===== 必ずお読みください =====

M16C シリーズ,R8C ファミリ用 C/C++コンパイラパッケージ
NC30 V.6.00 Release 00
リリースノート
（第 1 版）

誤記に関するお詫び：
本資料 P.22「5.1 アセンブラスタートアップを使用されている場合の設定」の【注意事項】に誤記があり、訂正いたしました。

株式会社ルネサスソリューションズ

2011 年 3 月 31 日

概要

本資料は M16C シリーズ,R8C ファミリ用 C/C++コンパイラパッケージ（M3T-NC30WA）V.6.00 のご使用にあたり、C/C++コンパイラパッケージの電子マニュアルの補足等について説明します。電子マニュアルの該当項目をご覧になる場合は、併せてこのリリースノートをご覧いただきますようお願い申し上げます。

1. C/C++コンパイラパッケージのインストール 3
2. 最新情報のご案内 3
3. 注意事項 3
3.1. リアルタイム OS　MR30 の対応バージョンについて 3
3.2. ファイル名に関する注意事項 3
3.3. ウィルスチェックプログラムに関する注意事項 3
3.4. 英語 PC 上でのモトローラ S フォーマットの比較方法について 4
3.5. 機種依存部に関する注意事項 4
3.5.1. 割り込み制御レジスタに関する注意事項 4
3.5.2. SFR 領域のアクセスに関する注意事項 5
3.5.3. 割り込み優先レベルに値を設定する場合の注意事項 5
3.5.4. オンチップデバッガ E8a を使用する場合、「E8」を選択して下さい。 5
3.6. コンパイラ、アセンブラ、最適化リンケージエディタおよびユーティリティに関する注意事項 6
3.6.1. 最適化リンケージエディタ(optlnk)のセクションの配置について 6
3.6.2. 同名のローカル変数が同じ関数内にある場合のデバッガ表示について 6
3.6.3. リンク時のエラーレベルを下げて生成した ABS ファイルの取扱いについて 6
3.6.4. C++言語で不完全列挙型を使用するオブジェクトを定義するとエラーとなる場合について 6
3.6.5. デバッグ指定時、template 関数に extern または static を指定するとエラーとなる場合について 6
3.6.6. キーワード mutable はサポートしていません。 6
3.6.7. 関数型キャスト「型(式)」形式で、型の near/far を先行して記述するとエラーとなる問題について 7
3.6.8. C++言語での#pragma ASM の使用について 7
3.6.9. C++言語でのアセンブラマクロ関数の仕様について 7
3.6.10. 関数内に#pragma ASM, asm()またはアセンブラマクロ関数呼び出しを使用して自関数を参照する場合の制限事項について 7
3.6.11. Ver.5.xx からの Ver.6.xx へ移行時の init.c,device.c の登録について 7
3.6.12. C++動的初期化に関する注意事項 8
3.6.13. コンパイラユーザーズマニュアルの標準ライブラリ関数 scanf の解説に誤りがあります。 8
3.6.14. printf 系関数の変換指定に対応する実引数の型が double 型の結果が正しくありません。 8
3.6.15. scanf 系関数の変換指定に対応する実引数の型が double 型へのポインタの場合、結果が正しくありません。 9
3.6.16. assert マクロの仕様について 10
3.6.17. コンパイラオプション-Wlarge_to_small(-WLTS)に関する注意事項 10
3.6.18. R8C ファミリ(ROM64KB 以上)において固定ベクタを避けてセクションを配置する方法 10

3.7. 制限事項、および既知の不具合 11
4. V.5.45 Release 01 からのバージョンアップ内容 13
4.1. C++で-ffar_pointer(-fFP)を使用する場合、関数内で配列型へのポインタが 16 ビットになります。 13
4.2. near/far のキャスト記法について 14
4.3. シンボル__SB__の.glb 宣言に関する注意事項 14
4.4. -ffar_pointer(-fFP)または-ffar_RAM(-fFRAM)使用時に標準ライブラリがリンクできない場合があります。 14
4.5. Code hoisting optimize(if-else 文の両方の複文から共通式を if 文の前にまとめる最適化)を強化しました。 14
4.6. NC30 Ver.5.xx では、引数が構造体または...のインライン関数定義がアセンブリマクロでした。しかし Ver.6.00 からは通常関数になります。 14
4.7. 「ファイルスコープに宣言した関数および変数」と同名の関数および変数を関数内で extern 宣言すると、型が異なっていてもエラーにならず、関数内の宣言が有効になる問題を修正しました。 15
4.8. 共用体のメンバに連続して書き込む場合に、順序を誤って書き込むことがある不具合を修正しました。 15
4.9. 整数と配列名の加減式に sizeof 演算子を使用すると誤ったサイズを返す問題を修正しました。 15
4.10. 浮動小数点型の定数式が正規化数の最小値より小さい場合にゼロとならない問題を修正しました。 16
4.11. 0.0 と NaN を比較する条件式に対して、誤って等値と判定してしまう問題を修正しました。 16
4.12. コンパイラオプション-OS_MAX 選択時に前方参照の inline 関数がエラーになる問題を修正しました。 16
4.13. 宣言で倍精度浮動小数点型と定義で単精度浮動小数点の型を持つ引数の読み取りを誤る問題を修正しました。 17
4.14. 旧形式の関数定義で_Bool 型を持つ引数から読み取みとりを誤る問題を修正しました。 18
4.15. switch 文の制御式の型が signed char の場合、正しい case ラベルに分岐しない問題を修正しました。 18
4.16. 自動変数の構造体の初期化時に構造体を返す関数を呼び出すと System Error が発生する問題を修正しました。 19
4.17. コンパイルオプション-Oloop_unroll(-OLU)指定時に for 文中のインライン関数呼び出しがアセンブルエラーになる問題を修正しました。 19
4.18. #if、#elif に続く定数式に記述した LONG_MAX より大きな定数を正しく解釈できない問題を修正しました。 20
4.19. 標準入出力関数に関する注意事項を修正しました。 20
4.20. アセンブラ指示命令 .id、.ofsreg に関する注意事項を修正しました。 20
4.21. 右シフト演算に関する注意事項を修正しました。 20
4.22. 構造体のメンバを sizeof 演算子を含む式で初期化する場合の注意事項を修正しました。 20
4.23. コンパイルオプション-Ostack_frame_align(-OSFA)を使用する場合の注意事項を修正しました。 20
4.24. 不完全型構造体または共用体の型定義に関する注意事項を修正しました。 21
5. 統合開発環境(High-performance Embedded Workshop)プロジェクト変換 22
5.1. アセンブラスタートアップを使用されている場合の設定 22
6. ソフトウエアのバージョン一覧 23
7. TM→HEW (NC30WA Ver.5.xx) 移行方法 24
7.1. 概要 24
7.2. 変換手順 24
7.3. 注意事項 26
7.3.1. 移行できる情報、できない情報 26
7.3.2. クロスツール 26
7.3.3. High-performance Embedded Workshop のバージョン 26
7.3.4. ロードモジュールコンバータ 27
7.3.5. 外部ツール 28
7.3.6. リンク順序 32
7.3.7. スタートアッププログラムの先頭リンク 33

1. C/C++コンパイラパッケージのインストール

インストールについては、インストールガイドをご覧ください。

2. 最新情報のご案内

本製品の最新情報については以下を参照してください。

http://japan.renesas.com/nc30wa

http://www.renesas.com/nc30wa

本製品の最新ユーザーズマニュアルとリリースノートについては以下を参照してください。

http://japan.renesas.com/nc30wa_document

http://www.renesas.com/nc30wa_document

本製品のツールニュースについては以下を参照してください。

http://tool-support.renesas.com/jpn/toolnews/p_m16c_1.htm

http://tool-support.renesas.com/eng/toolnews/p_m16c_1.htm

3. 注意事項

本製品をご使用いただく際に以下の注意事項があります。

3.1. リアルタイムOS　MR30 の対応バージョンについて

- M3T-MR30/4

  Ver.4.00 Release 01 以前のバージョンとの組み合わせは動作保証はしておりません。

- MR8C/4

  Ver.1.00 Release 00 との組み合わせは動作保証はしておりません。

3.2. ファイル名に関する注意事項

ソースプログラムファイルの名前や作業を行うディレクトリ名、ワークスペース名は、次の注意事項に従ってください。

- ieee-lmc30 が使用する入出力ファイルは、ASCII 以外の文字を含むディレクトリ名、ワークスペース名およびファイル名は使用できません。
- ファイル名に使用するピリオド(.)は一つのみ使用可能です。
- ネットワークパス名は使用できません。ドライブ名に割り当ててご使用ください。
- 「ショートカット」は使用できません。

