お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010 年 4 月 1 日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

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

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

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

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

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

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

Application Note

740ファミリ用Cコンパイラパッケージ M3T-ICC740

アプリケーションノート

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2006.07

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が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は負いません。
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、その他お気付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は、ルネサス 8 ビットシングルチップマイクロコンピュータ 740 ファミリ上で動作する応用プログラムを 740 ファミリ C コンパイラパッケージを用いて効果的に作成する方法を説明します。

なお、本アプリケーションノートで説明した内容の詳細については、次の関連マニュアルに記載されておりますのであわせて参照してください。

IMA **アセンブラ・プログラミング・ガイド（三菱** 740 **ファミリ）第２版**
IAR C **ライブラリ関数 リファレンス・ガイド**
ICC **コンパイラ・プログラミング・ガイド（三菱** 740 **ファミリ）第２版**
740 **ファミリ** 740 **ファミリ参考プログラム集**
740 **ファミリ ソフトフェアマニュアル**

本アプリケーションノートはIARシステムズ株式会社殿のご了解のもと、『ICCコンパイラ・プログラミング・ガイド』、『IMAアセンブラ・プログラミング・ガイド』の内容を転用しております。
また、『740ファミリプログラミング作成の手引き<C言語編>RJJ05B0468-0200/Rev2.00』をベースに作成したものです。

本アプリケーションノートは次のような構成になっております。
第 1 章では、C 言語入門。
第 2 章では、プロジェクトの設定に関する説明。
第 3 章では、C コンパイラ ： ICC740 の説明。
第 4 章では、アセンブラ ： A740 の説明。
第 5 章では、リンカ ： XLINK の説明。
第 6 章では、デバッガの説明。
第 7 章では、コーディングのコツ。
第 8 章では、スタックの見積り方法の説明。
第 9 章では、割り込み処理に関する説明。

〈本アプリケーションノートで使用する記号などの意味〉

(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。
[ ]　　：省略できることを示します。
abc　　：斜体の部分は、コマンドの一部として実際に入力しなければならない値またはラベルを示します。
{a | b}　：いずれかを選択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 "がついているのは 16 進数です。
0x　　：整数定数の先頭に" 0x "がついているのは 16 進数です。
[Menu->Menu Option]　：太字と->はメニューオプションを示します。

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 の米国およ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。

目次

第1章　C言語入門 ____ 6

1.1　C言語によるプログラミング 7

1.1.1　アセンブリ言語とC言語 7
1.1.2　プログラム開発の手順 8
1.1.3　わかりやすいプログラム 10

1.2　データ型 14

1.2.1　C言語で扱える“定数” 14
1.2.2　変数 16
1.2.3　データの特性 18

1.3　演算子 20

1.3.1　ICC740の演算子 20
1.3.2　数値計算のための演算子 21
1.3.3　データ加工のための演算子 23
1.3.4　条件を調べるための演算子 25
1.3.5　その他の演算子 26
1.3.6　演算子の優先順位 28
1.3.7　間違いやすい演算子の使用例 29

1.4　制御文 31

1.4.1　プログラムの構造化 31
1.4.2　条件による処理の分岐(分岐処理) 32
1.4.3　同じ処理の繰り返し(繰り返し処理) 36
1.4.4　処理の中断 39

1.5　関数 41

1.5.1　関数とサブルーチン 41
1.5.2　関数の作成 42
1.5.3　関数間でのデータの受け渡し 44

1.6　記憶クラス 45

1.6.1　変数と関数の有効範囲 45
1.6.2　変数の記憶クラス 46
1.6.3　関数の記憶クラス 48

1.7　配列とポインタ 50

1.7.1　配列 50
1.7.2　配列の作成 51
1.7.3　ポインタ 53
1.7.4　ポインタの活用 55
1.7.5　ポインタの配列化 57
1.7.6　関数ポインタを使ったテーブルジャンプ 59

1.8　構造体と共用体 60

1.8.1　構造体と共用体 60
1.8.2　新しいデータ型の作成 61

1.9 プリプロセスコマンド......65
1.9.1 ＩＣＣ740のプリプロセスコマンド......65
1.9.2 ファイルの取り込み......66
1.9.3 マクロの定義......67
1.9.4 条件コンパイル......69
第2章 プロジェクトの設定______71
2.1 設定内容......72
2.2 メモリモデルの説明......73
2.2.1 メモリモデルの詳細......73
2.2.2 メモリモデルの変更......74
2.3 セグメント構成......77
2.3.1 ＩＣＣ740のセグメント構成......77
2.3.2 セグメントマップ：ZページRAM(0H～FFH)......78
2.3.3 セグメントマップ：NページRAM(100H～)......79
2.3.4 セグメントマップ：ROM(～FFFFH)......81
2.4 スタック領域の説明......83
2.4.1 ＩＣＣ740スタック管理......83
2.4.2 CSTACKセグメントの変更......85
2.5 オブジェクトフォーマットの説明......87
2.5.1 オブジェクトフォーマットの変更......87
2.6 Cスタートアップ・モジュールの説明......88
2.6.1 Cスタートアップ・モジュールの解説......88
2.7 特殊な領域への値の設定方法......101
2.7.1 特殊な領域への値の設定方法......101
第3章 Cコンパイラ ： ICC740 ______102
3.1 基本的なコンパイラオプションの説明......103
3.1.1 コンパイラ・オプションの要約......103
3.2 拡張機能について......104
3.2.1 拡張キーワードの要約......104
3.2.2 #PRAGMA疑似命令の要約......105
3.2.3 定義済みシンボルの要約......106
3.2.4 その他の拡張機能......106
第4章 アセンブラ ： Ａ740 ______107
4.1 基本的なオプションの説明......108
4.1.1 アセンブラオプションの概要......108
4.2 アセンブリ言語インターフェース......109
4.2.1 関数宣言......109

4.2.2 C言語からアセンブリ言語サブルーチンをコール 110
4.2.3 アセンブリ言語からC言語関数をコール 110
第5章 リンカ ： ＸＬＩＮＫ 113
5.1 基本的なオプションの説明 114
5.1.1 オプションの概要 114
5.2 オプションファイルの説明 115
5.2.1 リンク・コマンド・ファイルの解説 115
第6章 デバッガ 121
6.1 デバッガの起動 122
6.1.1 シミュレータの接続 123
6.1.2 シミュレータの終了 122
6.2 シミュレータのセットアップ 123
6.2.1 740のInitダイアログ 123
6.3 シミュレータ用MCUファイルの作成 124
第7章 コーディングのコツ 125
第8章 スタックの見積もり方法 133
8.1 デフォルトのスタックサイズ 134
8.2 EXPR_STACK / INT_EXPR_STACKセグメント 134
8.3 CSTACKセグメント 134
8.4 C_ARGN / C_ARGZセグメント 135
8.5 RF_STACKセグメント 135
8.6 ICC740のランタム関数のスタック使用量 136
第9章 割り込み処理 139
9.1 割り込み処理 140
9.1.1 割り込み処理関数の記述例 140
9.1.2 割り込み処理関数の記述 141
9.1.3 割り込み禁止フラグ（Iフラグ）の設定 142
9.1.4 割り込みベクトル領域の登録 143
9.1.5 割り込みベクトルセグメントの設定 143
9.2 多重割り込み 144
9.2.1 多重割り込みの使用方法 144
9.2.2 多重割り込みに関する定義 145
9.2.3 多重割り込み処理関数の記述例 146

第　1　章

C言語入門

1.1 C言語によるプログラミング

1.2 データ型

1.3 演算子

1.4 制御文

1.5 関数

1.6 記憶クラス

1.7 配列とポインタ

1.8 構造体と共用体

1.9 プリプロセスコマンド

この章では、初めてC言語をお使いになる方を対象に、組み込み型プログラムを作成するために必要なC言語の基礎を紹介しています。

1.1　C 言語によるプログラミング

1.1.1　アセンブリ言語とC言語

主なC言語の特長と、C言語によるプログラムの記述方法を説明します。

C 言語の特長

(1)　処理の流れを追いやすいプログラムが記述できる
構造化プログラミングの基本である「順次処理」、「分岐処理」、「繰り返し処理」をすべて制御文で記述できます。このため、処理の流れを追いやすいプログラムが記述できます。

(2)　モジュール分割が容易にできる
C言語で記述したプログラムは「関数」と呼ばれる基本単位から構成されています。関数はパラメータの独立性が高いため、プログラムの部品化や再利用が容易にできます。また、アセンブリ言語で記述したモジュールを流用することができます。

(3)　保守性のよいプログラムを記述できる
(1)、(2)の理由から、運用後のプログラムのメンテナンスが容易にできます。また、C言語としての標準規格(ANSI規格[注])が定められているため、ソースプログラムを少し変更するだけで他機種への移植も可能です。

(注)　C言語の移植性を保つために、ANSI(American National Standards Institute)で定められたC言語の標準規格です。

C 言語とアセンブリ言語の比較

ソースプログラムの記述方法に関して、アセンブリ言語との違いをまとめます。

	C 言語	アセンブリ言語
プログラムの基本単位 (記述方法)	関数 (関数名( ){ })	サブルーチン (サブルーチン名:)
書式	フリーフォーマットで ANSI C 言語に準拠	1 行 1 命令
大文字/小文字の区別	大文字と小文字を区別する	区別しない
データ領域の確保	型で指定する	サイズ(バイト数)で指定する (擬似命令を使用する)

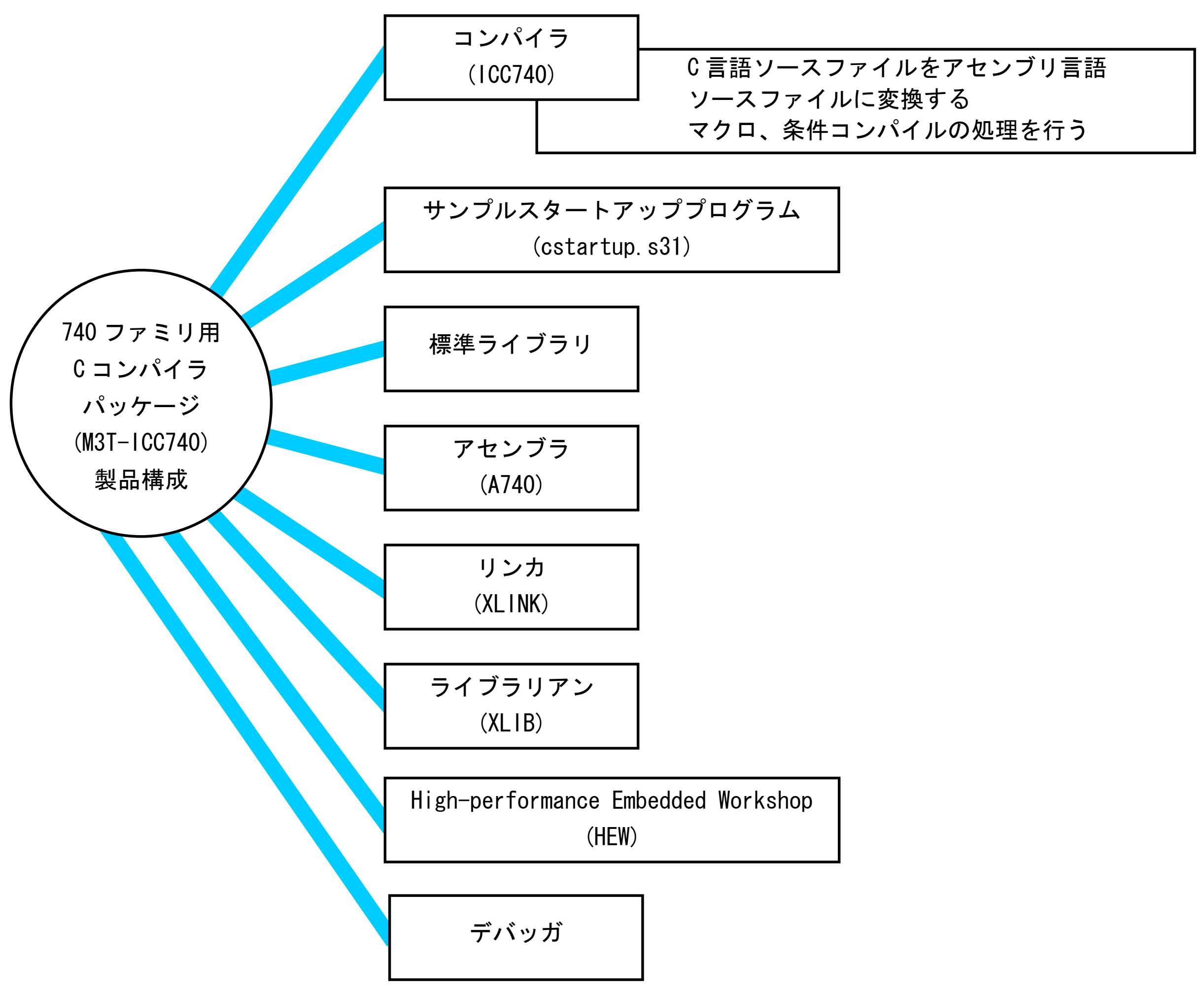
1.1.2　プログラム開発の手順

C言語で記述されたソースプログラムを機械語に翻訳する作業を「コンパイル」といいます。この作業を行うために用意されたソフトウェアが「コンパイラ」です。

この項では､ルネサス8ビットマイクロコンピュータ740ファミリ用Cコンパイラパッケージ(M3T-ICC740)を用いてプログラム開発を行う手順を説明します。

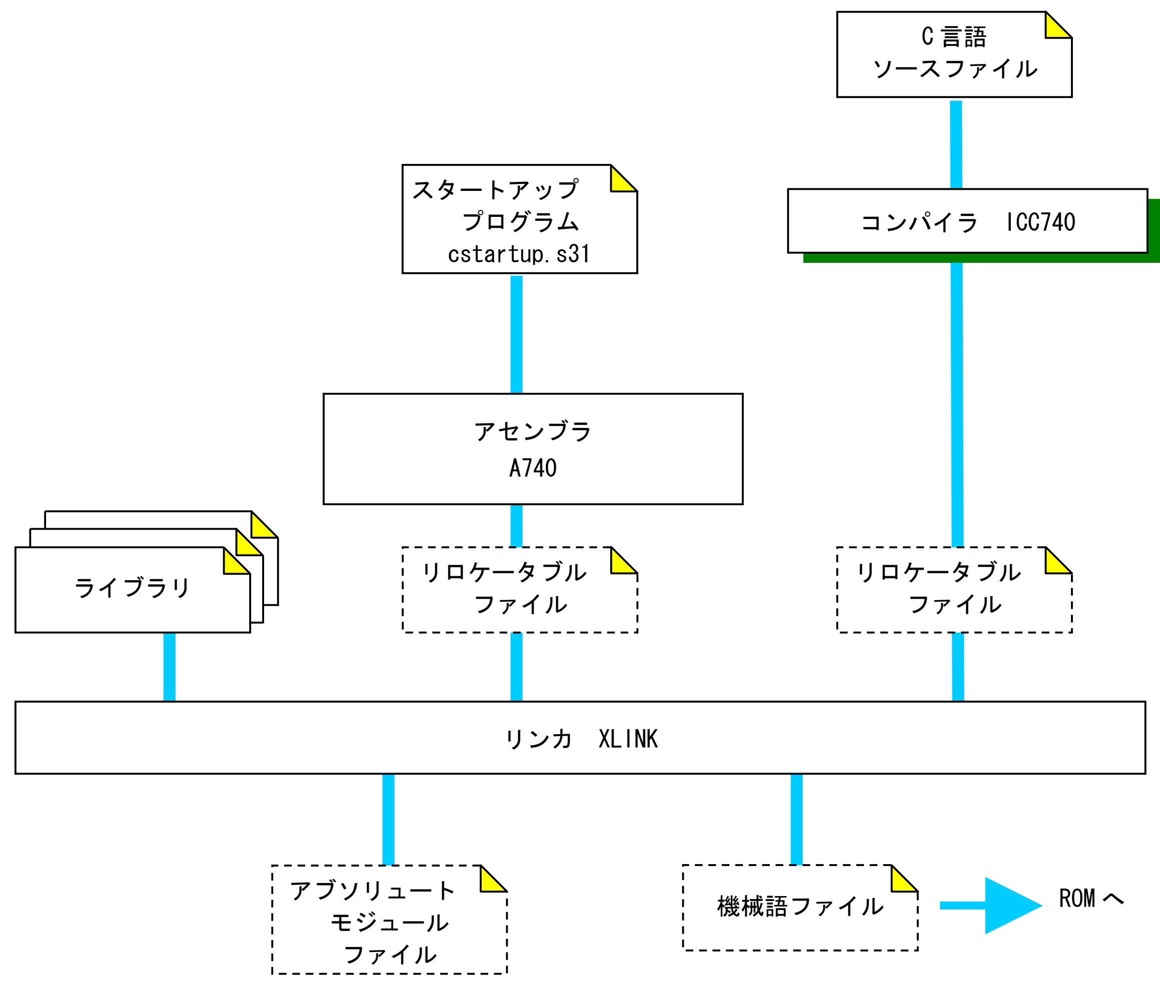
740ファミリ用Cコンパイラパッケージ(M3T-ICC740)の製品一覧

ルネサス8ビットマイクロコンピュータ740ファミリ用Cコンパイラパッケージ(M3T-ICC740)に含まれる製品の一覧を示します。



ソースファイルから機械語ファイルができるまで

ICC740 で機械語ファイルを生成するには、C 言語でプログラムを記述したソースファイルのほかに、アセンブリ言語で記述したスタートアッププログラムが必要です。
機械語ファイルができるまでのツールチェーンを示します。



1.1.3　わかりやすいプログラム

Ｃ言語のプログラムはフリーフォーマット形式なので、ある一定の規則を守ればあとは自由にプログラムを記述できます。しかし、プログラムは読みやすく、かつ保守性の高いものにする必要があり、そのためにはいつ誰が見てもそのプログラムを理解できるように記述しなければなりません。

この項では、「わかりやすいプログラム」を記述するためのポイントを説明します。

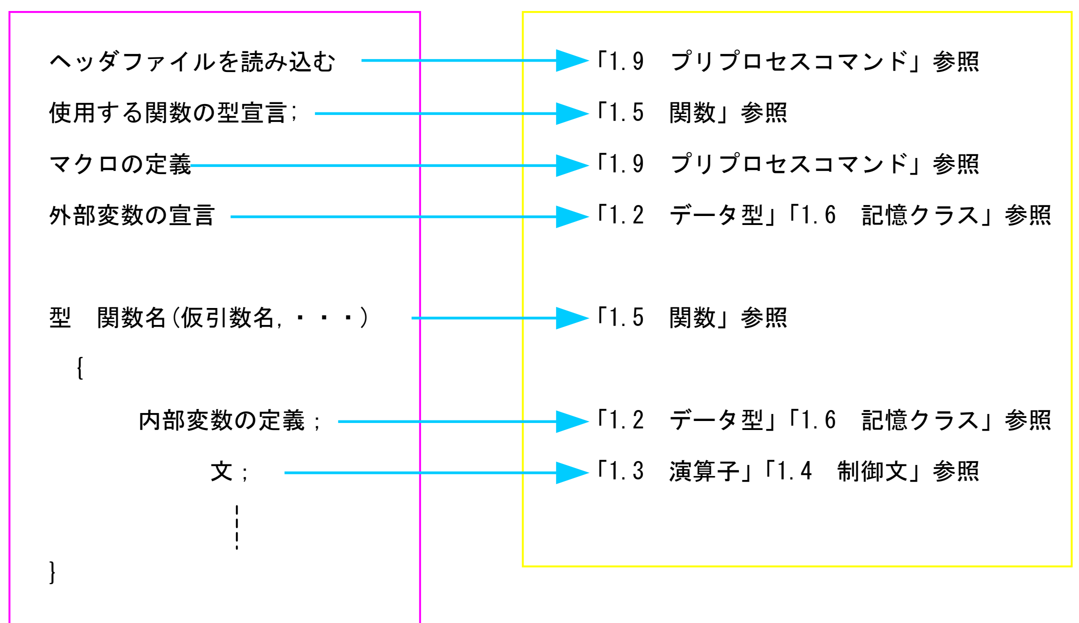
Ｃ言語の規則

Ｃ言語プログラムを記述するにあたって、守らなければならない規則は次の５項目です。

(1) プログラムの実行文は「；」で区切る。
(2) 関数や制御文の実行単位は「{」「}」で囲む。
(3) 関数や変数は型宣言が必要である。
(4) 予約語は識別子(関数名、変数名など)に使用できない。
(5) コメントは「/*　コメント　*/」又は、「//　コメント」(C++形式)で記述する。後者で記述する場合、「-K」オプションが必要。

Ｃ言語のソースファイルの構成

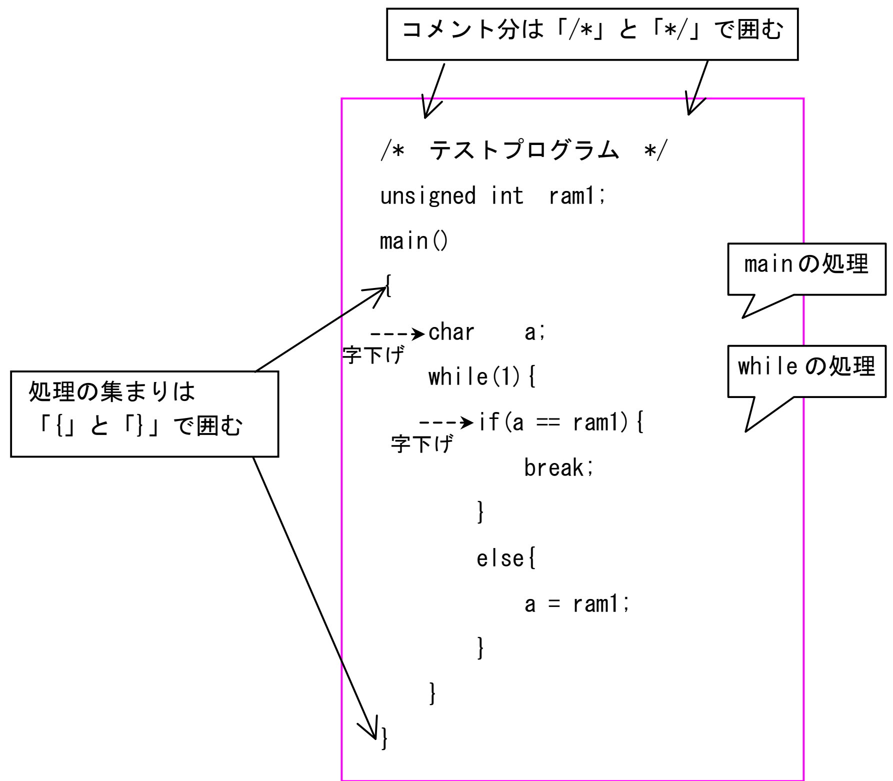
一般的なＣ言語のソースファイルの構成を以下にまとめます。各項目については矢印で示す節を参照してください。



プログラミングスタイル

プログラムの保守性を高めるためには、開発者の間でプログラムリストのテンプレートを定めます。これを「プログラミングスタイル」として共有し、誰もが読みやすく、誰もがメンテナンスできるソースプログラムを目指します。プログラミングスタイルの一例を示します。

(1) 機能ごとに関数にする。
(2) １つの関数内の処理を必要以上に大きくしない(50 行前後が目安)。
(3) １つの行に複数の実行文は書かない。
(4) 処理ブロックごとに字下げを行う(通常は 4 タブを使用)。
(5) コメント文を効果的に記述してプログラムの流れを明確にする。
(6) プログラムを複数のソースファイルから作成する場合は、共通部分を独立した別のファイルにして共有する。



コメント文の記述方法

コメント文の記述方法も、読みやすいプログラムを書くための重要なポイントになります。ヘッダとしてファイルや関数の機能を明示したり、プログラムの流れを明確にします。

ファイルヘッダの例

```
/* ""FILE COMMENT"" ************************************************************************
  * System Name       :テストプログラム
  * File Name         :TEST.C
  * Version           :1.00
  * Contents          :テストプログラム
  * Customer          :………………
  * Model             :………………
  * Order             :………………
  * CPU               :M38039MC-XXXFP
  * Compiler          :M3T-ICC740 (Ver.1.00)
  * Programmer        :XXXX
  * Note              :このファイルに含まれるモジュールは再利用可能なように設計されている。
  ****************************************************************************************
  * Copyright,XXXX xxxxxxxxxxxxxxxxx CORPORATION
  ****************************************************************************************
  * History           :XXXX.XX.XX          :Start
  * ""FILE COMMENT END"" *****************************************************************/

  /* ""プロトタイプ宣言"" **************************************************************/
  void main (void);
  void key_in (void);
  void key_out (void);
```

関数ヘッダの例

```
  /* ""FUNC COMMENT"" *********************************************************************
  * ID                :1.
  * モジュール概要      :メイン関数
  * ----------------------------------------------------------------------
  * Include           :"system.h"
  * ----------------------------------------------------------------------
  * 宣言               :void main (void)
  * ----------------------------------------------------------------------
  * 機能               : 全体の制御
  * ----------------------------------------------------------------------
  * 引数               :void
  * ----------------------------------------------------------------------
  * 戻り値             :void
  * ----------------------------------------------------------------------
  * 入力               : なし
  * 出力               : なし
  * ----------------------------------------------------------------------
  * 使用関数            :void  key_in (void)          :入力関数
  *                   :void  key_out (void)         :出力関数
  * ----------------------------------------------------------------------
  * 注意事項            :特になし
  * ----------------------------------------------------------------------
  * History           :XXXX.XX.XX                   :Start
  /* ""FUNC COMMENT END"" *******************************************************************/
  #include    "system.h"
  void  main (void)
  {
    while(1){         /*  無限ループ   */
        key_in();         /*  入力処理 */
        key_out();    /*  出力処理 */
    }
  }
```

コラム　ICC740 の予約語

次に示す語は ICC740 の予約語になっていますので、変数名や関数名には使用できません。

__asm ※	do	int	short	unsigned
auto	double	interrupt ※	signed	void
bit ※	else	long	sizeof	volatile
break	enum	monitor ※	static	while
case	extern	no_init ※	struct	zpage ※
char	float	npage ※	switch	
const	for	register	tiny_func ※	
continue	goto	return	typedef	
default	if	sfr ※	union	

※　「-e」オプション使用時、予約語になります。

1.2　データ型

1.2.1　C言語で扱える“定数”

C 言語では「整数」、「実数」、「１文字」、「文字列」の４種類の定数を扱うことができます。
この項では、それぞれの定数を使用するときの記述方法と注意点をまとめます。

整数定数

整数定数は 10 進数、16 進数および 8 進数の 3 種類の方法で記述できます。それぞれの表記方法を示します。また、定数データの場合は大文字と小文字を区別しません。

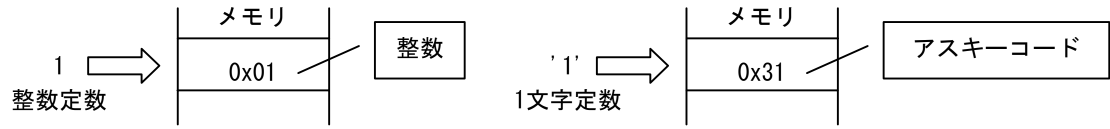
種類	表記方法	例
10 進数	通常の数学表記(何もつけない)	127, +127, -56
16 進数	数字の前に「0x(ゼロ・エックス)または「0X」を付ける	0x3b, 0x3B
8 進数	数字の前に「0(ゼロ)を付ける	07, 041

実数定数(浮動小数点定数)

符号付きの実数を 10 進数で表記したものを「浮動小数点定数」といいます。表記方法としては通常の小数点表記と、「ｅ」または「Ｅ」を用いた指数表記が記述できます。
① 通常の小数点表記　　　　　　　　例：175.5,-0.007
② 「e」または「E」を用いた指数表記　　例：1.755e2,-7.0E-3

1 文字定数

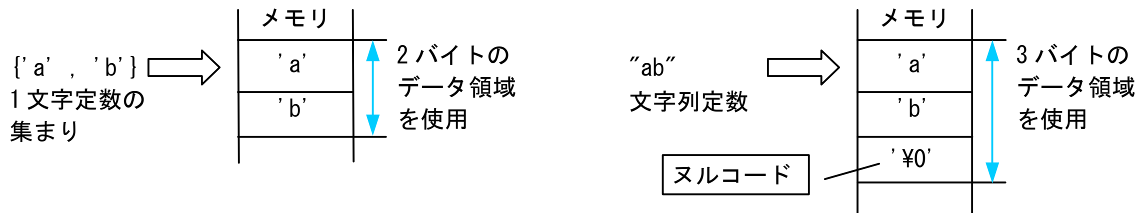
１文字定数はシングルクォーテーション(')で囲みます。英数字のほかに、制御コードも１文字定数として扱うことができます。以下に示すように内部的にはすべてアスキーコードとして扱われます。



文字列定数

英数字や制御コードの並びをダブルクォーテーション(")で囲むと文字列定数として扱うことができます。
文字列定数では、データの最後にヌルコード'¥0'が自動的に付けられ、文字列の終わりであることを表します。

例："abc", "012¥n", "Hello!"



