

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のですが、誤りが無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、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。

M16Cファミリ用Cコンパイラ

アプリケーションノート

M3T-NC308WA/M3T-NC30WA

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズム その他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ (<http://www.renesas.com>) などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものです。万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任を負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気づきの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。

はじめに

本アプリケーションノートでは、ルネサステクノロジ M16C ファミリの機能・性能を活かした応用プログラムを「NC308WA V.5.20 Release 02」、「NC30WA V.5.30 Release 02」を用いて効果的に作成する方法を説明します。

C コンパイラの詳細な仕様については、『NC308 ユーザーズマニュアル』、『NC30 ユーザーズマニュアル』を参照してください。

関連マニュアルは以下のとおりです。

- ・ M16C ファミリー 各マイコン別ハードウェアマニュアル
- ・ High-performance Embedded Workshop3 ユーザーズマニュアル
- ・ NC308 ユーザーズマニュアル
- ・ NC30 ユーザーズマニュアル
- ・ M16C/80 シリーズ プログラム作成の手引き < C 言語編 >
- ・ M16C/60 シリーズ、M16C/20 シリーズ プログラム作成の手引き < C 言語編 >

〈本アプリケーションノートで使用する記号などの意味〉

- [] : 省略できることを示します。
- (RET) : リターンキーの入力を示します。
- : 1 つ以上の空白またはタブコードを示します。
- abc** : 太字の部分はユーザがキー入力する部分を示します。
- : この記号で囲まれた内容を指定することを示します。
- ... : 直前の項目を 1 回以上指定することを示します。
- H : 整数定数の末尾に " H " が付いているのは 16 進数です。
- 0x : 整数定数の先頭に " 0x " が付いているのは 16 進数です。
- 0b : 整数定数の先頭に " 0b " が付いているのは 2 進数です。

UNIX は、X/Open カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。

MS-DOS は米国マイクロソフト社により管理されているオペレーティングシステムの名称です。

Microsoft® WindowsNT® operating system, Microsoft®, Windows®98 and Windows 2000 operating system, Microsoft® WindowsMe® operating system, Microsoft® WindowsXp® operating system は、米国 Microsoft Corporation の米国及びその他の国における登録商標です。

IBM PC は、米国 International Business Machines Corporation の登録商標です。

Linux は、Linus Torvalds 氏の米国およびその他の国における登録商標あるいは商標です。

Turbolinux の名称およびロゴは、Turbolinux, Inc. の登録商標です。

Solaris は米国 Sun Microsystems, Inc. の登録商標です。

目次

1. 概説	1-1
1.1 概要.....	1-3
1.1.1 概略仕様.....	1-3
1.2 特長.....	1-4
1.3 インストール方法	1-5
1.4 起動方法.....	1-5
1.4.1 コンパイラの起動方法.....	1-5
1.4.2 統合化環境からの起動方法.....	1-6
1.5 プログラム開発手順.....	1-7
2. プログラム作成の手順	2-1
2.1プロジェクトの構築	2-3
2.2スタートアッププログラム.....	2-13
2.2.1スタートアッププログラムの役割.....	2-13
2.2.2スタートアッププログラムの設定.....	2-14
3. コンパイラ	3-1
3.1 割り込み関数	3-3
3.1.1 割り込み処理関数の記述.....	3-3
3.1.2高速割り込み処理関数の記述.....	3-6
3.1.3 ソフトウェア割り込み（INT命令）処理関数の記述.....	3-8
3.1.4 割り込み処理関数を登録する	3-10
3.1.5 割り込み処理関数の記述例	3-11
3.2 アセンブルマクロ	3-13
3.2.1アセンブルマクロ関数で記述できるアセンブリ言語命令	3-13
3.2.2アセンブルマクロ関数"DADD_B"を使用した10進加算.....	3-15
3.2.3アセンブルマクロ関数 " SMOVE_B " を使用したストリング転送	3-16
3.2.4アセンブルマクロ関数 " RMPA_W " を使用した積和演算	3-17
3.2.5 #PRAGMA __ASMMACRO.....	3-18
3.3 コード効率向上のためのPRAGMA,オプション	3-19
3.3.1 #PRAGMA SBDATA.....	3-19
3.3.2 #PRAGMA SB16DATA	3-19
3.3.3 #PRAGMA BIT.....	3-20
3.3.4 #PRAGMA SPECIAL	3-20
3.3.5 -FJSRW.....	3-21
3.3.6 -OR	3-22
3.3.7 -FNO_ALIGN.....	3-23
3.3.8 -WNO_USED_FUNCTION.....	3-23
3.4 速度向上のためのPRAGMA、オプション	3-24
3.4.1 #PRAGMA STRUCT	3-24

3.4.2	-OSTACK_FRAME_ALIGN	3-27
3.4.3	-OS	3-29
3.4.4	-OLOOP_UNROLL[=回数]	3-29
3.4.5	-OFLOAT_TO_INLINE	3-30
3.4.6	-OSTATIC_TO_INLINE	3-31
3.5	ROM領域削減、速度向上のためのPRAGMA, オプション	3-32
3.5.1	-O[1-5]	3-32
3.5.2	-OSP_ADJUST	3-33
3.5.3	-FUSE_DIV	3-34
3.5.4	-WNO_USED_ARGUMENT	3-35
3.5.5	-FSMALL_ARRAY	3-35
3.5.6	-FDOUBLE_32	3-36
3.6	その他のPRAGMA, オプション	3-37
3.6.1	その他のPRAGMA	3-37
3.6.2	その他のオプション	3-39
3.7	セクション	3-43
3.7.1	NC308が管理するセクション	3-43
3.8	クロスソフト間の関連	3-45
3.8.1	アセンブリ言語プログラムとの関連	3-45
3.9	LONG LONG型	3-52
3.10	NEAR/FAR型	3-53
3.10.1	NEAR 領域とFAR 領域	3-53
3.10.2	NEAR / FAR 属性のデフォルト	3-53
3.10.3	関数のNEAR / FAR	3-53
3.10.4	変数のNEAR / FAR	3-53
3.10.5	ポインタのNEAR / FAR	3-55
3.10.6	ポインタのNEAR/FAR指定におけるNC308とNC30の相違点	3-57
3.10.7	FAR領域に配置された変数アドレスのNEARポインタへの代入	3-57
3.11	インライン展開	3-58
3.11.1	INLINE記憶クラスの概要	3-58
3.11.2	INLINE記憶クラスの宣言書式	3-58
3.11.3	INLINE記憶クラスの規定事項	3-60
4.	HIGH-PERFORMANCE EMBEDDED WORKSHOPの活用	4-1
4.1	HIGH-PERFORMANCE EMBEDDED WORKSHOPのオプション指定方法	4-3
4.1.1	Cコンパイラのオプション	4-4
4.1.2	アセンブラのオプション	4-14
4.1.3	リンケージエディタのオプション	4-20
4.1.4	ライブラリアンのオプション	4-27
4.1.5	ロードモジュールコンパータのオプション	4-31
4.1.6	コンフィグレーションのオプション	4-35
4.1.7	CPUオプション	4-38
4.2	ビルド編	4-39
4.2.1	メイクファイルの出力	4-39
4.2.2	メイクファイルの入力	4-40
4.2.3	カスタムプロジェクトタイプの作成	4-42
4.2.4	マルチCPU機能	4-46

4.2.5 ネットワーク機能.....	4-47
5. 効率の良いプログラミング技法.....	5-1
5.1 引数のレジスタ渡し.....	5-4
5.2 レジスタ変数の活用.....	5-5
5.3 M16C特有の命令の活用.....	5-7
5.4 キャリーフラグによるビット演算分岐.....	5-8
5.5 ループ内固定式のループ外への移動.....	5-9
5.6 SBDATA宣言 & SPECIALページ関数宣言ユーティリティ.....	5-10
5.7 ELSE IF のSWITCH化.....	5-11
5.8 ループカウンタの条件判定.....	5-12
5.9 RESTRICT.....	5-12
5.10 _BOOL の活用.....	5-13
5.11 AUTO変数を明示的に初期化.....	5-13
5.12 配列の初期化.....	5-14
5.13 インクリメント/デクリメント.....	5-15
5.14 SWITCH文.....	5-16
5.15 浮動小数点即値.....	5-17
5.16 外部変数のゼロクリア.....	5-17
5.17 スタートアップの整理.....	5-18
5.18 ループ内はテンポラリを使用する.....	5-20
5.19 32ビット用数学関数.....	5-21
5.20 なるべくUNSIGNED を使う.....	5-22
5.21 配列の添え字の型.....	5-23
5.22 プロトタイプ宣言の活用.....	5-23
5.23 CHAR型の範囲でしか値を返さない関数の戻り値をCHAR型にする。.....	5-25
5.24 BSS領域のクリア処理のコメントアウト.....	5-26
5.25 生成コードの削減.....	5-27
6. シミュレータデバッガの活用.....	6-1
6.1 仮想割り込み機能を利用する.....	6-4
6.1.1 ボタン押下により仮想割り込みを入れる.....	6-4
6.1.2 一定間隔で仮想割り込みを入れる.....	6-6
6.1.3 指定サイクルで仮想割り込みを入れる.....	6-8
6.1.4 指定アドレスの命令の実行時に仮想割り込みを入れる.....	6-11
6.2 仮想ポート入出力機能を利用する.....	6-13
6.2.1 ボタン押下によりデータを入力する.....	6-13
6.2.2 指定アドレスのリード時に仮想ポートからデータを入力する.....	6-15
6.2.3 指定サイクルで仮想ポートからデータを入力する.....	6-19
6.2.4 仮想割り込み発生時に仮想ポートからデータを入力する.....	6-21
6.2.5 仮想出力ポートへ出力したデータを確認する.....	6-23
6.3 仮想LEDやラベルでメモリ内容を確認する.....	6-29
6.4 デバッグ用にPRINTFを利用する.....	6-32
6.5 I/Oスクリプトの活用.....	6-34
7. MISRA C.....	7-1
7.1 MISRA C.....	7-3

7.1.1	MISRA Cとは	7-3
7.1.2	ルールの例	7-3
7.1.3	合致マトリクス	7-4
7.1.4	ルール違反	7-5
7.1.5	MISRA C 準拠	7-5
7.2	SQMLINT	7-5
7.2.1	SQMLINTとは	7-5
7.2.2	使用方法	7-8
7.2.3	検査結果の確認方法	7-9
7.2.4	開発手順	7-9
7.2.5	対応コンパイラ	7-10
8.	Q & A	8-1
8.1	C コンパイラ (M3T-NC308WA)	8-4
8.1.1	ビットフィールド	8-4
8.1.2	メモリ管理関数	8-5
8.1.3	-ONBSD	8-6
8.1.4	最適化オプションの優先順位	8-7
8.1.5	関数のライブラリ追加	8-8
8.1.6	CONSTのROMセクション配置	8-9
8.1.7	引数のレジスタ渡し	8-10
8.1.8	関数引数の渡し方	8-11
8.1.9	プロトタイプ宣言	8-12
8.1.10	構造体ビットフィールドメンバの配置	8-13
8.1.11	インクリメント・デクリメント演算子の記述	8-15
8.1.12	外部変数の配置	8-16
8.1.13	配列のFAR領域配置	8-17
8.1.14	関数の固定アドレス配置	8-18
8.1.15	#PRAGMA ADDRESSを使った絶対アドレス指定	8-19
8.1.16	#DEFINEでの文字列定義	8-20
8.1.17	ビットフィールドメンバの型	8-21
8.1.18	変数の2重定義	8-22
8.1.19	関数のプロトタイプ宣言	8-23
8.1.20	EXTERN宣言なし関数の外部参照	8-24
8.1.21	最適化によるコード削除	8-25
8.1.22	ビットアクセスの纏め上げ	8-26
8.1.23	ライブラリ関数のROMアドレス配置	8-27
8.1.24	負の整数演算の処理仕様	8-28
8.1.25	INT型のサイズ	8-29
8.1.26	ENTER命令の制御	8-31
8.1.27	浮動小数点ライブラリの性能	8-33
8.2	リンカ	8-34
8.2.1	LN308、LN30の-LOCオプション	8-34
8.2.2	リンク時のウォーニング	8-35
8.2.3	開始アドレスの変更	8-36
8.3	STK VIEWER	8-37
8.3.1	STK VIEWERのスタックサイズ	8-37

8.4	SQMLINT	8-38
8.4.1	検査ルールを選択.....	8-38
8.4.2	レポートファイルの出力	8-39
8.4.3	レポートメッセージ (1)	8-40
8.4.4	レポートメッセージ (2)	8-41
8.5	HIGH-PERFORMANCE EMBEDDED WORKSHOP.....	8-42
8.5.1	ファイルのリンク順序.....	8-42
8.5.2	リロケータブルファイルのリンク順序.....	8-43
8.5.3	モトローラSフォーマットファイルの生成.....	8-44
8.5.4	HIGH-PERFORMANCE EMBEDDED WORKSHOPのインストール (1)	8-45
8.5.5	HIGH-PERFORMANCE EMBEDDED WORKSHOPのインストール (2)	8-46
8.5.6	ビルドの停止.....	8-47
8.5.7	ビルド対象の選択.....	8-48
8.5.8	ビルドコンフィグレーション	8-49
8.5.9	デバッグ情報の出力	8-50
8.6	SBDATA宣言ユーティリティ	8-51
8.6.1	SBDATA宣言ユーティリティ	8-51
付録		付録-1
付録A	追加機能について	付録-3
A.1	VER1.00 RELEASE 1からVER2.00 RELEASE 1への追加機能	付録-3
A.2	VER 2.00 RELEASE 1からVER2.00 RELEASE 2への機能追加.....	付録-5
A.3	VER2.00 RELEASE 2からVER3 . 00 RELEASE 1 への機能追加	付録-6
A.4	VER3.00 RELEASE1 からVER3.10 RELEASE1への機能追加	付録-9
A.5	VER 3.10 RELEASE 1からVER 3.10 RELEASE 2への機能追加.....	付録-10
A.6	VER 3.10 RELEASE 2 からVER3.10 RELEASE 3への機能追加.....	付録-12
A.7	VER3.10 RELEASE 3からVER5 . 00 RELEASE 1への機能追加.....	付録-13
A.8	VER5.00 RELEASE 1からVER5.10 RELEASE1への機能追加	付録-15
A.9	VER5.10 RELEASE 1 からVER5.20 RELEASE1への機能追加	付録-18

1. 概説

1. 概説

1.1 概要

N C 30W A、N C 308W A コンパイラは、機器組み込み用シングルチップマイコンルネサステクノロジM16Cファミリの機能・性能を活かしたプログラムをC言語で効果的に作成できるようにしたコンパイラです。

本書では、このCコンパイラを用いて応用プログラムを作成する手法を説明します。

1.1.1 概略仕様

言語仕様	ANSI準拠
INT型	16ビット
サポートデータ型	8、16、32、64ビット整数型 32、64ビット浮動小数点型
特徴	マルチメモリモデル 漢字サポート 高ROM効率

1. 概説

1.2 特長

M16C ファミリコンパイラ NC30WA,NC308WA の特長を以下に示します。

性能	業界トップクラスの ROM 効率を実現
メモリモデル	near/far 指定による細かいメモリ指定が可能
拡張機能	#pragma による割り込み関数宣言、アドレス宣言など組み込み向け機能を豊富にサポート
地域文字	EUC コードサポートにより文字列定数、コメント等にヨーロッパ語圏、アジア語圏等の文字が記述可能
ユーティリティ	ROM 圧縮ユーティリティをサポート
付属ツール	統合化開発環境 TM, High-performance Embedded Workshop 構造化記述をサポートしたアセンブラ シミュレータ

1.3 インストール方法

PC 版

インストーラを起動し表示されるメッセージにしたがってインストールしてください。インストールの途中でライセンス ID を入力する必要があります。インストールを始める前にライセンス ID を確認してください。

インストールの途中で入力するデータは、ユーザ登録のためのファイルを作成するのに使用されます（ファイルを作成するのは PC 版のインストーラのみです）。

1.4 起動方法

1.4.1 コンパイラの起動方法

- コンパイルドライバのコマンドの入力書式

コンパイルドライバは、コンパイラの各コマンドとアセンブルコマンド及びリンクコマンドを起動し、機械語データファイルを生成します。このコンパイルドライバを起動するためには、以下の情報（入力パラメータ）が必要となります。

- 1.C 言語ソースファイル
- 2.アセンブリ言語ソースファイル
- 3.リロケートブルオブジェクトファイル
- 4.起動オプション（必要に応じて記述する項目）

これらの項目をコマンド行に入力します。項目 1、2、3 のいずれか一つは最低限、入力してください。図 1.1 に入力書式を、図 1.2 に入力例を示します。入力例では、

- 1.スタートアッププログラム ncrt0.a30 をアセンブル
- 2.C 言語ソースプログラム sample.c をコンパイル / アセンブル
- 3.リロケートブルオブジェクトファイル ncrt0.a30 と sample.r30 をリンク

を行い、アブソリュートモジュールファイル sample.x30 を作成する場合の記述例を示します。起動オプションには、

- ・アブソリュートモジュールファイル名 sample.x30 の指定..... -o オプション
- ・アセンブル時のリストファイル(拡張子.lst)の出力指定..... -as30 "-l"オプション
- ・リンク時のマップファイル(拡張子.map)の出力指定..... -ln30 "-ms"オプション

を行っています。

1. 概説

% nc30 [起動オプション] <[アセンブリ言語ソースファイル名]
[リロケータブルオブジェクトファイル名] [C 言語ソースファイル名]>
% : プロンプトを示します。
<> : 必須項目を示します。
[] : 必要に応じて記述する項目を示します。
 : スペースを示します。

図1.1 コンパイルドライバの入力書式

% nc30 -osample -as30 "-l" -ln30 "-ms" ncrt0.a30 sample.c<RET>
<RET> : リターンキーの入力を示します。
※リンク時には必ずスタートアッププログラムを先に指定してください。

図1.2 コンパイルドライバコマンドの入力例

1.4.2 統合化環境からの起動方法

統合化環境のインストーラは、インストール正常終了時、Windows のスタートメニューのプログラムフォルダの下に Renesas High-performance Embedded Workshop という名称のフォルダを作成し、そのフォルダ内に統合化環境の実行プログラムである統合化環境などの各ショートカットを登録します。なお、スタートメニューの表示内容は、ツールのインストール状況により異なる場合があります。

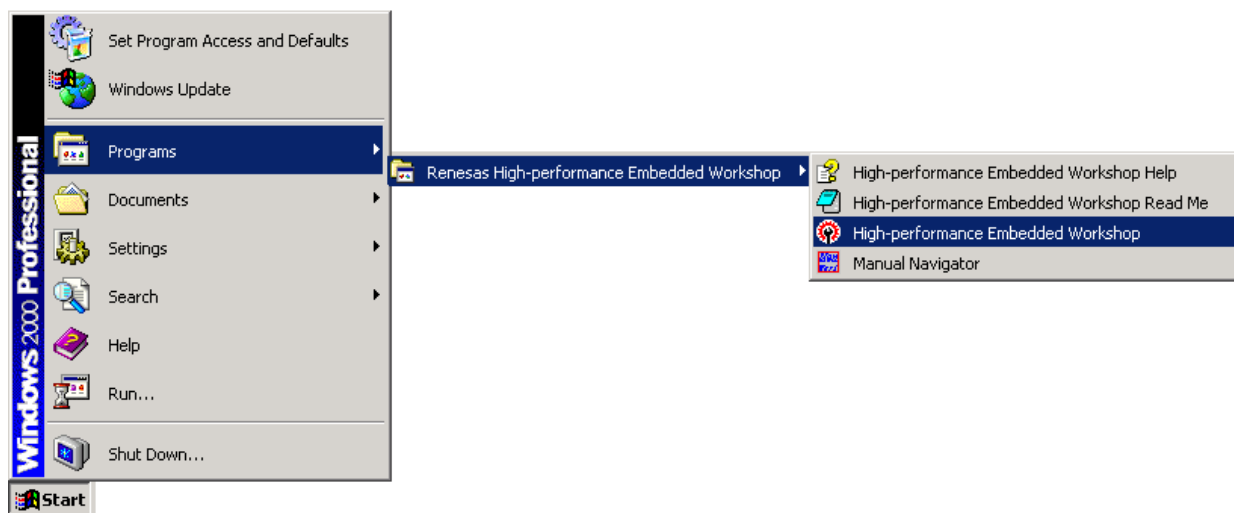


図1.3 スタートメニューによる統合化環境起動

このスタートメニューで、統合化環境をクリックすると起動メッセージを表示し、引き続き Welcome! ダイアログボックス (図 1.4) が表示されます。

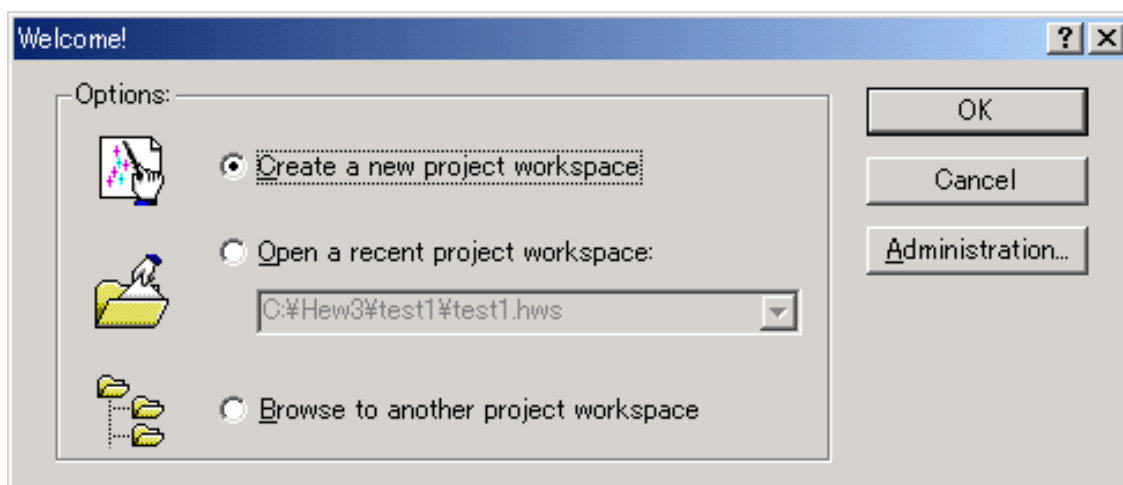


図1.4 Welcome!ダイアログボックス

統合化環境を初めて使用する場合や、新たにプロジェクトを作成して作業を開始する場合は、[Create a new workspace]を選択して[OK]をクリックしてください。また、既に作成したプロジェクトで作業する場合は、[Open a recent project workspace]または、[Browse to another project workspace]を選択して[OK]をクリックしてください。また、[Administration...]をクリックすると、統合化環境で使うシステムツールの登録や削除ができます。

1.5 プログラム開発手順

NC308 を使用したプログラム開発例の流れを図 1.5 に示します。このプログラムの概要を以下に示します（項目の(1)～(4)は図 1.5 の[1]～[4]に対応します）。

- (1)C 言語ソースプログラム（AA.c）を nc308 でコンパイル、アセンブラ as308 でアセンブルし、リロケータブルオブジェクトファイル（AA.r30）を作成します。
- (2)スタートアッププログラム ncrt0.a30 とセクション情報を記述したインクルードファイル sect308.inc を組み込むシステムに合わせて、セクションの配置 / セクションサイズ / 割り込みベクタテーブルの設定などを変更します。
- (3)変更したスタートアッププログラムをアセンブルします。この結果、リロケータブルオブジェクトファイル（ncrt0.r30）を作成します。
- (4)2 つのリロケータブルオブジェクトファイル、AA.r30 と ncrt0.r30 を nc308 から実行されるリンケージエディタ ln308 でリンクし、アプソリュートモジュールファイル（AA.x30）を作成します。

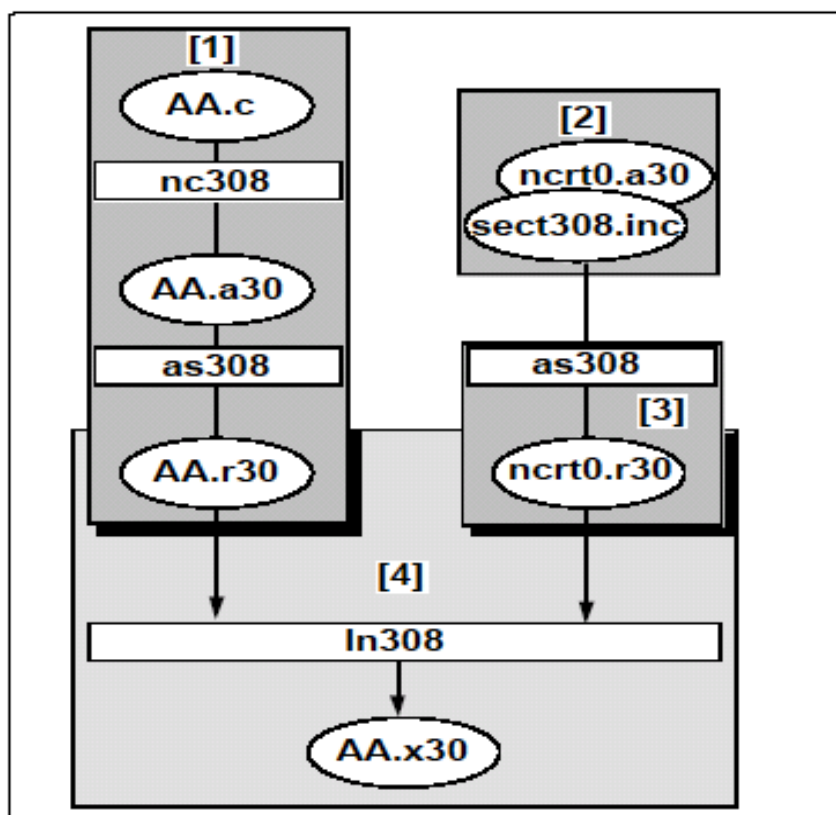


図1.5 プログラム開発フロー

2. プログラム作成の手順

2. プログラムの作成の手順

2.1 プロジェクトの構築

(1) プロジェクトの指定

“Welcome!”ダイアログで“Create a new project workspace”を選択して[OK]をクリックすると新しいワークスペースとプロジェクト作成用の“New Project Workspace”ダイアログボックス(図2.1)を表示します。このダイアログボックスでワークスペース名(新規作成時はプロジェクト名もデフォルトで同名です)やCPUの種類、プロジェクトのタイプなどを設定します。たとえば[Workspace Name]にワークスペース名として“tutorial”と入力すると[Project Name]も“tutorial”になり、[Directory]も“C:\Hew3\tutorial”となります。プロジェクト名を変更する場合は[Project Name]に直接入力し、ワークスペースとして使用するディレクトリを変更する場合は[Browse...]をクリックしてディレクトリを選択するか直接[Directory]に入力してください。

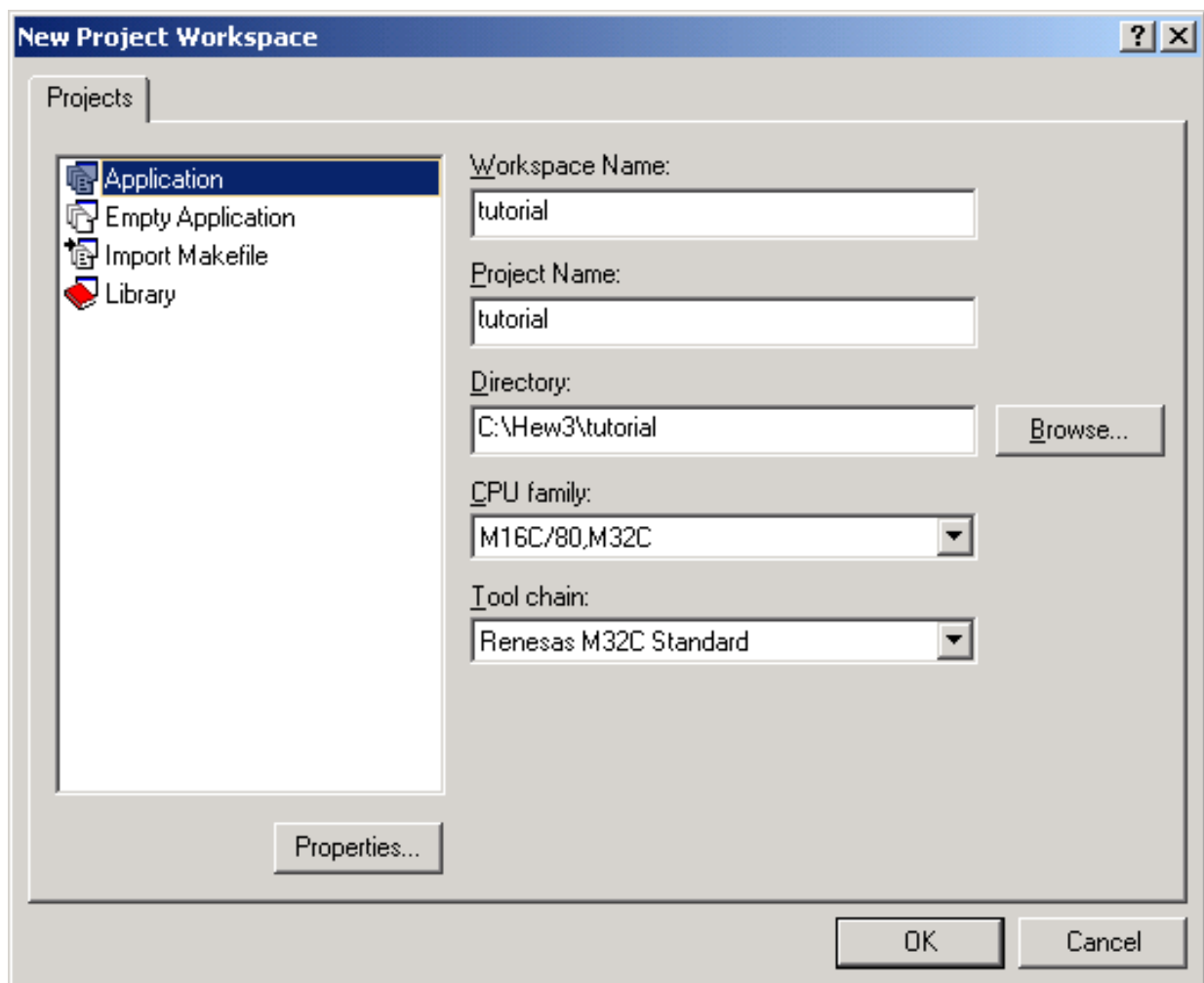


図2.1 New Project Workspace ダイアログ

2. プログラムの作成の手順

(2) CPUの選択

“ New Project Workspace ” ダイアログボックスで[OK]をクリックするとプロジェクトジェネレータを起動します。最初に使用するCPUを選択します。使用するCPUの種類([CPU Type])はCPUのシリーズ ([CPU Series]) ごとに分類しています。[CPU Series]および[CPU Type]の選択により生成するファイルが異なるので、開発するプログラムの対象となるCPUタイプを選択してください。選択したいCPUタイプがない場合は、ハードウェア仕様の近いCPUタイプまたは”Other”を選択してください。

- ・ [Next >]をクリックすると次の画面を表示します。
- ・ [<Back]をクリックするとこの画面を表示する前の画面またはダイアログボックスを表示します。
- ・ [Finish]をクリックすると”Summary”ダイアログボックスが開きます。
- ・ [Cancel]をクリックすると”New Project Workspace” ダイアログボックスに戻ります。

[<Back],[Next>],[Finish],[Cancel]の機能はこのウィザードダイアログボックスで共通の機能です。

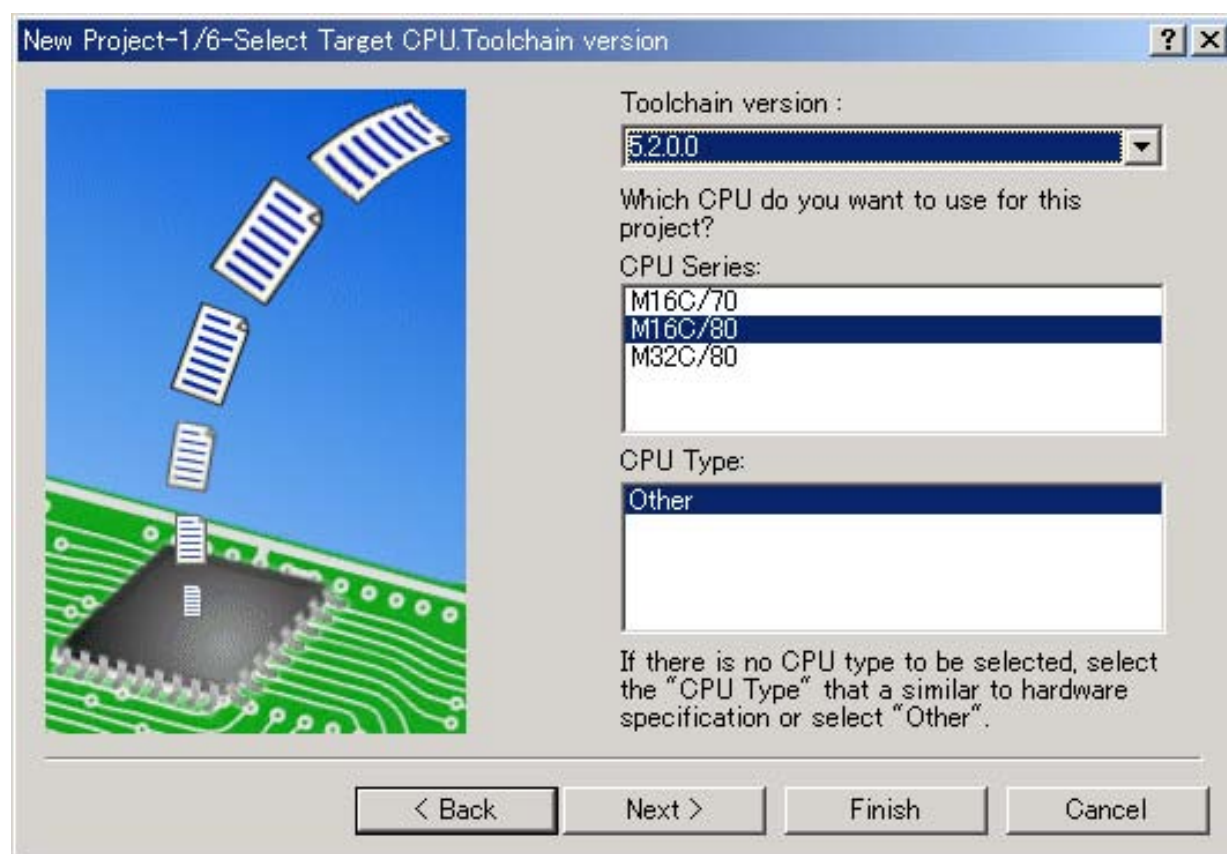


図2.2 New Project Step 1 ダイアログボックス

(3) RTOSの選択

Step1画面で[Next >]をクリックすると、図2.3に示す画面を表示します。この画面で、RTOSの有無やスタートアップファイルをデフォルトのものを使うかユーザ定義のものを使うかを選択できます。

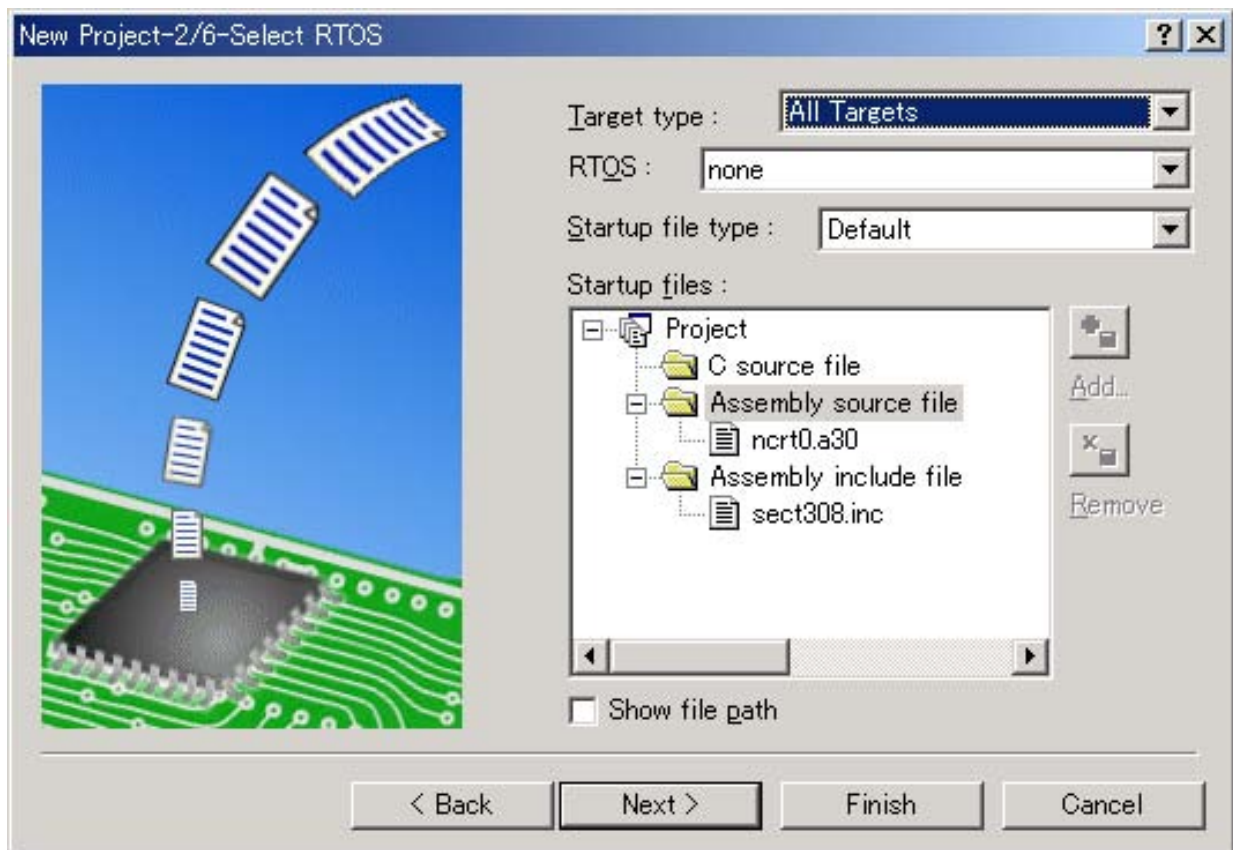


図2.3 New Project Step 2 ダイアログボックス

2. プログラムの作成の手順

(4) 入出力ライブラリ、ヒープ領域の設定

Step 2 画面で[Next >]をクリックすると、図2.4に示す画面を表示します。この画面でI/Oライブラリの使用の有無やヒープ領域のサイズの設定、main関数ファイルの生成有無を選択できます。

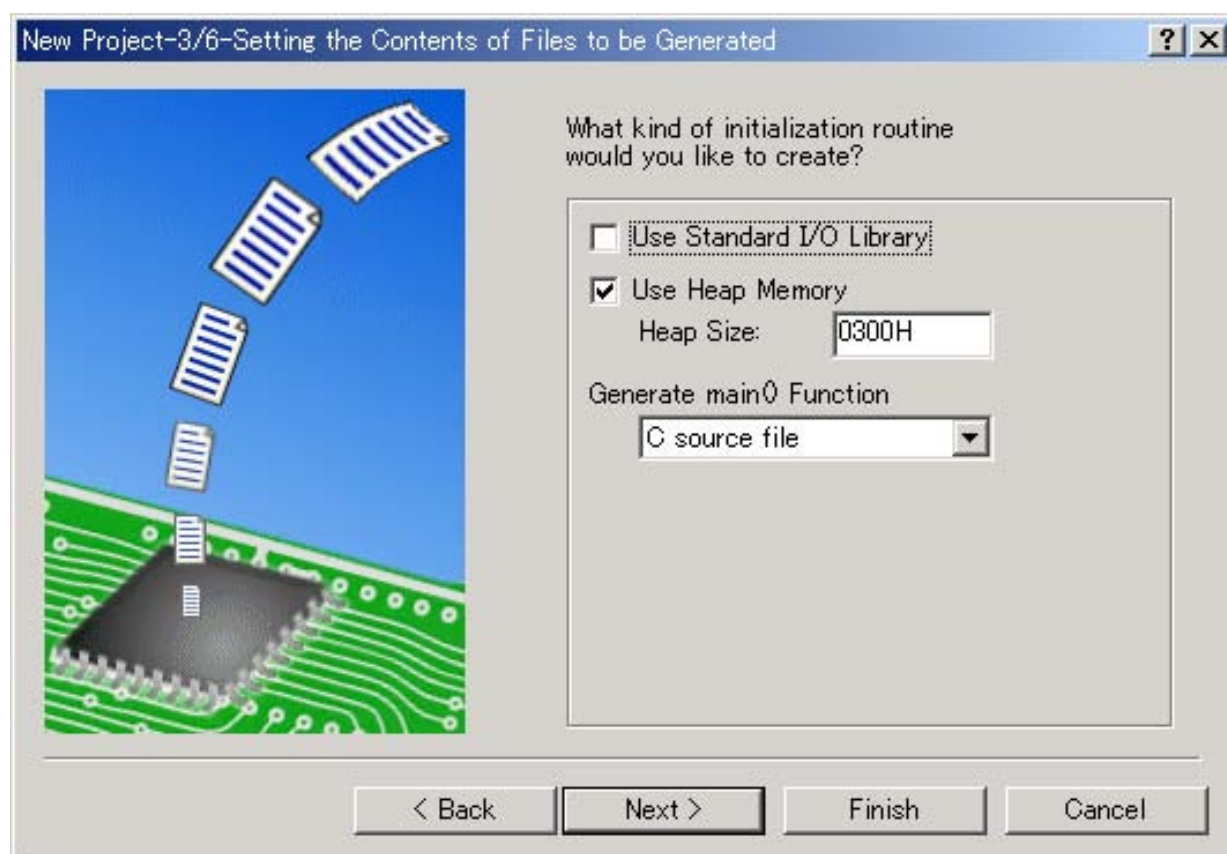


図2.4 New Project Step 3 ダイアログ

(5) スタック領域の設定

Step3 画面で[Next >]をクリックすると図2.5に示す画面を表示します。この画面でスタックサイズと割り込みスタックサイズの設定ができます。

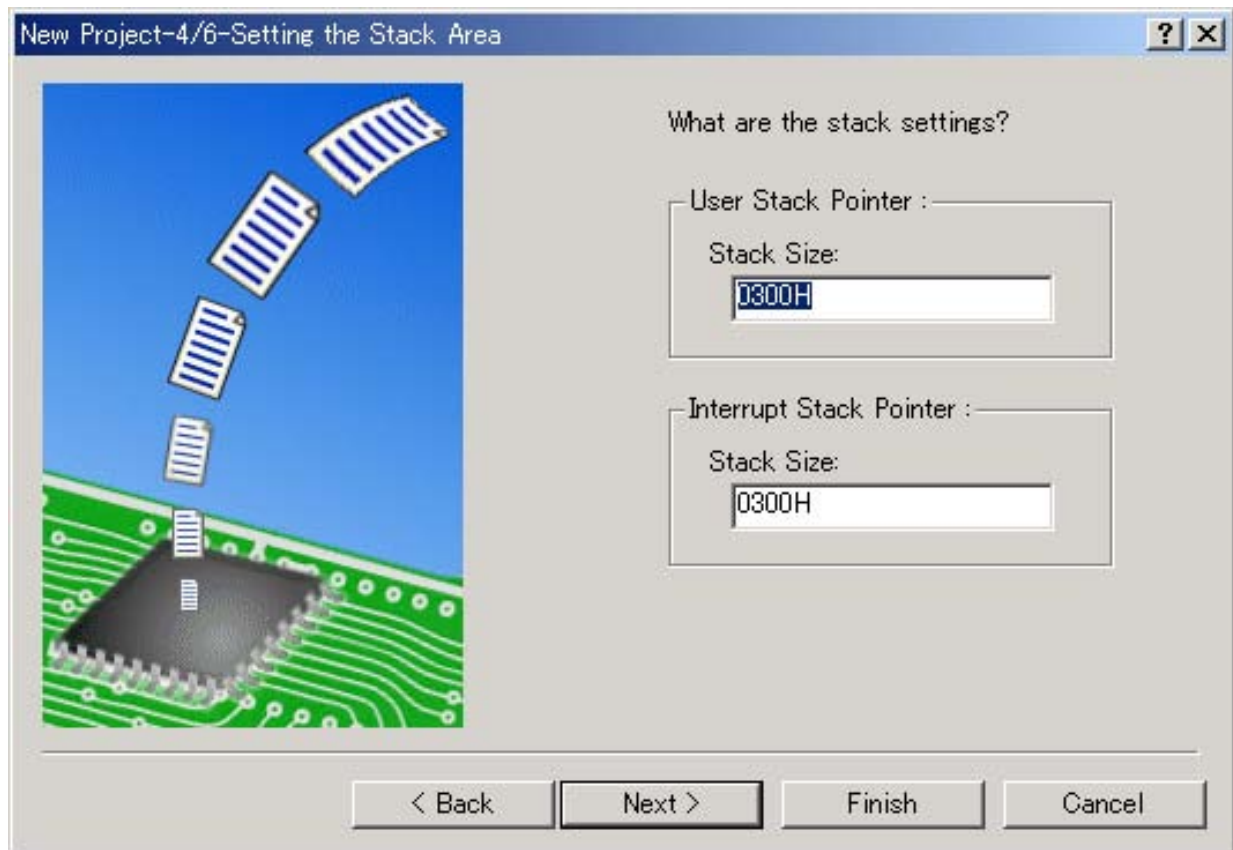


図2.5 New Project Step 4 ダイアログ

2. プログラムの作成の手順

(6) デバッガの設定

Step 4 の画面で[Next >]をクリックすると図2.6に示す画面が表示されます。この画面ではデバッガターゲットを設定します。[Targets:]から使用するデバッガターゲットを選択してください。デバッガターゲットは未選択でもかまいません。また外部パッケージのデバッガがあればそれを選択することもできます。

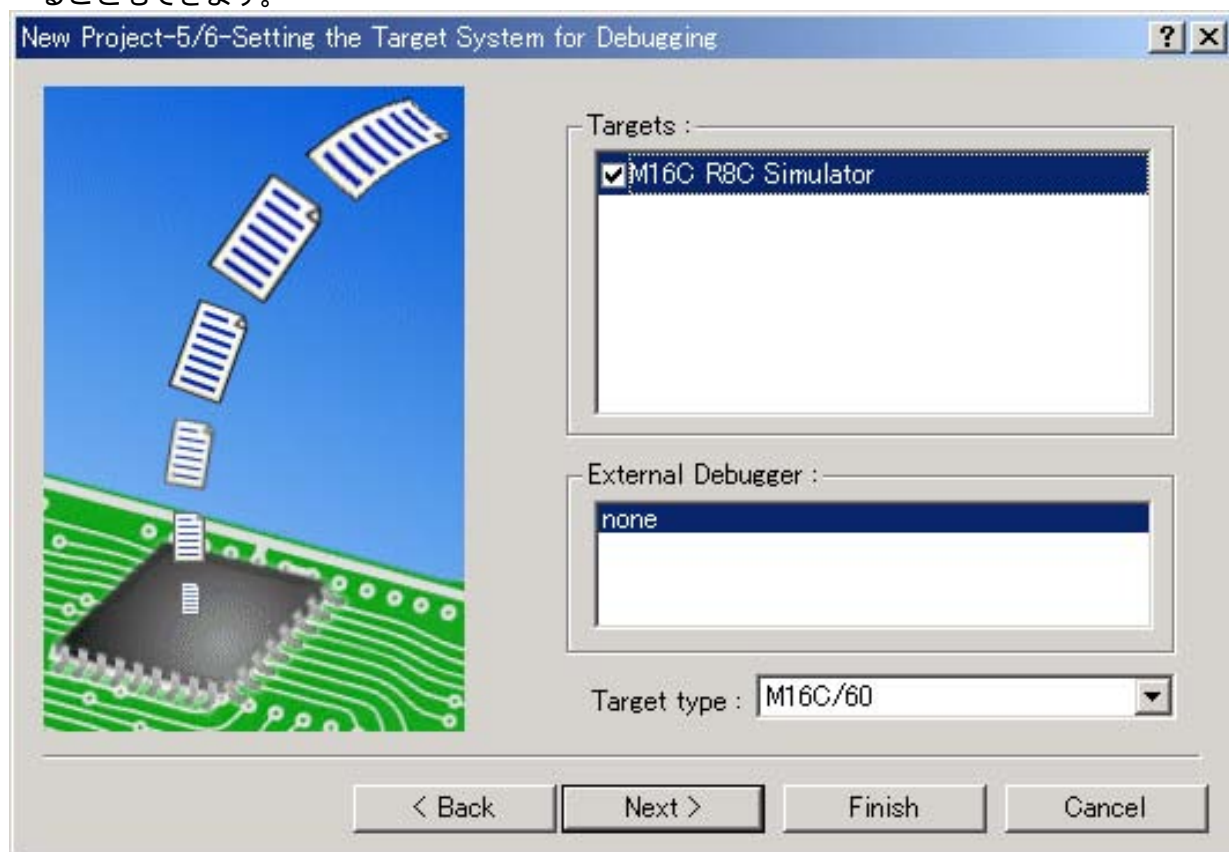


図2.6 New Project Step 5 ダイアログ

2. プログラムの作成の手順

(7) コンフィグレーションファイル名の設定 Step 5 の画面で[Next >]をクリックすると図2.7に示す画面が表示されます。この画面は選択したターゲット毎にコンフィグレーションファイル名を設定します。コンフィグレーションとはターゲット以外のHigh-performance Embedded Workshop の状態を保存するファイルです。

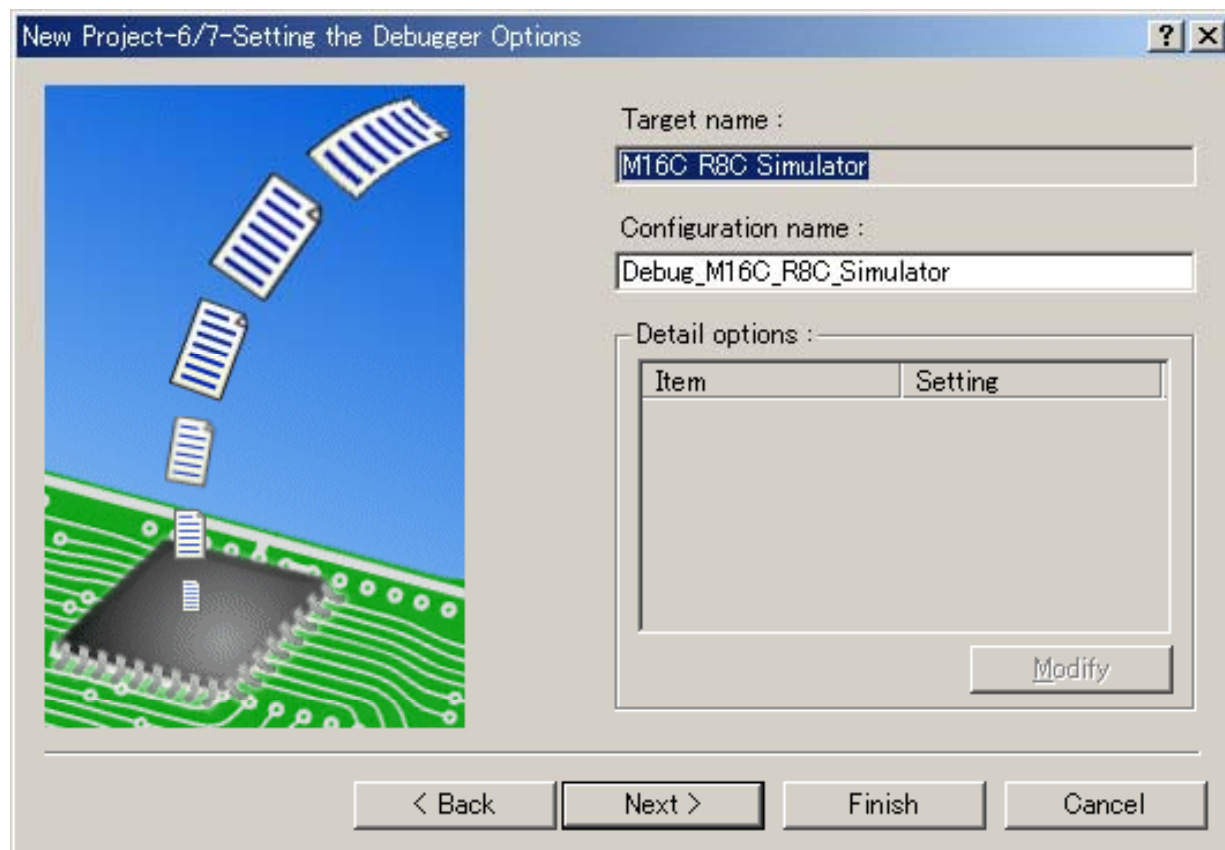


図2.7 New Project Step 6 ダイアログ

(8) 設定確認 (Summaryダイアログボックス)

Step6画面で[Next >]をクリックすると図2. 8に示す画面を表示します。この画面で生成するプロジェクトのソースファイル情報を表示します。確認後,[Finish]をクリックしてください。図2. 8の画面で[Finish]をクリックするとプログラムジェネレータは生成するプロジェクトに関する情報をSummaryダイアログボックス (図2. 9) で表示しますので確認後,[OK]をクリックしてください。

なお、[Generate Readme.txt as a summary file in the project directory]をチェックすると、Summaryダイアログボックスで表示したプロジェクトの情報を”Readme.txt”という名称のテキストファイルでプロジェクトディレクトリに保存します。

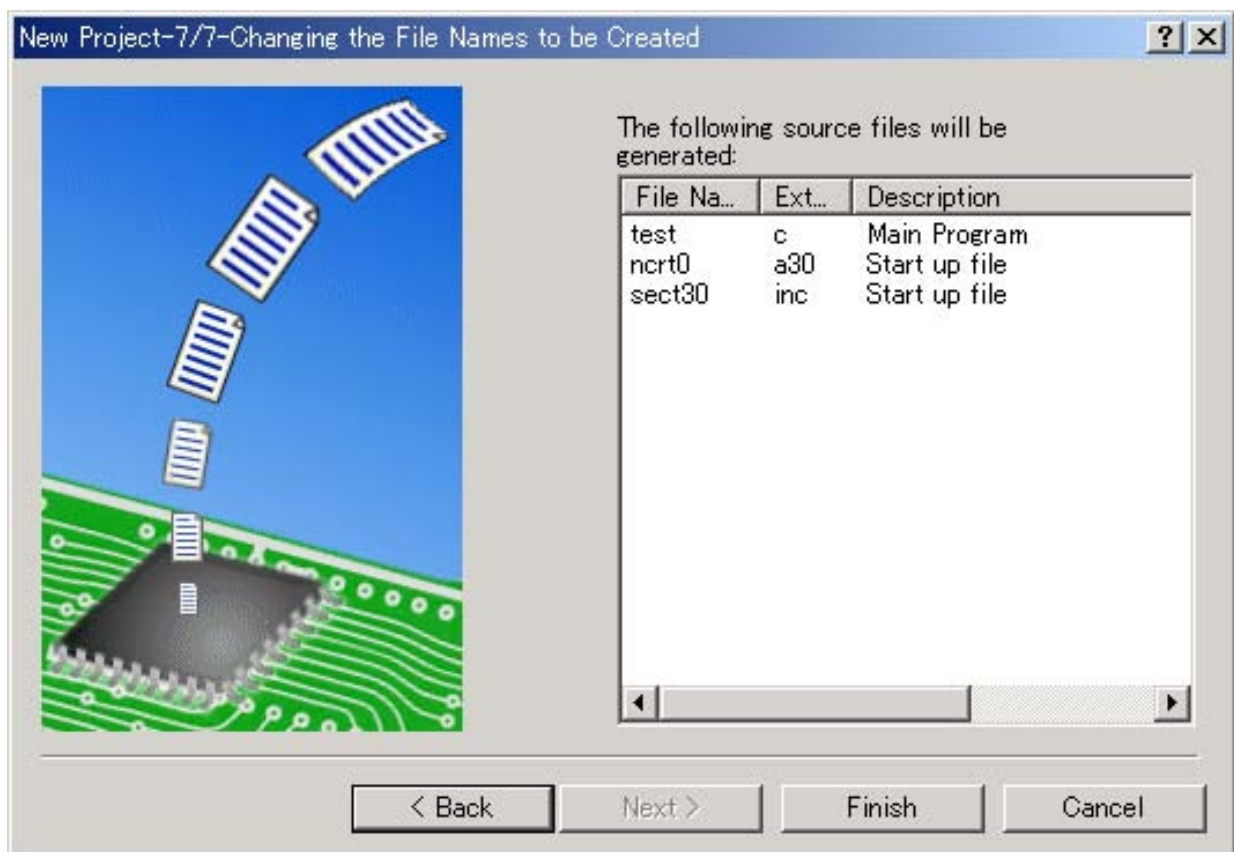


図2.8 New Project Step 7 ダイアログ

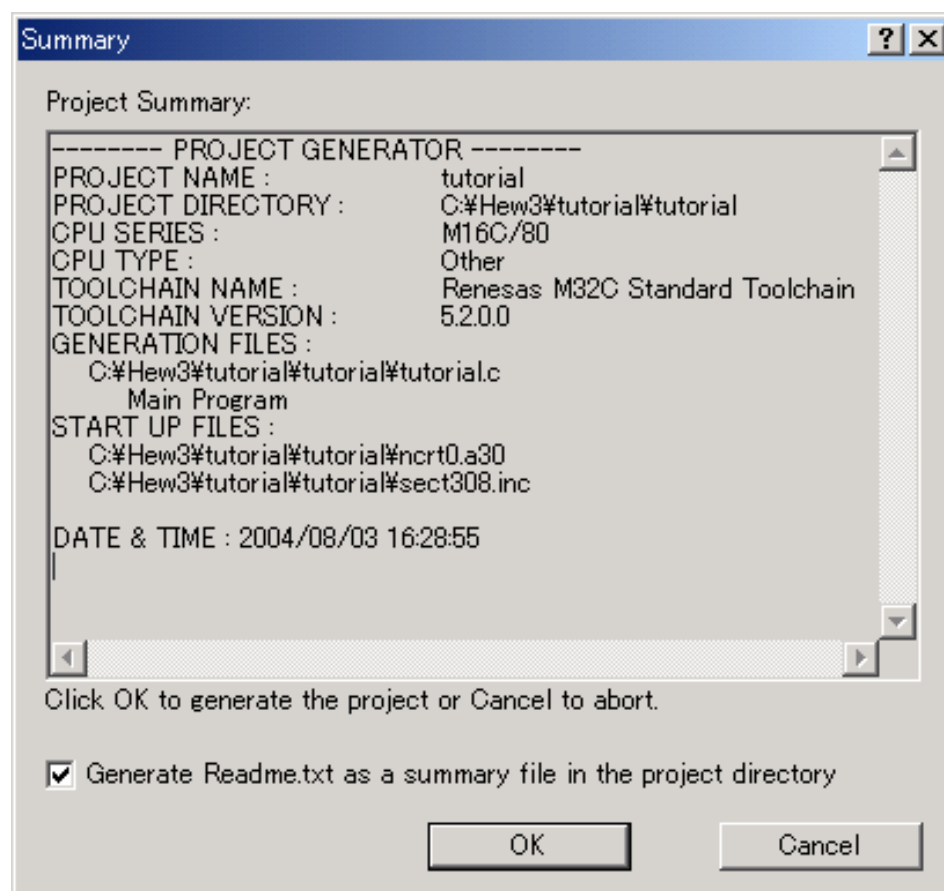


図2.9 Summary ダイアログボックス

2.2 スタートアッププログラム

2.2.1 スタートアッププログラムの役割

組み込み型のプログラムを正常に動作させるためには、処理の前にマイコンの初期化やスタック領域の設定を行わなくてはなりません。通常、これらの処理はC言語では記述できません。そこでC言語のソースプログラムとは別にアセンブリ言語で初期設定用のプログラムを記述します。これが「スタートアッププログラム」です。以下の項ではNC308が用意しているサンプルスタートアッププログラム“ncrt0.a30”と“sect308.inc”について説明します。

スタートアッププログラムの役割

- スタック領域の確保
- マイコンの初期設定
- 静的変数領域の初期化
- 割り込みテーブルレジスタ“INTB”の設定
- main関数の呼び出し
- 割り込みベクタテーブルの設定

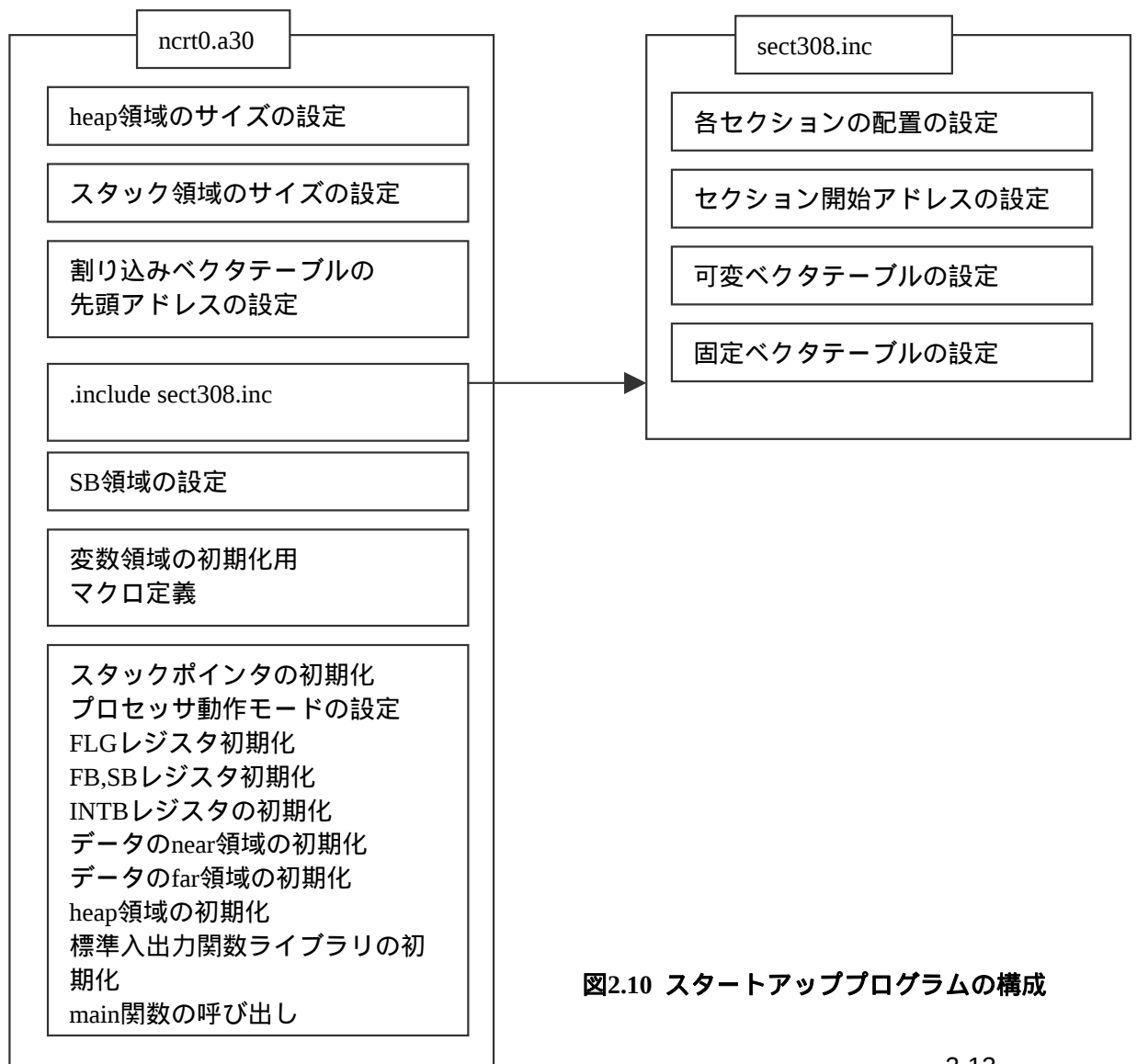


図2.10 スタートアッププログラムの構成

2.2.2 スタートアッププログラムの設定

(1) セクション名の追加

NC308 が生成するセクションはセクション定義ファイル"sect308.inc"に定義されています。
"#pragma SECTION"でセクション名を変更するということは、NC308 が生成するセクションベース名が追加されたということになります。そのため、必ずセクション定義ファイル"sect308.inc" でセクション名を追加定義しなければなりません。

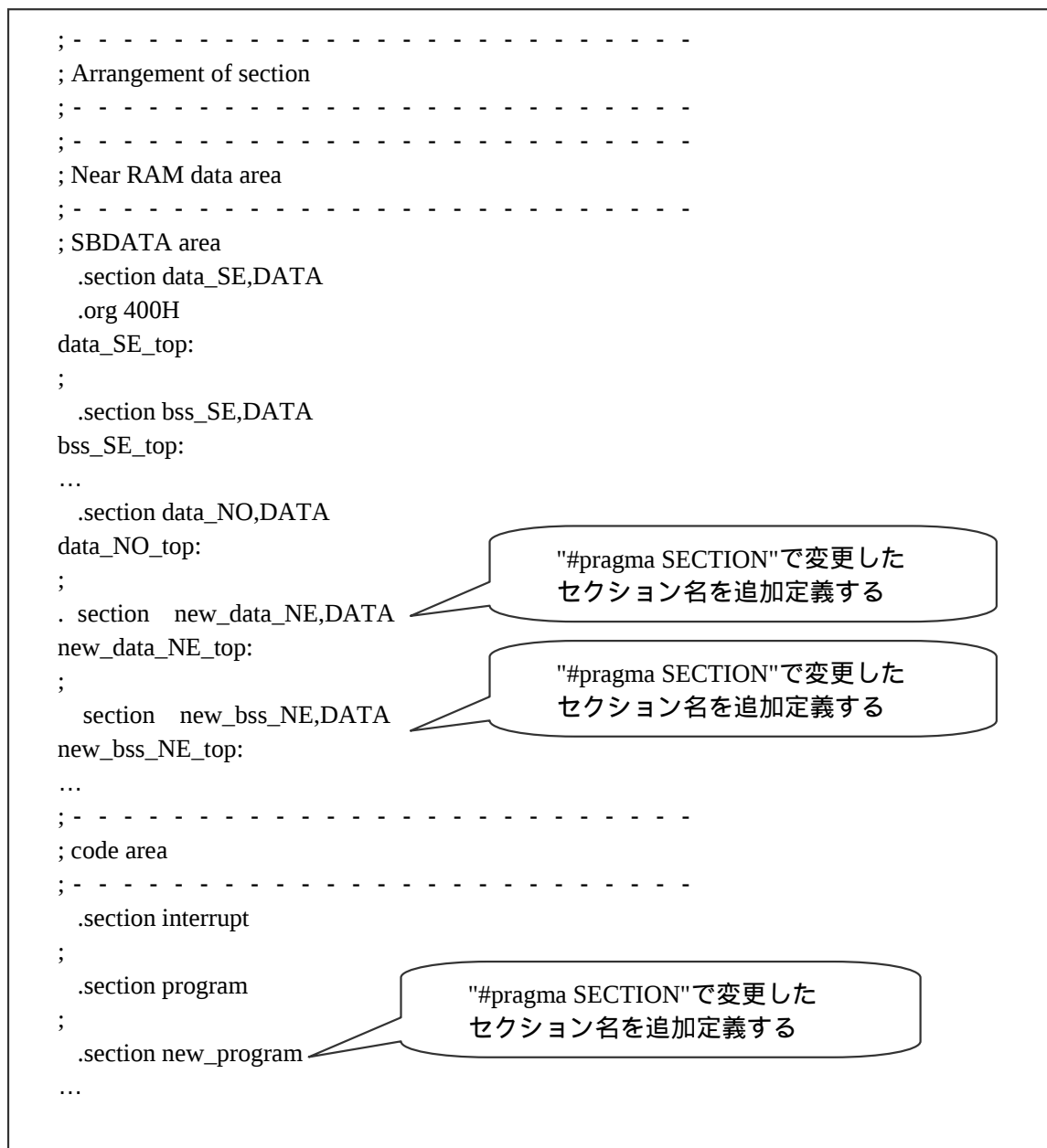


図2.11 セクション名の追加(sect308.inc)

data セクション,bss セクションを追加した場合セクション名の追加定義のほかにそれぞれその領域

の初期値転送処理、ゼロクリア処理が必要になります。ncrt0.a30 にゼロクリア処理、初期値転送処理を追加してください。

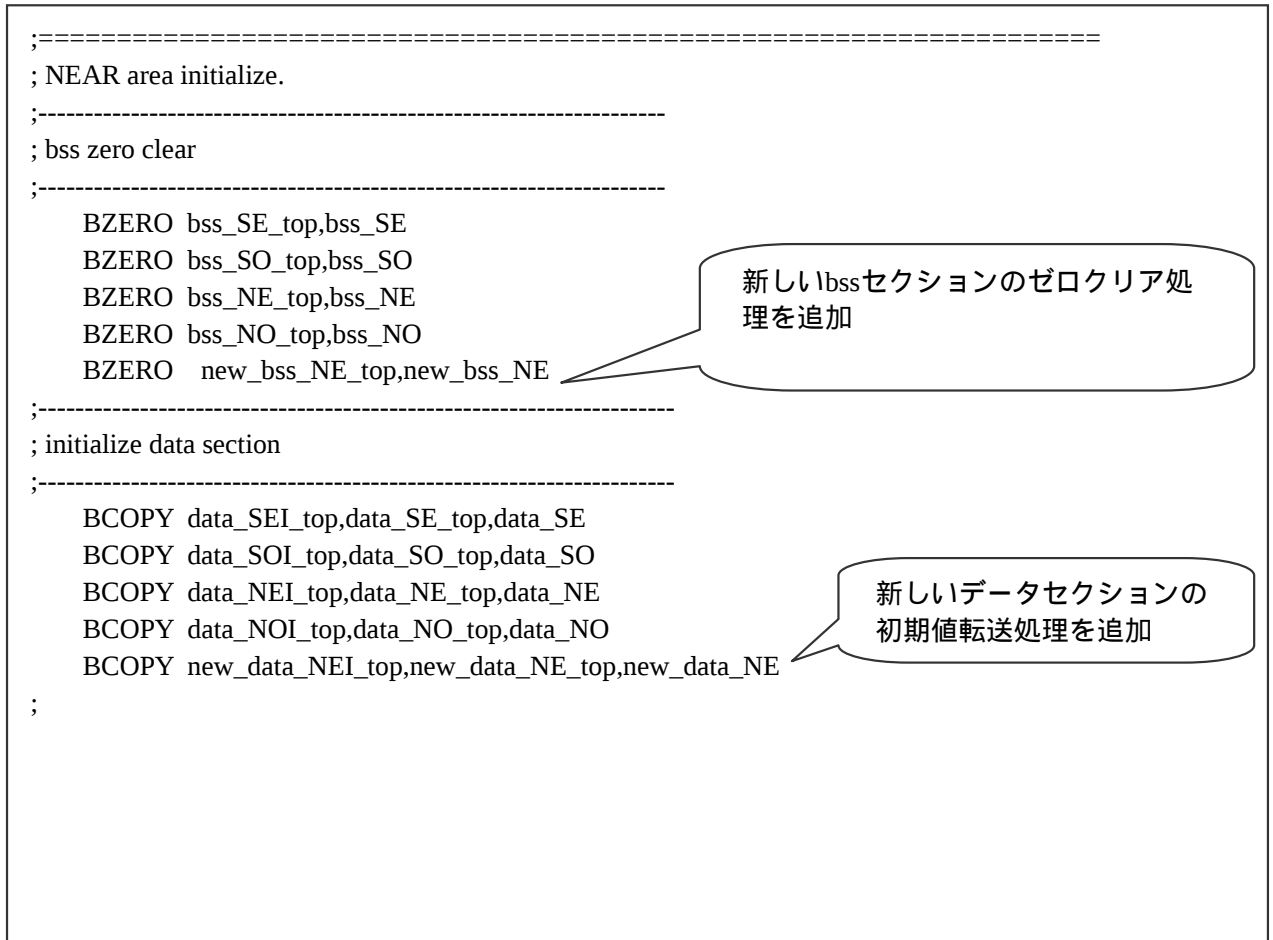


図2.12 追加したセクションの初期化処理の追加

2. プログラムの作成の手順

(2) 割り込み関数の登録

割り込みを正常に使用するためには、割り込み処理関数を記述するとともに割り込みベクタテーブルに登録する必要があります。この項では、割り込みベクタテーブルへの登録方法を説明します。

割り込み処理関数を記述する場合、サンプルスタートアッププログラム "sect308.inc" 中の割り込みベクタテーブルを変更することにより、割り込み処理関数を登録します。

割り込みベクタテーブルの変更は以下の手順で行います。

割り込み処理関数名を指示命令 ".glob " で外部参照宣言する。

使用する割り込みのダミー関数名 "dummy_int" を、割り込み処理関数名に変更する。

```
; - - - - -
; variable vector section
; - - - - -

.section vector ; variable vector table
.org VECTOR_ADR
;
.lword dummy_int ; vector (BRK)
.org ( VECTOR_ADR + 32 )
.lword dummy_int ; DMA0 (software int 8)
.lword dummy_int ; DMA1 (software int 9)
.lword dummy_int ; DMA2 (software int 10)
.lword dummy_int ; DMA3 (software int 11)
.glob _ta0
.lword _ta0 ; TIMER A0 (software int 12)
.lword dummy_int ; TIMER A1 (software int 13)
.lword dummy_int ; TIMER A2 (software int 14)
.lword dummy_int ; TIMER A3 (software int 15)
.lword dummy_int ; TIMER A4 (software int 16)
...
```



図2.13 割り込みベクタテーブル (sect308.inc)

3. コンパイラ

3. コンパイラ

3.1 割り込み関数

3.1.1 割り込み処理関数の記述

NC308 では割り込み処理を C 言語関数として記述することができます。手順は次の 4 つです。

割り込み処理関数の記述

割り込みベクタテーブルへの登録

割り込み許可フラグ (I フラグ) の設定

- ・インラインアセンブル機能により行います。

使用する割り込みの割り込み優先レベルの設定

- ・割り込み優先レベルは割り込みを許可する前に設定します。

この項では、割り込み処理の種類別に関数の記述方法を説明します。

(1) ハードウェア割り込みの記述 (#pragma INTERRUPT 割り込み関数名)

```
#pragma INTERRUPT 割り込み関数名
```

上のように宣言すると、指定した関数の入口と出口において、通常関数の手続き以外に関数内で使用する全レジスタの退避・復帰と reit 命令を生成します。割り込み処理関数の型は、引数 / 戻り値ともに void 型のみ有効です。それ以外の型を宣言した場合は、コンパイル時にウォーニングを出力します。