※　ワークスペースとは、統合化開発環境 High-performance Embedded Workshop 上でコンパイル/ビルド/デバッグ等を行う作業ディレクトリです。

3.3. ウィルスチェックプログラムに関する注意事項

ウィルスチェックプログラムが常駐した状態で M16C コンパイラを起動すると正常に起動しない場合があります。その場合は、ウィルスチェックプログラムの常駐を解除してから M16C コンパイラを起動しなおしてください。

3.4. 英語PC上でのモトローラSフォーマットの比較方法について

モトローラ S フォーマットの比較方法は、下記リンク先の Srecord パッケージ(srec_cmp)等を使って比較してください。

http://srecord.sourceforge.net/

(なお、このソフトウェアのご利用によって生じたトラブルやその他の損害について当社は責任を負うものではありません。)

[インストールフォルダ]¥Tools¥Renesas¥nc30wa¥vXXXrXX¥sample¥mot_compare

に S フォーマットファイルを比較するためのサンプルを用意しましたので参考にしてください。

3.5. 機種依存部に関する注意事項

3.5.1. 割り込み制御レジスタに関する注意事項

最適化オプション "-O5" を指定するとビット操作命令 (BTSTS,BTSTC) を生成する可能性があります。BTSTS、BTSTC 命令は、M16C の割り込み制御レジスタを書きかえる命令として使用できません。
本オプションを指定する場合は、必ず生成されたコードに問題が無いことをご確認ください。

- 発生例

以下のプログラムで最適化オプション "-O5" を指定した場合、最適化により BTSTC 命令を生成します。このため、割り込み要求ビットの判定が正常に行われず意図しない動作を行います。

```
#pragma ADDRESS TA0IC 55H
struct {
        char ILVL:3;
        char IR :1;                     /* 割り込み要求ビット **/
        char dmy :4;
}TA0IC;
void wait_until_IR_is_ON(void)
{
        while(TA0IC.IR ==0){            /* 1 になるまで待つ **/
                ;
        }
        TA0IC.IR =0;            /* 1 になったら 0 に戻す **/
}
```

- 対策

(1) 該当する最適化オプションに加えてオプション "-O5OA" を指定することにより BTSTC、BTSTS 命令を生成する最適化を抑止してください。

(2) 以下のように "asm 関数 "を挿入することにより最適化を抑止してください。

```
while(TA0IC.IR ==0){
        asm();                  /*  asm 関数を挿入。TA0IC に対して処理を抑止します。*/
}
```

- 注意

オプション "-O5OA" または asm 関数の使用による対策後は、BTSTC、BTSTS 命令が生成されていないことを必ずご確認ください。

3.5.2. SFR領域のアクセスに関する注意事項

SFR 領域のレジスタをアクセスする場合には、特定の命令を使用しなければならないことがあります。
この特定の命令は機種毎に異なりますので詳しくは各機種のユーザーズマニュアルなどを参照してください。本注意事項にかかわる命令は、asm 関数等のインラインアセンブル機能を使用してプログラム中に命令を直接記述してください。

3.5.3. 割り込み優先レベルに値を設定する場合の注意事項

テクニカルニュース(No.M16C-14-9804)「M16C/60、M16C/61、M16C/62、M16C/63 グループ割込み制御レジスタの注意事項」に対応するため割り込み優先レベルのセットおよび変更を行う関数をサポートしています。使用方法は、以下の通りです。

- セットする場合

SetLevel 関数をご使用ください。この時、intlevel.h ファイルを必ずインクルードしてください。

SetLevel(char *adr, char val);

adr　：割り込み制御レジスタのアドレス
val　：セットする値

- 変更の場合

ChgLevel 関数をご使用ください。この時、intlevel.h ファイルを必ずインクルードしてください。

ChgLevel(char *adr, char val);

adr　：割り込み制御レジスタのアドレス
val　：セットする値

【例】

```
#include <intlevel.h>
#pragma ADDRESS timerA 55H
char   timerA;
void   func(void)
{
        SetLevel(&timerA,2); // 割り込み優先レベルを 2 に設定

        ChgLevel(&timer A,4); // 割り込み優先レベルを 4 に変更
}
```

3.5.4. オンチップデバッガE8aを使用する場合、「E8」を選択して下さい。

オンチップデバッガ E8a を使用する場合、High-performance Embedded Workshop で以下の方法で新規プロジェクトを作成して下さい。
(1) プロジェクトタイプ「C source startup Application」を選択
(2) リスト「オンチップエミュレータ使用」から「E8」を選択

3.6. コンパイラ、アセンブラ、最適化リンケージエディタおよびユーティリティに関する注意事項

3.6.1. 最適化リンケージエディタ(optlnk)のセクションの配置について

最適化リンケージエディタ(optlnk)は、旧リンケージエディタ(ln30)とセクションの配置方法が変更になるので注意してください。
次のようなアセンブラソースを記述した場合、

```
.section    prg1
.org        100h
.section    prg2
.section    prg3
.org        200h
.section    prg4
.end
```

ln30 では、-order 指定無しの場合は、次のようなセクション順の配置になります。しかし最適化リンケージエディタでは相対セクションの配置が異なります。

```
100H    prg1
        prg2
200H    prg3
        prg4
```

最適化リンケージエディタで ln30 と同じセクション配置にするためには、-start オプションを用いてセクションを配置してください。

```
optlnk    . . . -start=prg1,prg2/100,prg3,prg4/200
```

.ORG を使用している場合には、-start オプションの指定アドレスと一致させてください。一致しない場合にはウォーニングとなり、.ORG の設定値が使われます。
.ORG を使用しない場合には、-start オプションで自由にアドレスを指定することができます。

3.6.2. 同名のローカル変数が同じ関数内にある場合のデバッガ表示について

関数内の異なるブロックに同名のローカル変数が宣言されている場合、デバッガでその値を正しく表示できない問題があります。そのため、異なる名前の変数名で宣言してください。

3.6.3. リンク時のエラーレベルを下げて生成したABSファイルの取扱いについて

リンク時にオプションでエラーレベルを下げると、エラーが発生していても強制的にロードモジュールファイル（ABS）を生成させることができますが、このようなロードモジュールファイルを使用しないでください。

3.6.4. C++言語で不完全列挙型を使用するオブジェクトを定義するとエラーとなる場合について

列挙型の定義を記述して完全型としてください。

3.6.5. デバッグ指定時、template関数にexternまたはstaticを指定するとエラーとなる場合について

デバッグ指定（-g オプション）時に extern または static 宣言を指定するとエラーになりますが、この記述は文法エラーですので extern/static を指定しないように記述を変更してください。

3.6.6. キーワードmutableはサポートしていません。

3.6.7. 関数型キャスト「型(式)」形式で、型のnear/farを先行して記述するとエラーとなる問題について

キャストへの near/far 記述は動作に影響しないので記述しないでください。

3.6.8. C++言語での#pragma ASMの使用について

C++言語のソースでは、関数外に#pragma ASM を書くとコンパイルエラーになります。asm()は関数外でも記述できますので、関数外にアセンブル命令を書く必要がある場合は、asm()をご使用ください。
C 言語のソースでは、従来通り#pragma ASM, asm()とも関数外で使用できます。

3.6.9. C++言語でのアセンブラマクロ関数の仕様について

#pragma ASM を用いて定義したアセンブラマクロ関数は、C++言語ソースファイルで使用することができません。
この場合には、下記の例のように、#pragma ASM ではなく asm 関数を用いるようにマクロ定義を変更してください。

＜例＞

```
#pragma __ASMMACRO addition(R0,R2)
static int addition(int, int);
    _asm("_addition .macro¥n"
        "           add.w     R2,R0¥n"
        "           .endm");
```

3.6.10. 関数内に#pragma ASM, asm()またはアセンブラマクロ関数呼び出しを使用して自関数を参照する場合の制限事項について

次のような制限事項があります。
・.map 内に出力される該当関数の参照回数が正しく出力されません。
・該当関数が未参照シンボルであっても optlnk の最適化で削除されません。