コラム　制御コード一覧(エスケープシーケンス)

C言語のプログラムでよく使う制御コード(エスケープシーケンス)を紹介します。

表記	内容
¥f	改ページ(FF)
¥n	改行復帰(NL)
¥r	復帰(CR)
¥t	水平タブ(HT)
¥¥	¥記号
¥'	シングルクォーテーション
¥"	ダブルクォーテーション
¥x 定数値	16進数
¥定数値	8進数
¥0	ヌルコード

1.2.2 変数

C 言語のプログラムの中で変数を使用する前には、必ず変数の「データ型」を宣言しなければなりません。データ型は、その変数に割り当てるメモリサイズと、扱う数値の範囲から決めます。

この項では、ICC740 で扱える変数のデータ型と宣言方法について説明します。

ICC740 の基本データ型

ICC740 で扱えるデータ型を示します。()内は省略して記述できます。

	データ型	ビット長	表現できる数値の範囲
整数	char	8 ビット	0～255
	unsigned char		0～255
	signed char		-128～127
	unsigned short (int)	16 ビット	0～65535
	(signed) short (int)		-32768～32767
	unsigned int	16 ビット	0～65535
	(signed) int		-32768～32767
	unsigned long (int)	32 ビット	0～4294967295
	(signed) long (int)		-2147483648～2147483647
実数	float	32 ビット	有効桁数 9 桁
	double	32 ビット	有効桁数 9 桁
	long double	32 ビット	有効桁数 9 桁

※「-c」オプション使用時、char 型は signed char と等価になるため、表現できる数値の範囲も -128～127 になります。

変数の宣言と定義

変数の宣言や定義は、「データ型　変数名;」という書式で行います。

例：変数 a を char 型として宣言する。

```
char  a;
```

「データ型　変数名　＝　初期値;」と記述すると、定義と同時にその変数に対して初期値を設定することができます。

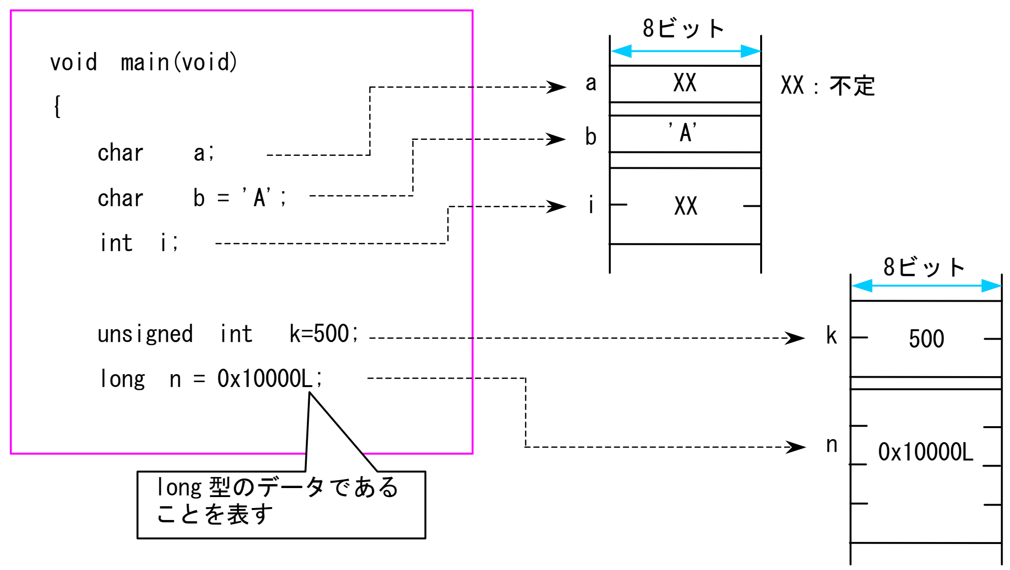
例：char 型の変数 a に初期値として'A'を設定する。

```
char  a = 'A';
```

また複数の変数名をカンマ(',')で区切って列記すると、同じデータ型の変数を同時に宣言・定義できます。

例：int　i,j;

例：int　i=1,　j=2;



1.2.3　データの特性

変数や定数を宣言するときにデータ型と合わせて、そのデータの特性を記述できます。
この項では、ICC740 で扱うデータの特性と指定方法を説明します。

符号付きか、符号なしかを明示する(signed/unsigned)

そのデータが符号を持つ場合"signed"を、符号を持たない場合は"unsigned"を記述します。宣言時にこれらの型を記述しない場合、ICC740 では char 型データのみ符号なし、それ以外のデータはすべて符号付きデータとなります。

```
void  main(void)
{
    char    a;          ← "unsigned char  a;"と同義
    signed  char    s_a;

    int  b;             ← "signed int  b;"と同義
    unsigned  int   u_b;
              ┆
}
```

※　「-c」オプション使用時、char 型は signed char と等価になります。

定数データであることを明示する(const 修飾子)

プログラムを実行しても全く値が変化しないデータについては宣言時に"const"を記述します。プログラム中にこの定数データを変化させるような記述があると、ICC740 はエラーを出力します。

```
void  main(void)
{
    char  a = 10;
    const signed  char  c_a=20;

    a = 5;
    c_a = 5;            ← エラー発生！
}
```

コンパイラによる最適化を禁止する(volatile 修飾子)

ICC740 はプログラム処理上意味のない命令については最適化を行い、不要な命令コードを生成しません。しかしデータによってはプログラムの処理とは関係なく、割り込みやポートからの入力によって変化するものもあります。このようなデータの宣言時には"volatile"を記述します。この型修飾子を記述したデータについては、ICC740 は最適化を行わず、命令コードを出力します。

```
char port1;
char volatile port2;

void func(void)
{
        port1 = 0;
        port2 = 0;
        if( port1 == 0 ){
              ¦
        }
        if( port2 == 0 ){
              ¦
        }
}
```

データの宣言時に"volatile"を記述していないため、最適化によって比較が取り除かれ、コードを出力しない

データの宣言時に"volatile"を記述しているため、最適化を行わず、コードを出力する

コラム　宣言の構文

データを宣言するときに型とともに、データの特性をいろいろな指定子や修飾子を使って記述します。以下に宣言の構文を示します。

宣言指定子			宣言子
記憶クラス指定子	型修飾子	型指定子	
static	const	char	
register	volatile	short	
auto		int	
extern		long	
typedef		float	
		struct	データ名
		union	
		enum	
		void	
		signed	
		unsigned	

1.3　演算子

1.3.1　ICC740の演算子

ICC740 ではプログラムを記述するために様々な演算子を用意しています。
以下の項では、用途別にこれらの演算子(ただし、アドレス演算子とポインタ演算子は除く)の記述方法と使用する上での注意点を説明します。

ICC740 で使用できる演算子

ICC740 で使用できる演算子を示します。

<table>
<tr><td>単項算術演算子</td><td>++　--　+　-</td></tr>
<tr><td>二項算術演算子</td><td>+　-　*　/　%</td></tr>
<tr><td>シフト演算子</td><td><<　>></td></tr>
<tr><td>ビット演算子</td><td>&　｜　^　~</td></tr>
<tr><td>関係演算子</td><td>>　<　>=　<=　==　!=</td></tr>
<tr><td>論理演算子</td><td>&&　||　!</td></tr>
<tr><td>代入演算子</td><td>=　+=　-=　*=　/=　%=　<<=　>>=　&=　|=　^=</td></tr>
<tr><td>条件演算子</td><td>?:</td></tr>
<tr><td>sizeof 演算子</td><td>sizeof</td></tr>
<tr><td>キャスト演算子</td><td>(型)</td></tr>
<tr><td>アドレス演算子</td><td>&</td></tr>
<tr><td>ポインタ演算子</td><td>*</td></tr>
<tr><td>カンマ演算子</td><td>,</td></tr>
</table>

1.3.2　数値計算のための演算子

数値計算のために使用される主な演算子は、計算を行う「算術演算子」と結果を格納する「代入演算子」です。この項では、算術演算子と代入演算子を説明します。

単項算術演算子

「単項算術演算子」は、１変数に対して１つの答えを返します。

演算子	記述形式	内容
++	++変数(前置式) 変数++(後置式)	式の値をインクリメントする
--	--変数(前置式) 変数--(後置式)	式の値をディクリメントする
+	+式	式の値を返す
-	-式	式の値の符号を反転した値を返す

インクリメント演算子(++)やディクリメント演算子(--)を、代入演算子や関係演算子と組み合わせて使用するとき、前置式で記述するか後置式で記述するかにより演算結果が変わることがあります。

＜例＞

前置式；インクリメントまたはディクリメントしてから代入します。

b = ++a; ⟶ a = a+1 ; b = a;

後置式；代入してからインクリメントまたはディクリメントします。

b = a++; ⟶ b = a ; a = a+1;

二項算術演算子

通常の四則演算のほかに、整数÷整数の「剰余(あまり)」を求める演算もできます。

演算子	記述形式	内容
+	式１ + 式２	式１の値と式２の値を加算した結果を返す
-	式１ - 式２	式１の値から式２の値を減算した結果を返す
*	式１ * 式２	式１の値と式２の値を乗算した結果を返す
/	式１ / 式２	式１の値を式２の値で除算した結果を返す
%	式１ % 式２	式１の値を式２の値で割った剰余を返す

代入演算子

「式 1＝式 2」で、式 2 の値を式 1 へ代入します。また、代入演算子'='は前述の算術演算子や後述のビット演算子、シフト演算子とも組み合わせて記述できます(「複合代入演算子」)。このとき、必ず代入演算子'='を右側に記述します。

演算子	記述形式	内容
=	式 1 = 式 2	式 2 の値を式 1 へ代入する
+=	式 1 += 式 2	式 1 の値と式 2 の値を加算し、式 1 へ代入する
-=	式 1 -= 式 2	式 1 の値から式 2 の値を減算し、式 1 へ代入する
*=	式 1 *= 式 2	式 1 の値と式 2 の値を乗算し、式 1 へ代入する
/=	式 1 /= 式 2	式 1 の値を式 2 の値で除算し、式 1 へ代入する
%=	式 1 %= 式 2	式 1 の値を式 2 の値で割った剰余を式 1 へ代入する
<<=	式 1 <<= 式 2	式 1 の値を式 2 の値だけ左シフトし、式 1 へ代入する
>>=	式 1 >>= 式 2	式 1 の値を式 2 の値だけ右シフトし、式 1 へ代入する
&=	式 1 &= 式 2	式 1 の値と式 2 の値のビット論理積を式 1 へ代入する
|=	式 1 |= 式 2	式 1 の値と式 2 の値のビット論理和を式 1 へ代入する
^=	式 1 ^= 式 2	式 1 の値と式 2 の値のビット排他的論理和を式 1 へ代入する

コラム　暗黙の型変換

ICC740 では、型の違うデータ間で演算を行うとき、以下の規則に従って「暗黙の型変換」を行います。
・ビット長が長いデータの型に合わせたのち、演算します。
・代入するときは、左辺の型に合わせます。

```
char    byte = 0x12;
int     word = 0x3456;
```

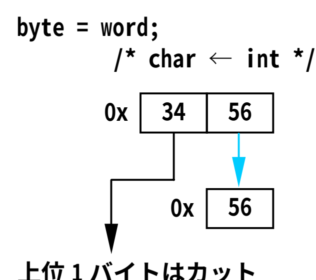
のとき・・・

```
word = byte;
        /* int ← char */
```

0x 12

0x 00 12

0x00 が拡張



上位 1 バイトはカット

1.3.3　データ加工のための演算子

データ加工によく使用される演算子は「ビット演算子」と「シフト演算子」です。
この項では、ビット演算子とシフト演算子を説明します。

ビット演算子

ビット演算子を使用すると、データのマスク処理やアクティブ変換を行うことができます。

<table>
<tr><th>演算子</th><th>記述形式</th><th>内容</th></tr>
<tr><td>&</td><td>式 1 & 式 2</td><td>式 1 の値と式 2 の値のビットごとの論理積を返す</td></tr>
<tr><td>|</td><td>式 1 | 式 2</td><td>式 1 の値と式 2 の値のビットごとの論理和を返す</td></tr>
<tr><td>^</td><td>式 1 ^ 式 2</td><td>式 1 の値と式 2 の値のビットごとの排他的論理和を返す</td></tr>
<tr><td>~</td><td>~式 1</td><td>式 1 の値のビット反転を返す</td></tr>
</table>

シフト演算子

シフト動作だけでなく、簡単な乗除算にも利用できます(「コラム シフト演算子を使った乗除算」参照)。

演算子	記述形式	内容
<<	式 1 << 式 2	式 1 の値を式 2 の値だけ左シフトした値を返す
>>	式 1 >> 式 2	式 1 の値を式 2 の値だけ右シフトした値を返す

算術シフトと論理シフトの比較

右シフトを実行するとき、対象データが符号付きか符号なしかでシフト動作が異なります。
・符号なしのとき→「論理シフト」：最上位ビットに'0'を挿入します。
・符号付きのとき→「算術シフト」：符号を保持するようにシフト動作をします。つまり、正の数のときは'0'を、負の数のときは'1'を最上位ビットから挿入します。

	<符号なし> unsigned int i = 0xFC18 (i= 64520)	<負の数> signed int i = 0xFC18 (i= -1000)		<正の数> signed int i = 0x03E8 (i= +1000)	
	1111 1100 0001 1000	1111 1100 0001 1000		0000 0011 1110 1000	
i >> 1	0111 1110 0000 1100	1111 1110 0000 1100	(-500)	0000 0001 1111 0100	(+500)
i >> 2	0011 1111 0000 0110	1111 1111 0000 0110	(-250)	0000 0000 1111 1010	(+250)
i >> 3	0001 1111 1000 0011	1111 1111 1000 0011	(-125)	0000 0000 0111 1101	(+125)
	論理シフト	算術シフト (符号が保持される)			

コラム　シフト演算子を使った乗除算

シフト演算子を使って簡単な乗除算ができます。通常の乗除算演算子を使用するよりも、演算速度が速くなります。ICC740 ではこの点を考慮し、"*2"、"*4"、"*8"などに対しては乗算命令ではなくシフト命令を生成します。

・乗算：シフト演算で行います。
a*2 → a<<1
a*4 → a<<2
a*8 → a<<3

・除算：下位ビットから押し出されたデータを検出することにより、剰余を知ることもできます。
a/4 → a>>2
a/8 → a>>3
a/16 → a>>4

1.3.4　条件を調べるための演算子

　制御文の中で条件を調べるために使用するのが「関係演算子」と「論理演算子」です。どちらの演算子も、条件が成立しているときは'1'、条件が成立していないときには'0'を返します。
　この項では、関係演算子と論理演算子を説明します。

関係演算子

2つの式の間の大小関係を調べます。そして結果が真ならば'1'を、偽ならば'0'を返します。

演算子	記述形式	内容
<	式1 < 式2	式1の値が式2の値より小さければ真、それ以外は偽
<=	式1 <= 式2	式1の値が式2の値より小さいか等しければ真、それ以外は偽
>	式1 > 式2	式1の値が式2の値より大きければ真、それ以外は偽
>=	式1 >= 式2	式1の値が式2の値より大きいか等しければ真、それ以外は偽
==	式1 == 式2	式1の値が式2の値と等しければ真、それ以外は偽
!=	式1 != 式2	式1の値が式2の値と等しくなければ真、それ以外は偽

論理演算子

関係演算子とともに用いる演算子で、複数の条件式の組み合わせ条件を調べます。

<table>
<tr><th>演算子</th><th>記述形式</th><th>内容</th></tr>
<tr><td>&&</td><td>式1 && 式2</td><td>式1と式2の両方が真ならば真、それ以外は偽</td></tr>
<tr><td>||</td><td>式1 || 式2</td><td>式1と式2の両方が偽ならば偽、それ以外は真</td></tr>
<tr><td>!</td><td>!式</td><td>式が真ならば偽、偽ならば真</td></tr>
</table>

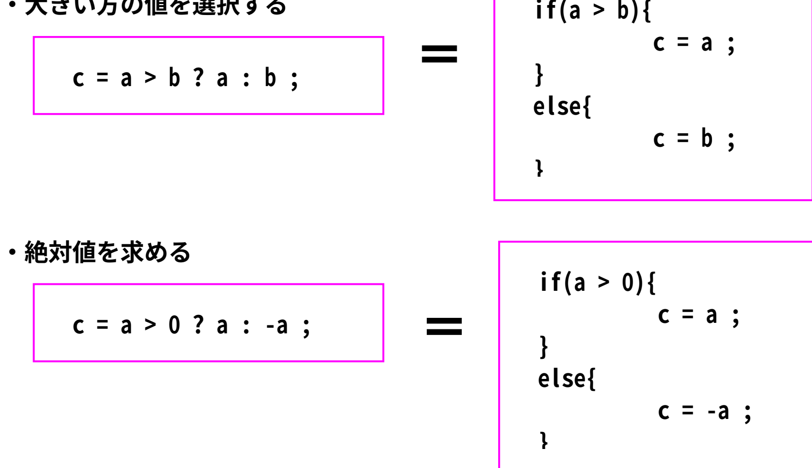
1.3.5　その他の演算子

この項では、C 言語ならではのちょっと変わった性質を持つ 6 種類の演算子を説明します。

条件演算子

条件式が真ならば式 1 を、偽ならば式 2 を実行する演算子です。条件式、式 1、式 2 ともに処理の記述が短いときに使用すると、条件分岐のコーディングを簡単にできます。条件演算子と利用例を以下に示します。

演算子	記述形式	内容
? :	条件式 ? 式 1 : 式 2	条件式が真ならば式 1 を、偽ならば式 2 を実行する



sizeof 演算子

あるデータ型または式が使用しているメモリのバイト数を知りたいときに使用します。

演算子	記述形式	内容
sizeof	sizeof 式 sizeof(データ型)	式またはデータ型のメモリ使用量をバイト単位で返す

キャスト演算子

異なる型どうしで演算を行うと、演算で使用するデータは暗黙のうちに式中のいちばん大きなデータ型へと変換されます。しかし、これが思わぬ不具合の原因となる可能性があるので、「キャスト演算子」を利用して型変換を明示します。

演算子	記述形式	内容
(型)	(新データ型)変数	変数のデータ型を新データ型に変換する

アドレス演算子

変数が割り付けられている記憶領域のアドレス値を返します。変数の部分は配列要素でもよく、その場合、要素番号が示す位置のアドレスがその値となります。

演算子	記述形式	内容
&	&変数	変数のアドレス値を返す

ポインタ演算子

ポインタ変数が示す記憶領域の内容を指定します。

演算子	記述形式	内容
*	*変数	ポインタ変数が示す記憶領域の内容を指定する

カンマ(順次)演算子

式 1 から式 2 へと、左から順に実行していきます。短い記述の処理を羅列するときに使用します。

演算子	記述形式	内容
,	式 1 , 式 2	式 1、式 2 と左から順に実行する

1.3.6　演算子の優先順位

数学の演算子と同じようにＣ言語で使用する演算子にも「優先順位」と「結合規則」があります。
この項では、演算子の優先順位と結合規則について説明します。

優先順位と結合規則

１つの式の中に複数の演算子が含まれているとき、まず「優先順位」の高いものから演算していきます。
「結合規則」は同じ優先順位の演算子が複数存在するとき、左右どちらから計算するのかを示しています。

<table>
<tr><th>優先順位</th><th>演算子の種類</th><th>演算子</th><th>結合規則</th></tr>
<tr><td>高</td><td>式</td><td>()　[]　->　.(注1)</td><td>→</td></tr>
<tr><td>↑</td><td>単項算術演算子 etc</td><td>!　~　++　--　+　-　*(注2)　&(注3)　(型)　sizeof</td><td>←</td></tr>
<tr><td></td><td>乗除算演算子</td><td>*(注4)　/　%</td><td>→</td></tr>
<tr><td></td><td>加減算演算子</td><td>+　-</td><td>→</td></tr>
<tr><td></td><td>シフト演算子</td><td><<　>></td><td>→</td></tr>
<tr><td></td><td>関係演算子(比較)</td><td><　<=　>　>=</td><td>→</td></tr>
<tr><td></td><td>関係演算子(等価)</td><td>==　!=</td><td>→</td></tr>
<tr><td></td><td>ビット演算子（AND)</td><td>&</td><td>→</td></tr>
<tr><td></td><td>ビット演算子（EOR)</td><td>^</td><td>→</td></tr>
<tr><td></td><td>ビット演算子（OR)</td><td>|</td><td>→</td></tr>
<tr><td></td><td>論理演算子（AND)</td><td>&&</td><td>→</td></tr>
<tr><td></td><td>論理演算子（OR)</td><td>||</td><td>→</td></tr>
<tr><td></td><td>条件演算子</td><td>?:</td><td>←</td></tr>
<tr><td>↓</td><td>代入演算子</td><td>=　+=　-=　*=　/=　%=　&=　^=　|=　<<=　>>=</td><td>←</td></tr>
<tr><td>低</td><td>カンマ演算子</td><td>,</td><td>→</td></tr>
</table>

(注1)　'.'は構造体と共用体のメンバを指定するメンバ演算子です。
(注2)　'*'はポインタ変数を表すポインタ演算子です。
(注3)　'&'は変数のアドレスを表すアドレス演算子です。
(注4)　'*'は乗算を表す算術演算子です。

1.3.7　間違いやすい演算子の使用例

演算子の「暗黙の型変換」や「優先順位」の解釈を間違うと、プログラムが期待通りに動作しません。
この項では間違いやすい演算子の使用例とその対処方法について説明します。

「暗黙の型変換」の解釈間違いと対処方法

型の違うデータ間で演算を行うとき、次のような「暗黙の型変換」が行われます。
(A)　ビット長が長いデータ型に型を合わせた後で演算が行われます。
(B)　int 型より短いデータ型は全て、算術演算で使用された場合には int 拡張されます。
(C)　定数は int 扱いになります。
期待通りに動作させるためには、キャスト演算子を使用した明示的な型変換で対応します。
まず、期待通りに動作する(if 文が真になる)例を示します。

(1)

```
unsigned char a,b;
a = 0;
b = 5;
if( (unsigned char)(a - 1) >= b){
        ¦
        ¦
}
else{
        ¦
        ¦
}
```

式(a-1)全体に対してキャスト演算子を使用しているため、
左辺は unsigned 0x00 - unsigned 0x01 = unsigned 0xFF
すなわち 255 となる。
したがって 255 >= 5 が比較されるため、真と判断する。

次に、期待通りには動作しない(if 文が偽になる)例を示します。

(2)

```
unsigned char a,b;
a = 0;
b = 5;
if( (a - 1) >= b ){
        ¦
        ¦
}
else{
        ¦
        ¦
}
```

式(a-1)は unsigned char - signed int の式になる。
「暗黙の型変換」によって unsigned char を signed int に型変換し、
左辺は signed 0x0000 - signed 0x0001 = signed 0x0000 + signed 0xFFFF = signed 0xFFFF
すなわち-1 となる。
したがって-1 >= 5 が比較されるため、偽と判断する。

(3)

```
unsigned char a,b;
a = 0;
b = 5;
if( ( a - (unsigned char)1 ) >= b ){
        ¦
        ¦
}
else{
        ¦
        ¦
}
```

定数 1 を unsigned char へ型変換しているので、
左辺は unsigned char - unsigned char の式になるが、
(B)の int 拡張によって signed int に型変換し、
上記(2)と同様の演算を行う。
したがって-1 >= 5 が比較されるため、偽と判断する。

なお、最終的にはリストファイルやアセンブリファイルで確認してください。記述の仕方により、プログラムサイズが変わったり、ランタイムライブラリが呼ばれたりすることがあります。

「優先順位」の解釈間違いと対処方法

１つの式の中に複数の演算子が含まれる場合は、演算子の「優先順位」と「結合規則」を正しく解釈する必要があります。また、期待通りに動作させるためには、式の"()"を使用して対応します。

```
int a = 5;
if(a & 0x10 == 0){
        ¦
}
else{
        ¦
}
```

ビット演算子"&"と関係演算子"=="では
関係演算子"=="の方が優先順位が高いため
0x10 == 0 の比較結果(偽 : 0)と変数 a との AND 演算を行うため
if 文の条件式の演算結果は常に偽となってしまう。