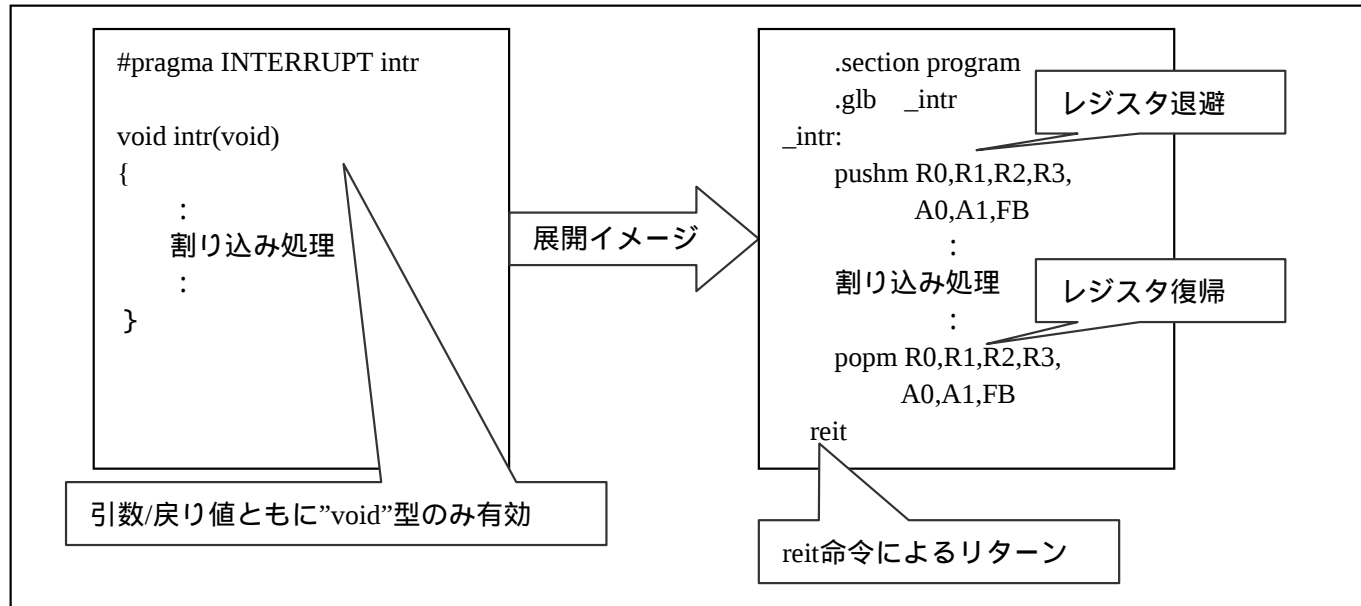


図3.1 割り込み処理関数の展開イメージ

3. コンパイラ

(2) レジスタバンクを使用した割り込みの記述 (#pragma INTERRUPT/B 割り込み関数名)

M16C/80 シリーズではレジスタバンクを切り換えることによって、レジスタの内容などを保護しつつ割り込み処理が起動されるまでの時間を短縮することができます。この機能を使用したい場合は以下のように記述します。

```
#pragma INTERRUPT/B 割り込み関数名
```

上のように記述するとレジスタ退避 / 復帰の命令の代わりに、レジスタバンクを切り換える命令が生成されます。ただし、M16C/80 シリーズのレジスタバンクはレジスタバンク 0 , 1 の 2 セットですので、指定できる割り込みはひとつです。短時間での起動が必要な割り込みに対してこの機能を使用するようにしてください。

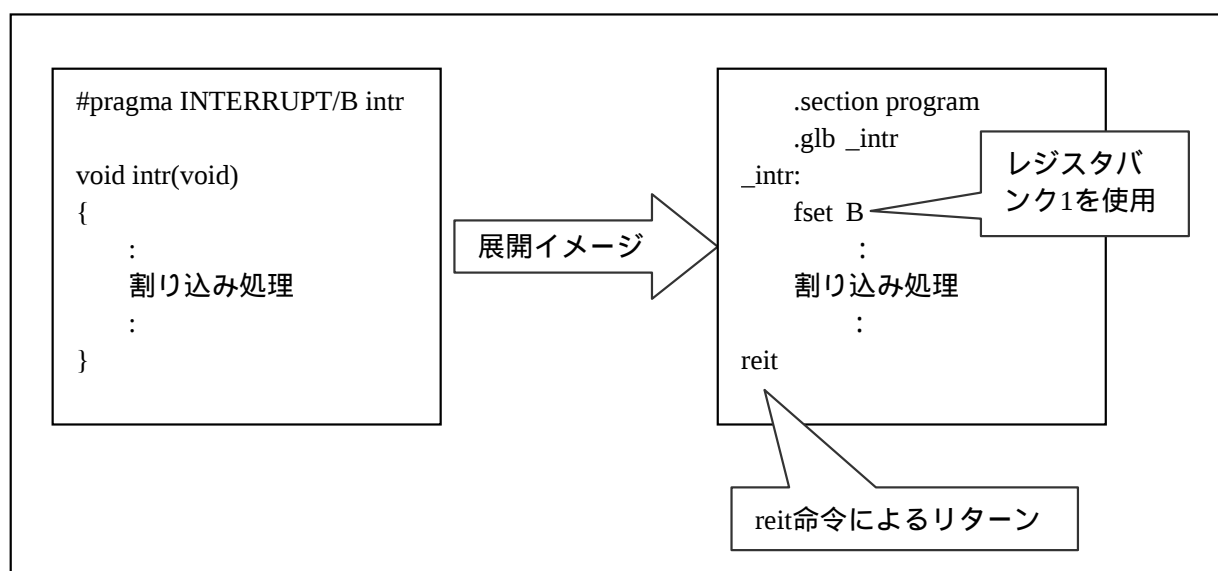


図3.2 レジスタバンクを利用した割り込み処理関数の展開イメージ

(3) 多重割り込みを許可にする割り込みの記述 (#pragma INTERRUPT/E)

M16C/80 シリーズでは、割り込み要求が受け付けられると割り込み許可フラグ (I フラグ) が "0" になり割り込み禁止の状態となります。そのため、割り込み処理関数の入り口 (割り込み処理関数に入った直後) で割り込み許可フラグ (I フラグ) を "1" にして多重割り込みを許可の状態にすると、割り込みの応答性を向上させることができます。この機能を使用したい場合は以下のように記述します。

```
#pragma INTERRUPT/E 割り込み関数名
```

上のように記述すると割り込み処理関数の入り口 (割り込み処理関数に入った直後) で割り込み許可フラグ (I フラグ) を "1" にする命令が生成されます。ただし、割り込み処理関数の途中で多重割り込み許可にしたい場合は、"#pragma INTERRUPT" 宣言を行い、割り込み処理関数の途中で asm() 関数を使用して割り込み許可フラグ (I フラグ) を "1" に設定してください。

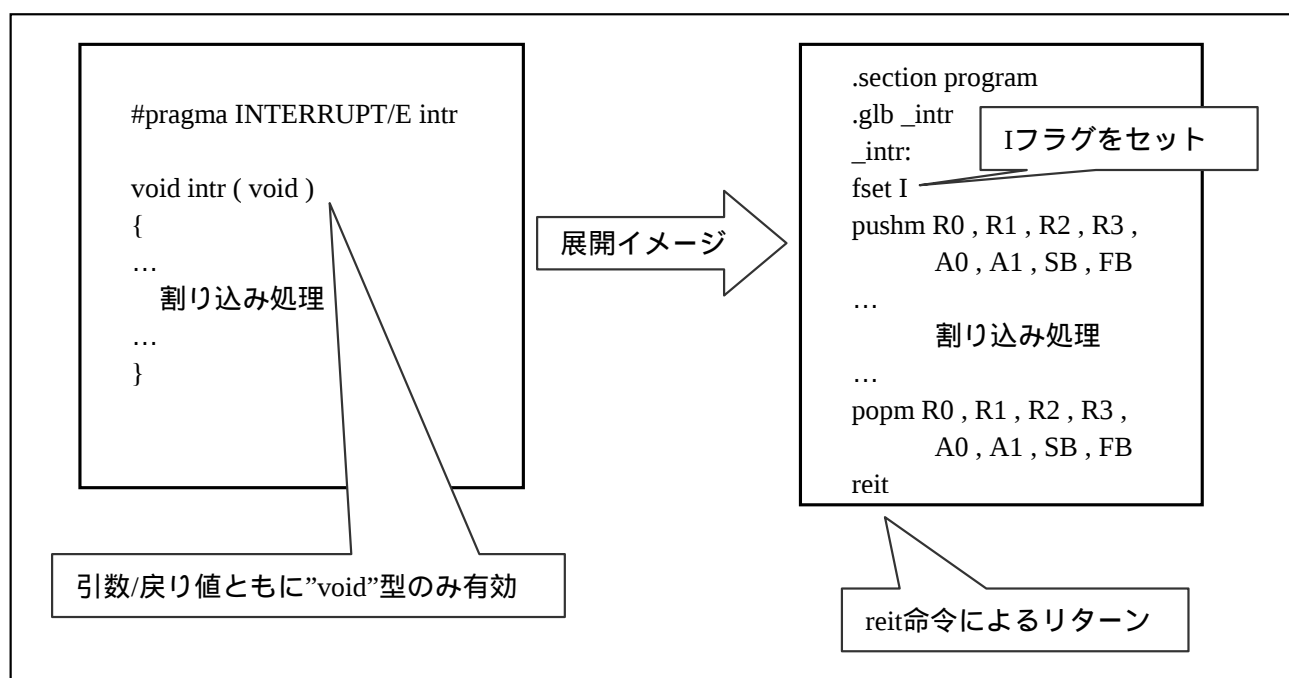


図3.3 多重割り込みを許可にする割り込み処理関数の展開イメージ

3.1.2 高速割り込み処理関数の記述

NC308 では割り込みの応答を 5 サイクルで、また割り込みの復帰を 3 サイクルで実行できる高速割り込み処理を C 言語関数として記述することができます。ただし、高速割り込みに設定できる割り込みは割り込み優先レベル 7 の 1 つの割り込みだけです。手順は次の 5 つです。

高速割り込み処理関数の記述

使用する高速割り込みの割り込み優先レベルの設定

- ・割り込み優先レベルは割り込みを許可する前に設定します。

高速割り込み指定ビットの設定

ベクタレジスタ (VCT) の設定

割り込み許可フラグ (I フラグ) の設定

- ・インラインアセンブル機能により行います。

この項では、高速割り込み処理の種類別に関数の記述方法を説明します。

(1) 高速ハードウェア割り込みの記述 (#pragma INTERRUPT/F 割り込み関数名)

```
#pragma INTERRUPT/F 割り込み関数名
```

上のように宣言すると、指定した関数の入口と出口において、通常の関数の手続き以外に関数内で使用する全レジスタの退避・復帰と高速割り込みルーチンからの復帰命令 `freit` を生成します。割り込み処理関数の型は、引数 / 戻り値ともに `void` 型のみ有効です。それ以外の型を宣言した場合は、コンパイル時にウォーニングを出力します。

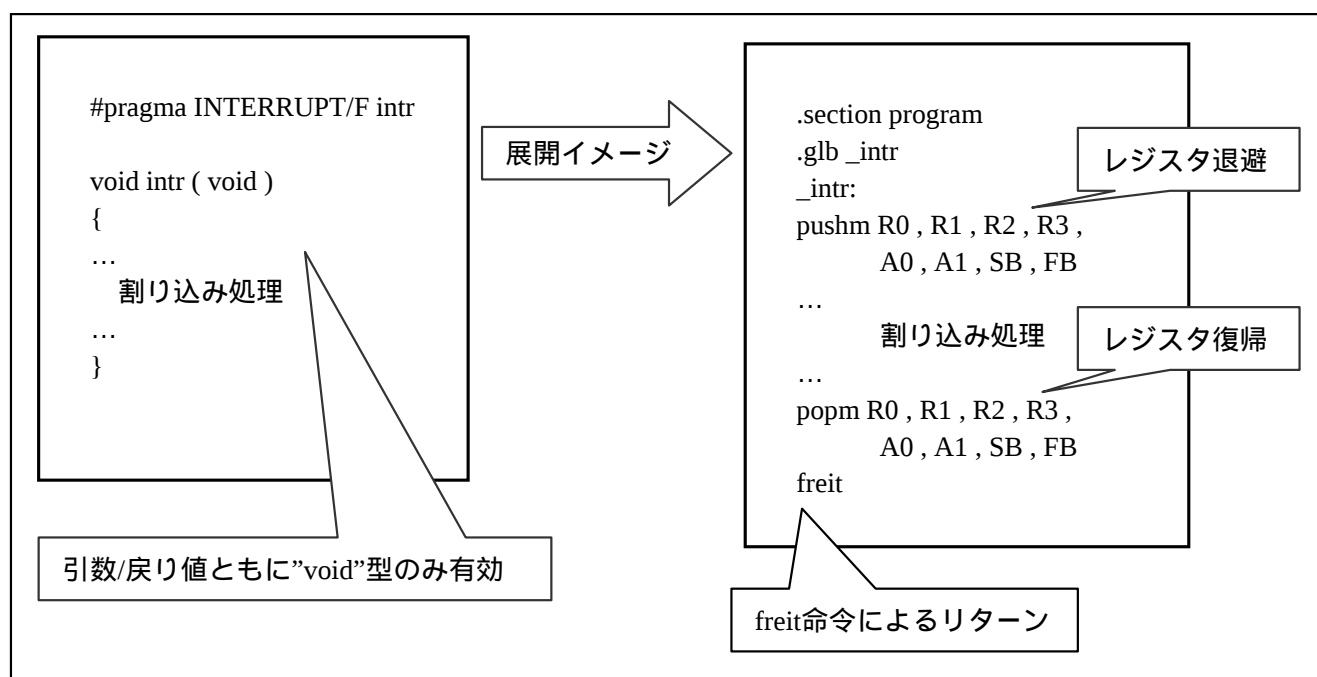


図3.4 高速割り込み処理関数の展開イメージ

(2) レジスタバンクを使用した高速割り込みの記述 (#pragma INTERRUPT/F/B 割り込み関数名)

M16C/80 シリーズではレジスタバンクを切り換えることによって、レジスタの内容などを保護しつつ高速割り込み処理が起動されるまでの時間を短縮することができます。この機能を使用したい場合は以下のように記述します。

```
#pragma INTERRUPT/F/B 割り込み関数名
```

上のように記述するとレジスタ退避 / 復帰の命令の代わりに、レジスタバンクを切り換える命令が生成されます。ただし、M16C/80 シリーズのレジスタバンクはレジスタバンク 0, 1 の 2 セットですので、指定できる割り込みはひとつです。最も短時間での起動が必要な割り込みに対してこの機能を使用するようにしてください。

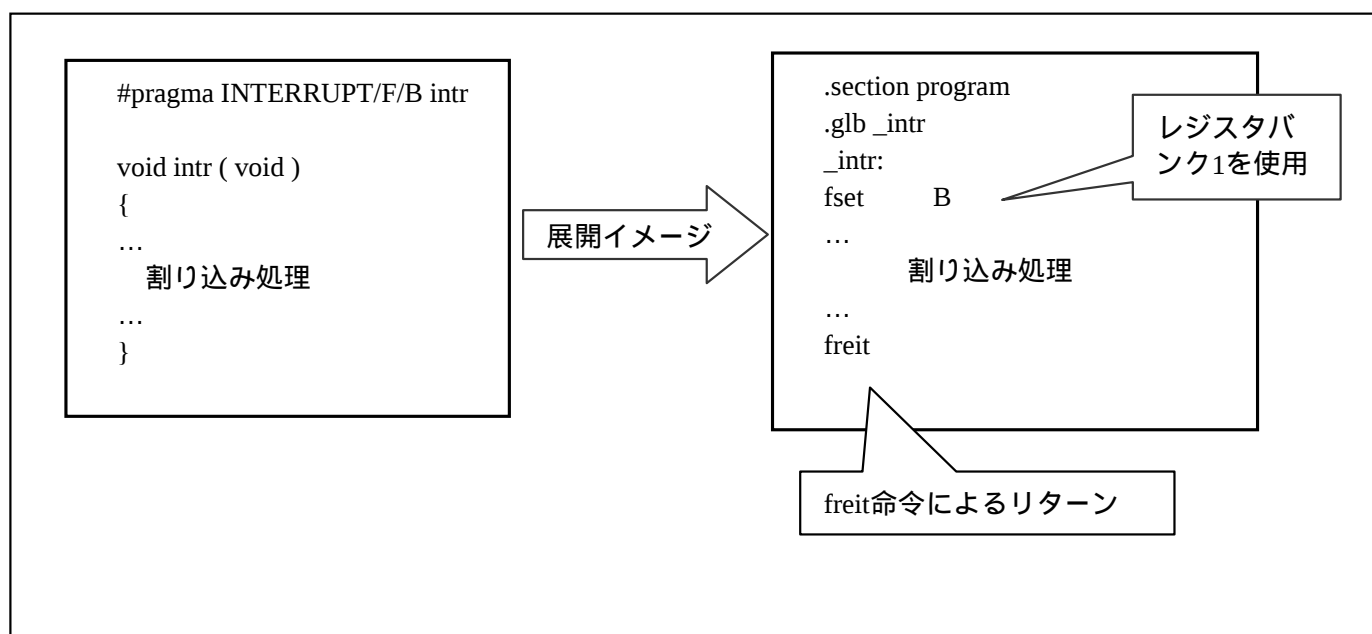


図3.5 レジスタバンクを使用した高速割り込み処理関数の展開イメージ

3.1.3 ソフトウェア割り込み (INT 命令) 処理関数の記述

(1) アセンブリ言語関数を呼び出すソフトウェア割り込みの記述 (#pragma INTCALL)

M16C/80 シリーズのソフトウェア割り込み (INT 命令) を使用する場合は "#pragma INTCALL" を記述します。この機能を使用するとデバッグ時に擬似的に割り込みを発生させることができます。

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合と C 言語で記述されている場合で記述方法が異なります。

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合は以下のように記述します。

```
#pragma INTCALL ソフトウェア割り込み番号 アセンブリ言語関数名(レジスタ名, レジスタ,...)
```

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合は、レジスタを経由して引数を渡すことができます。また構造体・共用体以外の戻り値を受け取ることができます。

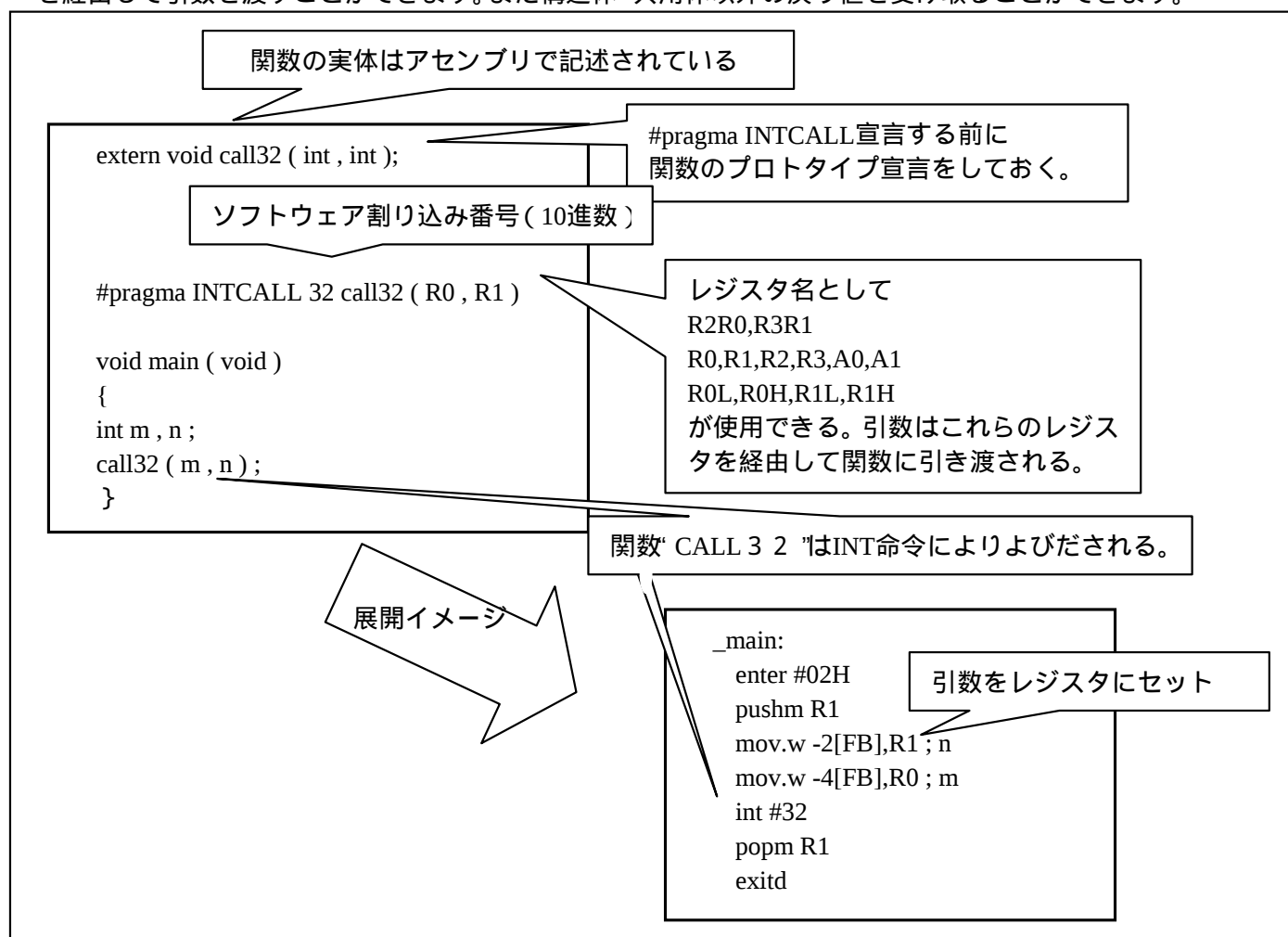


図3.6 アセンブリ言語関数を呼び出す "#pragma INTCALL" 記述例

(2) C言語関数を呼び出すソフトウェア割り込みの記述 (#pragma INTCALL)

ソフトウェア割り込み (INT 命令) で呼び出す関数の実体がC言語で記述されている場合は以下のように記述します。

```
#pragma INTCALL ソフトウェア割り込み番号 C言語関数
```

ソフトウェア割り込みで呼び出す関数の実体がC言語で記述されている場合は、引数の引き渡し規則によって、すべての引数がレジスタ渡しとなる関数のみ指定することができます。"#pragma INTCALL"宣言を行う関数の引数は記述できません。また構造体・共用体以外の戻り値を受け取ることができます。

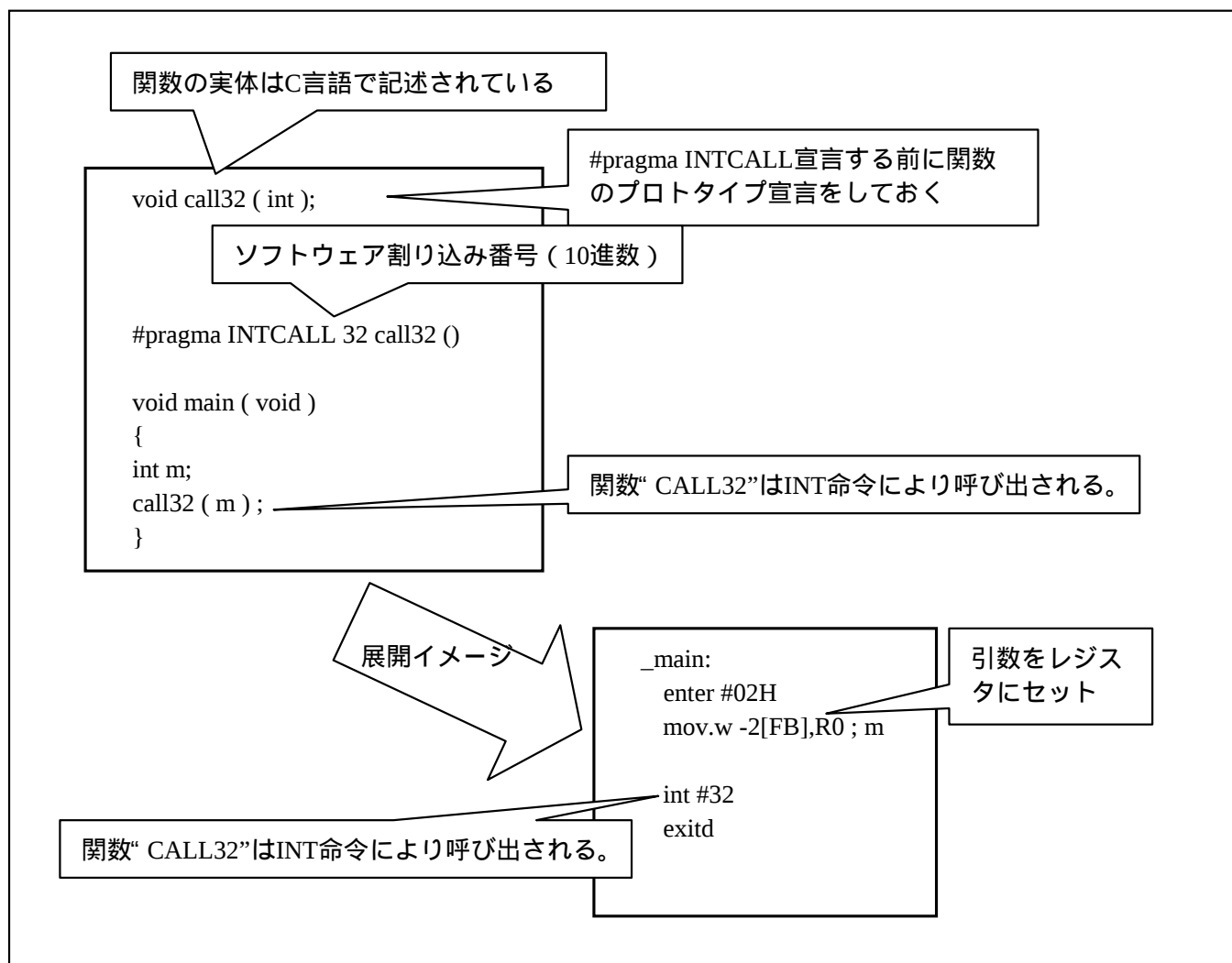


図3.7 C言語関数を呼び出す " #pragma INTCALL " 記述

3.1.4 割り込み処理関数を登録する

割り込みを正常に使用するためには、割り込み処理関数を記述するとともに割り込みベクタテーブルに登録する必要があります。

この項では、割り込みベクタテーブルへの登録方法を説明します。

(1) 割り込みベクタテーブルへの登録

割り込み処理関数を記述する場合、サンプルスタートアッププログラム"sect308.inc"中の割り込みベクタテーブルを変更することにより、割り込み処理関数を登録します。

割り込みベクタテーブルの変更は以下の手順で行います。

割り込み処理関数名を指示命令".glb"で外部参照宣言する。

使用する割り込みのダミー関数名"dummy_int"を、割り込み処理関数名に変更する。

TA0割り込みに関数
"ta0()"を登録する

```
; - - - - -  
; variable vector section  
; - - - - -  
  
.section vector ; variable vector table  
.org VECTOR_ADR  
;  
.lword dummy_int ; vector (BRK)  
.org ( VECTOR_ADR + 32 )  
.lword dummy_int ; DMA0 (software int 8)  
.lword dummy_int ; DMA1 (software int 9)  
.lword dummy_int ; DMA2 (software int 10)  
.lword dummy_int ; DMA3 (software int 11)  
.glb _ta0  
.lword _ta0 ; TIMER A0 (software int 12)  
.lword dummy_int ; TIMER A1 (software int 13)  
.lword dummy_int ; TIMER A2 (software int 14)  
.lword dummy_int ; TIMER A3 (software int 15)  
.lword dummy_int ; TIMER A4 (software int 16)  
...
```

図3.8 割り込みベクタテーブル (sect308.inc)

3.1.5 割り込み処理関数の記述例

この項では、INT0 割り込みが発生するたびに"counter" の内容を 0 クリアし、INT1 割り込みが発生するたびに" counter " の内容をカウントアップさせるプログラムの記述例を示します。

(1) 割り込み処理関数の記述

```
/* プロトタイプ宣言 * * * * * */
void int0 ( void );
void int1 ( void );
#pragma INTERRUPT/F int0
#pragma INTERRUPT int1
/* * * * * * */
unsigned int counter;
void int0 ( void ) /* 高速割り込み関数 */
{
    counter = 0;
}
void int1 ( void ) /* 割り込み関数 */
{
    if ( counter < 9 ) {
        counter ++;
    }
    else {
        counter = 0;
    }
}

void main ( void )
{
    INT0IC = 0x07; /* 高速割り込み優先レベルの設定 */
    RLVL = 0x08; /* 高速割り込み指定 */
    asm( " LDC #_int0,VCT " ); /* ベクタレジスタの設定 */
    INT1IC = 0x01; /* 割り込み優先レベルの設定 */
    asm( " fset i " ); /* 割り込み許可 */
    while (1); /* 割り込み待ちループ */
}
```

図3.9 割り込み処理関数の記述例

(2) 割り込みベクタテーブルへの登録

図3.10に割り込みベクタテーブルの登録例を示します。

```
; - - - - -  
; variable vector section  
; - - - - -  
.section vector ; variable vector table  
.org VECTOR_ADR  
...  
.org ( VECTOR_ADR + 32 )  
.lword dummy_int ; DMA0 (software int 8)  
.lword dummy_int ; DMA1 (software int 9)  
.lword dummy_int ; DMA2 (software int 10)  
.lword dummy_int ; DMA3 (software int 11)  
.lword dummy_int ; TIMER A0 (software int 12)  
.lword dummy_int ; TIMER A1 (software int 13)  
.lword dummy_int ; TIMER A2 (software int 14)  
.lword dummy_int ; TIMER A3 (software int 15)  
.lword dummy_int ; TIMER A4 (software int 16)  
.lword dummy_int ; uart0 trance (software int17)  
.lword dummy_int ; uart0 receive (software int18)  
.lword dummy_int ; uart1 trance (software int19)  
.lword dummy_int ; uart1 receive (software int 20)  
.lword dummy_int ;TIMER B0 (software int 21)  
.lword dummy_int ;TIMER B1 (software int 22)  
.lword dummy_int ;TIMER B2 (software int 23)  
.lword dummy_int ;TIMER B3 (software int 24)  
.lword dummy_int ;TIMER B4 (software int 25)  
.lword dummy_int ; INT5 (software int 26)  
.lword dummy_int ; INT4 (software int 27)  
.lword dummy_int ; INT3 (software int 28)  
.lword dummy_int ; INT2 (software int 29)  
.glb _int1  
.lword _int1 ; INT1 (software int 30)  
.glb _int0  
.lword _int0 ; INT0 (software int 31)  
.lword dummy_int ; TIMER B5 (software int 32)  
...
```

図3.10 割り込みベクタテーブルへの登録例

3.2 アセンブルマクロ

NC308 ではアセンブリ言語命令の一部をC言語の関数として記述することができます（「アセンブルマクロ関数機能」）。

通常のC言語の記述ではNC308 が展開しないアセンブリ言語命令を直接C言語のプログラム上に記述できるので、プログラムのチューンナップが行いやすくなります。

この項では、アセンブルマクロ関数の記述方法と使用例について説明します。

3.2.1 アセンブルマクロ関数で記述できるアセンブリ言語命令

NC308 ではアセンブルマクロ関数で18種類のアセンブリ言語命令を記述することができます。

アセンブルマクロ関数名はアセンブリ言語命令を半角小文字で表します。演算時のビット長は"_b"、"_w"、"_l"で表します。アセンブルマクロ関数で記述できるアセンブリ言語命令を表3.1、表3.2に示します。

表3.1アセンブルマクロ関数で記述できるアセンブリ言語命令(1)

アセンブリ言語命令	アセンブルマクロ関数名	機能	書式
DADD	dadd_b	val1 と val2 の 10 進加算結果を返す	char dadd_b(char val1,char val2)
	dadd_w		int dadd_w(int val1,int val2)
DADC	dadc_b	val1 と val2 のキャリー付 10 進加算結果を返す。	char dadc_b(char val1,char val2)
	dadc_w		int dadc_w(int val1,int val2)
DSUB	dsub_b	val1 と val2 の 10 進減算結果を返す	char dsub_b(char val1, char val2);
	dsub_w		int dsub_w(int val1, int val2);
DSBB	dsbb_b	val1 と val2 のボーロー付き 10 進減算結果を返す	char dsbb_b(char val1, char val2);
	dsbb_w		int dsbb_w(int val1, int val2);
RMPA	rmpa_b	初期値 init、回数 count、乗数の格納されている先頭アドレスを p1、p2 として積和演算結果を返す	long rmpa_b(long init, int count, char *p1, char *p2);
	rmpa_w		long rmpa_w(long init, int count,int *p1, int *p2);
MAX	max_b	val1 と val2 の最大値を選択し結果を返す	char max_b(char val1, char val2);
	max_w		int max_w(int val1, int val2);
MIN	min_b	val1 と val2 の最小値を選択し結果を返す	char min_b(char val1, char val2);
	min_w		int min_w(int val1, int val2);

3. コンパイラ

表 3. 2アセンブラマクロ関数で記述できるアセンブリ言語命令(2)

アセンブリ言語命令	アセンブラマクロ関数名	機能	書式
SMOVB	smovb_b	転送番地 p1 から転送番地 p2 に count 回数分逆方向のストリング転送を行う	void smovb_b(char *p1, char *p2, unsigned int count);
	smovb_w		void smovb_w(int *p1, int *p2, unsigned int count);
SMOVF	smovf_b	転送番地 p1 から転送番地 p2 に count 回数分順方向のストリング転送を行う	void smovf_b(char *p1, char *p2, unsigned int count);
	smovf_w		void smovf_w(int *p1, int *p2, unsigned int count);
SMOVU	smovu_b	転送番地 p1 から転送番地 p2 に順方向で 0 が検出されるまでストリング転送を行う	void smovu_b(char *p1, char *p2);
	smovu_w		void smovu_w(int *p1, int *p2);
SIN	sin_b	固定の転送番地 p1 から転送番地 p2 に count 回数分順方向のストリング転送を行う	void sin_b(char *p1, char *p2, unsigned int count);
	sin_w		void sin_w(int *p1, int *p2, unsigned int count);
SOUT	sout_b	転送番地 p1 から転送番地 p2 に count 回数分順方向のストリング転送を行う	void sout_b(char *p1, char *p2, unsigned int count);
	sout_w		void sout_w(int *p1, int *p2, unsigned int count);
SSTR	sstr_b	ストアするデータ val 転送番地 p、転送回数 count としてストリングストアを行う	void sstr_b(char val, char *p,unsigned int count);
	sstr_w		void sstr_w(int val, int *p,unsigned int count);
ROL	rolc_b	val をキャリーを含めて 1 ビット左回転した結果を返す	unsigned char rolc_b(unsigned char val);
	rolc_w		unsigned int rolc_w(unsigned int val);
ROR	rorc_b	val をキャリーを含めて 1 ビット右回転した結果を返す	unsigned char rorc_b(unsigned char val);
	rorc_w		unsigned int rorc_w(unsigned int val);
ROT	rot_b	val を count 回数分回転した結果を返す	unsigned char rot_b(signed char count,unsigned char val);
	rot_w		unsigned int rot_w(signed char count,unsigned int val);
SHA	sha_b	val を count 回数分算術シフトした結果を返す	unsigned char sha_b(signed char count, unsigned char val);
	sha_w		unsigned int sha_w(signed char count, unsigned int val);
	sha_l		unsigned long sha_l(signed char count, unsigned longval);
SHL	shl_b	val を count 回数分論理シフトした結果を返す	unsigned char shl_b(signed char count, unsigned char val);
	shl_w		unsigned int shl_w(signed char count, unsigned int val);
	shl_l		unsigned long shl_l(signed char count, unsigned longval);

3.2.2 アセンブラマクロ関数"dadd_b"を使用した 10 進加算

NC308 のアセンブラマクロ関数を呼び出して使用する場合は、アセンブラマクロ関数定義ファイル"asmmacro.h" を必ずインクルードしてください。

アセンブラマクロ関数"dadd_b" を使用した 10 進加算の例を図 3.11 に示します。

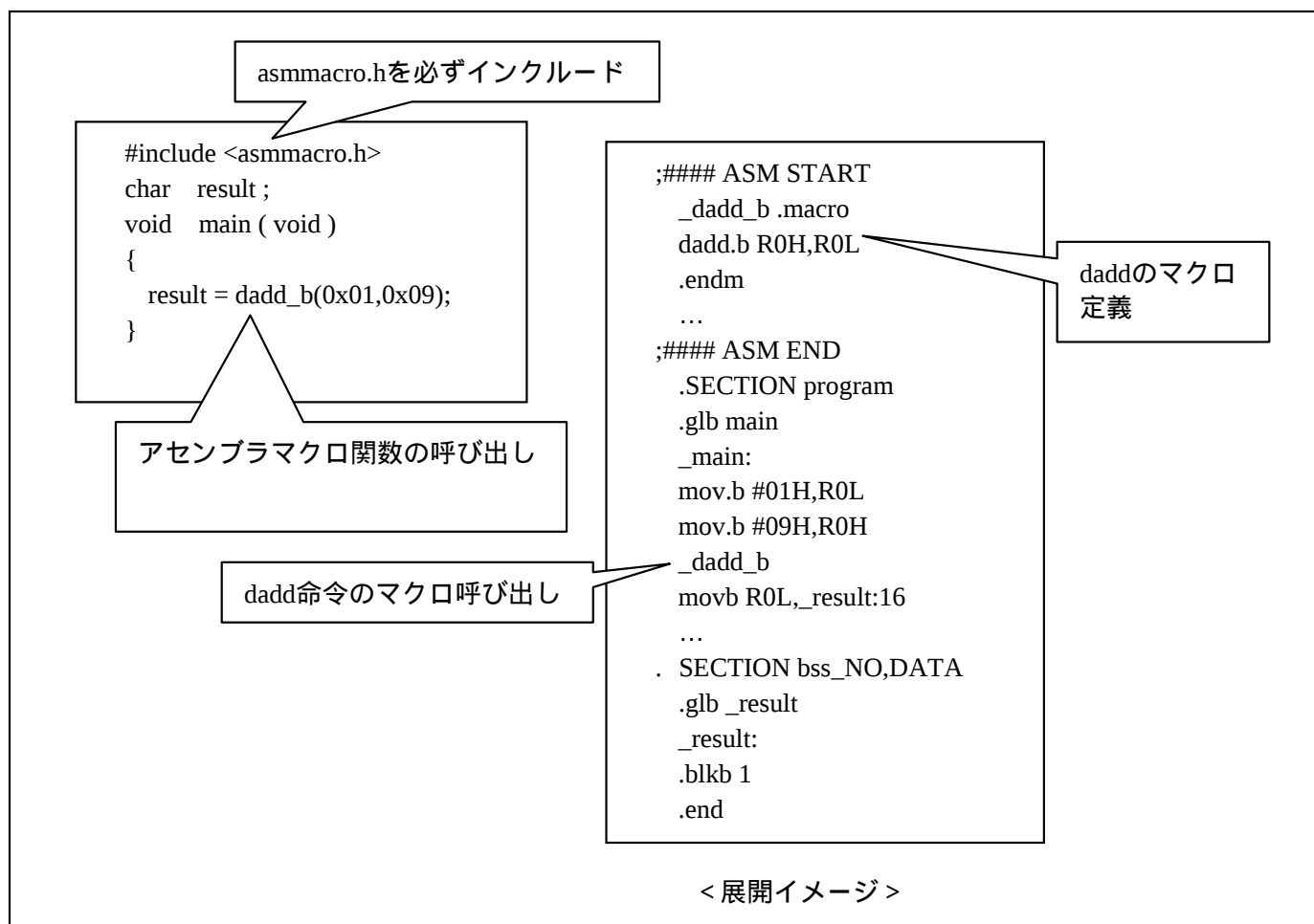


図3.11 アセンブラマクロ命令 “ dadd_b”を使用した10進加算

3.2.3 アセンブラマクロ関数"smovf_b"を使用したストリング転送

アセンブラマクロ関数"smovf_b" を使用したストリング転送の例を図 3.12 に示します。

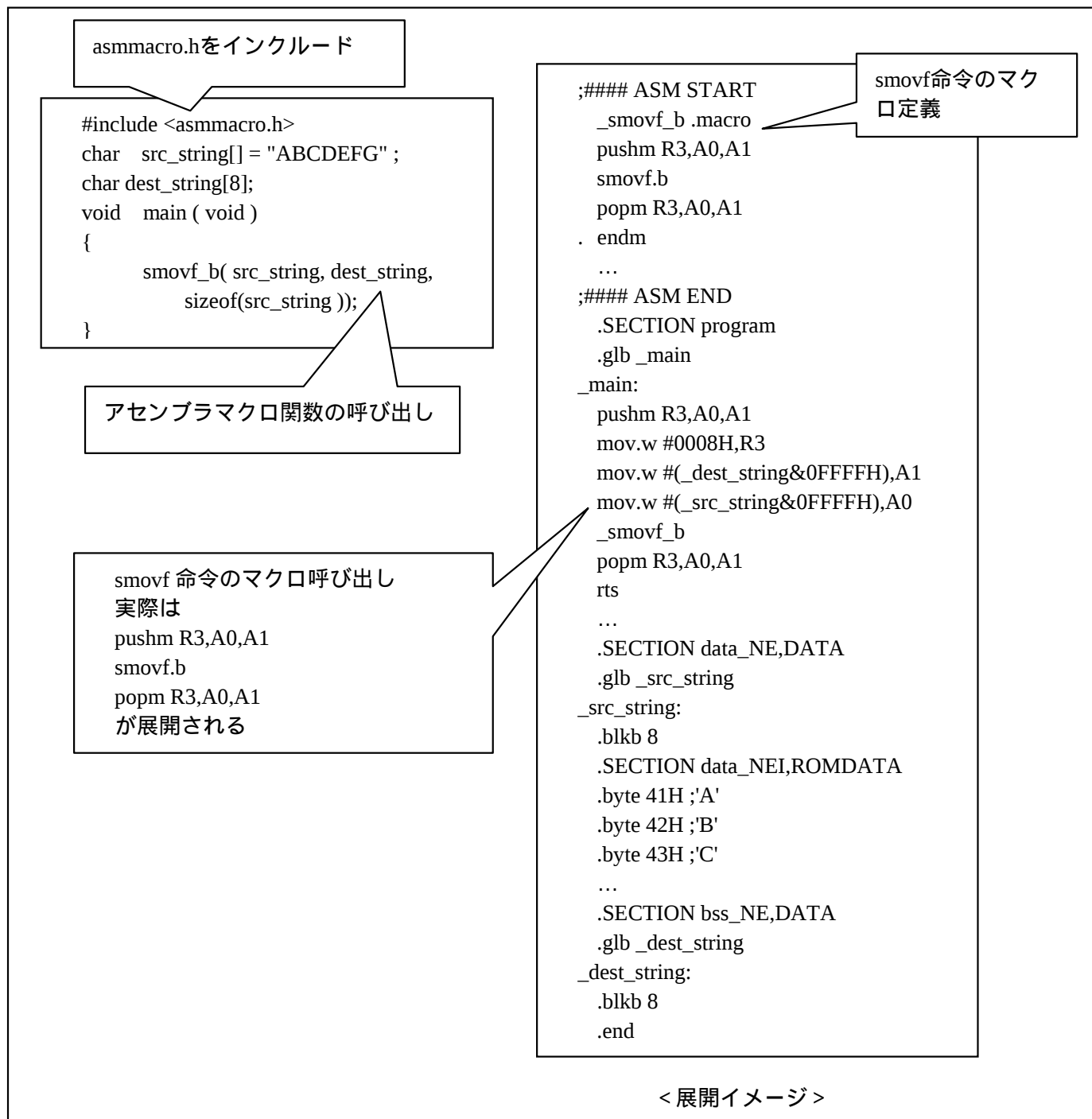


図3.12 アセンブラマクロ関数smovf_bを使用したストリング転送

3.2.4 アセンブラマクロ関数"rmpa_w"を使用した積和演算

アセンブラマクロ関数"rmpa_w" を使用したストリング転送の例を図 3.13 に示します。

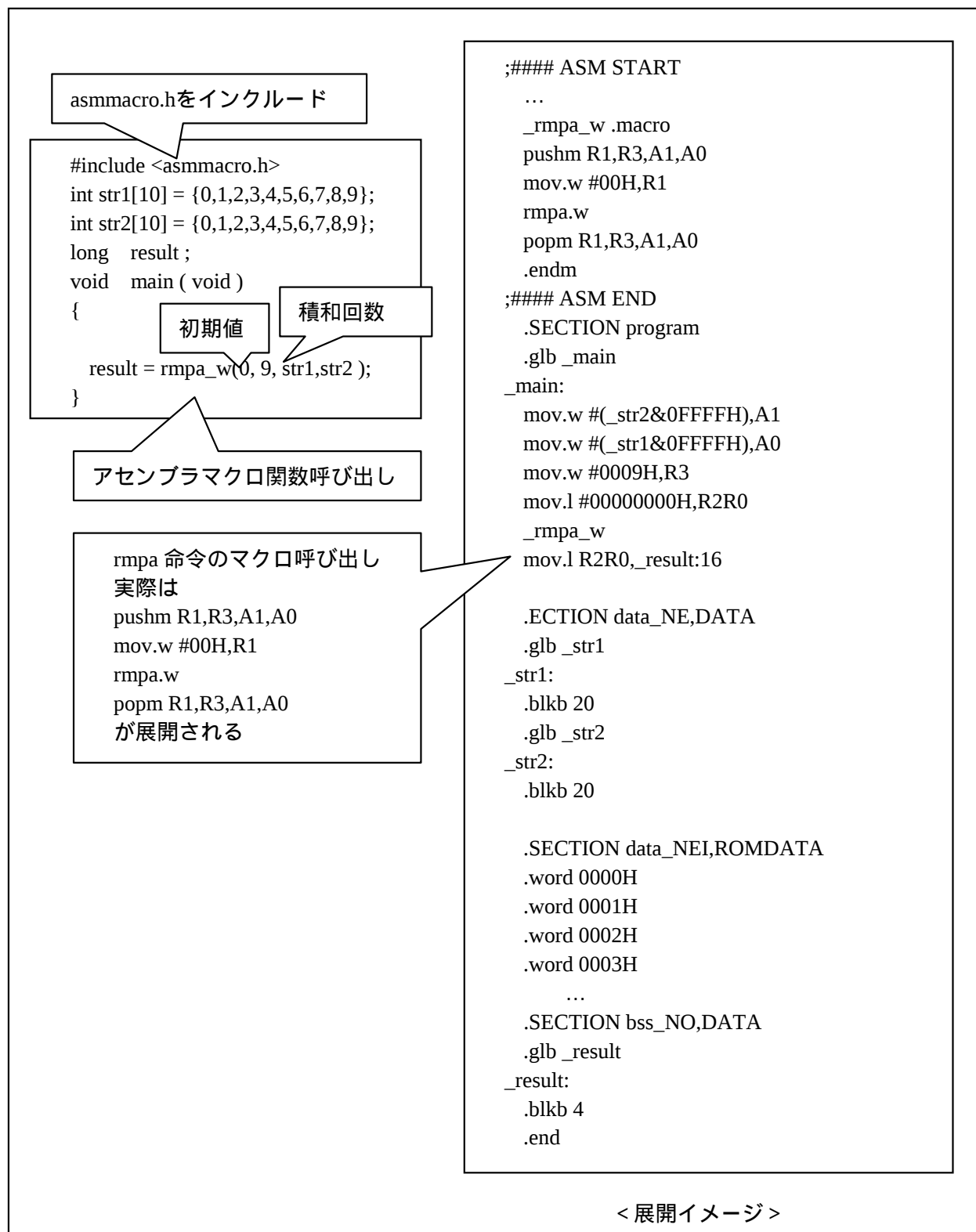


図3.13 アセンブラマクロ"rmpa_w"を使用した積和演算

3.2.5 #pragma __ASMMACRO

NC308 では#pragma __ASMMACRO を使用することで任意のアセンブラの命令列をアセンブラマクロ関数にすることができます。

[機能]

アセンブラのマクロで定義した関数を宣言します。

[書式]

#pragma __ASMMACRO 関数名(レジスタ名,...)

[規定]

- 1.本機能による宣言を行う前に、関数のプロトタイプ宣言を行ってください。アセンブラマクロ関数は、必ず static 宣言してください。
- 2.引数のない関数は宣言できません。引数はレジスタ渡しになります。引数の型と合致するレジスタを指定してください。(#pragma PARAMETER に準じます)
- 3.宣言した関数名の先頭にアンダスコア() を付加したマクロ名で、アセンブラマクロを定義してください。
4. 戻り値の返し方は、関数呼び出し規則に従い、以下ようになります。集合体(構造体／共用体)を戻り値とすることはできません。
char 型、_Bool型: R0L float 型: R2R0
int/short 型: R0 double 型: R3R2R1R0
long 型: R2R0 long long型: R3R1R2R0
- 5.アセンブラマクロ内で内容が変更されるレジスタは、アセンブラマクロの先頭で退避して、復帰直前に復帰してください。(戻り値の格納レジスタの退避・復帰は不要です)

[使用例]

```
static long mul( int, int ); /* 必ず static にしてください。 */
#pragma __ASMMACRO mul( R0, R2 )
#pragma ASM
_mul .macro
mul.w R2,R0 ;戻り値はR2R0 で返されます。
.endm
#pragma ENDASM
long l;
void test_func( void )
{
    l = mul( 2, 3 );
}
```

図3.14 アセンブラマクロの使用例

3.3 コード効率向上のための pragma, オプション

表3.3 コード効率向上のためのpragma, オプション一覧

項番	タイトル	内容
3.3.1	#pragma SBADATA	変数を SB 相対アドレッシングでアクセスします。
3.3.2	#pragma SB16DATA	変数を 2 バイト SB 相対アドレッシングでアクセスします。
3.3.3	#pragma BIT	16 ビット絶対アドレッシングモードによる 1 ビット操作命令を生成します。
3.3.4	#pragma SPECIAL	ジャンプサブルーチン命令を 4 バイトから 2 バイトへ圧縮します。
3.3.5	-fjsrw	JSR.W 命令で関数を呼び出します。
3.3.6	-OR	ROM 効率のための最大限の最適化を行います。
3.3.7	-fno_align	関数の先頭アドレスのアライメントを行いません。
3.3.8	-Wno_used_function	使用していない関数に対しウォーニングを出します。

3.3.1 #pragma SBADATA

指定した変数のアクセスを SB レジスタを用いた相対アドレッシングモードで行います。頻繁にアクセスする変数を SB 相対アドレッシングに変更することによりコード効率を向上できます。SB 相対アドレッシングにした変数はSBADATA 属性のセクションに割り付けられSB レジスタに格納されているSBADATA 属性のセクションの先頭アドレスからのオフセットで参照されます。そのため、アドレスをロードして参照するコードよりもコンパクトに展開され ROM 効率の向上に役立ちます。書式は以下のようになります。

#pragma SBADATA 変数名

SB レジスタを用いたアドレッシングは SB レジスタから 256 バイトまでの領域をアクセスでき、アドレス指定が 1 バイトで済みます。使用頻度の高い変数を SB レジスタを用いたアドレッシングにすることで ROM 領域を節約できます。

使用前	使用後
int a; a=1;	#pragma SBADATA a int a; a=1;
.GLB __SB__ .SB __SB__ .FB 0 ... mov.w #0001H,_a	.GLB __SB__ .SB __SB__ .FB 0 .SBSYM_a ... mov.w #0001H,_a

図3.15 SBADATAを用いたアドレッシングモードの使用例

3.3.2 #pragma SB16DATA

SBADATA が SB レジスタから 1 バイトのオフセットのアドレッシングに対し SB16DATA は 2 バイトのオフセットのアドレッシングです。SB レジスタから 64 K バイトまでの領域をアクセスでき、アドレス指定が 2 バイトで済みます。使用頻度の高い変数を #pragma SB16DATA に指定することで ROM 領域を節約できます。書式は以下のようになります。

#pragma SB16DATA 変数名

3. コンパイラ

使用前	使用后
int a; a=1;	#pragma SB16DATA a int a; a=1;
.GLB __SB__ .SB __SB__ .FB 0 ... mov.w #0001H,_a	.GLB __SB__ .SB __SB__ .FB 0 .SBSYM16 _a ... mov.w #0001H,_a

図3.16 SB16DATAを用いたアドレッシングの使用例

3.3.3 #pragma BIT

指定された外部変数が、16 ビット絶対アドレッシングモードによる 1 ビット操作命令を使用できる領域(00000H ~ 01FFFH 番地)にあることを宣言します。これにより 16 ビット絶対アドレッシングモードによる 1 ビット操作命令を生成することが出来ます。(* この機能は NC30WA のみです。)

書式は以下のようになります。

#pragma BIT 変数名

使用前	使用后
int sym sym=0x01 sym	#pragma BIT sym int sym sym=0x01 sym
or.w #01H,_sym	bset 0,_sym

図3.17 #pragma BITを用いたビット演算

3.3.4 #pragma SPECIAL

スペシャルページはジャンプサブルーチン命令を JSR.A _func の 4 バイトから JSRS #番号の 2 バイトへ圧縮します。これにより ROM 領域を節約することができます。

書式は以下のようになります。

#pragma SPECIAL [/C] 呼び出し番号 関数名()

#pragma SPECIAL [/C] 関数名(vect=呼び出し番号)

#pragma SPECIAL で宣言した関数は、スペシャルページベクタテーブルの各テーブルに設定した番地に 0FF0000H を加算したアドレスに配置されたものとして、スペシャルページサブルーチン呼び出しが行われます。宣言時に以下のスイッチを指定できます。

[/C]

宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。

宣言時に呼び出し番号を指定できます。

呼び出し番号を指定し、コンパイルオプション -fmake_special_table(-fMST)を指定してコンパイルすることで、スペシャルページベクタテーブルを自動的に生成することができます。

#pragma SPECIAL で宣言した関数は、 program_S セクションに配置されます。
 program_S セクションは、必ず 0FF0000H から 0FFFFFFH の領域に配置してください。
 呼び出し番号は、18 から 255 までです。また 10 進数のみ指定可能です。 #pragma SPECIAL で宣言した関数の先頭アドレスには、ラベルとして"__SPECIAL_呼び出し番号:"が出力されます。スタートアップファイルで、スペシャルページサブルーチンテーブルにこのラベルを設定してください。なお、-fmake_special_table(-fMST) オプションを指定した場合、上記設定は必要ありません。
 同じ関数に、異なる呼び出し番号を記述した場合は、後に宣言された呼び出し番号が有効になります。

例)

```
#pragma SPECIAL func(vect=20)
#pragma SPECIAL func(vect=30) // 呼び出し番号 30 が有効
```

関数が定義されているファイル、関数の呼び出しを行っているファイルが別のファイルの場合、その両方のファイルに本宣言を行ってください。

使用前	使用后
<pre>void func(unsigned int, unsigned int); void main() { int i, j; i = 0x7FFD; j = 0x007F; func(i, j); }</pre>	<pre>#pragma SPECIAL 20 func() void func(unsigned int, unsigned int); void main() { int i, j; i = 0x7FFD; j = 0x007F; func(i, j); }</pre>
<pre>push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsr \$func</pre>	<pre>push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsrs #20</pre>

図3.18 #pragma SPECIAL宣言の使用例

3.3.5 -fjsrw

本コンパイラでは、ファイル外で定義された関数を呼び出す場合には、"JSR.A"命令で呼び出しを行います。しかし、プログラムサイズがあまり大きくない場合、大半の関数が"JSR.W"命令で呼び出せる場合があります。

このような場合にオプション"-fJSRW"を指定してコンパイルを行い、リンク時にエラーの発生した関数のみを"#pragma JSRA 関数名"を用いて宣言することにより ROM 領域の圧縮が期待できます。

" -OGJ " オプションを使用すると、リンク時に最適な jmp 命令を選択します。

3. コンパイラ

C ソース	-fjsrw なし	-fjsrw あり
<pre>/*file 1*/ void f() {} /* file 2*/ int main() { f(); }</pre>	<pre>.GLB __SB__ .SB __SB__ .FB 0 ... jsr _f</pre>	<pre>.OPTJ JSRW .GLB __SB__ .SB __SB__ .FB 0 ... jsr _f</pre>

図3.19 -fjsrwの使用例

3.3.6 -OR

速度は低下する場合がありますが、ROM 領域を最小にする最大限の最適化を行います。このオプションは、-g オプション、-O オプションと同時に指定することができます。このオプションを使用した場合、ソース行情報の一部を変更する最適化を行う可能性があります。このため、デバッグ時に動作が異なって見える場合があります。ソース行情報を変更したくない場合、-Ono_break_source_debug (-ONBSD) オプションを使用して最適化を抑止してください。

図 3.20-OR による最適化の一例です。共通式をまとめて ROM 領域を圧縮しています。

C ソース	最適化なし	最適化あり
<pre>If(b==1){ sub(); return a; } else if(b==2){ sub() return a; } else { return 0; }</pre>	<pre>;;# # C_SRC : if(b==1) cmp.w #0001H,_b:16 jne L1 ;;# # C_SRC : sub(); jsr _sub ;;# # C_SRC : return a; mov.w _a:16,R0 rts ;;# # C_SRC : else if(b==2) L1: cmp.w #0002H,_b:16 jne L11 ;;# # C_SRC : sub(); jsr _sub ;;# # C_SRC : return a; mov.w _a:16,R0 rts ;;# # C_SRC : else L11: ;;# # C_SRC : return 0; mov.w #0000H,R0 rts</pre>	<pre>;;# # C_SRC : if(b==1) mov.w _b:16,R0 cmp.w #0001H,R0 jne L5 ;;# # C_SRC : sub(); L31: jsr _sub ;;# # C_SRC : return a; mov.w _a:16,R0 rts ;;# # C_SRC : else if(b==2) L5: cmp.w #0002H,R0 jeq L31 ;;# # C_SRC : return 0; mov.w #0000H,R0 rts</pre>

図3.20-ORによる最適化の例 共通式をまとめる最適化

3.3.7 -fno_align

関数の先頭アドレスのアライメントを行いません。関数の先頭に挿入される.align が省略され ROM 領域を節約できます。

C ソース	-fno_align なし	-fno_align あり
<pre>int f() { return 0; }</pre>	<pre>## # FUNCTION f ## # ARG Size(0) Auto Size(0) Context Size(4) .SECTION program, CODE, ALIGN .inspect 'U', 2, "program", "program", 0 .file 'C:/Hew3/fno_align/fno_align/fno_align. n.c' .type 256,'x',16,0 .func 'f','G',0,256,_f,0 .inspect 'F','s','f',"_f",'G',4 .align ## # C_SRC : { .glb _f _f:</pre>	<pre>## # FUNCTION f ## # ARG Size(0) Auto Size(0) Context Size(4) .SECTION program, CODE .inspect 'U', 2, "program", "program", 0 .file 'C:/Hew3/fno_align/fno_align/fno_align. c' .type 256,'x',16,0 .func 'f','G',0,256,_f,0 .inspect 'F','s','f',"_f",'G',4 ## # C_SRC : { .glb _f _f:</pre>

図3.21 -fno_alignの使用例

3.3.8 -Wno_used_function

未使用のグローバル関数をリンク時に表示します。未使用の関数を削除することで ROM 領域を節約できます。

C ソース	ウォーニングメッセージ
<pre>void f() { } int main() { }</pre>	<pre>C:\Hew3\test2\test2\test2.c(20) : Warning (ln308): Global function 'f' is never used</pre>

図3.22 -Wno_used_functionの使用例

3.4 速度向上のための pragma、オプション

表 3.4 速度向上のためのpragma,オプション一覧

項番	タイトル	内容
3.4.1	#pragma STRUCT	構造体のアラインをとりアクセス速度を向上させます。
3.4.2	-Ostack_frame_align	スタックフレームのアラインをとりアクセス速度を向上させます。
3.4.3	-OS	スピード向上を最大限に行う最適化をします。
3.4.4	-Oloop_unroll[=回数]	ループの展開を行います。
3.4.5	-Ofloat_to_inline	浮動小数点の実行時ルーチンをインライン展開します。

3.4.1 #pragma STRUCT

構造体のアラインメントをとりアクセス速度を向上させます。

書式 1.#pragma STRUCT 構造体タグ名 unpack

2.#pragma STRUCT 構造体タグ名 arrange

本コンパイラでは、構造体はパックされます。例えば、図 3.23 に示す構造体のメンバは、宣言された順にパディング(透き間)を入れずに配置されます。

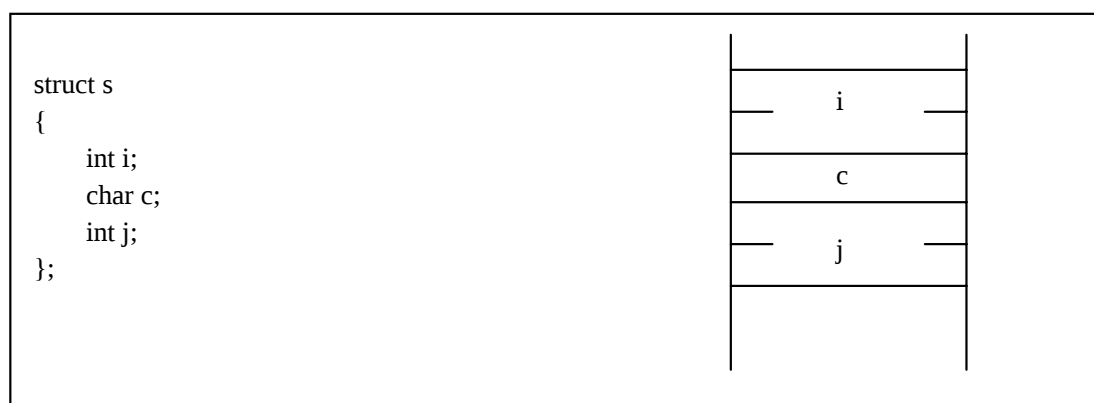


図3.23 構造体メンバの配置例 (1)

本コンパイラでは、拡張機能として構造体メンバの配置を制御することができます。図 3.23 に示した構造体を#pragma STRUCTunpack でパックを禁止した場合のメンバの配置例を図 3.24 に示します。

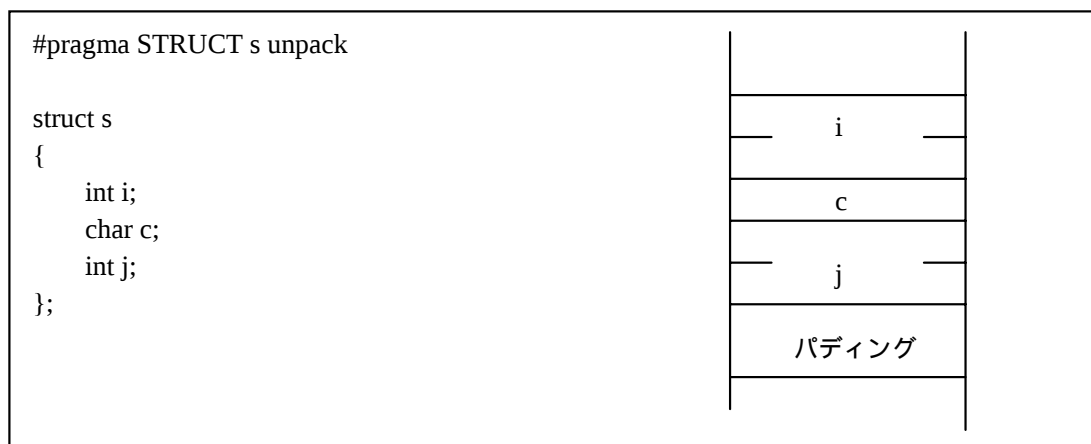


図3.24 構造体メンバの配置例（2）

図 3.24 に示したように、構造体メンバのサイズの合計が奇数バイトの場合、#pragmaSTRUCTUnpackを用いることにより最後のメンバ配置位置の後に、1 バイトのパディングが入ります。したがって、#pragma STRUCTUnpack でパックを禁止した場合の構造体は、すべて偶数バイトのサイズとなります。

本コンパイラでは、拡張機能として構造体の偶数サイズのメンバを先に配置し、奇数サイズのメンバを後に配置することができます。図 3.23 に示した構造体を#pragma STRUCT arrange で配置を並び替えた場合の配置例を図 3.25 に示します。

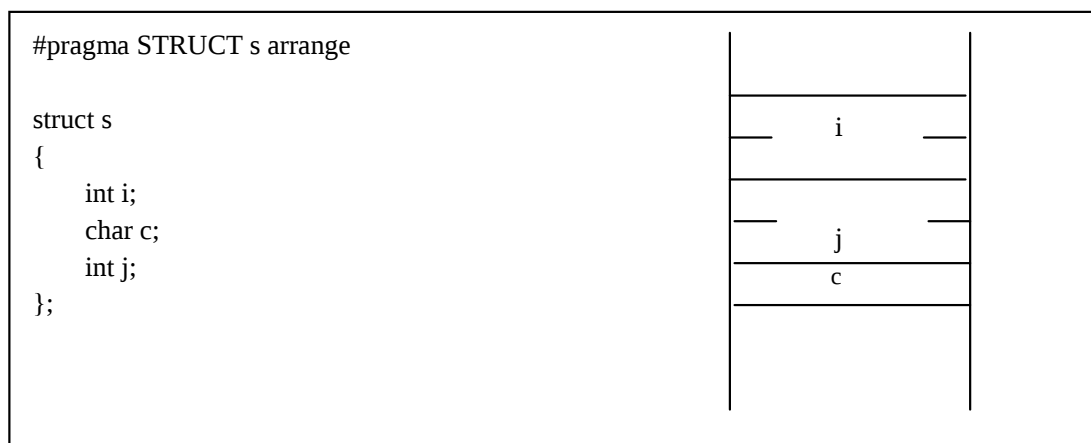


図3.25 構造体メンバの配置例（3）

3. コンパイラ

#pragma STRUCT unpack と #pragma STRUCT arrange を併用することで偶数サイズを持つメンバがアラインされた状態になります。

デフォルト	unpack
<pre> struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; } </pre>	<pre> #pragma STRUCT A unpack struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; } </pre>
<pre> ;## # C_SRC : a.a=1; mov.w #0001H,-10[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-7[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-3[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-2[FB] ; b </pre>	<pre> ;## # C_SRC : a.a=1; mov.w #0001H,-12[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-10[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-9[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-4[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b </pre>

図3.26 #pramga STRUCT の使用例 (1)

arrange	arrange+unpack
<pre>#pragma STRUCT A arrange struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; }</pre>	<pre>#pragma STRUCT A arrange #pragma STRUCT A unpack struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; }</pre>
<pre>;;# # C_SRC : a.a=1; mov.w #0001H,-10[FB] ; a ;;# # C_SRC : a.b=2; mov.b #02H,-6[FB] ; a ;;# # C_SRC : a.c=3; mov.w #0003H,-8[FB] ; a ;;# # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;;# # C_SRC : b.b=5; mov.b #05H,-1[FB] ; b ;;# # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b</pre>	<pre>;;# # C_SRC : a.a=1; mov.w #0001H,-12[FB] ; a ;;# # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;;# # C_SRC : a.c=3; mov.w #0003H,-10[FB] ; a ;;# # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;;# # C_SRC : b.b=5; mov.b #05H,-2[FB] ; b ;;# # C_SRC : b.c=6; mov.w #0006H,-4[FB] ; b</pre>

図 3.27 #pragma STRUCT の使用例 (2)

3.4.2 -Ostack_frame_align

偶数サイズの auto 変数が奇数アドレスに配置された場合は、偶数アドレスに配置された場合よりもメモリアクセスが 1 サイクル多く必要になります。本オプションを指定すると、偶数サイズの auto 変数を偶数アドレスに配置するようにアライメントを行うため、メモリアクセスを高速に行うことができます。(* 本オプションは NC30WA のみです。)

3. コンパイラ

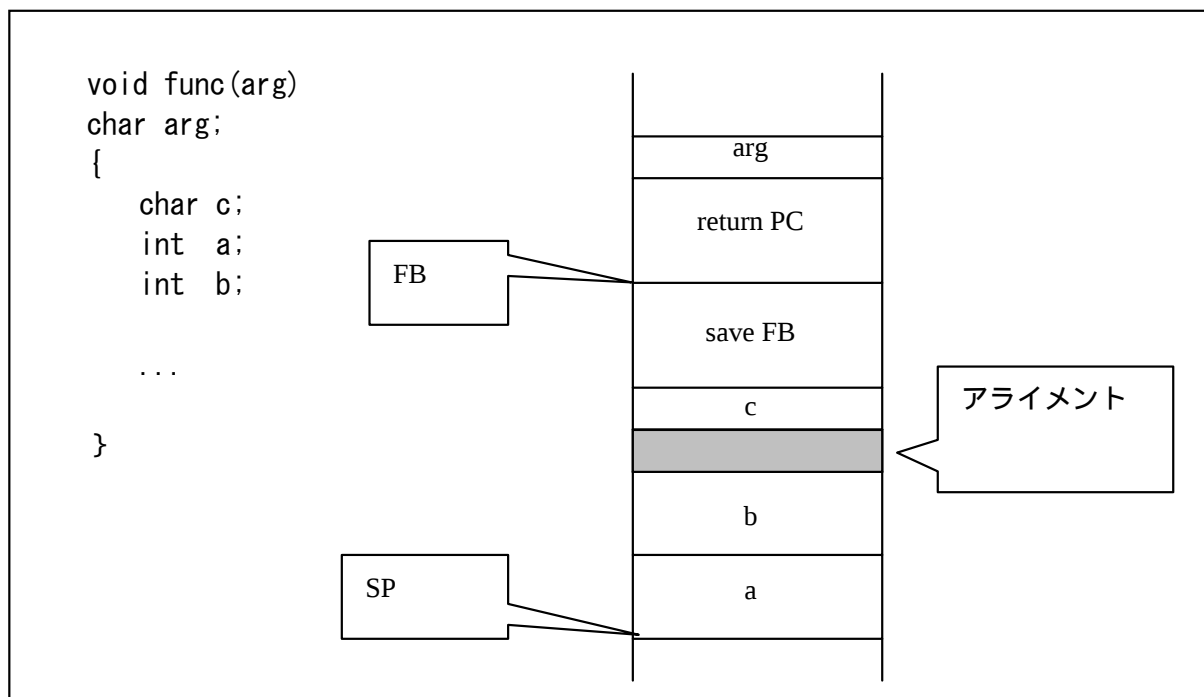


図3.28 -Ostack_frame_alignのアライメントの例

Cソース	-Ostack_frame_align なし	-Ostack_frame_align あり
<pre> void f() { int a; int b; a=1; b=2; ... } </pre>	<pre> ;## # C_SRC : a=1; mov.w #0001H,-2[FB] ; a ;## # C_SRC : b=2; mov.w #0002H,-4[FB] ; b </pre>	<pre> ;## # C_SRC : a=1; mov.w #0001H,-5[FB] ; a ;## # C_SRC : b=2; mov.w #0002H,-3[FB] ; b </pre>

図3.29 -Ostack_frame_alignの例

3.4.3 -OS

ROM 使用量は増大する場合がありますが、速度重視の最大限の最適化を行います。このオプションは、-g オプション、-O オプションと同時に指定することができます。

C ソース	最適化なし	最適化あり
<pre>for(i=0;i<100;i++) a[i]=l*4;</pre>	<pre>;;# # C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: ;;# # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ;;# # C_SRC : a[i]=l*4; mov.w _l:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 jmp L1 L5:</pre>	<pre>mov.w #0000H,_i:16 mov.w _l:16,R0 shl.w #2,R0 L3: .line 26 ;;# # C_SRC : a[i]=l*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L3</pre>

図3.30 -OSによる最適化の例

3.4.4 -Oloop_unroll[=回数]

ループ文を回せずに、ループ回数分コードを展開します。"ループ回数"は省略可能、省略時は最大5回のループ文が対象となります。分岐やループカウンタの計算がなくなり実行速度が向上します。ただし ROM 効率は悪くなります。

C ソース	最適化なし	最適化あり
<pre>for(i=0;i<3;i++) { a[i]=i; }</pre>	<pre>;;# # C_SRC : for(i=0;i<3;i++) mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC : for(i=0;i<3;i++) cmp.w #0003H,-2[FB] ; i jge L5 ;;# # C_SRC : a[i]=i; indexwd.w -2[FB] ; i mov.w -2[FB],_a:16 ; i add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>	<pre>mov.w #0000H,-2[FB] ; i mov.w #0002H,A0 mul.w #0000H,A0 mov.w A0,A0 mov.w #0000H,_a:16[A0] mov.w #0001H,-2[FB] ; i mov.w #0002H,A0 mul.w #0001H,A0 mov.w A0,A0 mov.w #0001H,_a:16[A0] mov.w #0002H,-2[FB] ; i ;;# # C_SRC : a[i]=i; mov.w #0002H,R0 indexwd.w R0 mov.w #0002H,_a:16 mov.w #0003H,-2[FB] ; i</pre>

図3.31 ループの展開例

3. コンパイラ

3.4.5 -Ofloat_to_inline

浮動小数点のランタイムライブラリをインライン展開し、浮動小数点演算処理を高速化します（比較、乗算のみ）。本機能は M32C/80 シリーズのみ有効です。使用時はコンパイルオプション “-M82” とあわせて指定してください。

ソースコード	-Ofloat_to_inline なし	-Ofloat_to_inline あり
float f; long l; l=f;	push.l _f:16 jsr.a __f4toi4 add.l #04H,SP mov.l R2R0,_l:16	;;# C_SRC : l=f; mov.w _f+2:16,R0 mov.w _f:16,R2 btst 7,R0H scc R3 mov.w R0,R1 shl.w #-07H,R1 and.w #0ffH,R1 xchg.w R0,R2 and.w #07fH,R2 mov.w R1,R1 jne ?+ mov.l R2R0,R2R0 jeq M1 ?: btst 7,R1L jc ?+ cmp.w #07fH,R1 jne M1 ?: add.w #062H,R1 tst.w #0ff00H,R1 jeq M2 mov.w R3,R3 jeq ?+ mov.l #80000000H,R2R0 jmp M3 ?: mov.l #7fffffffH,R2R0 jmp M3 M2: dec.w R1 or.w #0080H,R2 shlnc.l #8H,R2R0 ?: inc.w R1 cmp.w #0100H,R1 jeq M4 shlnc.l #-1,R2R0 jmp ?- M4:

		<pre> cmp.w #1,R3 jne M3 not.w R0 not.w R2 add.l #01H,R2R0 jmp M3 M1: mov.l #0,R2R0 M3: </pre>
--	--	--

図2.32 -Ofloat_to_inline の最適化例

3.4.6 -Ostatic_to_inline

static 宣言された関数（ static 関数 ）を inline 宣言されている関数（ inline 関数 ）として扱い、inline 展開したアセンブルコードを生成します。

以下の条件を満たした場合に、static 関数を inline 関数として扱い、inline 展開したアセンブルコードを生成します。

- (1) 関数呼び出しの前に、実体が記述されている static 関数を対象とします。
（関数の呼び出しと、その関数の実体が、同じソースファイル内に記述されていなければなりません。）
（-Ofoward_function_to_inline オプションを指定した場合は、本条件を無視してください。）
- (2) 対象となる static 関数に対して、プログラム中で、アドレス取得を行っていない場合。
- (3) 対象となる static 関数を再帰呼び出ししていない場合。
- (4) コンパイラのアセンブルコード出力において、フレーム（auto 変数等の確保）の構築が、行われない場合。（対象となる関数の記述内容、別の最適化オプションとの併用により、フレーム構築の有無の状況は、異なります。）
（-Ofoward_function_to_inline オプションを指定した場合は、本条件を無視してください。）

C ソース	-Ostatic_to_inline なし	-Ostatic_to_inline あり
<pre> static int f(int a,int b) { return a+b; } int c; void main() { c=f(2,3); } </pre>	<pre> push.w #0003H mov.w #0002H,R0 jsr \$f add.l #02H,SP mov.w R0,_c:16 </pre>	<pre> mov.w #0005H,_c:16 </pre>

図3.33 -Ostatic_to_inlineの使用例

3.5 ROM 領域削減、速度向上のための pragma, オプション

表3.5 ROM領域削減、速度向上のためのpragma,オプション一覧

項番	タイトル	内容
3.5.1	-O[1-5]	最適化を行います。
3.5.2	-Osp_adjust	スタックポインタ補正をまとめて行う最適化を行います。
3.5.3	-fuse_DIV	割り算を div 命令で計算します。
3.5.4	-Wno_used_argument	使用していない引数に対しウォーニングを出します。
3.5.5	-fsmall_array	配列の添え字の計算を 16 ビットで行います。
3.5.6	-fdouble_32	double を float として扱います。

3.5.1 -O[1-5]

速度及びROM領域ともに効果のある最大限の最適化を行います。このオプションは、-gオプションと同時に指定することができます。数字(レベル)を指定しない場合は、-O3と同じです。

-O1: -O3, -Ono_bit, -Ono_break_source_debug, -Ono_float_const_fold, -Ono_stdlib を有効にしたものと同等です。

-O2: -O1 と同じです。

-O3: 速度及び ROM 領域ともに効果のある最大限の最適化を行います。

-O4: -O3 に加え、-Oconst を有効にします。

-O5: 共通部分式の最適化(-OR 同時指定時)、文字列転送, 比較 (-OS 同時指定時)などを強化した最大限の最適化を行います。

但し以下の条件を満たす場合、正常なコードを出力できない可能性があります。

- ・異なるポインタ変数が同時に同じメモリ位置を指す
- ・それらの変数を同一関数内で使用する

例)

```
int a=3;
int *p=&a;

test()
{
    int b;
    *p=9;
    a=10;
    b=*p; //最適化により”p”を”9”に置き換えてしまう。
    print(“b=%d(expect b=10)\n”,b);
}
```

実行結果)

b=9(expect b=10)

図3.34-O5指定時の誤動作を起こすソースの例

注意事項： SFR 領域のレジスタへの書き込み、読み出しには、ビット操作命令（BTSTC、BTSTS）を使用することはできません。

本コンパイラでは、最適化オプション（-O5）を使用した場合、アセンブラコードに対して、ビット操作命令（BTSTC、BTSTS）を生成する場合があります。以下の例のような記述を行い、最適化オプション（-O5）を使用してコンパイルした場合、割り込み要求ビットの判定が正常に行われず動作は不定となります。

```
[例：最適化オプションを使用してはならない C ソース]
#pragma ADDRESS TA0IC 006Ch /* M16C/80 タイマ A0 割り込み制御レジスタ */
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 */
}
```

図3.35 最適化オプションを使用してはならない例

もし、SFR 領域に対してビット操作命令（BTSTC、BTSTS）が出力されていることが確認されたら、以下のような対策を行った上で、コンパイルしてください。いずれの場合にも、生成されたコードに問題が無いことを必ず確認してください。

- ” -O5 ” 以外の最適化オプションを使用する。
- ASM 関数を使用してプログラム中に直接命令を記述する。
- -Ono_asmopt(もしくは-ONA)オプションを追加指定する。

3.5.2 -Osp_adjust

関数呼び出し後のスタック補正コードをまとめる最適化を行います。通常は関数呼び出し毎に、関数の引数の領域を解放するために、スタックポインタを補正する処理をします。本オプションを使用することにより、このスタックポインタの補正を関数の呼出し毎ではなく、まとめて行うようにします。オプション-Osp_adjust により ROM 領域を削減し、かつ速度を向上することができます。ただし、使用するスタック量が多くなる可能性があります。

3. コンパイラ

C ソース	-Osp_adjust なし	-Osp_adjust あり
<pre>main() { f(1.1); g(1.1); }</pre>	<pre>_main: ;## # C_SRC : f(1.1); push.l #3ff19999H push.l #9999999aH jsr _f add.l #08H,SP ;## # C_SRC : g(1.1); push.l #3ff19999H push.l #9999999aH jsr _g add.l #08H,SP ;## # C_SRC : } rts</pre>	<pre>_main: ;## # C_SRC : f(1.1); push.l #3ff19999H push.l #9999999aH jsr _f ;## # C_SRC : g(1.1); push.l #3ff19999H push.l #9999999aH jsr _g add.l #010H,SP ;## # C_SRC : } rts</pre>

図3.36 -Osp_adjustの使用例

3.5.3 -fuse_DIV

除算に対する生成コードを変更します。

除算時に、被除数が 4byte 値、除数が 2byte 値、演算結果が 2byte 値の場合や、被除数が 2byte 値、除数が 1byte 値、演算結果が 1byte 値の様な演算を行う場合にマイクロコンピュータの div.w (divu.w) 及び div.b(divu.b) 命令を生成します。

本オプションを指定した場合に、除算結果が over flow すると ANSI の規定とは異なる動作になります。M16C の div 命令は演算結果が over flow した場合には、結果は不定になります。

このため NC308 でデフォルトでコンパイルを行った場合には、結果を保証するために、被除数が 4byte、除数が 2byte、演算結果が 2byte のような場合にもランタイムライブラリを呼び出します。

ソースプログラム	デフォルト	-fuse_DIV 使用時
<pre>int k,j; long l; k=l/j;</pre>	<pre>mov.w _j:16,R1 exts.w R1 push.l R3R1 mov.l _l:16,R2R0 glb __i4div jsr.a __i4div add.l #4H,SP mov.w R0,_k:16</pre> オーバーフローを考慮した コードを出力	<pre>mov.l _l:16,R2R0 div.w _j:16 mov.w R0,_k:16</pre> div 命令を使用

図3.37 -fuse_DIVの使用例

3.5.4 -Wno_used_argument

引数を持つ関数を定義した場合に、使用していない引数に対してウォーニングを出力します。ウォーニングをもとにソースプログラムを修正することでメモリを節約し実行速度を向上させることができます。

C ソース	ウォーニングメッセージ
<pre>int f(int a,int b,int c) { return a+c; }</pre>	C:\Hew3\test1\test1.c(68) : [Warning(ccom)] function "f()" has no-used argument(b).

