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。

RX ファミリ用 C/C++ コンパイラパッケージ

アプリケーションノート：＜サンプルプロジェクトＲＸ移行ガイド＞M16C 編

本ドキュメントでは、M16C で作成したサンプルプロジェクトを RX へ移行する手順についての説明を行います。

目次

本編では、シミュレータデバッガで動作確認ができる M16C サンプルプロジェクトを、RX へ移行する手順をガイドします。

1. M16C サンプルプロジェクト概要

M16C サンプルプロジェクト'M16C_Sample'は、大きく分けて初期化などを行う前後処理と、主要な処理を行うメイン処理に分かれています。本編では主要な処理を行うメイン処理を RX プロジェクトに移行し、動作確認をする手順を示します。メイン処理を構成するファイルを以下に示します。

表 1-1 メイン処理ファイル一覧

No	処理内容	ファイル名	参照
1	char 型の汎整数拡張仕様に依存	M16C_extended_integer.c	2.7.1
2	int 型のサイズに依存	M16C_int_size.c	2.7.2
3	構造体メンバ配置に依存	M16C_struct_member.c	2.7.3
4	double 型のサイズに依存	M16C_double_size.c	2.4.1
5	pragma BITADDRESS に依存	M16C_pragma_BITADDRESS.c	2.4.2
6	pragma ROM に依存	M16C_pragma_ROM.c	2.4.3
7	pragma PARAMETER に依存	M16C_pragma_PARAMETER.c	2.4.4
8	inline キーワードに依存	M16C_inline_keyword1.c M16C_inline_keyword2.c	2.4.5
9	main 関数	M16_Sample_main.c	ー

2. M16C サンプルプロジェクト RX 移行手順

2.1 RX プロジェクトの作成

M16C サンプルプロジェクトの移行先となる RX の新規プロジェクトワークスペースを作成します。
プロジェクトジェネレータ（HEW の[ファイル → 新規ワークスペース]メニューで起動）で、以降手順に従ったサンプルプロジェクトの生成手順について説明します。