そこで

```
int a = 5;
if((a & 0x10) == 0){
        ¦
}
else{
        ¦
}
```

a & 0x10 の演算を優先させるために、
式の"()"をつけて演算を優先させる。

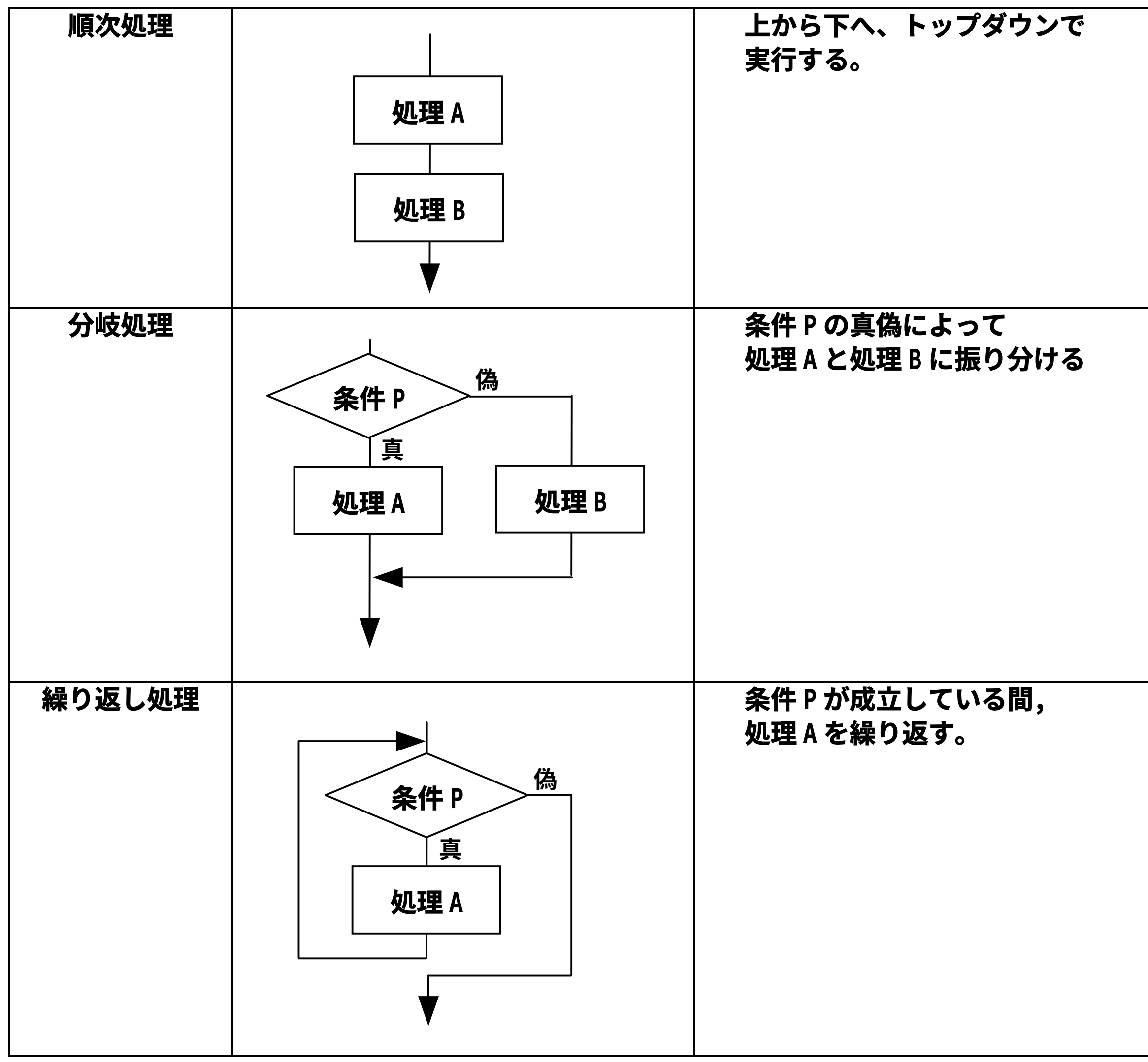
1.4　制御文

1.4.1　プログラムの構造化

C 言語では構造化プログラミングの基本処理「順次処理」、「分岐処理」、「繰り返し処理」をすべて制御文を使用して記述できます。したがって、C 言語で記述されたプログラムはすべて構造化されており、これが処理の流れが読みやすい理由です。
以下の項では、これらの制御文の記述方法と使用例を説明します。

プログラムの構造化

プログラムをわかりやすくするための最大のポイントは「いかにプログラムの流れを読みやすくするか」ということです。そこでプログラムの流れを好き勝手にするのではなく、「順次処理」、「分岐処理」、「繰り返し処理」の 3 つに限定しようという動きが盛んになりました。これが「構造化プログラミング」と呼ばれる手法です。構造化プログラミングの 3 つの基本形を示します。



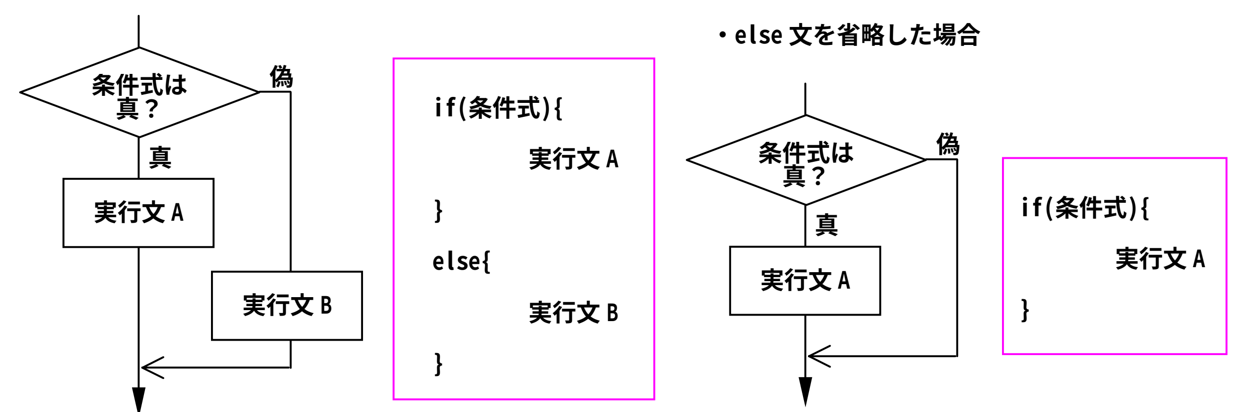
順次処理		上から下へ、トップダウンで実行する。
分岐処理		条件 P の真偽によって処理 A と処理 B に振り分ける
繰り返し処理		条件 P が成立している間，処理 A を繰り返す。

1.4.2　条件による処理の分岐(分岐処理)

分岐処理を記述するための制御文は、「if-else 文」、「else-if 文」、「switch-case 文」です。
この項では、これらの制御文の記述方法と使用例を説明します。

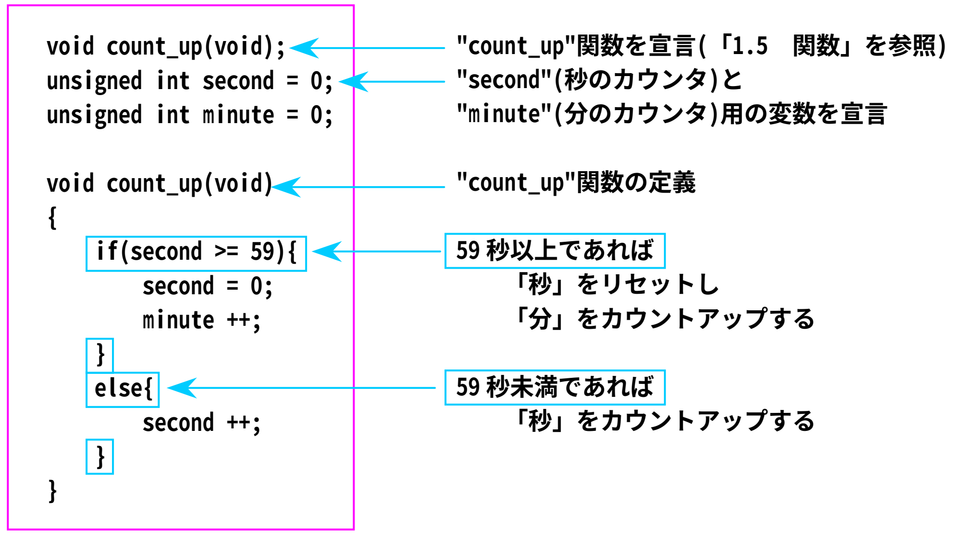
if-else 文

ある条件が真の場合はすぐ次のブロックを、偽であれば「else」のブロックを実行します。
"else"のブロックは省略できます。



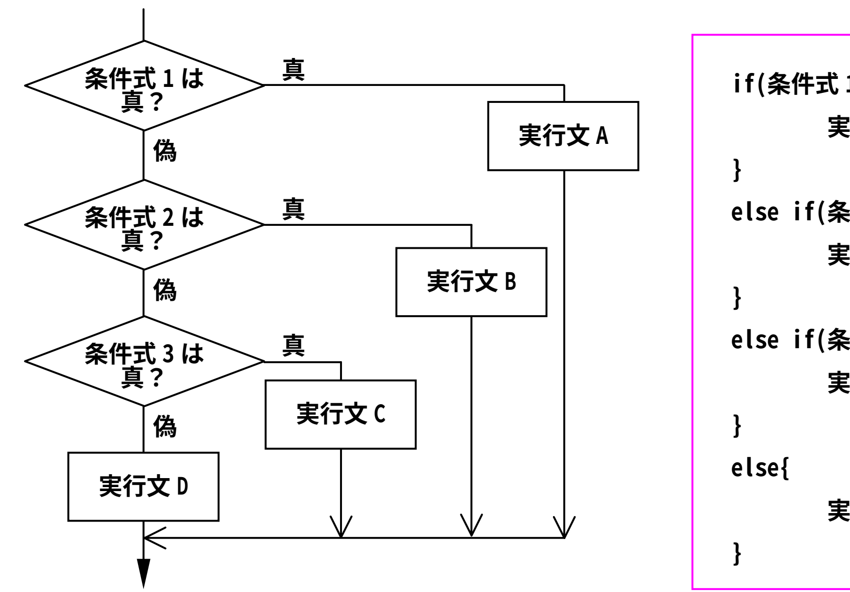
カウントアップ(if-else 文の記述例)

秒のカウンタ"second"と分のカウンタ"minute"をカウントアップします。このモジュールを 1 秒間隔で呼び出すと、時計の役割をはたします。



else-if 文

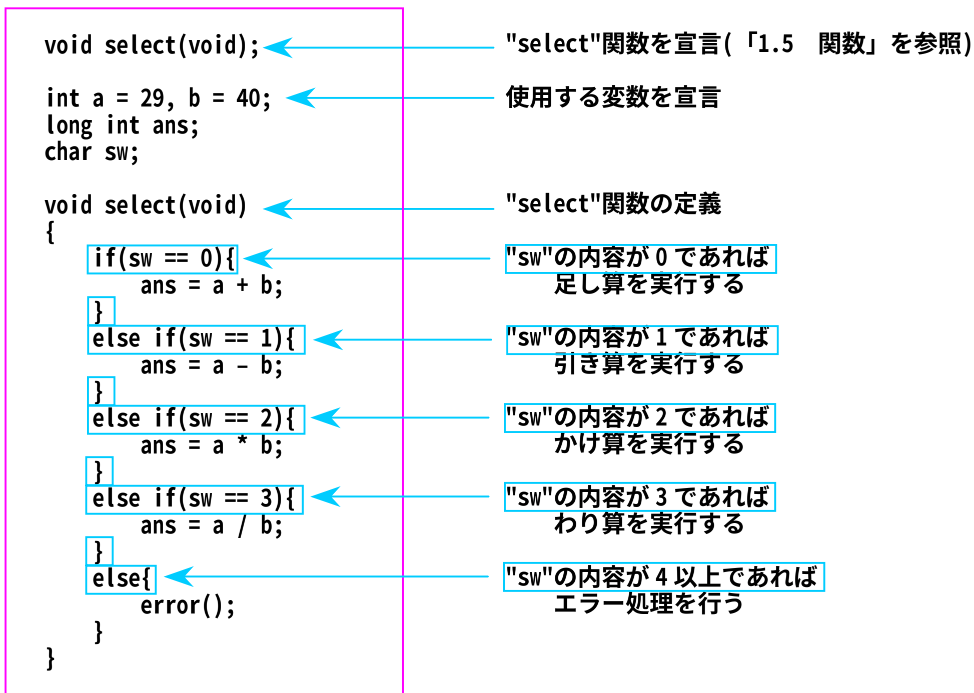
複数の条件により３個以上の処理に分けたいときに使用します。各条件が真のときに実行したい処理をすぐ次のブロックに記述します。すべての条件に当てはまらなかったときの処理を最後の"else"のブロックに記述します。




```
if(条件式１){
        実行文Ａ
}
else if(条件式２){
        実行文Ｂ
}
else if(条件式３){
        実行文Ｃ
}
else{
        実行文Ｄ
}
```

四則演算の切り換え-1-(else-if 文の記述例)

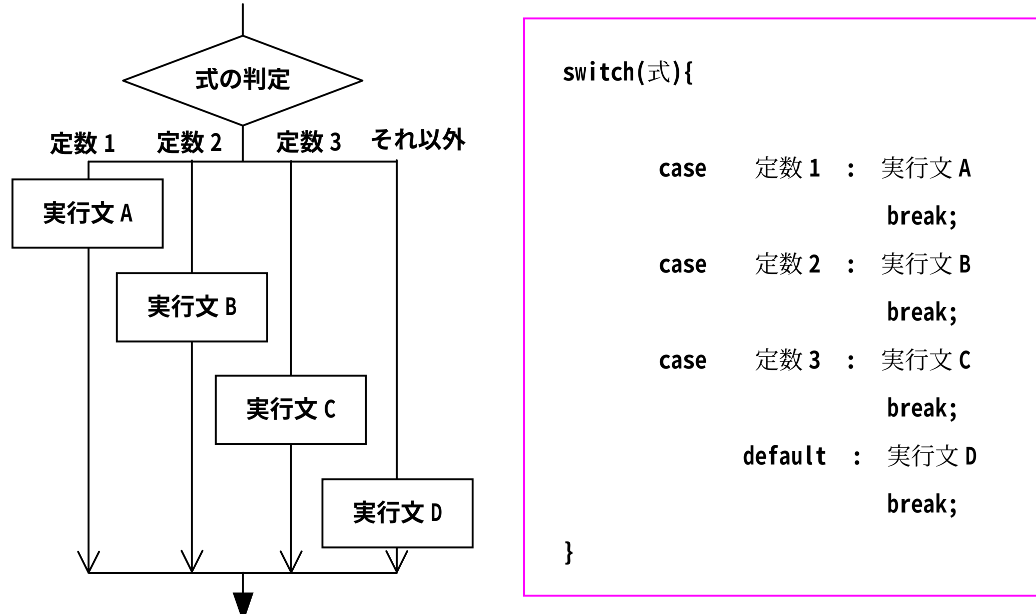
入力データ"sw"の内容により、実行する演算を切り換えます。



```
void select(void);

int a = 29, b = 40;
long int ans;
char sw;

void select(void)
{
    if(sw == 0){
        ans = a + b;
    }
    else if(sw == 1){
        ans = a - b;
    }
    else if(sw == 2){
        ans = a * b;
    }
    else if(sw == 3){
        ans = a / b;
    }
    else{
        error();
    }
}
```

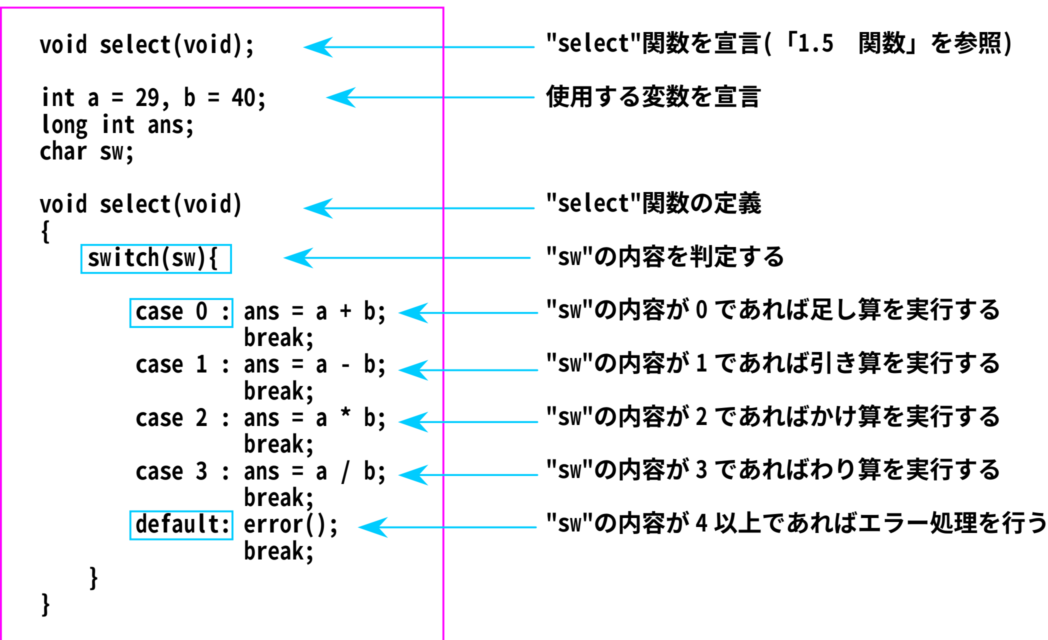
switch-case 文

ある式の結果により複数の処理に分岐します。式の結果は定数として判定しますので、関係演算子などは使えません。



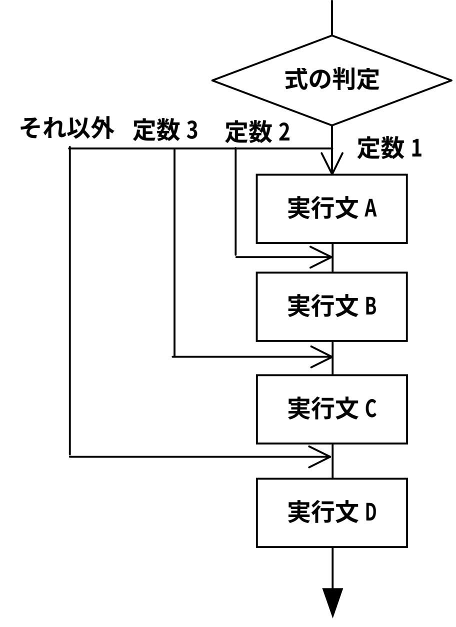
四則演算の切り換え-2-(switch-case 文の記述例)

入力データ"sw"の内容により、実行する演算を切り換えます。



コラム　break のない switch-case 文

switch-case 文の各実行文の終わりには通常 break 文を記述します。
break 文を記述していないブロックがあれば、そのブロックを終了すると次のブロックへと上から順にブロックを実行していきます。つまり、式の値により処理のスタート位置を変えることができます。



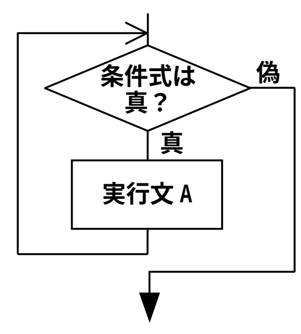

```
switch(式){

        case    定数 1  :  実行文 A

        case    定数 2  :  実行文 B

        case    定数 3  :  実行文 C

                default  :  実行文 D

}
```

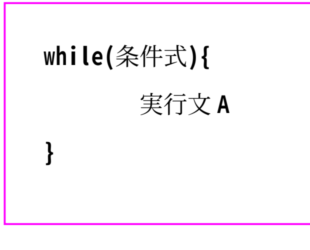
1.4.3　同じ処理の繰り返し(繰り返し処理)

繰り返し処理を記述するための制御文は「while 文」、「for 文」、「do-while 文」です。
この項では、これらの制御文の記述方法と使用例を説明します。

while 文

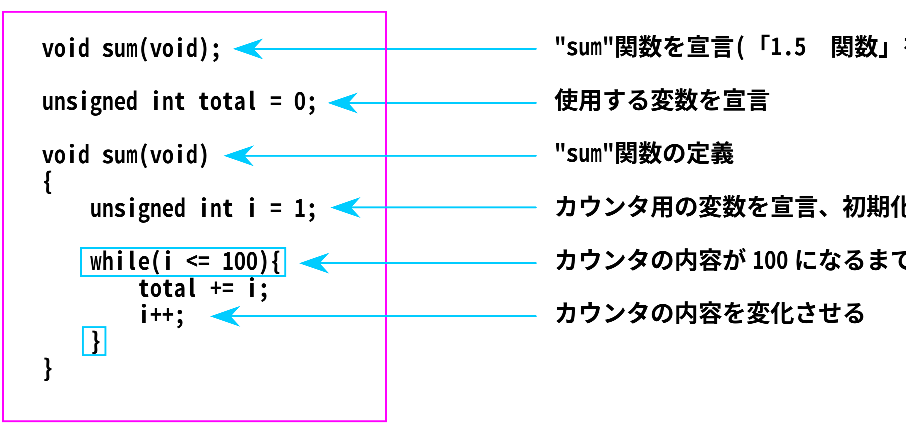
条件式が成立している間、ブロック内の処理を繰り返し実行します。条件式に０以外の定数を記述しておくと、条件式が常に「真」となり、無限ループを実現できます。





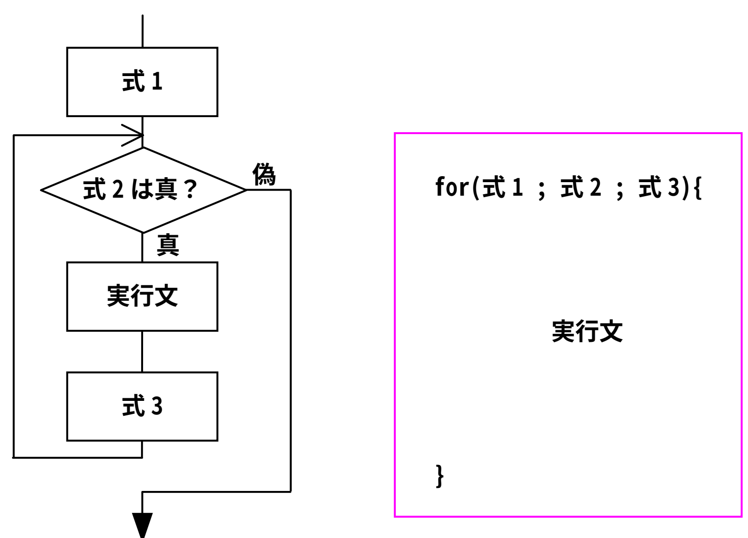
総和を求める-1-(while 文の記述例)

１から 100 までの整数の和を求めます。



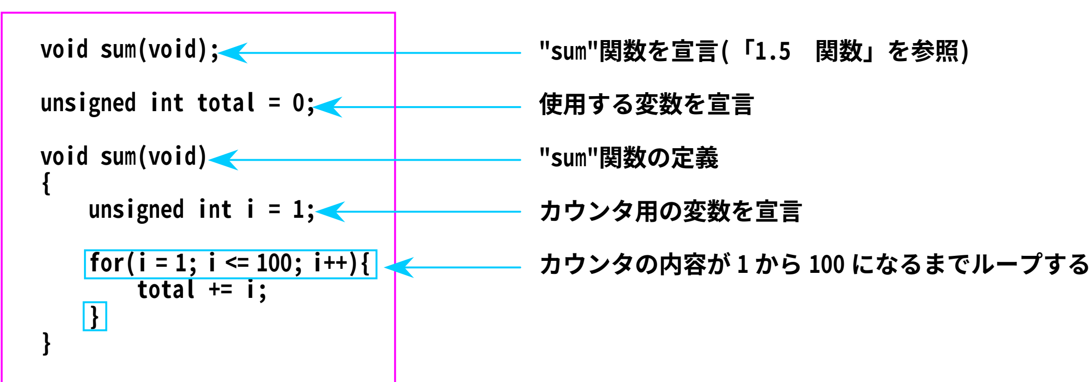
for 文

while 文の記述例のようにカウンタを使って繰り返し処理を行うとき、条件の判定とともに必ず「カウンタの内容の初期化」と「カウンタの内容の変化」という作業が必要です。for 文ではこれらの作業が条件式とともに記述できます。初期化(式 1)、条件式(式 2)、処理(式 3)はそれぞれ省略可能です。しかし、いずれの式を省略した場合でも間のセミコロン';'は必ず記述してください。また、for 文と前述の while 文は常に書き換えできます。



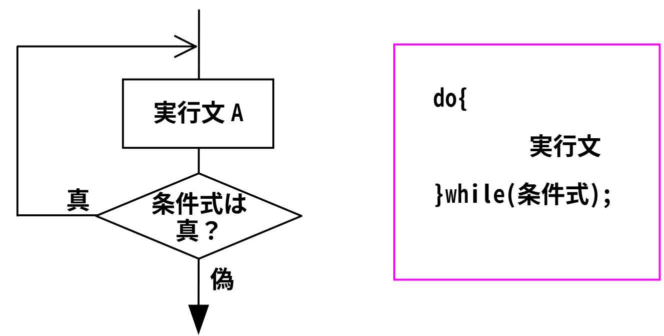
総和を求める-2-(for 文の記述例)

1 から 100 までの整数の和を求めます。



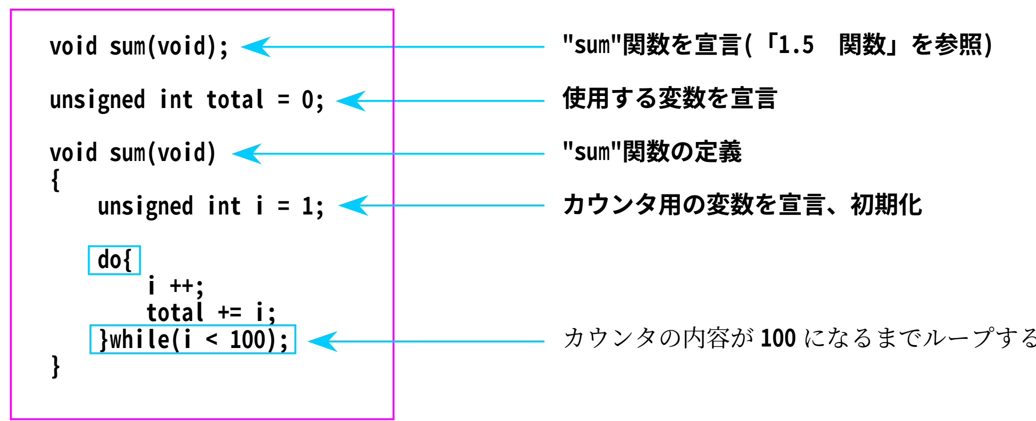
do-while 文

for 文や while 文とは異なり、処理を実行したのち条件判定を行います(「後判定」)。for 文や while 文では、条件によって１度も処理が実行されない場合がありますが、do-while 文では必ず１度は処理が行われます。