図3.38 -Wno_used_argumentの実行例

3.5.5 -fsmall_array

コンパイル時に総サイズが不明の far 型の配列を参照する場合、その総サイズが 64K バイト以内であると仮定し添字の計算を 16 ビットで行います。デフォルトでは、far 型配列の要素を参照場合に、配列のサイズが不明であれば、添字を 32 ビットで計算します。これは、配列のサイズが 64k バイト以上の場合に、対応するためです。例えば、

```
extern int array[];
```

```
int i = array[j];
```

の場合、配列「array」の総サイズがコンパイル時には分からないため、添字「j」を 32 ビットで計算します。

本オプションを指定することで、配列「array」の総サイズを 64K バイト以内と仮定して、添字「j」を 16 ビットで計算できます。この結果処理速度の向上やコードサイズの削減が可能となります。

どの配列のサイズも 64K バイトを超えないのであれば、本オプションを使用することを推奨します。

ソースプログラム	-fsmall_array なし	-fsmall_array あり
<pre>extern far int a[]; int i,j; i=a[j];</pre>	<pre>mov.w -2[FB],R0 ; j exts.w R0 mov.l R2R0,A0 shl.l #1,A0 mov.w _a[A0],-2[FB] ; i</pre>	<pre>indexws.w -4[FB] ; j mov.w:g_a,-2[FB] ; i</pre>

図3.39 -fsmall_array の使用例

3.5.6 -fdouble_32

本オプション指定時に、double 型を float 型として処理します。

注意事項：

- 1.本オプション指定時には、必ず、関数のプロトタイプ宣言を行ってください。関数のプロトタイプ宣言がない場合には、不正なコードを生成する場合があります。
- 2.本オプションを指定した場合、double 型に対するデバッグ情報は float 型として扱われます。このため、デバッガ PD308,シミュレータ PD308SIM の C ウォッチウィンドウ,グローバルウィンドウ等では float 型として表示されます。

C ソース	-fdouble_32 なし	-fdouble_32 あり
float data; data=data+123.456;	<pre> push.l _data:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H .glb __f8add jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 .glb __f8tof4 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0,_data:16 </pre>	<pre> push.l _data:16 push.l #42f6e979H .glb __f4add jsr.a __f4add add.l #08H,SP mov.l R2R0,_data:16 </pre>

図3.40 -fdouble_32オプションの使用例

3.6 その他の pragma, オプション

3.6.1 その他の pragma

(1) メモリ配置に関する拡張機能

拡張機能	機能の内容
#pragma ROM	指定した変数を rom セクションに配置します。 記述形式) #pragma ROM 変数名 記述例) #pragma ROM val 本機能は NC77, NC79 との互換のためにあります。通常は const 修飾子を用いて rom セクションに配置してください。
#pragma SECTION	本コンパイラが生成するセクション名を変更します。 記述形式) #pragma SECTION 既定セクション名 変更セクション名 記述例) #pragma SECTION bss nonval_data

(2) 組み込み機器に使用するための拡張機能

拡張機能	機能の内容
#pragma ADDRESS (#pragma EQU)	変数を絶対アドレスに割り付けます。 記述形式) #pragma ADDRESS 変数名 絶対アドレス 記述例) #pragma ADDRESS port0 2H
#pragma BITADDRESS	指定した絶対アドレスの指定したビット位置に変数を割り付けます。 記述形式) #pragma BITADDRESS 変数名 ビット位置, 絶対アドレス 記述例) #pragma BITADDRESS io 1,100H
#pragma DMAC	外部変数に対して、DMAC レジスタを割り付けます。(NC308WA のみ) 記述形式) #pragma DMAC 変数名 DMAC レジスタ名 記述例) #pragma DMAC dma0 DMA0
#pragma INTCALL	ソフトウェア割り込み(int 命令)で呼び出す関数を宣言します。 スイッチ[/C]は、宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。 記述形式 1) #pragma INTCALL [/C] INT 番号 アセンブラ関数名(レジスタ名) 記述例 1) #pragma INTCALL 25 func(R0, R1) #pragma INTCALL /C 25 func(R0, R1) 記述形式 2) #pragma INTCALL INT 番号 C 言語関数名() 記述例 2) #pragma INTCALL 25 func() #pragma INTCALL /C 25 func() 本宣言を行う前に、必ず関数のプロトタイプ宣言を行ってください。

3. コンパイラ

#pragma INTERRUPT (#pragma INTF)	C 言語で記述した割り込み処理関数を宣言します。この宣言により、関数の出入り口で割り込み処理関数の手続きを行うコードを生成します。 記述形式) <pre>#pragma INTERRUPT [/B/ E/ F] 割り込み処理関数名 #pragma INTERRUPT [/B/ E/ F] 割り込みベクタ番号 割り込み処理関数名 #pragma INTERRUPT [/B/ E/ F] 割り込み処理関数名(vect=割り込みベクタ番号) 記述例) #pragma INTERRUPT int_func #pragma INTERRUPT /B int_func #pragma INTERRUPT 10 int_func #pragma INTERRUPT /E 10 int_func #pragma INTERRUPT int_func (vect=10) #pragma INTERRUPT /F int_func (vect=20) C77 との互換のため#pragma INTF も使用できます。</pre>
#pragma PARAMETER	アセンブラで記述された関数を呼び出す際に、その引数をレジスタを介して渡すことを宣言します。 スイッチ[/C]は、宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。 記述形式) <pre>#pragma PARAMETER [/C] 関数名(レジスタ名) 記述例) #pragma PARAMETER asm_func(R0, R1) #pragma PARAMETER /C asm_func(R0, R1) 本宣言を行う前に、必ず関数のプロトタイプ宣言を行ってください。</pre>

(3) MR308 サポートに関する拡張機能

拡張機能	機能の内容
#pragma ALMHANDLER	MR308 のアラームハンドラ名を宣言します。 記述形式) #pragma ALMHANDLER 関数名 記述例) #pragma ALMHANDLER alm_func
#pragma CYCHANDLER	MR308 の周期起動ハンドラ名を宣言します。 記述形式) #pragma CYCHANDLER 関数名 記述例) #pragma CYCHANDLER cyc_func
#pragma INTHANDLER #pragma HANDLER	MR308 の割り込みハンドラ名を宣言します。 記述形式 1) #pragma INTHANDLER [/E] 関数名 記述形式 2) #pragma HANDLER [/E] 関数名 記述例) #pragma INTHANDLER int_func
#pragma TASK	MR308 のタスクの開始関数名を宣言します。 記述形式) #pragma TASK タスクの開始関数名 記述例) #pragma TASK task1

(4) その他の拡張機能

拡張機能	機能の内容
#pragma ASM #pragma ENDASM	アセンブリ言語で記述する領域を指定します。 記述形式) #pragma ASM #pragma ENDASM 記述例) #pragma ASM mov.w R0,R1 add.w R1,02H #pragma ENDASM
#pragma JSRA	JSR 命令を JSR.A 命令に固定して関数を呼び出します。 記述形式) #pragma JSRA 関数名 記述例) #pragma JSRA func
#pragma JSRW	JSR 命令を JSR.W 命令に固定して関数を呼び出します。 記述形式) #pragma JSRW 関数名 記述例) #pragma JSRW func
#pragma PAGE	アセンブラリスティングファイルの改ページの指定を行います。 記述形式) #pragma PAGE 記述例) #pragma PAGE
#pragma __ASMMACRO	アセンブラのマクロで定義した関数を宣言します。 記述形式) #pragma __ASMMACRO 関数名 (レジスタ名) 記述例) #pragma __ASMMACRO mul(R0,R2)

3.6.2 その他のオプション

(1) コンパイルドライバの制御オプション

オプション	機能
-c	リロケータブルファイル(拡張子.r30)を作成し、処理を終了します。
-D 識別氏名	識別子を定義します。#define と同じ機能です。
-I ディレクトリ名	プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。ディレクトリは最大 16 個まで指定可能です。
-E	プリプロセスコマンドのみを処理し結果を標準出力に出力します。
-P	プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成します。
-S	アセンブリ言語ソースファイル(拡張子.a30)を作成し、処理を終了します。
-U プリデファインドマクロ名	指定したプリデファインドマクロを未定義にします。
-silent	起動時のコピーライトメッセージを出力しません。
-dsources	C 言語ソースリストをコメントとして出力した、アセンブリ言語ソースファイル (拡張子 " .a30 ") を生成します。(アセンブル後も削除しません。)

3. コンパイラ

-dsourc_in_list	“ -dsourc ” の機能に加えて、アセンブリ言語リストファイル（拡張子 “.lst ” ）を生成します。
-----------------	---

(2) 出力ファイル指定オプション

オプション	機能
-o ファイル名	ln308 が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)の名称を指定します。また、ディレクトリ名を含んだパス名も指定できます。ファイルの拡張子は必ず省略してください。
-dir ディレクトリ名	ln308 が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)の出力先ディレクトリを指定できます。

(3) バージョン情報およびコマンドライン表示オプション

オプション	機能
-v	実行中のコマンドプログラム名およびコマンドラインを表示します。
-V	コンパイラの各プログラムの起動メッセージを表示し、処理を終了します。（コンパイル処理は行いません）

(4) デバッグ用オプション

オプション	機能
-g	デバッグ情報をアセンブリ言語ソースファイル（拡張子 .a30 ）に出力します。これにより、C言語レベルデバッグが可能になります。
-genter	関数呼び出し時に必ず enter 命令を出力します。デバッグのスタックトレース機能を使用する場合には必ずこのオプションを指定してください。
-gno_reg	レジスタ変数に関するデバッグ情報の出力を抑止します。

(5) 最適化オプション

オプション	機能
-Oconst	const 修飾子で宣言した、変数の参照を定数で置き換える最適化を行います。
-Ono_bit	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	浮動小数点の定数畳み込み処理を抑止します。
-Ono_stdlib	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Ono_logical_or_combine	論理 OR をまとめる最適化を抑止します。
-Ono_asmopt	アセンブラオプション “ aopt30 ” による最適化を抑止します。
-Ocompare_byte_to_word	連続した領域のバイト単位の比較をワード単位で行います。
-Oforward_function_to_inline	全てのインライン関数に対して、インライン展開を行います。
-Oglb_jump	分岐命令に関する外部参照の最適化を行います。

(6) 生成コード変更オプション

オプション	機能
-fansi	-fnot_reserve_far_and_near,-fnot_reserve_asm,-fnot_reserve_inline、及び -fextend_to_int を有効にします。
-fnot_reserve_asm	asm を予約語にしません (" _asm " のみ有効になります)。
-fnot_reserve_far_and_near	far、near を予約語にしません (_far、_near のみ有効になります)。
-fnot_reserve_inline	inline を予約語にしません。 (_inline のみ予約語となります。)
-fextend_to_int	char 型データを int 型に拡張し演算を行います (ANSI 規格で定められた拡張を行います)
-fchar_enumerator	enumerator(列挙子)の型を int 型ではなく unsigned char 型で扱います。
-fno_even	データ出力時に奇数データと偶数データを分離しないで、すべて odd 属性のセクションに配置します。
-ffar_RAM	RAM データのデフォルト属性を far にします。
-fnear_ROM	ROM データのデフォルト属性を near にします。
-fnear_pointer	ポインタおよびアドレスのデフォルトを near にします。
-fconst_not_ROM	const で指定した型を ROM データとして扱いません。
-fnot_address_volatile	#pragma ADDRESS(#pragma EQU)で指定した変数を volatile で指定した変数とみなしません。
-fenable_register	レジスタ記憶クラスを有効にします
-finfo	インスペクタ、 " Stk Viewer " 、 " Map Viewer " 、 " utl30 " 、 に必要な情報を出力します。
-M82	M32C/80 シリーズに対応したコードを生成します。
-fswitch_other_section	switch 文に対するテーブルジャンプをプログラムセクションとは別セクションに出力します。
-ferase_static_function=関数名	本オプションで指定された関数が static 関数の場合、コード生成を行いません。
-fno_switch_table	switch 文に対して、比較を行ってから分岐するコードを生成します。
-fmake_vector_table	可変ベクタテーブルを自動生成します。
-fmake_special_table	スペシャルページテーブルを自動生成します。

(7) ライブラリ指定オプション

オプション	機能
-l ライブラリファイル名	リンク時に使用するライブラリを指定します。

3. コンパイラ

(8) 警告オプション

オプション	機能
-Wnon_prototype	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	サポートしていない #pragma を使用した場合、警告を出します。
-Wno_stop	エラーが発生してもコンパイル作業を停止しません。
-Wstdout	エラーメッセージをホストマシンの標準出力 (stdout) に出力します。
-Werror_file<fileame>	タグファイルを出力します。
-Wstop_at_warning	コンパイル時にウォーニングが発生した場合、コンパイルを停止します。
-Wnesting_comment	コメント中に /* を記述した場合に警告を出します。
-Wccom_max_warnings =ウォーニング回数	ccom308 の出力するウォーニングの回数の上限を指定できます。
-Wall	検出可能な警告 (ただし、 " -Wlarge_to_small " 、 " -Wno_used_argument " で出力される警告を除く) をすべて表示します。
-Wmake_tagfile	error および warning が発生した場合にファイル毎にタグファイルを出力します。
-Wuninitialize_variable	初期化されていない auto 変数に対してウォーニングを出力します。
-Wlarge_to_small	大きいサイズから、小さいサイズへの暗黙の代入に対して、ウォーニングを出力します。
-Wno_warning_stdlib	" -Wnon_prototype " 指定時や " -Wall " 指定時に本オプションを指定すると、「プロトタイプ宣言されていない標準ライブラリに対する警告」を抑止します。
-Wno_used_static_function	コード生成が不要な static 関数名を表示します。
-Wundefined_macro	未定義のマクロを #if の中で使用した場合に警告します
-Wstop_at_link	リンク時に、ウォーニングが発生した場合、アブソリュートモジュールファイルの生成を抑止します。

(9) アセンブル/リンクオプション

オプション	機能
-as308 < オプション>	アセンブルコマンド as308 のオプションを指定します。2 個以上のオプションを渡す場合は、" (ダブルクォーテーション) で囲んでください。
-ln308 < オプション>	リンクコマンド ln308 のオプションを指定します。2 個以上のオプションを渡す場合は、" (ダブルクォーテーション) で囲んでください。

3.7 セクション

3.7.1 NC308 が管理するセクション

NC308 はデータ/コードの配置領域を「セクション」として管理しています。
本項では、NC308 が生成および管理するセクションの種類と管理方法を説明します。

(1) セクションの構成

NC308 ではデータを種類別にセクションに分けて管理します。NC308 が管理するセクションの構成を表 3.6 に示します。

表 3.6 NC308のセクション構成

セクションベース名	内容
data	初期値を持つ静的変数を格納
bss	初期値を持たない静的変数を格納
rom	文字列・定数を格納
program	プログラムを格納
program_s	#pragma SPECIALで指定したプログラムを格納
vector	可変ベクタ領域（コンパイラは生成しない）
fvector	固定ベクタ領域（コンパイラは生成しない）
stack	スタック領域（コンパイラは生成しない）
heap	ヒープ領域（コンパイラは生成しない）

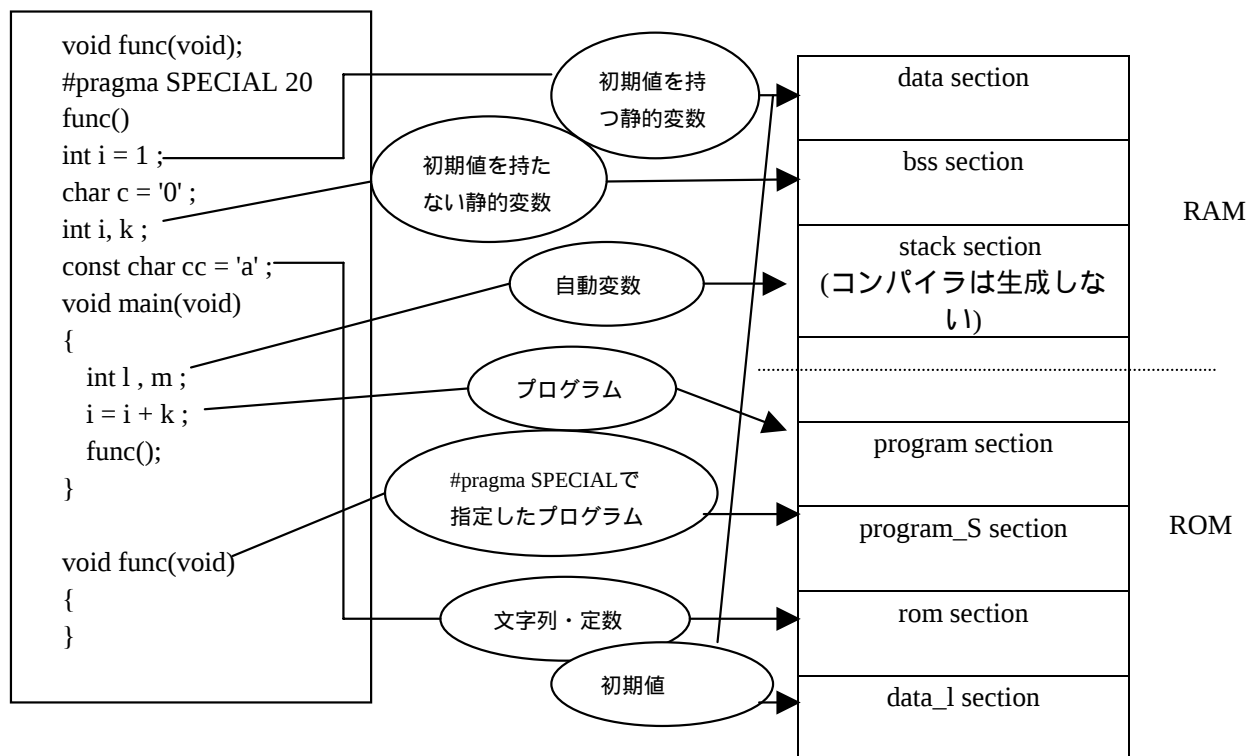


図3.41 データの種類によるセクションへの配置

3. コンパイラ

(2) セクションの属性

NC308 が生成するセクションは、初期値の有無、配置される領域、データサイズなどの「属性」によって、さらに細かく分類されます。

表 3.7 に各属性を表す記号と内容を示します。

表3.7 セクションの属性

属性	内容	対象セクションベース名
I	データの初期値を保持するセクション	data
N/F/S	N-near属性 (000000H ~ 00FFFFH番地の領域)	data,bss,rom
	F-far属性(000000 ~ FFFFFFFH番地の領域)	
	S-SBDATA属性(SB相対アドレッシングを使用できる領域)	data,bss
E/O	E-データサイズが偶数 O-データサイズが奇数	data,bss,rom

(3) セクションの命名規則

NC308 が生成するセクションの名前はセクションベース名と属性によって決められます。

各セクションベース名と属性の組み合わせを図 3.42 に示します。

セクション名 = セクションベース名_属性

		セクションベース名			
		data	bss	rom	program
属性	意味				
N	near 属性				
F	far 属性				
S	SBDATA 属性				
E	偶数データサイズ				
O	奇数データサイズ				
I	初期値を格納				

* 網掛け部分はセクションベース名と属性との組み合わせが存在するもの

図3.42 セクション名の命名規

3.8 クロスソフト間の関連

3.8.1 アセンブリ言語プログラムとの関連

ほとんどのプログラムはC言語で記述できますが、性能を追及する場合や特殊な命令を使用する場合はアセンブリ言語を使うことになります。本節ではC言語プログラムとアセンブリ言語プログラムの接続時に注意すべき事項について概説します。

(1) C言語プログラムからアセンブラ関数の呼び出し方法

(a) 引数のないアセンブラ

C言語プログラムからアセンブラ関数を呼び出す場合は、C言語で記述した関数呼び出しと同様にアセンブラ関数名で呼び出します。

アセンブラ関数の先頭ラベル名は名前の最初に_(アンダースコア)を付加する必要があります。C言語プログラムからアセンブラ関数を呼び出す場合は、アセンブラ関数の名前(先頭ラベル名)から、アンダースコアを除いた名前を使用します。呼び出すC言語プログラム中には、必ずアセンブラ関数のプロトタイプ宣言を記述してください。

図3.43 にアセンブラ関数 `asm_func` を呼び出す場合の記述例を示します。

```
extern void asm_func( void );      ← アセンブラ関数のプロトタイプ宣言
void main()
{
    :
    (省略)
    :
    asm_func();                    ← アセンブラ関数の呼び出し
}
```

図3.43 引数がない場合のアセンブラ関数の呼び出し例(smp1.c)

```
.glb _main
_main:
:
(省略)
:
jsr _asm_func  ← アセンブラ関数の呼び出し('_'を付加しています。)
rts
```

図3.44 smp1.cのコンパイル結果(抜粋)(smp1.a30)

3. コンパイラ

(b) アセンブラ関数に対して引数を与える場合

アセンブラ関数に引数を渡す場合、拡張機能の`#pragma PARAMETER` を使用します。

`#pragma PARAMETER` は、32bit 汎用レジスタ (R2R0、R3R1)、16bit 汎用レジスタ (R0、R1、R2、R3)、8bit 汎用レジスタ (R0L、R0H、R1L、R1H)、及び、アドレスレジスタ (A0、A1) を介して、アセンブラ関数に引数を渡します。

`#pragma PARAMETER` でアセンブラ関数を呼び出す手順を以下に示します。

`#pragma PARAMETER` 宣言を記述する前にアセンブラ関数のプロトタイプ宣言を行います。この場合には、必ず引数の型宣言を行ってください。

`#pragma PARAMETER` でアセンブラ関数の引数リストに使用するレジスタ名を宣言します。

図 3.45 に`#pragma PARAMETER` を使用したアセンブラ関数 `asm_func` を呼び出す場合の記述例を示します。

```
extern unsigned int asm_func(unsigned int, unsigned int);
#pragma PARAMETER asm_func(R0, R1) ← 引数をR0、R1レジスタを介して
                                     アセンブラ関数に渡します

void main()
{
    int i = 0x02;
    int j = 0x05;
    asm_func(i, j); ← アセンブラ関数の呼び出し
}
```

図3.45 引数がある場合のアセンブラ関数の呼び出し例(smp2.c)

```
.glob _main
_main:
    enter #04H
    pushm R1
    .line6
;## # C_SRC : int i = 0x02;
    mov.w #0002H,-4[FB]; i
    .line7
;## # C_SRC : int j = 0x05;
    mov.w #0005H,-2[FB]; j
    .line9
;## # C_SRC : asm_func(i, j);
    mov.w -2[FB],R1; j ← 引数を R0、R1 レジスタを介して
    mov.w -4[FB],R0; i アセンブラ関数に渡しています
    jsr _asm_func ← アセンブラ関数の呼び出し('を付加しています。)
    .line10
;## # C_SRC : }
    popm R1
    exitd
```

図3.46 smp2.cのコンパイル結果（抜粋）(smp2.a30)

(c) #pragma PARAMETER 宣言における引数型および戻り値型の制限

#pragma PARAMETER 宣言で以下の引数の型は宣言することはできません。

- 構造体型、共用体型の引数
- 64bit 整数型(long long 型)の引数
- 倍精度浮動小数点型(double 型)の引数

また、アセンブラ関数の戻り値として構造体型、共用体型の戻り値は定義できません。

(2) アセンブラ関数の記述方法

(a) 呼び出されるアセンブラ関数の記述方法

アセンブラ関数の入り口処理の記述手順を以下に示します。

アセンブラの疑似命令 .SECTION でセクション名を指定します。

関数名ラベルをアセンブラの疑似命令 .GLB でグローバル指定します。

関数名に_(アンダースコア)を付加して、ラベルとして記述します。

関数内で B 及び U フラグを変更する場合は、フラグレジスタをスタック上に退避してください。

関数内で破壊されるレジスタを退避してください。アセンブラ関数の出口処理の記述手順を以下に示します。

関数の入口処理で退避したレジスタを復帰してください。

関数内で B 及び U フラグを変更した場合は、スタックからフラグレジスタを復帰してください。

RTS 命令を記述します。

また、アセンブラ関数内で SB、FB レジスタ内容を書き換える操作は行わないでください。

SB、FB レジスタの内容を書き換える場合は、関数の入口でスタックに退避し、関数の出口でスタックから復帰してください。

図 3.47 にアセンブラ関数の記述例を示します。この例では、セクション名を本コンパイラが出力するセクション名と同じ program を用いています。

```
.SECTION program      ←
.GLB _asm_func        ←
_asm_func:            ←
    PUSHC FLG          ←
    PUSHM R3,R1        ←
    MOV.L SYM1, R3R1
    POPM R3,R1         ←
    POPC FLG           ←
    RTS                ←
.END
```

～ は、上記の手順に対応しています。

図3.47アセンブラ関数の記述例

3. コンパイラ

(b) アセンブラ関数からの戻り値の返し方

アセンブラ関数から C 言語プログラムに値を返す場合、整数型、ポインタ型、浮動小数点型については、レジスタ渡しで戻り値を返すことができます。表 3.8 に戻り値に関する呼び出し規則を、図 3.48 に戻り値を返すアセンブラ関数の記述例を示します。

表3.8 戻り値に関する呼び出し規則

戻り値の型	規則
_Bool型 char型	R0Lレジスタ
int型 nearポインタ型	R0レジスタ
float型 long型	下位 16 ビットは R0 レジスタに、上位 16 ビットは R2 レジスタに格納して返します。
double型 long double型	R3、R2、R1、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
long long 型	R3、R1、R2、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
構造体型 共用体型	呼び出しを行う直前に、戻り値を格納するための領域を指す far アドレスをスタックに積みます。呼び出された関数はリターンする前にスタックに積まれた far アドレスで指す領域に戻り値を書き込みます。

```
.SECTION program
.GLB _asm_func
_asm_func:

(省略)

MOV.I #01A000H, R2R0
RTS
.END
```

図3.48 long型の戻り値を返すアセンブラ関数の記述例

(c) C 言語の変数の参照方法

アセンブラ関数は C 言語プログラムとは別のファイルに記述するため、C 言語の大域変数のみ参照することができます。

C 言語の変数名をアセンブラ関数内で記述する場合は、変数名の前に_(アンダースコア)を付加します。また、アセンブリ言語プログラムでは外部参照する変数をアセンブラの疑似命令.GLB で外部参照宣言する必要があります。

図 3.49 に C 言語プログラムの大域変数 counter をアセンブラ関数 asm_func 内で参照する例を示し

ます。

```

[C 言語プログラム]
unsigned int counter;      ← C 言語プログラムの大域変数

main()
{
:
(省略)
:
}
[アセンブラ関数]
    .GLB _counter          ← C 言語プログラムの大域変数を外部参照宣言
asm_func:
:
(省略)
:
    MOV.W _counter, R0     ← 参照

```

図3.49 C言語の大域変数の参照方法

(d) 割り込み処理をアセンブラ関数で記述する場合の注意事項

割り込み処理を実行するプログラム(関数)では、出入り口で以下の処理を行う必要があります。

- 1.関数の入口でレジスタ(R0、R1、R2、R3、A0、A1、FB)を一括に退避します。
- 2.関数の出口でレジスタ(R0、R1、R2、R3、A0、A1、FB)を一括に復帰します。
- 3.関数からのリターンに REIT 命令を使用します。

図 3.50 に割り込み処理のアセンブラ関数の記述例を示します。

```

.section program
.glb _func
_func:
    pushm R0,R1,R2,R3,A0,A1,FB    ← レジスタの一括退避
    MOV.B #01H, R0L
:
(省略)
:
    popm R0,R1,R2,R3,A0,A1,FB     ← レジスタの一括復帰
    reit                          ← C 言語プログラムへリターン
.END

```

図3.50 割り込み処理のアセンブラ関数の記述例

3. コンパイラ

(e) アセンブラから C 言語関数を呼び出す場合の注意事項

アセンブリ言語プログラムから C 言語で記述された関数を呼び出す場合は、以下の点に注意してください。

C 言語の関数名に_ (アンダースコア) あるいは\$ (ドルー) を付加したラベル名で呼び出してください。

NC308 では、C 言語関数の入口処理で、R0 レジスタおよび戻り値に使用するレジスタの退避を行いません。このため、アセンブラから C 言語の関数を呼び出す場合、その前に R0 レジスタおよび戻り値に使用しているレジスタの退避を行ってください。

NC30 では、C 言語関数でレジスタの退避復帰を行わないため、アセンブラ関数中で使用しているレジスタは C 言語関数を呼び出す前に退避し、C 言語関数から戻った後に復帰してください。

(3) アセンブラ関数の記述に関する注意事項

C 言語プログラムから呼び出すアセンブリ言語の関数(サブルーチン)を記述する場合、以下の点に注意してください。

(a) B、U フラグの取り扱いに関する注意事項

アセンブラ関数から C 言語プログラムにリターンする場合は、必ず B フラグ及び U フラグを呼び出し時と同じ状態にしてください。

(b) FB レジスタの取り扱いに関する注意事項

アセンブラ関数の中で FB(フレームベースレジスタ)の値を変更した場合、呼び出し元の C 言語プログラムへ正常に復帰できなくなります。したがって、アセンブラ関数中で FB の値を変更しないでください。システムの設計上やむをえず変更する場合は、関数の先頭でスタックに退避して、リターンするときに復帰させてください。

(c) 汎用レジスタおよびアドレスレジスタの取り扱いに関する注意事項

アセンブラ関数の中で汎用レジスタ(R0 を除く、R1、R2、R3)及びアドレスレジスタ(A0、A1)の内容を変更する場合、アセンブラ関数の入口処理でそれらを退避し、出口処理で復帰する必要があります。

ただし、#pragma PARAMETER /C で宣言されたアセンブラ関数は、呼び出した側で待避・復帰を行うコードが生成されますので、アセンブラ関数内で、待避・復帰を行う必要はありません。(多少コードサイズは大きくなります)

(d) アセンブラ関数への引数に関する注意事項

アセンブリ言語で記述した関数に対して引数を渡す場合、#pragma PARAMETER 機能を使用してその引数をレジスタを介して渡すことができます。その書式を図 3.51 に示します(図中の asm_func はアセンブラ関数名です)。

```
unsigned int near asm_func(unsigned int, unsigned int);
                                     ↑ アセンブラ関数のプロトタイプ宣言
#pragma PARAMETER asm_func(R0, R1)
```

図3.51 アセンブラ関数の記述例

#pragma PARAMETER は、16 ビット汎用レジスタ(R0、R1、R2、R3)、8 ビット汎用レジスタ(R0L、R0H、R1L、R1H)及びアドレスレジスタ(A0、A1)を介してアセンブラ関数に引数を渡します。また、16 ビット汎用レジスタ及びアドレスレジスタを組み合わせる 32 ビットレジスタ(R3R1、R2R0、A1A0)としてアセンブラ関数に引数を渡します。

なお、#pragma PARAMETER 宣言の前には必ずアセンブラ関数のプロトタイプ宣言を行ってください。

ただし、#pragma PARAMETER 宣言で以下の引数の型は宣言することはできません。

- 構造体型、共用体型の引数
- 64bit 整数型(long long 型)の引数
- 倍精度浮動小数点型(double 型)の引数

また、アセンブラ関数の戻り値として構造体型、共用体型の戻り値は定義できません。

3.9 long long 型

long long, unsigned long long 型のデータ型をサポートします。

有符号整数については long long、無符号整数については unsigned long long と書きます。

型が long long の整数定数を作成する場合、整数値の後ろに接尾語 LL をつけ、型が unsigned long long の整数定数の場合は、整数値の後ろに接尾語 ULL を付けます。

表 3.9 整数型とその値の範囲

型	値の範囲	データサイズ
char	0 ~ 255	1 byte
signed char	-128 ~ 127	1 byte
unsigned char	0 ~ 255	1 byte
short	-32768 ~ 32767	2 byte
unsigned short	0 ~ 65535	2 byte
int	-32768 ~ 32767	2 byte
unsigned int	0 ~ 65535	2 byte
long	-2147483648 ~ 2147483647	4 byte
unsigned long	0 ~ 4294967295	4 byte
long long	-9223372036854775808 ~ 9223372036754775807	8 byte
unsigned long long	0 ~ 18446744073709551615	8 byte

3.10 near/far 型

M16C/80 シリーズのアクセス領域は最大 16M バイトです。NC308 ではこの領域を 000000H ~ 00FFFFH 番地の「near 領域」、000000H ~ FFFFFFFH 番地の領域を「far 領域」と分割して管理しています。この項では、これらの領域への変数、関数の配置方法とアクセス方法を説明します。

3.10.1 near 領域と far 領域

NC308 では最大 16M バイトのアクセス空間を「near 領域」と「far 領域」の 2 つの領域で管理しています。それぞれの領域の特徴を表 3.10 に示します。

表 3.10 near 領域と far 領域

領域名	内容
near 領域	M16C/80 シリーズがデータを効率よくアクセスできる空間。 絶対番地 000000H ~ 00FFFFH の 64K バイトの領域でスタックや 内部 RAM などが配置される。
far 領域	M16C/80 シリーズがアクセスできる絶対番地 000000H ~ FFFFFFFH の 16M バイトの全メモリ空間。 内部 ROM などが配置される。

3.10.2 near / far 属性のデフォルト

NC308 では near 領域に配置される変数、関数を「near 属性」、far 領域に配置される変数、関数を「far 属性」と区別しています。変数、関数のデフォルトの属性を表 3.11 に示します。

表 3.11 near / far 属性のデフォルト

分類	属性
プログラム	far 固定
RAM データ	near (ただしポインタ型は far)
ROM データ	far
スタックデータ	near 固定

near / far の属性をデフォルトから変更したい場合は、NC308 起動時に次の起動オプションを指定してください。

- ffar_RAM (- fFRAM) ; RAM データのデフォルト属性を" far " に変更
- fnear_ROM (- fNROM) ; ROM データのデフォルト属性を" near " に変更
- fnear_pointer (- fNP) ; ポインタ型のデフォルト属性を"near" に変更

3.10.3 関数の near / far

M16C/80 シリーズのアーキテクチャ上、NC308 の関数の属性は far 領域固定です。near を指定した場合、NC308 はコンパイル時にウォーニングを出力し、強制的に far 領域に配置します。

3.10.4 変数の near / far

[記憶クラス] 型指定子 near / far 変数名 ;

3. コンパイラ

型宣言時に `near` / `far` を指定しなければ、RAM データは `near` 領域に配置され、`const` 修飾子を指定したものや ROM データは `far` 領域へ配置されます。

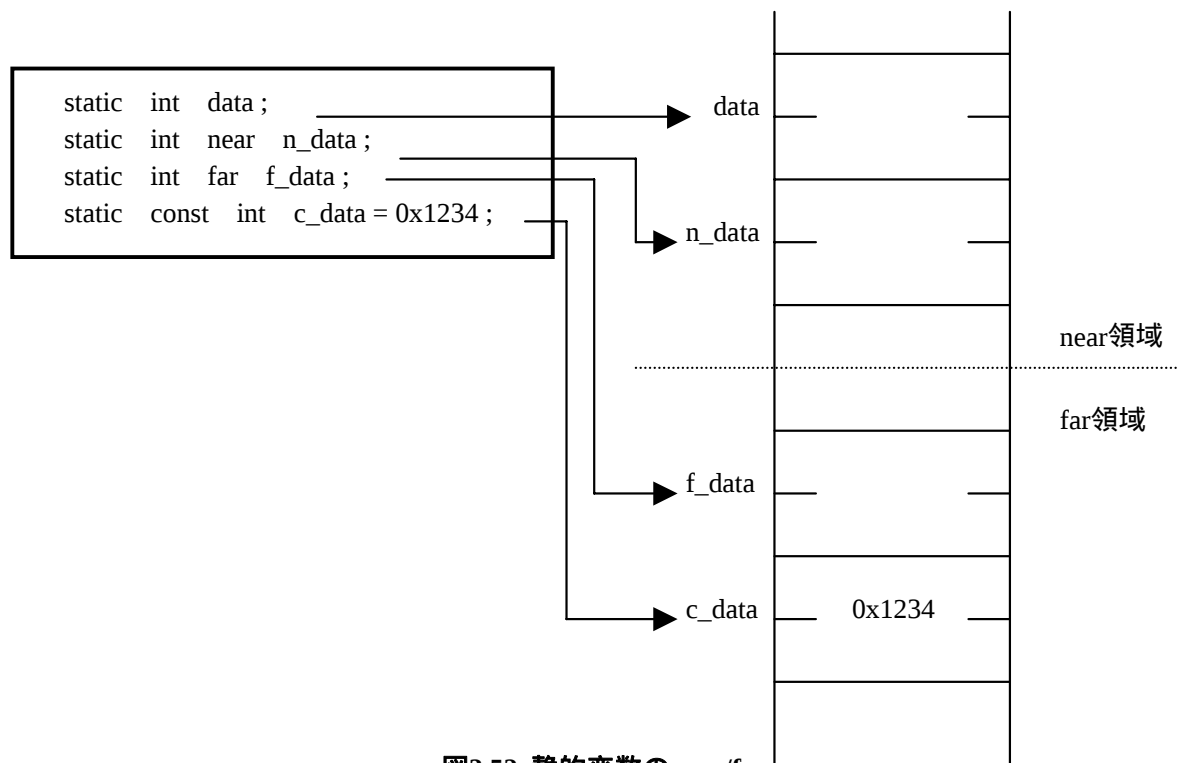


図3.52 静的変数のnear/far

自動変数に対しては `near` / `far` を指定しても配置には全く影響を及ぼしません（すべてスタック領域に配置されます）。

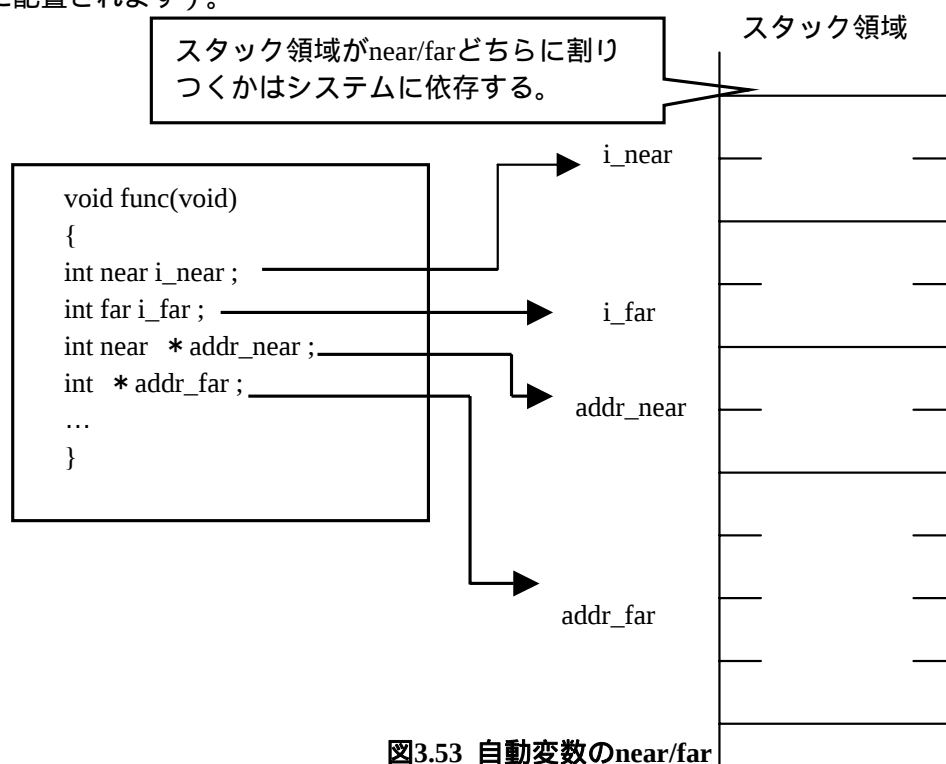


図3.53 自動変数のnear/far

3.10.5 ポインタの near / far

ポインタの near / far 指定により、ポインタに格納するアドレスのサイズやポインタ自身を配置する領域を指定します。

(1) ポインタに格納するアドレスのサイズを指定します。

なにも指定しなければ far 領域にある変数を指し示す 32 ビット長 (4 バイト) のポインタ変数として扱われます。

[記憶クラス] 型指定子 near / far *変数名 ;

near → ポインタ変数に格納するアドレスのサイズ 16 ビット長

far → ポインタ変数に格納するアドレスのサイズ 32 ビット長

```
int near *near_data;
int far *far_data;
```

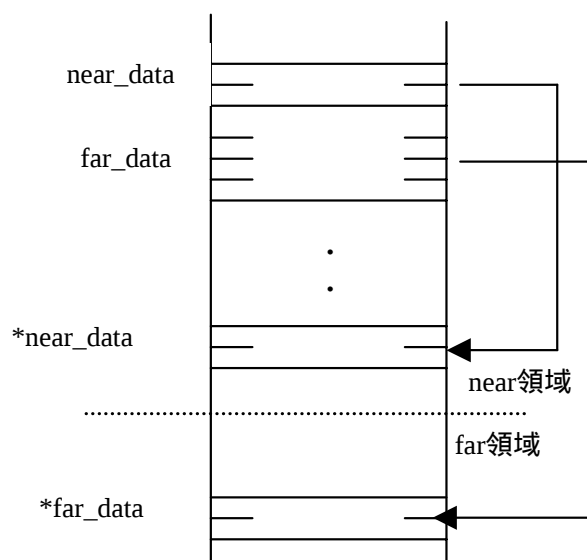


図3.54 ポインタに格納するアドレスサイズを指定

3. コンパイラ

- (2) ポインタ自身が配置される領域を指定します。
なにも指定しなければポインタ変数自身は near 領域に配置されます。

[記憶クラス] 型指定子 * near / far 変数名 ;

near → ポインタ変数自身の領域を near 領域へ配置

far → ポインタ変数自身の領域を far 領域へ配置

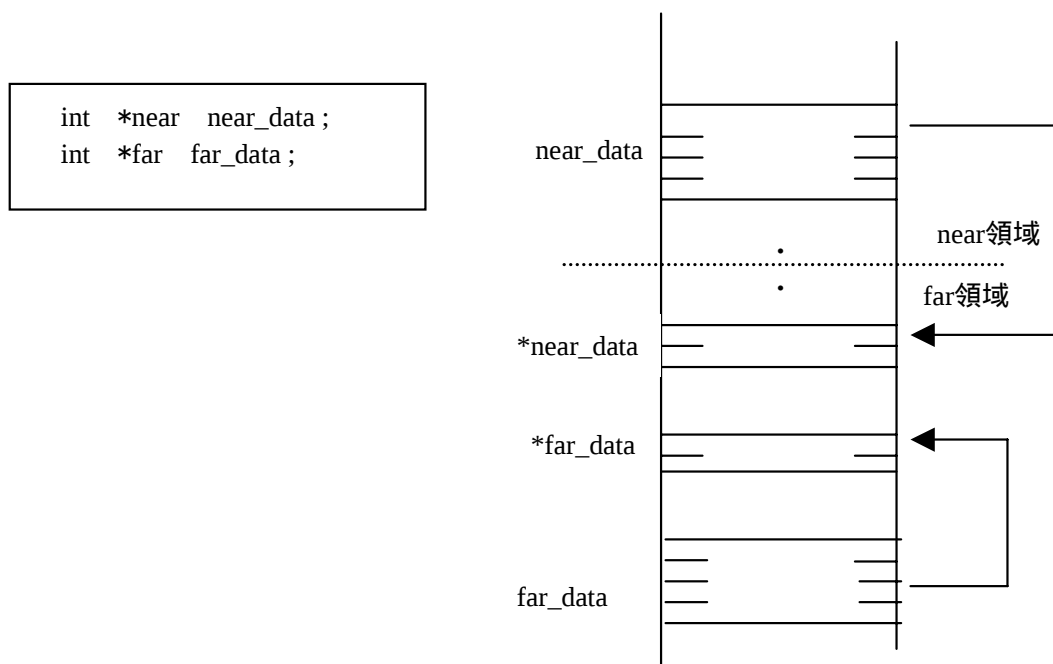


図3.55 ポインタ配置領域を指定

3.10.6 ポインタの near/far 指定における NC308 と NC30 の相違点

M16C/60、M16C/20 シリーズ用の C コンパイラ NC30 ではポインタの near/far をなにも指定していなければすべて near 属性として扱われていました。NC308 ではポインタに格納するアドレスのサイズを指定する場合に、なにも指定していなければポインタ変数のサイズは 32 ビット長 (4 バイト) となり、far 領域にある変数を指し示すポインタ変数として扱われます。

3.10.7 far 領域に配置された変数アドレスの near ポインタへの代入

NC308 は far 領域に配置された変数のアドレスを near ポインタへ代入しようとした場合、アドレスの上位が無視されて代入が行われることを示すウォーニングメッセージを出力します。

また、明示的にあるいは暗黙的に far ポインタを near ポインタに変換されたことを示すウォーニングメッセージを出力します。

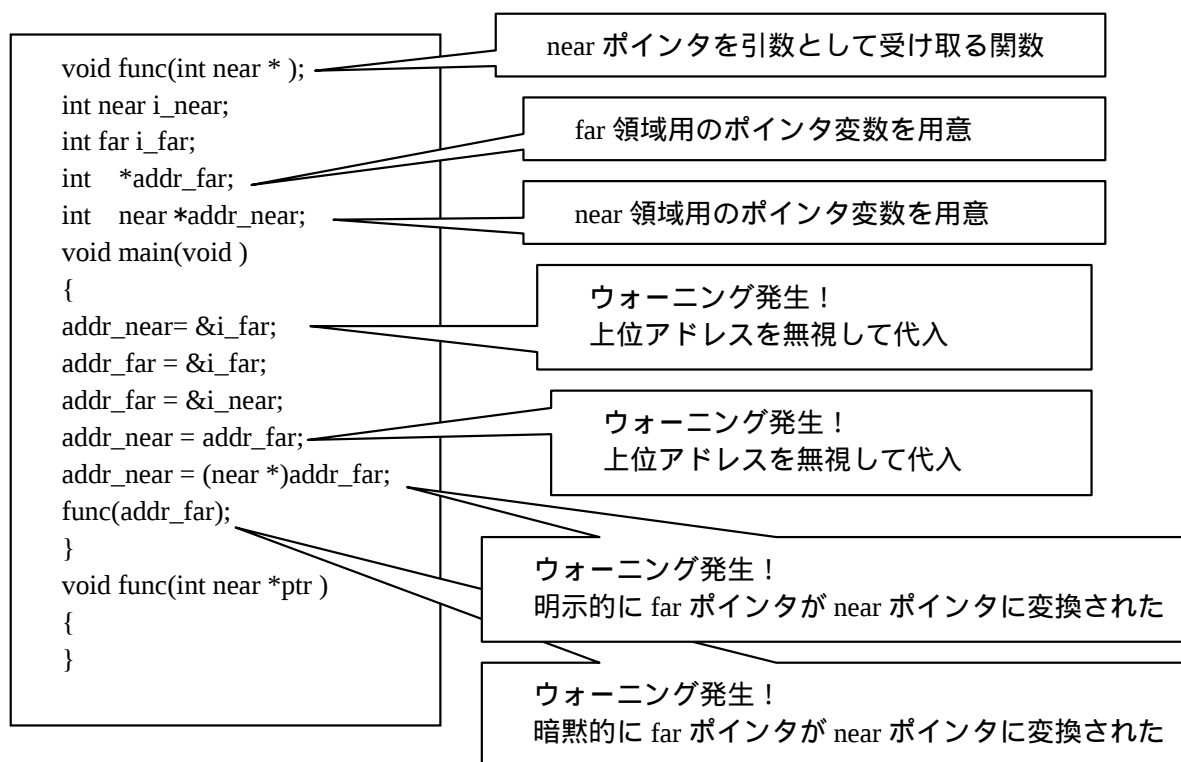


図3.56 far領域に配置された変数アドレスのnearポインタへの代入

3.11 インライン展開

C++ 風に inline 記憶クラスを指定することができます。関数に対して inline 記憶クラスを指定することにより関数をインライン展開することができます。

3.11.1 inline 記憶クラスの概要

inline 記憶クラス指定子は、関数に対してインライン展開される関数であることを宣言します。inline 記憶クラス指定した関数は、アセンブリ言語レベルでは直接コードが埋め込まれます。

3.11.2 inline 記憶クラスの宣言書式

inline 記憶クラス指定子は、文法的に static 、extern 型記憶クラス指定子と同様の書式で宣言時に記述します。図 3.57 に宣言時の書式を示します。

```
inline 型指定子 関数 ;
```

図3.57 inline記憶クラスの書式宣言

関数の宣言例を図 3.58 に、コンパイル結果を図 3.59 に示します。

```
inline int func(int i)           ← インライン関数の宣言および定義部
{
    return i++;
}
void main()
{
    int s;
    s = func(s);                 ← インライン関数呼び出し部
}
```

図3.58 インライン関数のサンプルプログラム

```

._LANG 'C','X.XX.XX','REV.X'
;## NC308 C Compiler OUTPUT
;## ccom308 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGH
RESERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,
2000,2001,2002,2003
;## COMMAND_LINE: ccom308 D:¥MTOOL¥nc308wa5¥TMP¥sss.i -o .¥smp.a30
d S
;## Normal Optimize O F F
;## ROM size Optimize O F F
;## Speed Optimize O F F
;## Default ROM is f a r
;## Default RAM is near
.GLB __SB__
.SB __SB__
.FB 0
;## # FUNCTION func
;## # FUNCTION main
;## # FRAME AUTO ( s) size 2, offset -4
;## # FRAME AUTO ( i) size 2, offset -2
;## # ARG Size(0) Auto Size(4) Context Size(8)
.SECTION program,CODE,ALIGN
._file 'smp.c'
._align
._line 7
;## # C_SRC : {
.glb _main
_main:
    enter #04H
    pushm R1
    ._line 9
;## # C_SRC : s = func(s);
    mov.w -4[FB],R0 ; s
    ._line 2
;## # C_SRC : {
    mov.w R0,-2[FB] ; i
    ._line 3
;## # C_SRC : return i++;
    mov.w R0,R1
    add.w #0001H,R0
    ._line 9
;## # C_SRC : s = func(s);
    mov.w R1,-4[FB] ; s
    ._line 10
;## # C_SRC : }
    popm R1
    exitd
E1:
.END
;## Compile End Time Tue Jul 16 13:12:00 20xx

```

←インライン関数が埋め込まれている

図3.59 サンプルプログラムのコンパイル結果

3.11.3 inline 記憶クラスの規定事項

inline 記憶クラス指定時には、以下の点に注意してください。

- インライン関数の引数について
インライン関数の引数には、構造体や共用体を使用することはできません。これらを使用した場合、コンパイルエラーとなります。
- インライン関数の間接呼び出しについて
インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合、コンパイルエラーとなります。
- インライン関数の再帰呼び出しについて
インライン関数の再帰呼び出しをすることはできません。再帰呼び出しの記述を行った場合、コンパイルエラーとなります。
- インライン関数の定義について
関数に対して inline 記憶クラスを指定する時には、宣言の記述の後に必ず実体定義を行ってください。実体定義は必ず、同一ファイル内に記述してください。図 3.60 の記述は本コンパイラでは、エラーとして処理します。

```
inline void func(int i);
void main( void )
{
    func(1);
}
```

【エラーメッセージ】

```
[Error(ccom):smp.c,line 5] inline function's body is not declared previously
==> func(1);
Sorry, compilation terminated because of these errors in main().
```

図3.60 inline関数の不適切な記述例

また、ある関数を通常関数として使用した後に、その関数をインライン関数として定義した時には、inline の指定は無効になりすべて static な関数として扱います。この時、本コンパイラでは警告を發します。

```
int func(int i);
void main( void ){
    func(1);
}
inline int func(int i){
    return i;
}
```

【ウォーニングメッセージ】

```
[Warning(ccom):smp.c,line 9] inline function is called as normal function before
,change to static function
```

図3.61 inline関数の不適切な記述例（ 2 ）

- インライン関数のアドレスについて

インライン関数は、関数自身はアドレスを持ちません。そのため、インライン関数に対して&演算子を使用した場合には、エラーになります。

```
inline int func(int i)
{
    return i;
}
```

```
main()
{
```

```
    int (*f)(int);
```

```
    f = &func;
```

```
}
```

【エラーメッセージ】

[Error(ccom):smp.c,line 10] can't get inline function's address by '&' operator

==> f = &func;

Sorry, compilation terminated because of these errors in main().

図3.62 inline関数の不適切な記述例（ 3 ）

- static データの宣言

インライン関数内で static データを宣言した場合、宣言した static データの実体はファイル単位で確保されます。そのため、複数のファイルにまたがったインライン関数ではアクセスする領域が異なります。

インライン関数内で使用する static データは関数外で宣言してください。本コンパイラでは、インライン関数内で static 宣言をした場合には、警告を発します。また、インライン関数内の static 宣言は、推奨しません。

```
inline int func( int j)
```

```
{
```

```
    static int i = 0;
```

```
    i++;
```

```
    return i + j;
```

```
}
```

【ウォーニングメッセージ】

[Warning(ccom):smp.c,line 3] static valuable in inline function

==> static int i = 0;

図3.63 inline関数の不適切な記述例(4)

- デバッグ情報について

本コンパイラではインライン関数に対する C 言語レベルのデバッグ情報を出力しません。従って、インライン関数のデバッグはアセンブリ言語レベルで行うことになります。

4. High-performance Embedded Workshop の活用

4.1 High-performance Embedded Workshop のオプション指定方法

Options メニューからオプションを指定することができます。統合化環境からのオプション指定方法を示します。Options メニューから Renesas M32C Standard Toolchain を選択します。

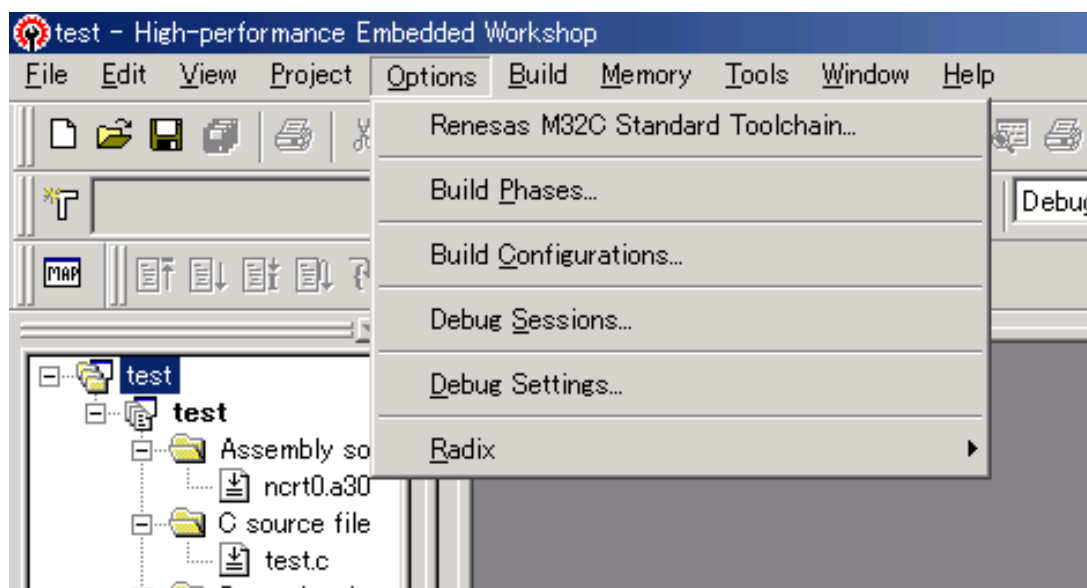


図 4.1 High-performance Embedded Workshop Options メニュー

4.1.1 C コンパイラのオプション

Renesas M32C Standard Toolchain ダイアログボックスから C タブを選択します。

(1) Category:[Source]

表 4.1 Category:[Source]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
Show entries for :	
Include files directories	I<ディレクトリ名>
Defines	D<sub><sub> : <マクロ名> [= <文字列>]
Predefines	U<sub><sub> : <プリデファインドマクロ名>

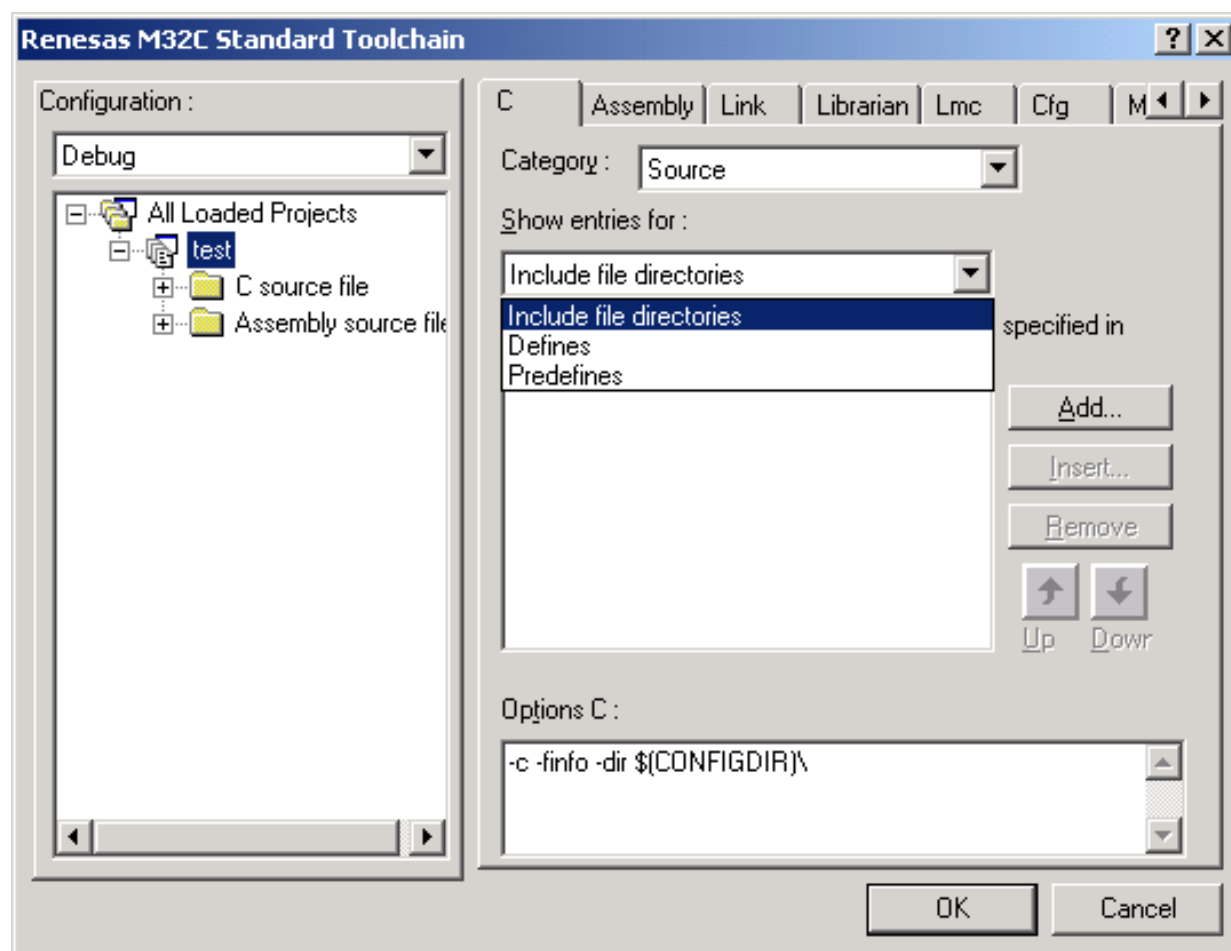


図 4.2 Category:[Source] のダイアログボックス

(2) Category:[Object]

表 4.2 Category:[Object]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
Output file type : [-c] Relocatable file (*.r30) [-S] Assembly language source file (*.a30) [-P] Preprocessed source file (*.i) [-E] Preprocessed output	c S P E
Debug options : [-finfo] Outputs information needed for Inspector, Stk Viewer, and utl30 [-g] Outputs debugging information. Therefore you can perform C language-level debugging [-genter] Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace function [-gno_reg] Suppresses the output of debugging information for register variables	finfo g genter gno_reg
[-dir] Specifies the directory to output the file(s) to :	dir<ディレクトリ名>

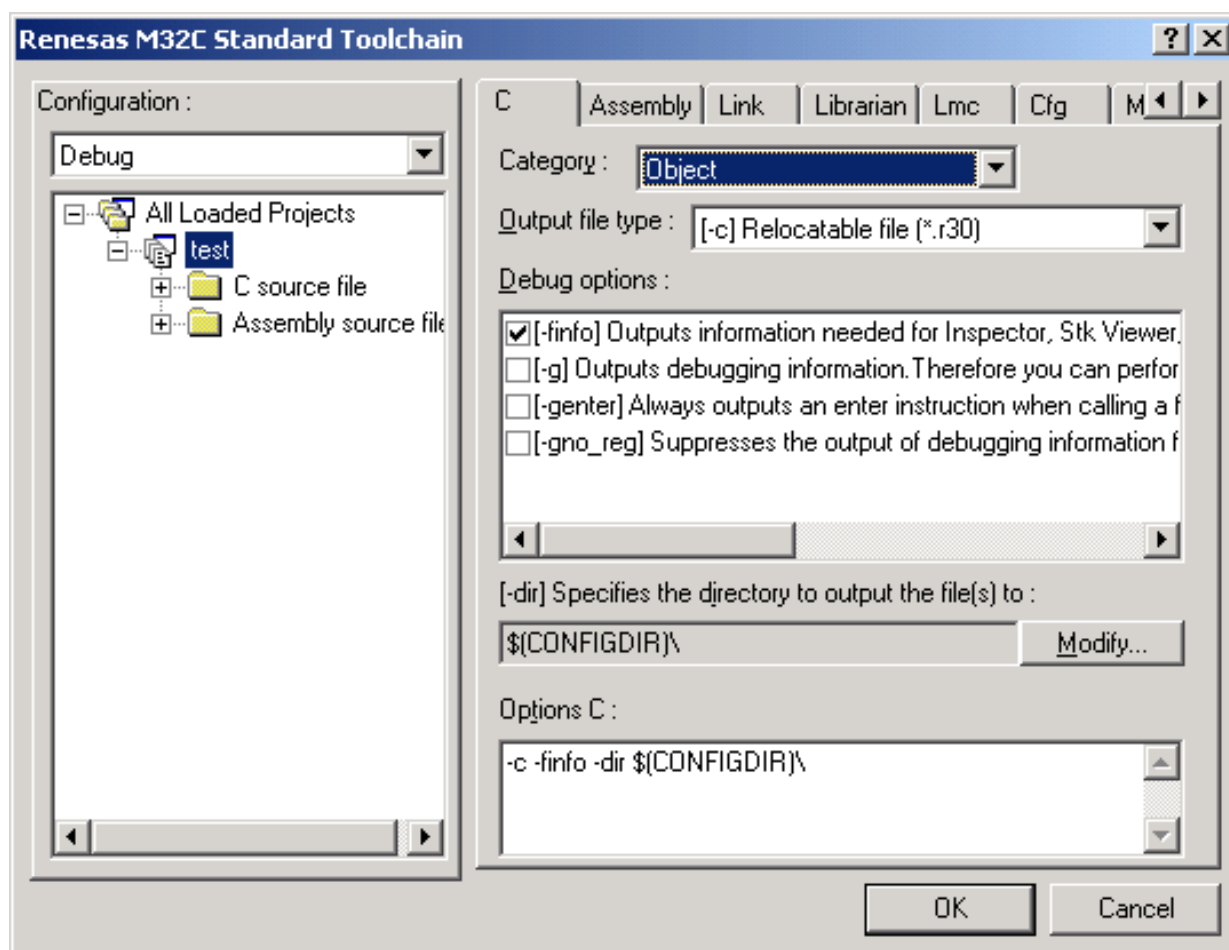


図 4.3 Category:[Object]のダイアログボックス

(3) Category:[List]

表 4.3 Category:[List]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション	短縮形
[-dS] Outputs C source code as comments in the output assembly language source list.	dsource	dS
[-dSL] Outputs C source code as comments in the output assemble list.	dsource_in_list	dSL

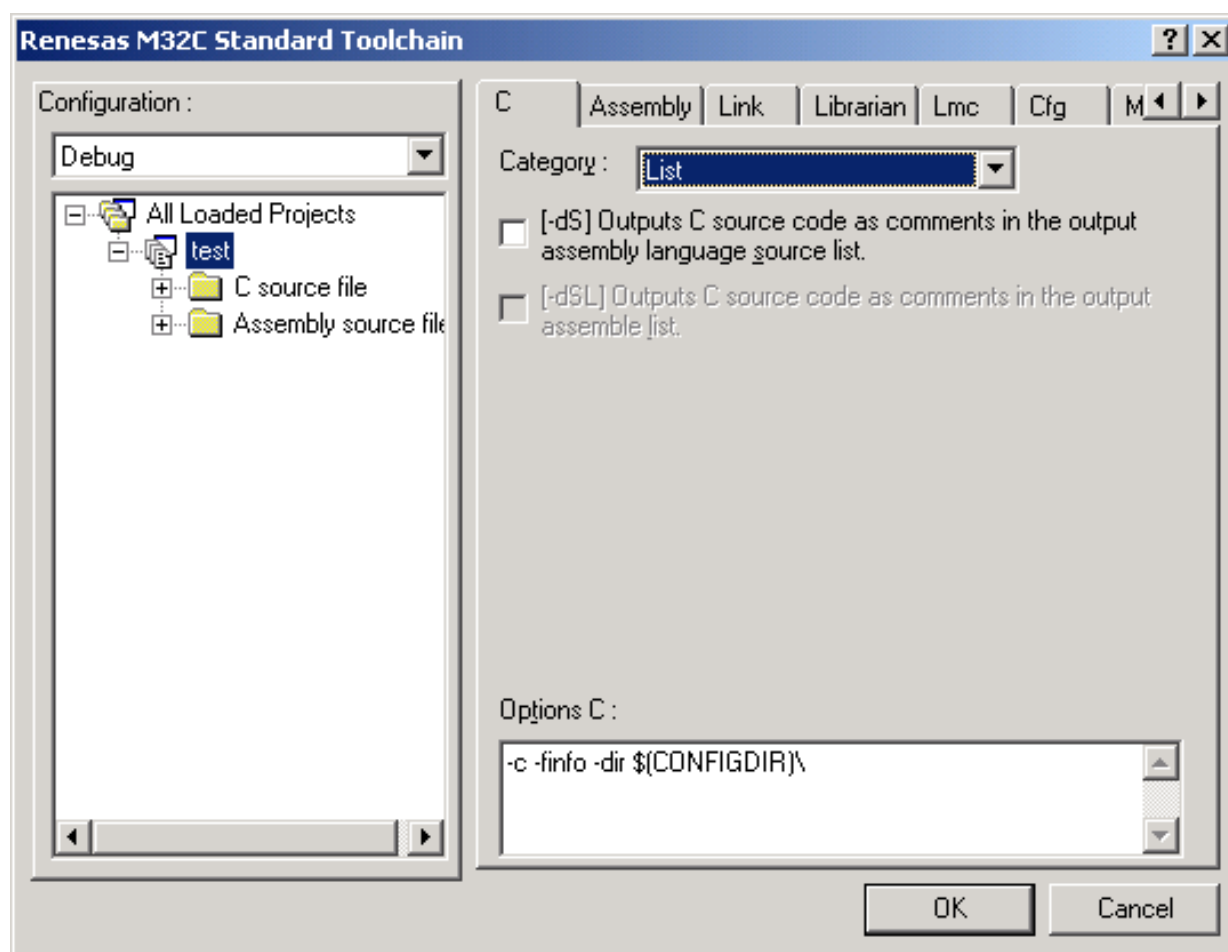


図 4.4 Category:[List]のダイアログボックス

(4) Category:[Optimize]