(1) 新規ワークスペースの作成

プロジェクトタイプ“Application”を選択。

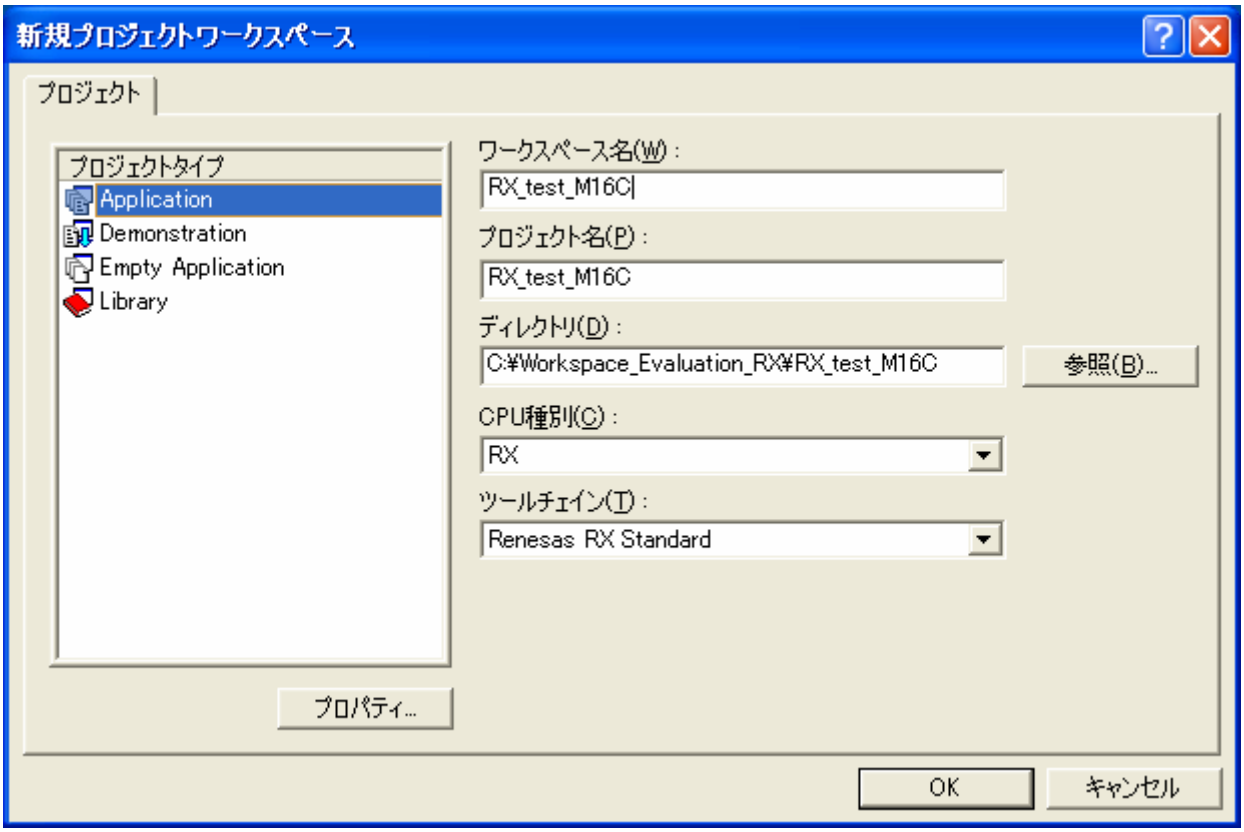


図 1-1

(2) CPU の選択

デフォルトのままで次に進む

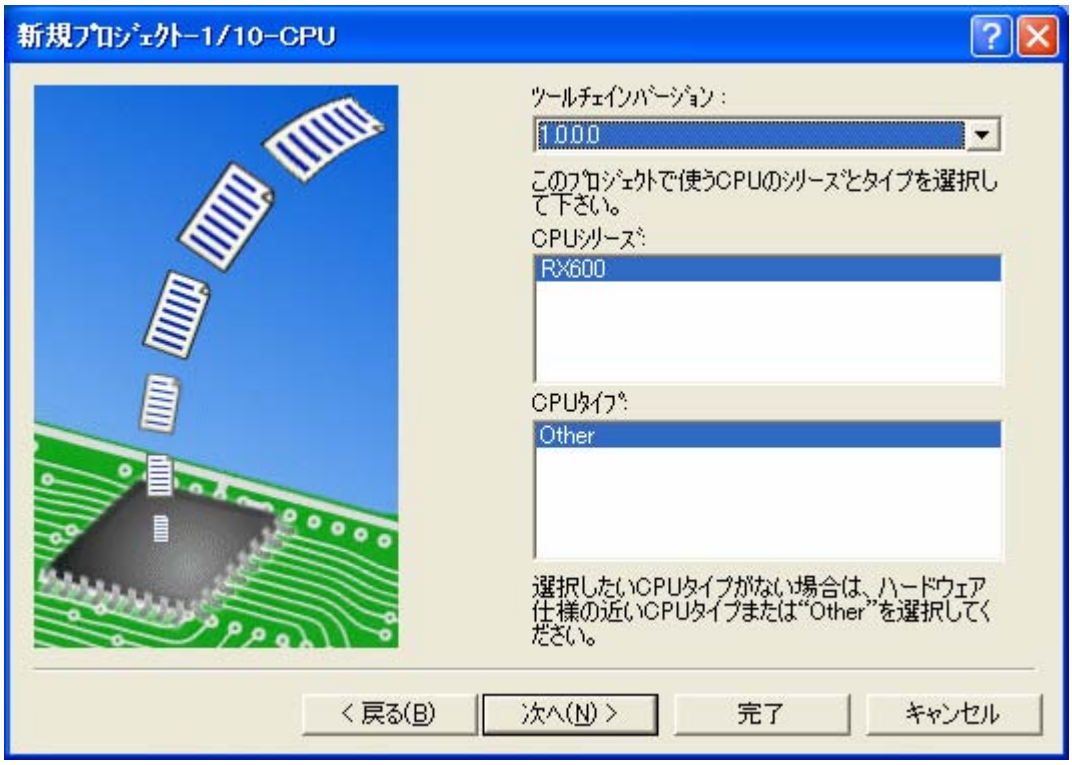


図 1-2

(3) オプション設定

デフォルトのままで次に進む

図 1-3

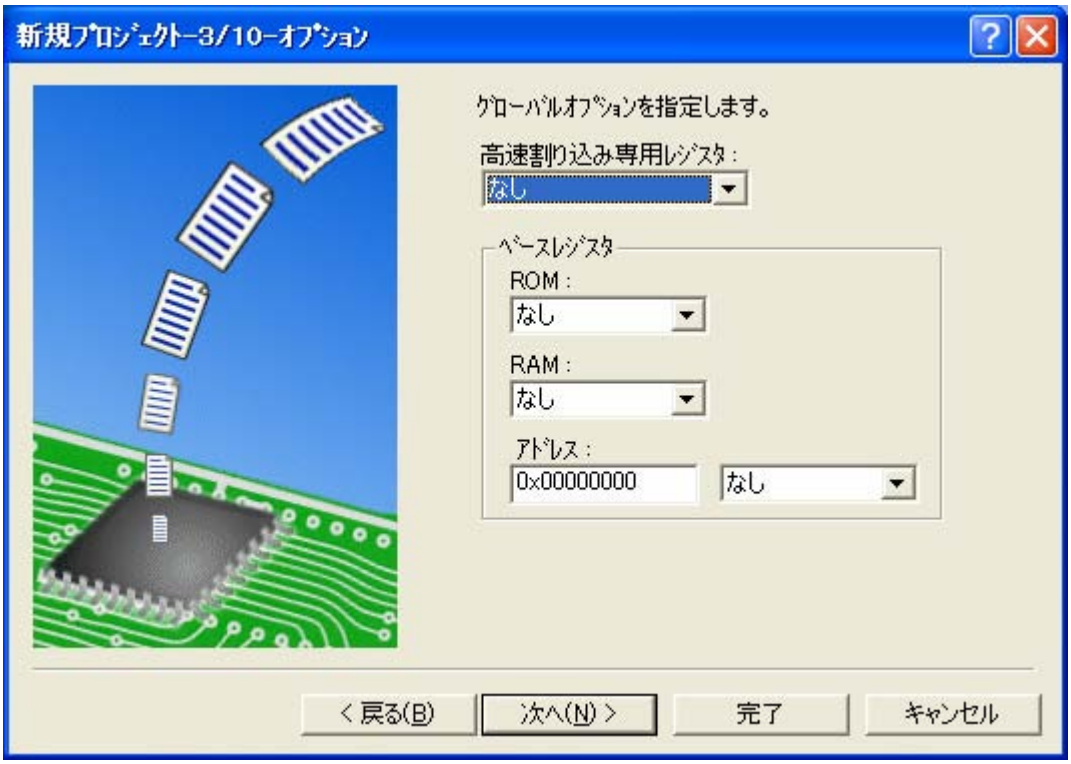


図 1-4

(4) 生成ファイルの設定

“I/O ライブラリ使用”をチェック
“I/O ストリーム数”に“20”を指定

図 1-5

(5) 標準ライブラリの設定

デフォルトのままで次に進む

図 1-6

(6) スタック領域の設定

デフォルトのままで次に進む

図 1-7

(7) ベクタの設定

デフォルトのままで次に進む

図 1-8

(8)デバッガの設定

“RX600 Simulator”をチェック

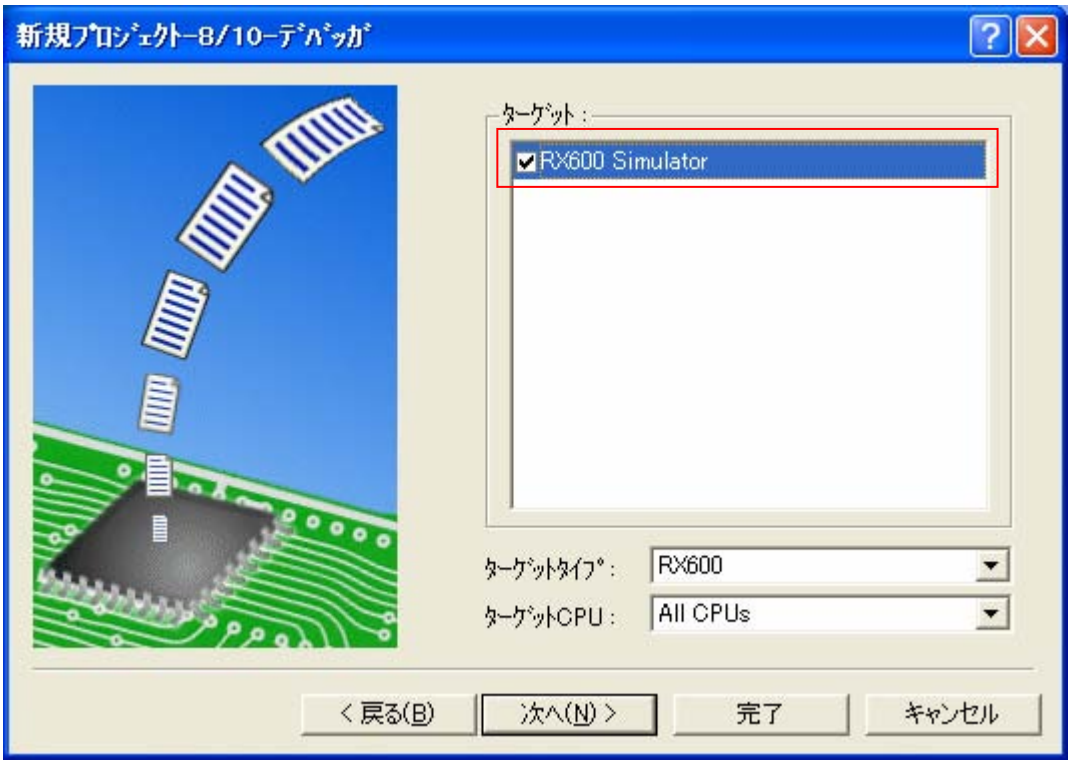


図 1-9

(9)デバッガオプションの設定

“初期セッション”をチェック

図 1-10

(10)生成ファイル名の確認

完了を選択

図 1-11

(11)シミュレータの設定

“OK”を選択

図 1-12

2.2 メイン処理ソースファイル移行