総和を求める-3-(do-while 文の記述例)

１から 100 までの整数の和を求めます。



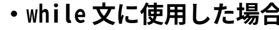
1.4.4　処理の中断

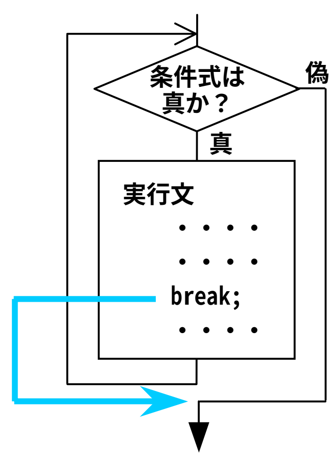
処理を中断して脱出するための制御文(「補助制御文」)には、「break 文」、「continue 文」、「goto 文」があります。

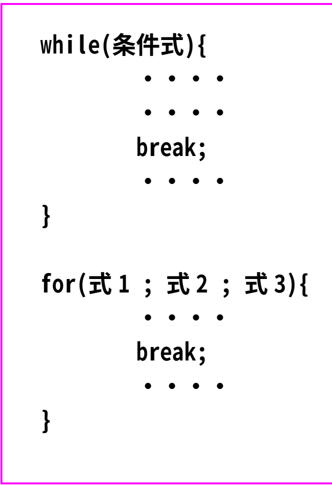
この項では、これらの制御文の記述方法と動作を説明します。

break 文

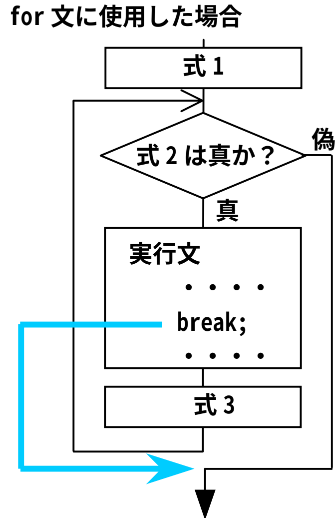
繰り返し処理や switch-case 文の中で使用します。"break;"が実行されると、処理を中断して 1 ブロックだけ脱出します。







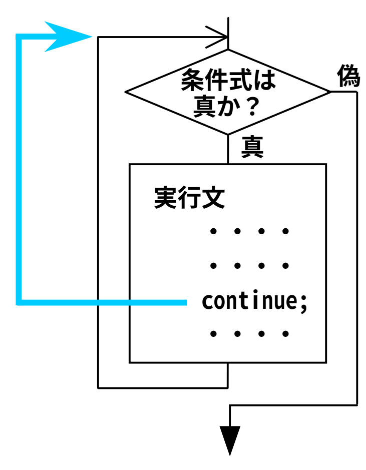
・for 文に使用した場合

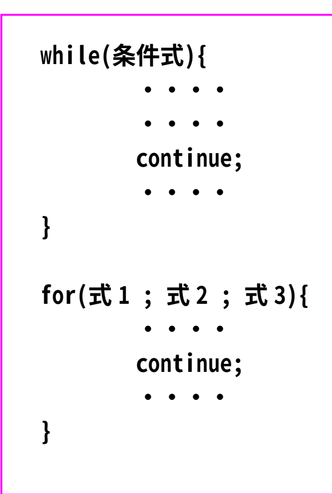


continue 文

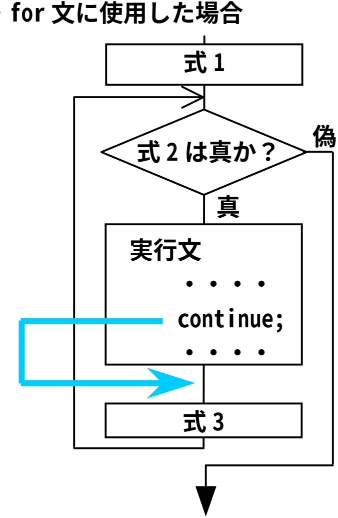
繰り返し処理の中で使用します。"continue;"が実行されると、処理を中断します。中断後 while 文では条件判定に戻り、for 文では式 3 を実行した後、条件判定に戻ります。

・while 文に使用した場合





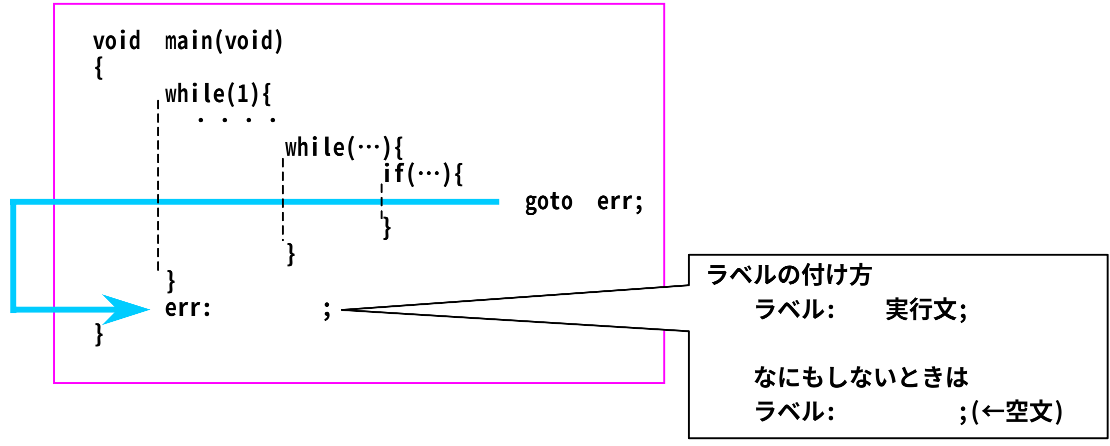
・for 文に使用した場合



goto 文

goto 文が実行されると、goto 文の後ろに記述されているラベルへ無条件に分岐します。break 文や continue 文とは違い、多重ブロックを一気に脱出して関数内のどの場所にでも分岐することができます。しかし、構造化プログラミングに反するため、エラー処理など緊急を要する場合にのみ、使用するほうがよいでしょう。

また、分岐先のラベルの後ろには必ず実行文を記述しなければなりません。なにもしたくない場合は空文(';'のみ)を記述します。



1.5　関数

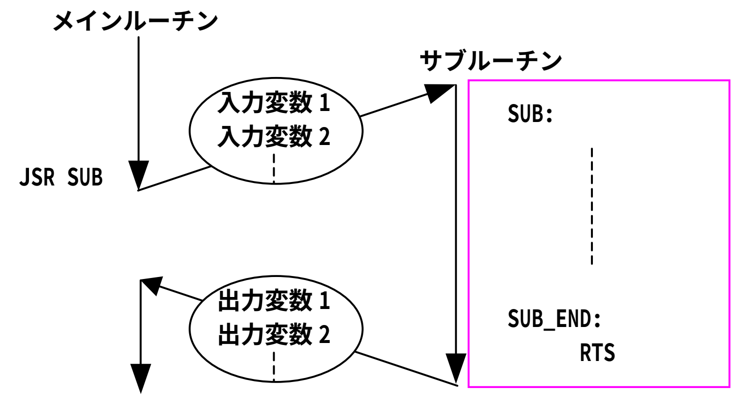
1.5.1　関数とサブルーチン

アセンブリ言語のプログラムがサブルーチンを基本単位としているのと同じように、C 言語では「関数」がプログラムの基本単位となります。
以下の項では、ICC740 での関数の記述方法について説明します。

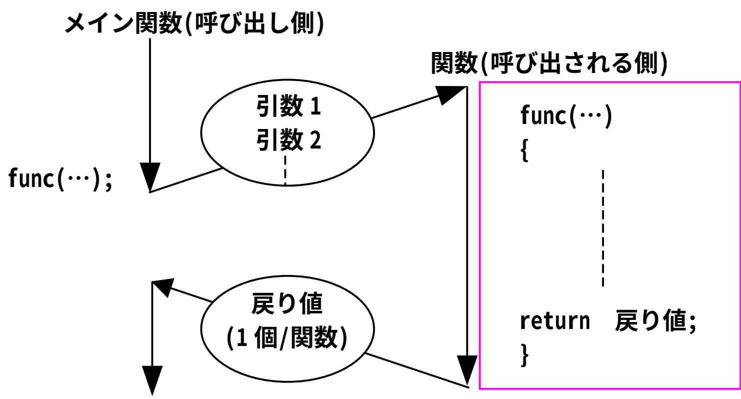
引数と戻り値

関数間のデータのやりとりは、サブルーチンの入力変数にあたる「引数」と出力変数にあたる「戻り値」で行います。
アセンブリ言語の場合は入力変数や出力変数の数に制約はありません。しかし、C 言語の戻り値は各関数につき１個と決められており、「return 文」を使って返します。引数については引数のサイズが１つの関数で合計 256 バイトまでと決められており、256 バイトを超えた場合、コンパイラはエラーを発生します。

・アセンブリ言語の「サブルーチン」



・C 言語の「関数」



1.5.2　関数の作成

関数を使用するためには「関数のプロトタイプ宣言」、「関数の定義」、「関数の呼び出し」の 3 つの手続きが必要です。

この項では、これらの手続きの方法を説明します。

関数のプロトタイプ宣言

C 言語で関数を使用する前には、関数のプロトタイプ宣言を行います。
次に関数のプロトタイプ宣言の書式を示します。

```
戻り値のデータ型　　関数名(引数のデータ型のならび);
```

戻り値や引数がないときは、空(から)を意味する"void"という型を記述します。

関数の定義

関数本体では、引数を受け取るための「仮引数」のデータ型と名称を定義します。また戻り値は「return 文」を使って返します。
次に関数定義の書式を示します。

```
戻り値のデータ型　　関数名(仮引数 1 のデータ型　仮引数 1，・・・・・・・・・・)
{
              ¦
      return　戻り値;
}
```

関数の呼び出し

関数を呼び出すとき、その関数に対する引数を記述します。また呼び出した関数からの戻り値は代入演算子などを用いて受け取ります。

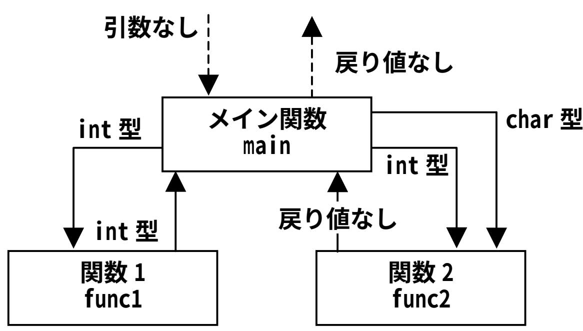
```
関数名(引数 1，・・・・・・・・・・);
```

戻り値があるとき

```
変数＝関数名(引数 1，・・・・・・・・・・);
```

関数の記述例

次のような関係のある 3 つの関数を記述します。




```
／＊　プロトタイプ宣言　＊／
void  main(void);
int  func1(int);
void  func2(int, char);

／＊　メイン関数　＊／
void  main()
{
    int  a = 40,b = 29;
    int  ans;
    char  c = 0xFF;

    ans = func1(a);
    func2(b, c);
}

／＊　関数１の定義　＊／
int  func1(int  x)
{
    int  z;
    z = x + 1;
    return  z;
}
／＊　関数2の定義　＊／
void  func2(int  y, char  m)
{
    :
}
```

a を引数として関数 1("func1")を呼び出す
戻り値は"ans"に代入する

b,c を引数として関数 2("func2")を呼び出す
戻り値はなし

「return 文」で戻り値を返す

1.5.3　関数間でのデータの受け渡し

C 言語では引数と戻り値の受け渡しは、各変数の値をコピーして渡します。したがって、関数を呼び出すときの引数の名前と、呼び出された関数が受け取る引数(仮引数)の名前を一致させる必要はありません。

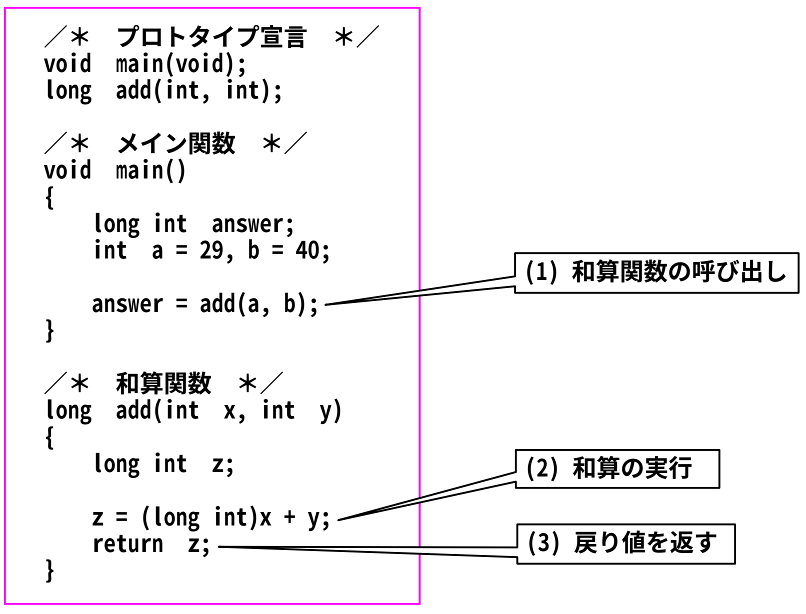
呼び出された関数内での処理はコピーした値(仮引数)を使って行われるので、呼び出し側の変数本体が破壊されることはありません。

これらの理由により、C 言語の関数は独立性が高く、関数の再利用も容易です。

この項では、関数間でのデータの受け渡しについて説明します。

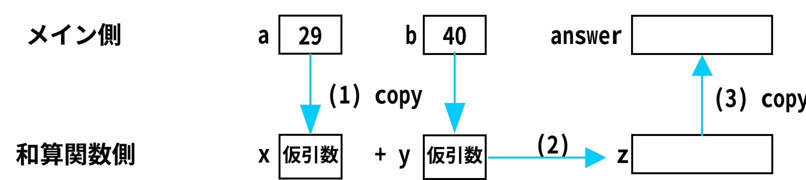
和を求める(関数の記述例)

-32768～32767 の範囲にある任意の 2 つの整数を引数として、その和を求める和算関数"add"を作成し、メイン関数から呼び出します。



```
／＊　プロトタイプ宣言　＊／
void  main(void);
long  add(int, int);

／＊　メイン関数　＊／
void  main()
{
    long int  answer;
    int  a = 29, b = 40;

    answer = add(a, b);
}

／＊　和算関数　＊／
long  add(int  x, int  y)
{
    long int  z;

    z = (long int)x + y;
    return  z;
}
```


<データの流れ>



1.6　記憶クラス

1.6.1　変数と関数の有効範囲

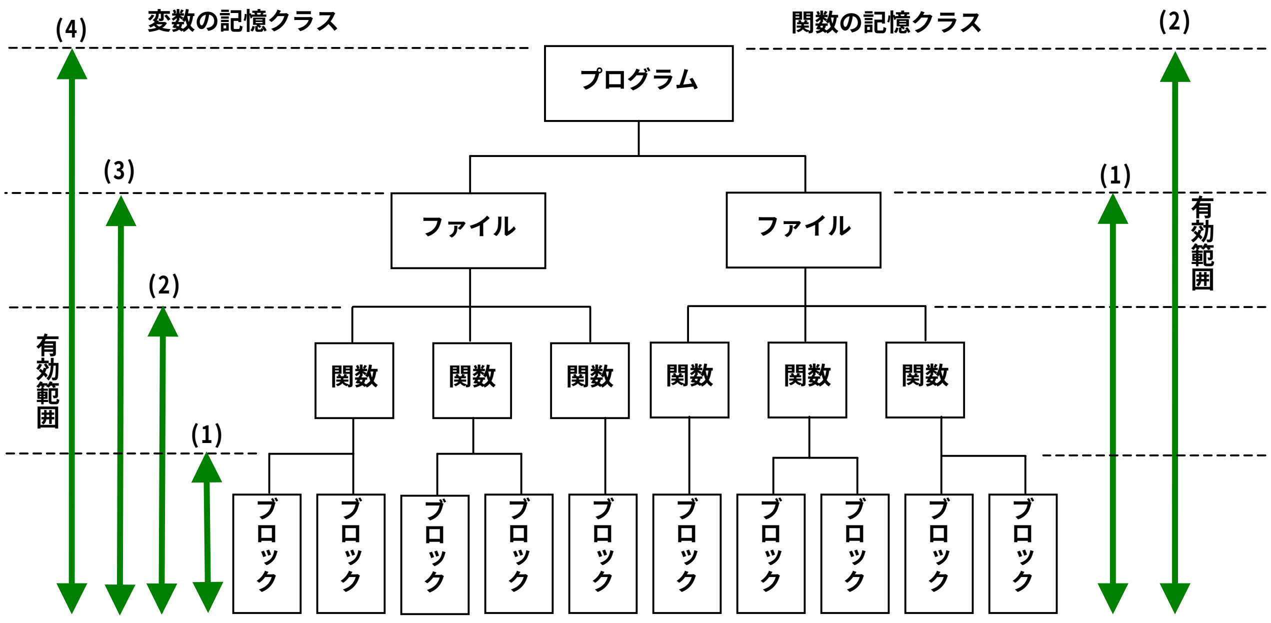
変数や関数はプログラム全体で使用するもの、1 関数でのみ使用するものなど、各々の性質によりその有効範囲を変えることができます。このような変数や関数の有効範囲を指定するのが「記憶クラス(Scope)」です。
以下の項では、変数と関数の記憶クラスの種類とその指定方法について説明します。

変数と関数の有効範囲

C 言語のプログラムは複数のソースファイルから構成されています。さらに、1 つのソースファイルは複数の関数によって構成されています。つまり、C 言語のプログラムは、次に示すような階層構造になっています。

変数には、次の 4 段階の記憶クラスがあります。
(1)ブロック内でのみ有効
(2)関数内でのみ有効
(3)ファイル内でのみ有効
(4)プログラム全体で有効
関数には、次の 2 段階の記憶クラスがあります。
(1)ファイル内でのみ有効
(2)プログラム全体で有効

C 言語ではこれらの記憶クラスを変数、関数ごとに指定することができます。そして、この記憶クラスを効率よく利用することによって、自作の変数や関数を非公開にしたり、逆に公開して共有したりすることができます。

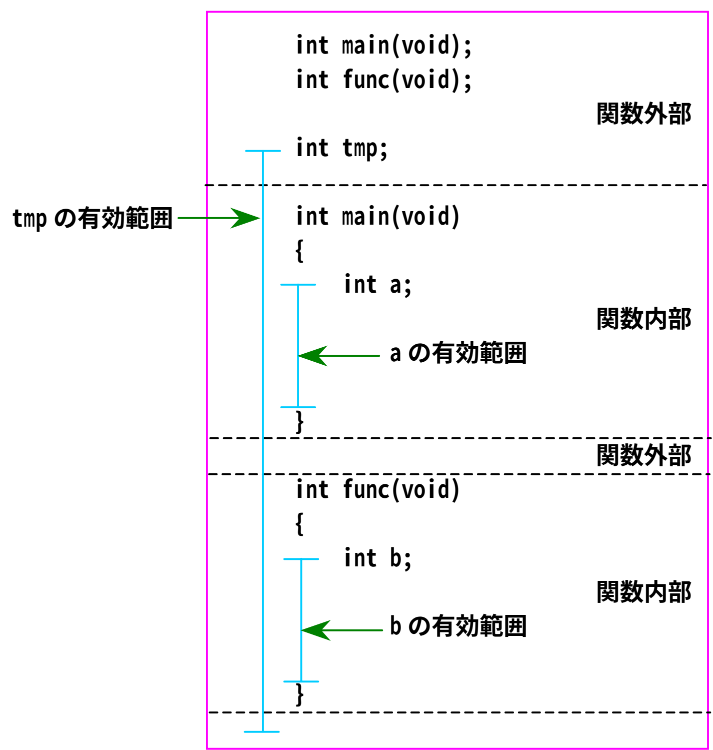


1.6.2　変数の記憶クラス

変数の記憶クラスは、型宣言を行うときに指定します。ポイントは次の 2 点です。
(1)外部変数と内部変数(→型宣言を行う場所)
(2)記憶クラス指定子(→型宣言に指定子を付加)
この項では、変数に対する記憶クラスの指定方法を説明します。

外部変数と内部変数

変数の有効範囲を指定するいちばん簡単な方法で、変数の型宣言を行う位置によって有効範囲が決まります。関数の外側で宣言した変数を「外部変数」、関数の内側で宣言した変数を「内部変数」といいます。「外部変数」は宣言以降どの関数からも参照できるグローバルな変数です。一方、「内部変数」は宣言以降その関数内でのみ有効なローカル変数となります。



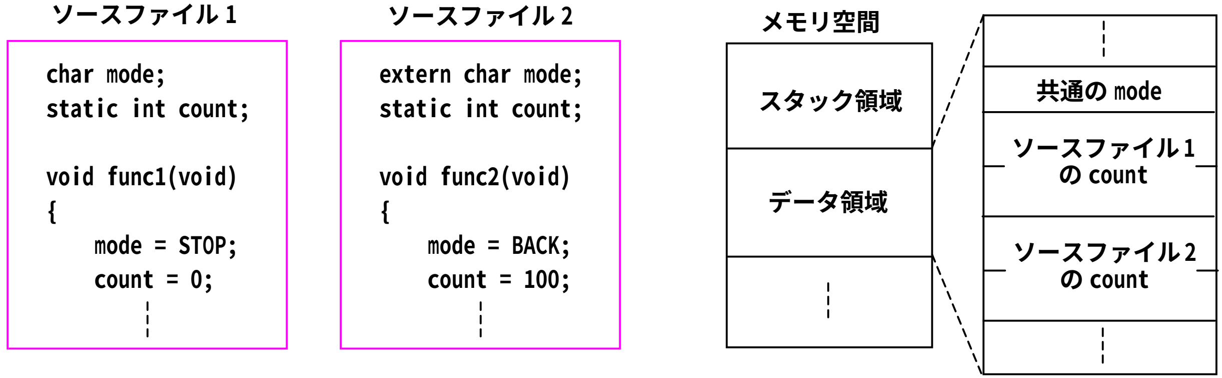
記憶クラス指定子

変数に対して使用する記憶クラス指定子は「auto」、「static」、「register」、「extern」があります。書式を次に示します。

記憶クラス指定子　　データ型　　変数名;

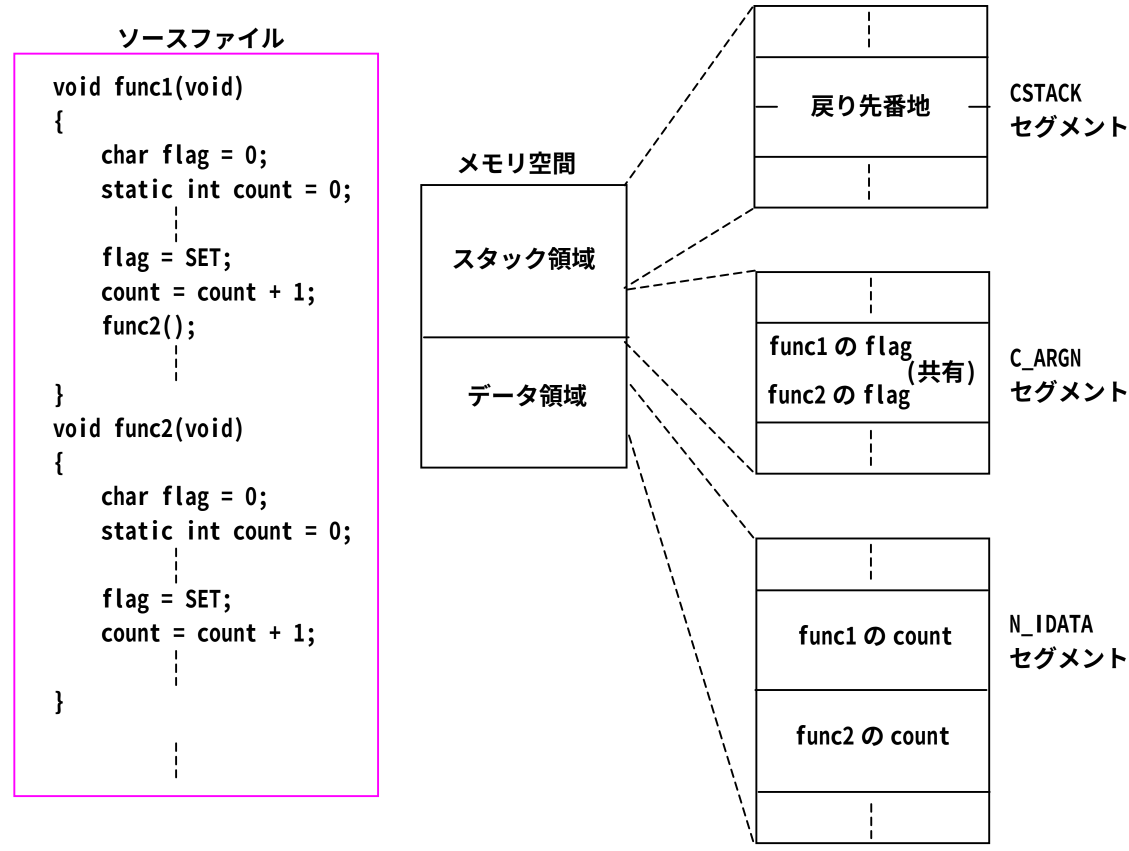
外部変数の記憶クラス

外部変数を宣言するときに､記憶クラス指定子をなにも付けなければプログラム全体で有効なグローバル変数となります。一方、宣言時に"static"を記述すると、宣言したファイル内のみ有効な変数となります。
ソースファイル２の"mode"のような､別ファイルで宣言した外部変数を使用する場合は"extern"を記述します。
外部変数は、初期値を設定しない変数はデータ領域上の N_UDATA セグメント、初期値を設定した変数はデータ領域上の N_IDATA セグメントにとられます。



内部変数の記憶クラス

記憶クラス指定子を付けずに宣言した内部変数はデータ領域上の C_ARGN セグメントに領域をとります。この領域は各関数で共有され、その関数が呼び出される度に初期化されます。一方、"static"を付けた内部変数は、初期値を設定しない変数は０に初期化してデータ領域上の N_UDATA セグメント、初期値を設定した変数は設定した値に初期化してデータ領域上の N_IDATA セグメントにとられます｡初期化されるのはプログラム起動時だけです。