＜例＞

```
void func(void)
{
#pragma asm
   .call _func,G
   jsr _func
#pragma endasm
}
```

3.6.11. Ver.5.xxからのVer.6.xxへ移行時のinit.c,device.cの登録について

標準入出力を使用しているVer.5.xx のプロジェクトをVer.6.xx へ移行する場合、init.c とdevice.c を移行後のプロジェクトへ登録する必要があります。以下の方法で生成した init.c と device.c のコピーを、移行後のプロジェクトに登録して下さい。
・HEW を立ち上げる。
・新規プロジェクトを作成を選択する。
・MCU を選択する。
・チェックボックス「I/O ライブラリを使用」を有効にする。
ただし、M16C/24 の場合、Ver.5.xx の標準入出力は UART0 でしたが、Ver.6.xx の標準入出力は UART1 です。コンパイラオプション-D__UART0__を使用して device.c をコンパイルすることで、標準入出力を UART0 に変更できます。

3.6.12. C++動的初期化に関する注意事項

const が far 属性を持つ場合、C++動的初期化を伴う関数外宣言 const 変数の配置セクションは bss_FE または bss_FO になります。該当変数宣言の const 修飾子をはずすことで、配置セクションを bss_NE または bss_NO に変更できます。

3.6.13. コンパイラユーザーズマニュアルの標準ライブラリ関数scanfの解説に誤りがあります。

[誤]
「argument は各変数の far 型ポインタでなければなりません。」
[正]
「argument は各変数の near 型ポインタでなければなりません。」

3.6.14. printf系関数の変換指定に対応する実引数の型がdouble型の結果が正しくありません。

・発生条件
次の条件を全て満たした場合に発生します。
(1) 以下のいずれかのコンパイラオプションを使用している。
(a) -fdouble_32(-fD32)
(b) -OR_MAX(-ORM)
(c) -OS_MAX(-OSM)

(2) 以下のライブラリ関数のいずれかを呼び出している。
(d) printf
(e) fprintf
(f) sprintf
(g) vprintf
(h) vfprintf
(i) vsprintf
(3) (2)の変換指定に対応する実引数の型が double 型である。

・発生例
[コマンドライン]
```
nc30 -c -fD32 sample.c /*  発生条件(1)(a) */
```
[sample.c]
```
#include <stdio.h>
void func(double x)
{
    printf("%f¥n", x); /*  発生条件(2)(3) */
}
```

・回避策
long double でキャストして下さい。
[回避例]
```
#include <stdio.h>
void func(double x)
{
    printf("%f¥n", (long double)x);
}
```

3.6.15. scanf系関数の変換指定に対応する実引数の型がdouble型へのポインタの場合、結果が正しくありません。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 以下のいずれかのコンパイラオプションを使用している。

(a) -fdouble_32(-fD32)

(b) -OR_MAX(-ORM)

(c) -OS_MAX(-OSM)

(2) 以下のライブラリ関数のいずれかを呼び出している。

(d) scanf

(e) fscanf

(f) sscanf

(3) (2)の変換指定に対応する実引数の型が double 型へのポインタ型である。

・発生例

[コマンドライン]

```
nc30 -c -fD32 sample.c /*  発生条件(1)(a) */
```

[sample.c]

```
#include <stdio.h>
void func(double* x)
{
    scanf("%lf¥n", x); /*  発生条件(2)(3) */
 }
```

・回避策

変換修飾子 l をはずして下さい。

[回避例]

```
#include <stdio.h>
void func(double* x)
{
    scanf("%f¥n", x); /*  変換修飾子 l をはずす  */
}
```

3.6.16. assertマクロの仕様について

マクロ assert は以下の仕様になっています。
機能:
　プログラム中に診断機能を付け加えます。
書式:
　#include <assert.h>
　void assert(test)
実現方法:
　マクロ
引数:
　test; 評価する式
戻り値:
　戻り値はありません。
解説:
　assert マクロは、test が真の時は値を返さずに処理を終了します。test が偽の時は、診断情報をコンパイラによって定義された書式で標準エラーファイルに出力し、その後 abort 関数を呼び出します。
　診断情報の中には、パラメータのプログラムテキスト、ソースファイル名、ソース行番号が含まれています。
　assert マクロを有効にする場合、標準入出力初期化関数_init を呼び出す必要があります。
　assert マクロを無効にする場合、assert.h を取り込む前にマクロ NDEBUG を定義します。
処理系定義仕様:
　assert(test)において、test が偽の時メッセージを出力します。
　Assertion failed:△式,△<ファイル名>(line△<行番号>)

3.6.17. コンパイラオプション-Wlarge_to_small(-WLTS)に関する注意事項

コンパイラオプション-Wlarge_to_small(-WLTS)を使用する場合、以下の点にご注意下さい。
(1) C++プログラムとしてコンパイルする場合、右辺が定数の場合のみ警告を出力します。
(2) C プログラムとしてコンパイルする場合、右辺が変数のみの場合、警告を出力しません。

3.6.18. R8Cファミリ(ROM64KB以上)において固定ベクタを避けてセクションを配置する方法

リンク時に-cpu オプションを設定することで、固定ベクタを避けてセクションを配置できます。
　　例:-cpu=RAM=400-11ff,ROM=4000-ffd7,ROM=10000-23fff -cpu=stride
※ 上記例では、ベクタ領域(0ffd8H 番地から 0ffffH 番地)の
範囲に対して ROM/RAM の割り当てを抑止します。
不連続な RAM(例: R8C/2A～R8C/2D)への自動配置も同様に可能です。
High-performance Embedded Workshop の GUI からは、以下の手順で設定します。

(1) メニュー「ビルド」から「Renesas M16C Standard Toolchain」を選択
(2) タブ「リンカ」を選択
(3) リスト「カテゴリ」から「ベリファイ」を選択
(4) リスト「オプション項目」から「ベリファイ」を選択
(5) ボタン「追加」でメモリ種別、先頭アドレス、終了アドレスを設定
(6) チェックボックス「-cpu=stride」を有効に設定

3.7. 制限事項、および既知の不具合

下記の制限事項、および既知の不具合がありますので、注意してください。

- 異なる内部プロセスが、重複する警告やエラーメッセージを出力する場合があります。
- C++言語で 2 進数表記のアンダースコア('_')を使うとエラーになります。
- EC++クラスライブラリ使用時には-fdouble_32(-fD32)を使わないでください。
- 関数原型宣言がない関数で #pragma を使用した場合、警告もエラーメッセージも表示されません。
- クラス内先頭で定義された関数にブレークポイントを設定できない場合があります。
- #pragma entry でフレーム渡し引数や引数ではないフレーム変数を宣言してもエラーになりません。
  #pragma entry を使用する関数は、関数先頭で、ユーザースタックポインタ、割り込みスタックポインタ、フレームベースレジスタが未設定であるため、引数や自動変数を使用できません。#pragma entry を使用する関数に引数や自動変数を使用しないようにして下さい。

- 負の浮動小数点定数を符号無し整数型にキャストする式を関数呼び出しの引数に書くと、C1841 の警告が出てキャスト後の値がゼロになります。

  ・発生例

```
int func(int x) { return x; }
void main( void )
{
    int i = (int)((unsigned int)-1.0f);
    int j;
    j = func((int)((unsigned int)-1.0f));

    printf("%d, %d¥n", i, j); /* -1, 0 となります */
}
```

  ・回避策

  関数呼び出しの前に一時変数に代入して、その一時変数を関数呼び出しに使用してください。

```
int func(int x) { return x; }
void main( void )
{
    int i = (int)((unsigned int)-1.0f);
    int j;
    int tmp = (int)((unsigned int)-1.0f); /* ここ */
    j = func(tmp);

    printf("%d, %d¥n", i, j); /* -1, 0 となります */
}
```

- near 配列を far ポインタで指して、far ポインタのアドレス計算に負の値を使うと、正しい要素を取れないことがあります。

  ・発生条件

  次の条件をすべて満たす場合に発生します。
  (1) near 配列を宣言する.
  (2) far ポインタを宣言する。
  (3) (2)のポインタは、(1)の配列の 1 番目以降の要素を指す。
  (4) (3)に対する 2 次元以上のアドレス計算、かつ負の値を要素数にして、ポインタが指す先の配列要素を参照する。