‘1.M16Cサンプルプロジェクト概要’で説明したM16Cサンプルプロジェクトのメイン処理を構成するファイルを、作成したRXプロジェクトにコピー・登録します。

(1)M16C サンプルプロジェクトフォルダからファイルをコピー

‘1.M16Cサンプルプロジェクト概要’で説明した 10 のファイルをRXプロジェクトへコピーします。

[M16C_Smapleプロジェクト]

図 1-13

(2)コピーしたファイルをプロジェクトに登録する

コピーしたファイルを、作成した RX プロジェクトに登録します。
HEW の[プロジェクト →ファイルの追加]を選択すると表示されるダイアログで、次のように選択します。

図 1-14

(3)不要ファイルの登録削除

プロジェクトジェネレータが生成した、main 関数を持つファイル’RX_test_M16C.c’は不要なので削除します。
(main 関数ファイルは M16C サンプルジェクトからコピーするため)
HEW の[プロジェクト →ファイルの削除]を選択すると表示されるダイアログで次のように選択します。

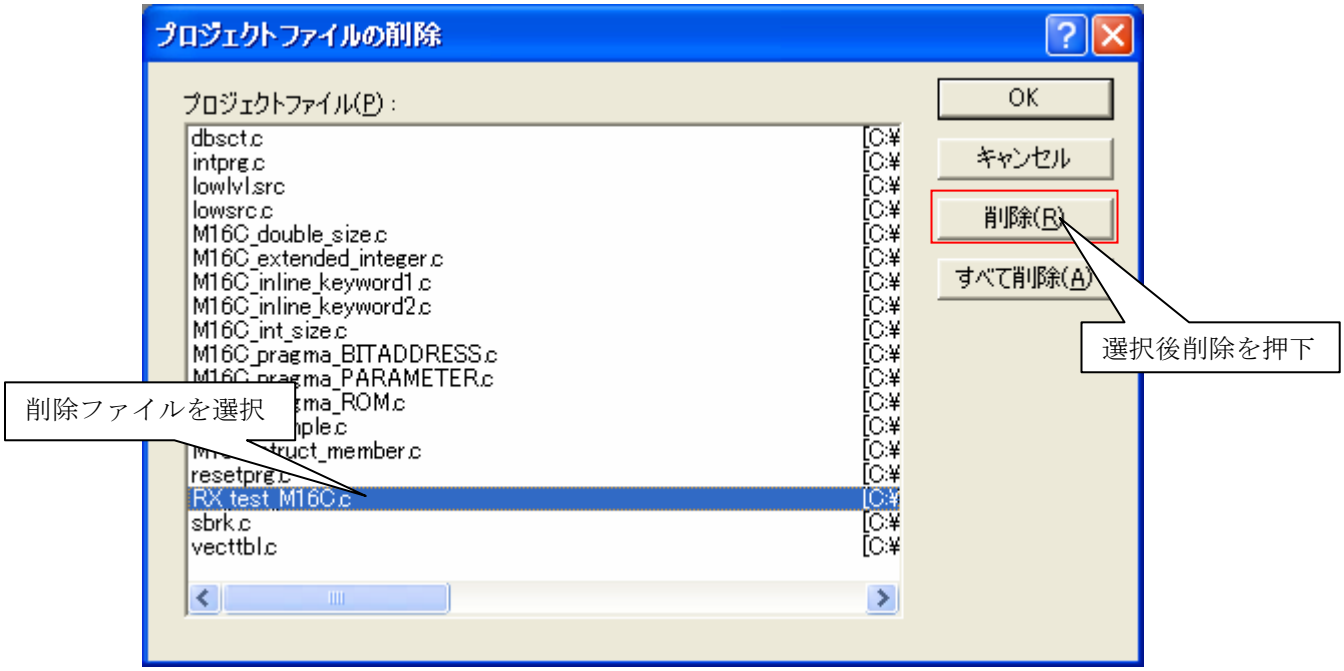


図 1-15

2.3 ビルド&M16C 互換性チェック

メイン処理ファイルをコピー・登録した RX プロジェクトをビルドします。HEW ではビルド時に、M16C 互換性チェック機能を有効にすることで、互換性に影響のあるオプション指定・ソース記述をチェックできます。本編では M16C 互換性チェック機能を有効にしてビルドし、表示される互換性チェックメッセージを確認します。