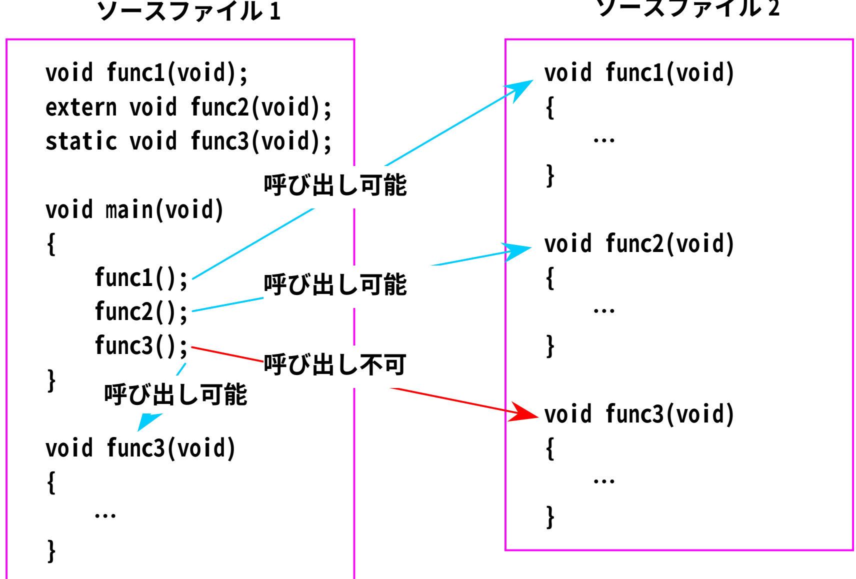


1.6.3　関数の記憶クラス

関数の記憶クラスは、関数の定義側と宣言側の両方で指定します。記憶クラス指定子「static」と「extern」を使用することもできます。
この項では、関数に対する記憶クラスの指定方法を説明します。

グローバルな関数とローカルな関数

(1)関数の定義で記憶クラスを指定しないとき
この関数は他のソースファイルからも呼び出して使用できるグローバルな関数となります。
(2)関数のプロトタイプ宣言で「extern」と宣言したとき
この関数は関数が記述されるソースファイルになく、他のソースファイルの関数を呼び出すことを明示しています。ただし、"extern"を付けなくてもプロトタイプ宣言してあれば、関数がソースファイル内にないときは、自動的に"extern"をつけたのと同じ扱いになります。
(3)関数の定義で「static」と宣言したとき
この関数は他のソースファイルからは呼び出せないローカルな関数になります。



記憶クラスのまとめ

変数の記憶クラスと関数の記憶クラスを次にまとめます。

変数の記憶クラス

記憶クラス	外部変数	内部変数
記憶クラス指定子省略	他のソースファイルからも参照できるグローバルな変数 [セグメント N_UDATA、N_IDATA に割り付ける] [データ保持 ○]	関数内でのみ有効な変数 [実行時にセグメント C_ARGN に割り付ける] [データ保持 ×]
auto		関数内でのみ有効な変数 [実行時にセグメント C_ARGN に割り付ける] [データ保持 ×]
static	他のソースファイルからは参照できないローカルな変数 [セグメント N_UDATA、N_IDATA に割り付ける] [データ保持 ○]	関数内でのみ有効な変数 [セグメント N_UDATA、N_IDATA に割り付ける] [データ保持 ○]
register		関数内でのみ有効な変数 [実行時にセグメント C_ARGN に割り付ける] [データ保持 ×]
extern	他のソースファイルの変数を参照する変数 [割り付けない]	他のソースファイルの変数を参照する変数 (他の関数からの参照はできない) [割り付けない]

関数の記憶クラス

記憶クラス	関数の種類
記憶クラス指定子省略	他のソースファイルからも呼び出し実行できるグローバルな関数 [定義側で指定する]
static	他のソースファイルからは呼び出しできないローカルな関数 [定義側で指定する]
extern	他のソースファイルにある関数 [宣言側で指定する]

1.7　配列とポインタ

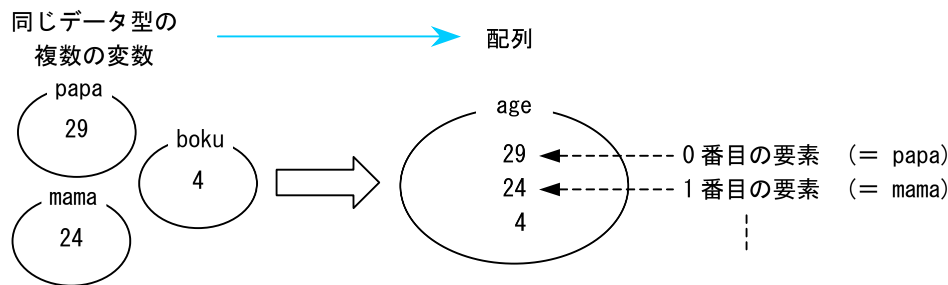
1.7.1　配列

この項では、配列の考え方を説明します。

配列とは何か

家族の年齢の合計を求めるプログラムを例とします。家族の構成は両親(父＝29 歳、母＝24 歳)と子ども１人(僕＝4 歳)です。

このプログラムでは、家族が増えると変数名が増えます。これに対処する手段として、C 言語では「配列」という概念があります。配列とは、同じ型をもつデータ(int 型)を１つの集合体として扱います。この例では父の年齢(papa)、母の年齢(mama)‥‥と別の変数として扱うのではなく、家族の年齢(age)という集合体とします。各データは集合体の「要素」となります。つまり、0 番目の要素が父、1 番目の要素が母、2 番目が僕です。



家族の年齢の合計を求める-1-

家族(父、母、僕)の年齢の合計を求めます。

```
void main(void)
{
    int papa = 29;
    int mama = 24;
    int boku = 4;
    int gokei;

    gokei = papa + mama + boku;
}
```

家族が増えると変数の型宣言、初期化の実行文が増え、演算式も長くなる。

```
void main(void)
{
    int papa = 29;
    int mama = 24;
    int boku = 4;
    int imouto1 = 1;
    int imouto2 = 1;
        :
    int gokei;

    gokei = papa + mama + boku + imouto1 + imouto2 + …;
}
```

1.7.2　配列の作成

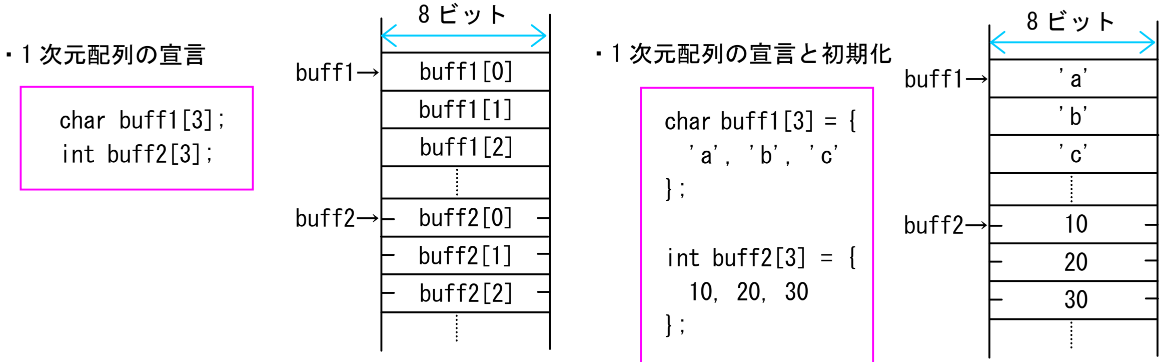
C 言語で扱う配列に、「1 次元配列」や「2 次元配列」があります。
この項では、各々の配列の作成と参照方法を説明します。

1 次元配列

1 次元配列は、 1 次元の(直線的な)広がりをもつ配列です。 1 次元配列の宣言の書式を示します。

データ型　配列名［要素数］

上の宣言を行うと、配列名を先頭ラベルとして要素数分だけメモリ上に領域が確保されます。
1 次元配列を参照するためには、配列名に要素番号を添字として付けます。ただし、要素番号は'0'から始まるので、最後の要素番号は「要素数-1」になります。



家族の年齢の合計を求める-2-

配列を利用して家族の年齢の合計を求めます。

```
#define MAX 3 (注)

void main(void)
{
    int age[MAX];
    int gokei = 0;
    int i;

    age[0] = 29;
    age[1] = 24;
    age[2] = 4;

    for(i = 0; i < MAX; i++){
        gokei += age[i];
    }
}
```

または

```
#define MAX 3

void main(void)
{
    int age[MAX] = {
        29, 24, 4
    };

    int gokei = 0;
    int i;

    for(i = 0; i < MAX; i++){
        gokei += age[i];
    }
}
```

宣言と同時に初期化している

配列にしておくと要素数を変数にして繰り返し文が利用できる

(注)　#define MAX 3　：　MAX = 3 とマクロの定義をしている
（「1.9　プリプロセスコマンド」参照)

2 次元配列

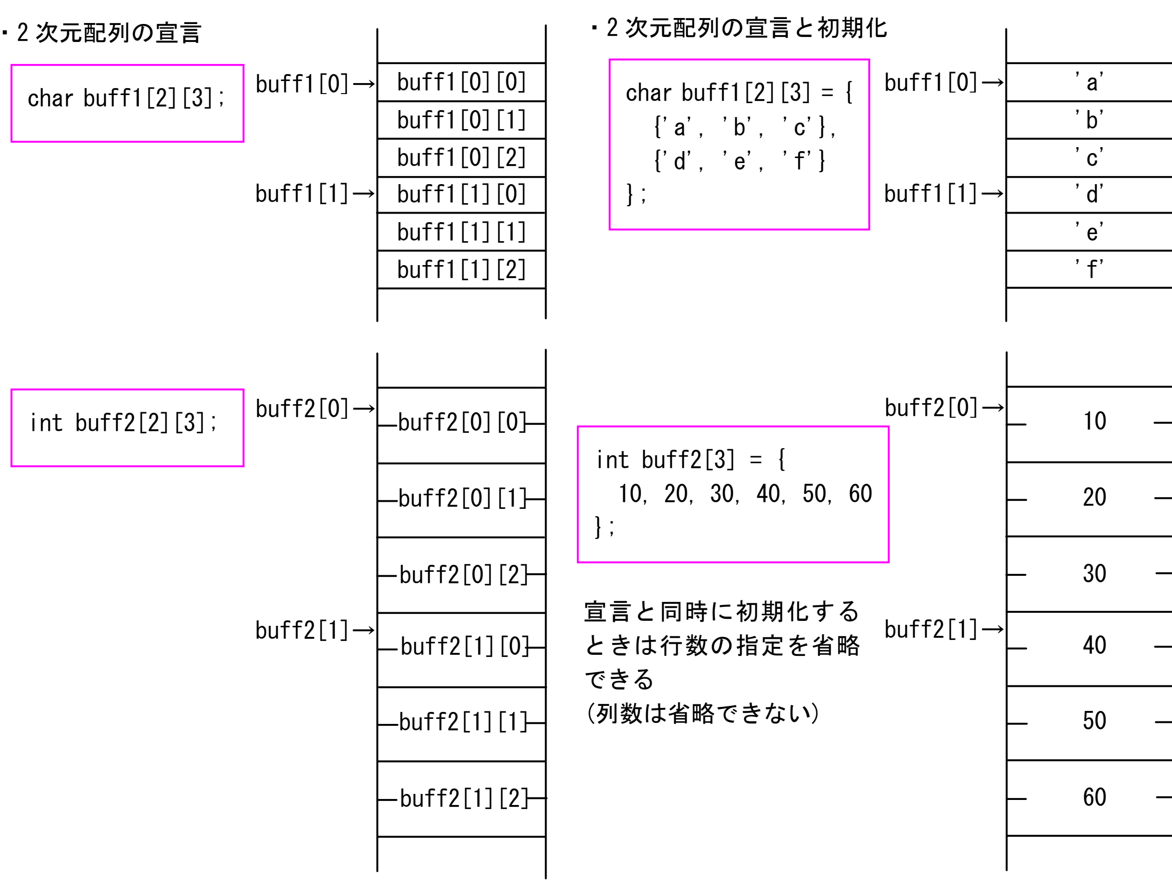
2 次元配列は「行」と「列」からなる平面的な広がりをもつ配列です。また、1 次元配列の配列とみることもできます。次に宣言の書式を示します。

データ型　配列名［行数］［列数］；

2 次元配列を参照するときは配列名に「行番号」と「列番号」の 2 つを添字として付けます。行番号、列番号ともに’0’から始まるので、最後の番号は「行数(列数)-1」になります。

・2 次元配列の概念

行↓		
0 行 0 列	0 行 1 列	0 行 2 列
1 行 0 列	1 行 1 列	1 行 2 列



1.7.3　ポインタ

ポインタとはデータを指し示すもの、つまりアドレスを意味します。ポインタは、配列を扱う上で便利な手段となります。

これから紹介する「ポインタ変数」は、データが格納されている「アドレス」を変数として扱います。アセンブリ言語でいうところの「間接アドレッシング」にあたるものです。

この項では、ポインタ変数の宣言と参照方法を説明します。

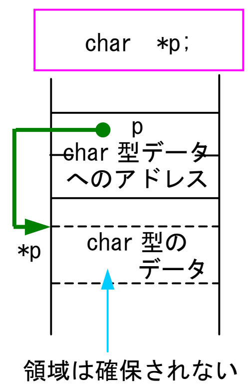
ポインタ変数の宣言

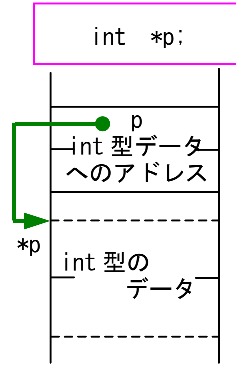
ポインタ変数は次のような書式で宣言します。

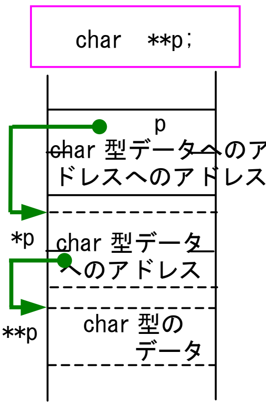
```
指し示すデータの型　*ポインタ変数名；
```

ただし、上の宣言で確保される領域はアドレスを格納する領域のみです。データ本体の領域を確保するには、別に型宣言する必要があります。

・ポインタ変数の宣言

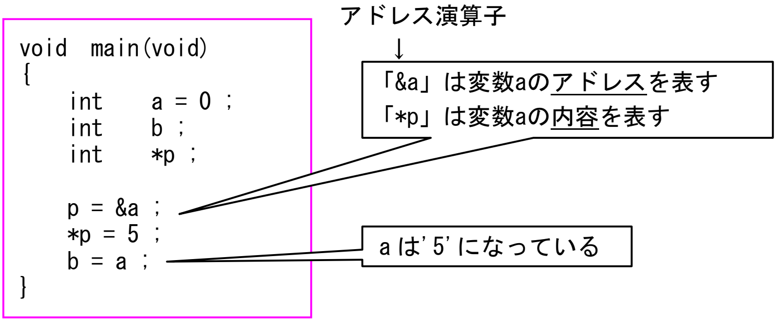






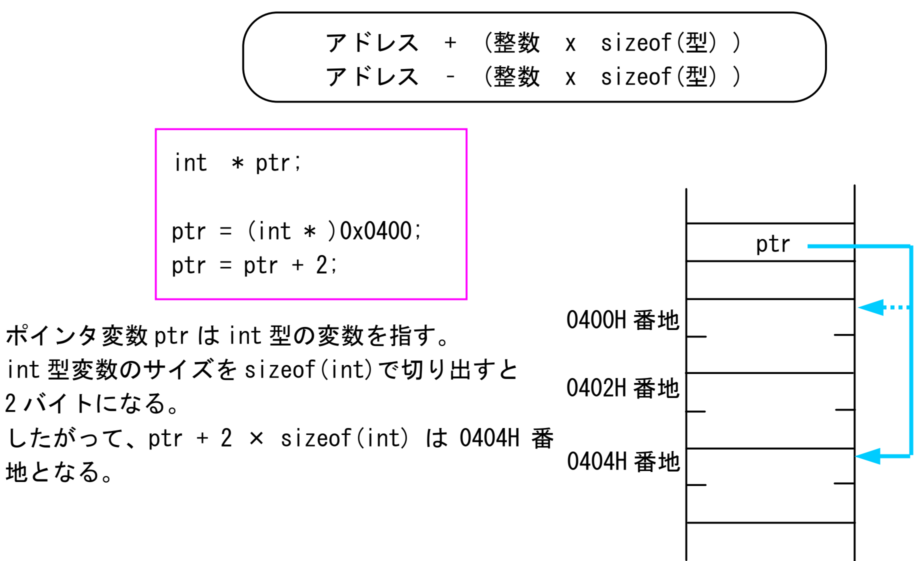
ポインタと変数の関係

次にポインタ変数と変数の関係を、int 型の変数 a に対して int 型へのポインタ変数 p を使って定数 5 を代入する方法を例として説明します。



ポインタ変数の演算

ポインタ変数を加算、または減算することができます。ただし、このポインタ変数演算は整数の演算とは異なり、演算結果はアドレス値となります。したがって、ポインタ変数が指すデータのサイズによってアドレス計算が異なります。



コラム　ポインタ変数のデータ長は？

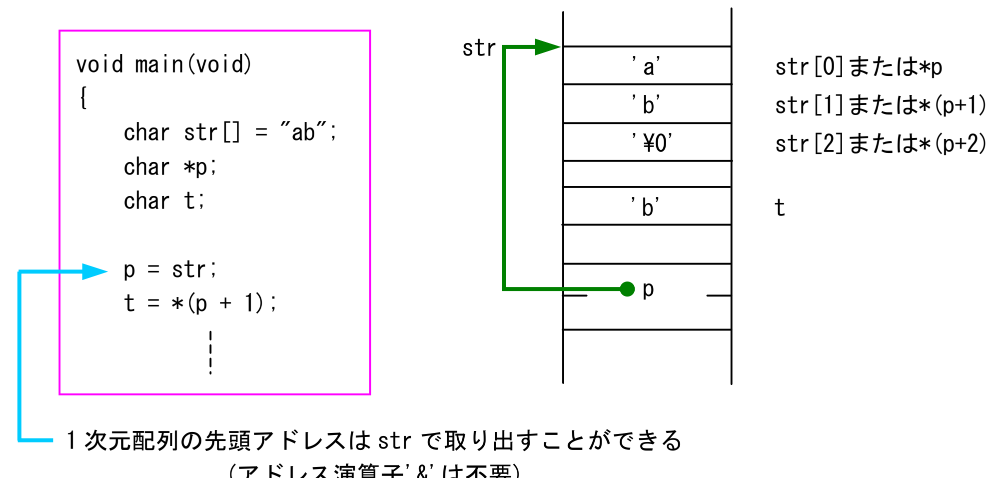
C 言語プログラムの変数のデータ長は、データ型により決まります。ポインタ変数の場合、その内容はアドレスです。したがって、一般的に使用するマイクロプロセッサがアクセスできる全アドレス空間を表現できるだけのデータ長がポインタ変数に対して用意されることになります。

1.7.4　ポインタの活用

この項では、ポインタの活用例を説明します。

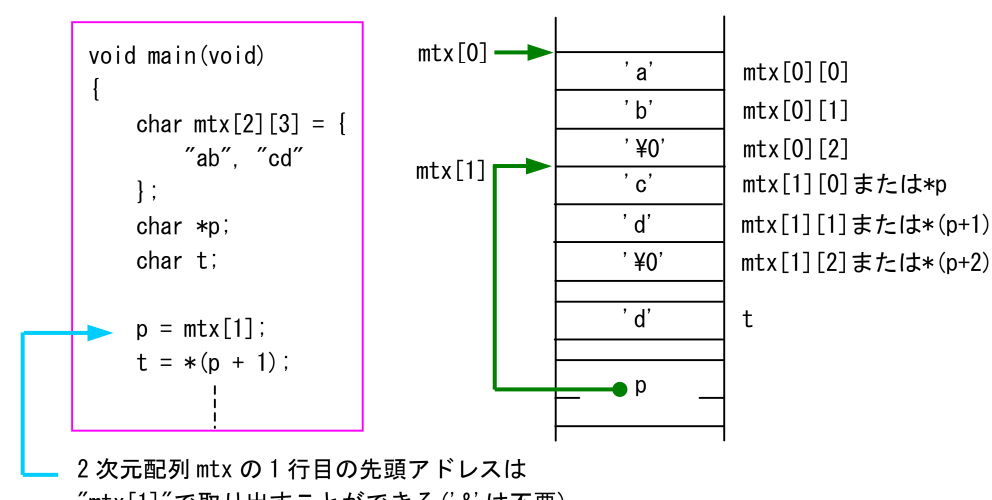
ポインタ変数と 1 次元配列

配列名に要素番号を示す添字を記述する方法では、「インデックスアドレッシング」としてコード化されます。このため、配列をアクセスする際には常に「先頭から何番目」というアドレス計算が必要になります。
一方、ポインタ変数を利用すると間接アドレッシングとなります。



ポインタ変数と 2 次元配列

2 次元配列も 1 次元配列と同様、ポインタ変数を使ってアクセスできます。



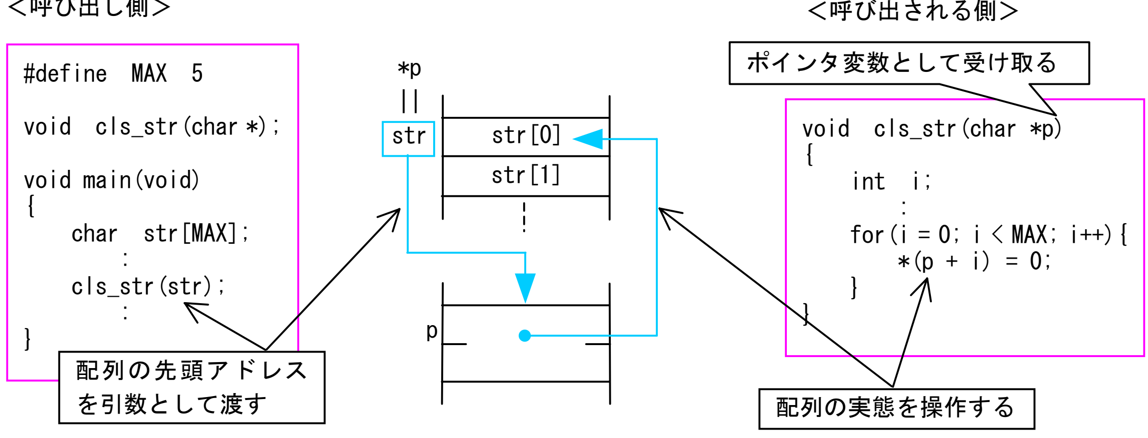
関数のアドレス渡し

C 言語の関数間の基本的なデータの受け渡しは「値渡し」です。しかしこの方法では、配列や文字列を引数や戻り値にすることができません。

そこで使用される方法が、ポインタ変数を利用した「アドレス渡し」です。この方法は配列や文字列のアドレスを受け渡しする以外に、複数のデータを戻り値にしたいときにも利用できます。

ただし値渡しとは違い、アドレス渡しでは、呼び出された側でポインタ変数の中身を書き換えると、呼び出し側のデータ領域を直接書き換えることになるので、関数の独立性は低くなります。

アドレス渡しにより配列を受け渡す例を示します。



コラム　関数間で高速にデータを受け渡しする

関数間でデータを受け渡しする方法として、値渡しとアドレス渡し以外に、受け渡しするデータを外部変数にする方法があります。

この方法では関数の独立性が失われてしまうので、C 言語のプログラムとしては推奨できません。しかし関数呼び出し時の入口処理と出口処理(引数と戻り値の受け渡し)がなくなるので、高速に関数を呼び出すことができるというメリットがあります。この性質を利用して、汎用性をあまり必要とせず、しかも高速処理を行いたい ROM 化プログラムではよく使用されます。

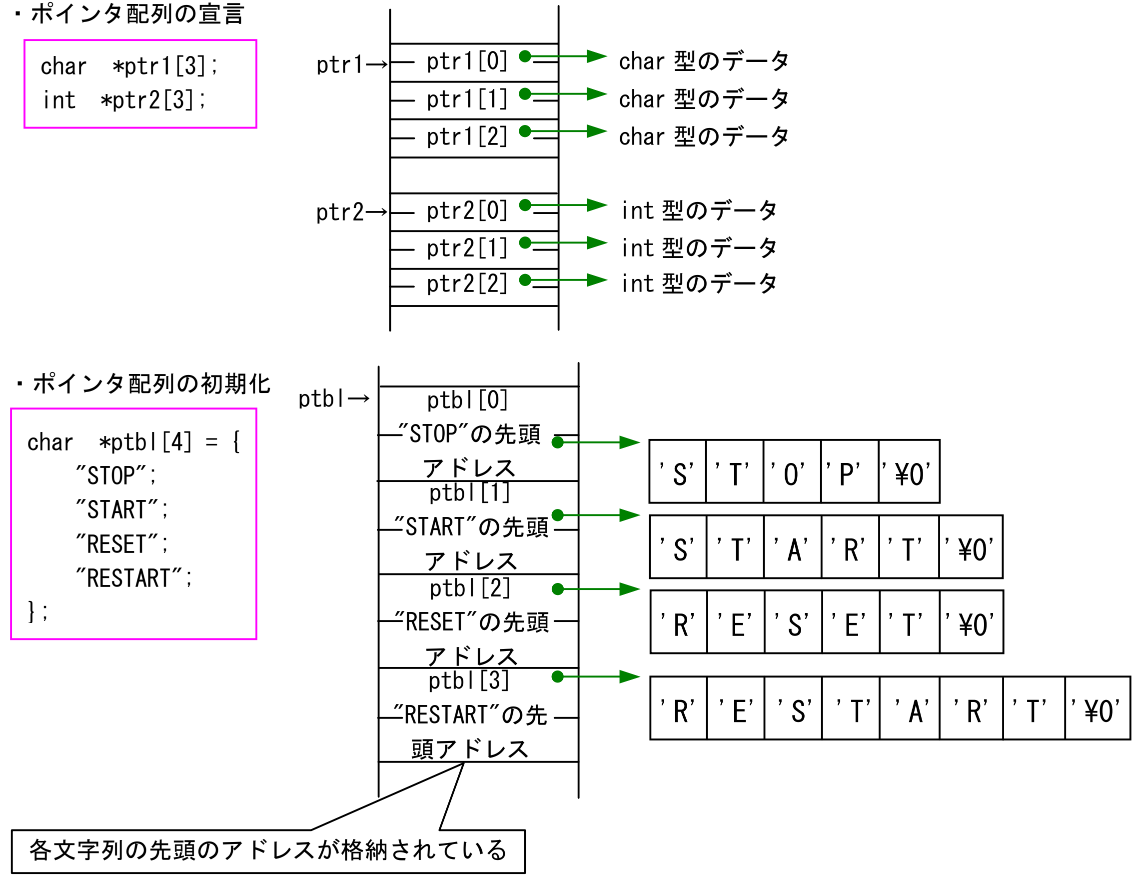
1.7.5 ポインタの配列化

この項では、ポインタ変数を配列にした「ポインタ配列」を説明します。

ポインタ配列の宣言

ポインタ配列の宣言方法を示します。

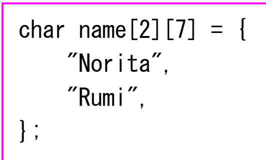
データ型　*配列名［要素数］；



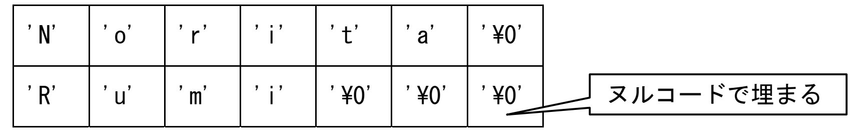
ポインタ配列と 2 次元配列

ポインタ配列と 2 次元配列の違いを説明します。2 次元配列で文字数の異なる複数の文字列を宣言したとき、空いた領域はヌルコード'¥0'で埋まります。同じことをポインタ配列で定義するとメモリの空きが発生しません。