表 4.4 Category:[Optimize]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション	短縮形
Optimization level :		
[-O1] Makes -O3,-ONB,-ONBSD,-ONFCF,and -ONS valid	O1	なし
[-O2] Makes no difference with -O1	O2	なし
[-O3] Optimizes speed and ROM size to the maximum	O3	なし
[-O4] Makes -O3 and Oconst valid	O4	なし
[-O5] Effect the best possible optimization	O5	なし
Size or speed :		
[-OR] ROM size followed by speed	OR	なし
[-OS] Speed followed by ROM size	OS	なし
Miscellaneous options :		
[-OC] Performs optimization by replacing references to the const-qualified external variables with constants	Oconst	OC
[-OCBTW] Compares consecutive bytes of data at contiguous addresses in words	Ocompare_byte_to_word	OCBTW
[-OFFTI] In line deployment is performed to the function described ahead.	Oforward_function_to_inline	OFFTI
[-OFTI] A floating point runtime library function is developed.	Ofloat_to_inline	OFTI
[-OGJ] Optimizes the branch instruction which refers to the global label.	Oglb_jump	OGJ
[-ONA] Suppresses execution of assembler optimizer aopt308	Ono_asmopt	ONA
[-ONB] Suppresses optimization based on grouping of bit manipulations	Ono_bit	ONB
[-ONBSD] Suppresses optimization that affects source line information	Ono_break_source_debug	ONBSD
[-ONFCF] Suppresses the constant folding processing of floating point numbers	Ono_float_const_fold	ONFCF
[-ONLOC] Suppresses the optimization that puts consecutive ORs together	Ono_logical_or_combine	ONLOC
[-ONS] Inhibits inline padding of standard library functions and modification of library functions	Ono_stdlib	ONS
[-OSA] Performs optimization to remove stack correction code after calling a function	Osp_adjust	OSA
[-OSTI] A static function is treated as an inline function	Ostatic_to_inline	OSTI
[-OLU] Expands sentences the number of times to loop without loop :	Oloop_unroll=<数値>	OLU

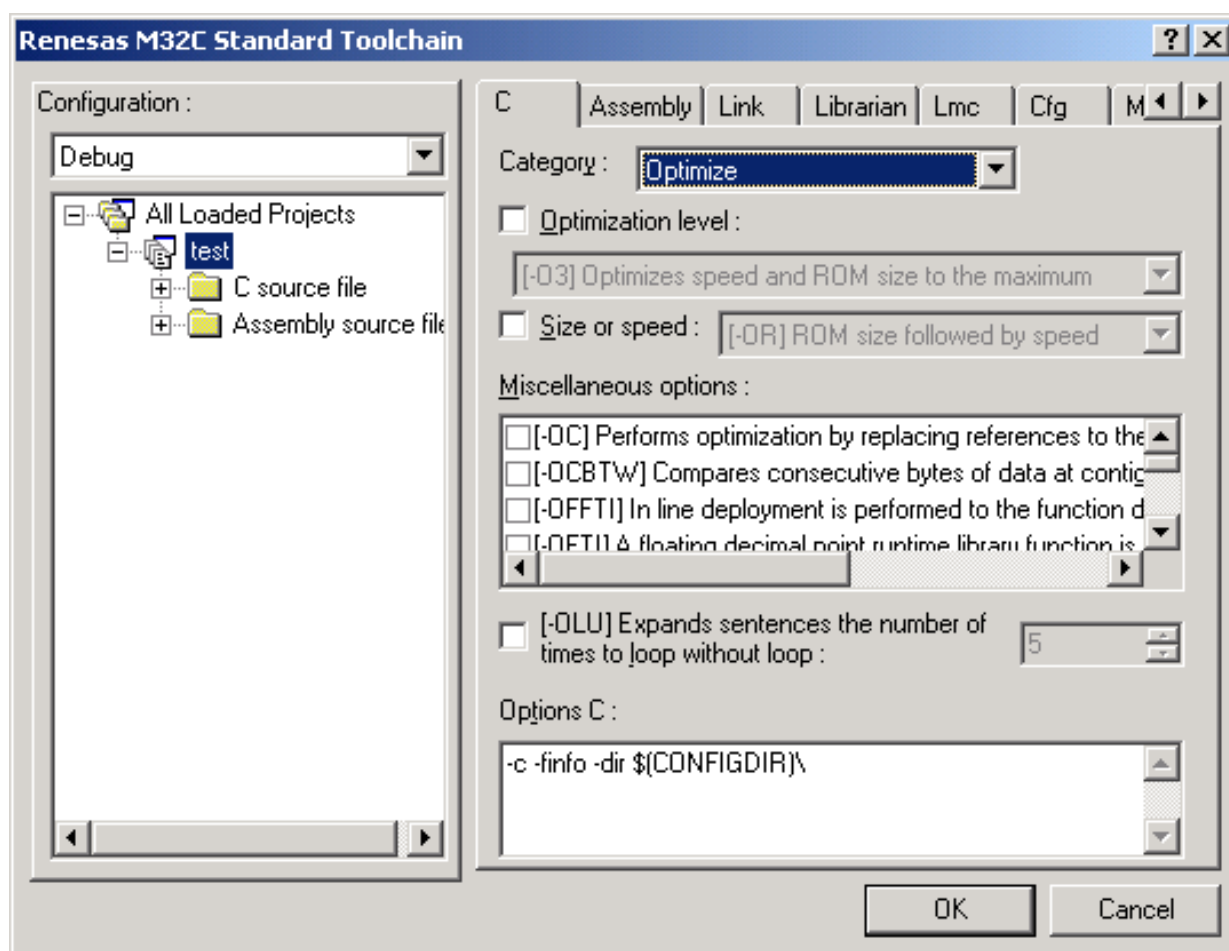


図 4.5 Category:[Optimize]のダイアログボックス

(5) Category:[Code Modification]

表 4.5 Category:[Code Modification]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション	短縮形
Miscellaneous options :		
[-fansi] Makes -fNRA, -fNRFAN, -fNRI, and -fETI valid	fansi	なし
[-fCE] Handles the enumerator type as an unsigned char, not as an int type	fchar_enumerator	fCE
[-fCNR] Does not handle the types specified by const as ROM data	fconst_not_ROM	fCNR
[-fD32] Handles the double type as the float type	fdouble_32	fD32
[-fER] Make register storage class available	fenable_register	fER
[-fETI] Performs operation after extending char-type data to the int type. (Extended according to ANSI standards.)	fextend_to_int	fETI
[-fFRAM] Changes the default attribute of RAM data to far	ffar_RAM	fFRAM
[-fJSRW] Changes the default instruction for calling functions to JRSW	fJSRW	なし
[-fMST] Generates special page vector table.	fmake_special_table	fMST
[-fMVT] Generates variable vector table.	fmake_vector_table	fMVT
[-fNA] Does not align the starting address of functions	fno_align	fNA
[-fNAV] Does not regard the variables specified by #pragma ADDRESS as those specified by volatile	fnot_address_volatile	fNAV
[-fNE] Allocate all data to the odd section, with no separating odd data from even data when outputting	fno_even	fNE
[-fNP] Changes the default type of pointer data to near.	fnear_pointer	fNP
[-fNRA] Exclude asm from reserved words. (Only _asm is valid.)	fnot_reserve_asm	fNRA
[-fNRFAN] Exclude far and near from reserved words. (Only _far and _near are valid.)	fnot_reserve_far_and_near	fNRFAN
[-fNRI] Exclude inline a reserved words. (Only _inline is valid.)	fnot_reserve_inline	fNRI
[-fNROM] changes the default attribute of ROM data to near	fnear_ROM	fNROM
[-fNST] To a switch sentence, it always compares and branched code is generated.	fno_switch_table	fNST
[-fSA] When referencing a far-type array, this option calculates subscripts in 16 bits if the total size of the array is within 64K bytes	fsmall_array	fSA
[-fSOS] Outputs the from ROM table corresponding to the section differing from the program section	fswitch_other_section	fSOS
[-fUD] Ignores an overflow when using a divide operation.	fuse_DIV	fUD
[-fESF] If the function specified is a static function, no codes are generated.	ferase_static_function=<関数名>	fESF=<関数名>

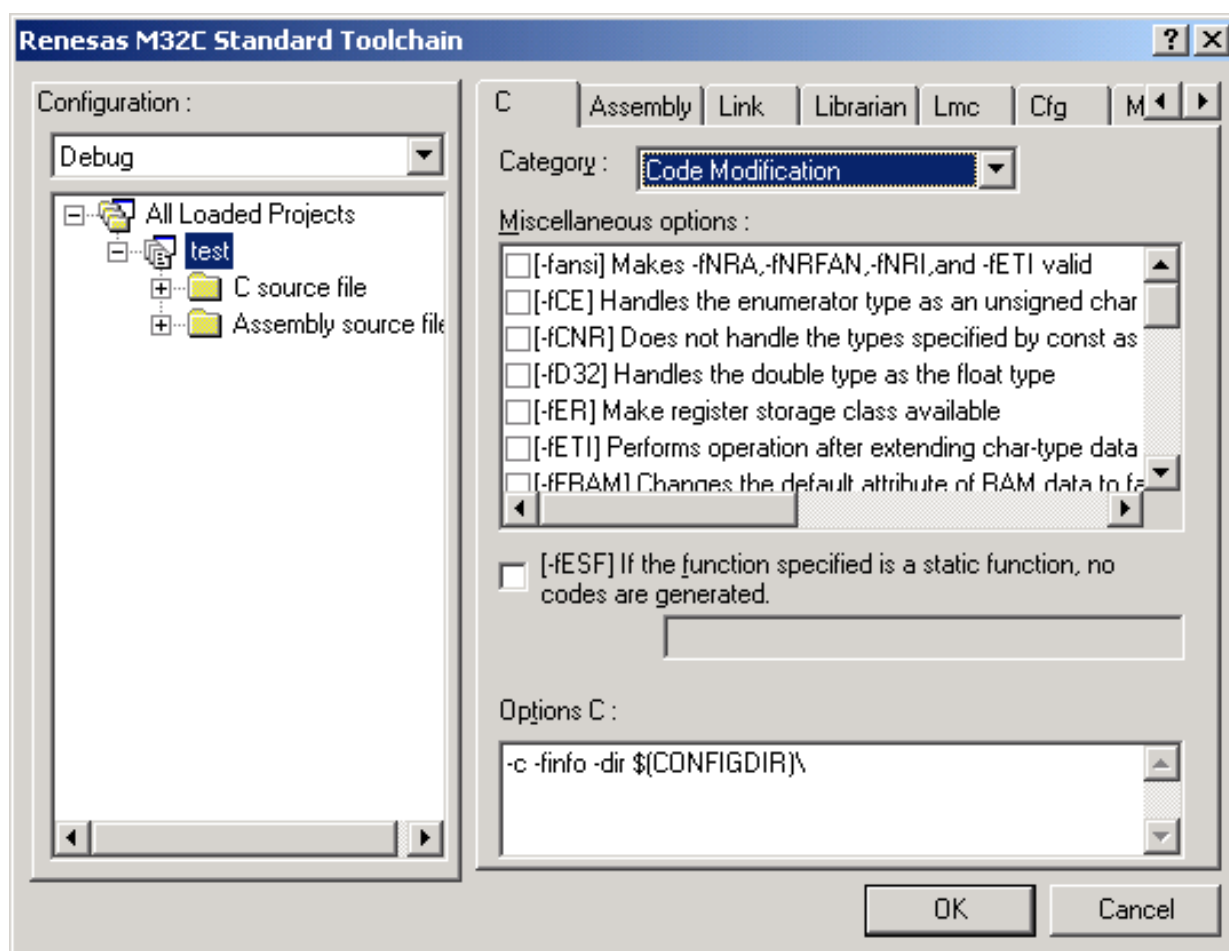


図 4.6 Category:[Code Modification]のダイアログボックス

(6) Category:[Warning]

表 4.6 Category:[Warning]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション	短縮形
Miscellaneous options :		
[-Wall] Become effective all options for warning	Wall	なし
[-WLTS] Outputs an alarm for implicit transfers from large size to smaller size	Wlarge_to_small	WLTS
[-WMT] Outputs error messages to every file	Wmake_tagfile	WMT
[-WNC] Outputs a warning for a comment including /*	Wnesting_comment	WNC
[-WNP] Outputs warning messages for functions without prototype declarations	Wnon_prototype	WNP
[-WNS] Prevents the compiler stopping when an error occurs	Wno_stop	WNS
[-WNUA] Outputs a warning to a function having an unused argument	Wno_used_argument	WNUA
[-WNUF] Outputs a warning for the unused function names.	Wno_used_function	WNUF
[-WNUSF] A static function name is output that does not require code generation	Wno_used_static_function	WNUSF
[-WNWS] Suppresses the warning for missing include file using standard library	Wno_warning_stdlib	WNWS
[-WSAL] Link processing is stopped when warning occurs at the time of a link.	Wstop_at_link	WSAL
[-WSAW] Stops the compiling process when a warning occurs	Wstop_at_warning	WSAW
[-Wstdout] Outputs error messages to the host machine's standard output (stdout)	Wstdout	なし
[-WUM] Output the warning for undefined macro in #if .	Wundefined_macro	WUM
[-WUP] Outputs warning messages for non-supported #pragma	Wunknown_pragma	WUP
[-WUV] Outputs the warning for uninitialized auto variables	Wuninitialize_variable	WUV
[-WCMW] Specifies the maximum number of warnings output by ccom30 :	Wccom_max_warnings=<数値>	WCMW=<数値>
[-WEF] Outputs error messages to the specified file :	Error_file=<ファイル名>	WEF=<ファイル名>

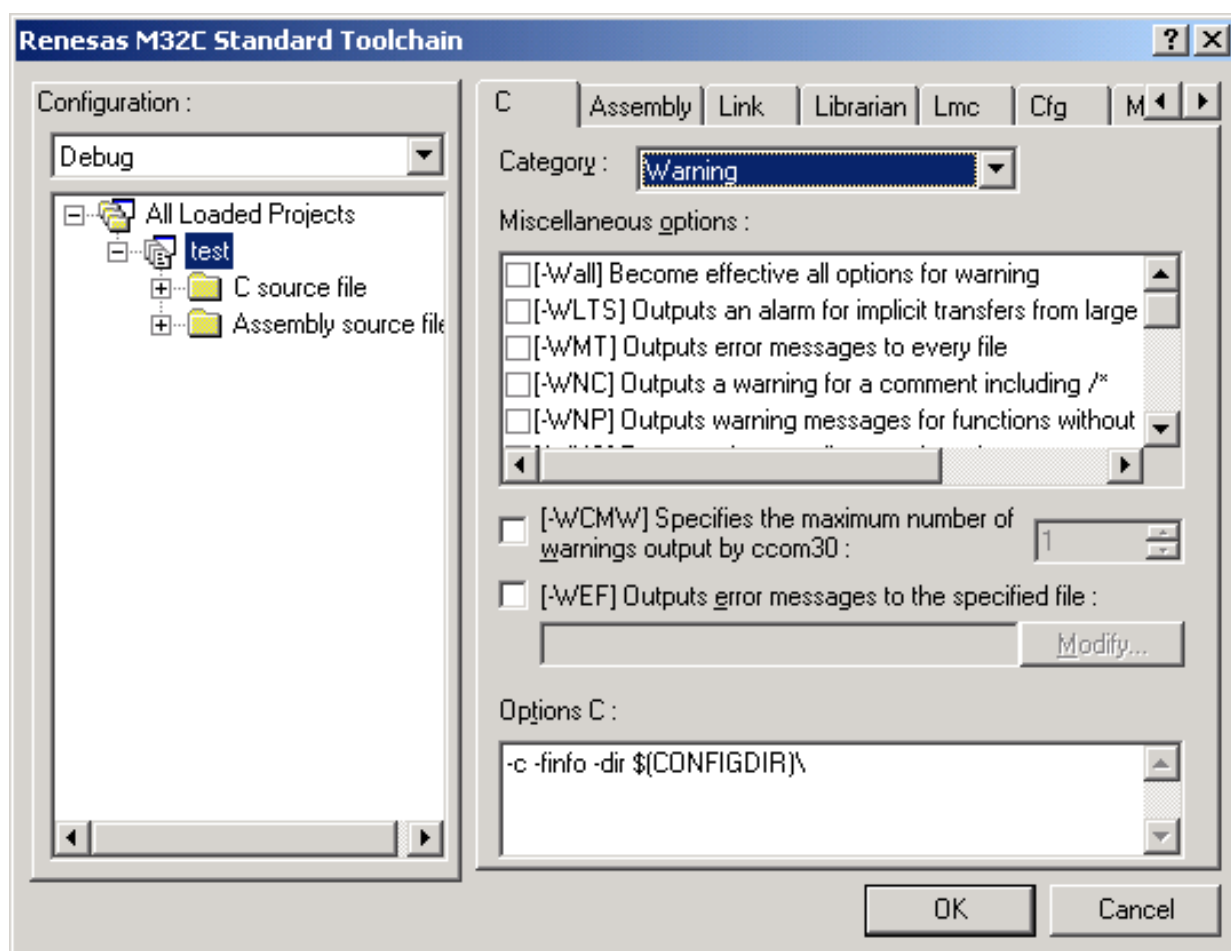


図 4.7 Category:[Warning]のダイアログボックス

(7) Category:[Other]

表 4.7 Category:[Other]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-silent] Suppresses the copyright message display at startup	silent
[-v] Displays the name of the command program and the command line during execution	v
[-V] Displays the startup messages of the compiler programs, then finishes processing (without compiling)	V
[-l] Library file :	l<ファイル名>
User defined options :	

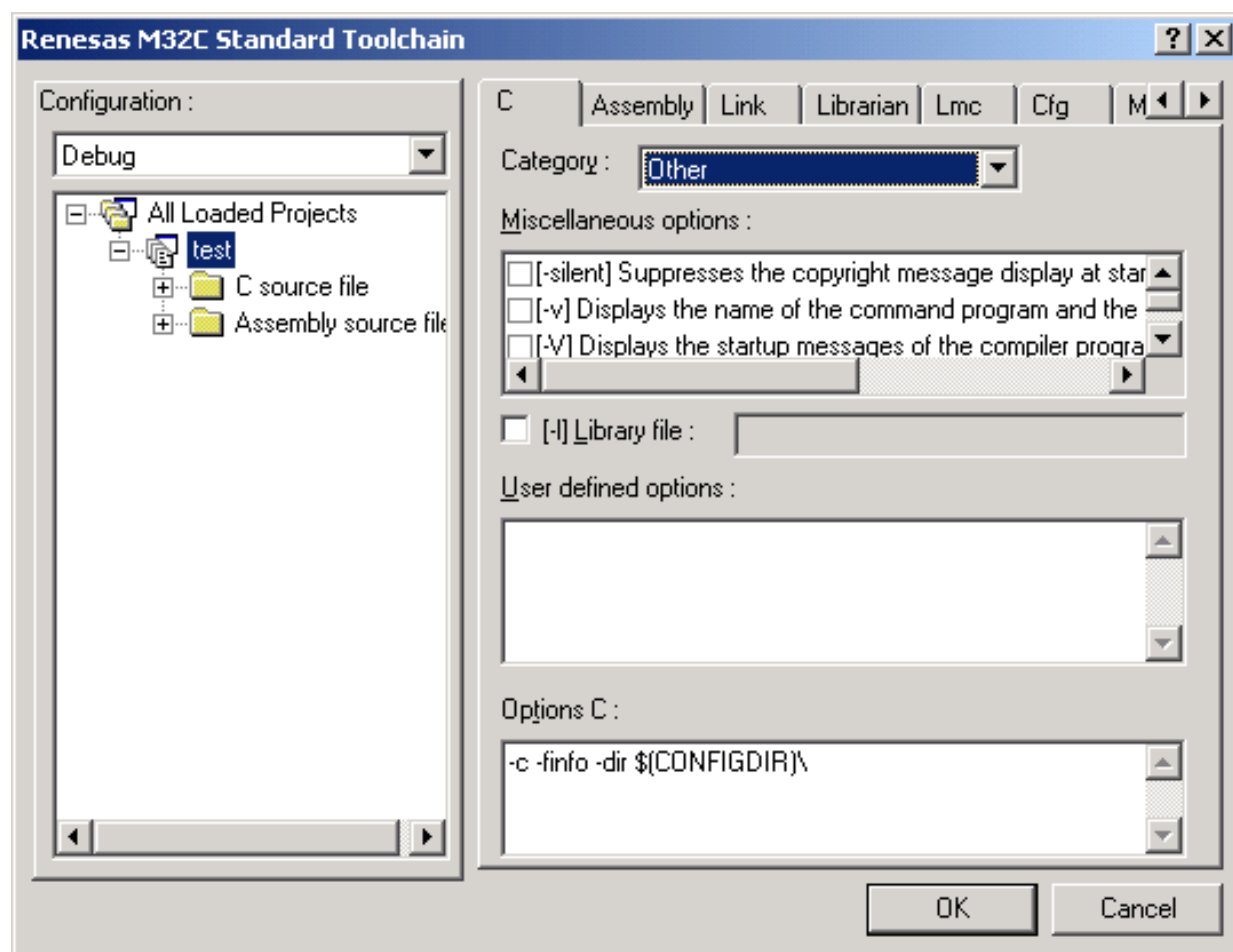


図 4.8 Category:[Other]のダイアログボックス

4.1.2 アセンブラのオプション

Renesas M32C Standard Toolchain ダイアログボックスから Assembly タブを選択します。

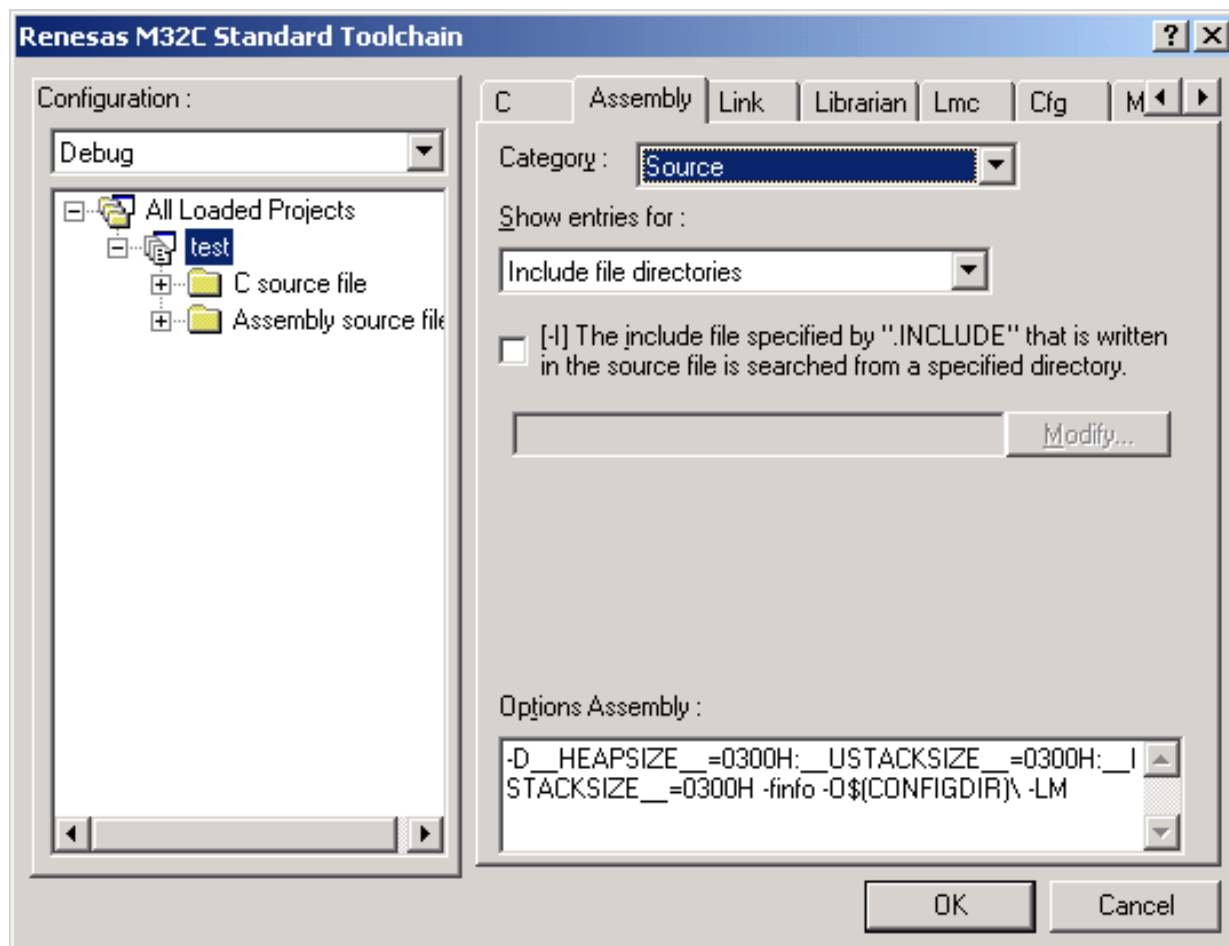


図 4.9 Assembly タブのダイアログボックス

(1) Category:[Source]

表 4.8 Category:[Source]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
Show entries for :	
Include file directories	I<ディレクトリ名>
[-I] The include file specified by ".INCLUDE" that is written in the source file is searched from a specified directory.	
Defines	
[-D__HEAP__=1] Disable heap are in startup (ncrt0.a30).	D__HEAP__=1
[-D__STANDARD_IO__=1] Enable initialization for standard I/O library.	D__STANDARD_IO__=1
[-D] Sets constants to symbols :	D<sub>>
	<sub> : <マクロ名> [= <文字列>]

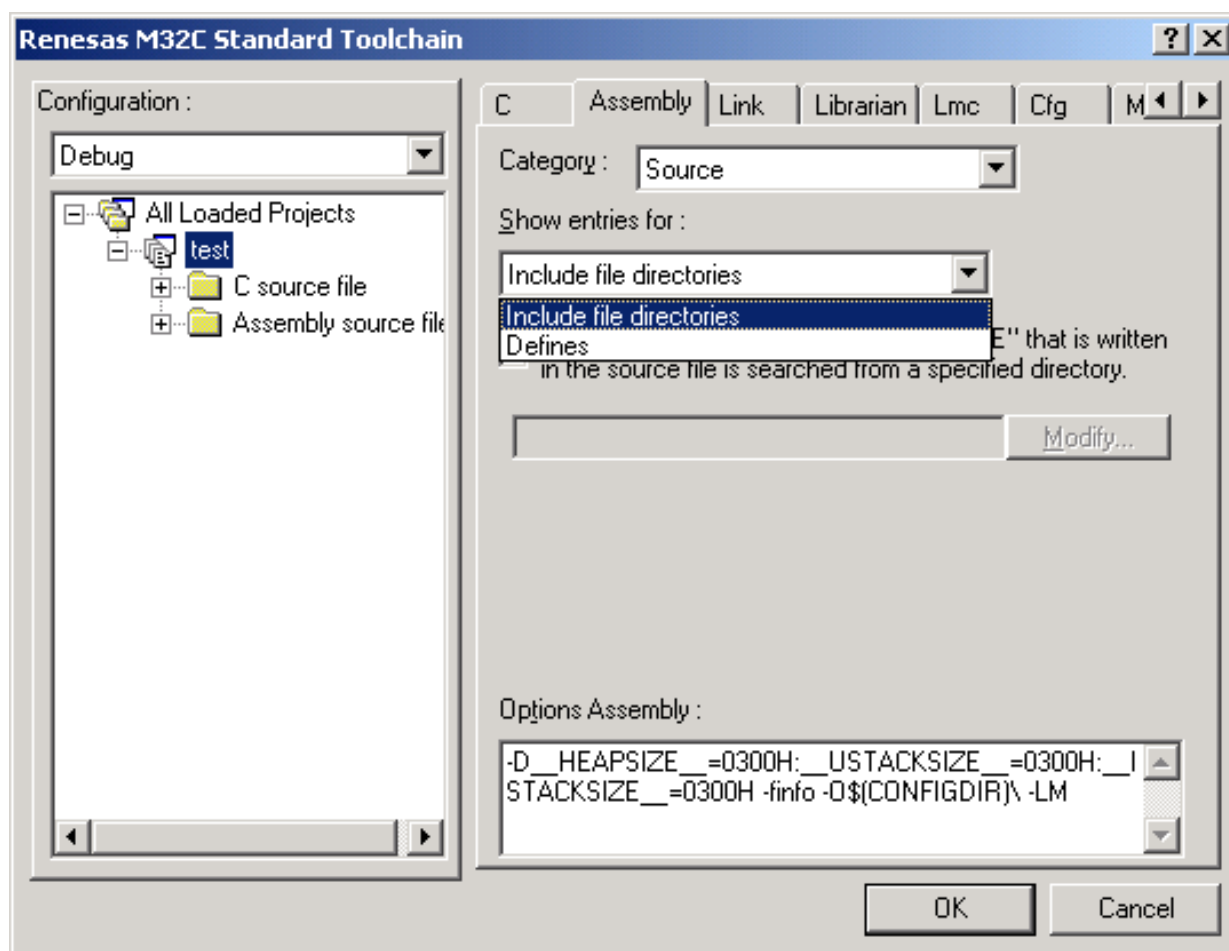


図 4.10 Category:[Source]のダイアログボックス

(2) Category:[Object]

表 4.9 Category:[Object]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
[-S] Specifies the local symbol information be output.	S
[-SM] Specifies system label and local symbol information output.	SM
[-finfo] Generates inspector information.	finfo
[-N] Disables output of macro command line information.	N
[-mode60] Running AS308 with this parameter to process a program written in AS30 allows some code to be assembled by AS308.	mode60
[-mode60p] Runs structured processor(pre30) and processes parameter -mode60.	mode60p
[-M] Generates structured description command variables in byte type.	M
[-O] Output file directory :	O<ディレクトリ名>

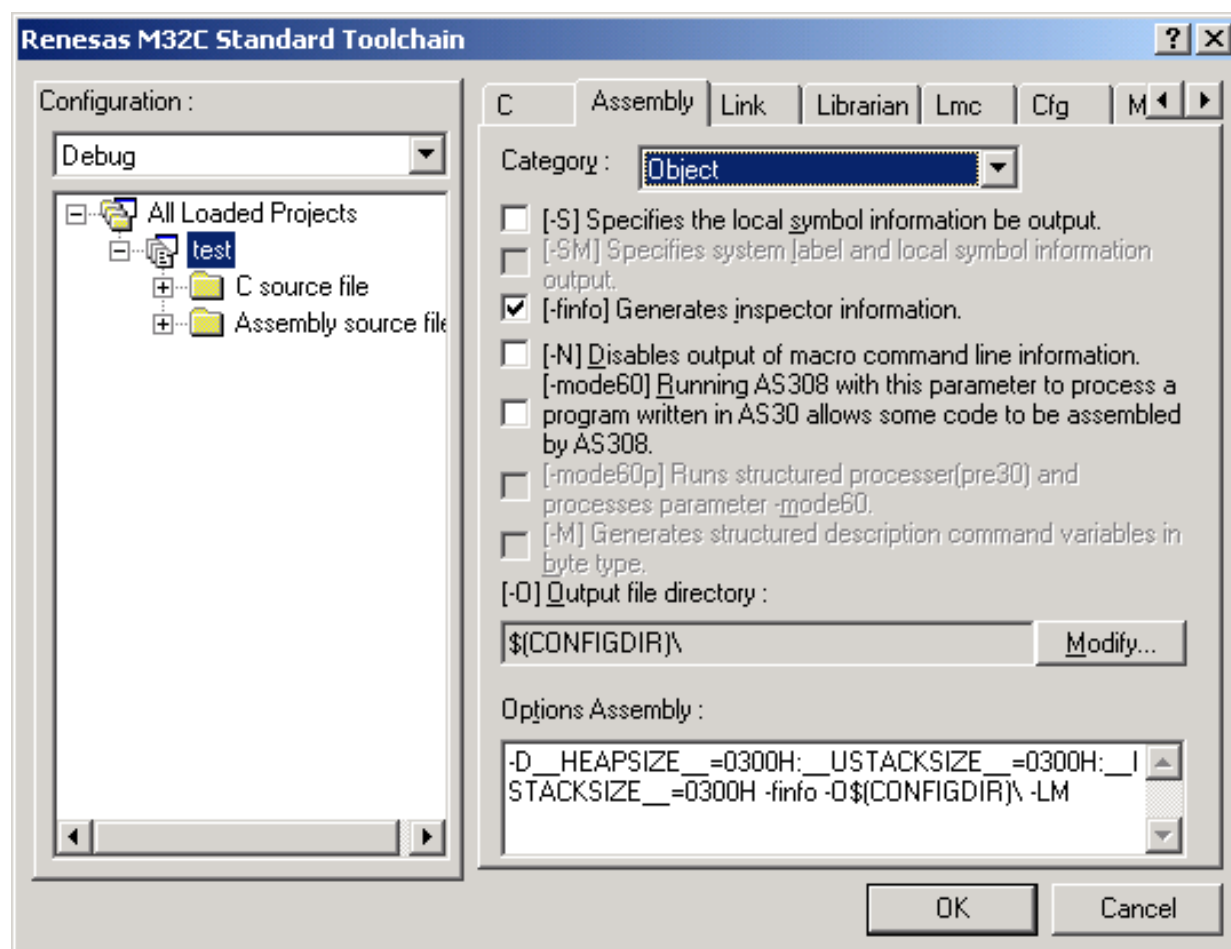


図 4.11 Category:[Object]のダイアログボックス

(3) Category:[List]

表 4.10 Category:[List]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
[-L] Generates assembler list file.	L
File format : *1	
[+C] Line concatenation is output directly as is to a list file	LC
[+D] Information before .DEFINE is replaced is output to a list file	LD
[+I] Even program sections in which condition assemble resulted in false conditions are output to the assembler list file	LI
[+M] Even macro description expansion sections are output to the assembler list file	LM
[+S] Even structured description expansion sections are output to the assembler list file	LS
[-H] Header information is not output to an assembler list file.	H
*1 : L の後にチェックした項目を追記したものが、オプションとして指定される。 例) [+C] [+D] [+I]チェック : -LCDI	

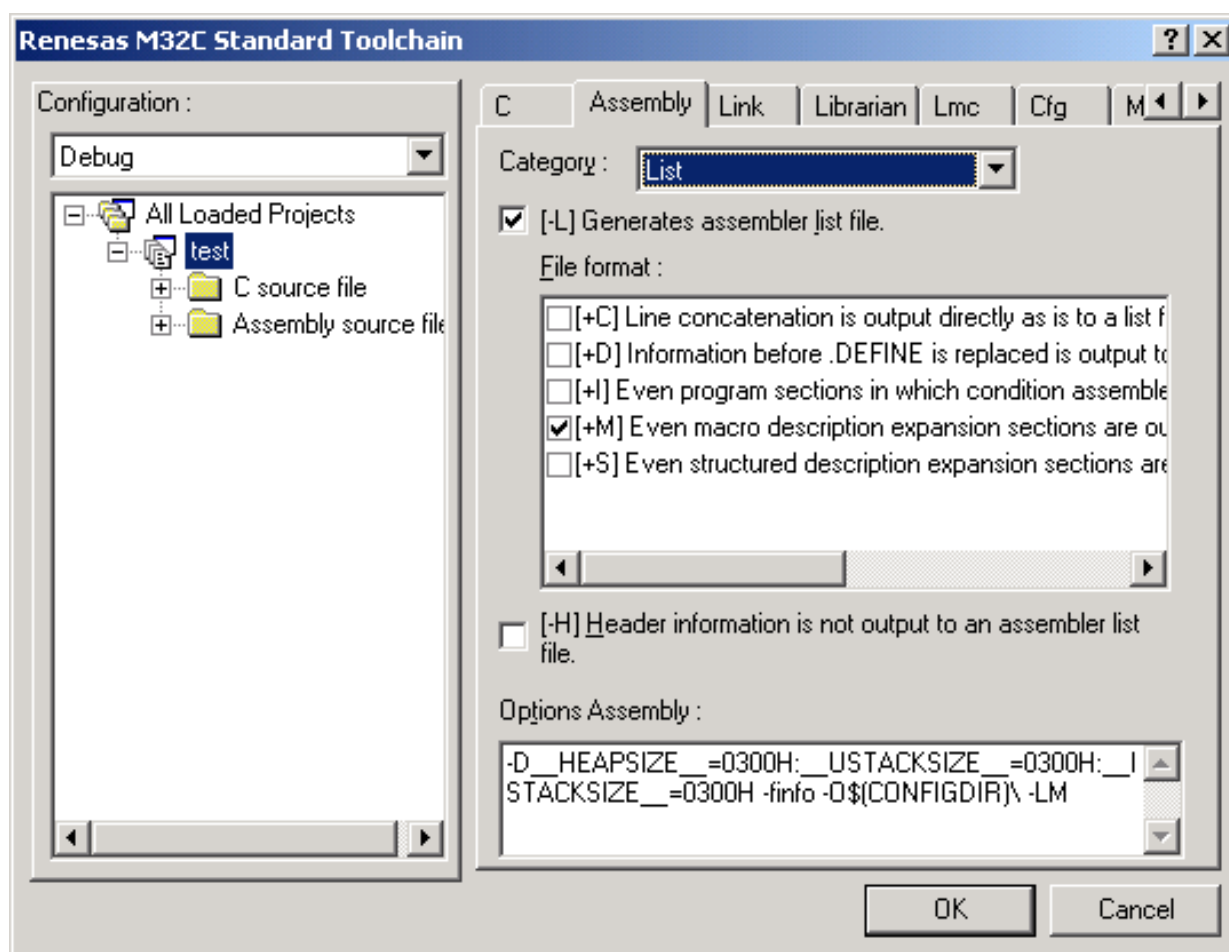


図 4.12 Category:[List]のダイアログボックス

(4) Category:[Tuning]

表 4.11 Category:[Tuning]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
[-fMST] Generates special page vector table.	fMST
[-fMVT] Generates variable vector table.	fMVT
[-abs16] Selects 16-bit absolute addressing.	abs16
[-JOPT] Optimizes the branch instrument which refers to the global label.	JOPT
[-PATCH_TA] Escaping precautions No.1 on the timer functions for three-phase motor control : Number :	PATCH_TA[n] [n] : Number で指定した数値

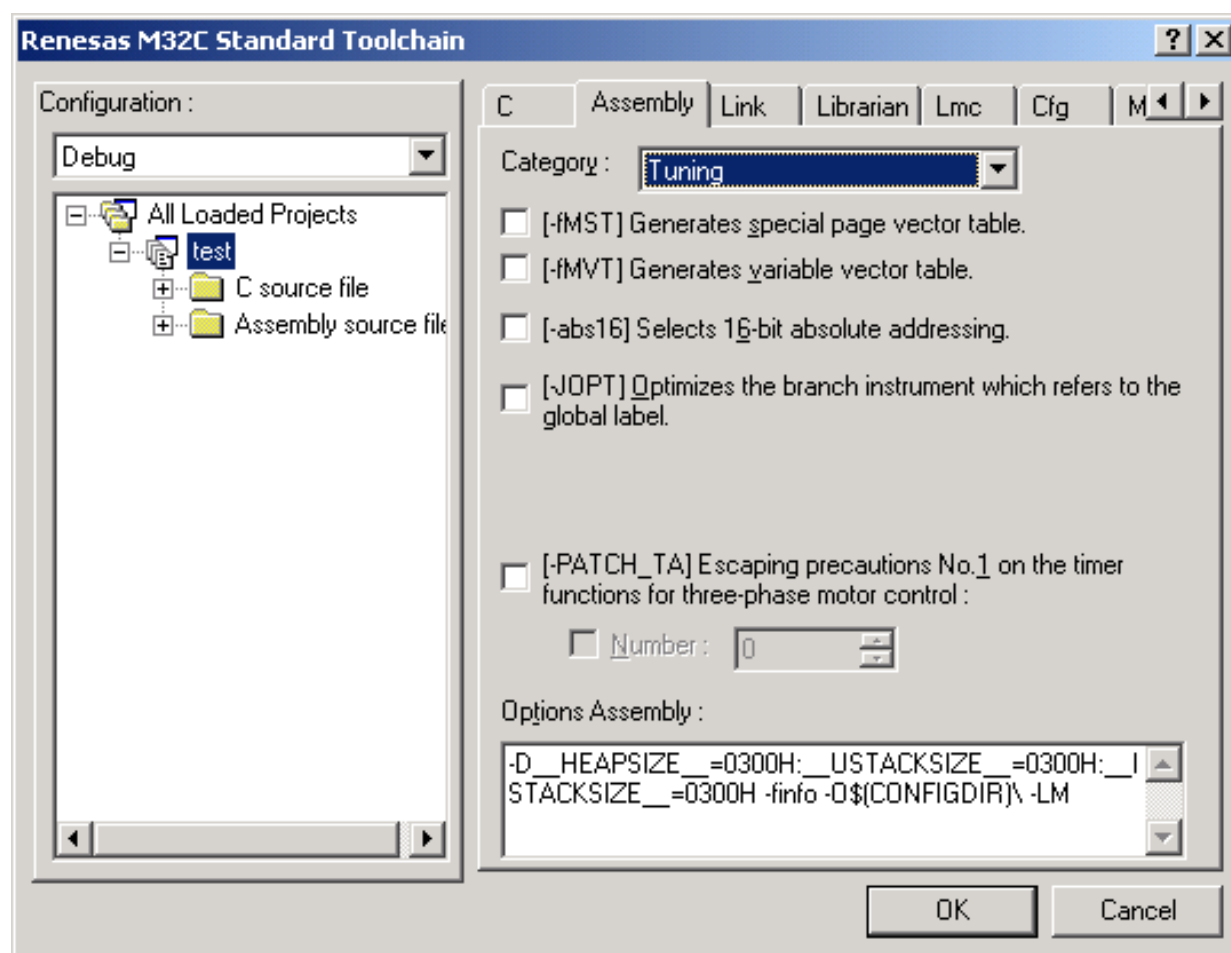


図 4.13 Category:[Tuning]のダイアログボックス

(5) Category:[Other]

表 4.12 Category:[Other]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-.] Disables message output to a display screen	.
[-C] Indicates contents of command lines when as30 has invoked mac30 and asp30	C
[-F] Fixes the file name of ..FILE expansion to the source file name	F
[-T] Generates assembler error tag file	T
[-V] Indicates the version of the assembler system program	V
User defined options :	

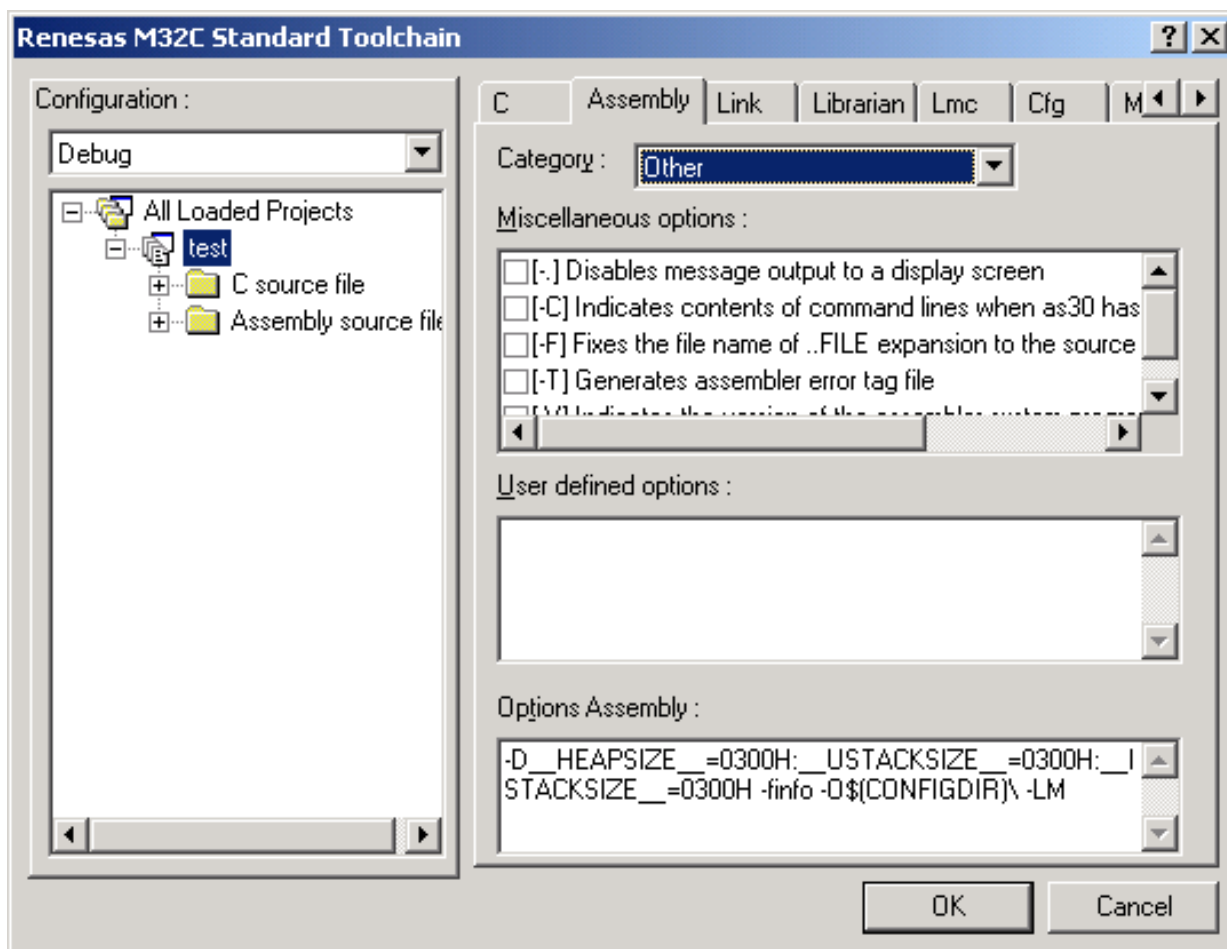


図 4.14 Category:[Other]のダイアログボックス

4.1.3 リンケージエディタのオプション

Renesas M32C Standard ToolchainダイアログボックスからLinkタブを選択します。

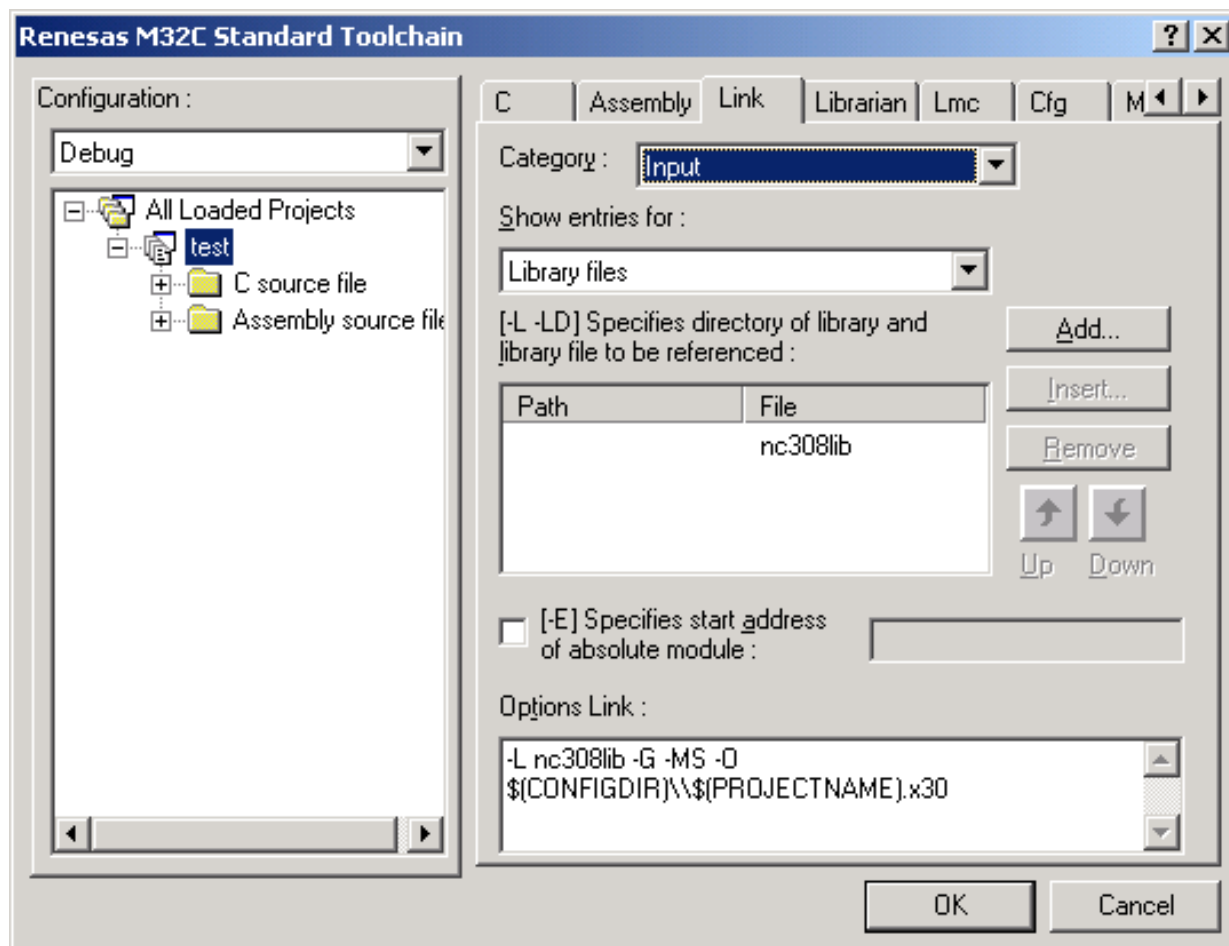


図 4.15 Link タブダイアログボックス

(1) Category:[Input]

表 4.13 Category:[Input]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
Show entries for : Library files	L△<ファイル名> LD△<ディレクトリ名>
[-E] Specifies start address of absolute module :	E△<sub> <sub> : <数値> <ラベル名>

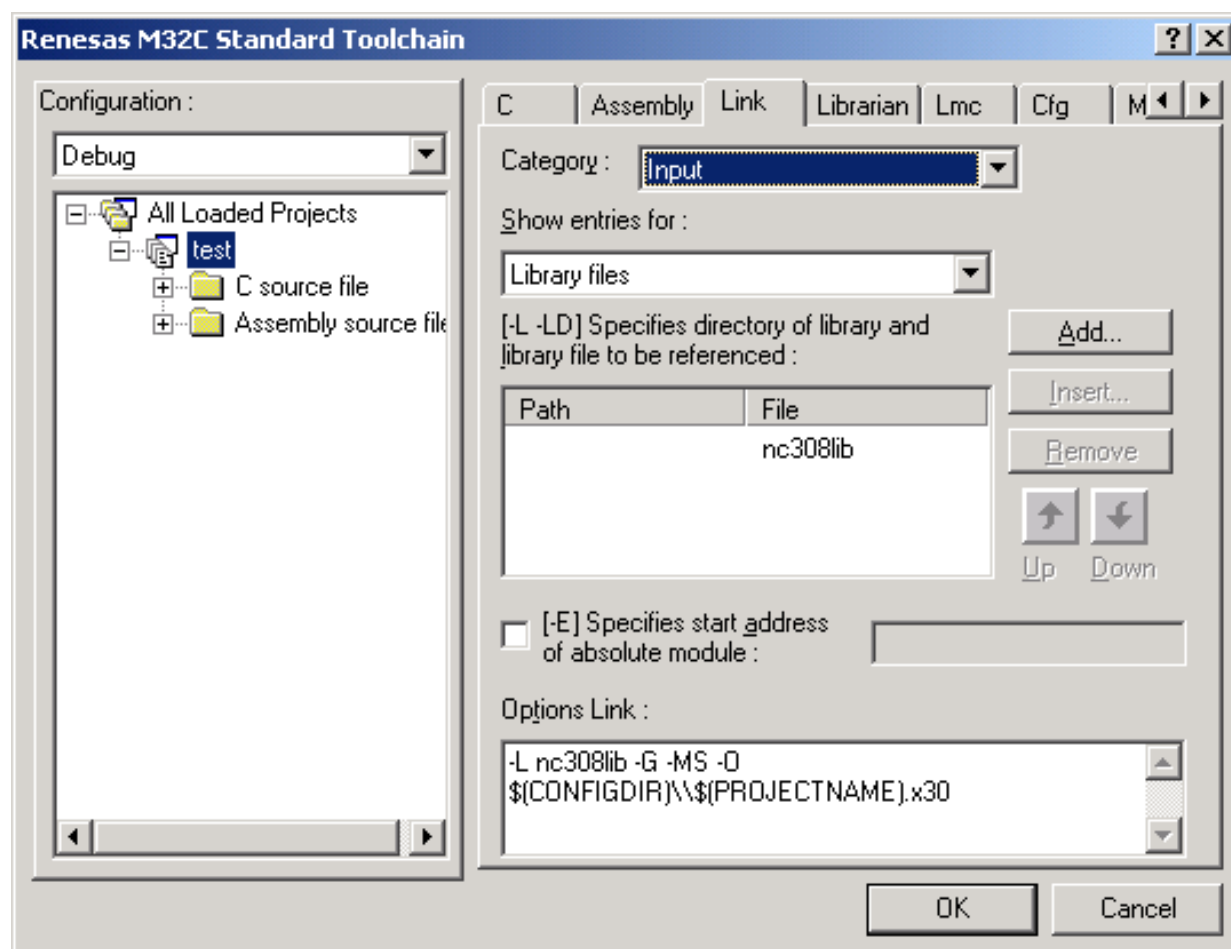


図 4.16 Category:[Input]のダイアログボックス

(2) Category:[Output]

表 4.14 Category:[Output]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
[-G] Outputs source debug information to absolute module file.	G
[-U] Outputs a warning for the unused function names.	U
[-W] Link processing is stopped when warning occurs at the time of a link.	W
Generate map file :	
None	-
[-M] Generates map file	M
[-MS] Includes symbol information	MS
[-MSL] Includes the fullname of symbol information	MSL
[-O] Specifies absolute module file name :	O△<ファイル名>

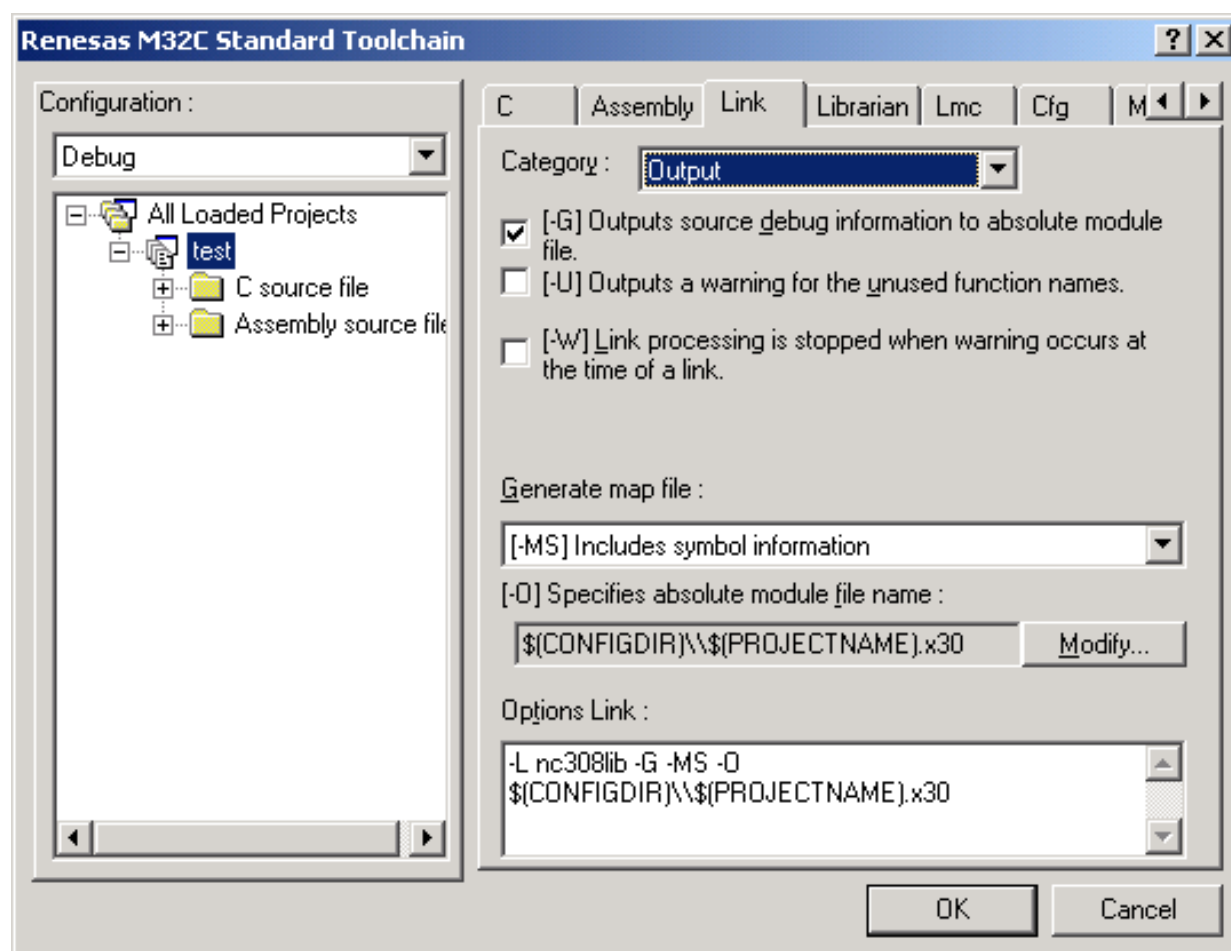


図 4.17 Category:[Output]のダイアログボックス

(3) Category:[Tuning]

表 4.15 Category:[Tuning]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
[-fMST] Generates special page vector table.	fMST
[-fMVT] Generates variable vector table.	fMVT
[-VECT] Sets the address to the free area at the result of performing automatic generation of a variable vector table.	VECT△<sub> <sub> : <数値> <ラベル名>
[-JOPT] Optimizes the branch instrument which refers to the global label.	JOPT

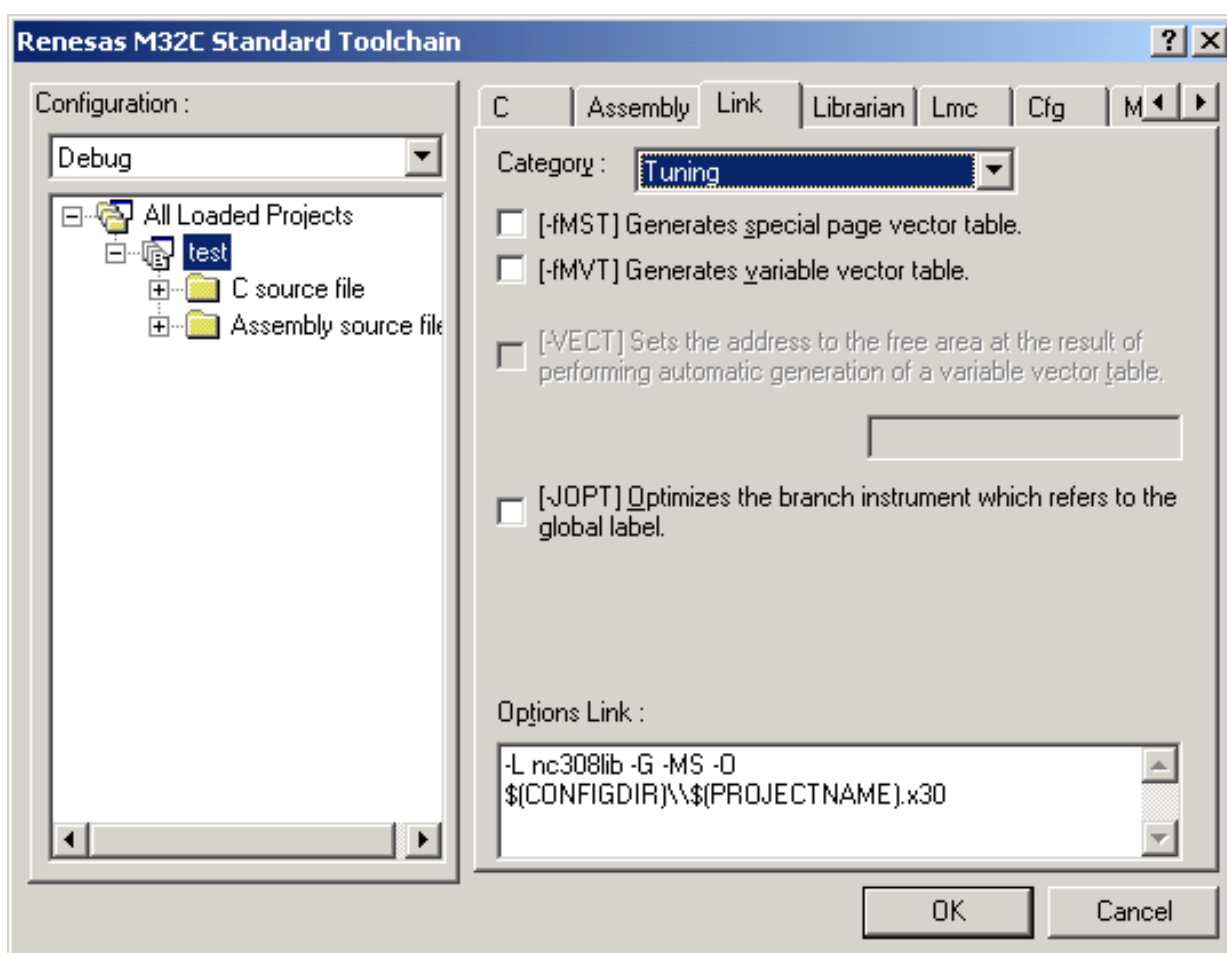


図 4.18 Category:[Tuning]のダイアログボックス

(4) Category:[Section]

表 4.16 Category:[Section]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
Show entries for : Section Order	ORDER△<sub>[,...] <sub> : <セクション名> [=アドレス]
Section Location	LOC△<sub>[,...] <sub> : <セクション名> =<アドレス>

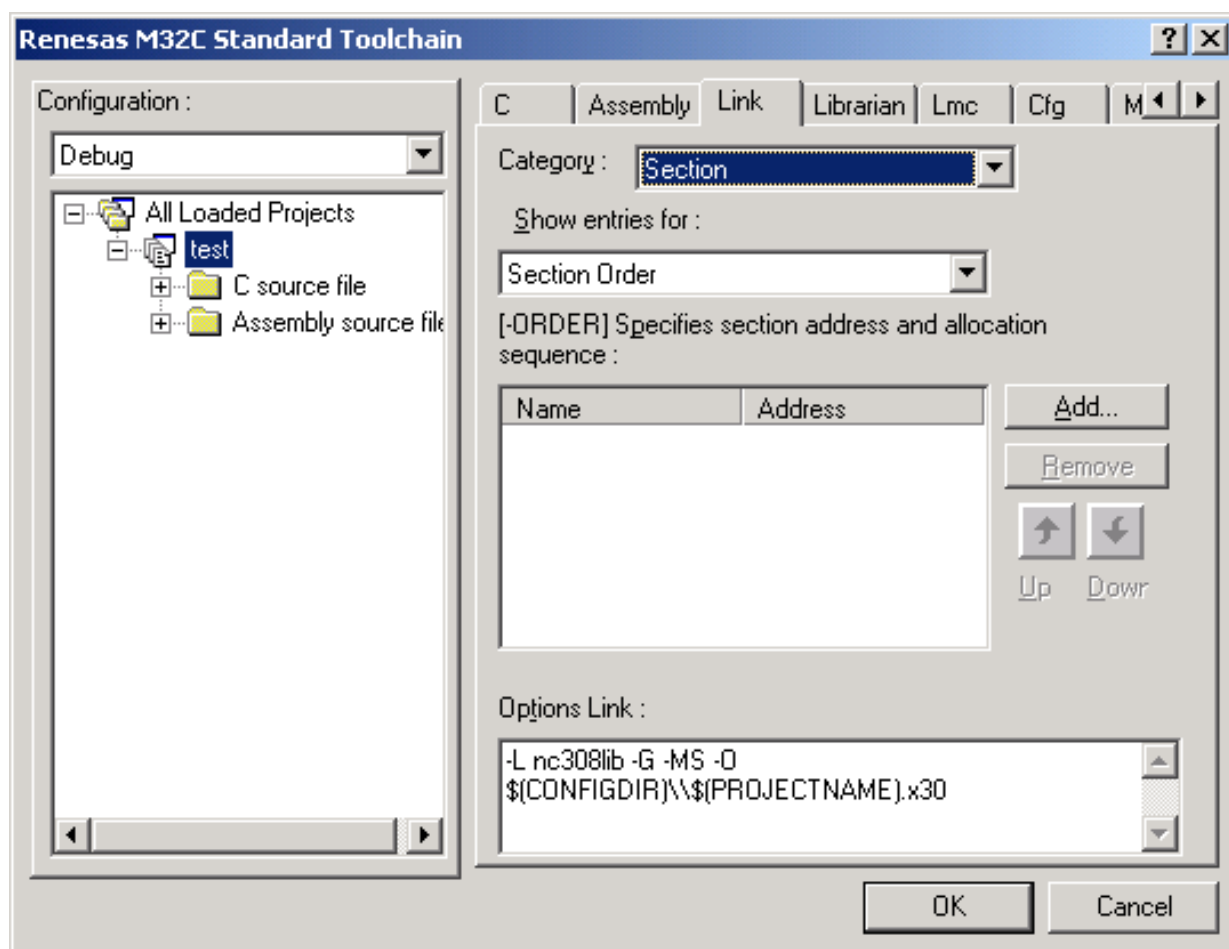


図 4.19 Category:[Section]のダイアログボックス

(5) Category:[Other]

表 4.17 Category:[Other]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-.] Disables message output to screen	.
[-NOSTOP] Outputs all encountered errors to screen	NOSTOP
[-T] Generates link error tag file	T
[-V] Indicates version number of linkage editor	V
User defined options :	

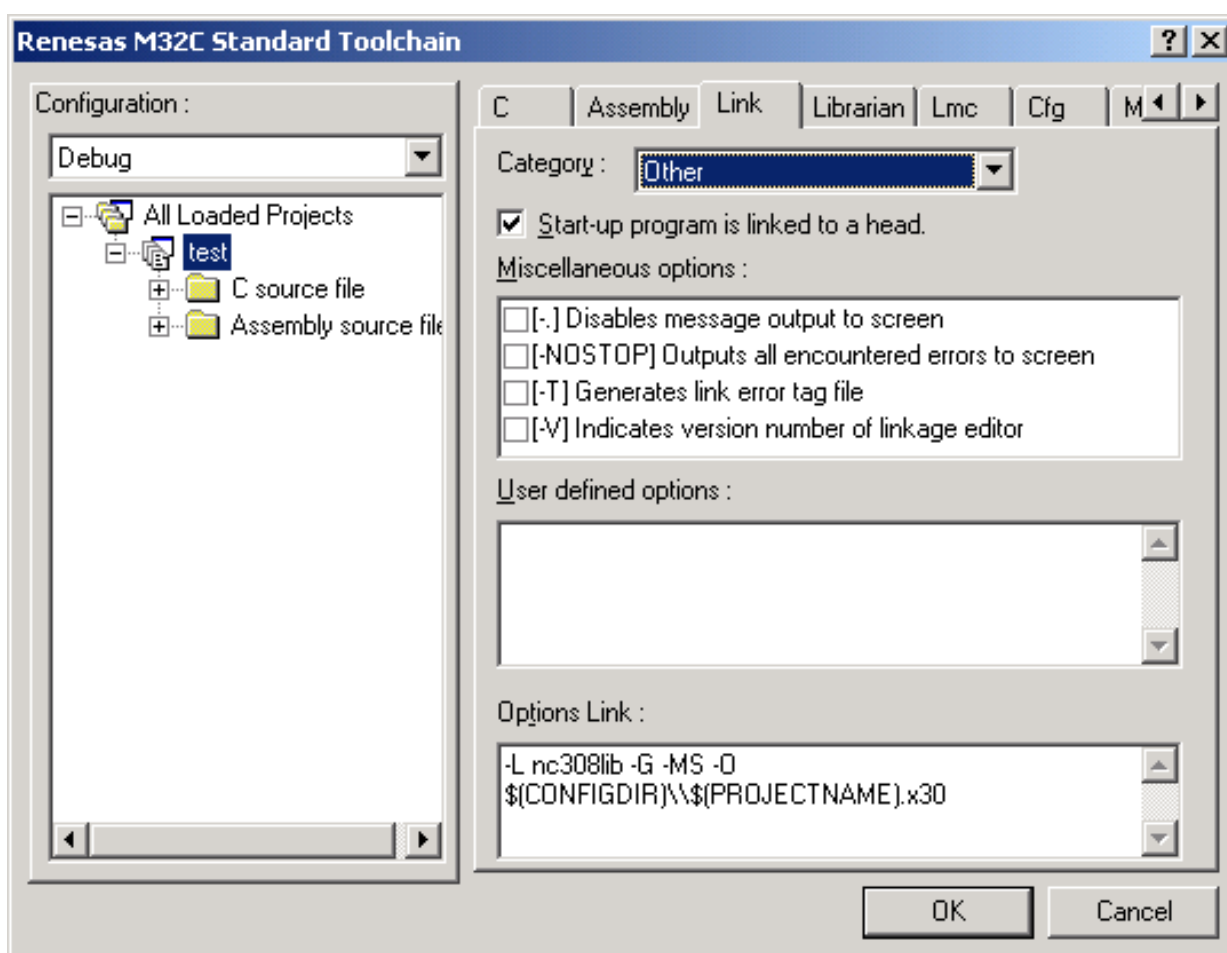


図 4.20 Category:[Other]のダイアログボックス

(6) Category:[Subcommand file]

表 4.18 Category:[Subcommand file]の項目名とリンケージエディタオプションの対応表

ダイアログボックス	オプション
[@] Use external subcommand file.	@<ファイル名>

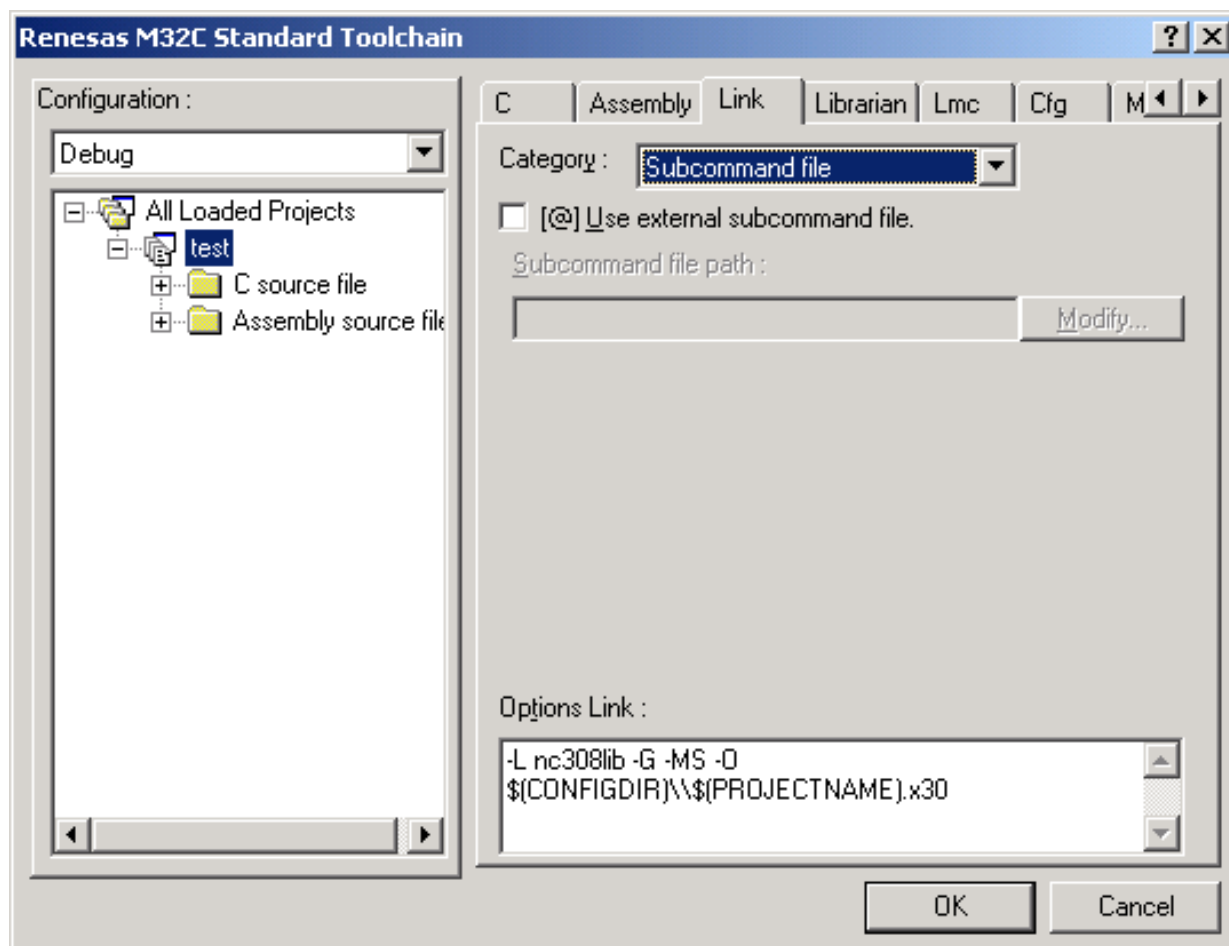


図 4.21 Category:[Subcommand file]のダイアログボックス

4.1.4 ライブラリアンのオプション

Renesas M32C Standard Toolchain ダイアログボックスから Librarian タブを選択します。

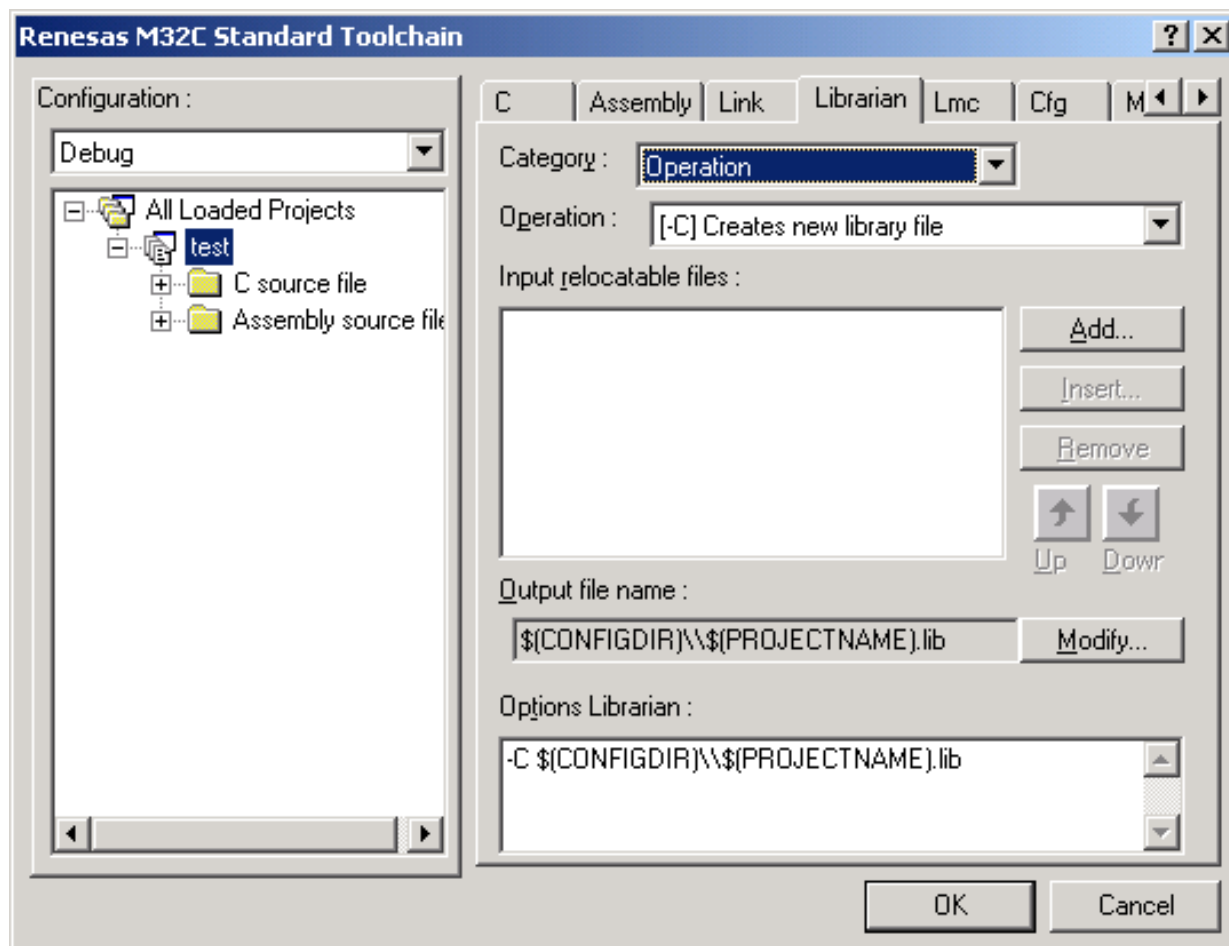


図 4.22 Librarian タブのダイアログボックス

(1) Category:[Operation]