(1)M16C 互換性チェック機能設定

HEW の[ビルド →RX Standard Toolchain]を選択すると表示されるダイアログで次のように選択します。

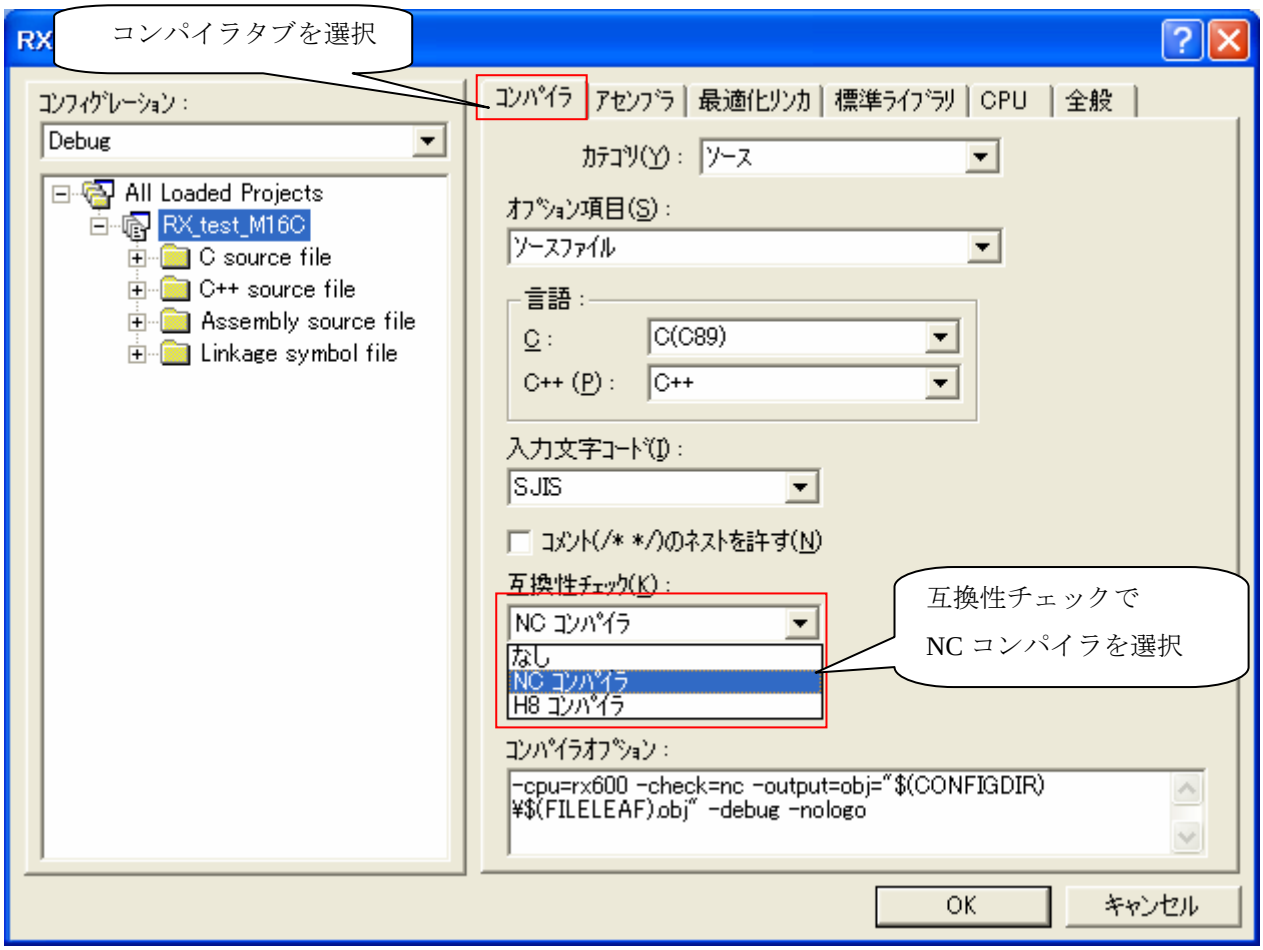


図 1-16

(2)ビルド

HEW の[ビルド →ビルド]を選択するとビルドを開始し、アウトプットウィンドウにビルド時のメッセージが表示されます。

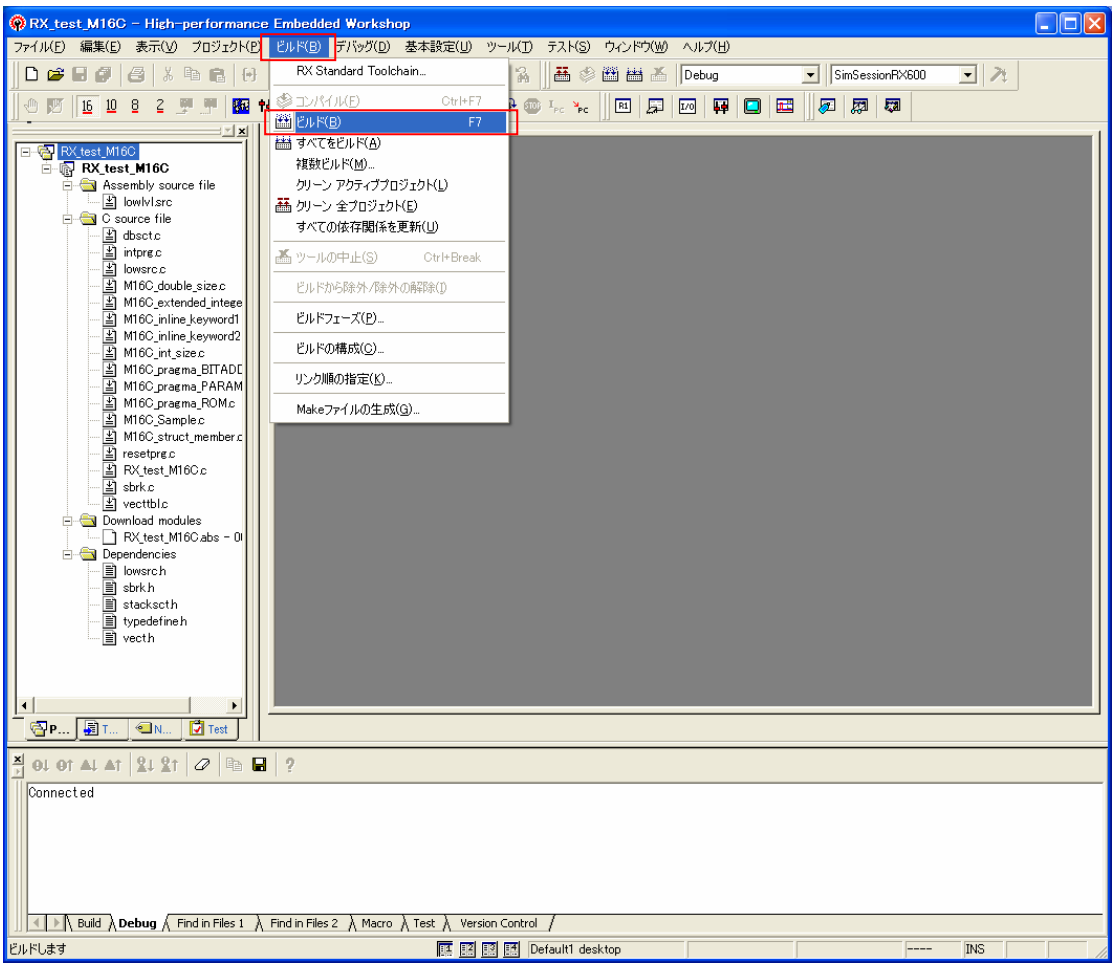


図 1-17

図 1-18

(3)互換性チェックメッセージ確認

アウトプットウィンドウに出力されたメッセージの中に「C1801」のウォーニングが出力されています。これが M16C と互換性に問題がある箇所に対するメッセージです。