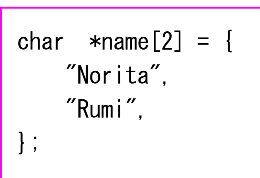
・2 次元配列



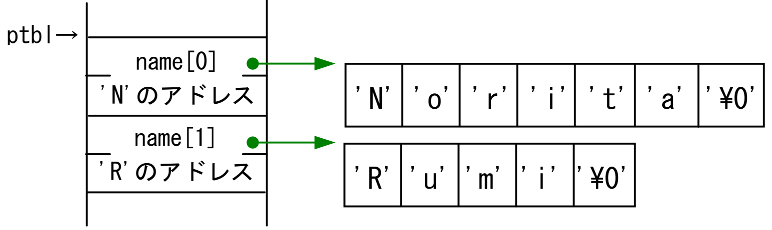
```
char name[2][7] = {
    "Norita",
    "Rumi",
};
```



・ポインタ配列



```
char  *name[2] = {
    "Norita",
    "Rumi",
};
```



1.7.6 関数ポインタを使ったテーブルジャンプ

アセンブリ言語のプログラムでは、あるデータの内容によって切り換えるべき処理が増えると「テーブルジャンプ」を使用します。前述のポインタ配列を利用すると、同様のことがC言語でも記述できます。
この項では、「関数ポインタ」を使ったテーブルジャンプの記述方法を説明します。

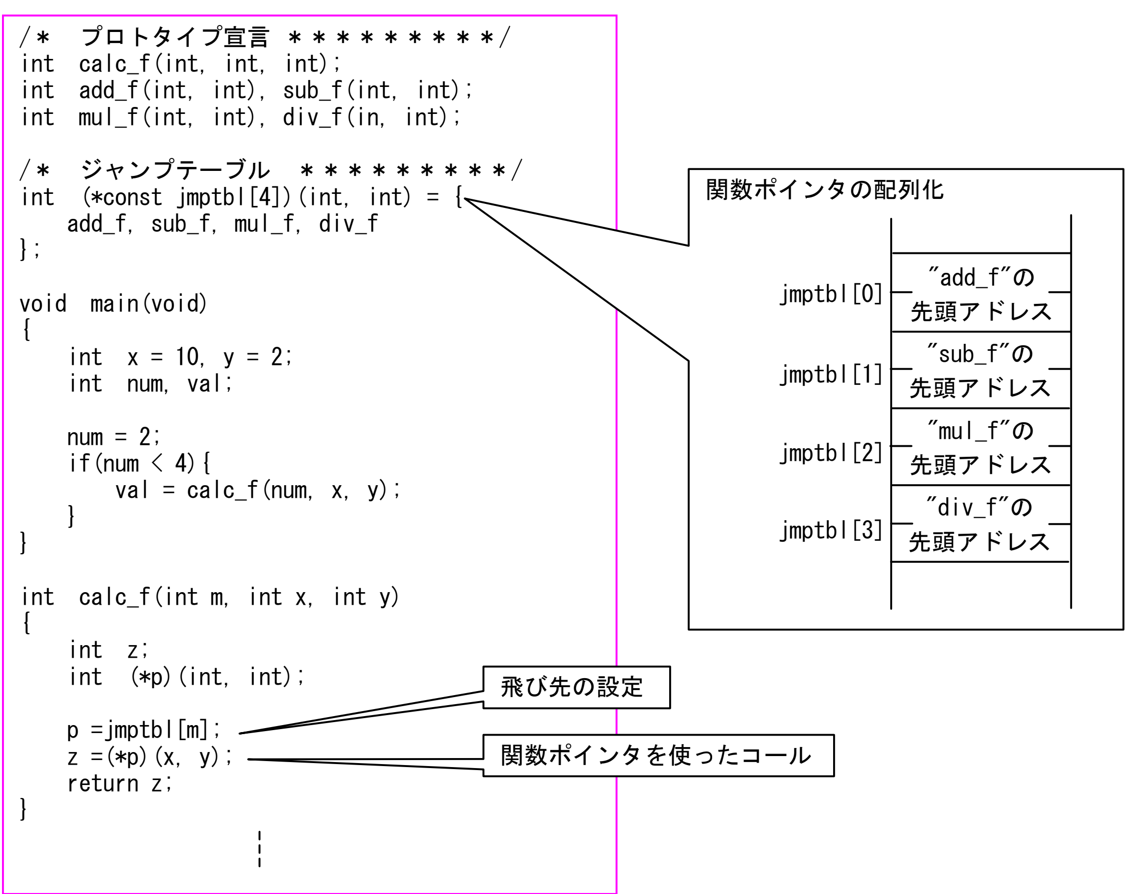
関数ポインタとは

前述のポインタと同様、関数の先頭アドレスを指し示すものが「関数ポインタ」です。関数ポインタを使用すると呼び出す関数をパラメータにできます。宣言と参照の書式を次に示します。

```
<宣言の書式> 戻り値の型(*関数ポインタ名)(引数のデータ型);
<参照の書式> 戻り値を格納する変数=(*関数ポインタ名)(引数);
```

テーブルジャンプを使った四則演算の切り換え

変数"num"の内容によって演算方法を切り換えます。



1.8　構造体と共用体

1.8.1　構造体と共用体

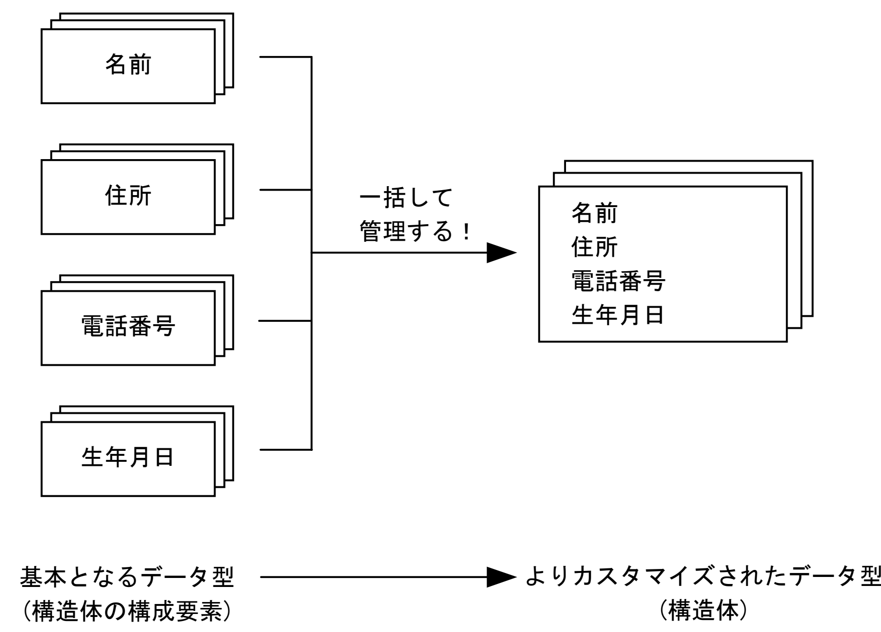
これまでのデータ型(char 型、signed int 型、unsigned int など)は、コンパイラ仕様で決められた「基本データ型」といわれるものです。

C 言語ではこれらの基本データ型を組み合わせた新しいデータ型を作ることができます。それが「構造体」と「共用体」です。

以下の項では、構造体と共用体の宣言方法と参照方法を説明します。

基本データ型から構造体へ

構造体や共用体を使用すれば、基本データ型をもとにして目的に応じた、よりカスタマイズされたデータ型を作成できます。しかも、新たに作成されたデータ型は基本データ型と同じように参照したり、配列にすることができます。



1.8.2 新しいデータ型の作成

新しいデータ型の基本となる構成要素のことを「メンバ」といいます。新しいデータ型を作るには、構成するメンバを定義します。この定義によって、これまでの変数と同じように、宣言して領域を確保し、必要に応じて参照できます。

この項では、構造体と共用体、各々の定義と参照方法を説明します。

構造体と共用体の違い

構造体と共用体では領域を確保する際、メンバの配置方法が異なります。

(1)構造体：メンバをシーケンシャルに配置します。

(2)共用体：メンバを同一アドレスに配置します。
(複数のメンバが同一メモリ領域を共有します。サイズは同一アドレスに割り当てられるものの中で、最も長い値のサイズが採られます。)

構造体の定義

構造体型を定義するためには「"struct"」を記述します。

```
struct  構造体タグ{
        メンバ1 ;
        メンバ2 ;
            :
};
```

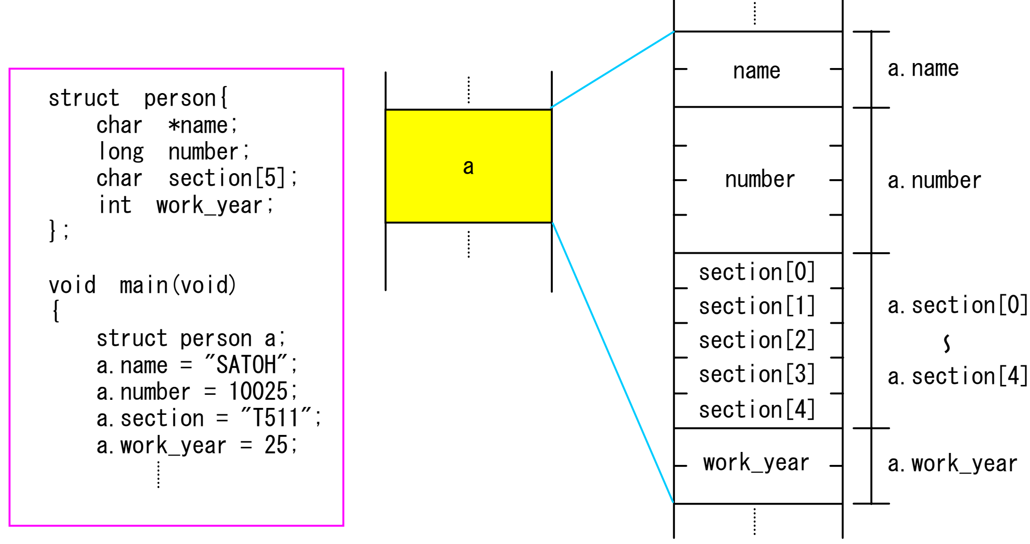
上のように記述すると「struct 構造体タグ」というデータ型ができます。このデータ型を使って変数定義すれば、通常の変数と同様、メモリ領域が確保されます。

```
struct  構造体タグ  構造体変数名 ;
```

構造体の参照

構造体の各メンバを参照するときは、「構造体メンバ演算子'.'」を用います。

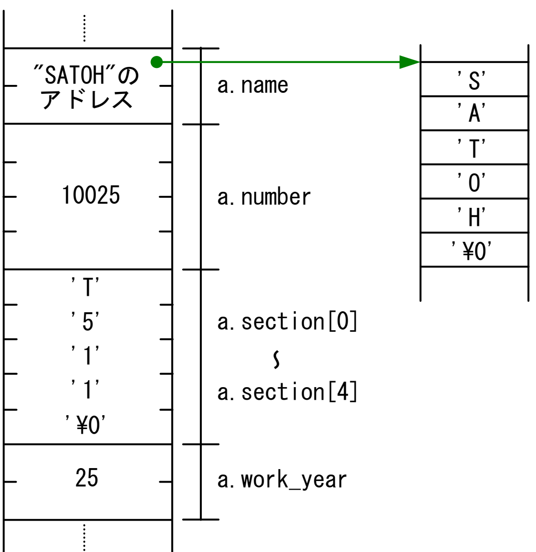
構造体変数名.メンバ名



構造体変数を初期化するときは、各メンバの初期化データを宣言順に、型を合わせて書き並べます。

・**構造体変数の初期化**

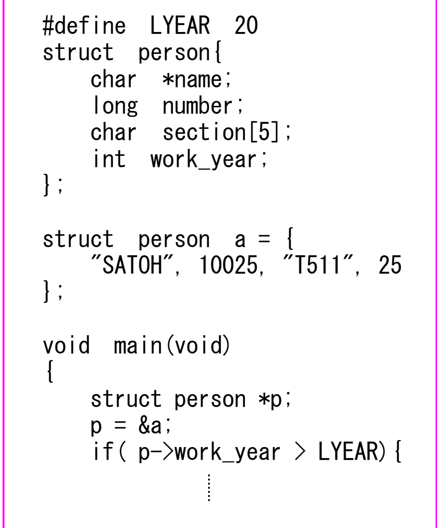
```
struct  person a = {
    "SATOH", 10025, "T511", 25
};
```



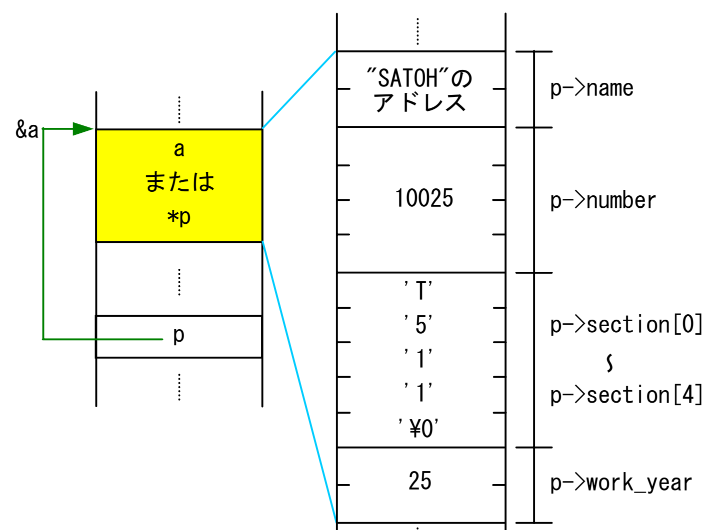
ポインタを使った参照例

ポインタを使って各メンバを参照するときは、「'->'」を使用します。

ポインタ->メンバ名

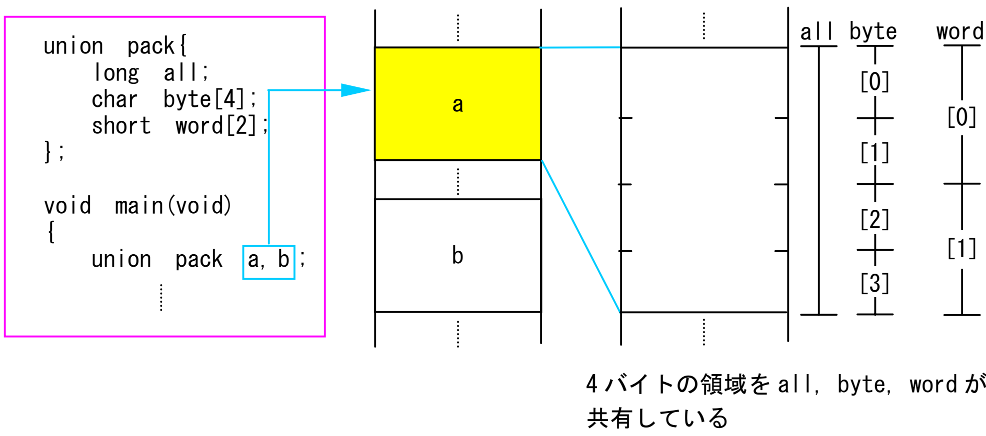


```
#define  LYEAR  20
struct  person{
    char  *name;
    long  number;
    char  section[5];
    int  work_year;
};

struct  person  a = {
    "SATOH", 10025, "T511", 25
};

void  main(void)
{
    struct person *p;
    p = &a;
    if( p->work_year > LYEAR){
            ⋮
```



共用体

共用体はメモリ上のある領域を全メンバで共用します。このため、絶対に同時に存在しない複数のデータを共用体にするとメモリ使用量を節約できます。また、状況によって 16 ビット単位、8 ビット単位など、扱う単位を切り換えたいデータに対して便利な機能です。
共用体を定義するためには「"union"」を記述します。この記述以外の定義、宣言、参照については構造体の場合と同じです。



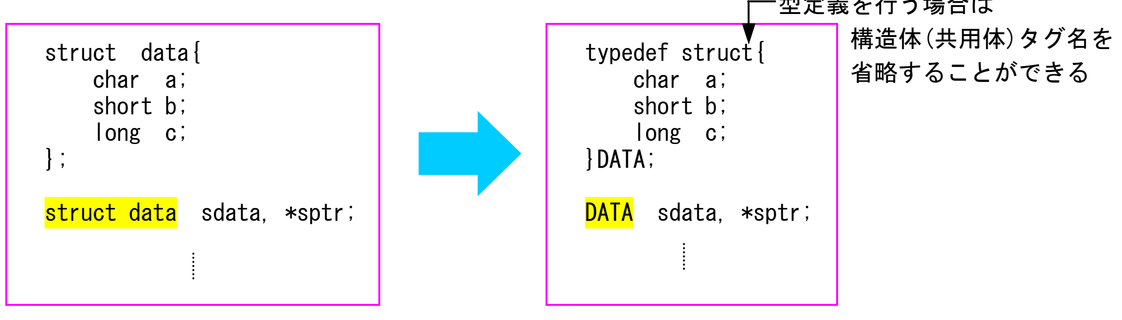
4 バイトの領域を all, byte, word が共有している

コラム 型定義

構造体や共用体では"struct"や"union"のキーワードが必要なため、定義されたデータ型の文字数が増えてしまいます。これを回避する方法に"typedef"があります。

```
typedef 既存型名 新規型名;
```

上のように記述すると、新規型名が既存型名と同義語とみなされ、プログラム中ではどちらの型名も使用できます。では実際に"typedef"を使用した例を示します。



1.9 プリプロセスコマンド

1.9.1 ICC740のプリプロセスコマンド

C言語では、ファイルの取り込み、マクロの定義、条件コンパイルのような機能を「プリプロセスコマンド」としてサポートしています。
以下の項では、ICC740で用意している主なプリプロセスコマンドを説明します。

ICC740のプリプロセスコマンド一覧

プリプロセスコマンドは他の実行文と区別するため、先頭が「'#'」で始まる文字綴りになっています。記述位置は任意ですが、区切りを表すセミコロン';'はつけません。ICC740で使用できる主なプリプロセスコマンドを示します。

記述	機能
#include	指定したファイルを取り込みます。
#define	文字列の置換およびマクロの定義を行います。
#undef	#defineによる定義を取り消します。
#if～#elif～#else～#endif	条件コンパイルを行います。
#ifdef～#elif～#else～#endif	条件コンパイルを行います。
#ifndef～#elif～#else～#endif	条件コンパイルを行います。
#error	メッセージを標準出力に出力し処理を中断します。
#line	ファイルの行番号を指定します。
#pragma	ICC740の拡張機能の処理を指示します。

1.9.2　ファイルの取り込み

別のファイルを取り込むためには「"#include"」を使用します。検索するディレクトリによって記述方法が異なります。
この項では、目的別に"#include"の記述方法を説明します。

標準ディレクトリを検索する

```
#include　＜ファイル名＞
```

コンパイラオプション'-I'で指定したディレクトリ内のファイルを取り込みます。このディレクトリにファイルが存在しない場合は、ICC740の環境変数"C_INCLUDE"で設定した標準ディレクトリを検索し、ファイルを取り込みます。
通常、標準ディレクトリとして「標準インクルードファイル」が入っているディレクトリを指定します。

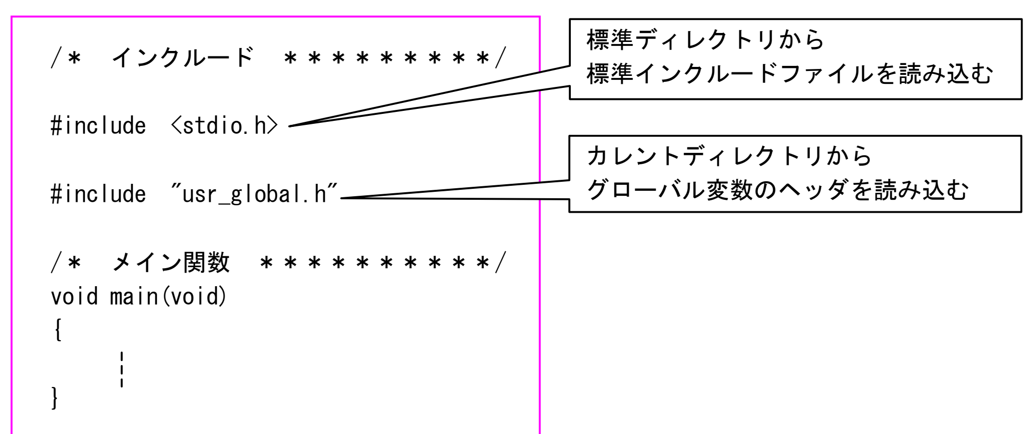
カレントディレクトリを検索する

```
#include　″ファイル名″
```

カレントディレクトリのファイルを取り込みます。カレントディレクトリにファイルが存在しなければ、コンパイラオプション'-I'で指定したディレクトリ、ICC740の環境変数"C_INCLUDE"で設定したディレクトリの順で検索し、ファイルを取り込みます。
標準インクルードファイルと区別するために独自に作成したインクルードファイルをカレントディレクトリに入れて、この記述方法で指定します。

″#include″使用例

検索対象のどのディレクトリにも該当するファイルが存在しない場合はインクルードエラーを出力します。

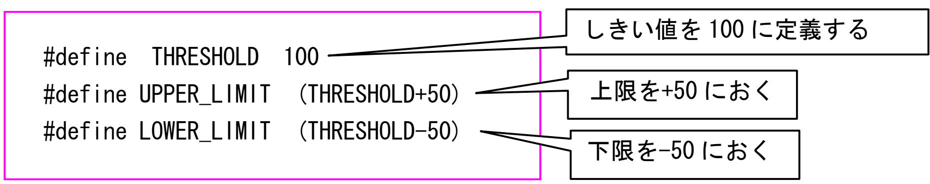


1.9.3　マクロの定義

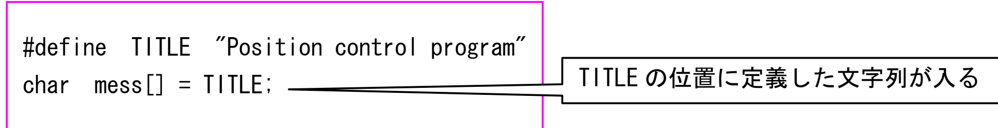
　文字列の置換やマクロの定義には「#define　識別子」を使用します。識別子は通常、変数や関数と区別するために、一般的には大文字を使用します。
　この項では、マクロの定義と取り消し方法を説明します。

定数の定義

　C 言語では定数に名前を付けることができます。プログラム中のマジックナンバー(意味不明の即値)をなくすためや、定義を共通化するのに有効な方法です。



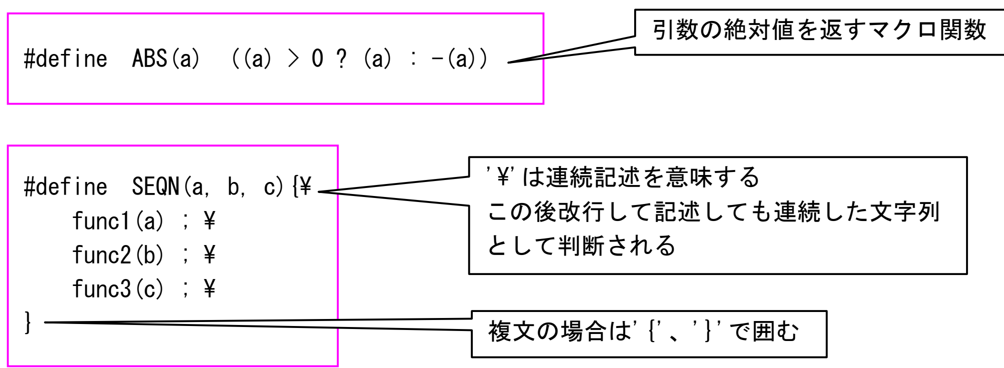
文字列の定義



文字列に名前を付ける場合に使用します。

マクロ関数の定義

"#define"を使用するとマクロ関数を定義することもできます。このマクロ関数では通常の関数と同様、引数・戻り値の受け渡しができます。しかも通常の関数のように入口処理と出口処理がないために実行速度が速くなります。
また、マクロ関数の場合、引数のデータ型を宣言する必要はありません。



定義の取り消し

```
#undef　識別子
```

"#define"で定義した識別子の置換を"#undef"以降行いません。
ただし、以下の８つの識別子はコンパイラの予約語のため、"#undef"で無効にしないでください。

・__FILE__　;　ソースファイルの名前
・__LINE__　;　現在のソースファイルの行番号
・__DATE__　;　コンパイルの日付
・__TIME__　;　コンパイルの時間
・__IAR_SYSTEMS_ICC__;　ICC コンパイラ識別子
・__STDC__　;　ICC コンパイラ識別子
・__TID__　;　ターゲット識別子
・__VER__　;　コンパイラのバージョン番号