・発生例

```
#include <stdio.h>

int near buf[2][1] = { { 1 }, { 2 } };
int far (*p)[1] = buf + 1;

void main( void )
{
    int i = 0;
    int j = -1;

    if( p[i][j] != 1 ) {
        printf( "NG¥n" );
    } else {
        printf( "OK¥n");
    }
}
```

・回避策

配列参照の添え字に負の値を使わないでください。

- #pragma interrupt(割り込みベクタ番号あり)を使用した関数に対するコードを生成せず、可変割り込みベクタに関数アドレスを設定しません。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 以下のいずれかに該当する。

(a) コンパイラオプション-OS_MAX(-OSM)を使用している。

(b) コンパイラオプション-Oforward_function_to_inline(-OFFTI)と以下のコンパイラオプションを併用している。

(b1) -O[1-5]

(b2) -OR_MAX(-ORM)

(2) #pragma interrupt(割り込みベクタ番号あり)を使用している。

(3) (2)の対象が内部結合関数である。

(4) (3)の関数は、アドレス参照がない。

・発生例

[コマンドライン]

```
nc30 -c -OSM sample.c /*  発生条件(1)(a) */
```

[sample.c]

```
#pragma interrupt func(vect=31)    /*  発生条件(2) */
static void func(void)    /*  発生条件(3) */
{
}
```

・回避策

発生条件に該当する関数のアドレスを参照するダミー変数を定義して下さい。

```
#pragma interrupt func(vect=31)
static void func(void)
{
}
void (*dummy)(void) = &func; /*  アドレス参照  */
```

- アセンブリ言語ファイル中で、CODE セクションの終端となる .SECTION(.END)と、末尾の命令が別のファイルに記述されている場合、正しくデバッグできません。

・発生条件

次の条件を全て満たした場合に発生します。

(1) アセンブラオプション"-N"が設定されていない。

(2) CODE セクションの終端となる .SECTION(.END)と、末尾の命令が別のファイルに記述されている。

・発生例

<sampe.a30>

```
    .SECTION prg,CODE
     NOP
     NOP
     .INCLUDE    sample.inc
     .END                    ; セクション終端となる .END
```

<sample.inc>

```
     RTS                     ; prg セクション末尾の命令
```

4. V.5.45 Release 01 からのバージョンアップ内容

V.5.45 Release01 からアップデートされた内容は以下のとおりです。また、M16C シリーズ,R8C ファミリ用 C/C++コンパイラパッケージ Ver.6.00 C/C++コンパイラユーザーズマニュアルの付録Kも合わせてご覧ください。

4.1. C++で-ffar_pointer(-fFP)を使用する場合、関数内で配列型へのポインタが 16 ビットになります。

C++言語で関数内で配列型へのポインタを 16 ビットで扱います。far 修飾子を明示することで、常に 32 ビットで扱えます。

```
int i1, i2;
void func(void)
{
    i1 = sizeof(int(*)[]);       // -fFP 使用時、C 言語で 4、C++言語で 2
    i2 = sizeof(far int(*)[]);   //  常に 4
}
```

4.2. near/farのキャスト記法について

コンパイラユーザーズマニュアルに、far ポインタから near ポインタへの変換に cons_cast を使用できると記述していますが、誤りです。

[誤]
代入する far ポインタが near 領域を指していることが明らかな場合、キャスト記法または const_cast の演算子により far ポインタを near ポインタにキャストしてエラーを回避してください。
[正]
代入する far ポインタが near 領域を指していることが明らかな場合、キャスト記法により far ポインタを near ポインタにキャストしてエラーを回避してください。

4.3. シンボル__SB__の.glb宣言に関する注意事項

コンパイラが生成するシンボル __SB__の.glb 宣言を以下のように変更しました。

・Ver.5.xx
__SB__を使う・使わないと関係なく、常に .glb __SB__を出力します。

・Ver.6.xx
__SB__を C ソースでご使用されていることが分かる場合のみ、.glb __SB__を出力します。
出力条件は以下のいずれかを満たす場合になります。

a) #pragama SBDATA を記述する。
b) コンパイラオプション-fauto_over_255(-fAO2)を使用し、スタックが 255 バイト以上使用する関数を定義する。
c) #pragma TASK(リアルタイム OS 用プラグマ)を使用した関数を定義する。

上記変更により、スタートアッププログラムを C ソースで自作している Ver.5.xx プロジェクトを Ver.6.xx プロジェクトに移行する場合、シンボル__SB__に対して未定義シンボルのエラーが発生します。
シンボル__SB__を定義しているファイルに以下の記述を追記してエラーを回避して下さい。

```
_asm("　　.glb　　__SB__");
```

4.4. -ffar_pointer(-fFP)または-ffar_RAM(-fFRAM)使用時に標準ライブラリがリンクできない場合があります。

4.5. Code hoisting optimize（if-else文の両方の複文から共通式をif文の前にまとめる最適化）を強化しました。

4.6. NC30 Ver.5.xxでは、引数が構造体または...のインライン関数定義がアセンブリマクロでした。しかしVer.6.00 からは通常関数になります。

4.7. 「ファイルスコープに宣言した関数および変数」と同名の関数および変数を関数内でextern宣言すると、型が異なっていてもエラーにならず、関数内の宣言が有効になる問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) ファイルスコープに変数または関数の宣言がある。

(2) 関数定義に内に、(1)と同名で型が異なる extern 宣言がある。

(3) (2)の関数定義内で、(2)の宣言が有効な位置で、(2)の変数または関数にアクセスする。

4.8. 共用体のメンバに連続して書き込む場合に、順序を誤って書き込むことがある不具合を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 最適化オプション-O[1-5], -OS, -OR, -OS_MAX,もしくは-OR_MAX を選択している。

(2) 1 つの共用体の異なるメンバに定数値を代入する代入文が複数連続している。

(3) (2)の代入先は、volatile 修飾されていない。

(4) (2)の代入先の複数のメンバは、異なるサイズのものが混在している。

・発生例

```
union{
    unsigned int   unim01;
    unsigned int   unim02;
    unsigned long unim05;
} uni[2];

void main(){
    uni[0].unim01 = 0x1111;
    uni[1].unim01 = 0x1111;
    uni[0].unim02 = 0x2222;
    uni[0].unim05 = 0x55555555; /*  順序が誤って入れ替わるため、この値が残らない  */
}
```

4.9. 整数と配列名の加減式にsizeof演算子を使用すると誤ったサイズを返す問題を修正しました。

整数と配列名の加減式に sizeof 演算子を使用すると、ポインタサイズでなく配列のサイズが加算された値になる問題がありました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) sizeof 演算子により、整定数と配列名の加減式の型のサイズを取得している。

(2) (1)の加減式は、加減演算子の左オペランドが整数定数で、右オペランドは配列の名前である。

・発生例

```
short arr[30],i;
int test(void)
{
    i = sizeof(0+arr); /* &arr[0]のアドレス式のサイズでなく、誤って配列のサイズを返す  */
}
```

4.10. 浮動小数点型の定数式が正規化数の最小値より小さい場合にゼロとならない問題を修正しました。

浮動小数点型の定数または定数同士の演算結果について、その絶対値が正規化数の最小値より小さくてもゼロにまるめずに演算してしまうことがありました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 浮動小数点型の定数又は浮動小数点型の定数式を使用している。

(2) (1)の絶対値が正規化数の最小値(float 型の場合 FLT_MIN,double 型の場合 DBL_MIN)より小さい。

・発生例

```
float test()
{
    float f = 1.1760000000e-38F - 0.0020000000e-41F;
    return f;   /* 減算結果が FLT_MIN より小さく 0 を返すべきところ、非正規化数を返す */
}
```

4.11. 0.0 とNaNを比較する条件式に対して、誤って等値と判定してしまう問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) コンパイルオプション -O[1-5]、-OR、-OS、-OR_MAX(-ORM)、または-OS_MAX(-OSM)を選択している。

(2) 等価、または不等価演算で比較する制御式がある。

(3) (2)の制御式の一方は NaN の値を持つ変数である。

・発生例

```
int test(void)
{
    float f  = 3.402823466e+38 + 0.001e+38; /* オーバフローにより無限大になる */
    float f1 = 3.402823466e+38 + 0.001e+38; /* オーバフローにより無限大になる */
    f /= f1;                                /* 無限大同士の除算により NaN になる */
    if (f == 0.0f) {                        /* 最適化により誤って常に真と判定される */
        return 0;
    }
    return 1;
}
```