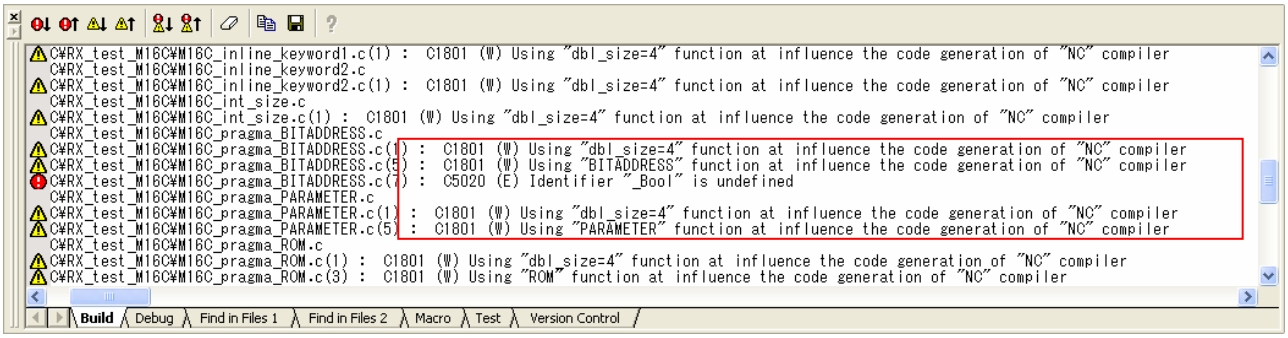


図 1-19

2.4 互換性チェック指摘対応

‘2.3ビルド＆M16C互換性チェック’で指摘のあった、互換性に影響するC1801 メッセージの内容を確認、対処方法を示します。対象となるメッセージは以下のとおりです。

表 1-2 C1801 **メッセージ一覧**

No	**互換性に影響するメッセージ**	**参照**
1	Using "dbl_size=4" function at influence the code generation of "NC" compiler	2.4.1
2	Using "BITADDRESS" function at influence the code generation of "NC" compiler	2.4.2
3	Using "PARAMETER" function at influence the code generation of "NC" compiler	2.4.3
4	Using "ROM" function at influence the code generation of "NC" compiler	2.4.4
5	Using "inline" function at influence the code generation of "NC" compiler	2.4.5

2.4.1 double 型変数のサイズ指定

メッセージ’ Using "dbl_size=4" function at influence the code generation of "NC" compiler’は、”dbl_size=4”オプション指定が互換性に問題があることを指摘しています。M16C ファミリコンパイラでは double 型のサイズは 8byte です。対して RX ファミリコンパイラではデフォルトでは dbl_size=4 が指定されているため、サンプルプログラムの”M16C_double_size.c”で double 型のサイズが 8byte あることを前提として記述してあるため” dbl_size=4”オプションが指定されている場合、M16C とは異なる動作結果となります。

【サンプルプログラム” M16C_double_size.c”】

ソースコード

```
double d1 = 1E30;
double d2 = 1E20;

void double_size(void)
{
    d1 = d1 * d1;
    d2 = d2 * d2;

    printf("(5) double type size : ");

    if (d1 > d2) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

double 型のサイズが 8byte であることを前提に作成したプログラムを RX に移行するには、”dbl_size=8”オプションを指定します。オプション指定の詳細は、コンパイラユーザーズマニュアルを参照してください。また、作成した RX プロジェクトのオプション指定を変更してください。

2.4.2 #pragma BITADDRESS 指定

メッセージ’ Using "BITADDRESS" function at influence the code generation of "NC" compiler’は、ソース中の#pragma BITADDRESS に互換性の問題があることを指摘しています。M16C ファミリコンパイラは#pragma BITADDRESS をサポートしていますが、RX ファミリコンパイラはサポートしていません。

サンプルプログラムの”pragma_BITADDRESS.c”は、#pragma BITADDRESS が記述してあるため RX で意図どおりに動作しません。#pragma BITADDRESS を利用したプログラムを RX に移行する方法として、ビットアクセス処理を構造体のビットフィールドに置き換える方法があります。以下に#pragma BITADDRESS を使ったソースコードと、それを RX に移行したソースコードを示します。
(但し M16C と RX ではメモリマップが異なるため指定する絶対アドレスは適宜変更しています。例としてご活用ください)

【サンプルプログラム” pragma_BITADDRESS.c ”】

M16Cソースコード

```
#pragma ADDRESS bit_data 410H

int bit_data;

#pragma BITADDRESS bit 0,410H

_Bool bit;

void pragma_bitaddress(void)
{
    printf("(6) pragma BITADDRESS : ");
    bit_data = 1;

    if (bit == 1) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

RXソースコード

```
#pragma address bit_data 0x2000

int bit_data;

struct bit_address {
    unsigned char b0:1;
    unsigned char b1:1;
    unsigned char b2:1;
    unsigned char b3:1;
    unsigned char b4:1;
    unsigned char b5:1;
    unsigned char b6:1;
    unsigned char b7:1;
};

#define bit (((struct bit_address*)0x2000)->b0)

void pragma_bitaddress(void)
{
    printf("(6) pragma BITADDRESS : ");
    bit_data = 1;

    if (bit == 1) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

【補足】

“pragma_BITADDRESS.c”は#pragma BITADDRESS の他に、ANSI 規格の C89 で問題のある”_Bool”型を使用しています。RX で”_Bool”型を使用するには、ANSI 規格の C99 を有効にする”lang=c99”オプションを指定します。オプション指定の詳細は、コンパイラユーザーズマニュアルを参照してください。

2.4.3 #pragma PARAMETER 指定

メッセージ’ Using "PARAMETER" function at influence the code generation of "NC" compiler’は、ソース中の#pragma PARAMETER に互換性の問題があることを指摘しています。M16C ファミリコンパイラは#pragma PARAMETER をサポートしていますが、RX ファミリコンパイラはサポートしていません。

サンプルプログラムの”M16C_pragma_PARAMETER.c”は、#pragma PARAMETER が記述してあるため RX で意図どおり動作しません。#pragma PARAMETER を利用したプログラムを RX に移行するには、#pragma PARAMETER で指定されたアセンブラ関数の引数インタフェースを C/C++の生成規則に従い変更する必要があります。

関数インタフェースの詳細は、コンパイラユーザーズマニュアルを参照してください。

#pragma PARAMETER は RX コンパイラで無視されるため、”M16C_pragma_PARAMETER.c”は変更の必要はありません。

【サンプルアセンブラソースコード】

M16Cソースコード ”M16C_pragma_PARAMETER_asm.a30 ”

アセンブラソースコード