1.9.4　条件コンパイル

ICC740 では 3 種類の条件でコンパイルを制御できます。仕様による関数の切り換え、デバッグ用関数の組み込みの有無を制御するときなどに使用します。

この項では、条件コンパイルの種類と記述方法を説明します。

いろいろな条件コンパイル

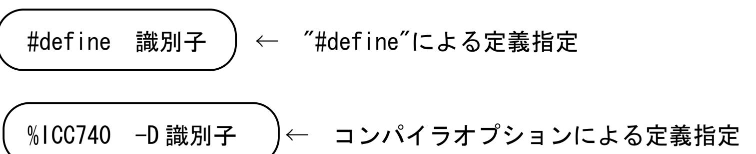
ICC740 で使用できる条件コンパイルの種類を示します。

記述方法	内容
#if　定数式 　　A #else 　　B #endif	定数式が真(0 でない)の場合は A のブロックをコンパイルし、 真でない場合は B のブロックをコンパイルする
#ifdef　マクロ名 　　A #else 　　B #endif	マクロ名が定義されている場合は A のブロックをコンパイルし、 定義されていない場合は、B のブロックをコンパイルする
#ifndef　マクロ名 　　A #else 　　B #endif	マクロ名が定義されていない場合は A のブロックをコンパイルし、 定義されている場合は B のブロックをコンパイルする

3 種類とも"#else"のブロックは省略可能です。また 3 つ以上のブロックに分類したい場合は"#elif"で条件を追加してください。

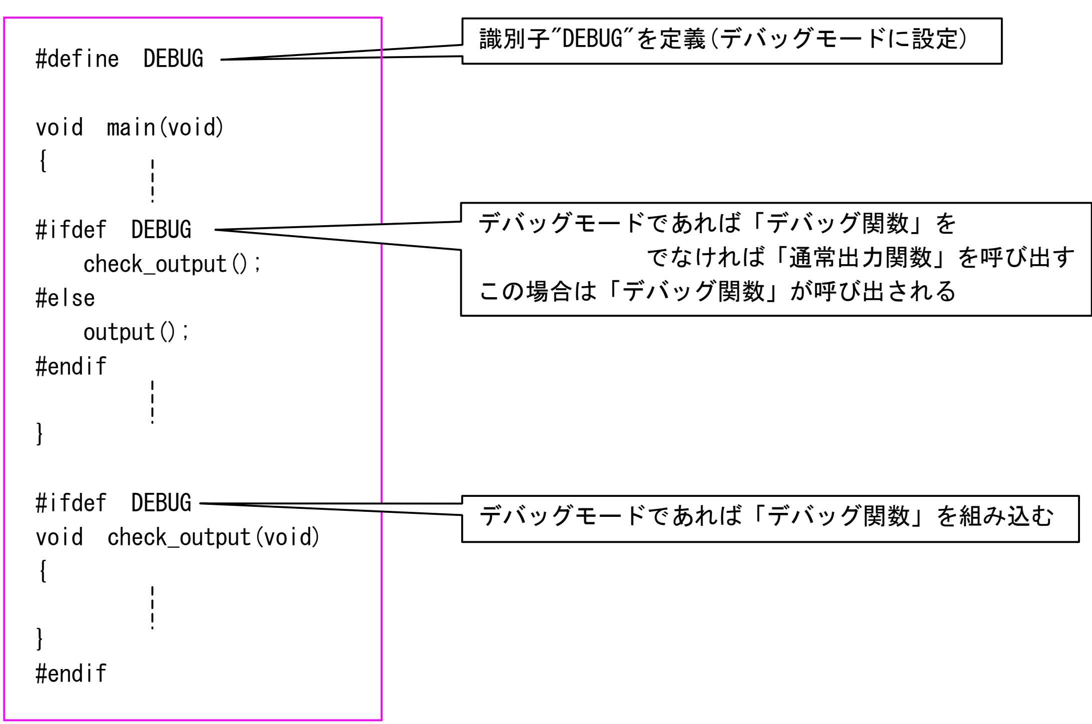
識別子の定義指定

識別子の定義指定は"#define"または ICC740 のコンパイラオプション'-D'によって指定します。



条件コンパイル記述例

条件コンパイルを利用して、デバッグ用関数の組み込みを制御した例を示します。



第　2　章
プロジェクトの設定

2.1　設定内容

2.2　メモリモデルの説明

2.3　セグメント構成

2.4　スタックの領域の説明

2.5　オブジェクトフォーマットの説明

2.6　Cスタートアップ･モジュールの説明

2.7　特殊な領域への値の設定方法

　この章では、メモリモデル、セグメントの構成、スタック領域、オブジェクトフォーマット、Cスタートアップモジュールの説明、特殊な領域への値の設定方法について説明します。

2.1　設定内容

　ICC740 ではプロセッサ・グループ、メモリモデル、スタック領域を設定して、プログラムの開発を行います。次に各項目の設定内容一覧を示します。

項目	選択肢	デフォルト設定
プロセッサ・グループ	MUL/DIV 命令有り	○
	MUL/DIV 命令無し	
	MUL/DIV 命令有り、 拡張データアクセス有り	
メモリモデル	Large モデル	○
	Tiny モデル	
	ゼロページモデル	
スタック領域	1 ページ(100H~1FFH)	○
	0 ページ(00H~FFH)	
オブジェクトフォーマット	UBROF(IAR format)	
	IEEE695(for HEW)	○
	intel-standard(for ROM)	
	motorola(for ROM)	

2.2　メモリモデルの説明

ICC740では以下のメモリモデルでプロジェクトを作成します。

1）Largeモデル
Largeモデルは、変数のデフォルトの配置位置がゼロページ以外(0x100番地以降）に配置されます。

2）Tinyモデル
Tinyモデルは、変数のデフォルトの配置位置がゼロページ（0x0～0xFF番地）に配置されます。

3）ゼロページモデル
ゼロページモデルは、ゼロページのみしか使えません。

2.2.1　メモリモデルの詳細

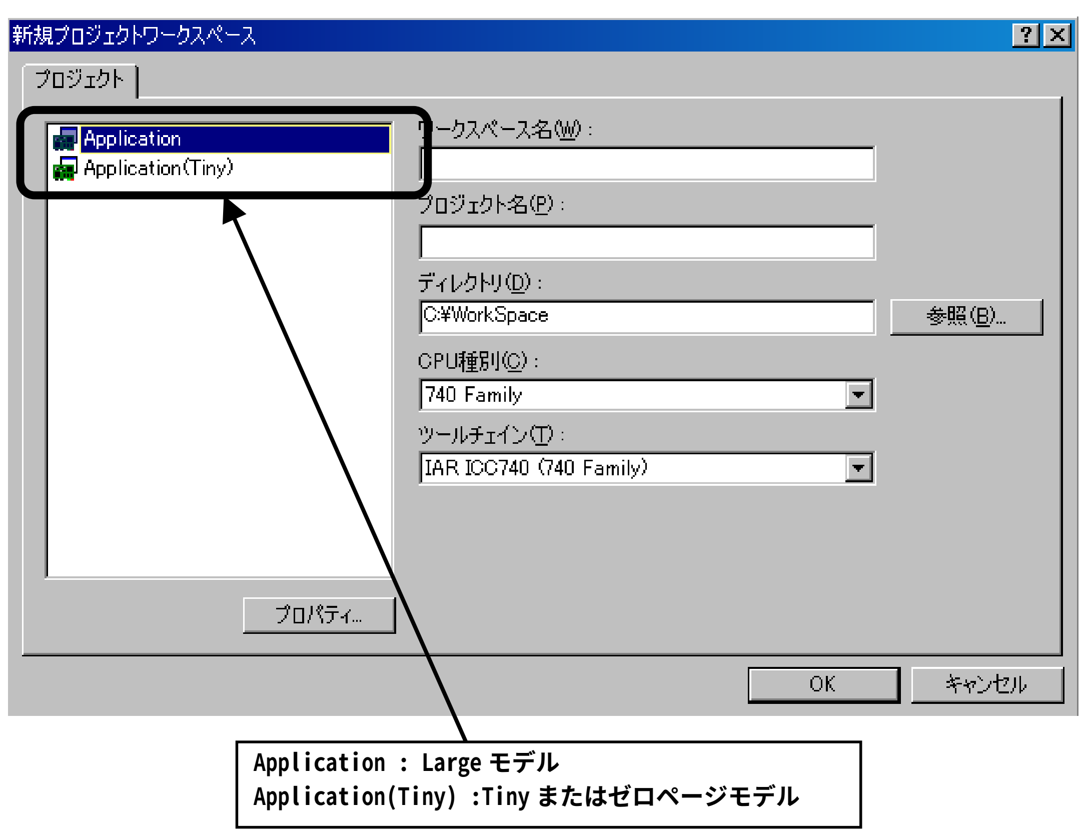
それぞれのメモリモデルの違いを下記に示します。

項目	Large モデル	Tiny モデル	ゼロページモデル
変数の配置場所	0 x 100 番地以降	0 x FF 番地以下	0 x FF 番地以下
C 言語での 0 x 100 番地以降への配置	―	npage を使用しての定義 npage　int　v1; extern　npag　int　v2;	配置不可
C 言語での 0 x FF 番地以下への配置	zpage を使用しての定義 zpage　int　v3; extern　zpag　int　v4;	―	―
アセンブルソースプログラムでの 0 x 100 番地以降のアクセス方法	―	オペランド np:を使用 lda　np:v1	アクセス不可
アセンブルソースプログラムでの 0 x FF 番地以下のアクセス方法	オペランド zp:を使用することでコードサイズ削減 lda　zp:v3	―	―

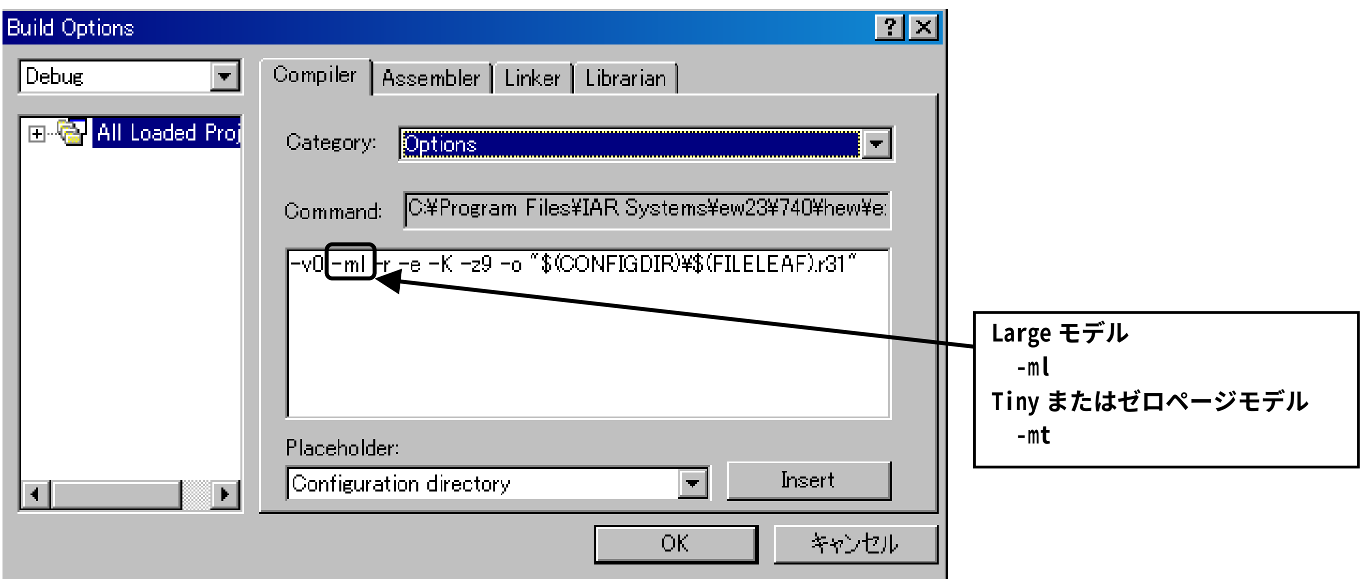
拡張キーワード zpage，npage は外部変数、自動変数、および関数引数に指定可能

2.2.2　メモリモデルの変更

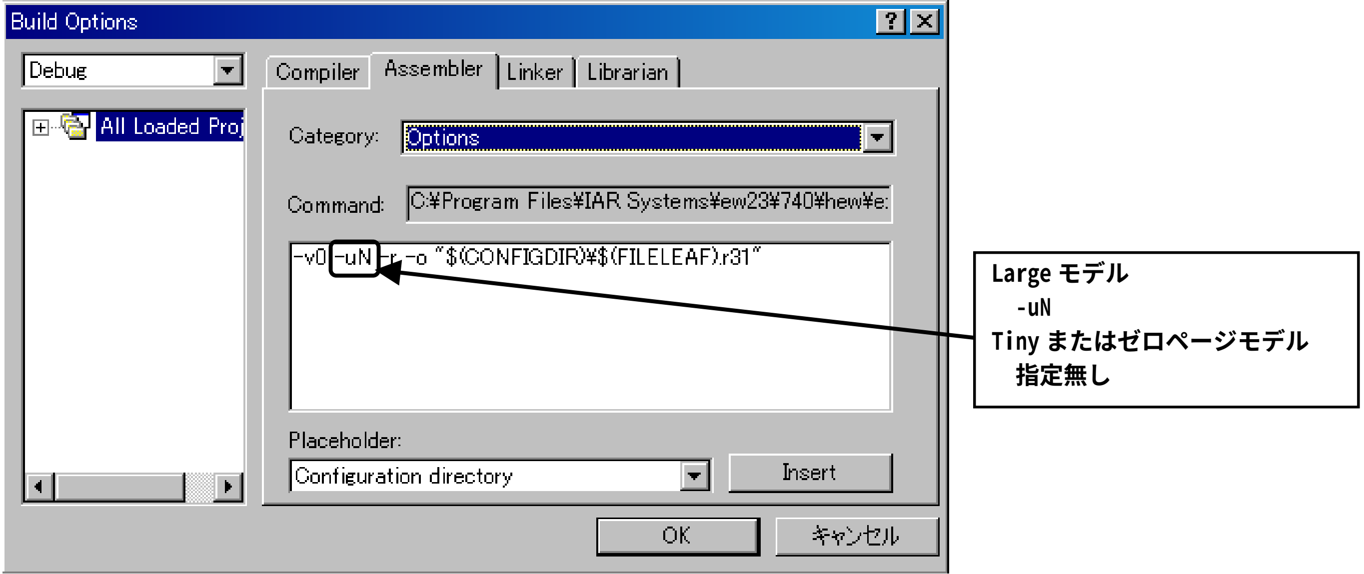
メモリモデルを変更するには、新規プロジェクト作成時にプロジェクト種別を指定します。
デフォルト<Application>のメモリモデルはLargeモデルに設定されています。
Tinyモデルまたは、ゼロページモデルに変更するには、<Application(Tiny)>を指定します。



設定内容を確認する場合は、メニューの[ビルド->IAR　ICC740　Toolchain...]を指定して Build Options を開きます。



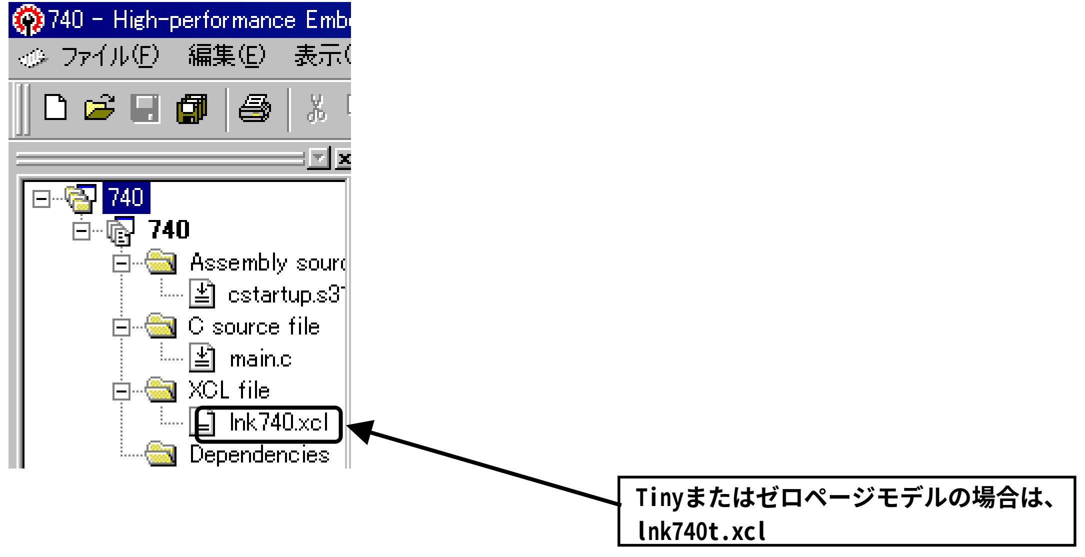
Compiler タブの-m オプションが、Large モデルの場合は-ml に設定されていますが、Tiny モデルまたはゼロページモデルの場合は-mt に設定されています。



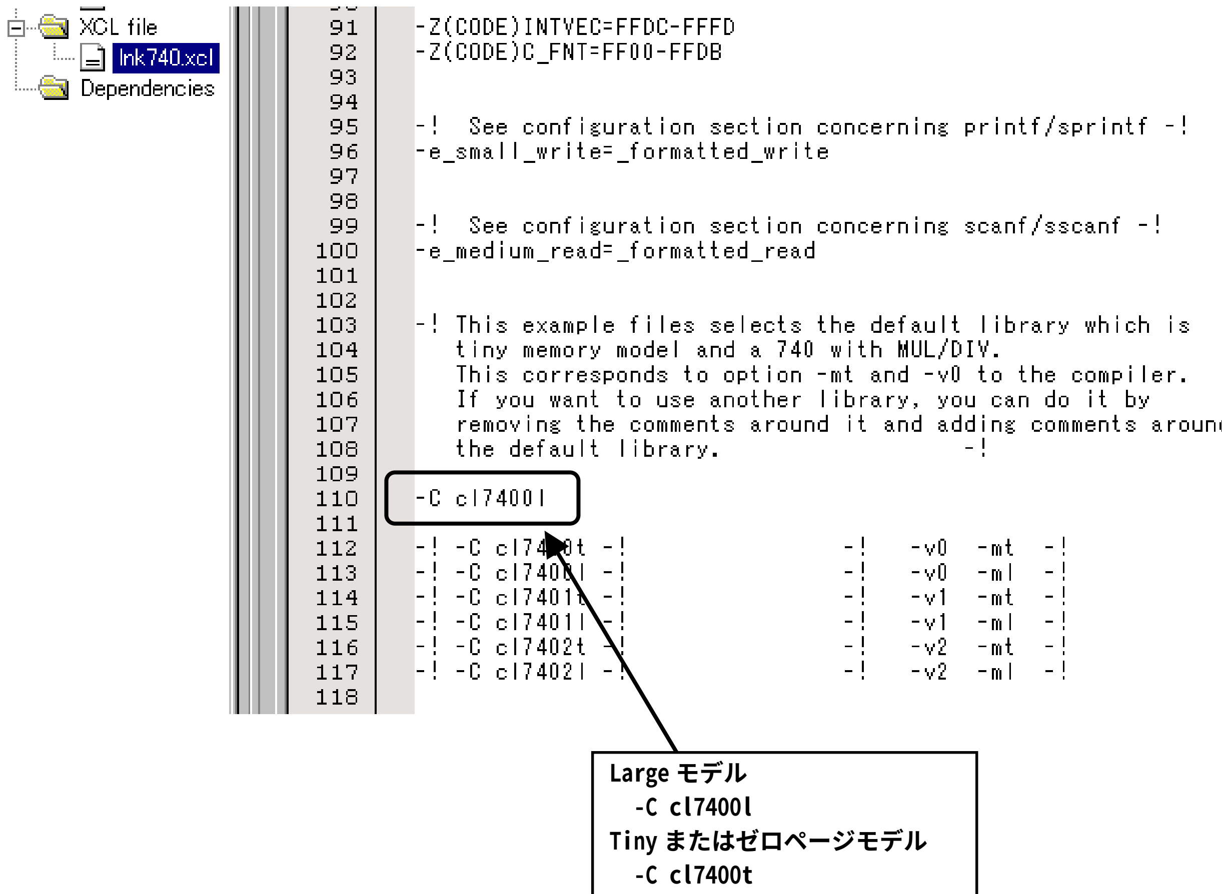
Assembler タブでは、Large モデルの場合は-uN オプションが指定されていますが、Tiny モデルまたはゼロページモデルの場合は-uN オプションは設定されません。
-uN オプションは、16 ビットアドレッシングを使用します。以下は影響を受けません。

8 ビットアドレッシング指定：
```
LDA   ZP:label
```
16 ビットアドレッシング指定：
```
LDA   NP:label
```

次にリンカの設定を確認します。
リンカの設定は、Linker タブ内ではなく、リンク・コマンド・ファイルを確認します。
リンク・コマンド・ファイルは、ワークスペース内の lnk740.xcl ファイルを開きます。
Tiny またはゼロページモデルの場合は、lnk740t.xcl を開きます。



lnk740.xcl ファイルの最後の方に-C オプションでライブラリが指定されています。
Large モデルの場合は cl7400l.r31 に設定されていますが、Tiny モデルまたはゼロページモデルの場合は cl7400t.r31 に設定されています。ゼロページモデルでは Tiny モデルから CSTACK セグメントの配置を変更する必要があります。85 ページを参照して下さい。



2.3　セグメント構成

2.3.1　ICC740のセグメント構成

ICC740 のセグメント構成の分類を示します。

・ICC740 では、740 ファミリの資源を有効に活用するため、メモリを分類します。

- Z ページ RAM (0H~FFH)：ゼロページ
  (Zero PAGE)
- N ページ RAM (100H~)
  (Normal PAGE or Non-zero PAGE)
- ROM(~FFFFH)

・それぞれにセグメントを設定してデータを管理します。

Memory

0000H

SFR

Zページ
RAM

0100H

Nページ
RAM

ROM

FFFFH

2.3.2 セグメントマップ：ZページRAM(0H～FFH)

セグメントマップ：Zページ RAM(0H～FFH)について下記に示します。

アドレス	セグメント	説明
0H	SFR	
	(BITVARS)	アドレス未確定のbit変数
	ZPAGE	ライブラリ・データ（ゼロページ配置）
	C_ARGZ	zpageを使用したauto変数、引数：func(zpage int a)
	Z_UDATA	zpageを使用した初期化無し外部変数：zpage short c;
	Z_IDATA	zpageを使用した初期化有り外部変数：zpage char b=1;
	EXPR_STACK	式スタック：レジスタXで操作します。 サイズをlnk740.xclで指定します。
	INT_EXPR_STACK	割り込み時の式スタック：レジスタXで操作します。 サイズをlnk740.xclで指定します。
FFH		

名前（概略）	タイプ	説明
BITVAR (ビット変数)	読み書き用	アセンブリ・アクセス可能 静的 bit 変数を保持します。このセグメントは、ゼロ・ページ（0-0xFF）に位置づけされなければなりません。
ZPAGE (ゼロ・ページ・アセンブラのライブラリ・データ)	読み書き用	コンパイラ専用 ゼロページ内部ライブラリ変数を保持します。このセグメントは、ゼロ・ページ（0-0xFF）に配置されなければなりません。
C_ARGZ (ローカル変数)	読み書き用	アセンブリ・アクセス可能 静的 auto 変数を保持します。このセグメントは、ゼロ・ページ（(0-0xFF）に配置されなければなりません。
Z_UDATA (非初期化静的変数)	読み書き用	アセンブリ・アクセス可能 明示的に初期化されることのない、メモリの変数を保持します。これらの変数はすべて、黙示的に 0 に初期化されます。この初期化は、CSTARTUP によって実行されます。 このセグメントは、ゼロ・ページ（0-0xFF）に配置されなければなりません。
Z_IDATA (初期化済み静的変数)	読み書き用	アセンブリ・アクセス可能 cstartup.s31 のZ_CDATAから自動的に初期化される、内部データ・メモリの静的変数を保持します。Z_CDATA(注1)も参照してください。 このセグメントは、ゼロ・ページ（0-0xFF）に配置されなければなりません。
EXPR_STACK (式スタック)	読み書き用	アセンブリ・アクセス可能 通常の処理で式の評価を行っている間一時的に結果を保持します。このセグメントは、ゼロ・ページ（0-0xFF）に位置づけされなければなりません。
INT_EXPR_STACK (割込み式スタック)	読み書き用	アセンブリ・アクセス可能 割込み処理で式の評価を行っている間一時的に結果を保持します。このセグメントは、ゼロ・ページ（0-0xFF）に位置づけされなければなりません。

(注1) Z_CDATA(初期化定数)
タイプ: 読出し専用
説明：アセンブリ・アクセス可能
CSTARTUPは、このセグメントからZ_IDATAセグメントへ初期化値をコピーします。

2.3.3　セグメントマップ：NページRAM(100H～)

セグメントマップ：Nページ RAM(100H～)について下記に示します。

100H

セグメント	説明
CSTACK	通常のスタック：サイズはlnk740.xclで指定します。
NPAGE	ライブラリ・データ（ゼロページ以外配置）
C_ARGN	auto変数、引数：func(int a)
N_UDATA	初期化無し外部変数：short c;
N_IDATA	初期化有り外部変数：char b=1;
ECSTR	書き込み可能文字列。ICC740オプション-y指定時に確保されます。
RF_STACK	再帰呼出し時のスタック：通常0。使用時258バイト以上確保されます。

23FH

RAMの最終アドレス(マイコンにより異なる)

名前（概略）	タイプ	説明
CSTACK (スタック)	読み書き用	アセンブリ・アクセス可能 ハードウェア・スタックを保持します。このセグメントは、CSTARTUP で設定された位置に応じてゼロページ（0-0xFF)、あるいはページ 1（0x100-0x1FF）に位置づけされなければなりません。 通常、このセグメントと長さは次のコマンドで XLINK ファイルに定義されます。 -Z(DATA)CSTACK + nn = start ここで、DATA は ZPAGE または NPAGE、nn は長さ、start は位置を示します。
NPAGE (正規ページ・アセンブラのライブラリ・データ)	読み書き用	コンパイラ専用 非ゼロ・ページ内部ライブラリ変数を保持します。
C_ARGN (ローカル変数)	読み書き用	アセンブリ・アクセス可能 静的 auto 変数を保持します。このセグメントは、N・ページ（0x100 以降）に配置されなければなりません。
N_UDATA (非初期化静的変数)	読み書き用	アセンブリ・アクセス可能 明示的に初期化されることのない、メモリの変数を保持します。これらの変数はすべて、黙示的に 0 に初期化されます。この初期化は、CSTARTUP によって実行されます。
N_IDATA (初期化済み静的変数)	読み書き用	アセンブリ・アクセス可能 cstartup.s31 の N_CDATA から自動的に初期化される、内部データ・メモリの静的変数を保持します。2.3.4 セグメントマップ：ROM(～FFFFH～)の 81 ページの N_CDATA も参照してください。