4.12. コンパイラオプション-OS_MAX選択時に前方参照のinline関数がエラーになる問題を修正しました。

コンパイラオプション-OS_MAX を選択したとき、-Oforward_function_to_inline(-OFFTI)オプションが有効にならず、前方参照の inline 関数がエラーになる問題がありました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) コンパイルオプション -OS_MAX(-OSM) を指定している。

(2) inline 修飾された関数が定義されている。

(3) (2)の inline 修飾された関数は、定義より前に呼び出されている。

・発生例

```
inline int add(int, int);

int func(void)
{
    int rc = add(1, 2);
    return rc;
}

inline int add(int a, int b)
{
    return a + b;
}
```

4.13. 宣言で倍精度浮動小数点型と定義で単精度浮動小数点の型を持つ引数の読み取りを誤る問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 以下のコンパイラオプションを使用している。
  (a) -O[1-5]
  (b) -OR
  (c) -OR_MAX(-ORM)
  (d) -OS
  (e) -OS_MAX(-OSM)

(2) 倍精度浮動小数点型引数を持つ関数を宣言している。

(3) (2)の関数が旧形式で定義されている。

(4) (2)の引数が、定義では単精度浮動小数点型である。

(5) (4)の引数の最初のアクセスが読み取りである。

・発生例

```
nc30 -c -O1 xxxx.c /*  発生条件(1) */

#include <stdio.h>
float func(double); /*  発生条件(2) */
float func(f) /*  発生条件(3) */
    float f; /*  発生条件(4) */
{
    return f; /*  発生条件(5) */
}
void main(void)
{
    float x = func(1.0); /*  発生条件(4) */
    if (x == 1.0) {
        printf("OK¥n");
    } else {
        printf("NG¥n");
    }
}
```

4.14. 旧形式の関数定義で_Bool型を持つ引数から読み取みとりを誤る問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 関数が旧形式で定義されている。

(2) (1)の引数が_Bool 型である。

(3) (2)の引数を変更する前の値を読み取っている。

(4) (2)の引数の値が 2 以上の偶数である。

・発生例

```
#include <stdio.h>
_Bool func(x) /*  発生条件(1) */
    _Bool x; /*  発生条件(2) */
{
    return x; /*  発生条件(3) */
}
void main(void)
{
    _Bool x = func(2); /*  発生条件(4) */
    if (x == 1) {
        printf("OK¥n");
    } else {
        printf("NG¥n");
    }
}
```

4.15. switch文の制御式の型がsigned charの場合、正しいcaseラベルに分岐しない問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) switch 文の制御式の型が signed char である。

(2) (1)の switch 文が、以下のいずれかを満たす。

(A) case 値の最小値が "-127" かつ最大値が "127" で、その case 値が-127 から 127 まで隙間なく連続している

(B) case 値の最小値が "-128" かつ最大値が "126" で、その case 値が-128 から 126 まで隙間なく連続している

(C) case 値の最小値が "-128" かつ最大値が "127" で、その case 文(default を含まない)の総数が 135 個以上存在する。

(3) (1)の制御式の値がゼロより小さい。

・発生例

```
#include <stdio.h>
signed char d = -1; /*  発生条件(3) */
void main( void )
{
    switch( d ) { /*  発生条件(1) */
        case -127 : printf( "NG...[-127]--->[%d]¥n", d ) ; break ; /*  発生条件(2)(a) */
        …
        case   -1 : printf( "OK¥n", d ) ; break ;
        …
        case  127 : printf( "NG...[127]--->[%d]¥n", d ) ; break ; /*  発生条件(2)(a) */
    }
}
```

4.16. 自動変数の構造体の初期化時に構造体を返す関数を呼び出すとSystem Errorが発生する問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 構造体、または共用体を返す関数を宣言している。

(2) (1)の関数の第 1 引数はスタック渡しである。

(3) (1)の関数を呼び出して、自動変数を初期化している。

・発生例

```
struct S {
    int i;
    int j;
};

struct S func(long);

int test(void)
{
    struct S s = func(0x1234);
    return 1;
}
```

4.17. コンパイルオプション-Oloop_unroll(-OLU)指定時にfor文中のインライン関数呼び出しがアセンブルエラーになる問題を修正しました。

コンパイルオプション-Oloop_unroll (-OLU)と-Oforward_function_to_inline(-OFFTI)を併用し、ループ展開の対象となる繰り返し文中にインライン展開される関数呼び出しを記述するとアセンブルエラーが発生する問題を修正しました。

・発生条件

次の条件を全て満たした場合に発生します。

(1) 以下のコンパイラオプションを両方とも選択している。

・-Oforward_function_to_inline(-OFFTI)

・-Oloop_unroll(-OLU)

(2) コンパイラオプション-Oloop_unroll(-OLU)で
展開対象となるループ中にインライン展開される関数呼び出しを記述している。
(例 1) inline 宣言した関数をループ中で呼び出す場合
(例 2) コンパイラオプション-Ostatic_to_inline(-OSTI)を選択し、かつ static 宣言した関数をループ中で呼び出す場合

・発生例

```
int gi;
inline void inline_func(void)
{
    ++gi;
}
```

```
void func(void)
{
    char i = 0;
    for( i = 0; i <= 4; i++ ){
        inline_func();
    }
}
```

4.18. #if、#elifに続く定数式に記述したLONG_MAXより大きな定数を正しく解釈できない問題を修正しました。

・発生例

```
#include <stdio.h>
void main()
{
#if   2147483648 > 0
    printf("2147483648 > 0¥n");     // Ver.6.xx
#else
    printf("2147483648 <= 0¥n"); // Ver.5.xx
#endif
}
```

4.19. 標準入出力関数に関する注意事項を修正しました。

該当 FAQ
http://japan.renesas.com/support/faqs/faq_results/Q1000000-Q9999999/tools/coding_tools/compiler_100706a.jsp
http://www.renesas.com/support/faqs/faq_results/Q1000000-Q9999999/tools/coding_tools/compiler_100706a.jsp

4.20. アセンブラ指示命令 .id、.ofsregに関する注意事項を修正しました。

該当 FAQ
http://japan.renesas.com/support/faqs/faq_results/Q104601-Q104700/tool_faq_2005020701.jsp
http://www.renesas.com/support/faqs/faq_results/Q107401-Q107500/compiler_107459_en_GL.jsp

4.21. 右シフト演算に関する注意事項を修正しました。

該当ツールニュース
http://tool-support.renesas.com/jpn/toolnews/070716/tn4.htm
http://tool-support.renesas.com/eng/toolnews/070716/tn4.htm

4.22. 構造体のメンバをsizeof演算子を含む式で初期化する場合の注意事項を修正しました。

該当ツールニュース
http://tool-support.renesas.com/jpn/toolnews/080716/tn2.htm
http://tool-support.renesas.com/eng/toolnews/080716/tn2.htm

4.23. コンパイルオプション-Ostack_frame_align(-OSFA)を使用する場合の注意事項を修正しました。

該当ツールニュース
http://tool-support.renesas.com/jpn/toolnews/070701/tn5.htm
http://tool-support.renesas.com/eng/toolnews/070701/tn5.htm

4.24. 不完全型構造体または共用体の型定義に関する注意事項を修正しました。

該当リリースノート

http://tool-support.renesas.com/jpn/toolnews/100401/tn3.htm
http://tool-support.renesas.com/eng/toolnews/100401/tn3.htm

5. 統合開発環境(High-performance Embedded Workshop)プロジェクト変換

5.1. アセンブラスタートアップを使用されている場合の設定

C/C++コンパイラユーザーズマニュアル「付録 K バージョンアップ内容と移行方法」の”K.1.2 統合開発環境(High-performance Embedded Workshop)プロジェクト変換”に記載されている内容は、M16C シリーズを使用されている場合の設定方法になります。したがいまして、R8C ファミリのアセンブラスタートアップを使用されている場合は、optlnk のセクションの開始アドレス指定オプション”-start”に下記内容を設定する必要があります。

[ROM 64K 未満]
-start=data_SE,bss_SE,data_SO,bss_SO,data_NE,bss_NE,data_NO,bss_NO,istack,stack,heap/0400,rom_NE,rom_NO,data_SEI,data_SOI,data_NEI,data_NOI,switch_table,program,interrupt/0E000,vector/0FED8