```
        . FB     0
        .glb     _asm_func
        .section  program
_asm_func:
        enter    #02H
        mov.w    R1,-2[FB]
        add.w    -2[FB],R0
        exitd
        .end
```

#pragma PARAMETERにより‘ i ’はR1、’ j ’はR0に格納して渡ることが前提のコード

RXソースコード ”RX_pragma_PARAMETER_asm.src ”

アセンブラソースコード

```
        .GLB      _asm_func
        .SECTION   P,CODE
_asm_func:
        ADD       R2,R1
        RTS
        .END
```

‘ i ’はR1、’ j ’はR2に格納して渡ることを前提にコードを記述する

M16C サンプルプロジェクトのアセンブラソースコード”M16C_pragma_PARAMETER_asm.a30”は、そのままでは RX プロジェクトに移行できないため、新規にアセンブラソースファイルを作成・RX 用のアセンブラコードを記述し、登録する必要があります。M16C サンプルプロジェクト内に、RX 用のアセンブラコードを記述したファイル”RX_pragma_PARAMETER_asm.src”がありますので、それを RX プロジェクトに登録してください。

2.4.4 #pragma ROM 指定

メッセージ’ Using "ROM" function at influence the code generation of "NC" compiler’は、#pragma ROM のソース記述に互換性の問題があることを指摘しています。M16C ファミリコンパイラは#pragma ROM をサポートしていますが、RX ファミリコンパイラは、デフォルトで-lang=c が指定されているためこのままでは inline キーワードを認識できません。

サンプルプログラム”M16C_pragma_ROM.c”に記述のある#pragma ROM は RX で意図どおりに機能しません。RX に移行するには#pragma ROM 指定している変数宣言に const キーワードを指定してください。

【サンプルプログラム”M16C_pragma_ROM.c ”】

M16Cソースコード

```
#pragma  ROM  rrr
unsigned short rrr = 100;

void pragma_rom(void)
{
    printf("(7) pragma ROM : ");

    if (rrr == 100) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

アセンブラソースコード

```
            .SECTION    rom_FE,ROMDATA,align
            .glb        _rrr
_rrr:
            .word       0064H
```

RXソースコード

```
const unsigned short rrr = 100;

void pragma_rom(void)
{
    printf("(7) pragma ROM : ");

    if (rrr == 100) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

アセンブラソースコード

```
            .SECTION    C_2,ROMDATA,ALIGN=2
            .glb        _rrr
_rrr:                               ; static: rrr
            .word        0064H
```

2.4.5 inline キーワード指定

メッセージ’ Using "inline" function at influence the code generation of "NC" compiler’は、inline キーワードのソース記述に互換性の問題があることを指摘しています。M16C ファミリコンパイラは inline キーワードをサポートしていますが、RX ファミリコンパイラはサポートしていません。

サンプルプログラムの”M16C_inline_keyword2.c”は inline キーワードが記述してあるため、そのまま RX の ANSI 規格 C89 でビルドした場合、コンパイルエラーとなります。inline キーワードを利用したプログラムを RX に移行するには以下の 2 つの方法があります。

・ANSI 規格の C89 でビルドを実行する場合は、inline 記述を#pragma inline へ変更する。
・ANSI 規格の C99 でビルドを実行する。

以下に#pragma inline を使った記述例を示します。RX サンプルプロジェクトも以下ように修正してください。

【サンプルプログラム” M16C_inline_keyword2.c ”】

M16Cソースコード

```
inline int inline_func(void)
{
    return 4;
}

int get_value()
{
    return inline_func();
}
```

RXソースコード

```
#pragma inline inline_func
int inline_func(void)
{
    return 4;
}

int get_value()
{
    return inline_func();
}
```

【補足】

ソースを修正せずに inline キーワードを使用するには ANSI 規格 C99 を有効にする”lang=c99” オプションを指定します。オプション指定の詳細は、コンパイラユーザーズマニュアルを参照してください。

但し、C99 の inline キーワードは以下のような特性があるため注意が必要です。

・”noinline”オプション指定や”inline”オプションの条件不成立など、必ず inline 展開を行うとは限らない。
・inline キーワード指定の関数は内部リンケージであり削除対象である。

サンプルプログラム”M16C_inline_keyword2.c”は上記の 2 つの条件にあてはまるため”lang=c99”オプションでコンパイルした場合、inline 展開を行わないにもかかわらず、inline_func 関数の定義を削除します。結果として inline_func 関数が未定義でリンクエラーとなります。これを回避するには以下のいずれかの方法を適用してください。

・inline キーワードを#pragma inline に置き換え、無条件に inline 展開を行う。
・inline 関数に extern 指定を追加し、外部リンケージとする。

【例 inline 関数を外部リンケージに変更する】

RXソースコード

```
extern inline int inline_func(void)     // inline_func関数が外部リンケージとなり削除されない
{
    return 4;
}