表 4.19 Category:[Operation]の項目名とライブラリアンオプションの対応表

ダイアログボックス	オプション
Operation	
[-A] Adds modules to library file	A
[-C] Creates new library file	C
[-D] Deletes modules from library file	D
[-L] Generates library list file	L
[-R] Replaces modules	R
[-U] Updates modules	U
[-X] Extract	X

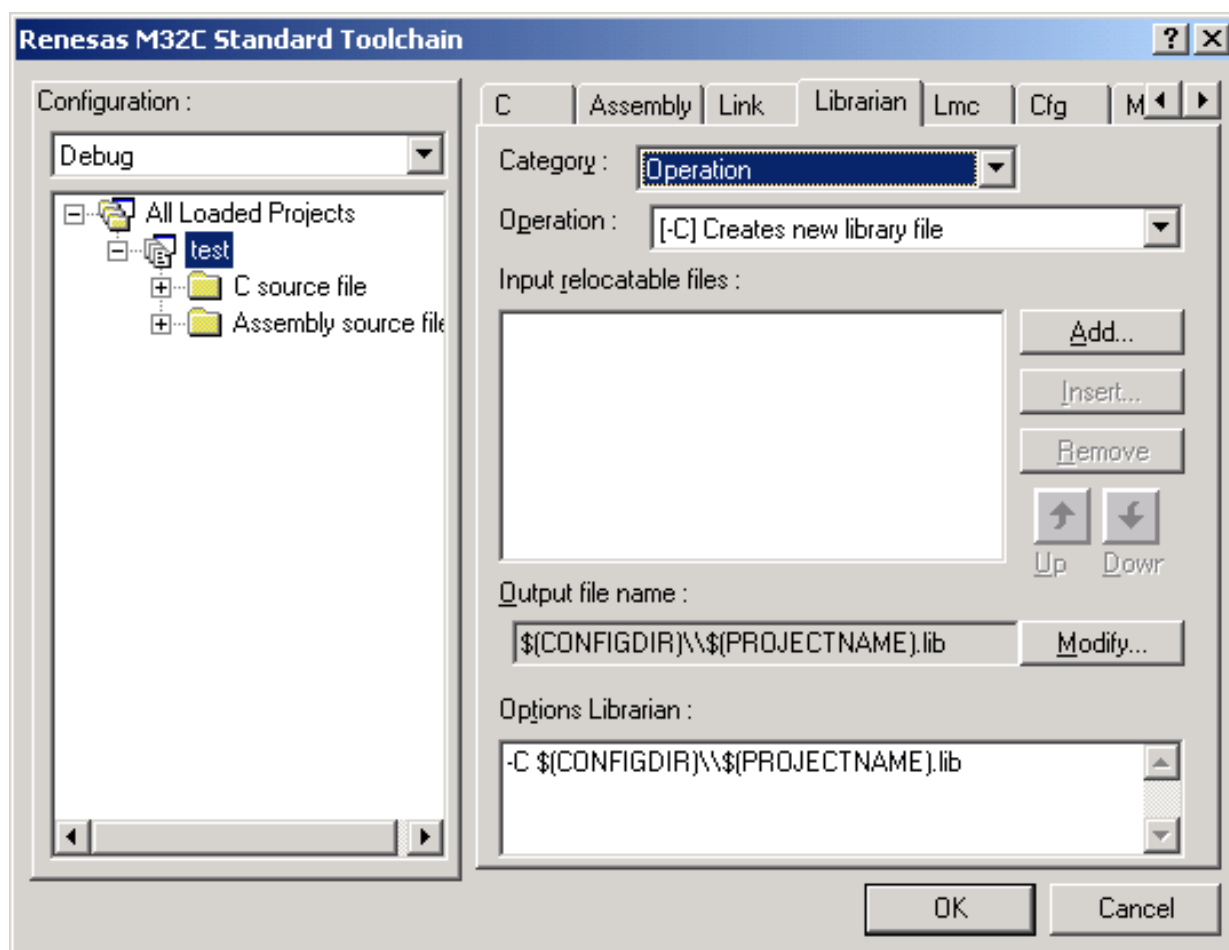


図 4.23 Category:[Operation]のダイアログボックス

(2) Category:[Other]

表 4.20 Category:[Other]の項目名とライブラリアンオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-.] Disables message output to screen	.
[-V] Indicates version of librarian	V
User defined options :	

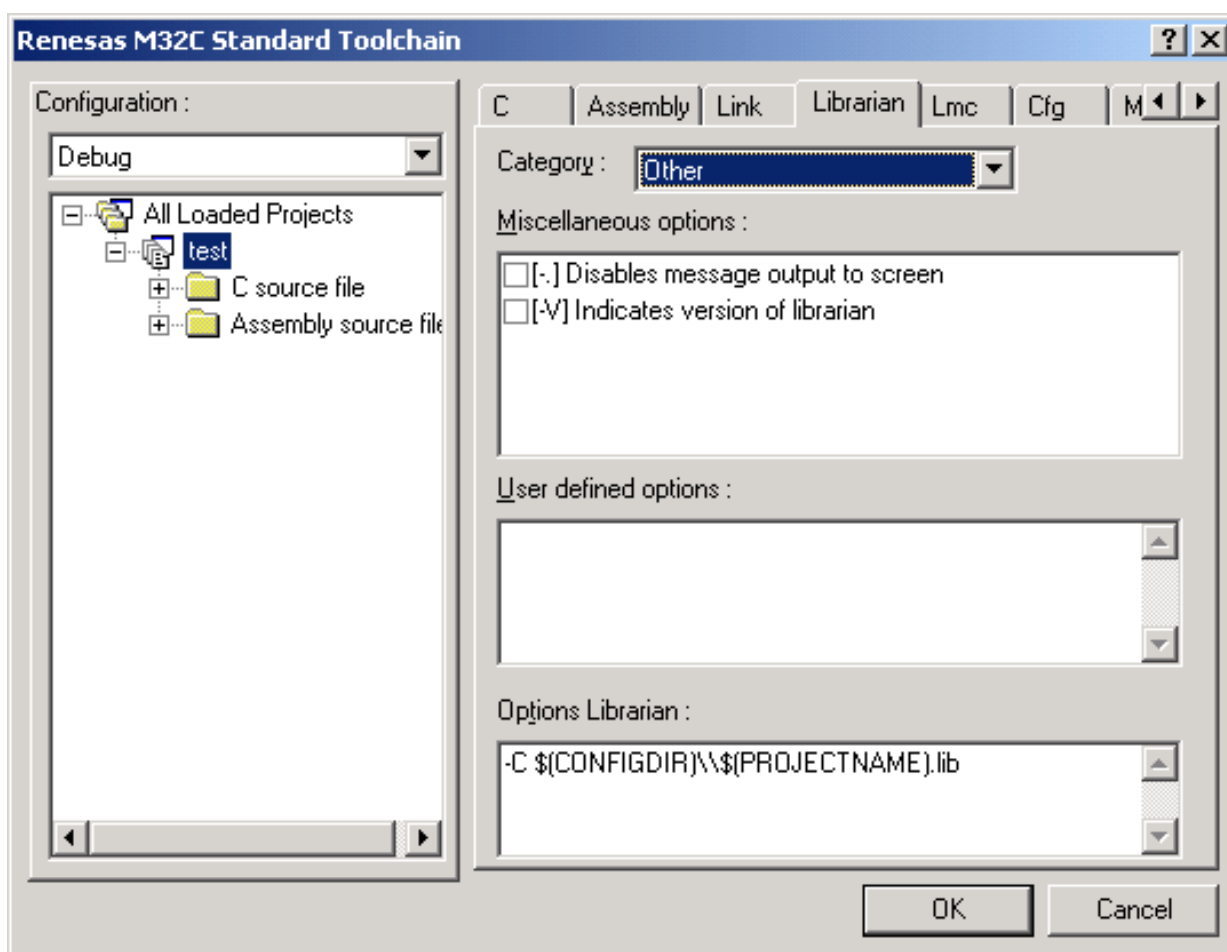


図 4.24 Category:[Other]のダイアログボックス

(3) Category:[Subcommand file]

表 4.21 Category:[Subcommand file]の項目名とオプションの対応表

ダイアログボックス	オプション
[@] Use external subcommand file.	@<ファイル名>

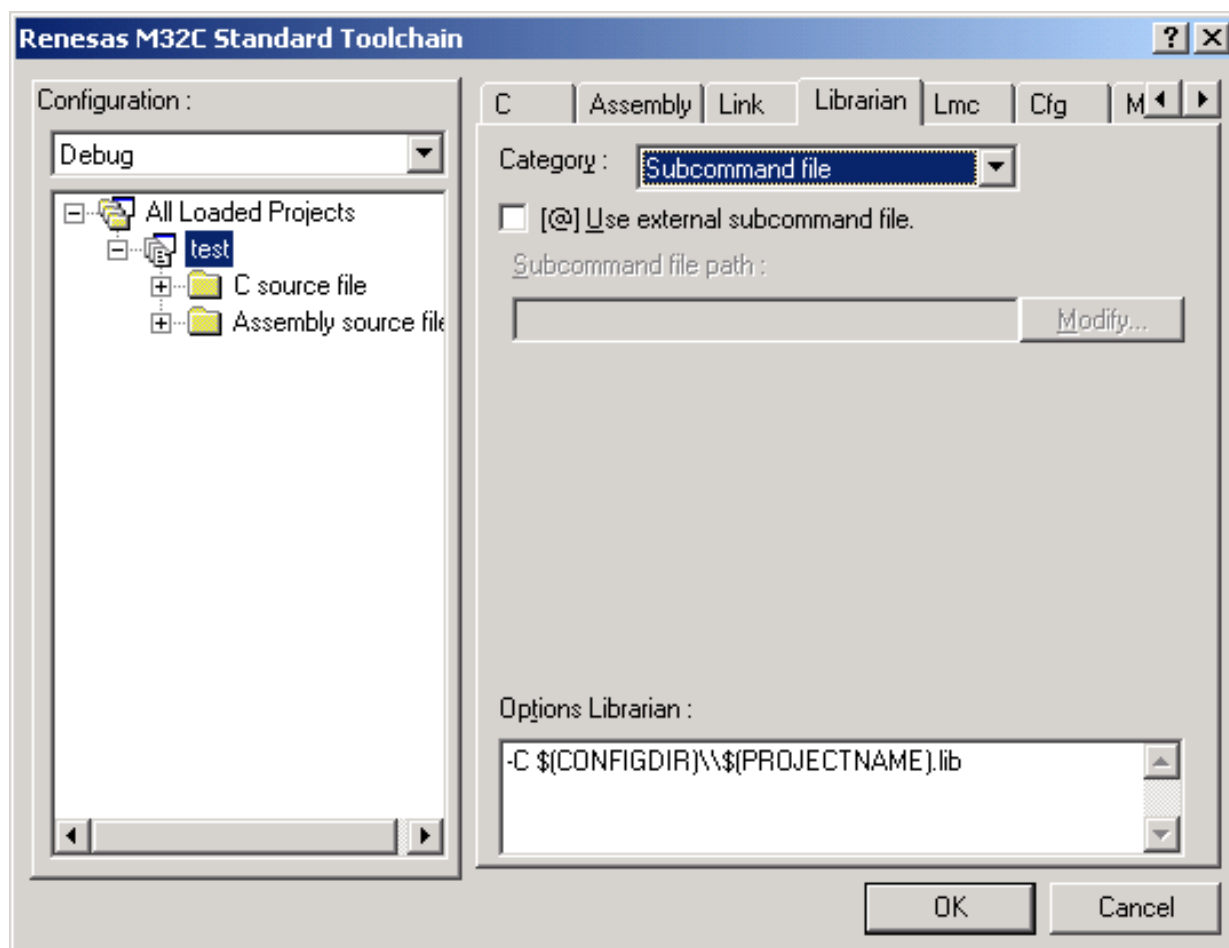


図 4.25 Category:[Subcommand file]のダイアログボックス

4.1.5 ロードモジュールコンパータのオプション

Renesas M32C Standard Toolchain ダイアログボックスから Lmc タブを選択します。

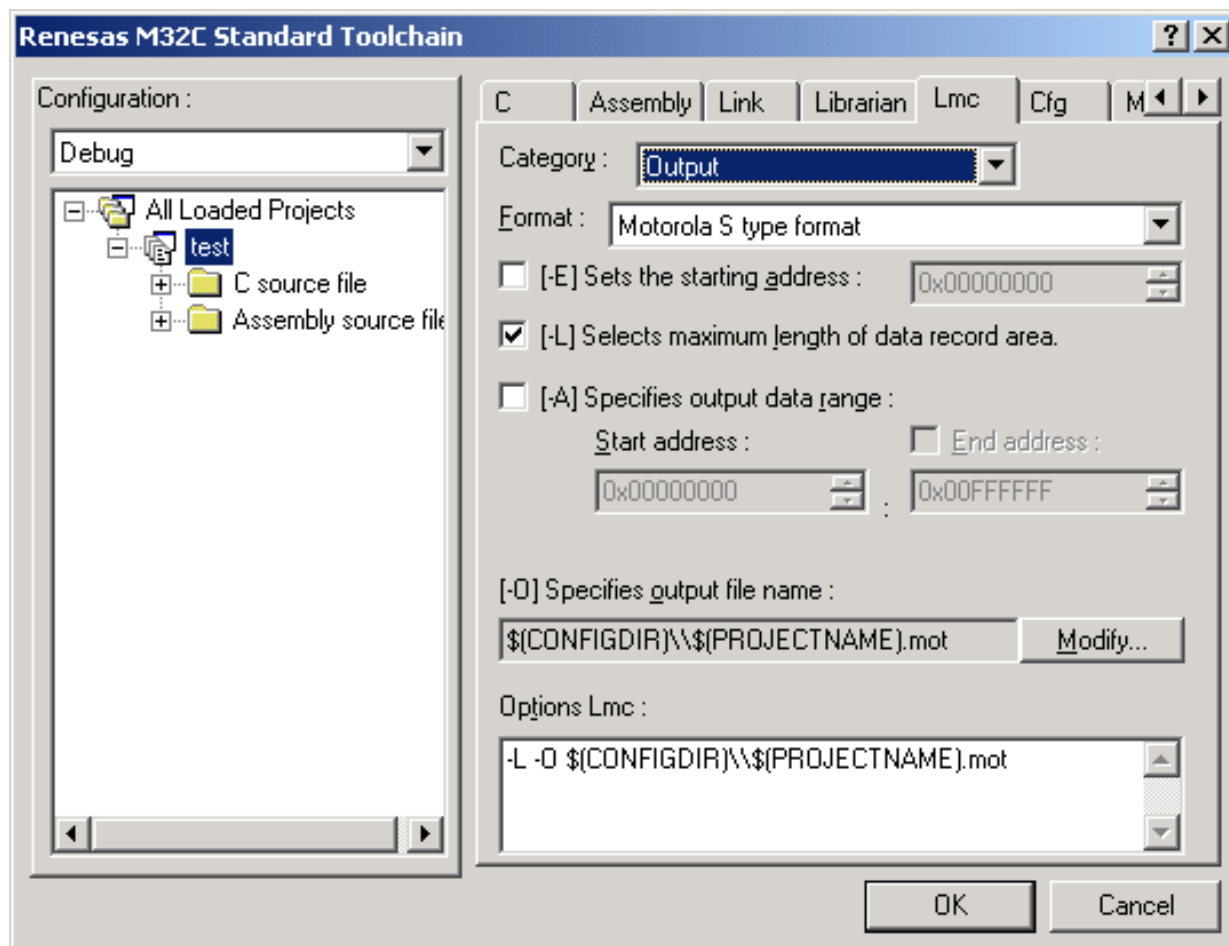


図 4.26 Lmc タブのダイアログボックス

(1) Category:[Output]

表 4.22 Category:[Output]の項目名とロードモジュールコンバータオプションの対応表

ダイアログボックス	オプション
Format :	
Motorola S type format	-
[-H] Converts file info Intel HEX format	H
[-E] Sets the starting address :	E△<アドレス値>
[-L] Selects maximum length of data record area.	L
[-A] Specifies output data range :	A△<開始アドレス値>[:終了アドレス値]
[-O] Specifies output file name :	O△<ファイル名>

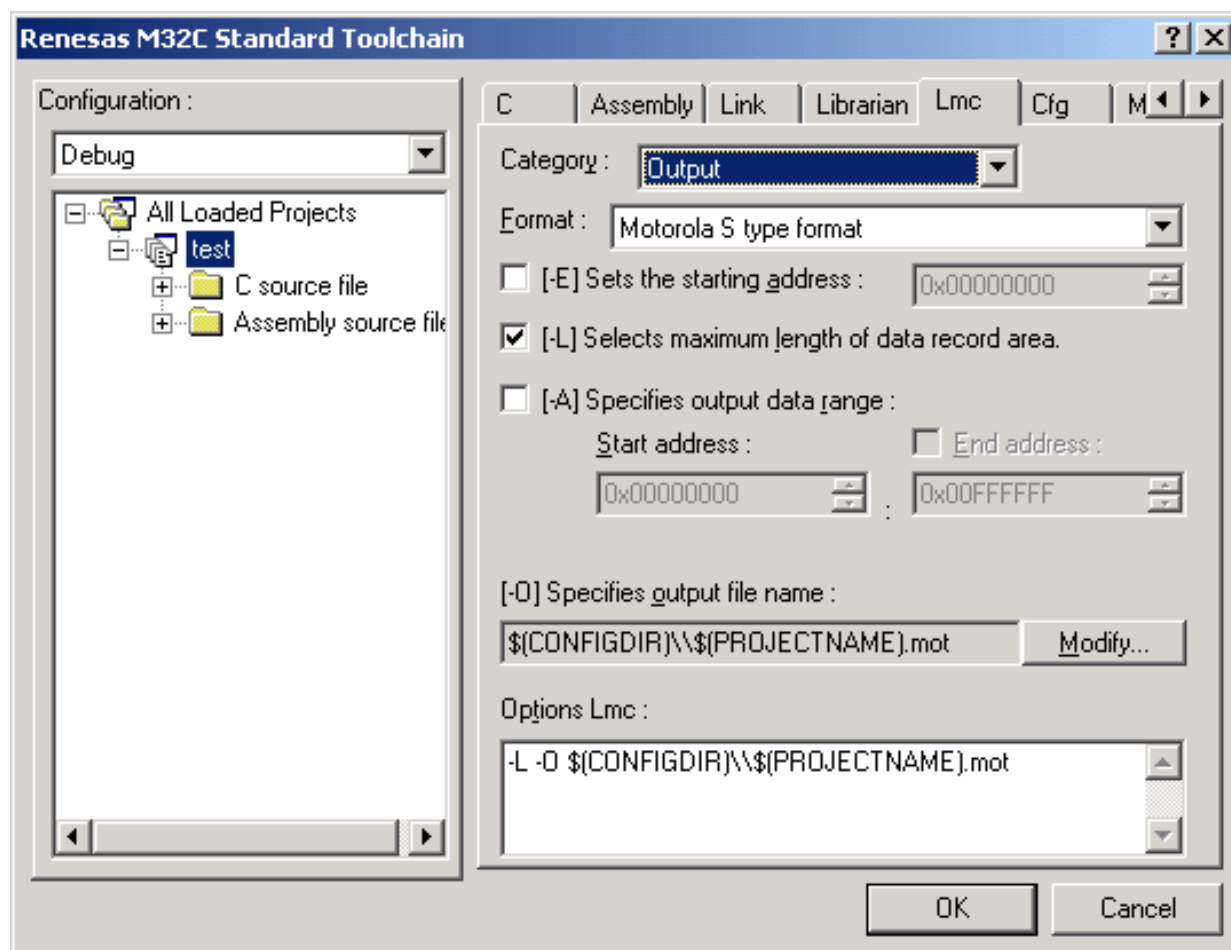


図 4.27 Category:[Output]のダイアログボックス

(2) Category:[Code]

表 4.23 Category:[Code]の項目名とロードモジュールコンバータオプションの対応表

ダイアログボックス	オプション
[-ID] ID code check ID code setting :	ID[sub] [sub] : <コードプロテクト値> <#数値>
ROM code protect function : [-protect1] Level 1 setting [-protect2] Level 2 setting [-protectx] Set the ROM code protect value	protect1 protect2 protectx△<数値>
[-F] Sets the fill data in the free area :	F△<空き領域設定データ値>[sub] [sub] : <:開始アドレス値>[:終了アドレス値]

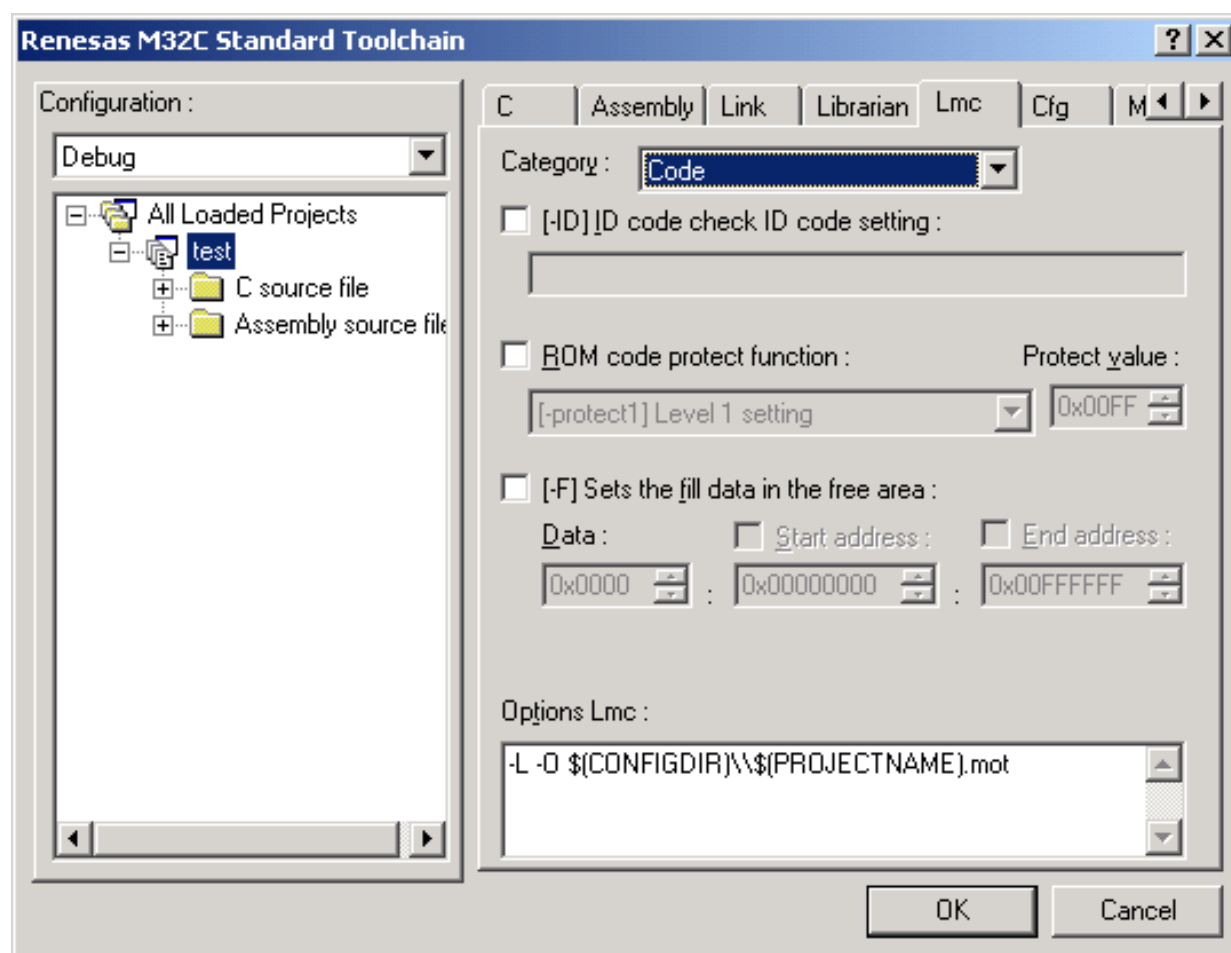


図 4.28 Category:[Code]のダイアログボックス

(3) Category:[Other]

表 4.24 Category:[Other]の項目名とロードモジュールコンバータオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-.] Disables message output to screen	.
[-V] Indicates version of load module converter	V
User defined options :	

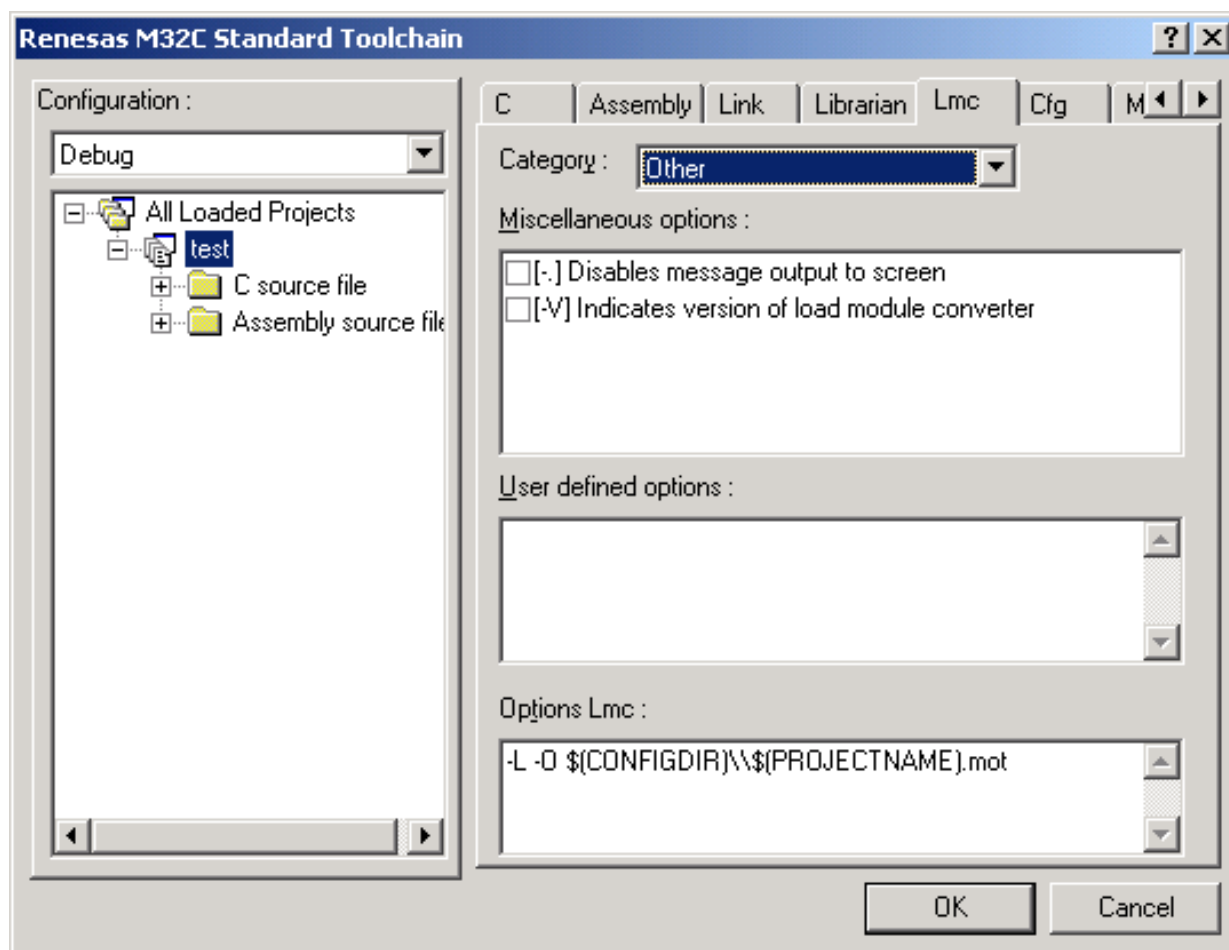


図 4.29 Category:[Other]のダイアログボックス

4.1.6 コンフィグレーションのオプション

Renesas M32C Standard Toolchain ダイアログボックスから Cfg タブを選択します。

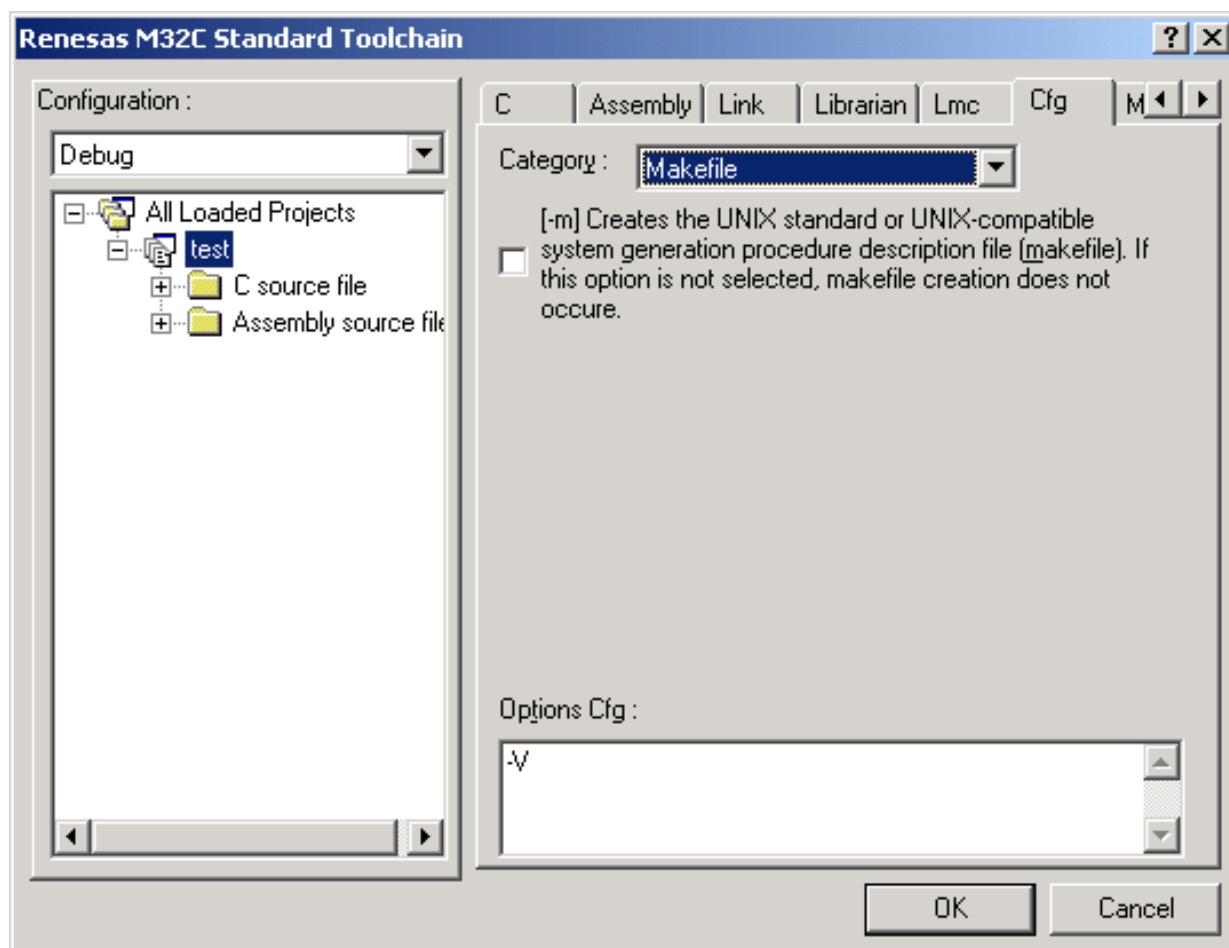


図 4.30 Cfg タブのダイアログボックス

(1) Category:[Makefile]

表 4.25 Category:[Makefile]の項目名とコンフィグレータオプションの対応表

ダイアログボックス	オプション
[-m] Creates the UNIX standard or UNIX-compatible system generation procedure description file (makefile). If this option is not selected, makefile creation does not occur.	m

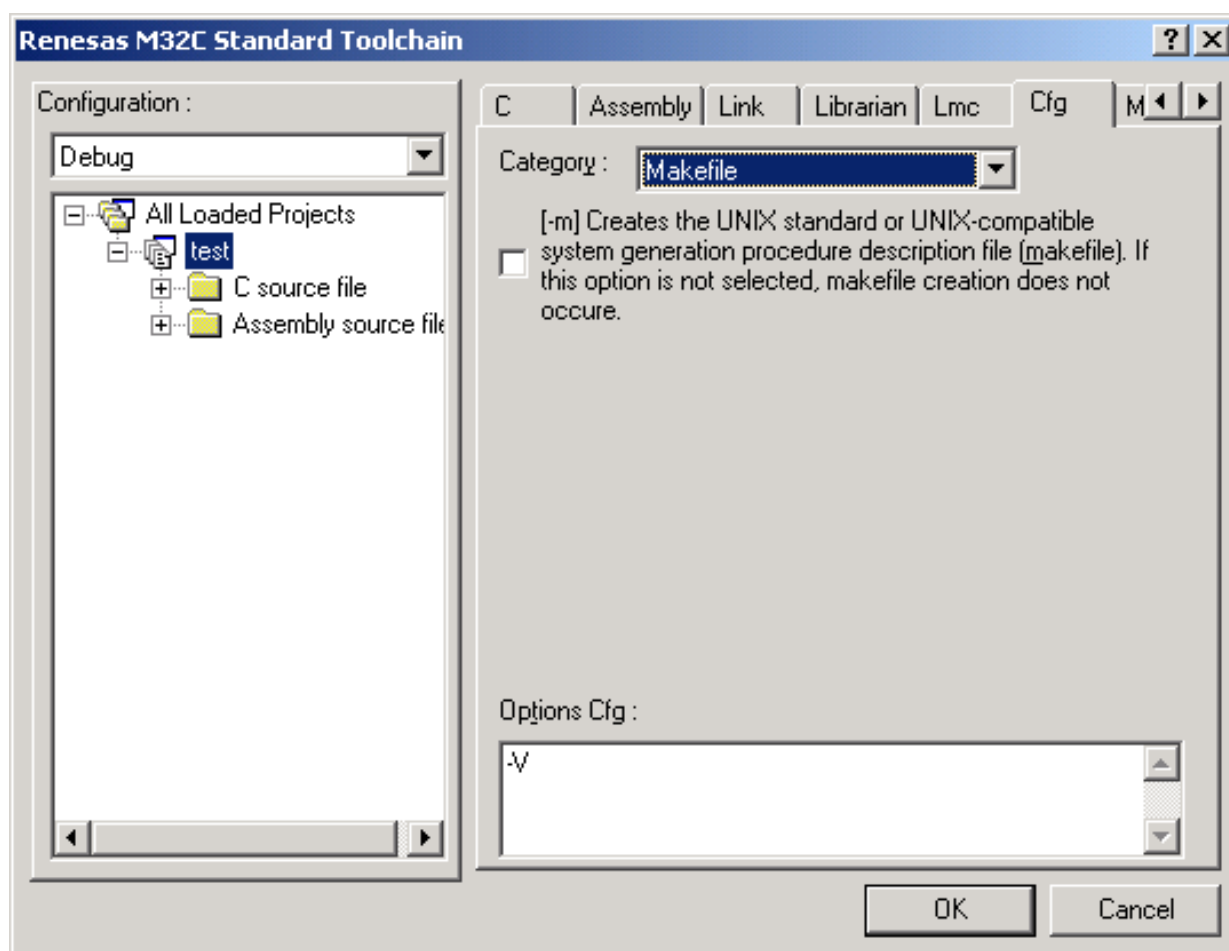


図 4.31 Category:[Makefile]のダイアログボックス

(2) Category:[Other]

表 4.26 Category:[Other]の項目名とコンフィグレータオプションの対応表

ダイアログボックス	オプション
Miscellaneous options :	
[-V] Displays the information on the files generated by the command	V
[-v] Displays the command option descriptions and detailed information on the version	v
User defined options :	

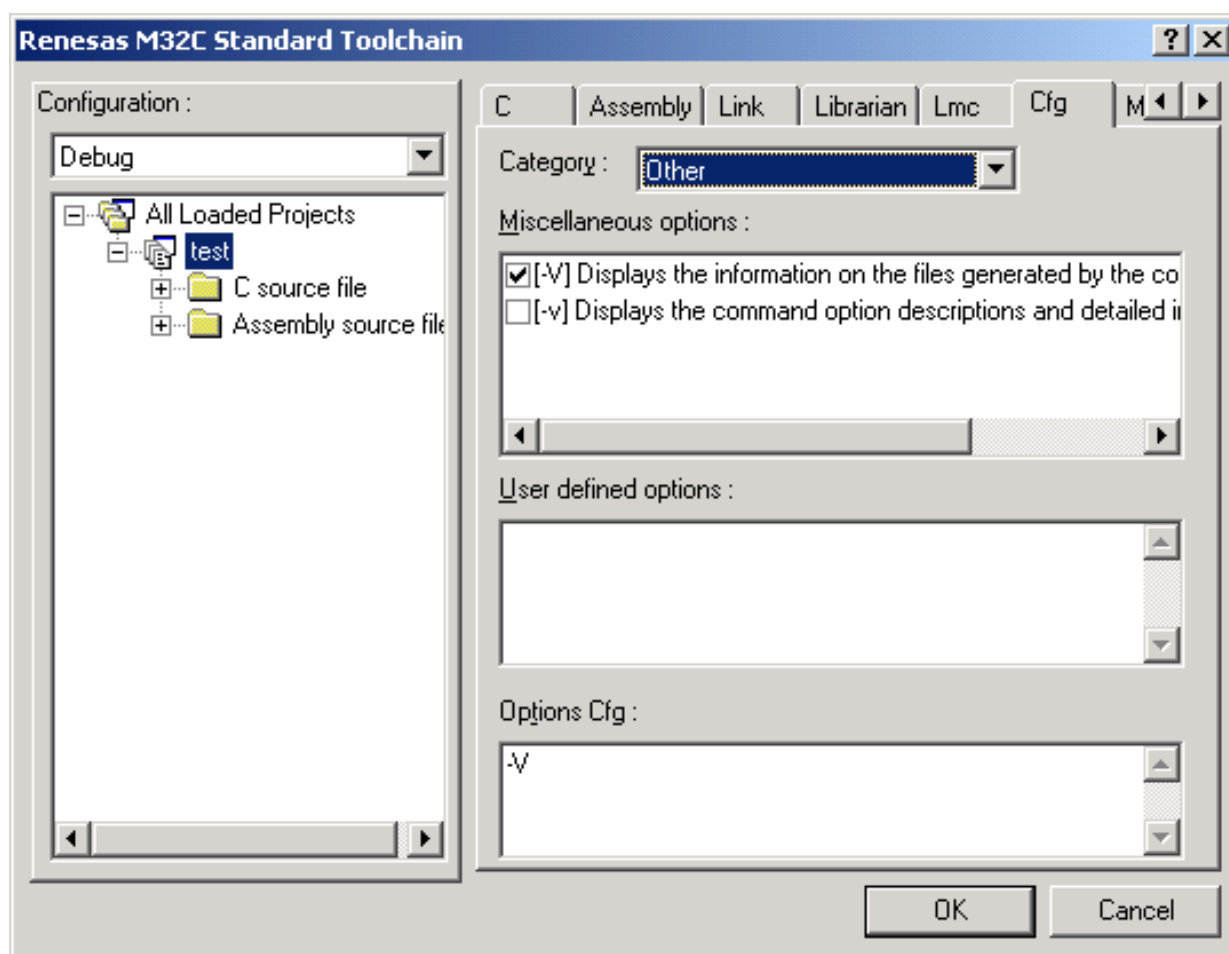


図 4.32 Category:[Other]のダイアログボックス

4.1.7 CPU オプション

Renesas M32C Standard Toolchain ダイアログボックスから[CPU]タブを選択します。

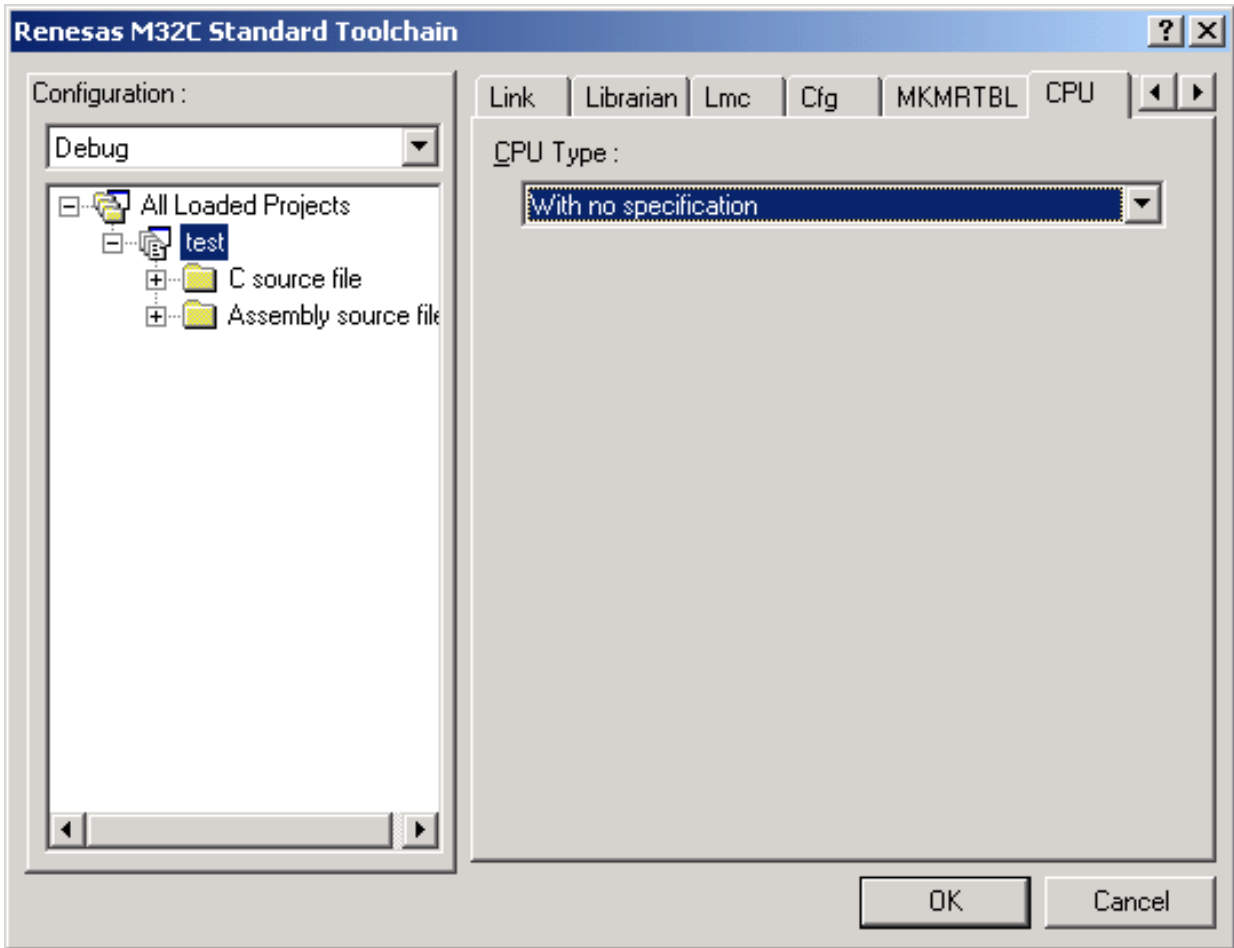


図 4.33 [CPU]タブのダイアログボックス

表 4.27 [CPU]タブの項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
CPU Type :	
With no specification	-
Generates code for M32C/80 series	M82

4.2 ビルド編

4.2.1 メイクファイルの出力

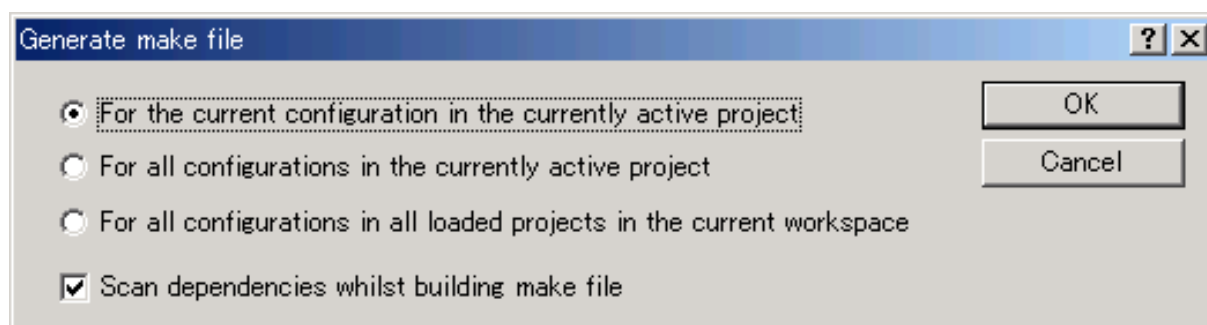
■説明

High-performance Embedded Workshop では、現在のオプション指定状況を元にメイクファイルを生成することができます。

メイクファイルを使用すると、完全に High-performance Embedded Workshop をインストールしていても、現在のプロジェクトをビルドすることができます。High-performance Embedded Workshop をインストールしていない相手にプロジェクトを送ったり、メイクファイルを含むビルド全体をバージョン管理したりする場合に便利です。

■メイクファイル出力方法

1. メイクファイルを生成するプロジェクトが現在のプロジェクトであることを確認してください。
2. プロジェクトをビルドするビルドコンフィグレーションが現在のコンフィグレーションであることを確認してください。
3. [Build>Generate Makefile]を選んでください。
4. 以下のダイアログが表示されるので、メイクファイルをどのように生成するかを選択します。



■メイクファイル生成ディレクトリ

High-performance Embedded Workshop は現在のワークスペースディレクトリ内に “ make ” サブディレクトリを作り、その中にメイクファイルを作成します。メイクファイルの名前は、現在のプロジェクトやコンフィグレーションに拡張子.mak を付けたものです。（例：debug.mak）。High-performance Embedded Workshop により生成されたメイクファイルは、High-performance Embedded Workshop をインストールしたディレクトリ（例：c:\hew3）にある実行ファイル HMAKE.EXE で実行できます。ただし、ユーザが変更したメイクファイルは実行できません。

■メイクファイル実行方法

1. コマンドウインドウを開き、メイクファイルが生成された “ make ” ディレクトリに移動してください。
2. HMAKE を実行してください。コマンドラインは HMAKE.EXE <メイクファイル名>です。

4.2.2 メイクファイルの入力

■ 説明

High-performance Embedded Workshop では、High-performance Embedded Workshop で作成したメイクファイルや UNIX 環境で使ったメイクファイルを入力することができます。

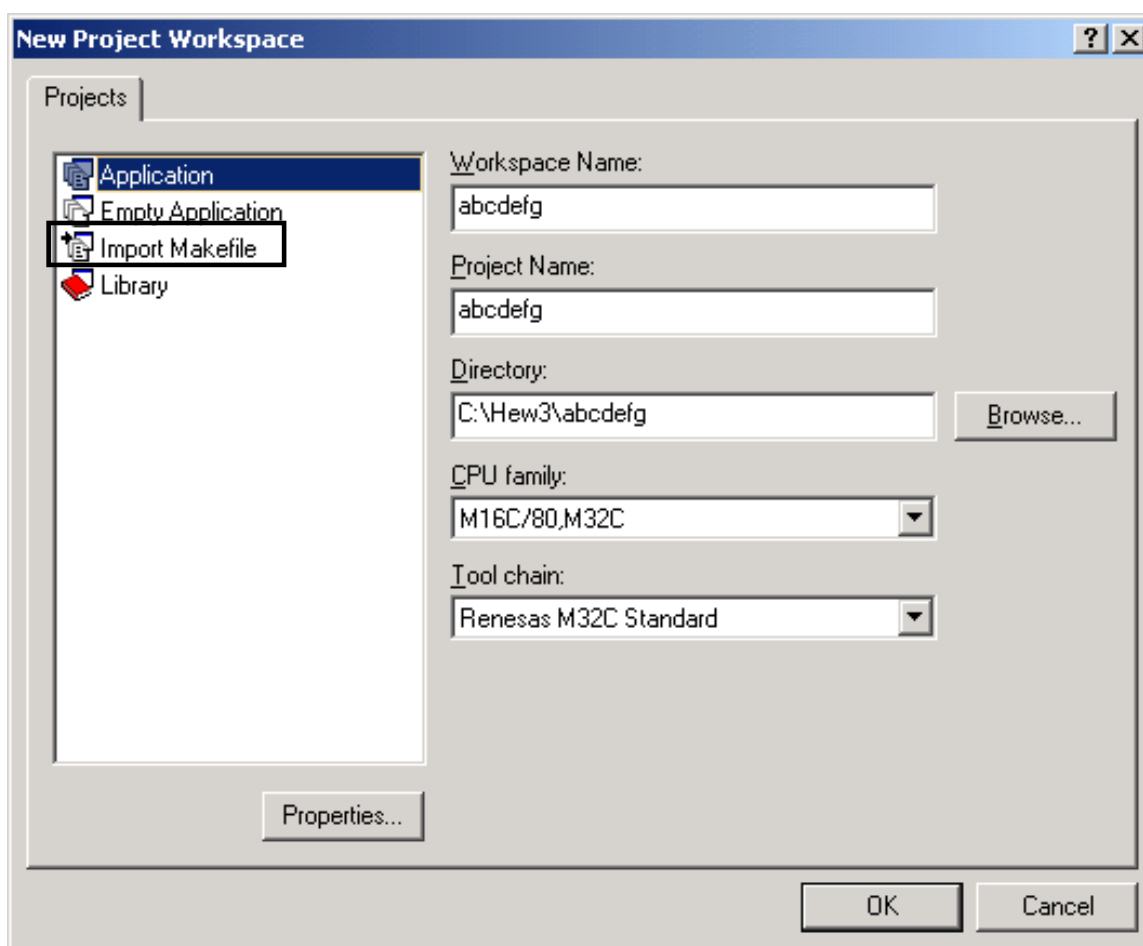
メイクファイルからプロジェクトのファイル構成を自動で取得することができます。

(オプション指定等は取得できません。)

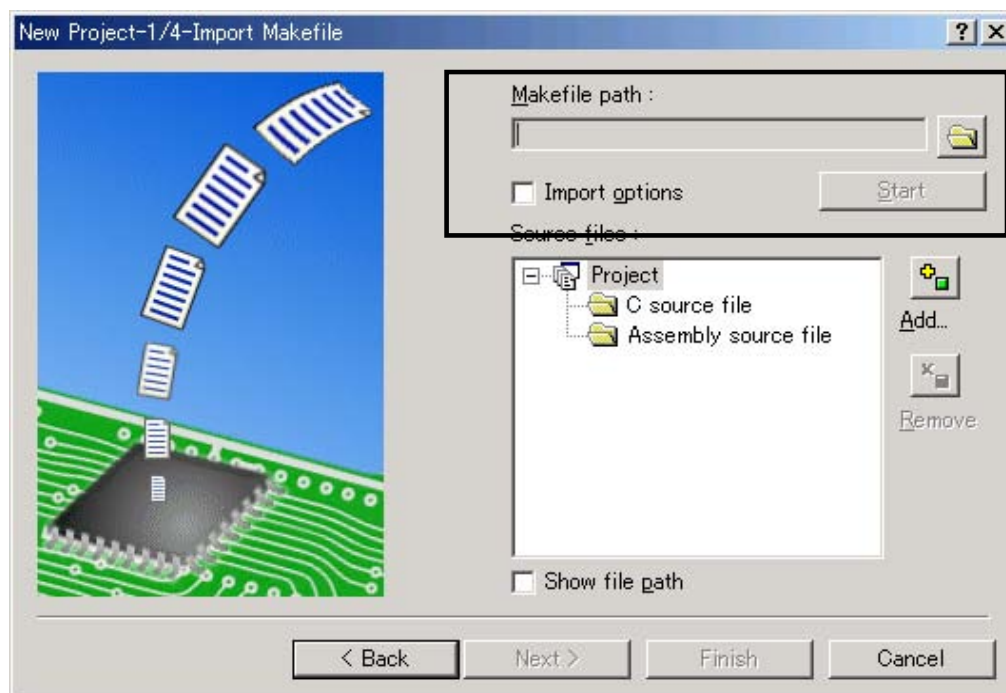
これにより、コマンドラインから High-performance Embedded Workshop への移行が容易になります。

■ メイクファイルの入力方法

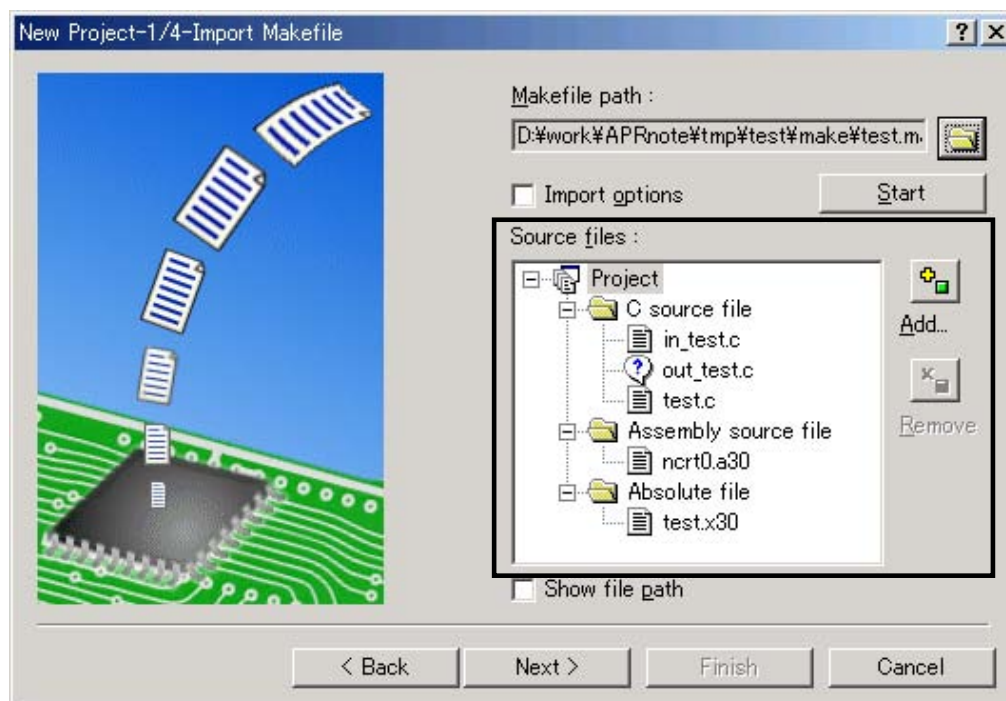
1. 新規ワークスペース作成時、New Project Workspace ダイアログのプロジェクトタイプの中から Import Makefile を選択します。



2. New Project-Import Makefile ダイアログの Makefile path に入力するメイクファイルのパスを設定し Start ボタンを押下します。



3. Source files にメイクファイルのソースファイルの構成が表示されます。このとき、 が表示されたファイルはメイクファイルの内容を解析した結果、ファイルの実体が無い状態を示しています。このファイルはプロジェクトに追加しません。（無視されます）



4. その後はウィザードに従い、CPU やオプション設定等を設定後、ワークスペースをオープンし、プログラム開発作業を開始できます。

4.2.3 カスタムプロジェクトタイプの作成

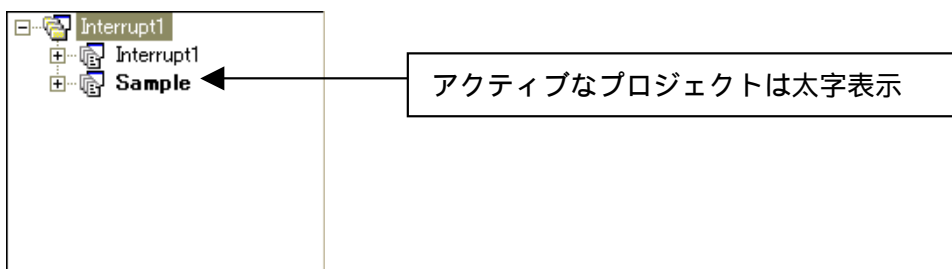
■ 説明

カスタムプロジェクト作成機能を使うと、あるユーザが作成したプロジェクトを雛型として、他のユーザが別なマシンでプログラム開発することができます。

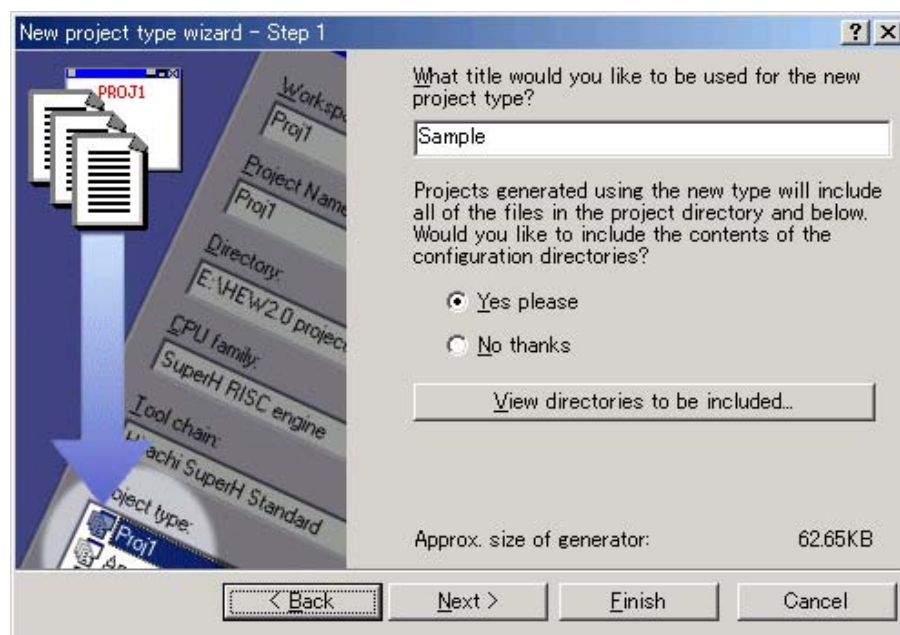
雛型として保存できる情報は、プロジェクトファイル構成やビルドオプションやデバッガ設定状況等のプロジェクト内容を全て保存することが可能です。

■ プロジェクトタイプ保存方法

- 1 ワークスペースをオープンしている状態で、アクティブになっているプロジェクトが保存されるプロジェクトになるので、対象プロジェクトをアクティブにします。プロジェクトをアクティブにするには[Project->Set Current Project]でプロジェクトを選択します。

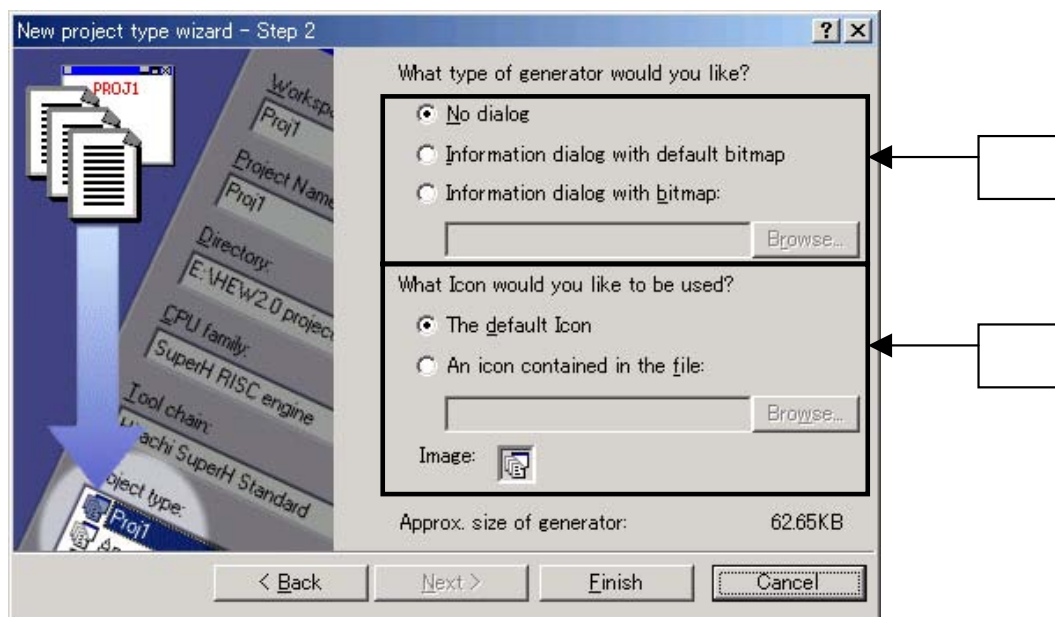


- 2 [Project->Create Project Type]により以下のプロジェクトタイプウィザードを表示し、雛型として使用するプロジェクトタイプの名前を決めて、ビルド後の実行ファイル等を含むコンフィグレーションディレクトリについても雛型とするか選択します。
ここで Finish ボタンを押下し、プロジェクトタイプウィザードを終了しても構いません。

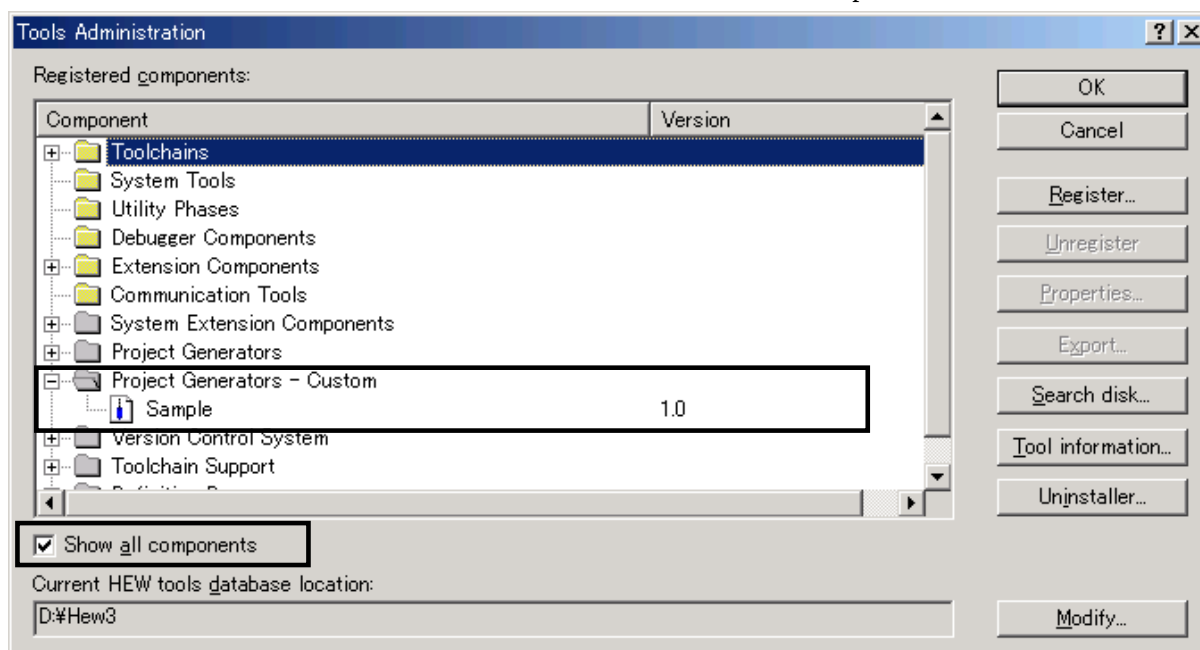


4. High-performance Embedded Workshop

- 3 [New project type wizard – Step 1]で[Next >]ボタンを押下すると以下のウィザードが表示されます。以下の ではプロジェクトタイプの雛型をオープンするときに、プロジェクトの情報やビットマップを表示するか選択します。
- また、 ではプロジェクトタイプのアイコンをユーザ指定のアイコンに変更することもできます。その後、Finish ボタンを押下します。
- これらの指定は必須ではありません。

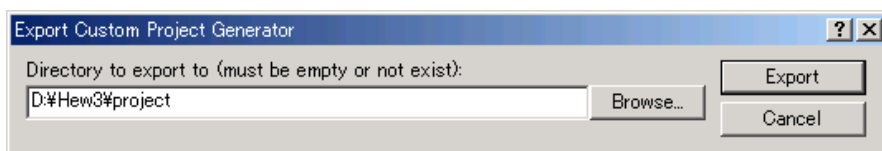


- 4 これでカスタムプロジェクトジェネレータというプロジェクトタイプの雛型が作成できました。この雛型を他のマシンで使用するときは、[Tools->Administration]を選択し以下のダイアログを表示します。
- 以下の[Show all components]チェックボックスをチェックすると、[Project Generators - Custom]という項目が表示されます。
- 表示されたら、作成したプロジェクトタイプをクリックし Export ボタンを押下します。



そうすると、以下のダイアログが表示されるのでカスタムプロジェクトジェネレータを出力するディレクトリを指定します。ディレクトリの中身は空である必要があります。

これで、プロジェクトタイプの保存は終了です。

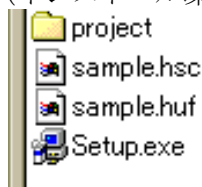


■ カスタムプロジェクトジェネレータのインストール

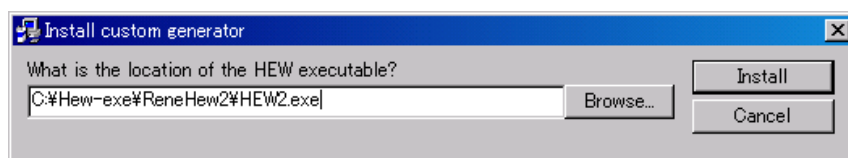
上記の[プロジェクトタイプ保存方法]で作成したカスタムプロジェクトジェネレータを他のマシンにインストールする方法を説明します。

1. [プロジェクトタイプ保存方法]の 5.で作成したディレクトリに、以下のようにインストール環境が作成されています。

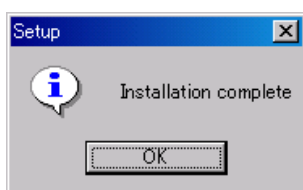
(インストール環境ディレクトリ)



2. 上記のインストール環境を他のマシンにコピーしインストールします。
Setup.exe を実行すると、以下のダイアログが表示されるので High-performance Embedded Workshop のインストール場所を指定し、Install ボタンを押下します。
(ディレクトリ例：C:\Hew-exe\ReneHew2\HEW2.exe)



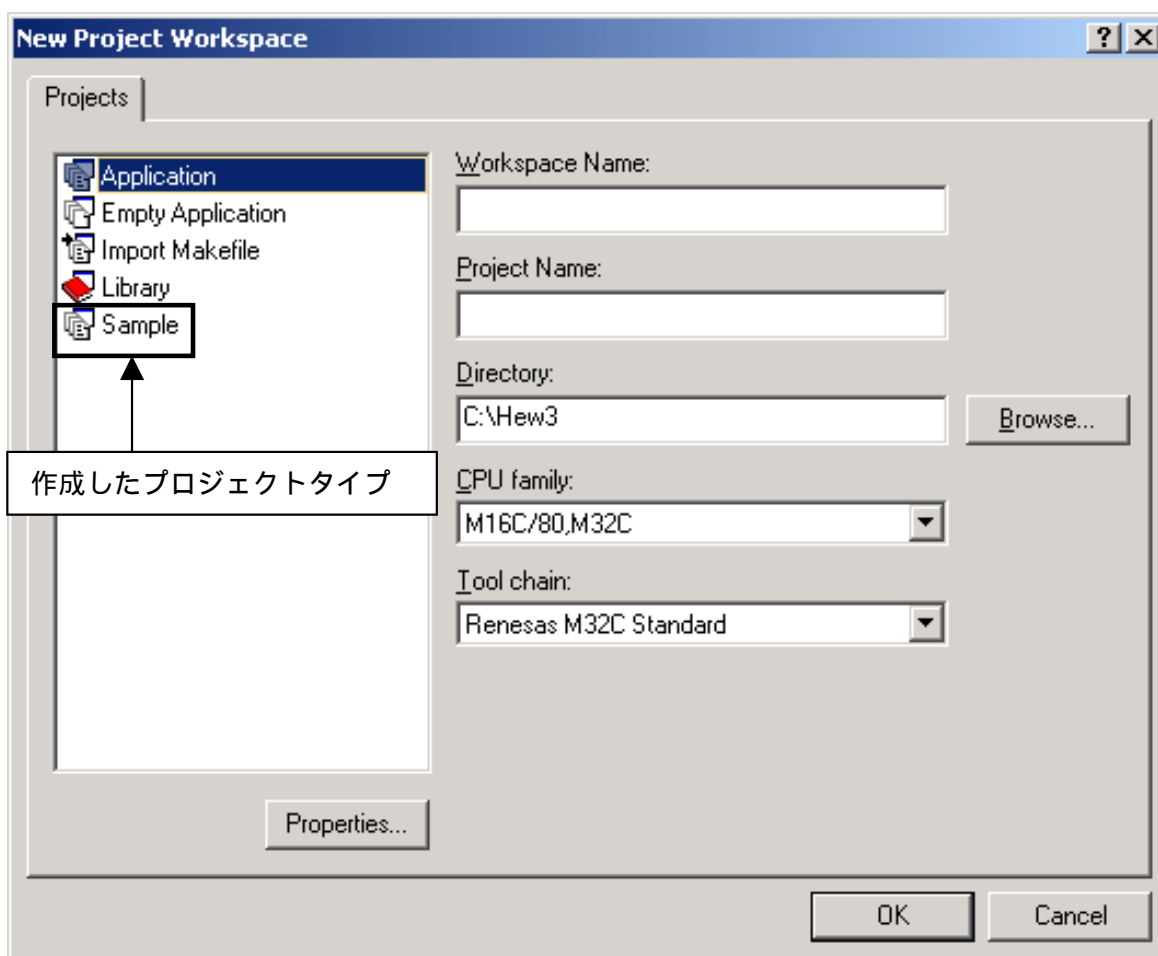
3. これで、環境の構築は終了です。



■ カスタムプロジェクトジェネレータの使用例

インストールしたカスタムプロジェクトジェネレータの使用例を以下に示します。

1. High-performance Embedded Workshop を起動し、[Welcome!]ダイアログで[Create a new project workspace]の作成を選択すると[Projects]にインストールしたプロジェクトタイプが追加されているので、クリック後 OK ボタンを押下します。
これで、新しいプロジェクトでも保存したプロジェクトの雛型を利用し、プログラム開発をすることができます。



