(または DOS プロンプト)上で実行してください。

【gensni 操作例】

```
C:¥> gensni  -o  sample.sni  sample.x30
```

4.1.3. Call Walker の入力ファイルの選択方法

Call Walker の入力ファイルは、Call Walker の「File」メニューの「Import Stack File」をクリック後に表示される「Stack File」ウィンドウから選択してください。

4.2. High-performance Embedded Workshop の Map Section Information ウィンドウ対応

V.5.41 Release 00 より、MAP ビューワに加え、High-performance Embedded Workshop の Map Section Information ウィンドウを使用することにより、アブソリュートモジュールファイルのマップ情報を表示することができます。
Map Section Information ウィンドウをご使用の際は、High-performance Embedded Workshop V.4.01 ユーザーズマニュアルの「13.マップ」をご参照ください。

[1] Windows XP の場合。