int get_value()
{
    return inline_func();
}
```

2.5 再ビルド

‘ 2.4互換性チェック指摘対応 ’ で互換性に問題があったオプション指定やソース記述を変更後、プロジェクトを再ビルドします。ビルド方法については ’ 2.3(3)ビルド ’ を参照してください。ビルドが成功すると以下のダイアログが表示されるので ’ はい ’ を選択し、ロードモジュールをダウンロードしてください。

図 1-20

2.6 シミュレータ実行

再ビルドしたロードモジュールをシミュレータで実行します。

(1) I/O シミュレーション設定

プログラムは実行結果を標準出力に表示します。標準出力を表示するには’I/O シミュレーション’ウィンドウを有効にする必要があります。HEW の[表示 →CPU →I/O シミュレーション]を選択すると’I/O シミュレーション’ウィンドウが表示されます。

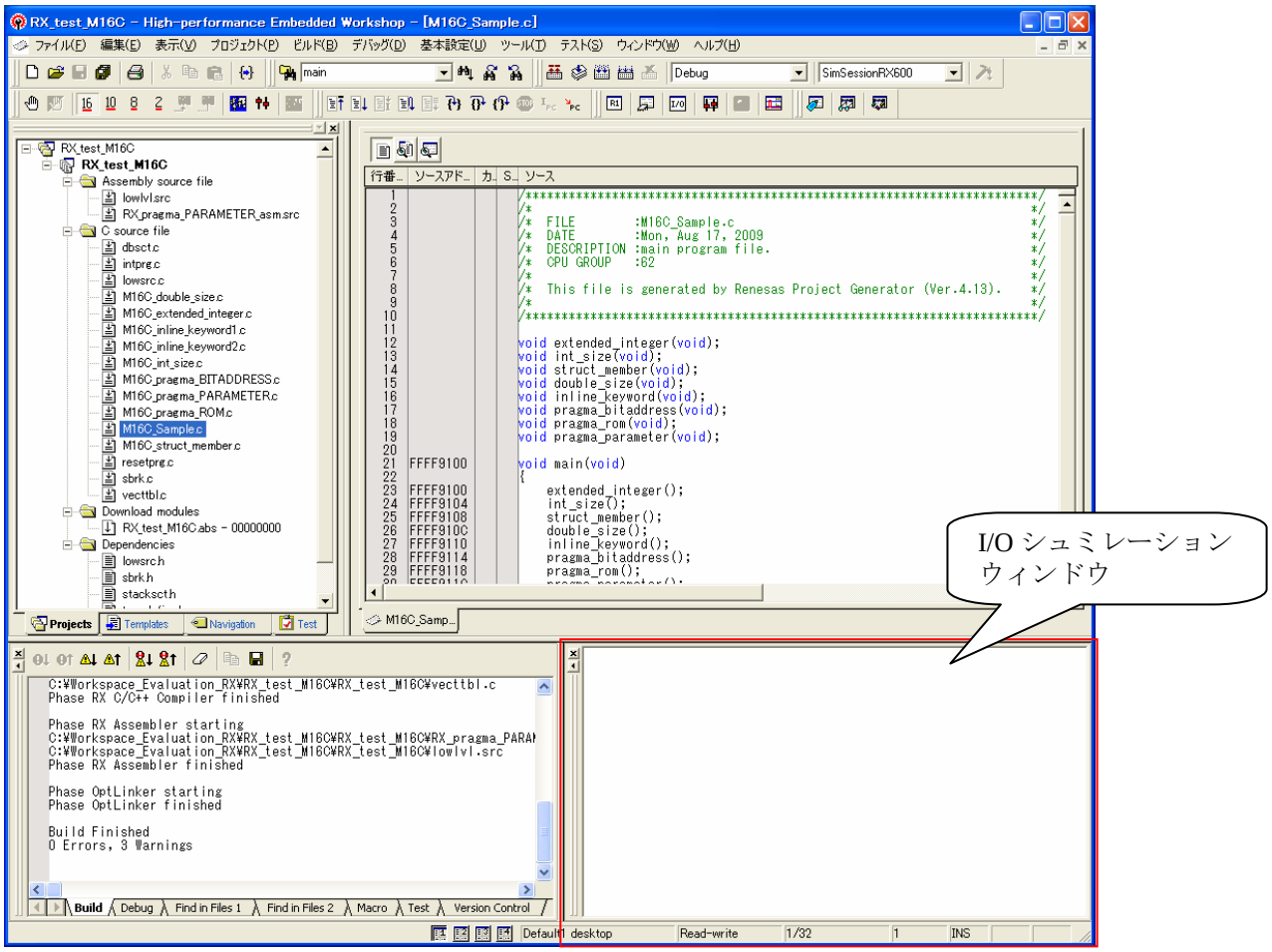
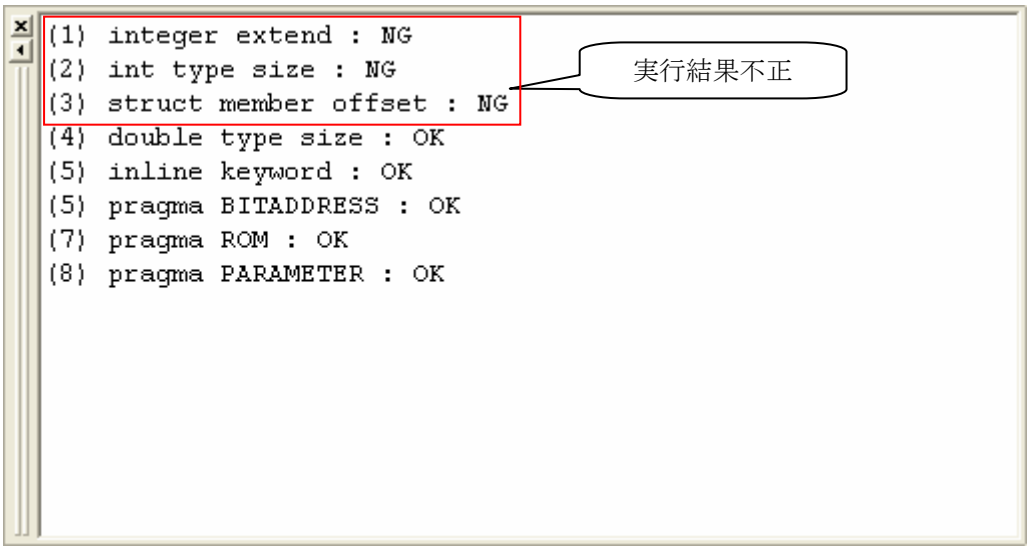


図 1-21

(2)シミュレータ実行

HEW の[デバッグ →リセット後実行]を選択するとプログラムがシミュレータにより実行され、'I/O シミュレーション'ウィンドウにプログラムの標準出力が表示されます。表示結果を見ると (1)(2)(3)の結果が不正であることがわかります。



2.7 実行結果不正対応

移行したサンプルプロジェクトを実行、発生した結果不正の内容を確認し、互換性に問題がある箇所の対処方法を示します。結果不正は以下のとおりです。

No	結果不正	互換性問題	参照
1	integer extend	汎整数拡張仕様	2.7.1
2	int type size	int型のサイズ	2.7.2
3	struct member offset	構造体メンバの配置	2.7.3

2.7.1 汎整数拡張仕様

ANSI 規格では char 型データ(signed char, unsigned char を含む)を評価するとき、必ず int 型に拡張します。RX の仕様は ANSI 規格に準拠していますが、M16C は ROM 効率を改善するため char 型データを評価するとき int 型に拡張していません。このため M16C で char 型データの演算中にオーバフローするコードを、RX に移行した場合 int 型の拡張によりオーバフローが発生せず、結果が異なる場合があります。M16C において char 型データ演算中にオーバフローすることに依存したコードを、RX に移行する場合確認が必要になります。

【サンプルプログラム” M16C_extended_integer.c ”】

M16Cソースコード