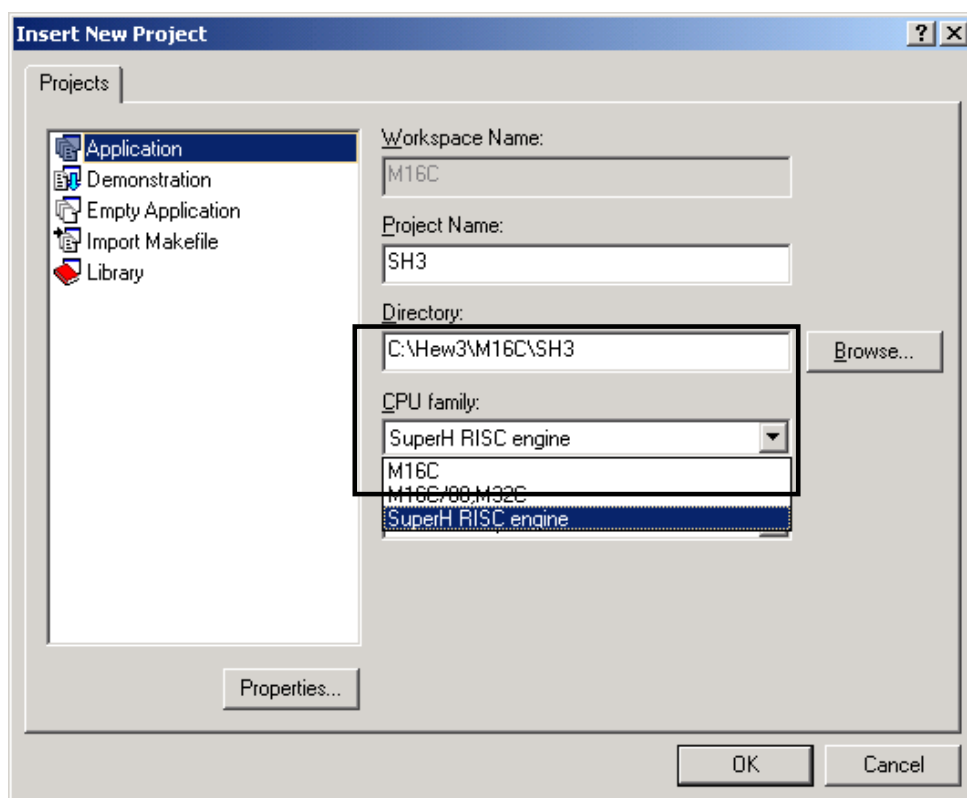
4.2.4 マルチ CPU 機能

■説明

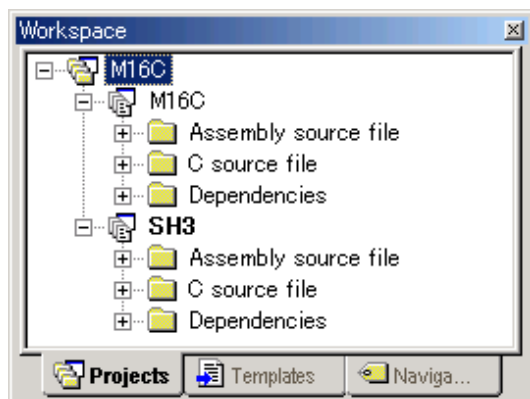
ワークスペースに新規プロジェクトを挿入する時、別なCPU種別のCPUを挿入することができます。これによりM16CとSH等のプロジェクトが一つのワークスペースで管理することができます。

■別 CPU ファミリ挿入の例

1. M16C(SH)のプロジェクトを開いている時、[Project->Insert Project]をクリックし、Insert Project ダイアログで New project を選択後に OK ボタンを押下します。
2. 以下の新規プロジェクトの挿入ダイアログが出現します。ここでプロジェクト名とCPU 種別で M16C(SH)を選択後に OK を押下すると、別な CPU 種別をワークスペースに混載することができます。



3. このような手順で以下のように一つのワークスペースに M16C と SH のプロジェクトを混載させることができます。



4.2.5 ネットワーク機能

■ 説明

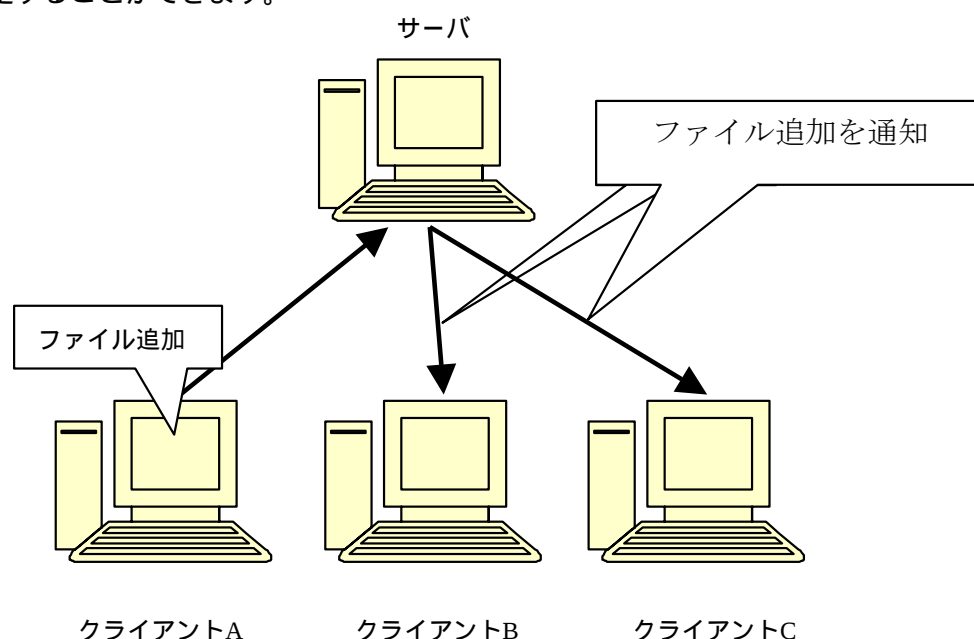
High-performance Embedded Workshop は、ネットワークを介してワークスペースやプロジェクトを共有することができます。

よって、ユーザは共有したプロジェクトを同時に操作してお互いの変更を知ることができます。

このシステムは一つのコンピュータをサーバとして使います。

これにより例えば、クライアントがプロジェクトに新規ファイルを追加すると、サーバマシンに通知され、サーバが全てのクライアントに追加を通知することができます。

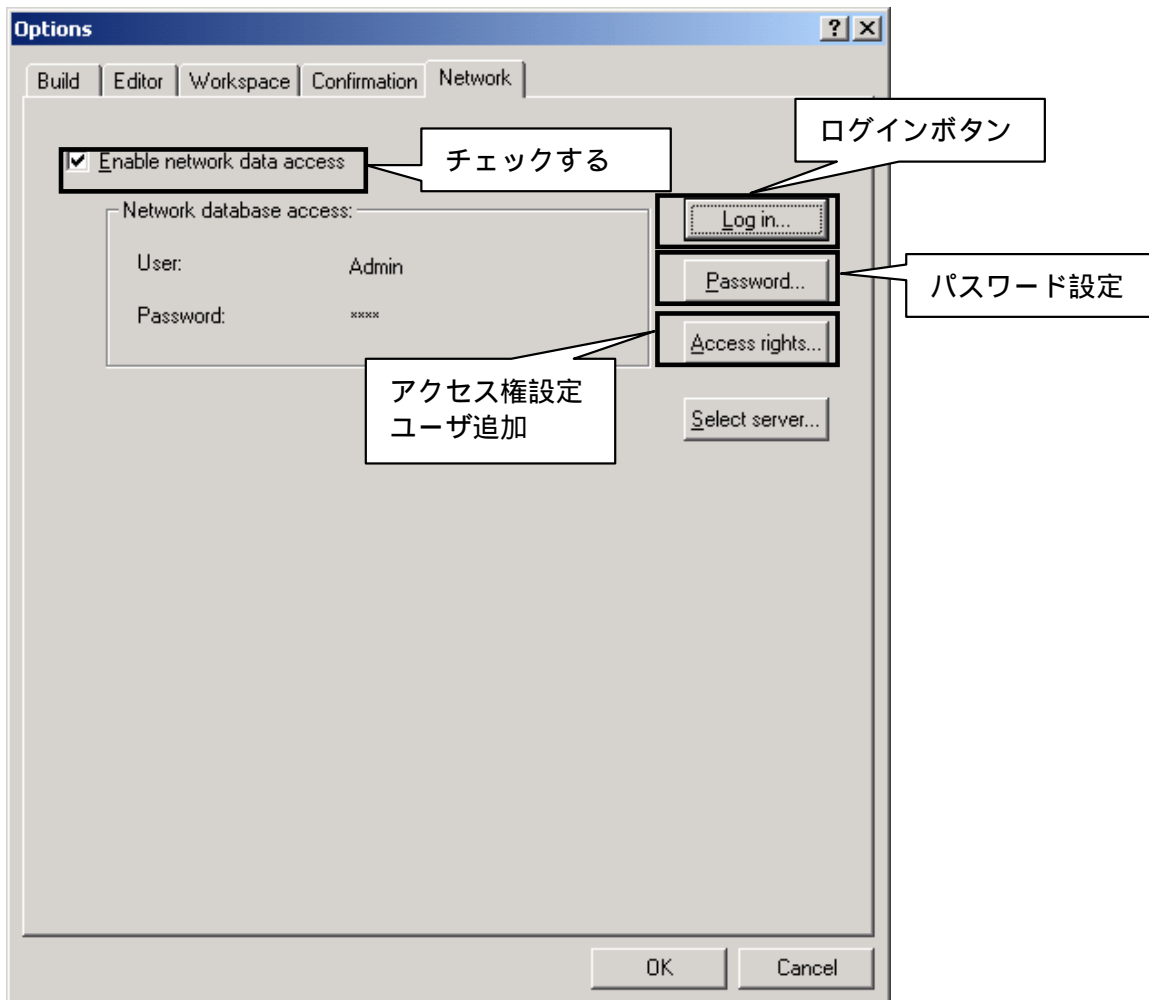
さらに、ユーザにアクセス権を持たせることができ、プロジェクトやファイルに対する書き込みの権限を指定することができます。



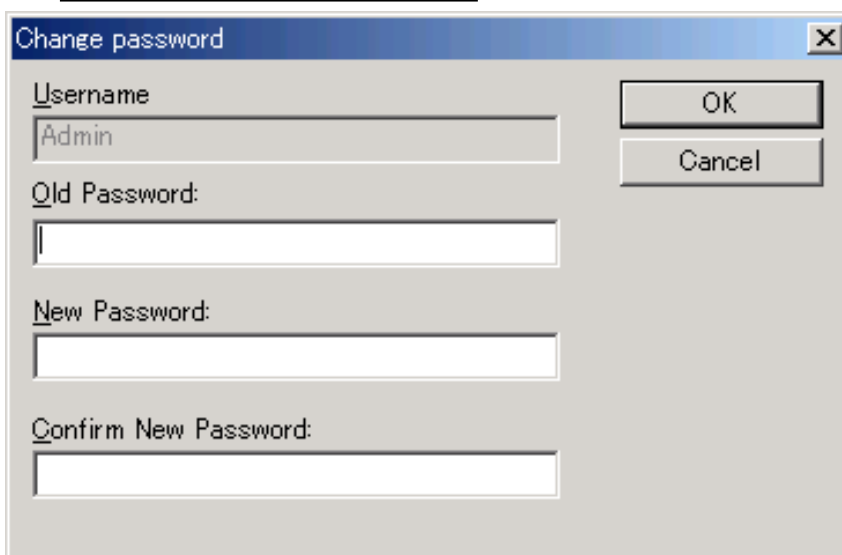
■ ネットワークアクセスの設定

1. [Tools->Options]を選んだ後、Network タブを選択し、Enable network data access チェックボックスをチェックします。
2. アドミニストレータが追加されます。初期状態ではパスワードがないので、指定する必要があります。アドミニストレータは最高レベルのアクセス権を持ちます。
3. Password ボタンをクリックし、アドミニストレータのパスワードを設定します。
4. OK ボタンを押下します。これでネットワークアクセスが可能になります。

Optionsダイアログ/Networkタブ



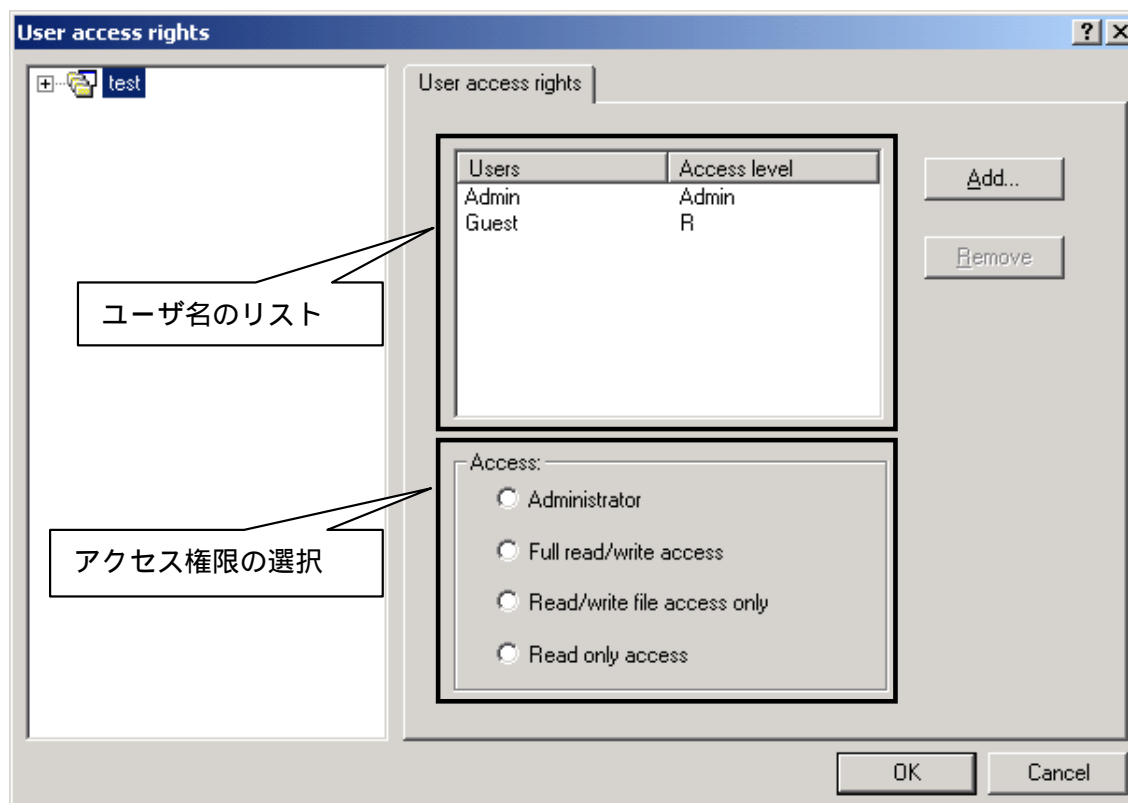
Change passwordダイアログ



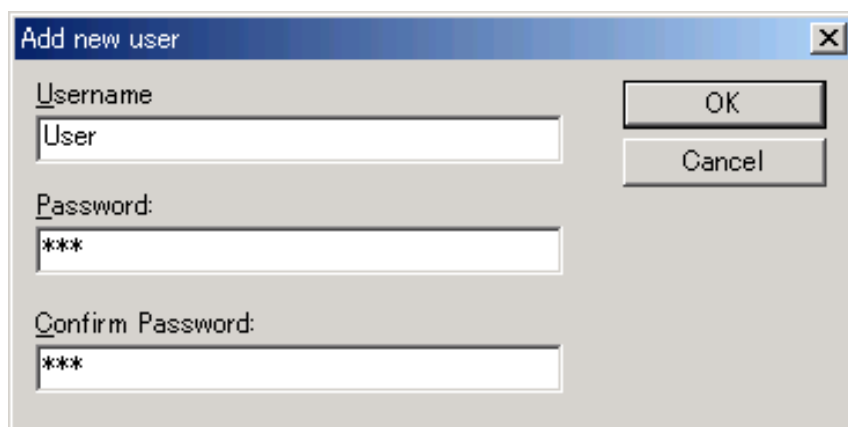
■新規ユーザの追加

デフォルトではアドミニストレータとゲストが追加されていますが、新たなユーザを登録することもできます。

1. 前ページの Log in ボタンを押下し、アドミニストレータのアクセス権を持つユーザでログインします。
2. Access rights ボタンをクリックし、以下の User access rights ダイアログを表示します。



3. Add ボタンをクリックし、Add new user ダイアログを表示します。
4. 新たな Username と Password を登録します。（パスワード指定は必須）



5. これによりユーザのリストに新たなユーザ名が追加されます。ここで、ユーザ名を選択し、このユーザの権限を決定します。
OK ボタンを押下すると設定が反映されます。

■ サーバマシンの決定

どのマシンをサーバにするか決定します。自分のマシンをサーバにする場合は、何もする設定する必要はありません。

他のマシンをサーバにする場合は、Network タブの Select server ボタンを押下し、以下のダイアログで Remote を選んだ後に、Computer name を指定します。

OK ボタンを押下すると、設定が反映されます。



■ 備考

本機能を使用すると High-performance Embedded Workshop の性能が低下します。

5. 効率の良いプログラミング技法

NC308WA コンパイラは最適化を行っていますが、プログラミングの工夫により一層の性能向上が可能です。本章では、効果的なプログラム作成のために、ユーザに試みて頂きたい手法について記述します。プログラムの評価基準には、実行速度が速いこととサイズが小さいことの2種類があります。効果的なプログラムを作成するための原則を以下に示します。

実行速度向上の原則

実行頻度の高い文、複雑な文で実行速度は決まるので、これらの処理を把握して、重点的に改良してください。

サイズ縮小の原則

プログラムサイズ縮小のためには、類似処理の共通化、複雑な関数の見直しを行ってください。

コンパイラの最適化のため、実行速度が机上で検討した場合とは異なる結果になることがあります。様々な手法を駆使し、実際にコンパイラで実行して確認しながら性能追及を進めてください。効率の良いプログラミング技法の一覧を表5.1に示します。

表 5.1 効率の良いプログラミング技法一覧

項番	項目	RAM 効率	ROM 効率	実行速度
5.1	引数のレジスタ渡し	○	○	○
5.2	レジスタ変数の活用	○	○	○
5.3	M16C 特有の命令の活用	—	○	○
5.4	キャリーフラグによるビット演算	—	○	○
5.5	ループ内固定式のループ外への移動	—	—	○
5.6	SBDATA 宣言 & SPECIAL ページ関数 宣言ユーティリティ	SBDATA 宣言	○	—
		SPECIAL ページ 関数宣言	○	×
5.7	else if の switch 化	—	—	○
5.8	ループカウンタの条件判定	—	○	○
5.9	restrict	—	○	○
5.10	_Bool の活用	—	○	—
5.11	auto 変数を明示的に初期化	○	○	○
5.12	配列の初期化	—	—	○
5.13	インクリメント/デクリメント	○	○	○
5.14	switch 文	—	—	○
5.15	浮動小数点即値	—	○	○
5.16	外部変数のゼロクリア	—	○	○
5.17	スタートアップの整理	—	○	○
5.18	ループ内はテンポラリを使用する	×	—	○
5.19	32 ビット用数学関数	—	○	○
5.20	なるべく unsigned を使う	—	○	○
5.21	配列の添え字の型	○	○	○
5.22	プロトタイプ宣言の活用	○	○	○
5.23	char 型の範囲でしか値を返さない関数の戻り値を char 型にする	—	○	—
5.24	bss 領域のクリア処理のコメントアウト	—	○	○
5.25	生成コードの削減	○	○	—

5.1 引数のレジスタ渡し

関数に引数を渡す方法としてスタックに積んで渡すスタック渡しとレジスタに代入して渡すレジスタ渡しがあります。スタック渡しはスタックへの積み込み、取り出しのコストが発生するのに対しレジスタ渡しはそのようなコストがなく高速に動作します。引数がレジスタ渡しになる条件は次の3つです。

関数のプロトタイプ宣言が行われている。

プロトタイプ宣言に可変引数”...”を使用していない。

関数の引数の型が以下の型と一致している。

図 5.1 はプロトタイプ宣言の有無でレジスタ渡しになるかスタック渡しになるかが変わる様子をあらわしています。

表 5.2 レジスタ渡しになる型

コンパイラ	第一引数	第二引数
NC30WA	_Bool 型 char 型 int 型 near ポインタ型	int 型 near ポインタ型
NC308WA	_Bool 型 char 型 int 型 near ポインタ型	なし

またレジスタ渡し時のレジスタへの割り付けは以下のようになっています。

表 5.3 レジスタ渡しの割り付け

引数の型	コンパイラ	第 1 引数	第 2 引数	第 3 引数以降
_Bool 型 char 型	NC30WA	R1L	スタック	スタック
	NC308WA	R0L		
int 型 near ポインタ型	NC30WA	R1	R2	スタック
	NC308WA	R0	スタック	
その他の型	NC30WA	スタック	スタック	スタック
	NC308WA			

改善前	改善後
<pre>int main() { f(3); } int f(a) int a; { ... }</pre>	<pre>int f(int a); int main() { f(3); } int f(int a) { ... }</pre>
<pre>push.w #0003H jsr _f</pre>	<pre>mov.w #0003H, R0 jsr \$f</pre>

図5.1 引数のレジスタ渡し例

5.2 レジスタ変数の活用

変数宣言に `register` 修飾子をつけることで変数をレジスタに割り付けることができます。使用頻度の高い変数をレジスタに割り付けることによってプログラムを高速化することができます。ただしむやみに `register` 修飾子をつけるとレジスタが足りなくなり逆効果になることもあります。また NC30WA では関数呼び出しをまたいで生存する変数をレジスタに割り付けると関数呼び出し前後でレジスタの退避復帰命令が生成され逆効果になることもあります。`register` 修飾子を有効にするにはコンパイル時に `-fER` オプションが必要です。図 5.4 に改善例を示します。

```
int f()
{
    register int i;
    for(i=0;i<100;i++)
    {
        ...
    }
}
```

変数iを強制的にレジスタに配置

図5.2 レジスタ変数の宣言

```
int a;
int f()
{
    register int i;
    i=a;
    g();
    i=a+1;
}
```

NC30WAではレジスタの退避復帰命令が生成されるため逆効果

図5.3 レジスタの退避復帰

5. 効率の良いプログラミング技法

改善前	改善後
<pre>int i; sum=0; for(i=0;i<100;i++) { sum+=a[i]; }</pre>	<pre>register int i; sum=0; for(i=0;i<100;i++) { sum+=a[i]; }</pre>
<pre>;;# # C_SRC : sum=0; mov.w #0000H,-4[FB] ; sum ;;# # C_SRC : for(i=0;i<100;i++) mov.w #0000H,-4[FB] ; i L1: ;;# # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,-4[FB] ; i jge L5 ;;# # C_SRC : sum+=a[i]; mov.w -4[FB],A0 ; i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5:</pre>	<pre>;;# # C_SRC : sum=0; mov.w #0000H,-2[FB] ; sum ;;# # C_SRC : for(i=0;i<100;i++) mov.w #0000H,R0 L1: ;;# # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,R0; i jge L5 ;;# # C_SRC : sum+=a[i]; mov.w R0,A0 ; i i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,R0; i jmp L1 L5:</pre>

図5.4 レジスタ変数の活用

5.3 M16C 特有の命令の活用

次のような if 文によって変数に値を代入するようなコードは if 節と else 節で同じ変数に即値を代入するようなコードにすると S T Z X 命令に展開され分岐の削減、R O M 効率の向上が見込めます。

改善前	改善後
<pre>void main(void) { int i=2; int port; if(port == 1){ i = 3; } }</pre>	<pre>void main(void) { int i; int port; if(port == 1){ i = 3; }else{ i = 2; } }</pre>
<pre>mov.w #0002H,R0; i cmp.w #0001H,-2[FB] ; port jne L3 mov.w#0003H,R0; i L3:</pre>	<pre>cmp.w#0001H,-2[FB] ; port stzx.w #0003H,#0002H,R0 ; i</pre>

図5.5 M16C特有の命令の活用

5.4 キャリーフラグによるビット演算分岐

次のようなコードの場合、`&& (||)` よりも `& (|)` のほうが良いコードが出ます。

改善前	改善後
<pre>struct A { int a:1; int b:1; int c:1; } a; main() { if(a.a && a.b && a.c) func(); }</pre>	<pre>struct A { int a:1; int b:1; int c:1; } a; main() { if(a.a & a.b & a.c) func(); }</pre>
<pre> btst 00H,-2[FB]; a jz L1 btst 01H,-2[FB]; a jz L45 btst 02H,-2[FB]; a jz L47 jsr _func L47: L45: L1:</pre>	<pre> btst 00H,-2[FB]; a band 01H,-2[FB]; a band 02H,-2[FB]; a jnc L29 jsr _func L29:</pre>

図5.6 キャリーフラグによるビット演算分岐

5.5 ループ内固定式のループ外への移動

次のようなコードの場合、ループ内固定式をループの外へ移動することで計算回数を減らし高速化することができます。この最適化はコンパイラに最適化オプションを与えることで自動的に行ってくれます。

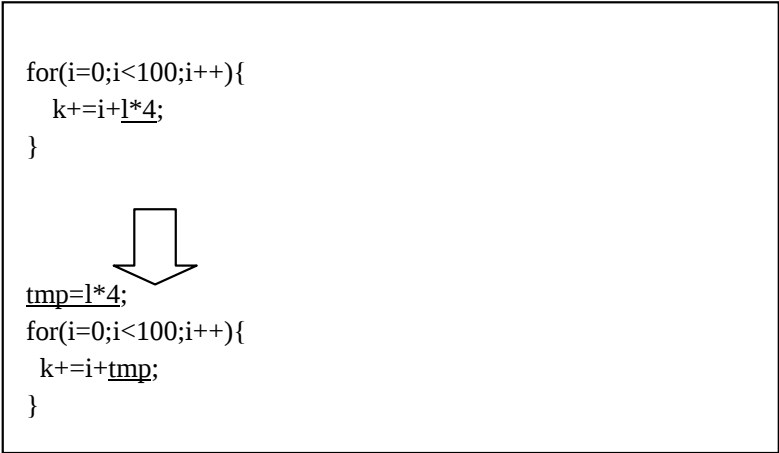


図5.7 ループ内固定式のループ外への移動

C ソース	最適化なし	最適化あり
for(i=0;i<100;i++) a[i]=l*4;	### C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: _line 18 ### C_SRC : for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ### C_SRC : a[i]=l*4; mov.w _l:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 jmp L1 L5:	### C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 mov.w _l:16,R0 shl.w #2,R0 L1: ### C_SRC : a[i]=l*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L1

図5.8 最適化によるループ内固定式のループ外への移動

5.6 SBDATA 宣言 & SPECIAL ページ関数宣言ユーティリティ

SBDATA 宣言 & SPECIAL ページ関数宣言ユーティリティ `utl308` はアブソリュートモジュールファイル (拡張子 `.x30`) を処理して

1. SBDATA 宣言

使用頻度の高い変数から SB 領域に割り当てるための宣言

(`#pragma SBDATA`)

2. SPECIAL ページ関数宣言

使用頻度の高い関数からスペシャルページ領域に割り当てるための宣言

(`#pragma SPECIAL`)

を出力します。

`utl308` を使用するには、コンパイル時に、コンパイルドライバに起動オプション "`-finfo`" を指定してアブソリュートモジュールファイル (拡張子 `.x30`) を生成してください。

NC308 の処理フローを図 5.9 に示します。このツールを使用することで SBDATA 機能と SPECIAL ページ機能を最適に使用できます。詳しくは NC308 ユーザーズマニュアル付録 G をご参照ください。

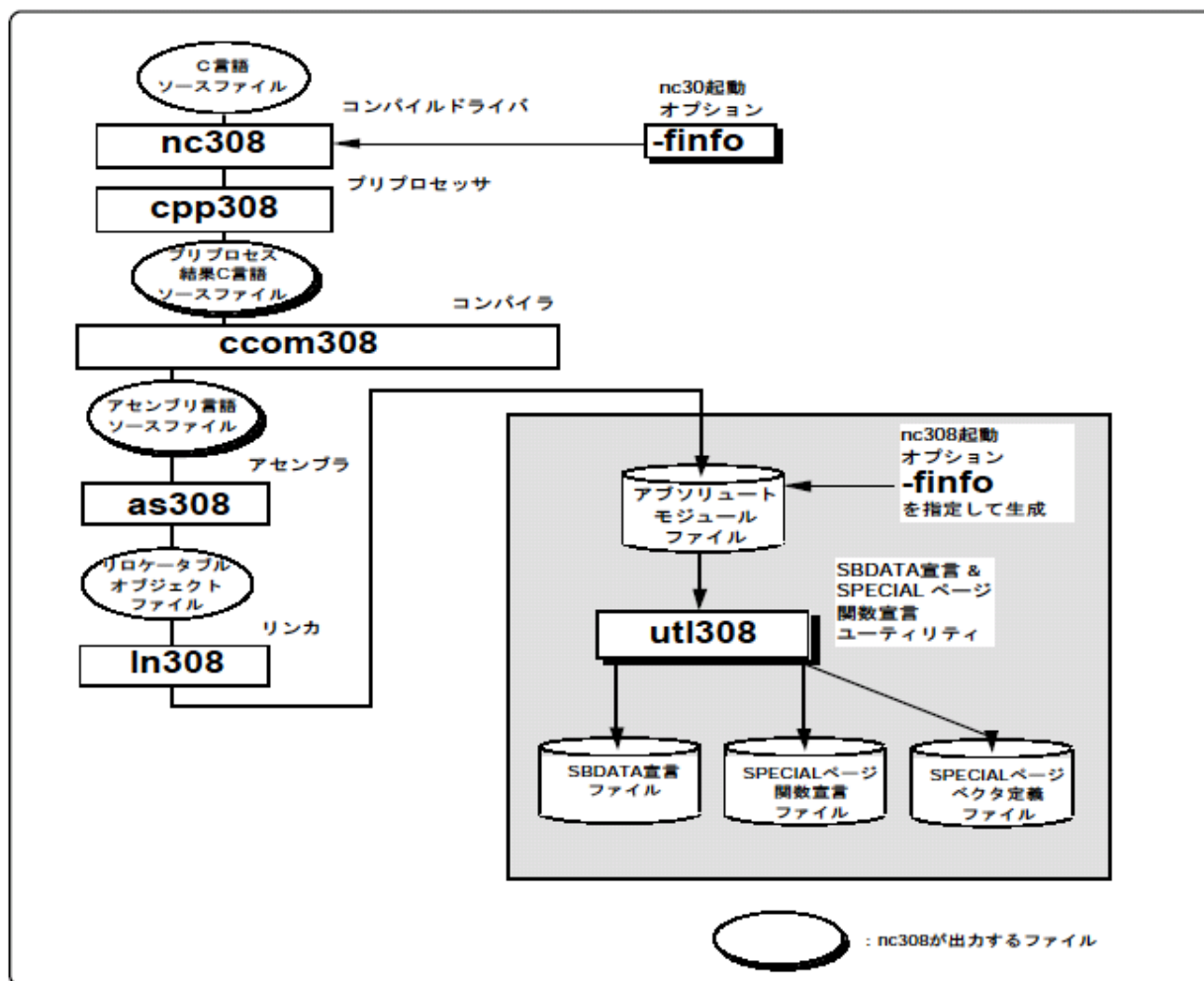


図5.9 SBDATA宣言 & SPECIALページ関数宣言ユーティリティ

5.7 else if の switch 化

配列や構造体を複数回比較する場合は else if よりも switch 文の方が高速になります。else if が間接アドレッシングで比較されるのに対し、switch はレジスタに確保しレジスタで比較を行います。

改善前	改善後
<pre> if(a[i]==0){ ... } else if(a[i]==1){ ... } else if(a[i]==2){ ... } else { ... } </pre>	<pre> switch(a[i]) { case 0: ... break; case 1: ... break; case 2: ... break; default: ... beak; } </pre>
<pre> ;## # C_SRC : if(a[i]==0) mov.w R0,R1 ; i i shl.w #1,R0 mov.w R0,A0 mov.w _a:16[A0],R0 jne L1 ... ;## # C_SRC : else if(a[i]==1) L1: mov.w R1,A0 ; i i shl.w #1,A0 cmp.w #0001H,_a:16[A0] jne L11 ... ;## # C_SRC : else if(a[i]==2) L11: mov.w R1,A0 ; i i shl.w #1,A0 cmp.w #0002H,_a:16[A0] ... </pre>	<pre> indexws.w R0 ; i mov.w:g_a:16,R0 cmp.w #0000H,R0 jeq L3 cmp.w #0001H,R0 jeq L5 cmp.w #0002H,R0 jeq L7 jmp L9 ... </pre>

図5.10 else if のswitch化

5.8 ループカウンタの条件判定

ループの条件判定を 0 との比較にした場合 ROM 効率、実行速度ともに向上します。

改善前	改善後
<pre>for(i=0;i<10;i++){ ... }</pre>	<pre>for(i=9;i!=0;i--){ ... }</pre>
<pre>add.w #0001H,_i cmp.w #000aH,_i jlt label</pre>	<pre>adjnz.w #-1,_i,label</pre>

図5.11 ループカウンタの条件判定

5.9 restrict

ポインタが他の変数と同じ領域を指さないことがわかっている場合、修飾子 restrict をつけることができます。修飾子 restrict をつけることで最適化が掛かりやすくなります。但し修飾子 restrict をつけ他の変数と同じ領域を指した場合は誤動作を起こす可能性があります。

改善前	改善後
<pre>int a; int *q; void f() { a=10; *q=20; sub(a); }</pre> aの読み込みが必要	<pre>int a; int * restrict q; void f() { a=10; *q=20; sub(a); }</pre> a=10に置き換えられる
<pre>_f: ;## # C_SRC : a=10; mov.w #000aH,_a:16 ;## # C_SRC : *q=20; mov.w #0014H,[_q:16] ;## # C_SRC : sub(a); mov.w _a:16,R0 jsr \$sub ;## # C_SRC : } rts</pre>	<pre>_f: ;## # C_SRC : a=10; mov.w #000aH,_a:16 ;## # C_SRC : *q=20; mov.w #0014H,[_q:16] ;## # C_SRC : sub(a); mov.w #000aH,R0 jsr \$sub ;## # C_SRC : } rts</pre>

図5.12 restrictの使用例

5.10 _Bool の活用

_Bool は 0 か 1 をとる型です。フラグに _Bool 型を使用することで余計な例外処理を省くことができます。

改善前	改善後
<pre>char flag; if(flag==0){ ... } else if(flag==1) { ... } else { 例外処理 }</pre>	<pre>_Bool flag; if(flag==0){ ... } else { ... }</pre>

図5.13 _Bool の活用例

5.11 auto 変数を明示的に初期化

auto 変数を初期化した場合はレジスタに割り付けられます。初期化しない場合はスタック上に確保されます。この最適化には最適化オプションが必要です。

改善前	改善後
<pre>extern unsigned int *p, max_val, min_val; void func(void) { unsigned int max = 0; unsigned int min; //スタックに積まれる while (1) { if (max == 0) min = max = *p; if (max < *p) max = *p; if (min > *p) min = *p; p++; if (*p == 0) break; } max_val = max; min_val = min; }</pre>	<pre>extern unsigned int *p, max_val, min_val; void func(void) { unsigned int max = 0; unsigned int min=*p;//レジスタに割り当て while (1) { if (max == 0) min = max = *p; if (max < *p) max = *p; if (min > *p) min = *p; p++; if (*p == 0) break; } max_val = max; min_val = min; }</pre>
<pre>.glob _func _func: enter #02H pushm R1</pre>	<pre>.glob _func _func: pushm R1 ;## # C_SRC : unsigned int max = 0;</pre>

<pre> ;## # C_SRC : unsigned int max = 0; mov.w #0000H,R0; max ;## # C_SRC : while (1) L3: ;## # C_SRC : if (max == 0) min = max = *p; cmp.w #0000H,R0; max jne L7 mov.w[_p:16],R0 mov.w R0,R1 mov.w R1,-2[FB] ; min L7: ;## # C_SRC : if (max < *p) max = *p; cmp.w[_p:16],R0 ; max jgeu L17 mov.w[_p:16],R0 L17: ;## # C_SRC : if (min > *p) min = *p; cmp.w[_p:16],-2[FB] ; min mov.w[_p:16],-2[FB] ; min L27: ;## # C_SRC : p++; add.l #00000002H,_p:16 ;## # C_SRC : if (*p == 0) break; mov.w[_p:16],R1 jne L3 ;## # C_SRC : max_val = max; mov.w R0,_max_val:16 ; max ;## # C_SRC : min_val = min; mov.w -2[FB],_min_val:16 ; min ;## # C_SRC : } popm R1 exitd </pre>	<pre> mov.w #0000H,R0; max ;## # C_SRC : unsigned int min=*p; mov.w[_p:16],R1 ; min ;## # C_SRC : while (1) L3: ;## # C_SRC : if (max == 0) min = max = *p; cmp.w #0000H,R0; max jne L7 mov.w[_p:16],R0 mov.w R0,R1 L7: _line 27 ;## # C_SRC : if (max < *p) max = *p; cmp.w[_p:16],R0 ; max jgeu L17 mov.w[_p:16],R0 L17: _line 28 ;## # C_SRC : if (min > *p) min = *p; cmp.w[_p:16],R1 ; min jleu L27 mov.w[_p:16],R1 L27: ;## # C_SRC : p++; add.l #00000002H,_p:16 ;## # C_SRC : if (*p == 0) break; cmp.w #0000H,[_p:16] jne L3 ;## # C_SRC : max_val = max; mov.w R0,_max_val:16 ; max ;## # C_SRC : min_val = min; mov.w R1,_min_val:16; min ;## # C_SRC : } popm R1 rts </pre>
---	---

図5.14 auto変数を明示的に初期化

5.12 配列の初期化

ループ文で2つの配列を初期化する場合は個別にループを作ってください。個別のループにすることにより sstr 命令に展開されるようになり速度が向上します。

ただし、3つ以上の配列を初期化する場合は1つのループ内で初期化を行ってください。sstr 命令は各レジスタの初期設定を行う必要があるため ROM 効率が悪く、3つ以上の配列を初期化する場合は1つのループ内で初期化を行ったほうが ROM 効率が向上します。

改善前	改善後
<pre>/* 一つのループ内での初期化 */ void loop2_1(void) { int i; for(i = 0; i < 10; i++){ a[i] = 0x0a; b[i] = 0x0b; } }</pre>	<pre>/* 個別ループでの初期化*/ void loop2_2(void) { int i; for(i = 0; i < 10; i++){ a[i] = 0x0a; } for(i = 0; i < 10; i++){ b[i] = 0x0b; } }</pre>
<pre>_loop2_1: pushm A0 mov.w #0000H,R0 L3: mov.w R0,A0 mov.b #0aH,_a:16[A0] mov.b #0bH,_b:16[A0] add.b #01H,A0 mov.w A0,R0 cmp.w #000aH,R0 jlt L3 popm A0 rts</pre>	<pre>_loop2_2: pushm R3,A1 mov.b #0aH,R0L mov.w #(_a&0FFFFH),A1 mov.w #0aH,R3 sstr.b mov.b #0bH,R0L mov.w #(_b&0FFFFH),A1 mov.w #0aH,R3 sstr.b popm R3,A1 rts</pre>

図5.15 配列の初期化

5.13 インクリメント/デクリメント

インクリメント、デクリメントは式から分けてください。コンパイラはインクリメント、デクリメント前の値を保持しようとします。

改善前	改善後
<pre>a[i++] = b[j++];</pre>	<pre>a[i] = b[j]; i++; j++;</pre>
<pre>mov.w -22[FB],R0; j add.w #0001H,-22[FB] ; j indexws.w R0 mov.w:g -22[FB],R0; b indexwd.w -2[FB]; i mov.w R0,-22[FB]; a add.w #0001H,-2[FB] ; i</pre>	<pre>indexws.w -4[FB]; j mov.w:g -24[FB],R0 ; b indexwd.w -2[FB]; i mov.w R0,-24[FB]; a add.w #0001H,-2[FB] ; i add.w #0001H,-4[FB] ; j</pre>

図5.16 インクリメント/ デクリメント

5.14 switch 文

caseの多いswitch文はcaseの値の隙間を減らしたほうが良いコードが出ます。隙間の大きいコードはif else形式に展開されるのに対し、隙間の小さいコードは分岐テーブルに展開されます。case値にはenum型を使うと良いでしょう。

改善前	改善後
<pre>switch(a) { case 100: ... case 200: ... case 300: ... }</pre>	<pre>switch(a) { case 1: ... case 2: ... case 3: ... }</pre>
<pre> cmp.w #0064H,R0 jeq L5 cmp.w #00c8H,R0 jeq L7 cmp.w #012cH,R0 jeq L9 ...</pre>	<pre>S2: jmp.w L47 L47: .word L45-S2&0ffffH .word L23-S2&0ffffH .word L25-S2&0ffffH ...</pre>

図5.17 switch文

5.15 浮動小数点即値

精度が必要ない場合浮動小数の即値には f を末尾につけるようにしてください。末尾に f が付かない場合 double 型として処理されます。

改善前	改善後
float f; f=f+123.456;	float f; f=f+123.456f;
<pre> push.l _f:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0,_f:16 </pre>	<pre> push.l _f:16 push.l #42f6e979H jsr.a __f4add add.l #08H,SP mov.l R2R0,_f:16 </pre>

図5.18 浮動小数点即値

5.16 外部変数のゼロクリア

外部変数の初期値は明示的にゼロクリアしなくてもスタートアップ時にゼロクリアされます。明示的にゼロクリアした場合、ROM 領域に 0 を確保しスタートアップで初期値転送することになり ROM 領域が無駄になります。外部変数の初期値でゼロクリアは行わないでください。

改善前	改善後
int i=0	int i;
<pre> .SECTION data_NE,DATA ._inspect 'U', 1, "data_NE", "data_NE", 0 .glb _i _i: .blkb 2 .SECTION data_NEI,ROMDATA ._inspect 'U', 1, "data_NEI", "data_NEI", 0 ;## # init data of i. .word 0000H </pre>	<pre> .SECTION bss_NE,DATA ._inspect 'U', 3, "bss_NE", "bss_NE", 0 .glb _i _i: .blkb 2 </pre>

図5.19 外部変数のゼロクリア

5.17 スタートアップの整理

標準入出力関数やメモリ管理関数を使わないことが分かっている場合、これらの初期設定を省くことでプログラムを高速化できます。ncrt0.a30 をアセンブル時に-D__STANDARD_IO__=1 および-D__HEAP__=1 を指定することによってそれぞれの初期化処理を無効化することができます。図 5.21 は High-performance Embedded Workshop 上で-D__STANDARD_IO__=1 と-D__HEAP__=1 を設定するダイアログです。

[illegible]

図5.20 スタートアップの整理

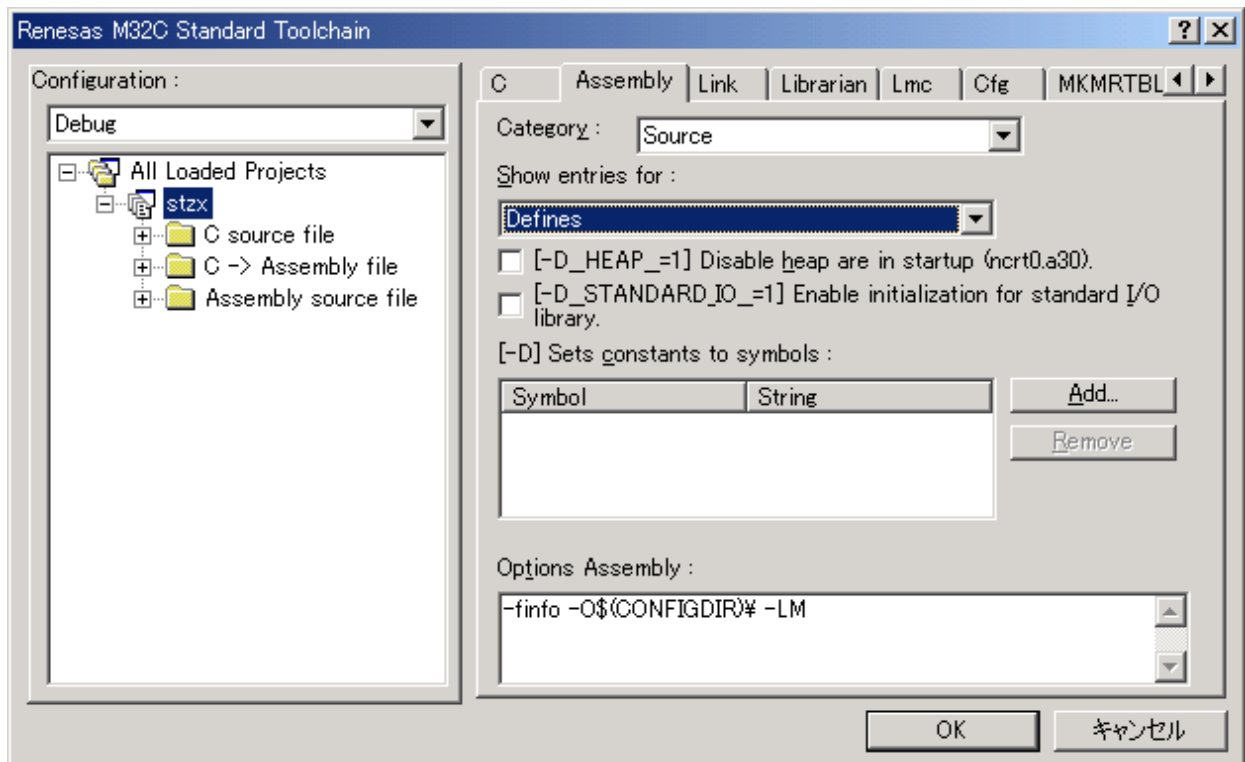


図 5.21 High-performance Embedded Workshopのスタートアップ整理用ダイアログ

5.18 ループ内はテンポラリを使用する

ループ内で外部変数を使用すると毎回メモリを参照することになります。テンポラリ変数を使用することでレジスタで計算されやすくなります。

改善前	改善後
<pre>int total; void func(int p[]) { int i; total=0; for(i=0;i<100;i++){ total+=p[i]; //毎回メモリを参照する } }</pre>	<pre>int total; void func(int p[]) { int i,tmp; tmp=0; for(i=0;i<100;i++){ tmp+=p[i]; //レジスタで計算される } total=tmp; }</pre>
<pre>_func: enter #02H pushm R2,A0 ;## # C_SRC : total=0; mov.w#0000H,_total:16 ;## # C_SRC : for(i=0;i<100;i++){ mov.w#0000H,-2[FB] ; i L1: ;## # C_SRC : for(i=0;i<100;i++){ cmp.w#0064H,-2[FB] ; i jge L5 ;## # C_SRC : total+=p[i]; //毎回メモリを参照する mov.w-2[FB],R0 ; i exts.w R0 mov.l R2R0,A0 shl.l #1,A0 add.l 8[FB],A0 ; p add.w [A0],_total:16 add.w #0001H,-2[FB] ; i jmp L1 L5: ;## # C_SRC : } popm R2,A0 exitd</pre> ループ内で 毎回メモリ を参照して いる	<pre>_func: enter #00H pushm R1,R2,R3,A0,A1 mov.l 8[FB],A0 ; p ;## # C_SRC : tmp=0; mov.w#0000H,R1; tmp ;## # C_SRC : for(i=0;i<100;i++){ mov.w#0000H,R0; i L1: ;## # C_SRC :tmp+=p[i];//レジスタで計算される mov.wR0,R3 ; i exts.w R0 mov.l R2R0,A1 shl.l #1,A1 add.l A0,A1; p add.w [A1],R1 add.w #0001H,R3; i mov.wR3,R0 ; i cmp.w#0064H,R0; i jltL1 ;## # C_SRC : total=tmp; mov.wR1,_total:16 ; tmp ;## # C_SRC : } popm R1,R2,R3,A0,A1 exitd</pre> レジスタ上での 演算に置き換わ っている

図5.22 ループ内はテンポラリを使用する

5.19 32ビット用数学関数

変数が float の場合、32 ビット用数学関数を使用すると実行速度が向上します。通常の数学関数を使用するとまず引数の double 型への昇格が起こり、double 型の値が返ってきて変数に代入する場合に float 型に変換されます。32 ビット用数学関数を使用するとすべて float のまま計算が行われます。

改善前	改善後
<pre>#include<math.h> float fdata,result; result=sin(fdata);</pre> doubleに変換 floatに変換	<pre>#include<mathf.h> float fdata,result; result=sinf(fdata)</pre> floatのまま計算
<pre>;;# C_SRC : result=sin(fdata); push.l _fdata:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R0,R1,R2,R3 jsr _sin add.l #08H,SP pushm R3,R2,R1,R0 .glb __f8tof4 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0,_result:16</pre>	<pre>;;# C_SRC : result=sinf(fdata); push.l _fdata:16 jsr _sinf add.l #04H,SP mov.l R2R0,_result:16</pre>

図5.23 32ビット用数学関数

5.20 なるべく unsigned を使う

整数型変数は必要がない限り符号なし整数を使用してください。unsignedの方が分岐命令に関してコード効率が良くなります。(ただしM16C/20,M16C/60シリーズのみ、逆にNC308ではunsignedをつけると符号拡張が起こりコード効率が悪くなる場合があります。)また符号付整数の比較を行う場合は可能な限り<=、>=、<、>を使用しないで==、!=を使用して条件判断を行ってください。コード効率が良くなります。

改善前	改善後
<pre>int data[100]; void main() { int i; int sum=0; for(i=0;i<100;i++) sum+=data[i]; }</pre>	<pre>int data[100]; void main() { unsigned int i; int sum=0; for(i=0;i!=100;i++)sum+=data[i]; }</pre>
<pre>;;# # C_SRC : int sum=0; mov.w #0000H,-4[FB] ; sum ;;# # C_SRC : for(i=0;i<100;i++)sum+=data[i]; mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC : for(i=0;i<100;i++)sum+=data[i]; cmp.w #0064H,-2[FB] ; i jge L5 mov.w -2[FB],A0 ; i shl.w #1,A0 add.w _data:16[A0],-4[FB]; sum add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>	<pre>;;# # C_SRC : int sum=0; mov.w #0000H,-4[FB] ; sum ;;# # C_SRC :for(i=0;i!=100;i++)sum+=data[i]; mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC :for(i=0;i!=100;i++)sum+=data[i]; cmp.w #0064H,-2[FB] ; i jeq L5 mov.w -2[FB],A0 ; i shl.w #1,A0 add.w _data:16[A0],-4[FB]; sum add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>

図5.24 unsigned の使用例

5.21 配列の添え字の型

配列の添え字は、その配列の1つの要素のサイズにより演算時において型拡張されます。

サイズが2バイト以上（char 型もしくは signed char 型以外）の場合

添え字は必ず int 型に拡張されて演算されます。

サイズが 64K 以上の far 型配列の場合

添え字は必ず long 型に拡張されて演算されます。

したがって、配列の添え字になる変数を char 型で宣言すると int 型への拡張が参照する度に行われコード効率が良くありません。このような場合、配列の添え字になる変数を int 型の変数にしてください。（*この最適化は NC30WA のみです。）

改善前	改善後
char i; int a[10]; a[i]=0;	int i; int a[10]; a[i]=0;
mov.b -1[FB],R0L ; i mov.b #00H,R0H shl.w #01H,R0 mova -21[FB],A0; a add.w R0,A0 mov.w #0000H,[A0]	mov.w -2[FB],R0 ; i shl.w #01H,R0 mova -22[FB],A0; a add.w R0,A0 mov.w #0000H,[A0]

図5.25 配列の添え字の型

5.22 プロトタイプ宣言の活用

本コンパイラでは、関数のプロトタイプ宣言を行うことにより、効率の良い関数呼び出しを行うことができます。すなわち、本コンパイラでは関数のプロトタイプ宣言を行わない場合、その関数を呼び出す場合に、その関数の引数を表に示す規則によりスタック領域に積みます。

表5.4 引数に関するスタックの使用規則

データ型	スタックに積む場合の規則
char 型 signed char 型	int 型に拡張して積む。
float 型	double 型に拡張して積む。
その他の型	型の拡張は行わずに積む

このため、関数のプロトタイプ宣言を行わない場合、冗長な型拡張を行う場合があります。関数のプロトタイプ宣言を行うことにより、これらの冗長な型拡張を抑止し、また、レジスタに引数を割り当てることが可能になるため、効率の良い関数呼び出しを行うことができます。

改善前	改善後
<pre> int main() { char data1; char data2; char data3; int data; data=f(data1,data2,data3); } int f(i, j, k) char i,j,k; { ... } </pre>	<pre> int f(char ,char ,char); int main() { char data1; char data2; char data3; int data; data=f(data1,data2,data3); } int f(i, j,k) char i,j,k; { ... } </pre>
<pre> extz -2[FB],R0 ; data3 push.w R0 extz -2[FB],R0 ; data2 push.w R0 extz -2[FB],R0 ; data1 push.w R0 jsr _f </pre>	<pre> push.b -2[FB]; data3 push.b -2[FB]; data2 mov.b -2[FB],R0L ; data1 jsr \$f </pre>

図5.26 プロトタイプ宣言の活用

5.23 char 型の範囲でしか値を返さない関数の戻り値を char 型にする。

char 型の範囲でしか値を返さない関数の戻り値は char 型にすると ROM 領域が圧縮できます。使用するデータの型は可能な限り小さくしてください。

改善前	改善後
<pre> int func2(void) { switch (x) { case 0: return 255; case 1: return 254; default: return 253; } } </pre>	<pre> char func2(void) { switch (x) { case 0: return 255; case 1: return 254; default: return 253; } } </pre>
<pre> .glb _func2 _func2: ;## # C_SRC : switch (x) { mov.w_x:16,R0 jeq L3 cmp.w #0001H,R0 jeq L5 jmp L7 ;## # C_SRC : case 0: L3: ;## # C_SRC : return 255; mov.w #00ffH,R0 rts ;## # C_SRC : case 1: L5: ;## # C_SRC : return 254; mov.w #00feH,R0 rts ;## # C_SRC : default: L7: ;## # C_SRC : return 253; mov.w #00fdH,R0 rts </pre>	<pre> .glb _func2 _func2: ;## # C_SRC : switch (x) { mov.w_x:16,R0 jeq L3 cmp.w #0001H,R0 jeq L5 jmp L7 ;## # C_SRC : case 0: L3: ;## # C_SRC : return 255; mov.b #0ffH,R0L rts ;## # C_SRC : case 1: L5: ;## # C_SRC : return 254; mov.b #0feH,R0L rts ;## # C_SRC : default: L7: ;## # C_SRC : return 253; mov.b #0fdH,R0L rts </pre>

図5.27 char型の範囲でしか値を返さない関数の戻り値をchar型にする。

5.24 bss 領域のクリア処理のコメントアウト

スタートアッププログラム ncrt0.a30 には bss 領域のクリア処理が含まれています。この処理は C 言語の言語仕様として初期化されていない変数は初期値として 0 を持つという規格を満たすための処理です。

例えば、図 5.28 に示す記述の場合、初期値を記述していませんので、スタートアップ処理時に初期値として 0 を与える処理（bss 領域のクリア処理）が必要になります。

```
static int i;
```

図5.28 初期値を持たない変数の宣言例

応用によっては初期値を持たない変数を 0 クリアする必要が無いものがあります。この場合にはスタートアッププログラム内の bss 領域のクリア処理部をコメントアウトすれば、スタートアップ処理を高速化することができます。

```
=====
; NEAR area initialize.
;-----
; bss zero clear
;-----
;      N_BZERO bss_SE_top,bss_SE
;      N_BZERO bss_SO_top,bss_SO
;      N_BZERO bss_NE_top,bss_NE
;      N_BZERO bss_NO_top,bss_NO
;
; (省略)
;
;-----
; FAR area initialize.
;-----
; bss zero clear
;-----
;      BZERO bss_FE_top,bss_FE
;      BZERO bss_FO_top,bss_FO
```

図5.29 bss領域のクリア処理のコメントアウト例

5.25 生成コードの削減

int 型で宣言しているデータで次のようなデータ範囲に収まるものがある場合生成コードが削減できます。

0 ~ 255 以内 → unsigned char 型に修正してください。

-128 ~ 127 以内 → signed char 型に修正してください。

変数の型を小さくすることで比較命令や加算命令が小さくなり ROM 効率の向上につながります。

改善前	改善後
<pre> int型 int data[128]; void main(void) { int cnt = 128; int i; int sum=0; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>	<pre> unsigned char型 void main(void) { unsigned char cnt = 128; unsigned char i; int data[128]; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>
<pre> ;## # C_SRC : int cnt = 128; mov.w #0080H,-6[FB] ; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.w #0000H,-4[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.w -6[FB],-4[FB] ; cnt i jge L5 ;## # C_SRC : sum += data[i]; mov.w -4[FB],A0 ; i shl.w #1,A0 mova -262[FB],A1 ; data add.l A1,A0 add.w [A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0,A1 exitd </pre>	<pre> ;## # C_SRC : unsigned char cnt = 128; mov.b #80H,-2[FB] ; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.b #00H,-1[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.b -2[FB],-1[FB] ; cnt i jgeu L5 ;## # C_SRC : sum += data[i]; mov.b -1[FB],A0 ; i shl.w #1,A0 mova -260[FB],A1 ; data add.l A1,A0 add.w [A0],-4[FB] ; sum add.b #01H,-1[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0,A1 exitd </pre>

図5.30 生成コードの削減（１）

5. 効率の良いプログラミング技法

long 型で宣言しているデータで次のようなデータ範囲に収まるものがある場合生成コードが削減できます。

0 ~ 65535 → unsigned int 型に修正してください。

-32768 ~ 32767 → signed int 型に修正してください。

変数の型を小さくすることで比較命令や加算命令 が小さくなり、ROM 効率向上につながります。

改善前	改善後
<pre> unsigned int data[65535]; void main(void) { long cnt = 65535; long i; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>	<pre> unsigned int data[65535]; void main(void) { unsigned int cnt = 65535; unsigned int i; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>
<pre> ;## # C_SRC : long cnt = 65535; mov.l #0000ffffH,-10[FB] ; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.l #00000000H,-6[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.l -10[FB],-6[FB] ; cnt i jge L5 ;## # C_SRC : sum += data[i]; mov.l -6[FB],A0 ; i shl.l #1,A0 add.w _data:16[A0],-2[FB] ; sum add.l #00000001H,-6[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0 exitd </pre>	<pre> ;## # C_SRC : unsigned int cnt = 65535; mov.w #0ffffH,-6[FB] ; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.w #0000H,-4[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.w -6[FB],-4[FB] ; cnt i jgeu L5 ;## # C_SRC : sum += data[i]; mov.w -4[FB],A0 ; i shl.l #1,A0 add.w _data:16[A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0 exitd </pre>

図5.31 生成コードの削減（ 2 ）

6. シミュレータデバッガの活用

本章では NC30WA に同梱しているシミュレータデバッガの有効な活用方法について記述します。
本章の内容一覧を示します。

No.	大項目	中項目	参照
1	仮想割り込み機能を利用する	ボタン押下により仮想割り込みを入れる	6.1.1
2		一定間隔で仮想割り込みを入れる	6.1.2
3		指定サイクルで仮想割り込みを入れる	6.1.3
4		指定アドレスの命令の実行時に仮想割り込みを入れる	6.1.4
5	仮想ポート入出力機能を利用する	ボタン押下によりデータを入力する	6.2.1
6		指定アドレスのリード時に仮想ポートからデータを入力する	6.2.2
7		指定サイクルで仮想ポートからデータを入力する	6.2.3
8		仮想割り込み発生時に仮想ポートからデータを入力する	6.2.4
9		仮想出力ポートへ出力したデータを確認する	6.2.5
10	仮想 LED やラベルでメモリ内容を確認する		6.3
11	デバッグ用に printf を使用する		6.4
12	I/O スクリプトの活用		6.5

6.1 仮想割り込み機能を利用する

6.1.1 ボタン押下により仮想割り込みを入れる

■説明

仮想割り込みはボタンをある割り込み要因に見立てて、ボタンを押下することにより手動で仮想的な割り込みを発生させることができます。

ボタンには割り込み優先度と割り込み条件を指定します。

■設定方法

1. メニューより[表示->グラフィック->GUI I/O]をクリックし、GUI ウィンドウを表示します。
2. ボタン作成アイコンをクリックするか、右クリックから[ボタン作成]をクリックし、仮想割り込みを発生させるボタンを作成します。

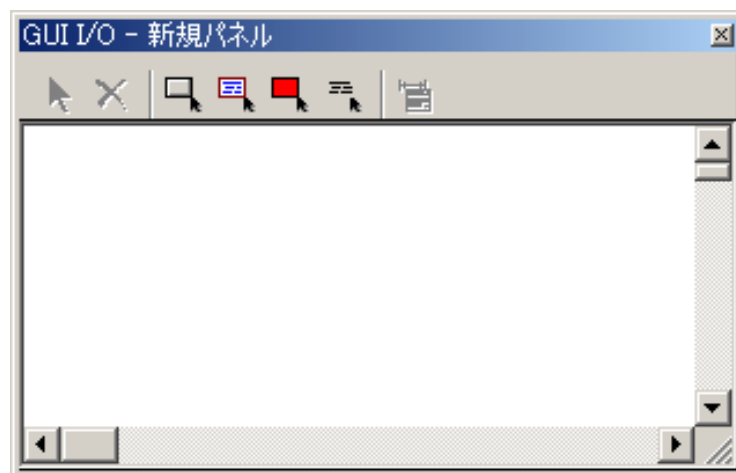


図6.1 GUIウィンドウ

3. 作成したボタンをクリックしてボタンの設定画面を表示してください。

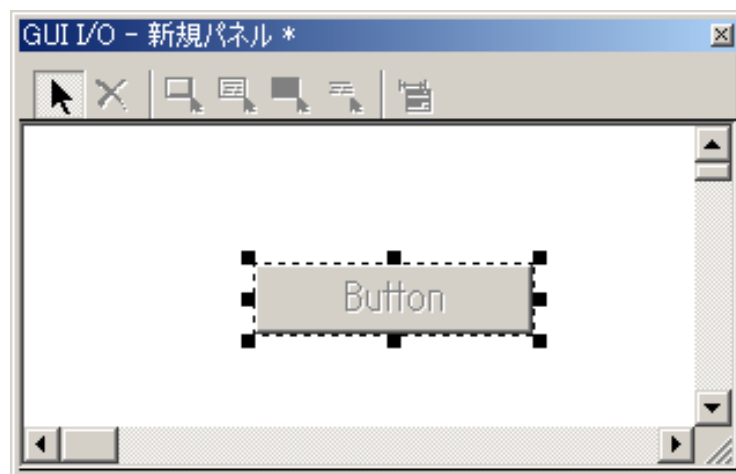


図6.2 GUIウィンドウ（ボタン配置後）

4. ボタン種別の選択を「割り込み用」にし、割り込みのベクタ、IPL(優先順位)を設定します。

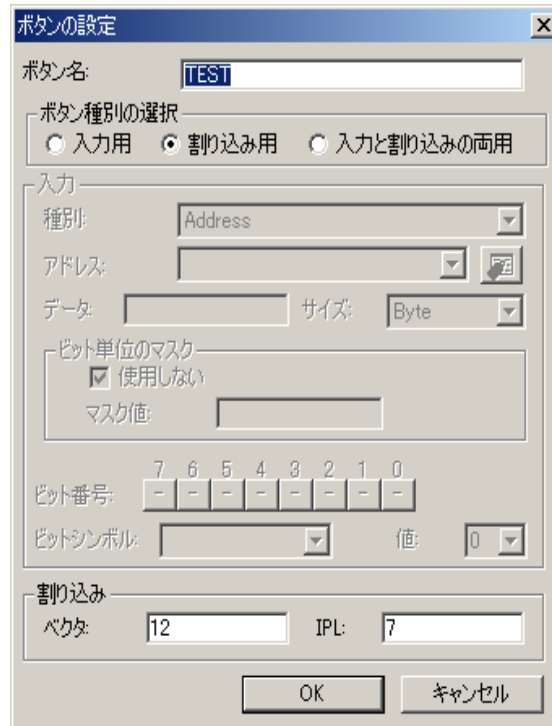


図6.3 ボタン設定

5. ベクタテーブルを設定しているファイルで割り込みのベクタを設定します。今回の場合は「vector 12」と記述されている行の「.lword dummy_int」を「.lword _test」に書き換えると、ボタンを押したときにソースコード中の” test” 関数が呼び出されます。

```

;-----
; variable vector section
;-----
.section    vector,ROMDATA ; variable vector table
.org      VECTOR_ADR

.lword  dummy_int      ; vector 0
.lword  dummy_int      ; vector 1

:

.glob  _test
.lword  _test          ; vector 12

:

```

6. デバッグ中に2.で作成したボタンをクリックすると仮想割り込みが発生します。

6.1.2 一定間隔で仮想割り込みを入れる

■説明

I/O タイミング設定ウィンドウで、一定時間間隔に同期した仮想割り込みを設定することができます。

■設定方法

1. メニューより[表示->CPU->I/O タイミング設定]をクリックし、I/O タイミング設定ウィンドウを表示します。
2. 時間間隔同期割り込みアイコンをクリックするか、右クリックから[時間間隔同期割り込み]をクリックし、時間間隔同期割り込み設定画面を表示します。

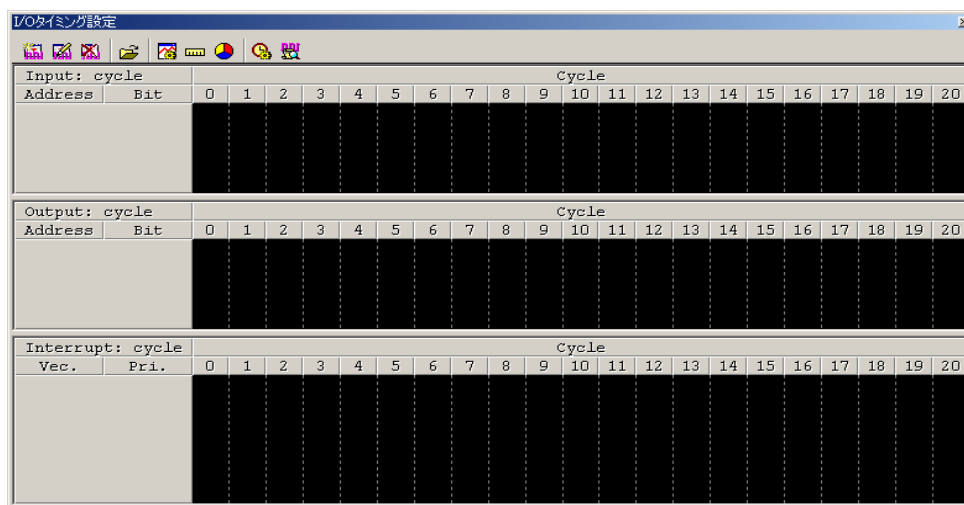


図6.4 I/Oタイミング設定ウィンドウ

3. 読み込みボタンを押すことで、設定ファイルを読み込むことができます。
4. 設定ファイルを読み込まない場合は、時間間隔、ベクタ、優先順位を入力し、追加ボタンを押します。

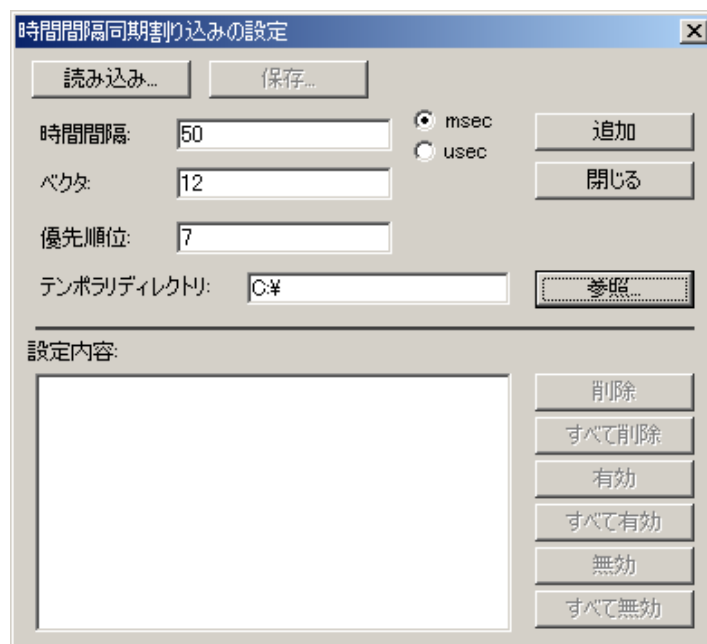


図6.5 時間間隔同期割り込み設定

5. 設定内容に先ほど入力した設定が表示されます。
6. 保存ボタンを押すと、現在の設定を保存することができます。

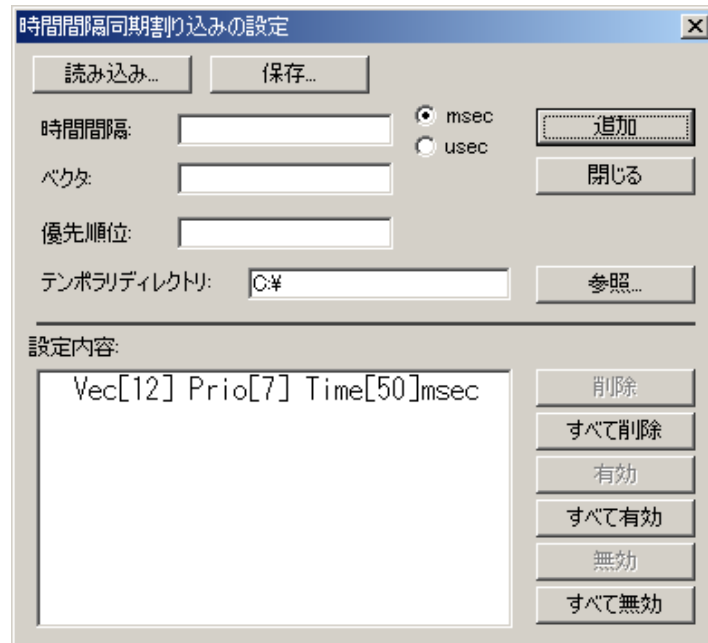


図6.6 時間間隔同期割り込み設定（設定入力後）

7. 複数の設定を追加することもできます。
8. 各設定毎に有効・無効を切り替えることもできます。

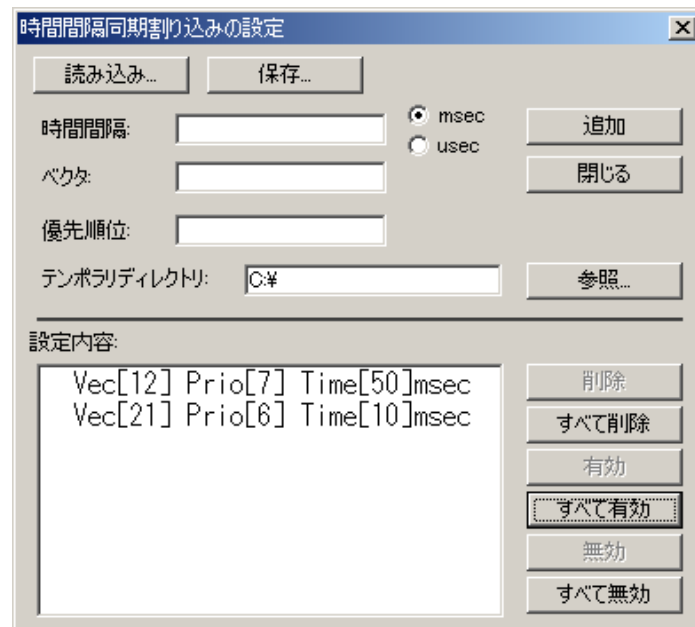


図6.7 時間間隔同期割り込み設定（複数設定）

6. シミュレータデバッガの活用

6.1.3 指定サイクルで仮想割り込みを入れる

■説明

I/O タイミング設定ウィンドウで、指定サイクルでの仮想割り込みを設定することができます。

■設定方法

1. メニューより[表示->CPU->I/O タイミング設定]をクリックし、I/O タイミング設定ウィンドウを表示します。
2. データ設定アイコンをクリックするか、右クリックから[設定]をクリックしてください。

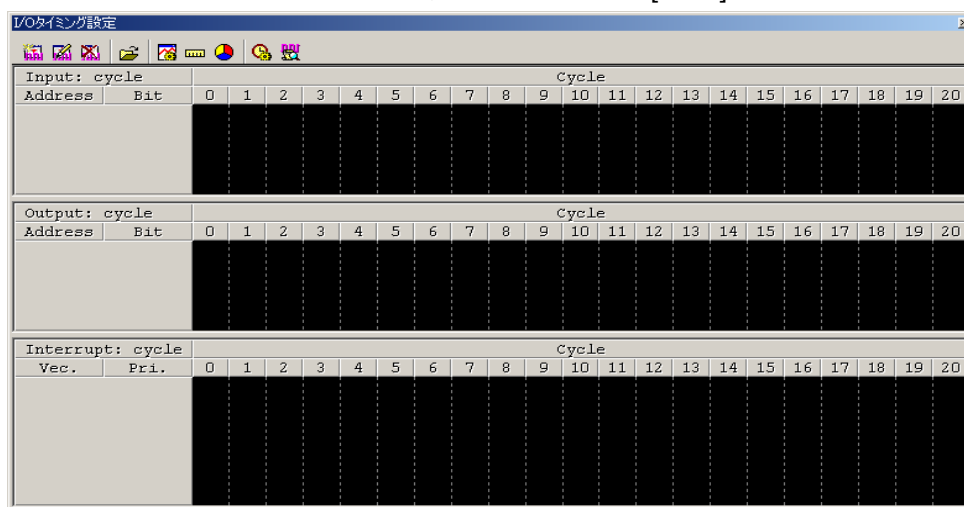


図6.8 I/Oタイミング設定ウィンドウ

3. 「仮想割り込みの設定」を選択して、[次へ>]を押下してください。

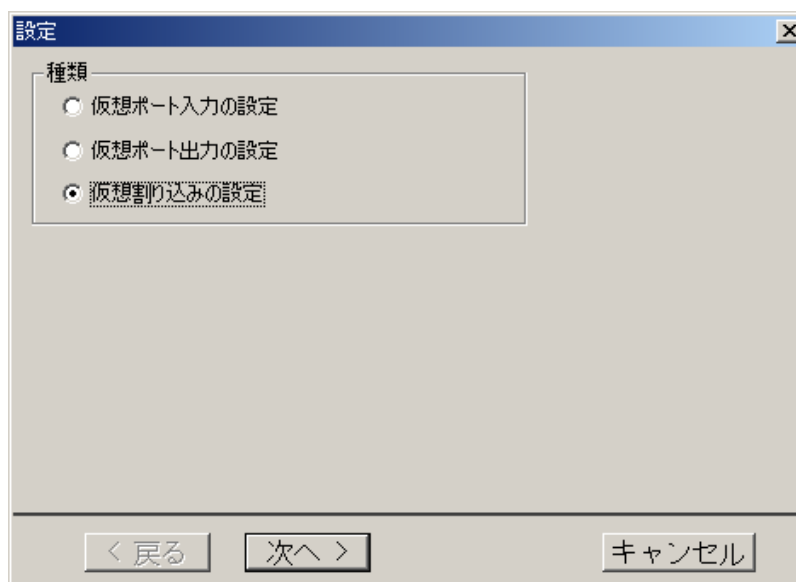


図6.9 仮想割り込みの設定

4. 割り込み発生タイミングをサイクルにしてください。
5. 開始サイクルと終了サイクル、ベクタと優先順位を設定し、[次へ>]を押下してください。

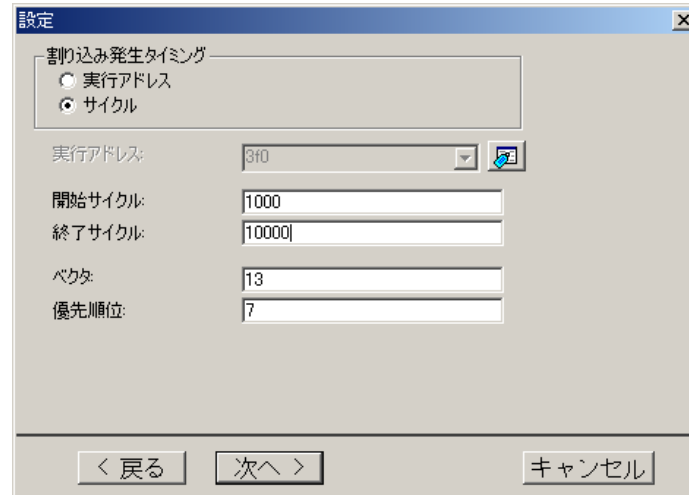


図6.10 サイクル・ベクタの設定

6. 仮想割り込みを設定したいサイクルの箇所までマウスを移動し、左ボタンをクリックすると、割り込み発生を設定できます。
7. すべての割り込み発生を設定したら[次へ>]を押下してください。



図6.11 割り込み発生の設定

6. シミュレータデバッガの活用

8. 自動的に I/O スクリプトファイルが編集されるので、保存してください。

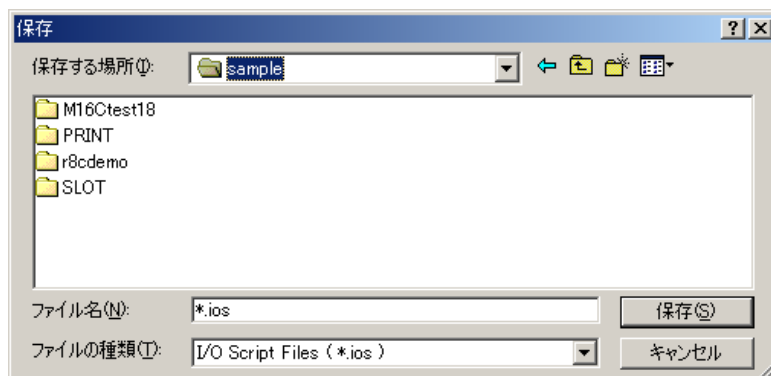


図6.12 保存ダイアログ

9. I/O スクリプトファイルは下記になります。

```
; IOSSCRIPT FILE FOR I/O WINDOW (INT WAITC)
{
cycle 1001
int 13 , 7
waitc 31
int 13 , 7
waitc 20
int 13 , 7
waitc 12
int 13 , 7
}
```

6.1.4 指定アドレスの命令の実行時に仮想割り込みを入れる

■説明

I/O タイミング設定ウィンドウで、指定アドレスの命令の実行時に発生する仮想割り込みを設定することができます。

■設定方法

1. 6.1.3.「指定サイクルで仮想割り込みを入れる」の1～3の操作をし、仮想割り込み設定のダイアログを表示してください。
2. 割り込み発生タイミングを実行アドレスにしてください。
3. 実行アドレス、ベクタと優先順位を設定し、[次へ>]を押下してください。

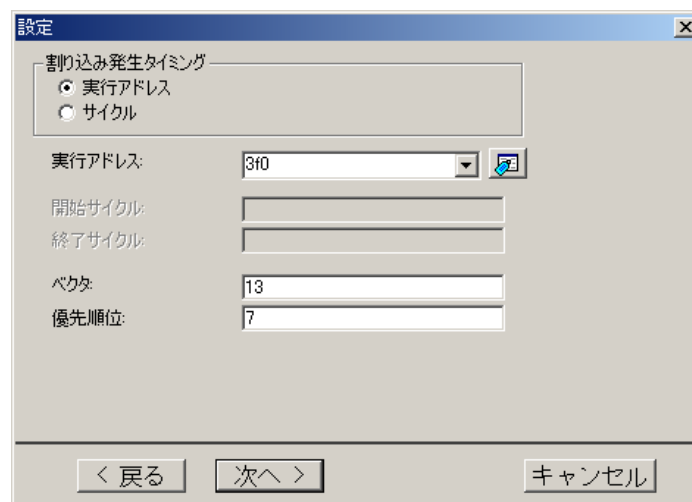


図6.13 実行アドレスとベクタの設定

4. 仮想割り込みを設定した実行アドレスが何回実行されたときに割り込みを発生するかを設定します。実行回数の箇所までマウスを移動し、左ボタンをクリックし、割り込み発生を設定してください。
5. すべての割り込み発生を設定したら[次へ>]を押下してください。

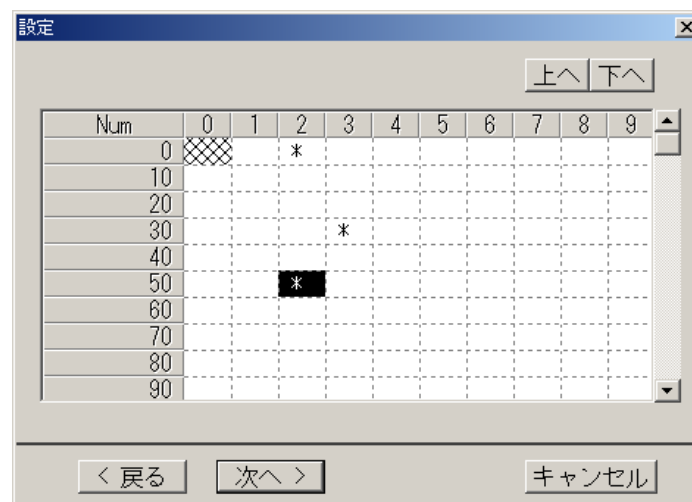


図6.14 割り込み発生の設定

6. シミュレータデバッガの活用

6. 自動的に I/O スクリプトファイルが編集されるので、保存してください。

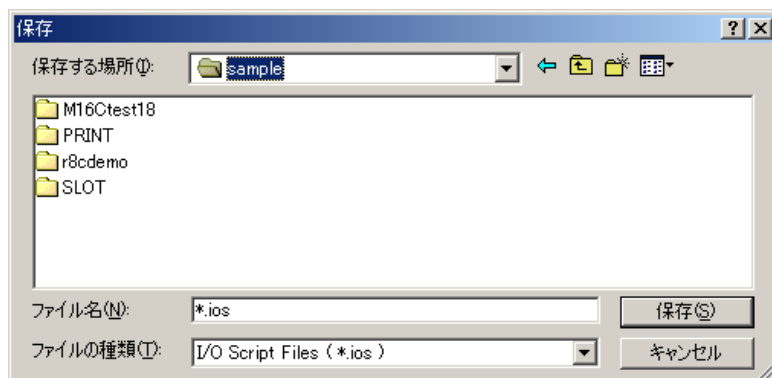


図6.15 保存ダイアログ

7. I/O スクリプトファイルは下記になります。

```
; IOScript FILE FOR I/O WINDOW (INT ISFETCH)
{
pass #isfetch:0x3f0 , 2
int 13 , 7
pass #isfetch:0x3f0 , 31
int 13 , 7
pass #isfetch:0x3f0 , 19
int 13 , 7
}
```

6.2 仮想ポート入出力機能を利用する

6.2.1 ボタン押下によりデータを入力する

■説明

ボタンを押下することにより手動で仮想ポート入力を発生させることができます。
ボタンにはアドレスまたはビットシンボルを指定します。

■設定方法

1. GUI ウィンドウにボタンを配置します。6.1.1「ボタン押下により仮想割り込みを入れる」の1～3を参照してください。
2. ボタン種別の選択を「入力用」にします。
3. 種別を「Address」「Address & Bit No.」「Bit Symbol」から選択します。
4. 「Address」のときにはアドレスとデータ、データのサイズを設定します。



図6.16 ボタン設定

6. シミュレータデバッガの活用

5. 「Address & Bit No.」のときには、ビット単位のマスクを使用しないときはアドレスとビット番号を設定します。ビット単位のマスクを使用するときはアドレスとデータ、データのサイズ、マスク値を設定します。

The screenshot shows the 'ボタン設定' (Button Settings) dialog box. The 'ボタン名' (Button Name) is 'TEST'. Under 'ボタン種別の選択' (Button Type Selection), '入力用' (Input) is selected. In the '入力' (Input) section, '種別' (Type) is 'Address & Bit No.', 'アドレス' (Address) is '_TEST', 'データ' (Data) is '0x01', and 'サイズ' (Size) is 'Byte'. In the 'ビット単位のマスク' (Bit Mask) section, '使用しない' (Do not use) is checked. Below this, the 'ビット番号' (Bit Number) is shown as a row of eight boxes with values: 7, 6, 5, 4, 3, 2, 1, 0. The 'ビットシンボル' (Bit Symbol) is empty, and the '値' (Value) is '0'. At the bottom, there are 'OK' and 'キャンセル' (Cancel) buttons.