[ROM 64K 以上]
-start=data_SE,bss_SE,data_SO,bss_SO,data_NE,bss_NE,data_NO,bss_NO,istack,stack,heap/0400,rom_NE,rom_NO,data_SEI,data_SOI,data_NEI,data_NOI,switch_table,program,interrupt/04000,vector/0FED8,rom_FE,rom_FO/010000

※ 各セクションの開始アドレスは、お客様の設定に変更する必要があります。
※ お客様でセクションを追加された場合は、上記設定に追加する必要があります。

オプション”-start”設定方法)

- 統合開発環境から[ビルド→Renesas M16C Standard Toolchain] → [リンカ] → [カテゴリ:セクション]を選択します。
- [追加]を選択し、”セクションの追加”で各セクションを割り付けます。

アドレス	セクション
0x400	data_SE,bss_SE,data_SO,bss_SO,data_NE,bss_NE,data_NO,bss_NO,istack,stack,heap
0x0E000	rom_NE,rom_NO,data_SEI,data_SOI,data_NEI,data_NOI,program,interrupt
0x0FED8	vector

- 各セクションの割り付けが完了します。

【注意事項】
各セクションの割り付け完了後、ビルド実行時に optlnk のウォーニング”L1323 (W) Section attribute mismatch:セクション名”が出力される場合があります。
ウォーニングを回避する場合は下記ソースの変更をお願い致します。

- ウォーニングが発生したセクション名に対するアドレス宣言(.ORG)をコメント化してください。 (上記例では、sect30.incのセクション名”data_SE”、”rom_NE”および”vector”が対象)

- sect30.incに記述されている下記コードがアセンブルエラーになる場合、アセンブラ指示命令”.EQU”のオペランドに記述されている”data_SE_top”を”data_SE”の開始アドレス値に変更してください。

```
; SBDATA area definition
        .glb    __SB__
__SB__          .equ            data_SE_top     ;   上記例では
                                                ;   ~~data_SE_top .equ 400H~~  ;  __SB__ .equ 400H
                                                ;   に変更
```

6. ソフトウエアのバージョン一覧

C/C++コンパイラパッケージ V.6.00 に含まれているソフトウェアの各バージョンは以下のとおりです。

- nc30.exe　V.7.00.01.000
- rcfrt.exe　V.3.10.1
- ccom30.exe　V.6.00.02.000
- aopt30.exe　V.1.05.02.000
- sbauto.exe　V.1.00.00.000
- as30.exe　V.6.00.01.000
- mac30.exe　V.3.44.01.000
- pre30.exe　V.1.12.01.000
- asp30.exe　V.6.00.04.000
- optlnk.exe　V. 10.01.00.000
- lb30.exe　V.1.02.00.000
- ieee-lmc30.exe　V.4.03.00.000
- utl30.exe　V.1.01.02
- lbg30.exe　V.1.00.000
- conv30.exe　V.1.00.00.000
- elfconv.exe　V.1.00.00.000
- prelnk.exe　V.1.3.0.0

7. TM→HEW (NC30WA Ver.5.xx) 移行方法

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。
TM→HEW(NC30WA Ver.6.xx)への移行方法につきましては、ルネサス開発環境 HomePage 内の FAQ サイトをご覧ください。

7.1. 概要

TM Ver.2.xx、Ver.3.xx で作成したプロジェクトを High-performance Embedded Workshop Ver.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。

7.2. 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。

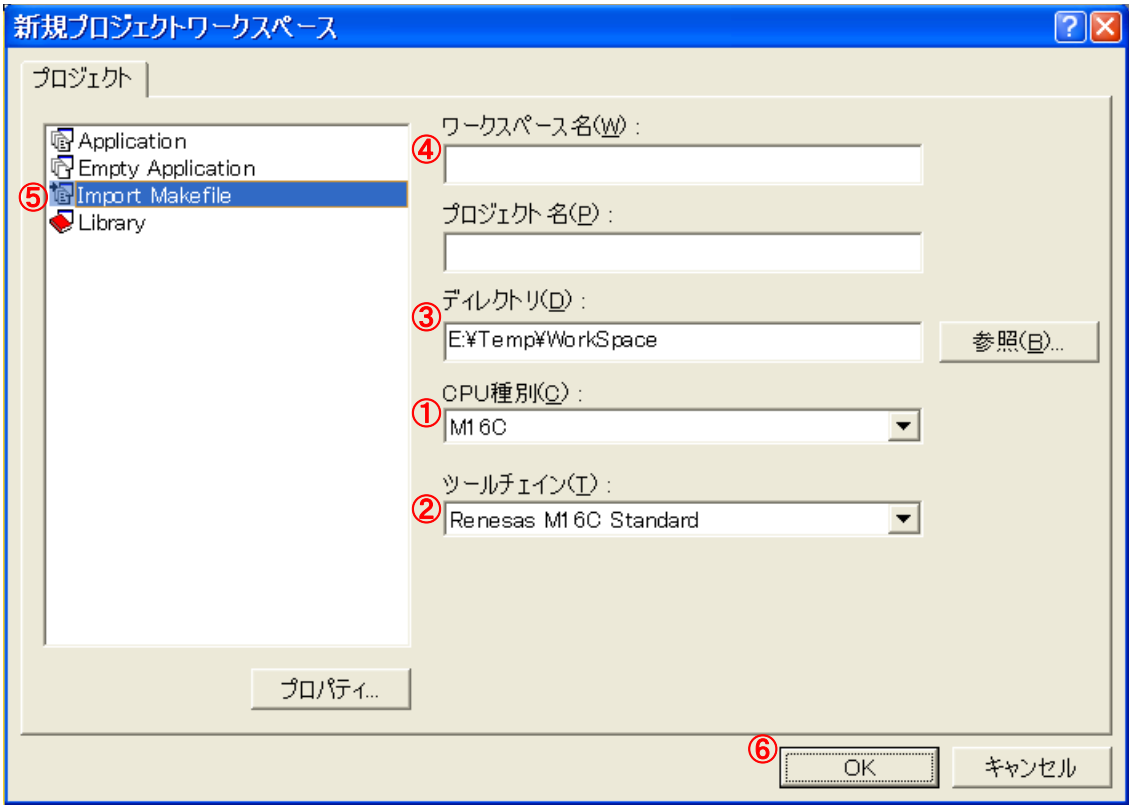

図 1 新規プロジェクトワークスペース

- 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。

- 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。

表 1 ツールチェイン名とクロスツール名

ツールチェイン名	クロスツール名
Renesas M16C Standard	NC30WA

- プロジェクトタイプから Import Makefile を選択します。
- ディレクトリを指定します。
- 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。
- 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。

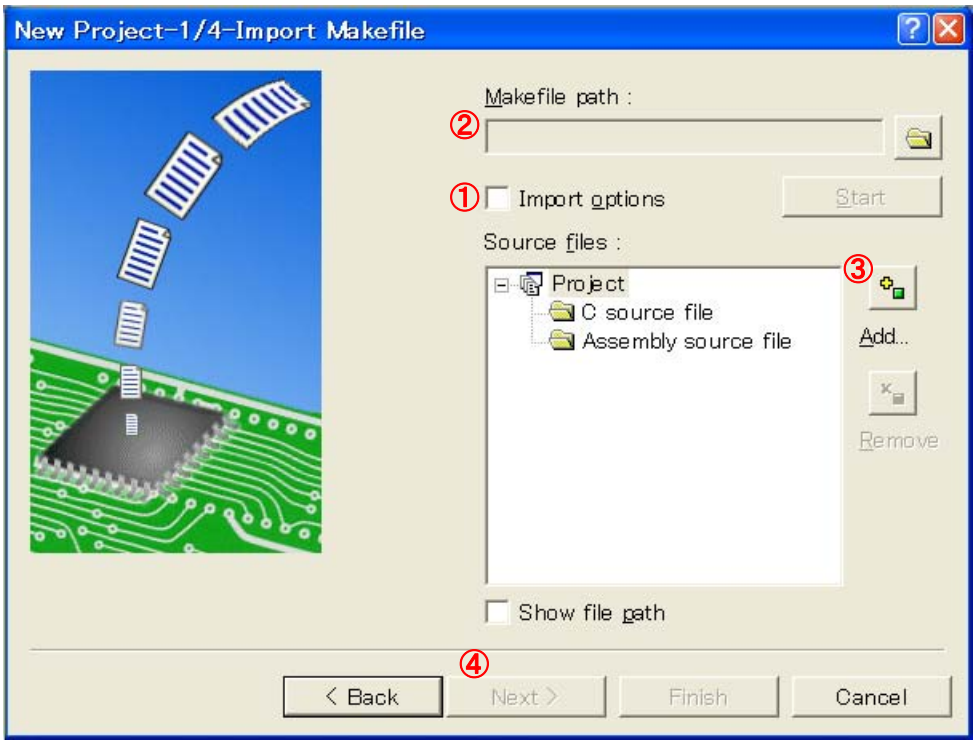

図 2 New Project-1/4-Import Makefile

- 「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)。

- 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。

- 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。
- 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。