```
void extended_integer(void)
{
    char  c1;
    char  c2 = 200;
    char  c3 = 200;

    c1 = (c2+c3)/2;

    printf("(1) integer extend : ");
    if (c1 != 200) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

M16C は c2+c3 で char 型のサイズをオーバフロー、 加算結果が 400 にならない

RX は c2+c3 は int 型に拡張後加算オーバフローせず加算結果が 400

RXソースコード

```
void extended_integer(void)
{
    char  c1;
    char  c2 = 200;
    char  c3 = 200;

    c1 = ((char)(c2+c3))/2;

    printf("(1) integer ext
    if (c1 != 200) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

M16C に合わせるには加算結果に char 型のキャストを挿入する

【補足】

M16C は char 型データを評価するとき int 型に拡張する以下の 2 点のオプションを用意しています。以下のオプションのいずれかが指定されている場合、本項で説明した汎整数拡張仕様差は発生しません。

- -fansi
- -fextend_to_int

2.7.2 int 型のサイズ

M16C ファミリコンパイラでは int 型のサイズは 2byte です。対して RX ファミリコンパイラでは int 型のサイズは 4byte です。サンプルプログラムの”M16C_int_size.c”で int 型のサイズが 2byte あることを前提として記述してあるため M16C とは異なった動作結果となります。

【サンプルプログラム” M16C_int_size.c”】

ソースコード

```
typedef union{
    long data;
    struct {
        int dataH;
        int dataL;
    } s;
} UN;

void int_size(void)
{
    UN u;
    u.data = 0x7f6f5f4f;

    printf("(2) int type size : ");

    if (u.s.dataH == 0x5f4f && u.s.dataL == 0x7f6f) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

int 型のサイズが 2byte であることを前提に作成したプログラムを RX に移行するには、” **int_to_short**”オプションを指定します。オプション指定の詳細は、コンパイラユーザーズマニュアルを参照してください。また、作成した RX プロジェクトのオプション指定を変更してください。

2.7.3 構造体メンバの配置

構造体において(共用体・クラスも同様です)、1 バイト、2 バイト、4 バイトのメンバが混在している場合、各メンバ配置はそれぞれのアライメント数に従うためにメンバとメンバの間に空き領域が発生する場合があります。M16C ファミリコンパイラは、構造体メンバの配置をアラインメント数 1 で配置します。対して RX ファミリコンパイラは、構造体メンバの最大アラインメント数に従い配置します。そのため M16C の構造体配置を前提に作成したプログラムを、RX に移行すると正しく動作しない場合があります。

サンプルプログラムの”M16C_struct_member.c”は構造体のアラインメント数 1 を前提として記述してあるため、そのままでは RX で正常に機能しません。サンプルプログラムを RX へ移行するには以下の 2 つの方法があります。

・”pack” オプションを指定する。
・構造体に対して#pragma pack を指定する。

指定方法の詳細については、
「RX ファミリ C/C++ コンパイラパッケージ アプリケーションノート：
＜RX 移行ガイド＞M16C 編 2.4 構造体のメンバ配置」を参照してください。

【サンプルプログラム” M16C_struct_member.c”】

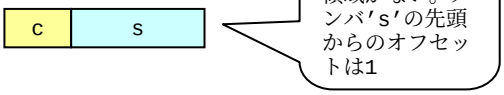
ソースコード

```
#include <stdio.h>
#include <stddef.h>

void struct_member(void)
{
    struct s {
        char c;
        short s;
    } ss;

    printf("(3) struct member offset : ");
    if (offsetof(struct s, s) == 1) {
        printf("OK\n");
    } else {
        printf("NG\n");
    }
}
```

M16Cのメンバ配置



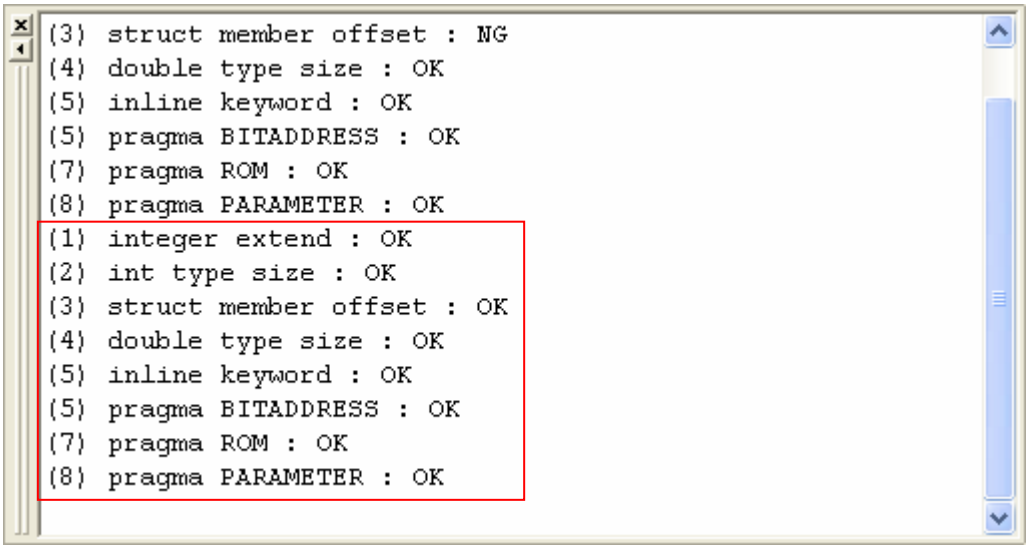
アライメント数が1のため空き領域がない。メンバ’s’の先頭からのオフセットは1

RXのメンバ配置

アライメント数に合わせる空き領域ができる。メンバ’s’の先頭からのオフセットは2

作成した RX プロジェクトの”pack” オプションを指定してください。

オプションとコードを変更後、再ビルド(’2.3(3)ビルド’を参照)、シミュレータ実行(2.6 シミュレータ実行を参照)を実施すると以下のような実行結果となり、RX プロジェクトへの移行が完了しました。



ホームページとサポート窓口<website and support>

ルネサステクノロジホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/inquiry
csc@renesas.com

改訂記録<revision history,rh>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2009.10.1	—	初版発行