図6.17 ボタン設定

6. 「Bit Symbol」のときにはビットシンボルと値を設定します。

The screenshot shows the 'ボタン設定' (Button Settings) dialog box. The 'ボタン名' (Button Name) is 'TEST'. Under 'ボタン種別の選択' (Button Type Selection), '入力用' (Input) is selected. In the '入力' (Input) section, '種別' (Type) is 'Bit Symbol', 'アドレス' (Address) is empty, 'データ' (Data) is empty, and 'サイズ' (Size) is 'Byte'. In the 'ビット単位のマスク' (Bit Mask) section, '使用しない' (Do not use) is checked. Below this, the 'ビット番号' (Bit Number) is shown as a row of eight boxes with values: 7, 6, 5, 4, 3, 2, 1, 0. The 'ビットシンボル' (Bit Symbol) is 'BIT1', and the '値' (Value) is '1'. At the bottom, there are 'OK' and 'キャンセル' (Cancel) buttons.

図6.18 ボタン設定

7. ボタンを押したときに設定したアドレス、またはビットシンボルに値が入ります。

6.2.2 指定アドレスのリード時に仮想ポートからデータを入力する

■説明

I/O タイミング設定ウィンドウで、指定アドレスのリード時に仮想ポートからの入力の設定ができます。

■設定方法

1. メニューより[表示->CPU->I/O タイミング設定]をクリックし、I/O タイミング設定ウィンドウを表示します。
2. データ設定アイコンをクリックするか、右クリックから[設定]をクリックしてください。

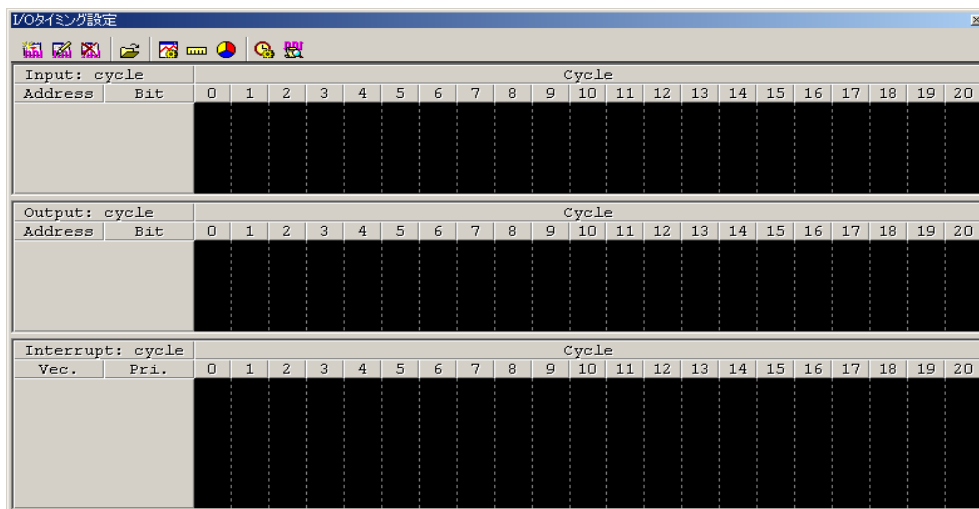


図6.19 I/Oタイミング設定ウィンドウ

3. 「仮想入力ポートの設定」を選択して、[次へ>]を押下します。

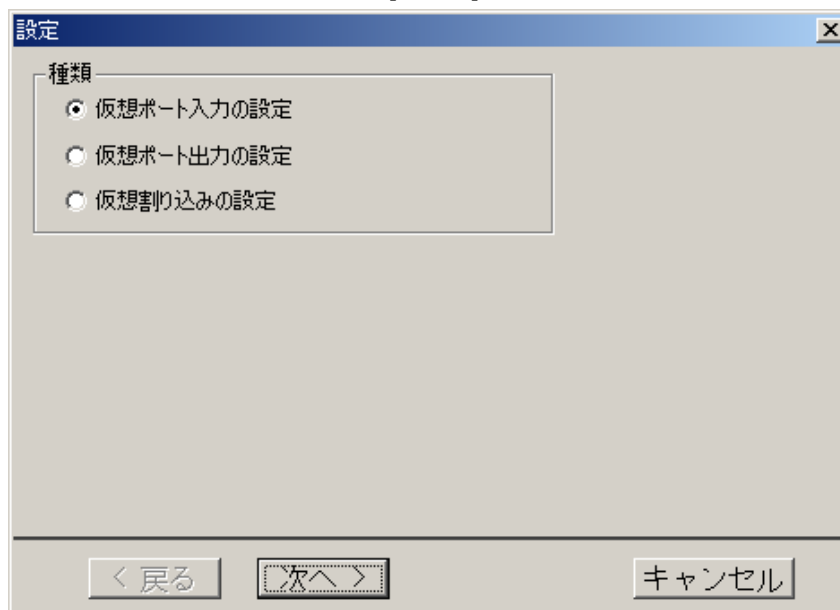


図6.20 処理種類設定

6. シミュレータデバッグの活用

4. データ入力タイミングを「リードアクセス」にしてください。
5. 入力アドレス欄に仮想ポート入力を行いたいアドレスを 16 進数値で入力してください。
6. リードアドレス欄にメモリの読み込みが行われるアドレスを 16 進数値で入力して下さい（ここで指定したメモリにリードアクセスが発生した時に仮想ポート入力を行います）。
7. 入力アドレスとリードアドレスを同じアドレスにしておけば、そのアドレスにアクセスする度に、次のデータが参照されます。
8. [次へ>]を押下します。

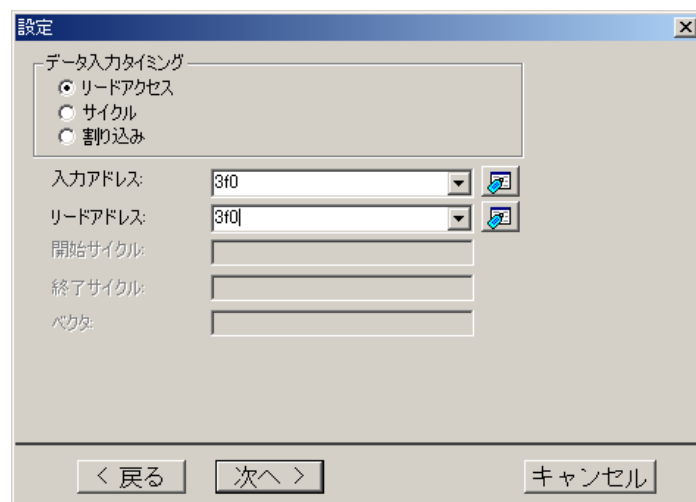


図6.21 リードアクセスの設定

9. 仮想入力ポートから入るデータを設定します。データを設定したいリードアクセス発生回数の箇所までマウスを移動し左ボタンをダブルクリックすると、入力カーソルが表示されるので、データを 16 進数値で入力してください。
10. データを入力し終わったら、[次へ>]を押下します。



図6.22 入力データ設定

1 1 . 自動的に I/O スクリプトファイルが編集されるので、保存してください。

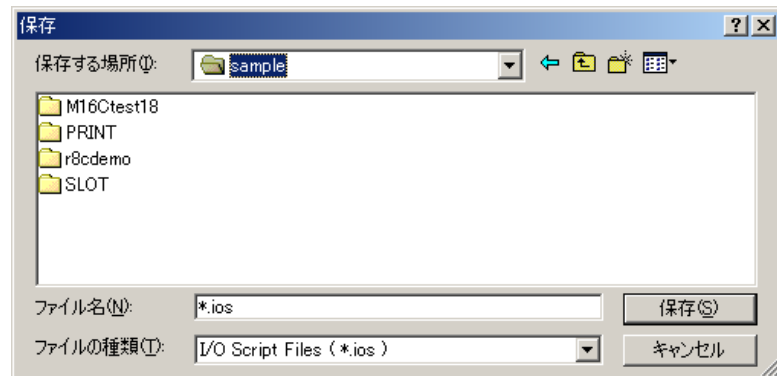


図6.23 保存ダイアログ

1 2 . I/O スクリプトファイルは下記になります。

```
; IOSRIPT FILE FOR I/O WINDOW (SET ISREAD)
{
pass #isread:0x3f0 , 1
set [0x3f0] = 0x68
pass #isread:0x3f0 , 1
set [0x3f0] = 0x65
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6f
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0xa
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
}
```

6. シミュレータデバッガの活用

1 3 . 下記のコードは仮想ポート入力の例です。

<pre>#include <stdio.h> char *PORT_IN; int i; char buf[10]; void main(void) { PORT_IN = (char *)0x3f0; for(i=0; i<10; i++){ buf[i] = *PORT_IN; if(buf[i] == '¥0') break; } printf("%s", buf); }</pre>	→ 入力アドレス / リードアドレスで指定した アドレス → for文 → 読み込む度に次のデータが参照される → 0x00をエンドコードとする → 標準入出力に入力されたデータを出力する
---	---

6.2.3 指定サイクルで仮想ポートからデータを入力する

■説明

I/O タイミング設定ウィンドウで、指定サイクルからの仮想ポート入力を設定できます。

■設定方法

1. I/O タイミングウィンドウよりデータ設定ダイアログを表示します。6.2.2「指定アドレスのリード時に仮想ポートからデータを入力する」の1～3を参照してください。
2. データ入力タイミングを「サイクル」にしてください。
3. 入力アドレス欄に仮想ポート入力を行いたいアドレスを16進数値で入力してください。
4. 開始サイクルと終了サイクルを入力し、[次へ>]を押下します。

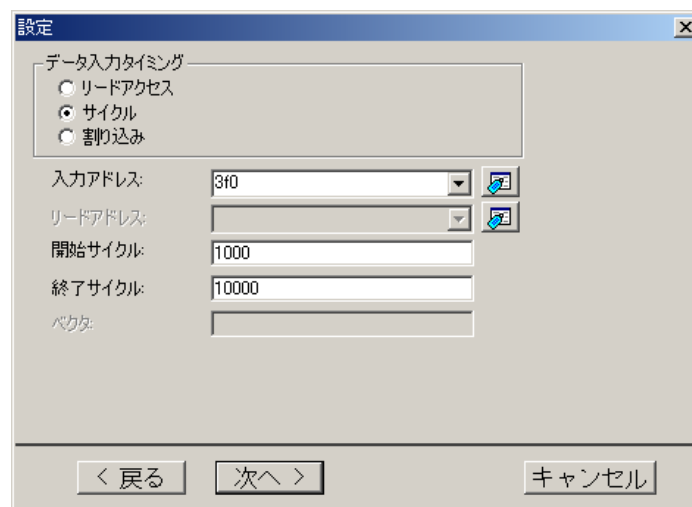


図6.24 サイクルの設定

5. 仮想入力ポートから入るデータを設定します。データを設定したいサイクルの箇所までマウスを移動し左ボタンをダブルクリックすると、入力カーソルが表示されるので、データを16進数値で入力してください。
6. データを入力し終わったら、[次へ>]を押下します。

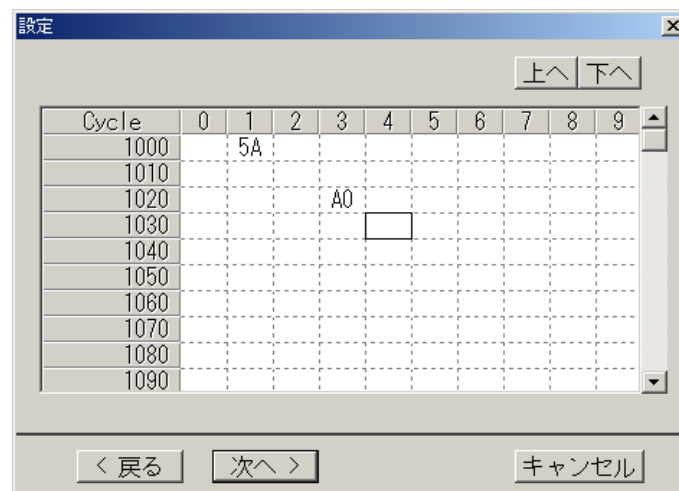


図6.25 入力データ設定

6. シミュレータデバッガの活用

7. 自動的に I/O スクリプトファイルが編集されるので、保存してください。

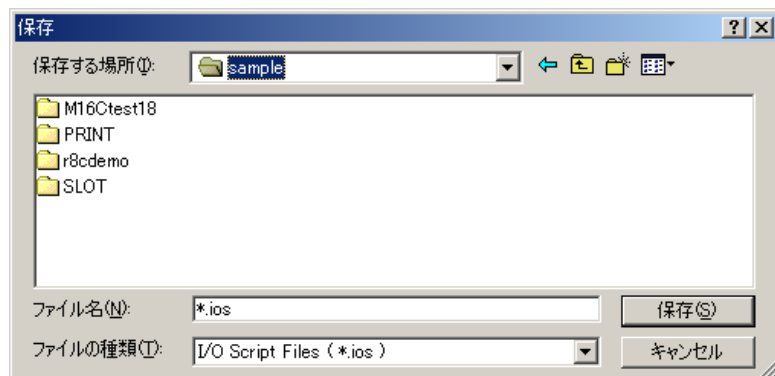


図6.26 保存ダイアログ

8. I/O スクリプトファイルは下記になります。

```
; IOSRIPT FILE FOR I/O WINDOW (SET WAITC)
{
cycle 1001
set [0x3f0] = 0x5a
waitc 22
set [0x3f0] = 0xa0
}
```

6.2.4 仮想割り込み発生時に仮想ポートからデータを入力する

■説明

I/O タイミング設定ウィンドウで、仮想割り込み発生時のポート入力を設定することができます。

■設定方法

1. I/O タイミングウィンドウよりデータ設定ダイアログを表示します。6.2.2「指定アドレスのリード時に仮想ポートからデータを入力する」の1～3を参照してください。
2. データ入力タイミングを「割り込み」に選択してください。
3. 入力アドレス欄に仮想ポート入力を行いたいアドレスを16進数値で入力してください。
4. 監視する割り込みのベクタを指定し、[次へ>]を押下します。

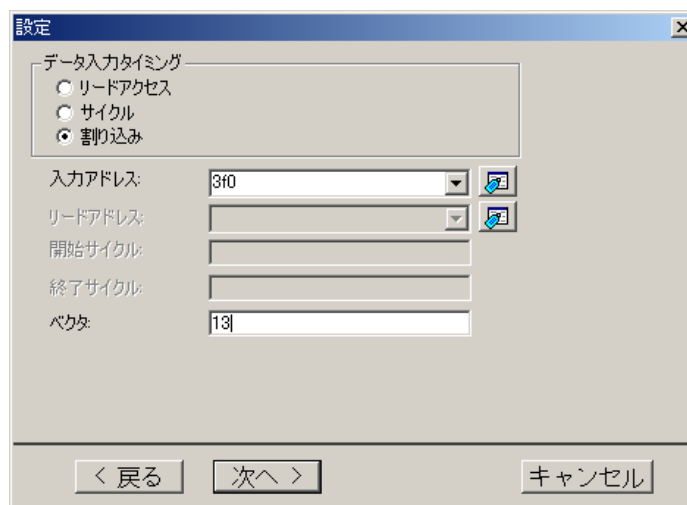


図6.27 割り込みの設定

5. 仮想入力ポートから入るデータを設定します。データを設定したいリードアクセス発生回数の箇所までマウスを移動し左ボタンをダブルクリックすると、入力カーソルが表示されるので、データを16進数値で入力してください。
6. データを入力し終わったら、[次へ>]を押下します。

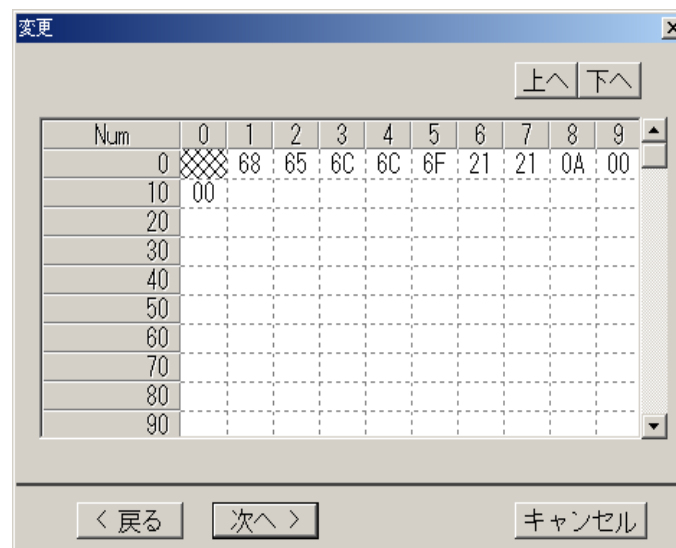


図6.28 入力データ設定

6. シミュレータデバッガの活用

7. 自動的に I/O スクリプトファイルが編集されるので、保存してください。

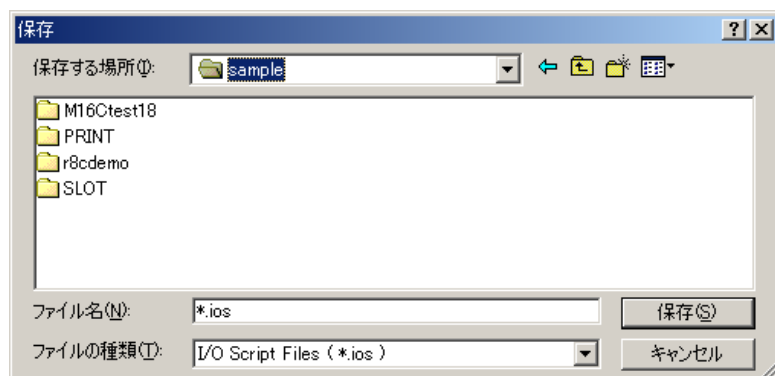


図6.29 保存ダイアログ

8. I/O スクリプトファイルは下記になります。

```
; IOSSCRIPT FILE FOR I/O WINDOW (SET ISINT)
{
pass #isint:13 , 1
set [0x3f0] = 0x68
pass #isint:13 , 1
set [0x3f0] = 0x65
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6f
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0xa
pass #isint:13 , 1
set [0x3f0] = 0x0
pass #isint:13 , 1
set [0x3f0] = 0x0
}
```

6.2.5 仮想出力ポートへ出力したデータを確認する

■ 説明

I/O タイミング設定ウィンドウや出力ポートウィンドウで、仮想出力ポートへ出力したデータの確認ができます。

(1) I/O タイミング設定ウィンドウ

1. メニューより [表示->CPU->I/O タイミング設定] をクリックし、I/O タイミング設定ウィンドウを表示します。
2. データ設定アイコンをクリックするか、右クリックから[設定]をクリックしてください。

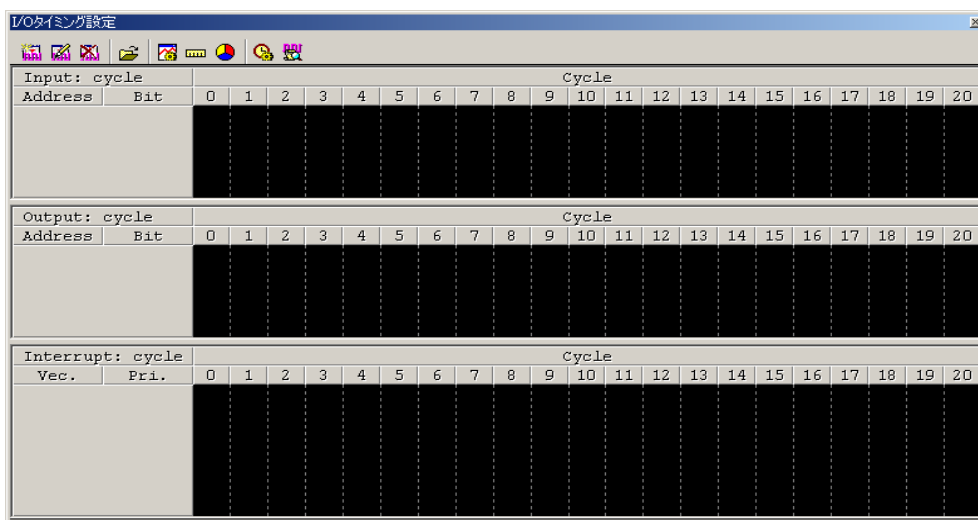


図6.30 I/Oタイミング設定ウィンドウ

3. 「仮想ポート出力の設定」を選択して、[次へ>]を押下します。

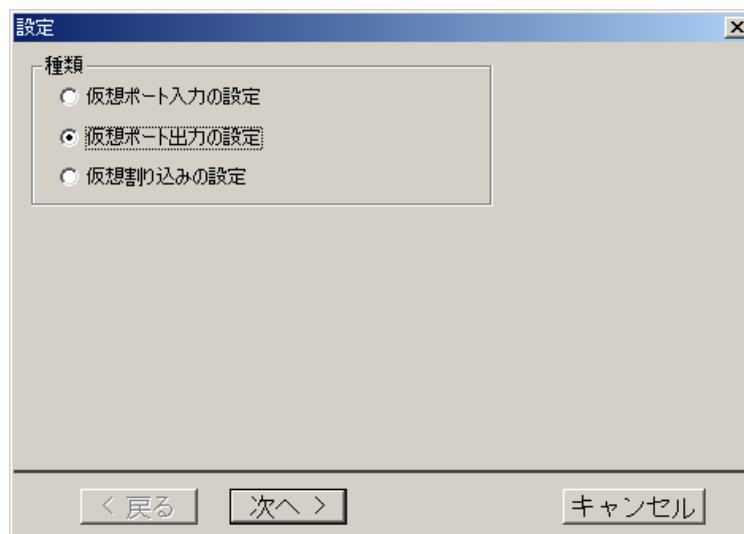


図6.31 処理種類選択

6. シミュレータデバッグの活用

4. 出力アドレスを設定し、[次へ>]を押下します。

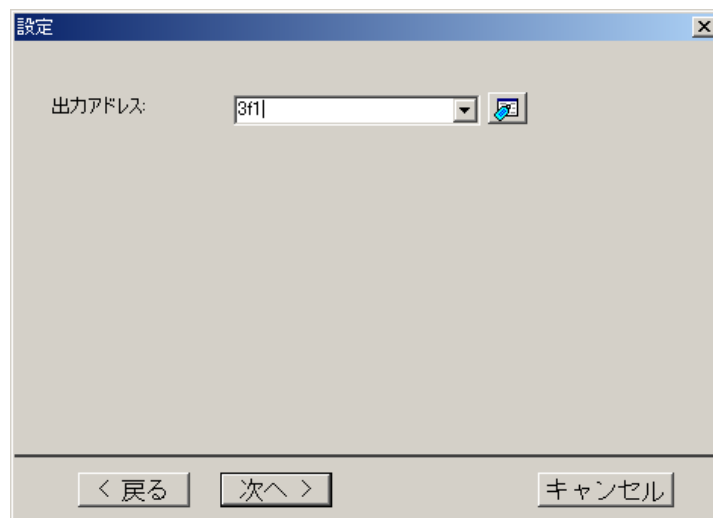


図6.32 出力アドレス設定

5. 仮想ポート出力の結果を保存する I/O スクリプトファイルを指定するためのダイアログが表示されます。ファイル名をつけて、保存してください。

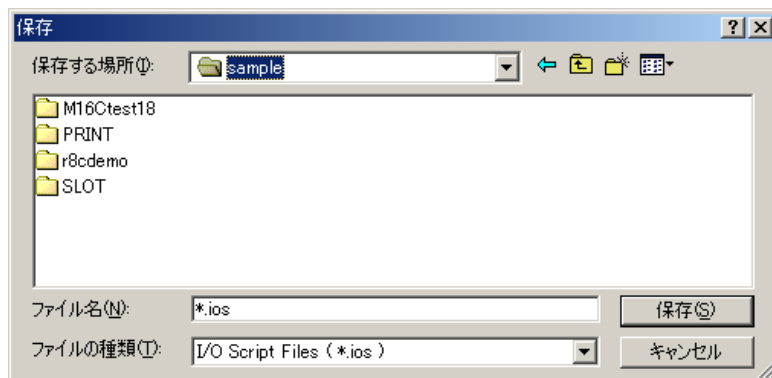


図6.33 保存ダイアログ

6. 下記のコードは仮想ポート出力の例です。

```
#include <stdio.h>
char *PORT_OUT;
static int i;
char buf[10];

void main(void)
{
    PORT_OUT = (char *)0x3f1;           → 出力アドレスで指定したアドレス

    sprintf(buf, "hello!!");           → 出力するデータ

    for(i = 0; i < 10; i++) {          → for文
        *PORT_OUT = buf[i];           → ポートに対し出力を行う
        if(buf[i] == '\0')            → 0x00をエンドコードとする
            break;
    }
}
```

7. 出力結果の I/O タイミング設定ウィンドウは下記ようになります。

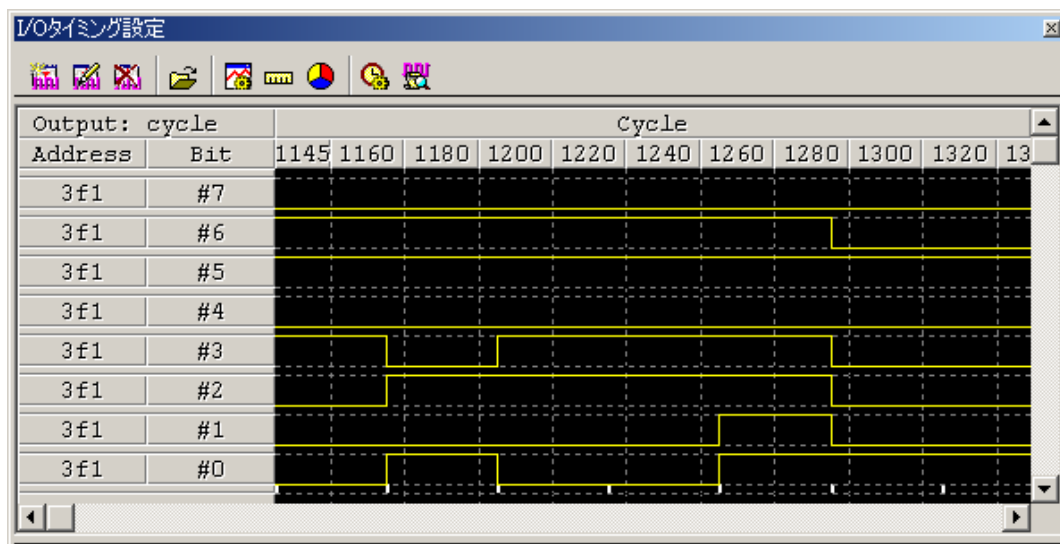


図6.34 I/Oタイミング設定ウィンドウ

6. シミュレータデバッガの活用

8.6. のコードを実行すると 5. で指定した I/O スクリプトファイルは下記になります。

```
; IOSRIPT FILE FOR I/O WINDOW (SET WAITC)
{
waitc 1145
set [0x3f1] = 0x68
waitc 30
set [0x3f1] = 0x65
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6f
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x0
}
```

(2) 出力ポートウィンドウ

1. メニューより [表示->CPU->出力ポート] をクリックし、出力ポートウィンドウを表示します。
2. ポート設定アイコンをクリックするか、右クリックから [ポート設定] をクリックしてください。

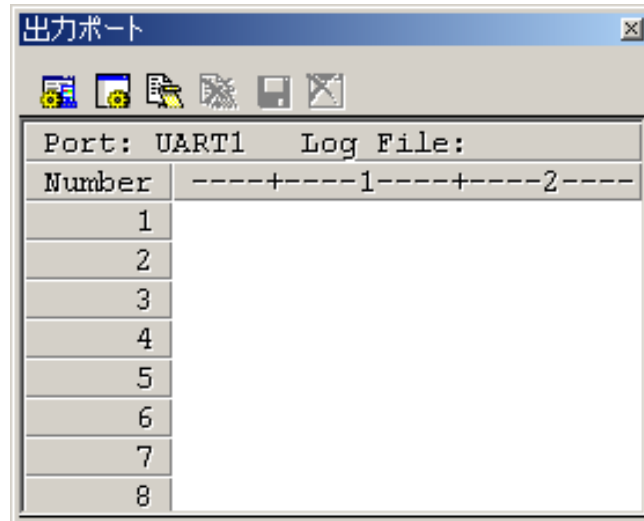


図6.35 出力ポートウィンドウ

3. ポートの設定ダイアログを表示します。
4. アドレスを選択し、ラベルもしくはアドレスを入力します。
5. OK ボタンを押すと設定が完了します。



図6.36 ポート設定

6. シミュレータデバッガの活用

6. プログラム中で指定アドレスへの出力があったときに、ポートに出力された内容が出力ポートウィンドウに出力されます。

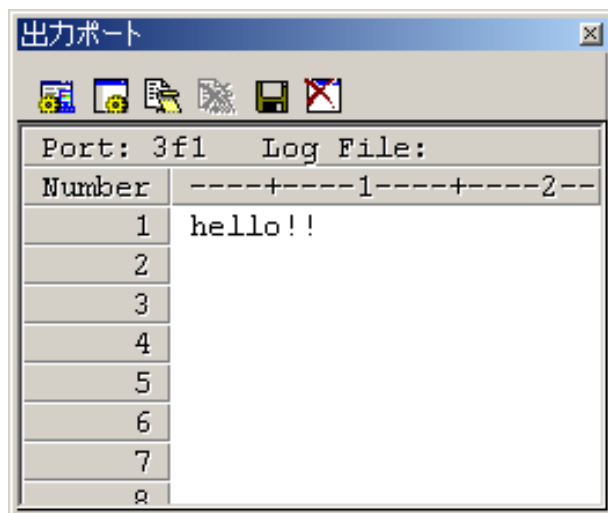


図6.37 出力ポートウィンドウ

7. ログ開始アイコンをクリックすれば、ポートに出力された内容がログファイルとしても出力されます。

6.3 仮想 LED やラベルでメモリ内容を確認する

■説明

仮想 LED やラベルは、メモリ内容を監視し、そのメモリ内容に応じて、リアルタイムに表示色や表示テキストを変えることができます。

■設定方法

1. メニューより[表示->グラフィック->GUI I/O]をクリックし、GUI ウィンドウを表示します。
2. LED 作成アイコンかラベル作成アイコンをクリックするか、右クリックから[LED の作成]か[ラベルの作成]をクリックし、仮想 LED またはラベルを配置します。

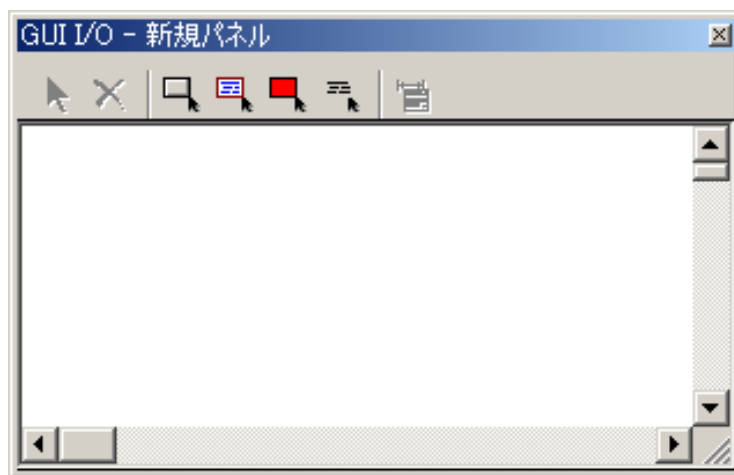


図6.38 GUIウィンドウ

3. 配置したオブジェクトをクリックし、設定画面を表示します。

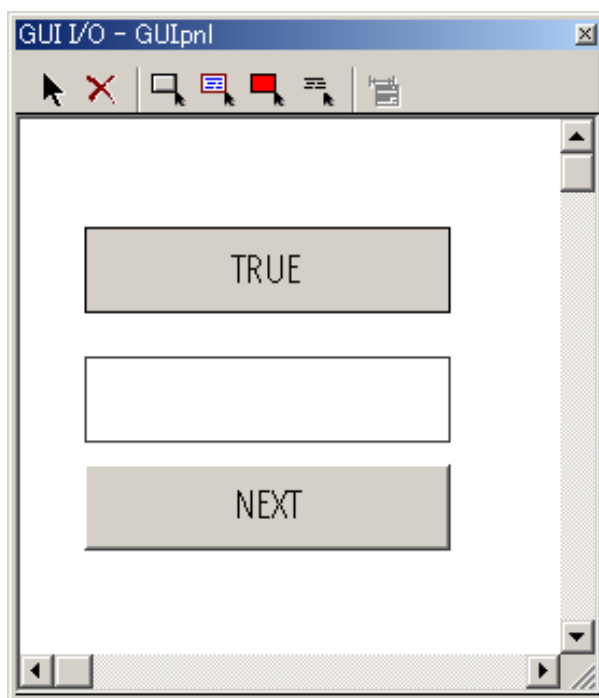


図6.39 GUIウィンドウ（配置後）

6. シミュレータデバッガの活用

4. アドレスを設定し、データの形がビットかデータかを選択します。
5. 仮想 LED の場合は表示色を、ラベルの場合は表示文字列を、それぞれ設定します。
6. ビットのときは正論理（正で表示 1）負論理（負で表示 1）を選択します。



図6.40 仮想LEDの設定



図6.41 ラベルの設定

7. 下記はコード例です。0～200 までの整数の 0 ビット目と 1 ビット目を参照し、仮想 LED とラベルが次々と変わります。

```
void main(void)
{
    for ( i = 0; i < 200; i++){
        TEST_LABEL = i;
        TEST_LED = i;
        while ( NEXT != 1);
        NEXT = 0;
    }
}
```

8. データのときは表示1と表示2に対応するデータを入力します。



図6.42 仮想LEDの設定



図6.43 ラベルの設定

9. 下記のコードを8.の設定で実行すると、7.と同じ動作をします。

```
void main(void)
{
    for ( i = 0; i < 200; i++){
        TEST_LABEL = ((i >> 1) & 1) ? 1 : 2;
        TEST_LED = ( i & 1) ? 1 : 2;
        while ( NEXT != 1);
        NEXT = 0;
    }
}
```

6.4 デバッグ用に printf を利用する

■説明

ソースコード中で printf により出力される内容を出力ポートウィンドウ及びログファイルに出力することができます。

■設定方法

1. メニューより[表示->CPU->出力ポート]をクリックし、出力ポートウィンドウを表示します。
2. ポート設定アイコンをクリックするか、右クリックから[ポート設定]をクリックしてください。

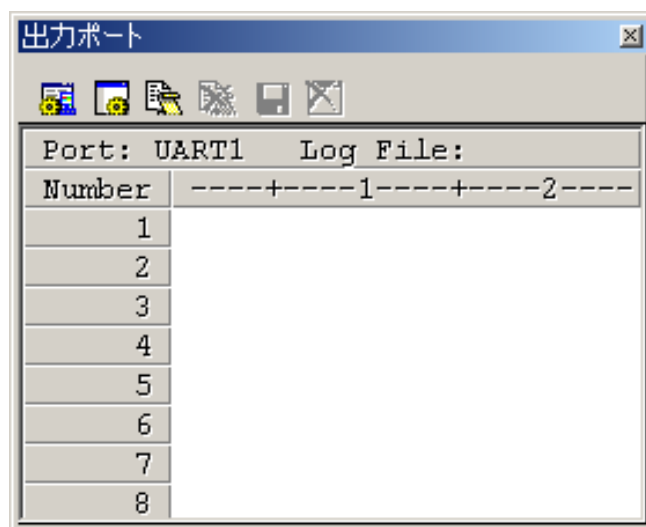


図6.43 出力ポートウィンドウ

3. ポートの設定ダイアログを表示します。
4. printf と UART1 を選択します。
5. OK ボタンを押すと設定が完了します。



図6.44 ポート設定

6. プログラム中で printf の出力があったときに、出力される内容が出力ポートウィンドウに出力されます。下記のコードを実行したときの出力はこのようなになります。

```
for(i = 0; i < 10; i++) {  
    :  
    #ifdef DEBUG  
        printf ( "i=%d\n",i);  
    #endif  
    :  
}
```

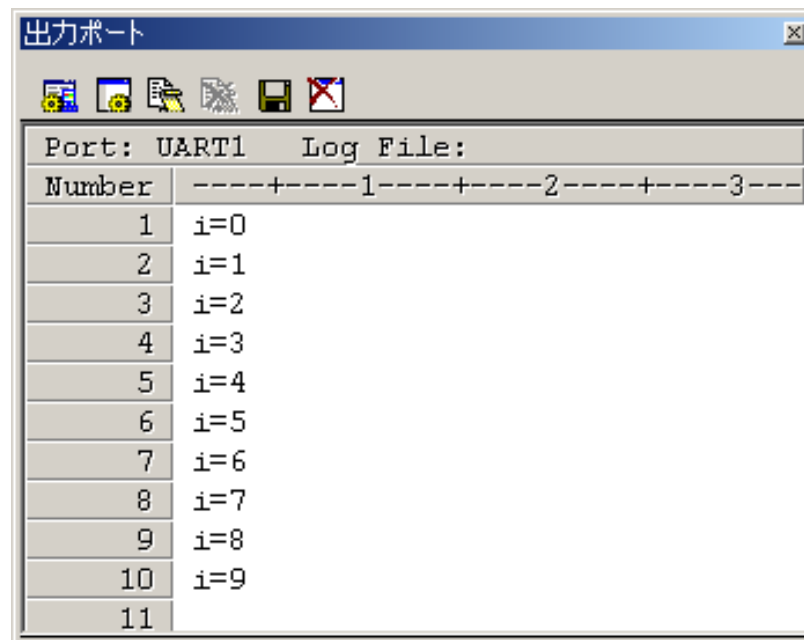


図6.45 出力ポートウィンドウ

7. ログ開始アイコンをクリックすれば、ポートに出力された内容がログファイルとしても出力されます。

6.5 I/O スクリプトの活用

■説明

仮想ポート入力や仮想割り込みの設定を、ファイルにスクリプト形式で記述できます。このスクリプトのことを I/O スクリプトといいます。また、I/O スクリプトを記述したファイルのことを I/O スクリプトファイルといいます。I/O スクリプトファイルは I/O タイミング設定ウィンドウで自動作成できますが、直接編集することによりさらに柔軟な設定を行うことができます。例えば、I/O タイミング設定ウィンドウでは設定できない次のような設定が行えます。

- タイマ割り込みのように周期的な仮想割り込みを発生させる場合、仮想割り込みの発生の繰り返しをwhile文で指定できます。
- 仮想割り込みを発生させる際の割り込み優先順位を、割り込み制御レジスタの割り込み優先レベル選択ビットに設定された優先順位を参照するように指定できます。
- 仮想ポート入力や仮想割り込みの入力/発生条件として、プログラムのフェッチ、メモリへのリード/ライトアクセス、メモリの比較等を組み合わせて利用できます。

■ I/O スクリプトファイルの使用方法

1. あらかじめ、エディタで I/O スクリプトファイルを作成しておきます。このとき、拡張子は” ios” にします。
2. I/O タイミング設定ウィンドウで、読み込みアイコンをクリックするか、右クリックから [読み込み] をクリックすると、I/O スクリプトファイルの読み込みダイアログボックスが開かれます。
3. 読み込む I/O スクリプトファイルを選択します。

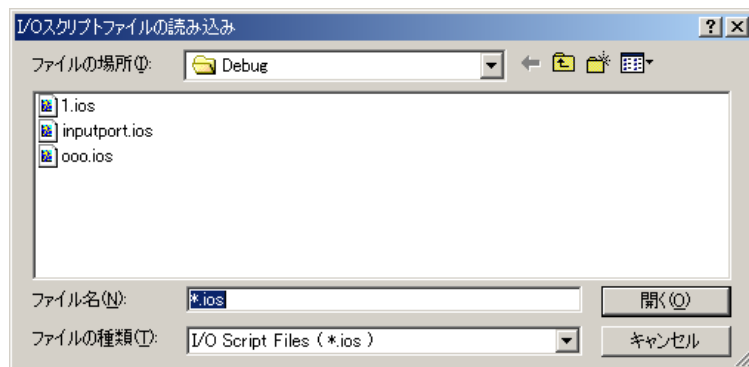


図6.46 読込ダイアログ

■ タイマ A0 のタイマモードの定義例

下記例のプロシジャーは、タイマ A0 のタイマモードの定義例です。

タイマ A0 に指定された分周比（サイクル数）毎に、タイマ A0 割り込みを発生させる例です。タイマ割り込みの優先順位は、割り込み制御レジスタに指定された値を参照します。

; 仮想割り込みの例	
{	→ プロシジャーの開始
while(1){	→ while 文
if([0x380].b & 0x01) {	→ タイマ A0 のカウント開始フラグのチェック
waitc[0x386].w+1	→ タイマ A0 に設定された分周比のサイクル数分 I/O スクリプトの実行をウェイトする
int 21, [0x55] & 0x7	→ タイマ A0 の割り込みを発生 (優先順位は、割り込み制御レジスタを参照)
} else {	
waiti 100	→ 100 命令分 I/O スクリプトの実行をウェイト
}	
}	
}	→ プロシジャーの終了

■ サイクルに同期した仮想ポート入力の定義例

下記例のプロシジャーは、サイクルに同期した仮想ポート入力の定義例です。

プログラムが 5000 サイクル実行された時に仮想ポート入力を行う例です。I/O タイミング設定ウィンドウでは、バイト単位の仮想ポート入力しかできませんが、I/O スクリプトではワード、ロングワード単位での仮想ポート入力が可能です。

; 仮想ポート入力の例	
{	→ プロシジャーの開始
waitc 5000	→ 5000 サイクル分 I/O スクリプトの実行をウェイト
set[0x3e0] = 0x34	→ 0x3e0 番地に 0x34 を入力
waitc 5000	
set[0x3e0].w = 0x4126	→ 0x3e0 番地から 2 バイトデータ 0x4126 を入力
}	→ プロシジャーの終了

7. MISRA C

7.1 MISRA C

7.1.1 MISRA C とは

MISRA C とは Motor Industry Software Reliability Association (MISRA) が 1998 年に発行した C 言語の使用ガイドライン「Guideline for the use of the C language in vehicle based software」、もしくはそのガイドラインで規格化された C 言語記述のルールです。C 言語は優れた言語ですが、いくつかの問題を持っています。MISRA C ガイドラインでは C 言語には 5 種類の問題があるとしています。プログラマによるエラー、言語に対する誤解、意図しないコンパイラの動作、実行時のエラー、コンパイラ自体のエラーです。MISRA C の目的はこれらの問題を回避し C 言語の安全な使用を促進することです。MISRA C には 127 のルールがあり全ルールに合致するようコードの開発を行います。ルールは必要項目と推奨項目の 2 種類に分けられています。全てのルールを守るのは現実的に難しい場合もあるのでしかたのないルール違反に対してはそれを文書化し認める手順もあります。またルール以外にもソフトウェアメトリックスを計測しなければいけないなど諸問題への対応が求められます。

7.1.2 ルールの例

実際に MISRA C のルールをいくつか紹介します。図 7.1 はルール 62「switch 文は全て最後に default 節を置かなければならない」です。これはプログラマによるエラーに分類される問題です。switch 文で”default”ラベルを”defalt”とタイプミスしてもコンパイラはエラーにしません。プログラマ自身が気づかなければデフォルト時に期待する動作は永久に実行されません。ルール 62 を適用することでこの問題を回避できます。

```
例)
switch(x) {
    :
    default:
        err = 1;
        break;
}
```

← スペルミス

図 7.1 ルール 62

図 7.2 はルール 46「式の値は規格が定めるどのような順序で評価されようとも同じでなければならない」です。これは言語に対する誤解に分類される問題です。++i が先に評価されると 2+2 となり i が先に評価されると 2+1 になります。同様に関数の引数の評価順序も未規定なので ++j が先に評価されると f(2,2)となり j が先に評価されると f(1,2)になります。ルール 46 を適用することでこの問題を回避できます。

```
例)
i = 1;
x = ++i + i;      x = 2 + 2?   x = 2 + 1?

j = 1;
func(j, ++j);     func(1, 2)?  func(2, 2)?
```

図 7.2 ルール 46

図 7.3 はルール 38「シフト演算子の右辺の項はゼロ以上、左辺の項のビット幅未満でなければならない」です。これは意図しないコンパイラの動作に分類される問題です。ANSI ではビットシフト演算子のシフト数が負の値の場合およびシフトされるオブジェクトのサイズ以上の場合演算結果は未定義としています。図 7.3 では us をシフトする場合のシフト数は 0 以上 15 以下でなければ結果は未定義となりコンパイラによってその値は異なります。ルール 38 を適用することでこの問題を回避することができます。

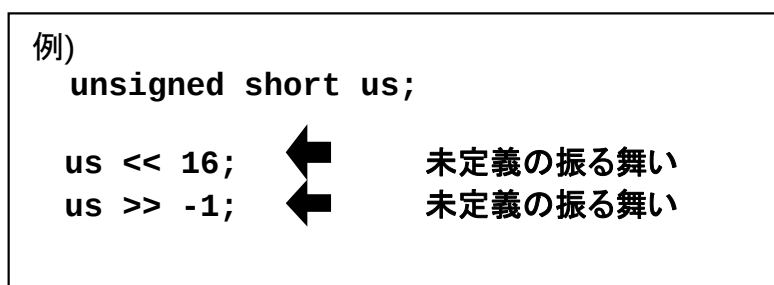


図 7.3 ルール 3 8

図 7.4 はルール 51「符号なし整数定数式の評価は結果の型にはまるべきである」です。これは実行時のエラーに分類される問題です。符号なし整数の演算の結果が論理的に負になる場合は論理的な負の値を期待しているのか符号なしとして演算した結果でよいのかが不明確になり不具合の原因となる恐れがあります。また足し算の結果がオーバーフローを起こして小さな値になることもあります。この問題はルール 51 を適用することで回避可能です。

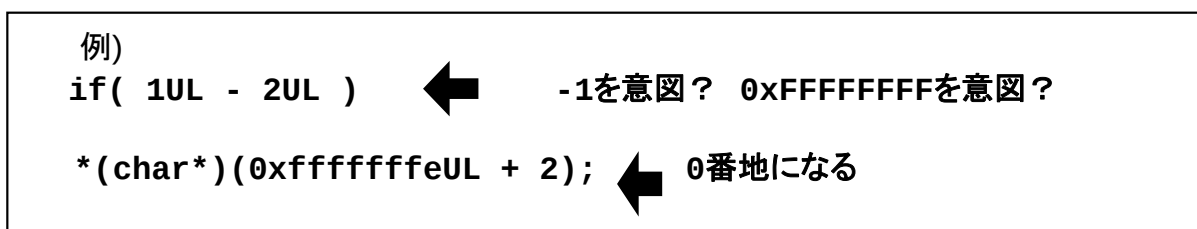


図 7.4 ルール 5 1

7.1.3 合致マトリクス

MISRA C では 1 2 7 あるルールの全てについてソースコードを確認することになっています。さらにそれぞれのルールが守られているかを一目で確認できるようにするために合致マトリクスと呼ばれる表を作成することが要求されています(表 7.1)。全てのルールを目視でチェックするのは大変なので静的チェックツールの使用が推奨されています。MISRA C ガイドラインでも「ルールを遵守するためにツールの使用が非常に重要、ツールの使用があたりまえになることが望ましい。」となっています。ツールではチェックできないルールもあるのでそれらについては目視でレビューを行う必要があります。

表 7.1 合致マトリクス

ルール番号	コンパイラ	ツール 1	ツール 2	レビュー（目視）
1	警告 3 4 7			
2		違反 3 8		
3			警告 9 7	
4				合格
...	...	...	...	...

7.1.4 ルール違反

ルール違反の中には安全であることがわかっているものもあり、そのほうが効果的なものもあります。そのようなルール違反は認められるべきですが安易にルール違反を認めるのも安全性を損ないます。そこで MISRA C ではルール違反を認める手順を定めています。ルール違反に対する正当な理由付けがあること、そのルール違反が安全であると証明できることが必要です。認める違反のそれぞれ全てについてその場所と正当な理由を文書化します。安易に違反を認められないように、専門家のアドバイスの下、それらの文書に組織で権威を持つ人の署名を入れます。一度認められたルール違反と同じパターンのルール違反を「認められたルール違反」と呼び上記の手続きなしで認められたものとして扱ってよいことになっています。しかしそれも定期的に見直す必要があるとしています。

7.1.5 MISRA C 準拠

MISRA C に準拠していることを主張するにはルールに合致したコードを開発することとルール以外の諸問題への対策が必要になります。コードがルールに合致していることを示すには合致マトリクス、認めるルール違反に関する文書、各ルール違反への署名が必要になります。諸問題への対策とは C 言語や使用するツールを使いこなすための技術的訓練を行ったり、コーディングスタイルを規定したり、ツールの選定時に妥当性を確認したり、各種ソフトウェアメトリックスを計測するなどの対応を求められます。またこれらの取り組みが正式に標準化されている必要があります。標準化とは文書化と実践の両方が行われていることを指します。MISRA C の準拠はガイドラインにしたがって開発された個々の製品に対してしか言えず、組織に対して言うことは出来ません。

7.2 SQLint

7.2.1 SQLint とは

SQLint はルネサス製 C コンパイラに、MISRA C ルールに違反していないかを検査する機能を付加する機能追加パッケージです。C ソースコードを静的に検査してルールに違反している個所をレポートします。SQLint はルネサス製品開発環境の中で C コンパイラの一部として動作します。（図 7.5）コンパイル時にオプションを追加するだけで SQLint が起動します。コンパイラが生成するコードに影響をあたえることはありません。SQLint が対応しているルールは表 7.2 のようになっています。

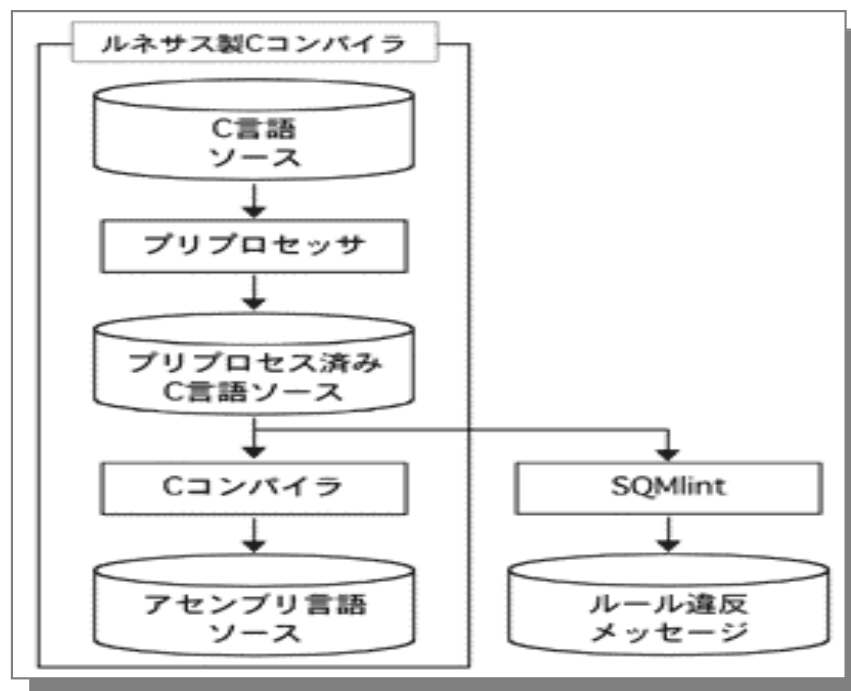


図 7.5 SQMint の位置付け

表 7.2 SQMlint 対応ルール

ルールの番号	可否	ルールの番号	可否	ルールの番号	可否	ルールの番号	可否	ルールの番号	可否	ルールの番号	可否
1	○	26	×	51	○*	76	○	101	○	126	○
2	×	27	×	52	×	77	○	102	○	127	○
3	×	28	○	53	○	78	○	103	○		
4	×	29	○	54	○*	79	○	104	○		
5	○	30	×	55	○	80	○	105	○		
6	×	31	○	56	○	81	×	106	○*		
7	×	32	○	57	○	82	○	107	×		
8	○	33	○	58	○	83	○	108	○		
9	×	34	○	59	○	84	○	109	×		
10	×	35	○	60	○	85	○	110	○		
11	×	36	○	61	○	86	×	111	○		
12	○	37	○	62	○	87	×	112	○		
13	○	38	○	63	○	88	×	113	○		
14	○	39	○	64	○	89	×	114	×		
15	×	40	○	65	○	90	×	115	○		
16	×	41	×	66	×	91	×	116	×		
17	○*	42	○	67	×	92	×	117	×		
18	○	43	○	68	○	93	×	118	○		
19	○	44	○	69	○	94	×	119	○		
20	○	45	○	70	○*	95	×	120	×		
21	○*	46	○*	71	○	96	×	121	○		
22	○*	47	×	72	○*	97	×	122	○		
23	×	48	○	73	○	98	×	123	○		
24	○	49	○	74	○	99	○	124	○		
25	×	50	○	75	○	100	×	125	○*		

○：検査可 ×：検査対象外 ○*：制限付で検査可

表 7.3 SQMlint 対応ルール数

ルール分類	検査可能なルールの数 (SQMlint 対応ルール数/全ルール数)
必要ルール	67/93
推奨ルール	19/34
合計	86/127

7.2.2 使用方法

High-performance Embedded Workshop のコンパイルオプション設定画面でも簡単に S Q Mlint の起動を設定できます。図 7.6 は High-performance Embedded Workshop のオプション指定ダイアログです。Category から MISRA C rule check を選びます。

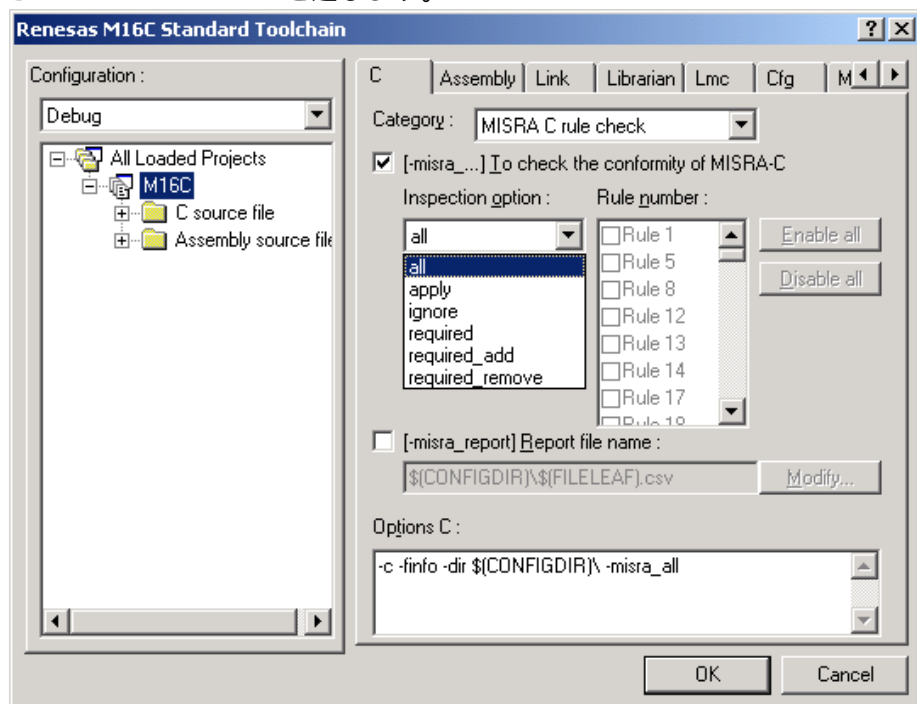


図 7.6 High-performance Embedded Workshop のオプション選択画面

これを選択することでコンパイル時に S Q Mlint が起動されるようになります。各オプションの意味は次のようになります。

- all・・・全てのルールを対象に検査します。
- apply・・・特定のルールについてのみ検査します。
- ignore・・・指定した番号以外のルールを全て検査します。
- require・・・MISRA C ルール中で、“必須”と書かれているルールのみを検査します。
- require_add・・・MISRA C ルール中で、“必須”と書かれているルール全てと、後ろに書いた番号のみを検査します。
- require_remove・・・MISRA C ルール中で、“必須”と書かれているルールのみを検査します。ただし、指定した番号のルールは検査しません。

7.2.3 検査結果の確認方法

検査結果の出力は次の 3 つの形式があります。

(a) 標準エラー出力

High-performance Embedded Workshop 上でコンパイルエラーと同様にメッセージが出力されます。タグジャンプも可能です。コンパイルエラーと同じ操作で手軽にソースコードを修正できます。

(b) CSV形式のファイル

表計算ソフトで読み込み可能な形式のファイルです。そのためレビューなどに使用しやすくなっています。

(c) SQMmerger

図 7.7 のようなソースファイルと検査結果の混合表示を行います。

```
1 : void func(void);
2 : void func(void)
3 : {
4 : LABEL:
[MISRA(55) Complain] label ('LABEL') should not be used
5 :
6 : goto LABEL;
[MISRA(56) Complain] the 'goto' statement shall not be used
7 : }
```

図 7.7 SQMmerger

7.2.4 開発手順

SQMLint を使用した開発手順を図 7.8 に示します。

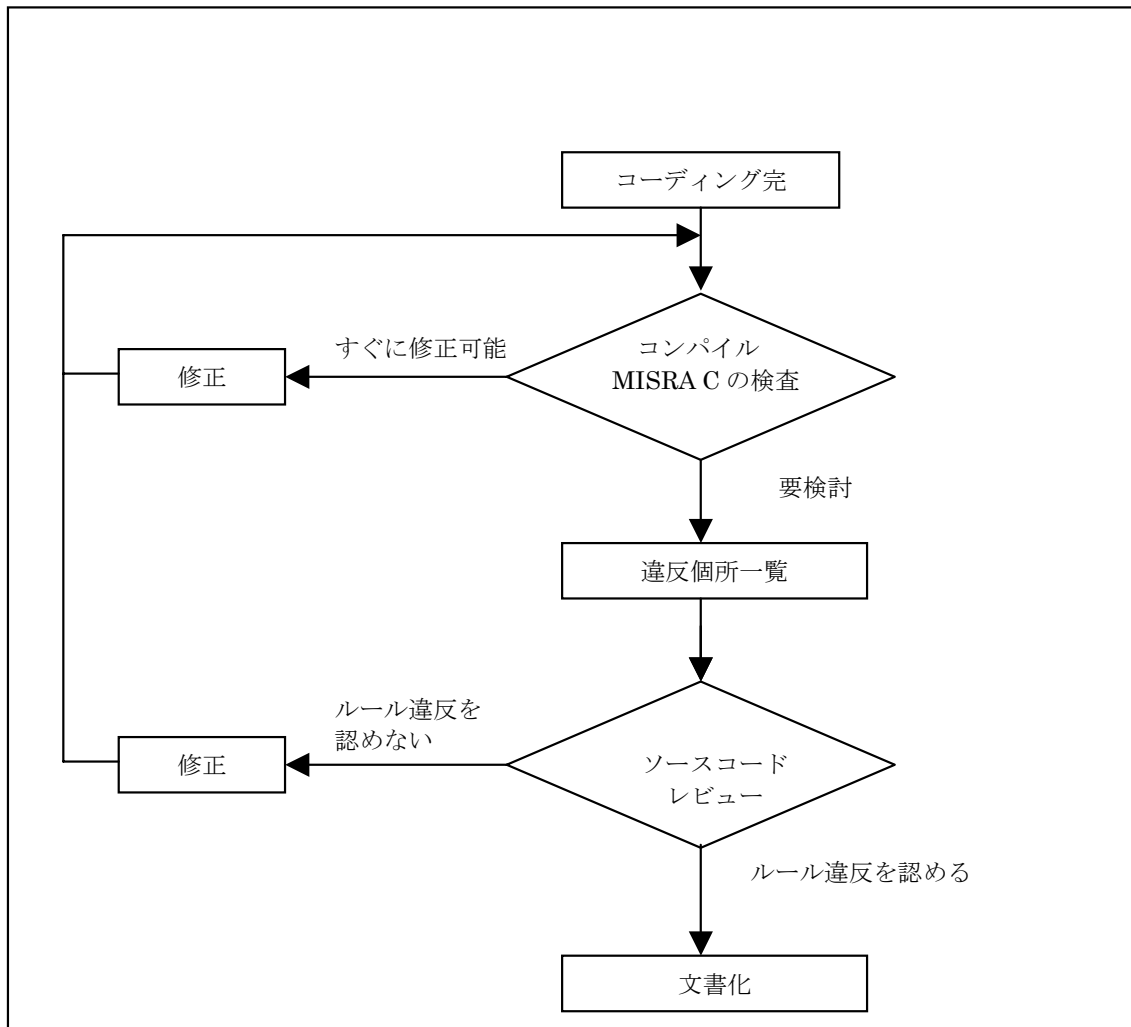


図 7.8 SQMlint を用いた開発手順

- (1) コンパイルエラーを全て取ります。SQMlint は正しいCソースコードを前提としています。
- (2) SQMlint で検出したエラーを調べます。
- (3) 簡単に修正できるものはすぐに修正してしまいます。
- (4) 検討が必要なものはルール違反箇所一覧を作成しレビューを行います。
- (5) レビューの結果ルール違反を認めないものは修正を行います。
- (6) レビューの結果ルール違反を認めるものはそれを文書化し記録として残します。

7.2.5 対応コンパイラ

以下のコンパイラが SQMlint に対応しています。

- M3T-NC30WA V.5.20 Release 1 以上
- M3T-NC308WA V.5.10 Release 1 以上
- M3T-CC32R V.4.10 Release 1 以上

8. Q & A

8. Q&A

本章では、ユーザから多く寄せられた質問についての回答を記載します。

8.1 C コンパイラ (M3T-NC308WA)

8.1.1 ビットフィールド

■ 質問

10 ビット幅、11 ビット幅、3 ビット幅の組み合わせのようなビットフィールドを各フィールドの間に空きビットを作らない（パディングしない）ように宣言するにはどうすればいいですか？

■ 回答

全ての連続させたいフィールドをフィールドのビット数の合計より大きいビット数である同一な型で宣言すると、空きビットを作らず（パディングせず）にビットフィールドを詰めて作成できます。

[プログラム例]

```
typedef struct {  
    unsigned long    BIT0_9    :10;  
    unsigned long    BIT10_20 :11;  
    unsigned long    BIT21_23 : 3;  
} BIT0_23;
```

8.1.2 メモリ管理関数

■ 質問

NCxxWA において、アプリケーションに不要な `MALLOC()`, `MEMCPY()` などの関数がリンクされないようにするにはどうすればいいでしょうか？

■ 回答

`MALLOC()`, `MEMCPY()` 関数は、スタートアッププログラム (`ncrt0.a30,sectxx.inc`) で設定されている `HEAP` 領域を確保するために使用しています。`MALLOC()`, `MEMCPY()` などのメモリ管理関数を使用しない場合は、`ncrt0.a30,sectxx.inc` をアセンブルするときにアセンブラオプション `-D__HEAP__=1` を与えてください。

8.1.3 -ONBSD

■ 質問

コンパイラの最適化オプション"-ONBSD"で最適化が抑止される場合としてどんなものがありますか？また最適化オプション"-ONBSD"で抑止される最適化の内容を細かく設定する事はできるのでしょうか？

■ 回答

最適化が抑止される場合には、次のようなものがあります。

- (1) 共通部分式の括り出し（同じ部分式を何度も計算しないようにする）
- (2) 共通命令ブロックの纏め上げ（同じ並びの命令列への分岐をまとめるなど）
- (3) 連続する領域へのバイト単位の格納をワード単位にする
- (4) 1 バイト定数の連続する push をワード単位にする
- (5) 連続するシフトをまとめる（ $b=a<<2$; $b<=<2$; を $b=a<<4$ にするなど ）
- (6) ビット操作をまとめて OR,AND 演算にする
- (7) for 文の条件判定式をループの末尾に移動し、評価回数を減らす
- (8) return 文への無条件分岐を return に置き換える（NC308 のみ）
- (9) 2 のべき乗での乗除算をシフトに変換する最適化
- (10) auto 変数を自動的にレジスタに割り当てる最適化
- (11) 定数を畳み込む最適化

なお、これらの最適化は、該当個所の前後の記述などにより、必ずしも適用されない場合もあります。

最適化の抑止を細かく設定するようなオプションはありません。

8.1.4 最適化オプションの優先順位

■ 質問

最適化オプションを複数指定した場合、オプションが処理される優先順位はどうなりますか？
また、最適化を行うオプションと最適化を抑止するオプションでは、どちらの優先順位が高いのでしょうか？

■ 回答

最適化に関するオプションを同時に指定された場合に有効となるオプションは以下のとおりです。

- (1) -O1 ~ -O5 のいずれかが同時に指定された場合は、後から指定されたものが有効になります。
- (2) -O[番号]と-OR が同時に指定された場合は、両方ともに有効になります。
- (3) -O[番号]と-OS が同時に指定された場合は、両方ともに有効になります。
-OR と-OS は同時に指定することはできません。

[例：-OR と-O1 を同時に指定した場合]