7.3. 注意事項

7.3.1. 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。

7.3.2. 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 Release 1 ～ V.5.45 Release 01

7.3.3. 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。

表 2 High-performance Embedded Workshop バージョン毎の移行可能情報

		High-performance Embedded Workshop				
		~V.3.01.02	V.3.01.04	V.3.01.05	V.3.01.06	V.4.00
NC30WA	V.5.20 Release1	△	△	△	△	○
	V.5.30 Release1	△	△	△	△	○
	V.5.30 Release 02	---	---	---	---	○

○: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
High-performance Embedded Workshop4 がバンドルされたコンパイラ製品から順次○になります。

7.3.4. 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。

図 3 ビルドフェーズダイアログボックス

- Mxxx Load Module Converter をチェックします。
- 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。

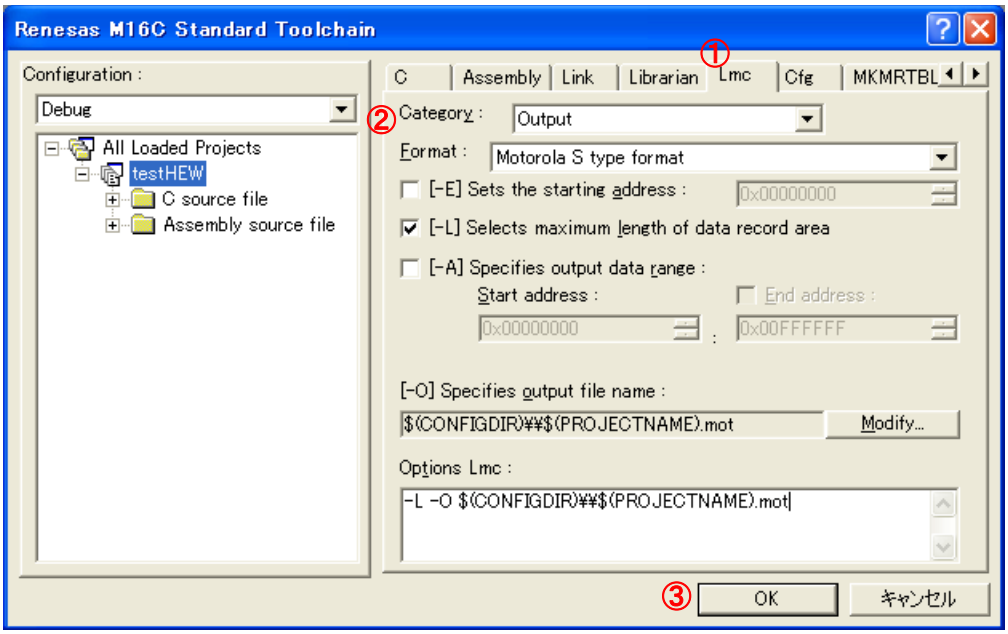

図 4 Renesas Mxxx Standard Toolchain ダイアログボックス

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

7.3.5. 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。

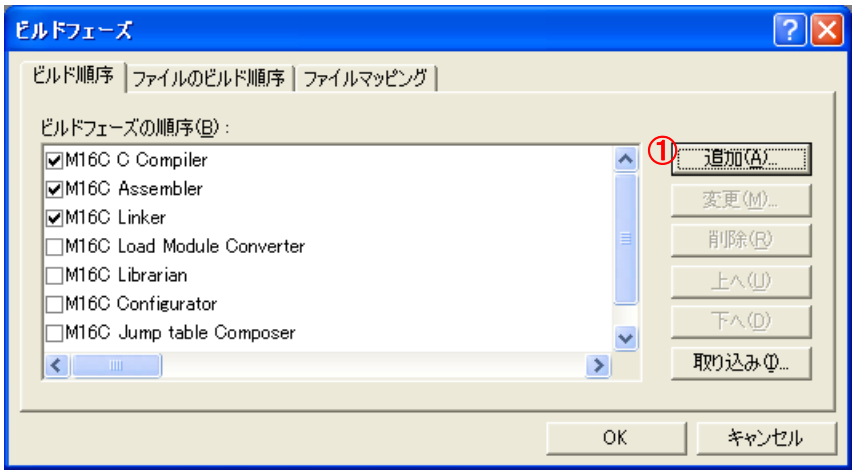

図 5 ビルドフェーズダイアログボックス

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。

図 6 新規ビルドフェーズ-1/4 ステップ

- [1/4 ステップ]次へボタンをクリックします。

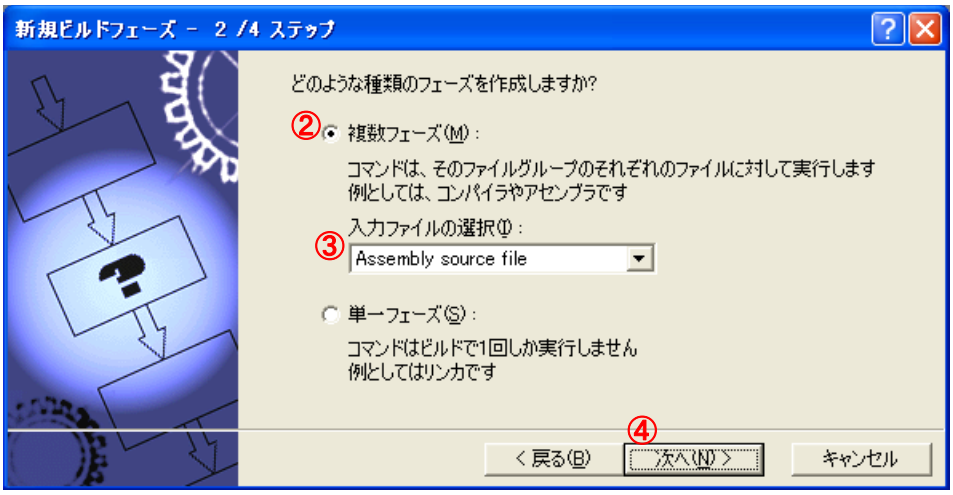

図 7 新規ビルドフェーズ-2/4 ステップ

- [2/4 ステップ]複数フェーズを選択します。
- [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。
- [2/4 ステップ]次へボタンをクリックします。

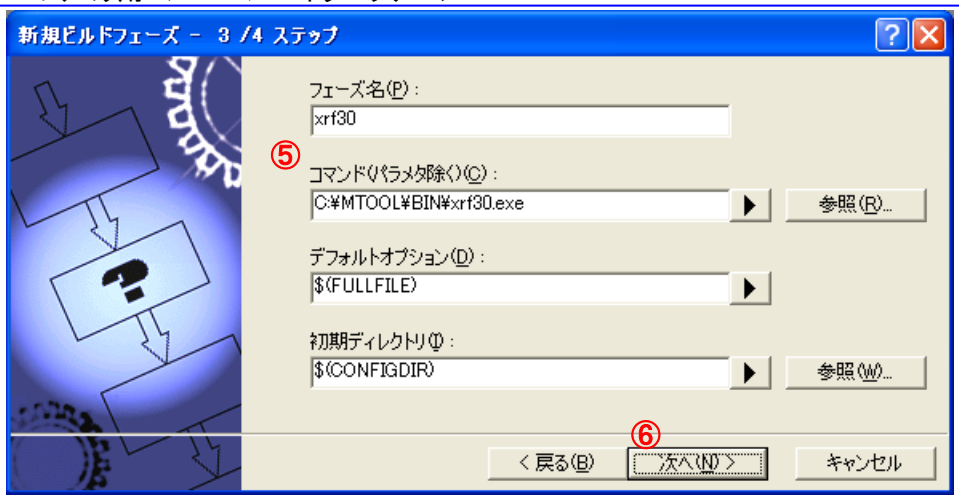

図 8 新規ビルドフェーズ-3/4 ステップ

- [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。
- [3/4 ステップ]次へボタンをクリックします。