-OR の最適化は行いますが、-O[番号]で抑止される最適化は行われません。

また最適化を行うオプションと最適化を抑止するオプションでは最適化を抑止するオプションのほうが優先順位が高くなります。最適化オプションと最適化抑止オプションを同時に指定した場合は、指定された最適化抑止オプションに基づいて最適化が抑止されます。

8.1.5 関数のライブラリ追加

■ 質問

関数を新しくコンパイラのライブラリに追加することはできますか？

■ 回答

次の手順で、関数をライブラリに追加できます。ここでは、NC308WA を使用した例を示します。

C ソースファイルをコンパイルしてリロケータブルファイルを生成してください。

[例：new.r30 を生成]

```
>nc308 -c new.c
```

ライブラリアン lb308 を使用してリロケータブルファイルをライブラリに追加してください。

[例：new.r30 ファイルを nc308 のライブラリファイルに追加]

```
>lb308 -a nc308lib.lib new.r30
```

[参考]

lb308 の使用方法については、AS308 ユーザーズマニュアルを参照してください。AS308 ユーザーズマニュアルは、製品 CD-ROM の nc308wa/manual ディレクトリにあります。NC308WA の Windows 版をインストールすると、同時にインストールされますので、Windows のスタートメニューからも参照できます。

8.1.6 const の ROM セクション配置

■ 質問

const 宣言したものは、すべて ROM セクションに配置されを考えていたのですが、以下のような記述をした場合、data セクションに配置されてしまうようです。ROM セクションに配置するにはどうすればよいでしょうか？

```
const S_TBL *sp_tbl[]={
    p00, /* 構造体へのポインタ */
    p01, /* 構造体へのポインタ */
    p02, /* 構造体へのポインタ */
};
```

■ 回答

上記の記述例において、sp_tbl を ROM に配置するには次のように記述してください。

```
S_TBL *const sp_tbl[]={
    p00, /* 構造体へのポインタ */
    p01, /* 構造体へのポインタ */
    p02, /* 構造体へのポインタ */
};
```

また、sp_tbl だけでなく p00, p01, p02 も ROM に配置するには次のように記述してください。

```
const S_TBL *const sp_tbl[]={
    p00, /* 構造体へのポインタ */
    p01, /* 構造体へのポインタ */
    p02, /* 構造体へのポインタ */
};
```

[参考]

ポインタに対して const 宣言する場合、const の位置によって ROM に配置されるものが異なります。

以下の記述では、ROM に配置されるのは、a, b, c になります。

```
const int *i={a,b,c};
```

次の記述では、i が ROM に配置されます。

```
int * const i={a,b,c};
```

8.1.7 引数のレジスタ渡し

■ 質問

関数の引数がレジスタ渡しとなるのはどのような条件のときですか？

また、関数の引数がレジスタ渡しになる場合とスタック渡しになる場合とでサイズ、速度に違いがあるのですか？

■ 回答

次の条件の場合、関数の引数はレジスタ渡しとなります。

(1) 関数のプロトタイプ宣言を行い、関数呼び出し時に引数の型が確定している。

(2) プロトタイプ宣言に可変引数"..."を使用していない。

(3) 関数の引数の型が次の表にあるものと一致している。

[NC30 の場合]

引数	引数の型	使用するレジスタ
第 1 引数	char 型	R1L レジスタ
第 1 引数	int 型 near ポインタ型	R1 レジスタ
第 2 引数	int 型 near ポインタ型	R2 レジスタ

[NC308 の場合]

引数	引数の型	使用するレジスタ
第 1 引数	char 型	R0L レジスタ
第 1 引数	int 型 near ポインタ型	R0 レジスタ

また、レジスタ渡しのほうが、サイズ、速度とも有利になります。

8.1.8 関数引数の渡し方

■ 質問

関数引数の渡し方は、プログラムの移植性に影響がありますか？

■ 回答

関数のプロトタイプ宣言をしていれば、関数引数の渡し方はコンパイラが判断します。よって、プログラムの移植性に影響はありません。

8.1.9 プロトタイプ宣言

■ 質問

プロトタイプ宣言は、関数引数の渡し方を決めるものですか？

■ 回答

プロトタイプ宣言は、関数引数の渡し方を決めるほかに、引数のサイズなども指定します。したがって、プロトタイプ宣言の無い関数呼び出しと、別ファイルに存在する関数実体側の引数の整合性が合わなくなる可能性もあります。そのためプロトタイプ宣言を記述されることをお勧めいたします。

8.1.10 構造体ビットフィールドメンバの配置

■ 質問

C 言語プログラムにおいて、構造体ビットフィールドを定義していますが、コンパイルの結果ビットフィールドメンバの順序が入れ替わります。入れ替わらないようにするにはどうすればいいですか？

■ 回答

構造体のビットフィールドメンバの配置は、以下のような規則で決定します。

- (1) #pragma STRUCT で unpack, arrange を使用しないとき。
- (2) 型が同一のビットフィールドが連続する場合、連続して配置する。
- (3) 型が異なるビットフィールドが出現した場合、以前に型が同一のビットフィールドがあれば、そこに連続して配置する。
- (4) 以前に型が同一のビットフィールドがなければ、次のアドレスから配置する。
- (5) ビットフィールドでないメンバを挟んで、連続して配置しない。

上記の条件にあてはまるメンバを記述順に配置するには、次のいずれかの方法で対処してください。

対処方法 1：型が違うビットフィールド毎に、struct { } で括る。

対処方法 2：連続で配置したいフィールドを全て同じ型で宣言する。

オプションや #pragma STRUCT などに変更することはできません。

以降にメンバの順序が入れ替わる記述例と各対処方法による記述例を示します。

[メンバの順序が入れ替わる記述例]

```
struct tagData {
    unsigned char    c0a : 5; /* (1) */
    unsigned char    c0b : 3; /* (2) */
    unsigned char    c1a : 6; /* (3) */
    unsigned char    c1b : 2; /* (4) */
    unsigned int     i2a : 10; /* (5) */
    unsigned int     i2b : 6; /* (6) */
    unsigned char    c4a : 3; /* (7) */
    unsigned char    c4b : 5; /* (8) */
    unsigned char    c5;      /* (9) */
    unsigned char    c6a : 5; /* (10) */
    unsigned char    c6b : 3; /* (11) */
} s;
```

このような記述の場合、以下のように配置されます。

- (1)(2)(3)(4)は連続に配置されます。
- (5)(6)は型が異なるので、同一な型の(7)(8)が先に配置されます。
- (9)はビットフィールドでないメンバなので、(10)(11)は(7)(8)には続きません。
- (7)(8)の次のアドレスから(5)(6)[型が異なるビットフィールド]が配置されます。
- (5)(6)の次のアドレスに(9)[ビットフィールドでないメンバ]が配置されます。
- (9)の次のアドレスから(10)(11)が配置されます。

8. Q&A

対処方法 1 : 型が違うビットフィールド毎に、struct { } で括る。

```
struct tagData {
    struct {
        unsigned char    c0a : 5; /* (1) */
        unsigned char    c0b : 3; /* (2) */
        unsigned char    c1a : 6; /* (3) */
        unsigned char    c1b : 2; /* (4) */
    } s1;
    struct {
        unsigned int     i2a : 10; /* (5) */
        unsigned int     i2b : 6; /* (6) */
    } s2;
    struct {
        unsigned char    c4a : 3; /* (7) */
        unsigned char    c4b : 5; /* (8) */
    } s3;
    unsigned char    c5;          /* (9) */
    struct {
        unsigned char    c6a : 5; /* (10) */
        unsigned char    c6b : 3; /* (11) */
    } s4;
} s;
```

なお、この構造体を参照している個所を変更する必要があります。

変更前 : s.c0a = 1;

変更後 : s.s1.c0a = 1;

対処方法 2 : 連続で配置したいフィールドを全て同じ型で宣言する。

```
struct tagData {
    unsigned int    c0a : 5; /* (1) */
    unsigned int    c0b : 3; /* (2) */
    unsigned int    c1a : 6; /* (3) */
    unsigned int    c1b : 2; /* (4) */
    unsigned int    i2a : 10; /* (5) */
    unsigned int    i2b : 6; /* (6) */
    unsigned int    c4a : 3; /* (7) */
    unsigned int    c4b : 5; /* (8) */
    unsigned char    c5;      /* (9) */
    unsigned char    c6a : 5; /* (10) */
    unsigned char    c6b : 3; /* (11) */
} s;
```

(1) ~ (8), (10) ~ (11) は連続で配置されます。

この方法は、コードサイズが多少大きくなることがあります。

8.1.11 インクリメント・デクリメント演算子の記述

■ 質問

C 言語で以下(1)のようにプログラミングしました。

生成されたコードを見ると、変数 `x` はインクリメントする前の値で 5 と比較しているようです。変数 `x` をインクリメントした後に 5 と比較するには、以下(2)のように記述する必要がありますか。

- ```
(1) if (x++ == 5){
 aaasub();
 }
(2) x++;
 if (x == 5){
 aaasub();
 }
```
- 

#### ■ 回答

インクリメント演算子 `++` とデクリメント演算子 `--` を記述する位置には、後置と前置の 2 種類があります。

後置：対象となる変数を使用した後にインクリメントもしくはデクリメントします。

前置：対象となる変数を使用する前にインクリメントもしくはデクリメントします。

ご指摘のプログラムでは後置となりますので、インクリメントする前の変数 `x` と 5 と比較することになります。期待される結果にするには前置でインクリメント演算子を記述してください。

```
if (++x == 5){
 aaasub();
}
```

## 8.1.12 外部変数の配置

## ■ 質問

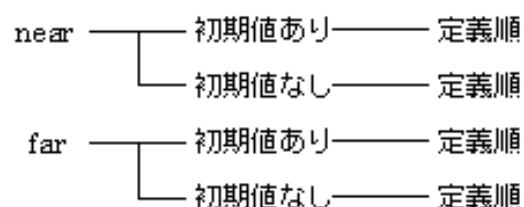
C 言語でプログラムを記述する場合、定義した順序通りに外部変数を配置するにはどうすればいいでしょうか？

## ■ 回答

通常 NCxxWA では、次に示す属性毎にまとめて外部変数を配置します。



初期値ありと初期値なしのデータを区別しないようにすることはできませんが、コマンドオプション `-fno_even` を指定してコンパイルすると、偶数サイズデータと奇数サイズデータの区別をしないで、以下のように外部変数を配置することができます。



### 8.1.13 配列の far 領域配置

#### ■ 質問

NC30WA で、配列の実体を far 領域に配置し、それを参照するポインタを near 領域に配置するにはどのように宣言すればよいでしょうか？

#### ■ 回答

例えば、short 型で 64 バイトの配列を宣言する場合は次のように宣言します。

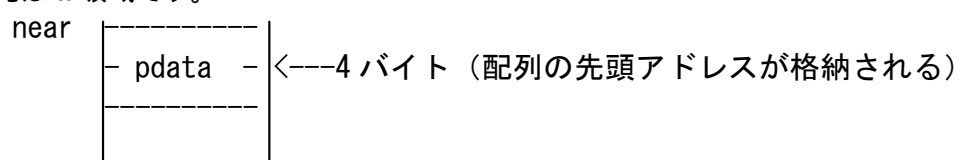
```
/* (1) 配列の実体を宣言 */
```

```
short far data[64];
```

```
/* (2) 配列へのポインタを宣言 */
```

```
short far *pdata;
```

上記(2)のポインタは次のようなイメージで near 領域に配置されます。このポインタが指し示す先は far 領域です。



また、配列へのポインタを配置する領域とその指し示す先の領域は以下のように変更することができます。

```
short * far pdata
```

```
/* (a) pdata は far 領域に、その指し示す先は near 領域に配置
(far を省略した場合は near として扱われます)*/
```

```
short far * far pdata
```

```
/* (b) pdata は far 領域に、その指し示す先も far 領域に配置 */
```

```
short far * pdata
```

```
/* (c) pdata は near 領域に、その指し示す先は far 領域に配置 */
```

```
short * pdata
```

```
/* (d) pdata は near 領域に、その指し示す先も near 領域に配置 */
```

### 8.1.14 関数の固定アドレス配置

---

#### ■ 質問

C 言語で記述した各関数を絶対アドレス（固定アドレス）にしたいのですがどのようにすればいいのでしょうか？

---

#### ■ 回答

NC30WA は、関数を program セクションに配置します。したがって、関数を絶対アドレスに配置するためには、program セクションの配置アドレスを指定してください。  
program セクションの配置アドレスは、sect30.inc ファイル中の ".section program" の行の後で、疑似命令.org を使用して指定してください。

[例：関数を 10000H 番地に配置する場合]

sect30.inc ファイルで以下のように指定してください。

```
.section program
.org 10000H
```

関数毎に別のアドレスに配置する場合は、対象となる関数に対して "#pragma SECTION" を用いて program セクションとは別のセクション名を作成して、そのセクションの配置アドレスを指定するように記述してください。

[例：関数 func()を"program1"というセクションに配置し、そのセクションを 20000H に配置する場合]

(1) 以下のように関数を配置するセクション名を指定できます。

```
#pragma SECTION program program1
func()
{
}
```

(2) sect30.inc ファイルでセクションの配置アドレスを以下のように指定してください。

```
.section program1
.org 20000H
```

### 8.1.15 #pragma ADDRESS を使った絶対アドレス指定

---

#### ■ 質問

"#pragma ADDRESS"を使って、次の記述と同じ絶対アドレス指定をするにはどうすればいいでしょうか？

[絶対アドレス指定例]

```
#define AAA (*(volatile unsigned char *)0x000406)
```

---

#### ■ 回答

"#pragma ADDRESS"を使って上記と同じ 0x0406 を指定するには、次のように記述してください。なお、#pragma ADDRESS で宣言された変数は、volatile 属性となります。

```
#pragma ADDRESS AAA 000406h
unsigned char AAA;
```

### 8.1.16 #define での文字列定義

---

#### ■ 質問

プリプロセスコマンド"#define"で文字列に数値を定義していますが、定義した文字列を使った式の結果が期待する値にならないのはなぜですか？

以下の例で、cul の値が期待値 0x04AAAA になりません。

[プログラム例]

```
#define V1 0x040000
#define V2 0x0AAAA + V1
```

```
cul = (WORD *)V2
```

---

#### ■ 回答

上記プログラム例では、#defineを展開した結果の式は以下のようになります。

#defineを展開する時は単に文字列置換になります。

```
cul = (WORD *)0x0AAAA+ 0x040000;
```

この式では、(WORD \*)のキャストは 0x0AAAA にのみかかり、これに対して 0x040000 が加算されます。実際には int へのポインタに加算しているので 0x80000 です。

期待する値を得るためには、以下のように#define で定義する式に対して () を付けてください。

[プログラム例]

```
#define V2 (0x0AAAA + V1)
```

---

### 8.1.17 ビットフィールドメンバの型

---

#### ■ 質問

以下のプログラムを実行すると、long 型変数"l"には-1 が代入されるものと考えますが、結果は 0xF(15)が代入されてしまいます。どのようにすれば、結果が-1 になるのですか。

```
void main(void)
{
 struct
 {
 short r : 10 ;
 short i : 4 ;
 short s : 2 ;
 } buf ;
 long l ;

 buf.i = -1 ;
 l = (long)buf.i ;
}
```

---

#### ■ 回答

ビットフィールドメンバの型はunsignedとして処理します。そのため型の大きい変数に格納 (転送) する場合には、ゼロ拡張されます。

これを符号拡張として扱う場合は、ビットフィールドメンバの型宣言時に明示的に signed で宣言するようにしてください。

[例]

```
struct
{
 signed short r : 10 ;
 signed short i : 4 ;
 signed short s : 2 ;
} buf ;
```

### 8.1.18 変数の 2 重定義

---

#### ■ 質問

C 言語のプログラムにおいて、同一ファイル内で、同じ型の同じ名前の変数を宣言した場合、エラーになりますか？

---

#### ■ 回答

同一ファイル内で同一名の変数を宣言した場合、エラー（もしくはウォーニング）は発生しません。

これは次の ANSI 仕様を採用しているためです。

- 同一ファイル内において名前に割り当てる型が異なる限り、その名前に対する外部宣言はいくつあっても良い。

例 1 (エラー発生しない)

```
int i;
int i=1;
```

```
main()
{
}
```

例 2 (2 重定義エラー発生)

```
int i;
char i;
```

```
main()
{
}
```

### 8.1.19 関数のプロトタイプ宣言

---

#### ■ 質問

リンク時に'関数名' value is undefined というエラーが出力されます。関数の実体は存在します。なぜでしょうか？

---

#### ■ 回答

これは、関数の呼び出し側、実体側において一方が\_関数名、もう一方が\$関数名になっているため発生しています。この原因は、関数のプロトタイプ宣言が記述されていないか、両関数の引数等の型が合っていないことが考えられます。もう一度プロトタイプ宣言を確認してください。

### 8.1.20 extern 宣言なし関数の外部参照

---

■ 質問

extern 宣言をしないで関数を外部参照した場合、コンパイラでエラーとならないのでしょうか？

---

■ 回答

extern 宣言 (プロトタイプ宣言) が無い場合は、ANSI 仕様によりその関数は int を返す関数であると解釈します。したがって、int 型で外部参照され、エラーとなりません。

extern 宣言 (プロトタイプ宣言) が無い関数を検出するには、コンパイル時に -Wno\_prototype(-WNP) もしくは -Wall を指定してください。プロトタイプ宣言の無い関数に対してウォーニングを出力します。

### 8.1.21 最適化によるコード削除

---

#### ■ 質問

ポートの読み込み等のために記述した式が最適化オプションを使用した場合にコードが出力されません。

---

#### ■ 回答

これは、最適化により意味の無いコードとして削除されています。ご質問のように読み込みに意味を持つような場合は、その変数の宣言時に `volatile` で宣言してください。

### 8.1.22 ビットアクセスの纏め上げ

---

#### ■ 質問

連続してビットアクセスを行うと1つにまとめられる。これでは、ビットアクセスの間に割込みが発生した場合、不正な動作を行ってしまう場合がある。どうしたらいいですか？

---

#### ■ 回答

本処理は、最適化処理により連続するビットの演算を1つに纏め上げています。ご質問のような場合には、コンパイル時のオプション-ONB(-Ono\_bit)を指定することにより本最適化を抑止することができます。

### 8.1.23 ライブラリ関数の ROM アドレス配置

#### ■ 質問

ライブラリ関数をリンク時に ROM アドレスに配置するように指定したいのですが、どうすればいいでしょうか？また、リンク時にセクション配置指定ができるようにするためのライブラリ関数用のセクションはありませんか。

#### ■ 回答

ライブラリ関数用のセクションはありません。

任意のライブラリ関数のセクションを変更するには、目的のライブラリ関数のソースプログラムを取り出して、拡張機能 `#pragma SECTION` でセクション名を指定してください。NC30WA、NC308WA の製品には、各ライブラリのソースを添付しています。

[`#pragma SECTION` の記述方法]

取り出したライブラリ関数の先頭に、次の行を追加してください。

```
#pragma SECTION program library
```

ライブラリ関数が配置されるセクション名は `library` になります。新しいセクション名は任意の名前で指定できます。

その後、通常関数と同様にコンパイル、アセンブルした後、`library` セクションを意図されるアドレスに配置するようにしてリンクしてください。

(注意)

アセンブラで記述されているライブラリに関しては、`#pragma SECTION` では、セクション名を変更できません。

直接 `.section program` を変更してください。

(参考)

ライブラリ関数のソースは、NC30WA(NC308WA)をインストールしたディレクトリ下の `src30¥lib(src308¥lib)` に格納されています。

(ただし、インストールの際、ライブラリソースのインストールをチェックされている場合です。)

### 8.1.24 負の整数演算の処理仕様

---

#### ■ 質問

以下の演算の処理仕様を教えてください。

- (1) 負の整数の除算 "/" の、結果の切り捨て方向
  - (2) 負の整数の剰余算 "%" の、結果の符号
  - (3) 負の整数の右シフト ">>" の動作が、算術シフト (MSB に符号ビットが入る) または論理シフト (MSB に 0 が入る) のいずれか
- 

#### ■ 回答

- (1) 負の整数の除算 "/" 結果は、0 (ゼロ) の方向に切り捨てられます。

| 式               | 演算結果 |
|-----------------|------|
| $(-10) / 3$     | -3   |
| $(-10) \% 3$    | -1   |
| $10 / (-3)$     | -3   |
| $10 \% (-3)$    | 1    |
| $(-10) / (-3)$  | 3    |
| $(-10) \% (-3)$ | -1   |

- (2) 負の整数の剰余算 "%" 結果符号は、被除数の符号に合わせます。

| 式               | 演算結果 |
|-----------------|------|
| $(-10) \% 3$    | -1   |
| $10 \% (-3)$    | 1    |
| $(-10) \% (-3)$ | -1   |

(注意) オプション `-fround_under_div(-fRUD)` を指定した場合、除数の符号に合わせます。

- (3) 負の整数の右シフト ">>" は、算術シフトです。MSB には、符号ビットが入ります。

## 8.1.25 int 型のサイズ

### ■ 質問

プログラム例で、「`aaa = (signed long)( BBB << CCC );`」式の右边が `0x0000f000` になるのはなぜですか？VISUAL C++ では、`0x0fff000` になっています。

| プログラム例                                                                                                                                                                 | 生成アセンブリコード                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <pre>#define BBB      (unsigned short)0xffff #define CCC      (unsigned char)12  signed long aaa;  test( void ) {     aaa = (signed long)( BBB &lt;&lt; CCC ); }</pre> | <pre>_test:     mov. w     #0f000H, _aaa     mov. w     #00000H, _aaa+2     rts</pre> |

### ■ 回答

コンパイラは、プログラム例の式を以下の順序で処理します。

- (1) BBB を int に型変換  
(元が int で表現できる場合は、signed int)  
(元が int で表現できない場合は、unsigned int(以上))
- (2) CCC でシフト
- (3) signed long に型変換して aaa に格納

NC30,NC308 では、int 型変数を 16 ビットで扱います。したがって、式 `aaa = (signed long)( BBB << CCC );` を次のように処理します。

- (1) `0xffff -> 0xffff` ( int で表現できないので unsigned int へ型変換します )
- (2) `0xffff<<12 -> 0xf000`
- (3) `0xf000` は符号無しですから ( signed long ) にキャストしても `0x0000f000` となります。 -> aaa

結果は、`0x0000f000` となります。

NC30,NC308 で式 `aaa = (signed long)( BBB << CCC );` の結果を `0x0fff000` にするには、シフトした結果を signed long にキャストするのでなく、変数 BBB をキャストするように記述してください。

```
#define BBB (unsigned short)0xffff
#define CCC (unsigned char)12
```

```
signed long aaa;
```

```
test(void)
{
 aaa = (signed long)BBB<<CCC;
```

```
}
```

上記プログラムをコンパイルした結果、次のアセンブリコードが生成されます。

```
;## # C_SRC : aaa = (signed long)BBB << CCC;
 mov.w #0f000H,_aaa
 mov.w #00fffH,_aaa+2
```

(参考)

VISUAL C++ は int 型を 32 ビットで扱います。

したがって、VISUAL C++ では、式 `aaa = (signed long)( BBB << CCC );` を次のように処理します。

- (1) `0xffff -> 0x0000ffff`
- (2) `0x0000ffff <<12 -> 0xffff000`
- (3) `0xffff000 ->aaa`

演算結果は、`0xffff000` です。

## 8.1.26 enter 命令の制御

### ■ 質問

NC308WA V.3.00 Release 1 では、関数先頭で enter 命令を出力していましたが、V.3.10 Release 2 では出力されないことがあるため、インラインアセンブリ記述でスタック引数が正しく参照できません。enter 命令の生成を制御できますか？

### ■ 回答

NC308WA V.3.10 では、不要な enter 命令を出さないよう最適化を強化していますが、インラインアセンブリ記述中の処理内容は考慮されないため、引数や自動変数が #pragma ASM ~ #pragma ENDASM、asm() 以外で一度も参照されていない場合、enter 命令は不要と判断し出力しません。

NC308WA では、インラインアセンブリで C 言語の変数を参照する方法として、asm("... \$\$ (又は \$@) ...", 変数名); の記述方法をサポートしています。

この方法では、変数の使用有無がコンパイラから正しく認識できますので、必要であれば enter 命令が生成されます。これ以外の方法で C 言語の変数を参照すると、期待する動作をしなかったり、たまたま動作していてもバージョン間の互換性がなくなります。そのような方法で変数や引数を参照したり、関数の戻り値を設定したりしないでください。

推奨する記述方法を以下に示します。

[例 1] -----

```
void memset(char *p, char c, unsigned short n)
{
 asm(" PUSHM A1,R0,R3 "); // 使用するレジスタは退避が必要
 asm(" MOV.L $$[FB],A1 ", d); // 引数 d を A1 に転送
 asm(" MOV.B $$[FB],R0L ", c); // 引数 c を R0L に転送
 asm(" MOV.W $$[FB],R3 ", n); // 引数 n を R3 に転送
 asm(" SSTR.B ");
 asm(" POPM A1,R0,R3 "); // 使用したレジスタを復旧
}
```

[例 2] -----

```
void memset(char *p, char c, unsigned short n)
{
 asm(" PUSHM A1,R0,R3 "); // 使用するレジスタは退避が必要
 asm(" MOV.L $@,A1 ", d); // 引数 d を A1 に転送
 asm(" MOV.B $@,R0L ", c); // 引数 c を R0L に転送
 asm(" MOV.W $@,R3 ", n); // 引数 n を R3 に転送
 asm(" SSTR.B ");
 asm(" POPM A1,R0,R3 "); // 使用したレジスタを復旧
}
```

\$\$ は、自動変数の場合、FB レジスタ値からのオフセットに置換されます。外部変数の場合は

シンボルと、レジスタ変数の場合はレジスタ名と置換されます。

`$@` は、自動変数、外部変数、レジスタ変数を示すオペランドに置換されます。(`$@` に対応する変数の種別は、コンパイラが自動的に判断します。)

最適化によって関数引数や自動変数がレジスタ変数に変更されることがありますが、`asm("$ $ 又は $@",変数)` で記述されている変数は対象外になります。従って、例えば自動変数のFBからのオフセットで書いたつもりの `"$ $"` がレジスタ変数になってレジスタ名に展開されてしまうようなことはありません。

## 8.1.27 浮動小数点ライブラリの性能

### ■ 質問

浮動小数点ライブラリの性能を教えてください。

### ■ 回答

浮動小数点ライブラリの性能は以下のようになります。

測定条件

( 1 ) 使用コンパイラ : M3T-NC30WA V.5.30 Release02

( 2 ) 使用エミュレータ : M16C/Tiny シリーズ用コンパクトエミュレータ(M30290T2-CPE)  
クロック 2 0 M H z 高速モード

測定結果

#### ・ 四則演算

|                 | float        | double        |
|-----------------|--------------|---------------|
| 加算 ( 7.1+2.9 )  | 33 $\mu$ sec | 37 $\mu$ sec  |
| 減算 ( 7.9-2.9 )  | 35 $\mu$ sec | 40 $\mu$ sec  |
| 除算 ( 3.5/0.5 )  | 79 $\mu$ sec | 164 $\mu$ sec |
| 乗算 ( 7.5*10.2 ) | 22 $\mu$ sec | 37 $\mu$ sec  |

#### ・ 数学関数

|                |             |
|----------------|-------------|
| acos(0.5)      | 9.718 msec  |
| asin(0.5)      | 10.682 msec |
| atan(0.5)      | 7.984 msec  |
| atan2(1.5,0.5) | 13.119 msec |
| ceil(10.8)     | 0.104 msec  |
| cos(0.5)       | 2.775 msec  |
| cosh(0.5)      | 1.934 msec  |
| exp(10.4)      | 1.533 msec  |
| fabs(x)        | 0.010 msec  |
| fmod(10.5,1.5) | 0.218 msec  |
| ldexp(10.4,3)  | 0.006 msec  |
| log(2.4)       | 4.587 msec  |
| log10(100.3)   | 5.295 msec  |
| pow(10.3,4.5)  | 7.061 msec  |
| sin(0.4)       | 3.047 msec  |
| sinh(0.6)      | 1.938 msec  |
| sqrt(100.1)    | 2.112 msec  |
| tan(0.5)       | 2.717 msec  |
| tanh(0.9)      | 1.966 msec  |

## 8.2 リンカ

### 8.2.1 ln308、ln30 の-LOC オプション

---

#### ■ 質問

ln308 又は ln30 の-LOC オプションはどのような場面で利用するのですか？

---

#### ■ 回答

プログラムを RAM 上で動作させるようなアプリケーションに利用できます。その場合、RAM 上で動作させるプログラムの RAM アドレスを-ORDER オプションで定義します。そして、RAM に転送するプログラムを登録する ROM アドレスを-LOC オプションで定義します。

なお、-LOC オプションの機能は定義されたプログラムを所定のアドレスに登録するだけです。そのプログラムを実行時のアドレス領域に転送する機能はありません。

## 8.2.2 リンク時のウォーニング

---

### ■ 質問

リンク時にウォーニング "16-bits unsigned value is out of range 0 -- 65535. address='xxxx'" が発生します。RAM サイズを減らすとウォーニングは発生しません。何が問題なのでしょう？

---

### ■ 回答

ビット命令が届かない領域 (0H ~ 1FFFH 以外) に対してビット命令が出力されると、リンク時に上記ウォーニングが発生します。RAM サイズを減らすとウォーニングが発生しないのは、0H ~ 1FFFH 以外の領域にある変数がRAM サイズが減ったことにより 0H ~ 1FFFH 内に収まり、ビット命令が届くようになったためです。変数が配置されるアドレスはコンパイル時には判断できません (リンク時に決定します)。

以下のとおりビット命令を出力する指定をはずしてください。

- (1) コンパイル時に `-fbit` (もしくは `-fB`) を指定している場合は、指定をはずす。
- (2) 該当する変数が `#pragma BIT` で指定されている場合は、指定をはずす。

### 8.2.3 開始アドレスの変更

---

#### ■ 質問

開始アドレスを 0xFC0000 で記述したプログラムを開始アドレスを 0x000000 のモトローラ S フォーマットに変換したいのですが？

また、次のように lmc308 を実行したのですが、開始アドレスが変更されないのはなぜですか？

```
>lmc308 -E 00 test.x30
```

#### ■ 回答

lmc308 は、「開始アドレス」を変更する機能はありません。また-E コマンドオプションは、「実行開始アドレス」を登録するためのオプションで、実行開始アドレスがモトローラ S フォーマットの最終行"S8" レコードの 5～8 桁に登録されます。しかし、このオプションを使用しても開始アドレスを変更することはできません。

## 8.3 Stk Viewer

### 8.3.1 Stk Viewer のスタックサイズ

---

#### ■ 質問

Stk Viewer で表示されるスタックサイズを確保したが、オーバーフローが発生しました。Stk Viewer で表示されるスタックサイズだけでは足りないのですか？

---

#### ■ 回答

Stk Viewer が表示するスタックサイズは参考値です。

Stk Viewer が呼び出す Stk は、コンパイラが出力するアブソリュートモジュールファイルの中にあるスタック情報をもとにスタックサイズを算出していますが、このサイズは理論上のものであり、実際にプログラムをトレースして導きだされたサイズではありません。

また、Stk は割り込み関数を考慮していません。

割り込み関数を考慮したスタックサイズを算出する場合は、Stk Viewer が表示している割り込みが発生する関数またはアセンブラルーチンのスタックサイズに割り込み関数のスタックサイズを加算してください。

## 8.4 SQMLint

### 8.4.1 検査ルールを選択

---

#### ■ 質問

検査可能なルールのうち、必要なルールのみを選択して検査できますか？

---

#### ■ 回答

SQMLint で検査可能なルールのうち、特定のルールを選んで検査することができます。以下の選択方法があります。

- (1) SQMLint で検査可能なルール全てを検査する。
- (2) SQMLint で検査可能なルールの中で、MISRA C で「必須」とされている項目のみを検査する。
- (3) SQMLint で検査可能なルールの中で、「必須」項目すべてと、「推奨」項目のなかで番号指定した項目のみ検査する。
- (4) SQMLint で検査可能なルールの中で、番号を指定した項目のみ検査する。

SQMLint で検査可能なルールとツール番号については、7 章「表 7 . 2 MISRA C 対応ルール」を参照してください。

## 8.4.2 レポートファイルの出力

### ■ 質問

複数の C 言語ソースファイルをコンパイルすると同時に、SQMlint でチェックした結果をレポートファイルに出力したいのですが、ソースファイル毎にレポートファイルを分けて出力するにはどうすればいいでしょうか？

### ■ 回答

High-performance Embedded Workshop または TM の統合開発環境をご使用ください。

- High-performance Embedded Workshop をご使用の場合

[M3T-NC30WA, M3T-NC308WA]

- (1) オプション - Renesas M16C Standard Tool Chain メニューを選択する。
- (2) 表示されたダイアログで C タブの category に MISRA C Rule Check を選択する。
- (3) [-r] Report file name:に以下を設定する。

\$(CONFIGDIR)¥\$(PROJECTNAME).csv

[M3T-CC32R]

- (1) オプション - Renesas M32R Standard Tool Chain メニューを選択する。
- (2) 表示されたダイアログで C タブの category の MISRA C Rule Check を選択する。
- (3) [-misra\_report] Report file name:に以下を設定する。

\$(CONFIGDIR)¥\$(PROJECTNAME).csv

- TM をご使用の場合

[M3T-NC30WA, M3T-NC308WA]

- (1) プロジェクト - オプションブラウザメニューを選択する。
- (2) CFLAGS を選択し[編集・・・]をクリックする。
- (3) オプション変更のカテゴリに「MISRA-C チェックオプション」を選択する。
- (4) -sqmlint オプションをチェックし、パラメータ文字列に以下を設定する。

-misra all -r \$\*.csv

[M3T-CC32R]

- (1) プロジェクト - オプションブラウザメニューを選択する。
- (2) CFLAGS を選択し[編集・・・]をクリックする。
- (3) オプション変更のカテゴリに「MISRA-C チェックオプション」を選択する。
- (4) -misra\_report オプションをチェックし、パラメータ文字列に以下を設定する。

\$\*.csv

### [参考]

コマンドラインからコマンドを入力する場合には、コンパイルするファイルごとにレポートファイル名を指定するようにしてください。

```

>nc308 -c test.c -sq -sqmlint "-misra all -r test.csv"

```

つまり、一度に 1 つの C ソースファイルをコンパイルするようにしてください。  
つぎのようにコマンドラインで実行した場合、test2.c に対するレポート結果のみが report.csv ファイルに保存されます。

```

>nc308 test1.c test2.c -sq -sqmlint "-misra all -r report.csv"

```

### 8.4.3 レポートメッセージ ( 1 )

---

#### ■ 質問

プロトタイプ宣言の引数に enum 型を記述して、かつその関数呼び出し時に実引数として列挙子を記述すると、同じ型であるにもかかわらずレポートメッセージが出力されます。なぜこのようなメッセージが出力されるのでしょうか？

C ソース例：

```

enum E { A, B, C };
void func1(enum E);

void func2()
{
 func1(A);
}

```

出力メッセージ：

Rule 77 (Complaining) "argument type shall be compatible with prototype, the 1st argument"

---

#### ■ 回答

列挙子の型は int 型です。

SQMLint は仮引数と対応する実引数を厳密に比較するため、"enum E"型と"int"型は異なるとみなしてレポートメッセージを出力しています。

この場合、レポートメッセージは無視してください。

### 8.4.4 レポートメッセージ ( 2 )

---

#### ■ 質問

三項演算子の ":" の左右に列挙子を記述して、かつその戻り値を列挙変数に代入すると、同じ型であるにもかかわらずレポートメッセージが出力されます。なぜこのようなメッセージが出力されるのでしょうか？

C ソース例：

```

enum E { A, B, C };

void func(int i)
{
 enum E e;
 e = (i == 0) ? A : B;
}

```

出力メッセージ：

```
Rule 43 (Complaining) "information loss conversion (from 'signed int' to 'enum E') in assignment
operation"
Rule 29 (Complaining) "enum type object to which has not been assigned own enumerator"
```

---

#### ■ 回答

三項演算子の ":" の左右に列挙子を記述すると、三項演算子は `int` 型の値を返します。このため列挙変数に `int` 型定数を代入することと同じ意味になり、レポートメッセージを出力します。この場合、レポートメッセージは無視してください。

## 8.5 High-performance Embedded Workshop

### 8.5.1 ファイルのリンク順序

---

#### ■ 質問

High-performance Embedded Workshop でプロジェクトを作成したのですが、ファイルのリンク順序がファイル名のアルファベット順になってしまいます。スタートアップファイル(ncrt0.r30)を最初にリンクするにはどうすればよいでしょうか？

---

#### ■ 回答

以下の手順で設定してください。

##### ● M3T-NC30WA V.5.20 R1 の場合

- (1) メニュー「オプション」->"Renesas M16C Standard Toolchain..." をクリックする。  
("Renesas M16C Standard Toolchain"ダイアログが開きます。)
- (2) 「link」タブをクリックする。
- (3) "Category:" から、"Input" を選択する。
- (4) "Show entries for:" から、"Relocatable files" を選択する。
- (5) "Add" ボタンをクリックする。  
("Add Relocatable files" ダイアログが開きます。)
- (6) "Relative to:" から、"Configuration directory" を選択する。
- (7) "File path:" に "ncrt0.r30" を入力する。
- (8) "Add Relocatable files" ダイアログの[OK]をクリックする。
- (9) "Renesas M16C Standard Toolchain" ダイアログの[OK]をクリックする。

これで、ncrt0.r30 を最初にリンクすることができます。

● M3T-NC30WA V.5.30 R1、M3T-NC308WA V.5.20 R1、M3T-NC8C V.5.30 R1 の場合  
プロジェクトワークスペースを新規に作成した場合は、スタートアッププログラム名がncrt0.a30 の場合のみ、自動的に先頭にリンクされます。  
自動先頭リンクは、以下のオプションになります。

- (1) メニュー「オプション」->"Renesas M16C Standard Toolchain..." (\*) をクリックする。  
("Renesas M16C Standard Toolchain"ダイアログが開きます。)

備考(\*) :

M3T-NC308WA の場合は、Renesas M32C Standard Toolchain... になります。

M3T-NC8C の場合は、Renesas R8C Standard Toolchain... になります。

- (2) 「link」タブをクリックする。
- (3) "Category:" から、"Other" を選択する。

"Start-up program is linked to a head."が自動先頭リンクオプションです。

ncrt0.a30 以外のファイル名のスタートアッププログラムの場合は、「M3T-NC30WA V.5.20R1 の場合」の方法で設定してください。

#### [Relocatable files]リストの概要

ワークスペースに登録されているファイルを[Relocatable files]リストに追加した場合、ワークスペースに登録されているファイルより先にリンクされます。

ワークスペースに登録されていないファイルを[Relocatable files]リストに追加した場合、ワークスペースに登録されているファイルの後にリンクされます。

[Relocatable files]リストに追加されたファイルは、リストの順にリンクされます。

## 8.5.2 リロケータブルファイルのリンク順序

---

### ■ 質問

リロケータブルファイルのリンク順序を変更するにはどうすればいいですか？

---

### ■ 回答

リロケータブルファイルは、以下に示す順番でリンクされます。

- (1) ワークスペースと[Relocatable files]リストの両方に登録されているファイルを[Relocatable files]リストの順にリンクします。
- (2) ワークスペースのみに登録されているファイルをファイル名のアルファベット順にリンクします。
- (3) [Relocatable files]リストのみに登録されているファイルをリストの順にリンクします。

必要に応じて、[Relocatable files]リストにファイルを追加してください。

### 8.5.3 モトローラ S フォーマットファイルの生成

---

■ 質問

モトローラ S フォーマットファイルを生成するにはどうすればいいでしょう？

---

■ 回答

次に示す設定を行ってください。

(1) メニュー [ オプション ] → [ ビルドフェーズ ] をクリックする。  
ビルドフェーズダイアログが表示されます。

(2) ビルドフェーズダイアログの「ビルド順序」タブをクリックする。  
ビルドフェーズの順序リストが表示されます。

M3T-NC30WA V.5.20 Release 1 の場合：「M16C Stype Converter」をチェックする。

M3T-NC30WA V.5.30 Release 1 の場合：「M16C Load Module Converter」をチェックする。

M3T-NC308WA V.5.20 Release 1 の場合：「M32C Load Module Converter」をチェックする。

M3T-NC8C V.5.30 Release 1 の場合：「R8C Load Module Converter」をチェックする。

M3T-CC32R の場合：「M32R Stype Converter」をチェックする。

(3) OK ボタンをクリックしてダイアログを閉じる。

(4) メニュー「オプション」→「Renesas M16C Standard Toolchain...」(\*)をクリックする。  
Renesas M16C Standard Toolchain ダイアログが表示されます。

備考(\*)：

M3T-NC308WA の場合は、Renesas M32C Standard Toolchain... になります。

M3T-NC8C の場合は、Renesas R8C Standard Toolchain... になります。

M3T-CC32R の場合は、Renesas M32R Standard Toolchain... になります。

(5) Renesas M16C Standard Toolchain ダイアログの Lmc タブをクリックする。  
ロードモジュールコンバータのオプション設定画面が表示されます。

(6) 必要なオプションを設定し、OK ボタンをクリックしてダイアログを閉じる。

以上で設定は終了です。

これ以降ビルド時にロードモジュールコンバータが実行されるようになります。

## 8.5.4 High-performance Embedded Workshop のインストール ( 1 )

---

### ■ 質問

High-performance Embedded Workshop 環境がすでにインストールされている PC に、High-performance Embedded Workshop が付属している M3T-NC30WA をインストールしましたが、High-performance Embedded Workshop を起動してもツールチェーンとして M16C が表示されません。

---

### ■ 回答

M3T-NC30WA に付属している High-performance Embedded Workshop をインストールする場合は、インストール済みの High-performance Embedded Workshop 環境と同じディレクトリに上書きしてインストールしてください。また、High-performance Embedded Workshop 環境は、M3T-NC30WA よりも先にインストールする必要がありますので、必ず M3T-NC30WA コンパイラ用のインストーラを使用してインストールしてください。

[SHC および H8C をご使用のお客様へ]

SHC Ver.7、H8C Ver.5 をご使用のお客様は、以下の URL から最新版コンパイラパッケージの High-performance Embedded Workshop へのリビジョンアップを行った上で、M3T-NC30WA コンパイラのインストールを行ってください。

<http://www.renesas.com/jpn/products/mpumcu/tool/download/hew/index.html>

### 8.5.5 High-performance Embedded Workshop のインストール ( 2 )

---

#### ■ 質問

High-performance Embedded Workshop を起動して、新規プロジェクトワークスペースを作成しようとしたところプロジェクトタイプ欄に何も表示されないため、ワークスペースを作成することができません。

---

#### ■ 回答

M3T-NC30WA に付属している High-performance Embedded Workshop 環境は、必ず M3T-NC30WA のインストーラを使用してインストールしてください。また、High-performance Embedded Workshop 環境がすでにインストールされている PC に High-performance Embedded Workshop が付属している M3T-NC30WA をインストールする場合は、既存の High-performance Embedded Workshop に上書きしてインストールしてください。

これらの対応をしても現象が改善されない場合は、以下に示す手順でツールチェーンの登録を行ってください。

- (1) High-performance Embedded Workshop を起動します。
- (2) 「ようこそ！」ダイアログが表示されますので、キャンセルボタンをクリックします。
- (3) High-performance Embedded Workshop のメニュー[ツール]→[アドミニストレーション]をクリックします。ツールアドミニストレーションダイアログが表示されます。
- (4) ツールアドミニストレーションダイアログの登録ボタンをクリックします。  
High-performance Embedded Workshop 登録ファイルの選択ダイアログが表示されます。
- (5) High-performance Embedded Workshop 登録ファイルの選択ダイアログのファイルの場所を M3T-NC30WA をインストールしたディレクトリ ( デフォルト¥MTOOL ) に移動します。
- (6) nc30wa.hrf を選択して、選択ボタンをクリックします。
- (7) ツールアドミニストレーションダイアログに戻りますので、OK ボタンをクリックします。

### 8.5.6 ビルドの停止

---

#### ■ 質問

ビルド中にコンパイルエラーなどが発生した場合に、ビルドを停止させる方法がありますか？

---

#### ■ 回答

High-performance Embedded Workshop のデフォルト設定では、ビルド中にエラーが発生してもそこでビルドを停止せず、リンクまで処理を実行します。次に示す設定を行うと、エラーが発生した時点でビルドを停止します。

- (1) メニュー「ツール」→「オプション」をクリックする。オプションダイアログが表示されます。
- (2) オプションダイアログのビルドタブをクリックする。ビルド処理に関する設定項目が表示されます。
- (3) 「エラーの数が超えた場合にビルド中止」をチェックする。  
(隣のテキストボックスの数値で、ビルド停止条件のエラー数を設定できます)
- (4) OK ボタンをクリックしてダイアログを閉じる。

ビルド停止条件として、ウォーニング発生を設定する場合は、「ウォーニングの数が超えたときにビルド中止」をチェックしてください。(エラー条件と同じく、隣のテキストボックスの数値で、ビルド停止条件のウォーニング数を設定できます)

### 8.5.7 ビルド対象の選択

---

#### ■ 質問

High-performance Embedded Workshop3 でビルドすると、指定ファイル（ワークスペースウィンドウの Project タブでツリーから選択するファイル）ではなく、エディタで開いているファイルがビルドされます。

---

#### ■ 回答

エディタで開いているファイルにフォーカスがある（カーソルがある）場合は、エディタで開いているファイルがビルド対象になります。これはソースファイル修正直後にビルドするケースが多いと想定しているためです。エディタにフォーカスがない場合、ワークスペースウィンドウの Project タブでツリーから選択されたファイルがビルド対象になります。

### 8.5.8 ビルドコンフィグレーション

---

#### ■ 質問

ビルドコンフィグレーションとして Release を選択するとビルドエラーになる場合でも、Debug を選択すると正常にコンパイルできます。なぜでしょうか？

---

#### ■ 回答

Debug と、Release で設定されているオプションが異なるためです。

ビルドコンフィグレーションの有用性について：

High-performance Embedded Workshop はプロジェクトを作成すると、Release と Debug の 2 つのビルドコンフィグレーションを作成します。コンフィグレーションにはそれぞれ異なったオプションパターンを設定することができ、ビルドコンフィグレーションを切り替えることで、簡単にオプションパターンを切り替えることができます。

ただし、High-performance Embedded Workshop が最初にビルドコンフィグレーションを作成する時、どちらのコンフィグレーションに設定されているオプションパターンも同じです。

上記の理由のため、High-performance Embedded Workshop は、一方のコンフィグレーションのオプションパターン設定が変更された場合、その変更を他方に自動的に反映しません。

### 8.5.9 デバッグ情報の出力

---

#### ■ 質問

NC8C V.5.30 Release 1 を使用しています。プロジェクト作成時のデフォルトのオプション設定でデバッグ情報は出力されますか？

---

#### ■ 回答

ワークスペース作成時のオプションには、デバッグ情報を出力するオプションは指定されていません。

デバッグ情報を出力する場合は、以下の手順でオプションを設定してください。

- (1) メニュー[オプション]→[Renesas R8C Standard Toolchain]をクリックします。
- (2) Renesas R8C Standard Toolchain ダイアログボックスが表示されます。
- (3) 「C」タブをクリックします。
- (4) Category から「Object」を選択します。
- (5) Debug options のリストから、[-g] をチェックします。
- (6) OK ボタンをクリックします。

## 8.6 SBDATA 宣言ユーティリティ

### 8.6.1 SBDATA 宣言ユーティリティ

---

#### ■ 質問

SBDATA 宣言ユーティリティ (utlxx) で sbdata.h を生成したところ、一部の変数がコメントになるのはなぜですか？

---

#### ■ 回答

既にプログラム上で SBDATA 宣言されている変数が優先的に SBDATA 領域に割り当てられます。SBDATA 宣言ユーティリティは、残りの部分に変数を割り当てます。そのため、SBDATA 領域に割り当てられなかった変数がコメントとして出力されています。出力されるコメントには次の 2 種類があります。

(1) 既に #pragma SBDATA で宣言されている変数を示すもの。

コメントの末尾に "@" がつきます。

```
//#pragma SBDATA ***** /* size=(1) / ref=[22] @ */
```

(2) 候補として挙げたが SBDATA には割り当てられなかったもの。

```
//#pragma SBDATA ***** /* size=(1) / ref=[22] */
```

---

## 付録

---



## 付録A. 追加機能について

### A.1 Ver1.00Release 1 から Ver2.00 Release 1 への追加機能

#### (1) SBADATA 宣言ユーティリティ SB308 追加

変数使用情報を解析し、アクセス頻度順にSBADATA宣言（#pragma SBADATA）を出力します。出力ヘッダファイルをインクルードしてリコンパイルすると、コードサイズを小さくできます。

#### (2) スペシャルページ宣言ユーティリティ SP308 追加

関数呼出し情報を解析し、呼出し回数順にスペシャルページ宣言（#pragma SPECIAL）を出力します。出力ヘッダファイルをインクルードしてリコンパイルすると、コードサイズを小さくできます。

#### (3) 最適化オプションのレベル化と追加

- ・ 最適化オプションを-O1～-O5までの五段階化し、最適化オプションの指定を容易にしました。

|      |                                                                                                                                                   |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| -O1: | デバッグ情報（行情報）に影響しない最適化のみを行います。                                                                                                                      |
| -O2: | V.2.00 Release 2では、-O1と同じです。                                                                                                                      |
| -O3: | デバッグ情報に影響する最適化も行います。-O指定時は、-O3と同じになります。                                                                                                           |
| -O4: | -O3の最適化に加え、const外部変数の参照を定数に置き換える最適化も行います。                                                                                                         |
| -O5: | -O4の最適化に加え、変数別名（間接参照等）を無視して共通部分式最適化を行います。但し、同一変数を操作する場合に、直接とポインタ間接を混在して操作したり、複数のポインタの間接で操作すると、予期した動作が行われないコードを生成することがありますので、生成コードを十分評価の上、御使用ください。 |

- ・ -Ono\_logical\_or\_combine(-ONLOC)  
連続した論理ORをまとめる最適化を抑止します。-O3以上のレベル指定時に有効です。
- ・ -Ocompare\_byte\_to\_word(-OCBTW)  
連続したメモリの連続したバイト単位の比較をワードで行います。  
本オプションは、他のオプション指定有無に関わらず有効です。

#### (4) 警告オプション追加

- ・ -Wlarge\_to\_small（-WLTS）  
大きいサイズから小さいサイズへの暗黙の型変換に対して警告します。  
本オプションは、-Wall指定時でも明示的に指定しなければ有効になりません。
- ・ -Wuninitialize\_variable(-WUV)

初期化されていないauto変数の使用に対して警告します。

-Wall指定時には、本オプションを指定しなくても自動的に有効になります。

なお、ユーザアプリケーションにおいてif、forなどによる条件分岐において初期化される場合、コンパイラは初期化されていないと判断します。そのため警告が出力されます。

例)

```
main()
{
 int i;
 int val;
 for(i=0;i<2;i++){
 f();
 val = 1; //論理上、ここで必ず初期化される
 }
 ff(val);
}
```

(5) 出力コード変更オプション追加

- -finfo  
SB308,SP308のための情報ファイルを出力します。
- -fuse\_CLIP(-fUC)  
CLIP命令も使用してコード生成します。
- -fuse\_MAX(-fUM)  
MIN,MAX命令も使用してコード生成します。

(6) asm 関数の変数名展開処理追加

\$@による変数名展開機能を追加しました。\$@を指定すると、該当する変数がauto変数、レジスタ変数、外部変数であるかをコンパイラ側で判断して出力します。

例) asm(" mov.w #10H,\$@,I);

(7) AS308 の機能追加

アセンブラ指示命令及びニーモニックのオペランドに文字列を含む演算が記述できます。

(8) XRF308 の機能追加

xrf308が一度にオープンできるファイル数の上限を600に変更しました。

A.2 Ver 2.00 Release 1 から Ver2.00 Release 2 への機能追加  
なし。

### A.3 Ver2.00 Release 2 から Ver3 . 00 Release 1 への機能追加

#### (1) アセンブラオブティマイザ aopt308 追加

アセンブラオブティマイザaopt308を追加し、”adjnz”命令による条件分岐の最適化などを行うようにしました。なお、aopt308は、コンパイルドライバnc308を-O,-O3,-O4,-O5,-OR,-OSのいずれかのオプションを指定してコンパイルした場合に自動的に起動されます。

#### (2) SBADATA 宣言・スペシャルページ宣言ユーティリティ utl308 追加

SBADATA宣言ユーティリティ、スペシャルページ関数宣言ユーティリティをutl308に統合し、従来のsb308・sp308は廃止しました。

本ユーティリティの出力ヘッダファイルをインクルードすると、コードサイズを小さくできます。

#### (3) STK ビューワの追加（W95J 版、SOLARIS 版のみ）

スタック使用量計算ユーティリティをGUI化し、操作性・視認性を向上しました。

#### (4) マップビューワの追加（W95J 版のみ）

マップ情報表示ユーティリティを追加しました。マップ情報がGUIにより見やすく表示できます。

#### (5) 統合化開発環境 TM V.3.00 対応（W95J 版のみ）

統合化開発環境TM V.3.00に対応しています。

なお、TM V.2.01以前のバージョンでは使用できませんので、ご注意ください。

#### (6) スタック情報、ユーティリティ情報の統合

従来、スタック情報(.stk)、SBADATA宣言・スペシャルページ宣言ユーティリティ情報(.utl)は、ソースファイル毎に個別のファイルに出力していましたが、これらをインスペクタ情報に統合し、個別のデータファイルを作成しないようにしました。

なお、インスペクタ情報は、統合化開発環境TM V.3.00で使用する情報で、リロケータブルファイル(.r30)、アプソリュートモジュールファイル(.x30)に格納されます。

#### (7) NC308 の機能追加

---

(a) 最適化オプションの追加

- ・ コマンドオプション '`-Oloop_unroll[=最大ループ回数](-OLU)`' の追加  
コンパイル時にループ回数が判明しているforループを展開し、実行速度を向上します。  
最大ループ回数を指定した場合、そのループ回数以内のループを展開対象とします。  
最大ループ回数の指定を省略すると5回以内のforループを対象とします。

(b) 警告機能の強化と抑止オプションの追加

- ・ 標準ライブラリ関数ヘッダファイルのインクルード忘れの警告機能追加  
'`-Wno_prototype`' または '`-Wall`' オプション指定時、標準ライブラリ関数を使用し、その使用に必要なヘッダファイルがインクルードされていない場合にウォーニングメッセージを出力します。
- ・ コマンドオプション '`-Wno_warning_stdlib(-WNWS)`' の追加  
上記の標準ライブラリ関数ヘッダファイルのインクルード忘れについて、警告を抑止します。  
本オプションは、'`-Wnon_prototype(-WNP)`' または '`-Wall`' 指定時に有効です。

(c) 関数先頭アライメントのデフォルト化と抑止オプションの追加

関数先頭の命令配置の偶数アライメントをデフォルトで行うようにしました。  
これに伴い、関数先頭の命令配置の偶数アライメントを抑止する '`-fno_align`' オプションを追加し、偶数アライメントを行う '`-falign`' オプションは廃止しました。

(d) ユーティリティ情報出力オプションの変更

- ・ コマンドオプション '`-finfo`' オプションの機能変更  
スタック情報、ユーティリティ情報の統合により、'`-finfo`' オプションでインスペクタ情報（スタック情報、ユーティリティ情報を含む）を出力するようにしました。  
従来の '`-fshow_stack_usage(-fSSU)`' オプションは廃止しました。

(e) CLIP・MAX・MIN 命令使用のデフォルト化

CLIP・MAX・MIN命令を使用するオプション '`-fuse_CLIP(-fUC)`'、'`-fuse_MAX(-fUM)`' を廃止して、指定がなくてもこれらの命令を使用するようにしました。

(8) AS308 の機能追加

(a) インスペクタ情報生成機能追加

- ・ コマンドオプション '`-finfo`' を追加

インスペクタ情報を生成するためのコマンドオプション ‘-finfo’ を追加しました。

- ・ 指示命令の追加  
以下の指示命令を追加しました。  
 ‘.INSF’ ..... インスペクタ情報の関数(サブルーチン)開始情報を示します。  
 ‘.EINSF’ ..... インスペクタ情報の関数(サブルーチン)終了情報を示します。  
 ‘.CALL’ ..... インスペクタ情報の関数(サブルーチン)呼出し先情報を示します。  
 ‘.STK’ ..... インスペクタ情報のスタック情報を示します。

#### (b) 分岐最適化の実施

‘adjnz’ および ‘sbjnz’ 命令に対して、分岐最適化を実施します。なお、最適選択規則については条件分岐命令と同等です。

#### (9) LMC308 の機能追加

- ・ コマンドオプション ‘-A’ を追加  
生成するファイルに出力する機械語データのアドレス範囲を指定するコマンドオプション ‘-A’ を追加しました。
- ・ コマンドオプション ‘-F’ を追加  
指定したアブソリュートモジュールファイル内のデータ登録されていないアドレスに対して任意のデータを出力するコマンドオプション ‘-F’ を追加しました。
- ・ 出力ファイル名指定オプションの機能を拡張  
出力ファイル名指定オプション ‘-O’ にて、出力ファイル名の拡張子を指定可能としました。
- ・ 三菱専用HEXフォーマットファイル生成条件の機能を変更  
コマンドオプション ‘-H’ 指定時、指定されたアブソリュートモジュールファイルに登録されているすべてのセクションに対して設定されているデータの最大アドレス値を元に、三菱専用HEXフォーマットで生成していましたが、本バージョンでは、CODEまたはROMDATA属性を持つセクションが1MBアドレスを越える場合にのみ生成されるように変更しました。また、三菱専用HEXフォーマットファイルを生成した場合、ウォーニングメッセージ “ Original HEX format for mitsubishi microcomputers is generated ” を出力します。
- ・ ROMデータが1バイトも存在しない場合の機能を変更  
指定されたアブソリュートモジュールファイルのCODEまたはROMDATA属性を持つセクションに  
1バイトもデータが存在しない場合、従来はデータが空の機械語ファイルを生成していましたが、エラーとなるように変更しました。
- ・ 実行経過表示の追加  
アセンブラ同様にlmc308実行状況を経過表示するようにしました。

## A.4 Ver3.00 Release1 から Ver3.10 Release1 への機能追加

### (1) NC308 の機能追加

#### (a) 最適化の強化

コンパイラ本体の条件分岐、ビット操作、レジスタ変数などの最適化を強化しました。  
また、アセンブラオプティマイザも、連続する領域やレジスタの同種論理演算をまとめる最適化や、btsts等の機種固有命令最適化などを拡充しました。

#### (b) 警告機能の強化とオプションの追加

コマンドオプション ‘-Wno\_used\_argument(-WNUA)’ を追加しました。  
本オプションを指定すると、関数定義中で未使用の仮引数を警告します。  
本機能は、‘-Wall’ を指定しても有効になりません。別に本オプションを指定してください。

#### (c) アセンブルリストファイル作成オプションの追加

コマンドオプション ‘-dsource\_in\_list(-dSL)’ を追加しました。  
本オプションを指定すると、リロケータブルファイルを作成するときに、同時にアセンブルリストファイルも作成し、C言語ソース行をコメントとして出力します。  
但し、‘-P’, ‘-E’, ‘-S’ オプション指定時には、アセンブルは行われないので、効果はありません。

#### (d) ‘-dsource(-dS)’ オプションの機能変更

本オプションを指定して、生成するアセンブリ語ソースにC言語ソース行をコメントとして出力する場合に、不要なりロケータブルファイル(.r30)を残さないようにしました。

### (2) AS308 の機能改訂

ビット操作命令 “BTST” のオペランドに絶対アドレッシング “base:19” を指定した場合に、Sフォーマット形式にてコードを生成するよう変更しました。

### (3) SBDATA 宣言&スペシャルページ関数宣言ユーティリティ utl308 機能強化

- ・ コマンドオプション ‘-sb308’, ‘-sp308’ が同時に指定できるようにしました。  
SBDATAの処理とスペシャルページの処理を同時に行うことができるようにしました。
- ・ コマンドオプション ‘-fsection’ を追加しました。  
本オプションを指定すると、#pragma SECTIONで配置セクションを変更した変数と関数も処理対象にします。
- ・ コマンドオプション ‘-o’ の仕様変更と ‘-fover\_write(-fOW)’ の追加  
‘-o’ オプションで出力ファイルを指定する場合、既存ファイルを指定すると、上書きせずに結果を標準出力に出力するようにしました。

‘-fover\_write(-fOW)’ オプションを同時に指定すると、‘-o’ で既存ファイルを指定した場合に、強制的に上書きします。

## A.5 Ver 3.10 Release 1 から Ver 3.10 Release 2 への機能追加

### (1) NC308 の機能追加

#### (a) #include ネスト上限値の増加

#include指令で取り込まれるファイルのネスト数の上限を8から40に増加しました。

#### (b) コメント処理変更

コメント処理の仕様を一般的なC++ の仕様と同一にしました。

以前のバージョンでは、/\* と \*/ で囲まれた部分を処理してから、// から改行文字までを処理していましたが、本バージョンでは、先に記述されたコメント区切り文字からコメント処理を行います。

従って、以下のような例では、以前のバージョンと動作が変わりますので、ご注意ください。

```
i = 4 /* comment */
 +2;
```

以前のバージョンでは、/\* \*/ の間をコメントとし、「 i = 4 / +2; 」としていました。

本バージョンでは、//から行末までをコメントとし、「 i = 4 +2; 」となります。

#### (c) #pragma 拡張機能名の小文字対応

#pragmaに続く拡張機能指定語(ADDRESS, INTERRUPT, SBDATA, ASM等)を小文字でも記述できるようにしました。これらは大文字と小文字のどちらで記述しても機能は同等です。

#### (d) #pragma ASM, #pragma ENDASM の仕様変更

従来、#pragma ASMから#pragma ENDASMの間で、クォート文字「"」又は「'」を対にせずに記述した場合、トークン解析上エラーとしていましたが、そのような記述ができるようになりました。

```
#pragma ASM

 nop ; Insert "NOP" ← 「"」が対なので、問題ありません。
 nop ; Don't care ← 「'」が対でないので、従来はエラーでしたが、
 このような記述ができるようになりました。

#pragma ENDASM
```

(e) 警告機能の強化

コマンドオプション ‘-Wno\_used\_argument(-WNUA)’ 指定時、未使用のスタック渡し仮引数を警告していましたが、未使用のレジスタ渡し引数についても警告するようにしました。

A.6 Ver 3.10 Release 2 から Ver3.10 Release 3 への機能追加  
なし。

## A.7 Ver3.10 Release 3 から Ver5 . 00 Release 1 への機能追加

(1) Linux 版の追加

本バージョンより、Linux版(日本語Turbolinux 7 workstation対応)を追加しました。

(2) NC308 の機能強化

(a) long long 型、\_Bool 型、restrict 修飾子をサポート

ISO/IEC 9899:1999 (ANSI C99) で新設された型(long long型、\_Bool型)、restrict修飾子をサポートしました。

(b) 拡張機能 #pragma BITADDRESS の追加

\_Bool型の外部変数を指定絶対アドレスの1ビットに割り付ける拡張機能 #pragma BITADDRESS を追加しました。

(c) 拡張機能 #pragma SB16DATA の追加

外部変数を dsp16[SB]アドレッシングでアクセスするよう指定する拡張機能 #pragma SB16DATA を追加しました。

(d) 拡張機能 #pragma DMAC の追加

外部変数をDMAC専用レジスタに割り付け、C言語でDMACレジスタをアクセスする拡張機能 #pragma DMAC を追加しました。本機能は、DMACのチャンネル0および1をサポートしています。

(e) 最適化機能強化

以下のような最適化（特に実行速度を向上する最適化）を強化しました。

- ・ inline関数機能強化
- ・ 定数伝播最適化強化
- ・ if文などの分岐を解析した最適化
- ・ 四則演算の最適化など

(f) 拡張機能 #pragma SECTION の仕様変更

1つのソースファイル内で“data”, “rom” のセクション名を複数回変更できるように #pragma SECTION の仕様を変更しました。

(g) M32C/80 シリーズ用プリデファインマクロの変更

M32C/80シリーズ用コード生成オプション“-M82”を使用したとき、プリデファインマクロとして“M16C80”ではなく“M32C80”が定義されるようにしました。

(h) 割込み処理関数の高速化

割込み処理関数のレジスタ退避および復帰処理などを高速化しました。

## (i) asm 関数の拡張

asm関数内に、\$\$、\$@ を2つまで使用できるようにしました。

## (j) 整数の二進数記述をサポート

整数として二進数が記述できるようにしました。接頭語"0b"で始まる整数は二進数として扱われます。例えば、16進数 "0x12" を二進数で "0b00010010" と記述することができます。

## (3) 標準ライブラリ関数のインライン展開マクロの追加

標準ライブラリ関数 strcpy, strcmp, memcpyおよび memsetのインライン展開マクロを標準ヘッダ <string.h> に追加しました。

## (4) AS308 の機能強化

## (a) アドレスレジスタ相対アドレッシングに対する最適化の実施

一般命令およびビット命令のアドレスレジスタ相対値が0(ゼロ)の場合、アドレスレジスタ間接アドレッシングを選択します。

```
mov.w #1234h, 0 [A0] → mov.w #1234h, [A0]
bclr 1, 0 [A0] → bclr 1, [A0]
```

## (b) 指示命令 ".SBSYM16" の追加

本指示命令にて定義されたシンボルを参照した場合、dsp:16[SB]アドレッシングを選択します。

```
.glb sym
.sbsym16 sym
abs.b sym → dsp:16 [SB] を選択
```

## (5) LN308 の機能強化

## (a) コマンドファイルの記述規則変更

一行に記述可能な文字数を255文字から2048文字に拡張しました。

## (6) マップビューワの機能強化

マップビューワに以下の機能を追加しました。

- ・ マップ情報印刷機能

- ・ 左側ウィンドウのスクロール機能
- ・ メモリサイズイメージの拡大・縮小表示

## A.8 Ver5.00 Release 1 から Ver5.10 Release1 への機能追加

### (1)NC308 機能追加

#### (a) コマンドオプション "-fdouble\_32[-fD32]"の追加

double 型を float 型と同じ 32 ビットのデータ長で扱うことを指定します。

##### – 注意

本オプションを使用するには、必ず関数のプロトタイプを明記してください。プロトタイプ宣言がない場合は正しいコードを生成しない場合があります。

#### (b) コマンドオプション "-Wno\_used\_static\_function[-WNUSF]"の追加

-Ostatic\_to\_inline[-OSTI]オプション指定により、static 関数がインライン展開される、または static 関数がどこからも参照されない場合に警告メッセージを表示します。

##### – メッセージ内容

[Warning(ccom):xxx.c,line xx] Code generation for static function (関数名) can be suppressed by using -ferase\_static\_function(-fESF) option.

#### (c) コマンドオプション "-ferase\_static\_function=static 関数名[-fESF=static 関数名]"の追加

指定された static 関数に対するコード生成を行いません。

##### – 備考

本オプションは、コマンドオプション "-Wno\_used\_static\_function[-WNUSF]"で検出した static 関数に対して指定してください。

#### (d) コマンドオプション "-Oconst"の強化

定数で初期化されている配列データの参照を定数の参照に置き換えます。

#### (e) 積和演算処理の強化

for 文中の積和演算処理に対して、rmpa 命令を生成するようにしました。

##### – プログラム例

```
signed char ch1[10];
signed char ch2[10];
int ih1[10];
int ih2[10];

#define LOOP_MAX 9

signed char *pc1,*pc2;
int *pi1,*pi2;

int i;
long l;

void func_c1(void)
{
 int j;

 i = 0;
 for(j=0; j<LOOP_MAX; j++){
 i = i + (int)*pc1 * (int)*pc2;
 pc1++;
 }
}
```

```

 pc2++;
 }
}

void func_c2(void)
{
 int j;

 i = 0;
 for(j=0; j<LOOP_MAX; j++){
 i = i + (int)ch1[j] * (int)ch2[j];
 }
}

```

– 生成コード例

```

_func_c1:
 pushm R1,R2,R3,A0,A1
 mov.w #0000H,_i:16
 mov.w _i:16,R0
 mov.l _pc1:16,A0
 mov.l _pc2:16,A1
 mov.w #0009H,R3
 rmpa.b
 mov.w R0,_i:16
 add.l #00000009H,_pc1:16
 add.l #00000009H,_pc2:16
 popm
 rts

_func_c2:
 pushm R1,R2,R3,A0,A1
 mov.w #0000H,_i:16
 mov.w _i:16,R0
 mov.w #(_ch1&0FFFFH),A0
 mov.w #(_ch2&0FFFFH),A1
 mov.w #0009H,R3
 rmpa.b
 mov.w R0,_i:16
 popm
 rts

```

– 補足

- 最適化オプション-O[3-5],-OS,-OR のいずれかの指定が必要です。
- for 文における積和演算の処理内容により、rmpa 命令を生成しない場合があります。

(2) AS308 機能追加

(a) AS308 オプション "-PATCH\_TA"および "-PATCH\_TAn"の追加

三相モータ制御用タイマ機能の注意事項を回避するコードを生成します。

(b) LN308 オプション "-U"の追加

未使用である関数が含まれている場合にウォーニングを出力します。

(c) LMC308 オプション " -F" の拡張

空き領域に任意データを埋め込む処理に対して、範囲指定を可能にしました。

## A.9 Ver5.10 Release 1 から Ver5.20 Release1 への機能追加

### (1) NC308 機能追加

#### (a) 統合開発環境 High-performance Embedded Workshop の追加

エディタ～デバッガまでの各ツールの結合および一括管理する統合開発環境を導入しました。

#### (b) コマンドオプション `"-fno_switch_table [-fNST]"` の追加

switch 文に対して、常に比較を行い分岐するコードを生成します。

– 備考

switch 文に対するテーブルジャンプを用いたコード生成を抑止する場合に指定します。

#### (c) コマンドオプション `"-fswitch_other_section [-fSOS]"` の追加

switch 文に対するテーブルコードをプログラムセクションとは別のセクションに指定します。

– 注意

コマンドオプション `"-fno_switch_table [-fNST]"` を指定した場合は、本オプションは無効になります。

#### (d) コマンドオプション `"-fmake_vector_table [-fMVT]"` の追加

割り込みベクタテーブルを自動生成します。

#### (e) コマンドオプション `"-fmake_special_table [-fMST]"` の追加

スペシャルページテーブルを自動生成します。

#### (f) コマンドオプション `"-Ofoward_function_to_inline [-OFFTI]"` の追加

全てのインライン関数に対してインライン展開を行います。

#### (g) コマンドオプション `"-Ofloat_to_inline [-OFTI]"` の追加

浮動小数点のランタイムライブラリをインライン展開し、浮動小数点演算処理を高速化します。

– 備考

コンパイルオプション `-M82` と同時に指定した場合に有効となります。

#### (h) コマンドオプション `"-Oglb_jmp [-OGJ]"` の追加

リンク時に分岐距離に応じて最適な分岐命令を選択します。

#### (i) コマンドオプション `"-Wno_used_function [-WNUF]"` の追加

未使用の関数を検出した場合にウォーニングを出力します。

#### (j) コマンドオプション `"-Wstop_at_link [-WSAL]"` の追加

リンク時にウォーニングが発生した場合、アブソリュートモジュールファイルを生成しません。

#### (k) コマンドオプション `"-Wundefined_macro [-WUM]"` の追加

`#if` 文中で未定義のマクロを使用した場合にウォーニングを出力します。

#### (l) 拡張機能 `"#pragma INTHANDLER(#pragma HANDLER)"` にオプションを追加

割り込みハンドラに入った直後に多重割り込みを許可します。

– 備考

本機能は、`#pragma INTHANDLER(#pragma HANDLER)` にオプション `/E` を指定すると有効になります。

### (2) AS308 機能追加

#### (a) コマンドオプション `"-fMVT"` の追加

可変ベクタテーブルを自動生成します。

## (b) コマンドオプション "-fMST" の追加

スペシャルページテーブルを自動生成します。

## (c) コマンドオプション "-JOPT" の追加

グローバルラベルを参照している分岐命令を最適化します。

## (d) 指示命令 ".ID" の追加

ID コードチェック機能の ID コードを設定します。

## (e) 指示命令 ".PROTECT" の追加

ROM コードプロテクト制御番地に値を設定します。

## (f) 指示命令 ".RVECTOR" の追加

ソフトウェア割り込み番号とソフトウェア割り込み名を設定します。

## (g) 指示命令 ".SVECTOR" の追加

スペシャルページ番号とスペシャルページ名を設定します。

## (3) LN308 機能追加

## (a) コマンドオプション "-fMVT" の追加

可変ベクタテーブルを自動生成します。

## (b) コマンドオプション "-fMST" の追加

スペシャルページテーブルを自動生成します。

## (c) コマンドオプション "-VECT" の追加

可変ベクタテーブルの自動生成を行った際の空き領域に対してアドレスを設定します。

## (d) コマンドオプション "-JOPT" の追加

グローバルラベルを参照している分岐命令を最適化します。

## (e) コマンドオプション "-W" の追加

ウォーニング発生時、アブソリュートモジュールファイルを生成しません。

## (f) コマンドオプション "-L" の注意事項解消

マニュアルに記載されていた注意事項を解消しました。

– 旧バージョンの注意事項

複数のライブラリファイルが指定されたとき、リンクは指定された順にライブラリファイルを参照します。したがって、次の条件を満たす場合にシンボル未定義エラーが発生します。

- リロケータブルファイル(sample.r30)がライブラリファイル(A.LIB)に登録されているグローバルシンボルを参照している。
- (1)でリンク対象となったライブラリファイル(A.LIB)内のリロケータブルモジュールが別のライブラリファイル(B.LIB)に登録されているグローバルシンボルを参照している。
- -L オプションで、上記(1)のライブラリファイル(A.LIB)よりも先に(2)の別のライブラリファイル(B.LIB)が指定されている (例 1) 。
  - 例1 :
 

```
>ln308 sample.r30 -L B.LIB A.LIB
```

## (4) LMC308 機能追加

## (a) コマンドオプション "-protectx" の追加

ROM コードプロテクト制御番地に値を設定します。

付録

---

MEMO

---

M16Cファミリ用Cコンパイラ  
アプリケーションノート  
M3T-NC308WA/M3T-NC30WA

発行年月日 2005年6月16日 Rev.2.00

発行 株式会社 ルネサス テクノロジ 営業企画統括部  
〒100-0004 東京都千代田区大手町2-6-2

編集 株式会社 ルネサス ソリューションズ ツール開発部

---

© 2005. Renesas Technology Corp. and Renesas Solutions Corp., All rights reserved. Printed in Japan.

# M16C ファミリ用 C コンパイラ アプリケーションノート



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

RJJ10J0793-0200