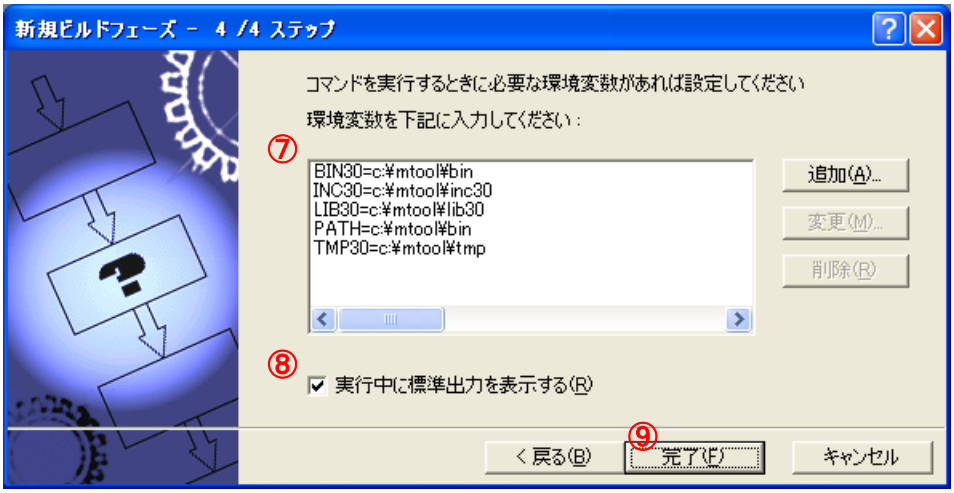

図 9 新規ビルドフェーズ-4/4 ステップ

- [4/4 ステップ]環境変数を指定します。
- [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。
- [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。

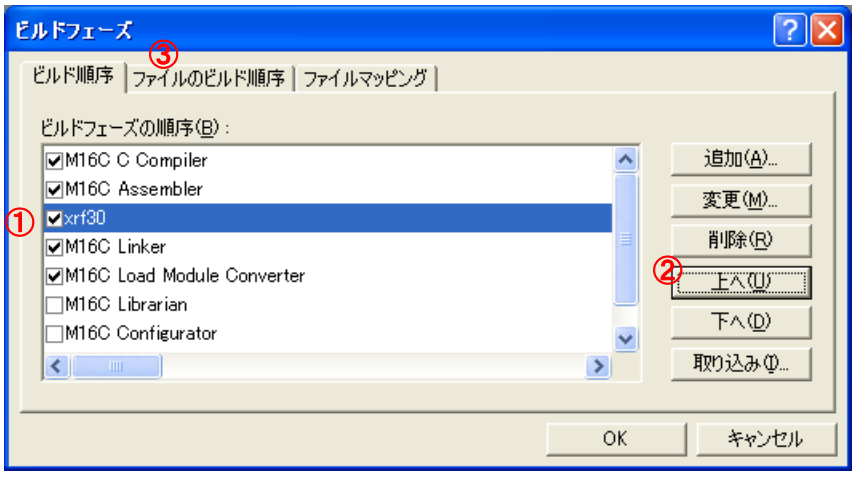

図 10 ビルドフェーズダイアログボックス

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

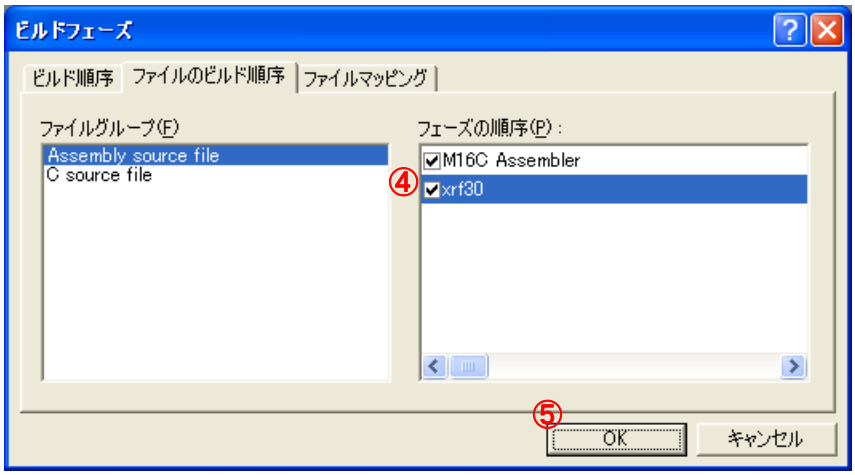

図 11 ビルドフェーズダイアログボックス

- xrf30 をチェックします。
- 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。

7.3.6. 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、アルファベット順となります。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。

本機能は、High-performance Embedded Workshop4 がバンドルされたコンパイラ製品からの対応となります。

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。

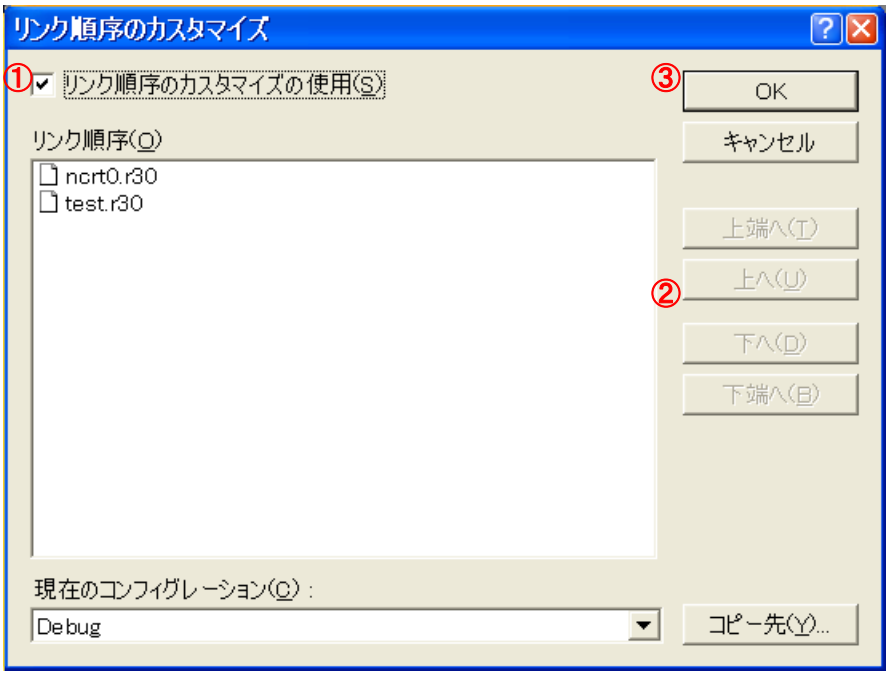

図 12 リンク順序のカスタマイズダイアログボックス

- リンク順序のカスタマイズの使用をチェックします。
- 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。
- OK ボタンをクリックします。

7.3.7. 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、アルファベット順となります。そ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、「C.3.6 リンク順序」の項の設定を行ってください。
「C.3.6 リンク順序」に対応していないバージョンのコンパイラは、以下の手順で設定してください。

- 1. メニュー「ビルド」→「Renesas XXX Standard Toolchain...」をクリックします。("Renesas XXX Standard Toolchain" ダイアログボックスが開きます)
- 2. 「link」タブをクリックします。
- 3. [Category:]~、「Input」を選択します。
- 4. 「Show entries for:」から、「Relocatable files」を選択します。
- 5. 「Add」ボタンをクリックします。(「Add Relocatable files」ダイアログボックスが開きます)
- 6. 「Relative to:」から、「Configuration directory」を選択します。
- 7. 「File path:」にスタートアッププログラム名(例:NC30WA の場合、ncrt0.r30)を入力します。
- 8. 「Add Relocatable files」ダイアログボックスの[OK]ボタンをクリックします。
- 9. 「Renesas XXX Standard Toolchain」ダイアログの[OK]ボタンをクリックします。

M16Cシリーズ,R8Cファミリ用C/C++コンパイラパッケージ
NC30 V.6.00 Release 00
リリースノート

発行 ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668

編集 株式会社 ルネサスソリューションズ ツールビジネス本部 ツール開発第一部