ECSTR **(書込み可能な文字列リテラルのコピー)**	**読み書き用**	**アセンブリ・アクセス可能** **C文字列リテラルを保持します。詳細については、「Writable strings (-y) 」(注1)を参照してください。-yオプションについては、３章コンパイラ：ICC740 の 103 ページの- y を参照してください。また、「WCSTR」(注2)、2.3.4 セグメントマップ：ROM(～FFFFH)の 81 ページの「CSTR」、82 ページの「CCSTR」も参照してください。**
RF_STACK **(再帰的スタック)**	**読み書き用**	**アセンブリ・アクセス可能** **再帰的関数の囲い込み呼出しのローカル変数とパラメータを保持します。RF_STACK は 1 ページ以降に配置され、且つコンパイラが 256 バイトのサイズを確保しますので、ゼロページモデルでは再帰呼び出しは使用できません。**

(注1)　Writable strings (-y)

構文：　-y

コンパイラに、文字列リテラルを書込み可能変数としてコンパイルするようにさせます。

通常、文字列リテラルは読み取り専用でコンパイルされます。文字列リテラルを書込み可能にするには、Writable strings (-y) オプションを使用し、文字列を書込み可能変数としてコンパイルさせます。

注記：文字列で初期化される配列（すなわち、char c[] = "string"）は、常に初期化済み変数としてコンパイルされ、Writable strings (-y)の影響を受けません。

(注2)　WCSTR（書込み可能な文字列リテラル）

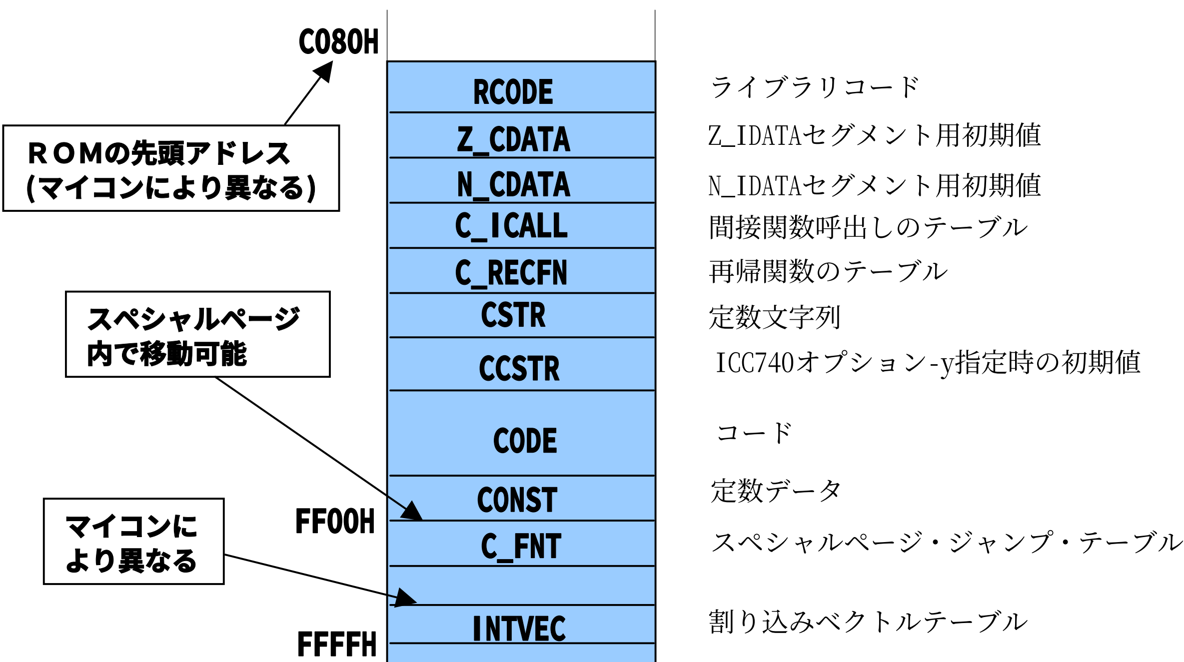
タイプ: 読み書き用

説明：アセンブリ・アクセス可能

通常、文字列はCSTR（ROM）あるいはWCSTR（RAM）領域に置かれます。書込み可能文字列やPROM化可能文字列を指定している場合、特殊セグメントCCSTR（ROM）は文字列を保持します。一方、ECSTR(RAM）は同じ容量のスペースをもちます。実行時には、CCSTRはECSTRにコピーされると想定されます。

2.3.4　セグメントマップ：ROM(～FFFFH)

セグメントマップ：ROM(～FFFFH)について下記に示します。



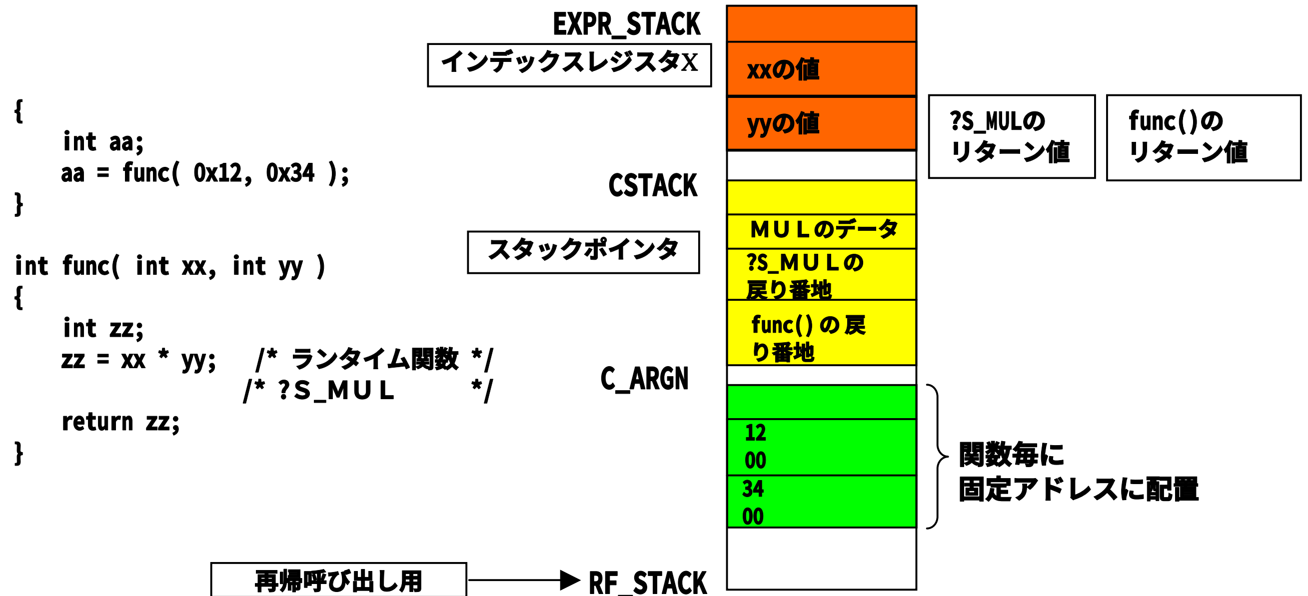
名前（概略）	タイプ	説明
RCODE (スタートアップ・コード)	読出し専用	コード生成組込み関数で使用されるアセンブリ・アクセス可能コード このセグメントは、C から呼び出すことのできないユーザ記述アセンブラ・コード（割込み処理ルーチンおよび同等の常駐コード）にも使用されます。
Z_CDATA （初期化定数）	読出し専用	アセンブリ・アクセス可能 CSTARTUP は、このセグメントから Z_IDATA セグメントへ初期化値をコピーします。
N_CDATA (初期化定数)	読出し専用	アセンブリ・アクセス可能 CSTARTUP は、このセグメントから N_IDATA セグメントへ初期化値をコピーします。
C_ICALL (間接関数呼出しのテーブル)	読出し専用	コンパイラ専用 間接関数呼出しのテーブルを保持します。
C_RECFN (再帰的関数のテーブル)	読出し専用	コンパイラ専用 再帰的関数テーブルを保持します。
CSTR (文字列リテラル)	読出し専用	アセンブリ・アクセス可能 C コンパイラの -y オプションが実行可能状態にあるときに C 文字列リテラルを保持します。詳細については、３章コンパイラ：ICC740 の 103 ページの- y を参照してください。 また、2.3.3 セグメントマップ：NページRAM(100H～)の 80 ページの「ECSTR」、「WCSTR」(注2)、82 ページの「CCSTR」も参照してください。

CCSTR (文字列リテラル)	読出し専用	アセンブリ・アクセス可能 C文字列リテラルを保持します。詳細については、３章コンパイラ：ICC740 の 103 ページの-ｙを参照してください。また、2.3.3 セグメントマップ：NページRAM(100H～)の 80 ページの「ECSTR」、「WCSTR」(注2)、81 ページの「CSTR」も参照してください。
CODE (コード)	読出し専用	アセンブリ・アクセス可能 ユーザ・プログラム・コードおよび各種のライブラリ・ルーチンを保持します。
CONST (定数)	読出し専用	アセンブリ・アクセス可能 const オブジェクトの格納に使用されます。定数データの宣言のためにアセンブリ言語ルーチンで使用されます。
C_FNT (特殊ページ分岐テーブル)	読出し専用	アセンブリ・アクセス可能 tiny_func 呼出し規則によって呼び出された関数のアドレスを保持します。このセグメントは、スペシャルページ(FF00H～FFFFH)に置かなければなりません。
INTVEC (割込みベクトル)	読出し専用	アセンブリ・アクセス可能 interrupt 拡張キーワード（ユーザ記述割込みベクトル・テーブル・エントリにも使用されます）を使用した場合に生成される割込みベクトル・テーブルを保持します。このセグメントの始まりは、割り込みベクトル領域の開始アドレスでなければなりません。

2.4　スタック領域の説明

2.4.1　ICC740スタック管理

ICC740 ではスタック管理を複数のセグメントを用いて行っています。
ICC740 スタック管理について下記の図に示します。



EXPR_STACK ／ INT_EXPR_STACK セグメント

関数のリターン値、テンポラリ変数領域として使用します。
C ランタイム関数の引数、テンポラリ変数、およびリターン値領域としても使用します。
インデックスレジスタ X で操作。0x00～0xFF 内に配置します。
通常時には EXPR_STACK セグメント、割り込み時に INT_ EXPR_STACK セグメントに自動切り替えをします。
※78 ページの 2.3.2 セグメントマップ：Z ページ RAM(0H～FFH)の EXPR_STACK、INT_EXPR_STACK を参照して下さい。

CSTACK セグメント

JSR、PHA、MUL、DIV 命令等で使用します。
割り込み発生時の戻り番地、レジスタ退避にも使用します。
CPU モードレジスタの設定により 0x00～0xFF または 0x100～0x1FF 内に配置します。
※79 ページの 2.3.3 セグメントマップ：N ページ RAM(100H～)の CSTACK を参照して下さい。

C_ARGN ／ C_ARGZ セグメント

関数のローカル変数領域です。
各ローカル変数が静的（固有アドレス）に配置されます。
全てのローカル変数が別アドレスは配置されると大量の RAM が必要となるため、リンカ XLINK では、コンパイラが出力する疑似命令 DEFFN の情報をもとに上位関数のローカル変数を破壊しないよう、共有アドレスに配置することで RAM の削減を図っています。
※C_ARGZセグメントは78ページの2.3.2セグメントマップ：ZページRAM(0H～FFH)のC_ARGZ、
C_ARGN セグメントは 79 ページの 2.3.3 セグメントマップ：N ページ RAM(100H～)の C_ARGN を参照して下さい。

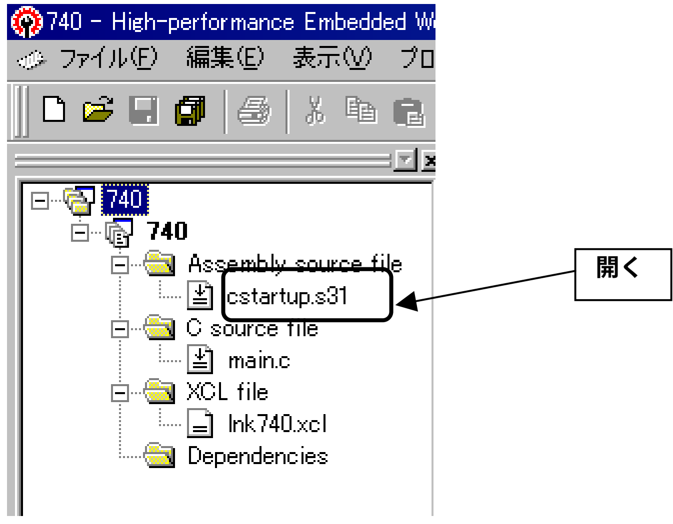
RF_STACK セグメント

再帰呼び出し時のローカル変数変数領域。 0x100 以降に配置します。
リンカで再帰呼び出しを判断し、再帰呼び出しが無い場合は 0 バイト、
ある場合は 256 バイト（ローカル変数用 256 バイト、管理用 2 バイト）をリンカが設定します。
※79 ページの 2.3.3 セグメントマップ：N ページ RAM(100H～)の RF_STACK を参照して下さい。

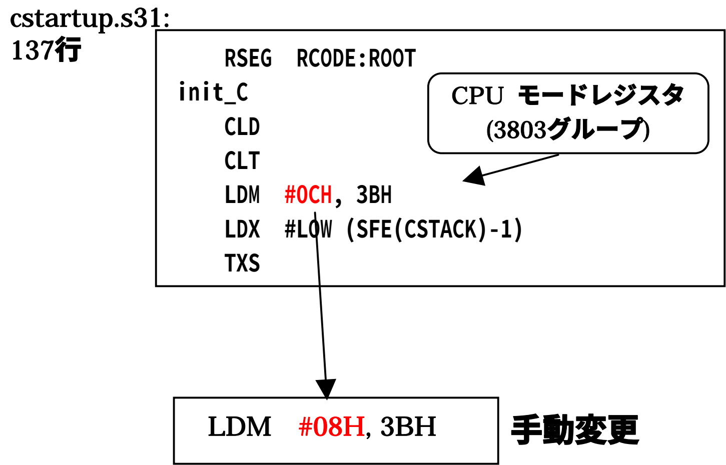
2.4.2　CSTACK セグメントの変更

CSTACK セグメントをゼロページに変更する方法を以下に示します。

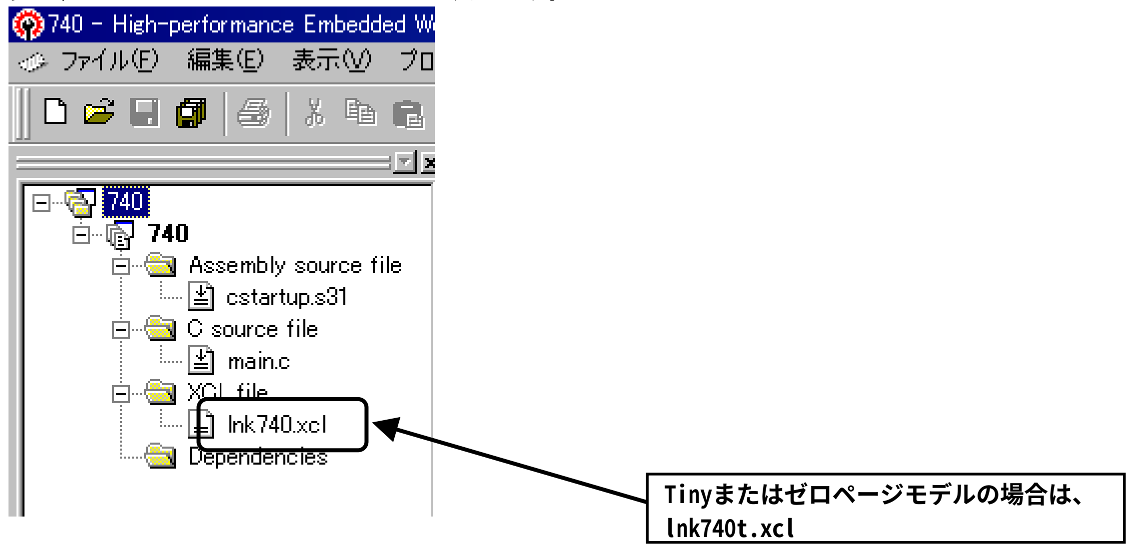
まず、ワークスペース内の cstartup.s31 を開きます。



cstartup.s31 のスタックページは 1 ページに設定(3803 グループ)されています。
ゼロページに変更するには、137 行目の「#0CH」を「#08H」に書き換えます。

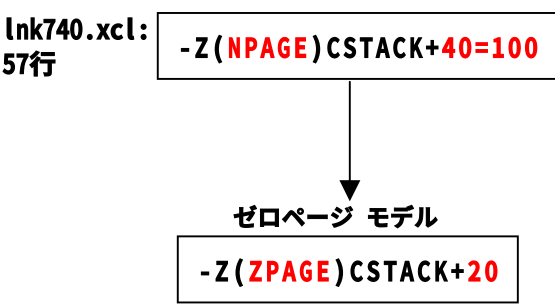


次に、ワークスペース内の lnk740.xcl を開きます。



CSTACK が 1 ページに設定されています。

ゼロページに変更するには、57 行目の「NPAGE」を「ZPAGE」ページに変更し、アドレス指定の「=100」を削除します。この変更により CSTACK はゼロページの最後のセグメントとして配置されます。

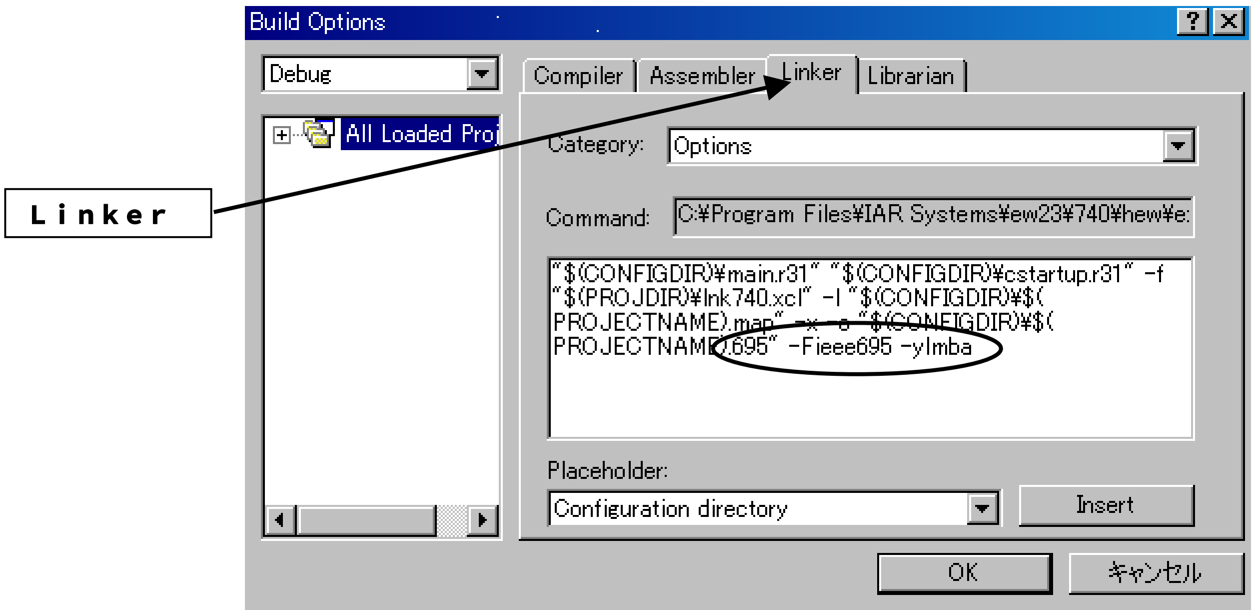


2.5 オブジェクトフォーマットの説明

2.5.1 オブジェクトフォーマットの変更

プロジェクト作成時には、IEEE695形式(注1)に設定されています。
設定内容を確認する場合は、メニューの[ビルド->IAR ICC740 Toolchain...]を指定してBuild Optionsを開きます。更にLinkerタブを選択します。

(注1) IEEE695形式はHEW上のデバッガでのデバッグに適したフォーマットです。



オブジェクト形式を変更する場合は、上記の○の中を変更してください。

インテル Hex 形式を選択する場合は、以下のように変更してください。
-o "$(CONFIGDIR)¥$(PROJECTNAME).hex" -Fintel-standard -Y0

モトローラ形式を選択する場合は、以下のように変更してください。
-o "$(CONFIGDIR)¥$(PROJECTNAME).mot" -Fmotorola

2.6　C スタートアップ・モジュールの説明

2.6.1　Cスタートアップ・モジュールの解説

M3T-ICC740 では、プロジェクト開発に C スタートアップ・モジュール cstartup.s31 を使用します。
cstartup.s31 は以下のように設定しています。

マイコン　　　　　　　　　　3803 グループ
スタックポインタ動作領域　　　1 ページ（100H～1FFH）

ここでは cstartup.s31 に解説を加えながら、必要に応じて変更する個所や方法を説明していきます。

C スタートアップ・モジュールは以下の場合に変更が必要となります。

1. init_C ルーチンのプロセッサモードレジスタを変更する。
   （スタックポインタ領域をゼロページ・メモリに変更する場合など）
2. リセット直後にポート等 SFR を設定する。
3. デフォルトの割り込みベクトル情報と異なるベクトル情報を持つターゲットマイコンを使用する。

なお、cstartup.s31 ではメモリモデルの変更に対する書き換えは必要ありません。

また、ライブラリの中に cstartup.モジュールが入っていますが、M3T-ICC740 ではこの cstartup.s31 を使用します。
M3T-ICC740 では lnk740.xcl では以下のように記述して、ライブラリの cstartup モジュールを使用しないようにしています。

```
-C cl7400l
```

この-C は削除しないでください。

cstartup.s31 の解説で使用しているセグメントの内容については、76～80 ページを参考にして下さい。

cstartup.s31 の解説

cstartup.s31：1～40 行

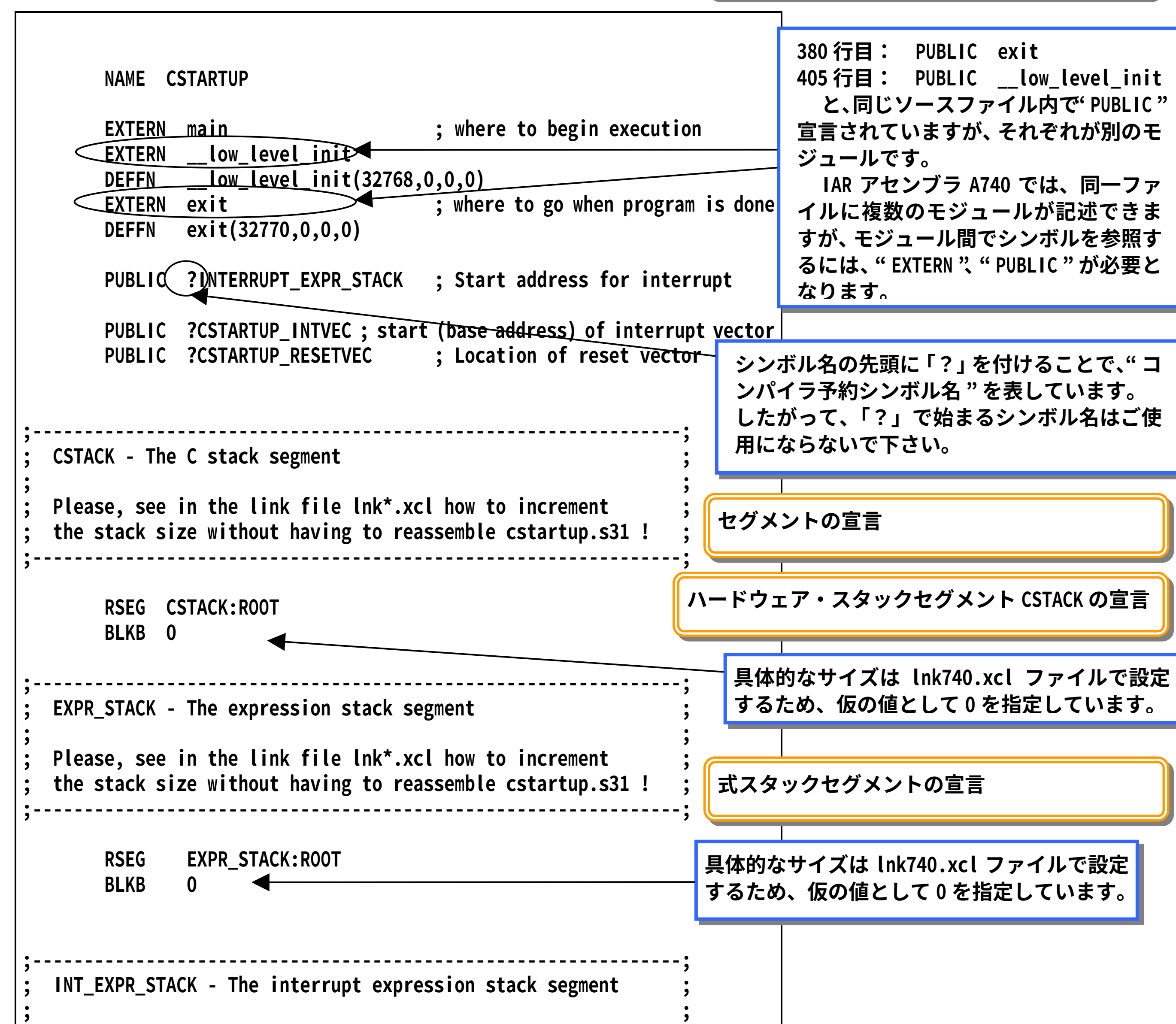
```
;----------------------------------------------------------------
;
;          cstartup.s31
;
; This file contains the MELPS 740 C startup routine
; and must usually be tailored to suit customer's hardware.
;
;
; You probably want to set up the mode register and you may
; have to change the reset vector.
;
; $Id: cstartup.s31 1.16 2001/07/16 14:16:13Z IJON Exp $
;
;----------------------------------------------------------------


#if 0
#if ((__TID__ >> 4) & 15) == 0
#define MELPS_MULDIV
#endif

#if ((__TID__ >> 4) & 15) == 2
#define MELPS_37600
#endif
#endif

;----------------------------------------------------------------
; Turning off 'interruptable ISRs':
; Do this if you need the extra byte(s)
;
; 1. Uncomment the define below
; 2. Assemble this file
; 3. Include the result in your linker command file:
;    -C cstartup.r31
;
; Variable '?IES_USAGE' and its initialization will no longer
; be included.
;----------------------------------------------------------------
;#define NO_INTERRUPTABLE_ISR
```

初期設定

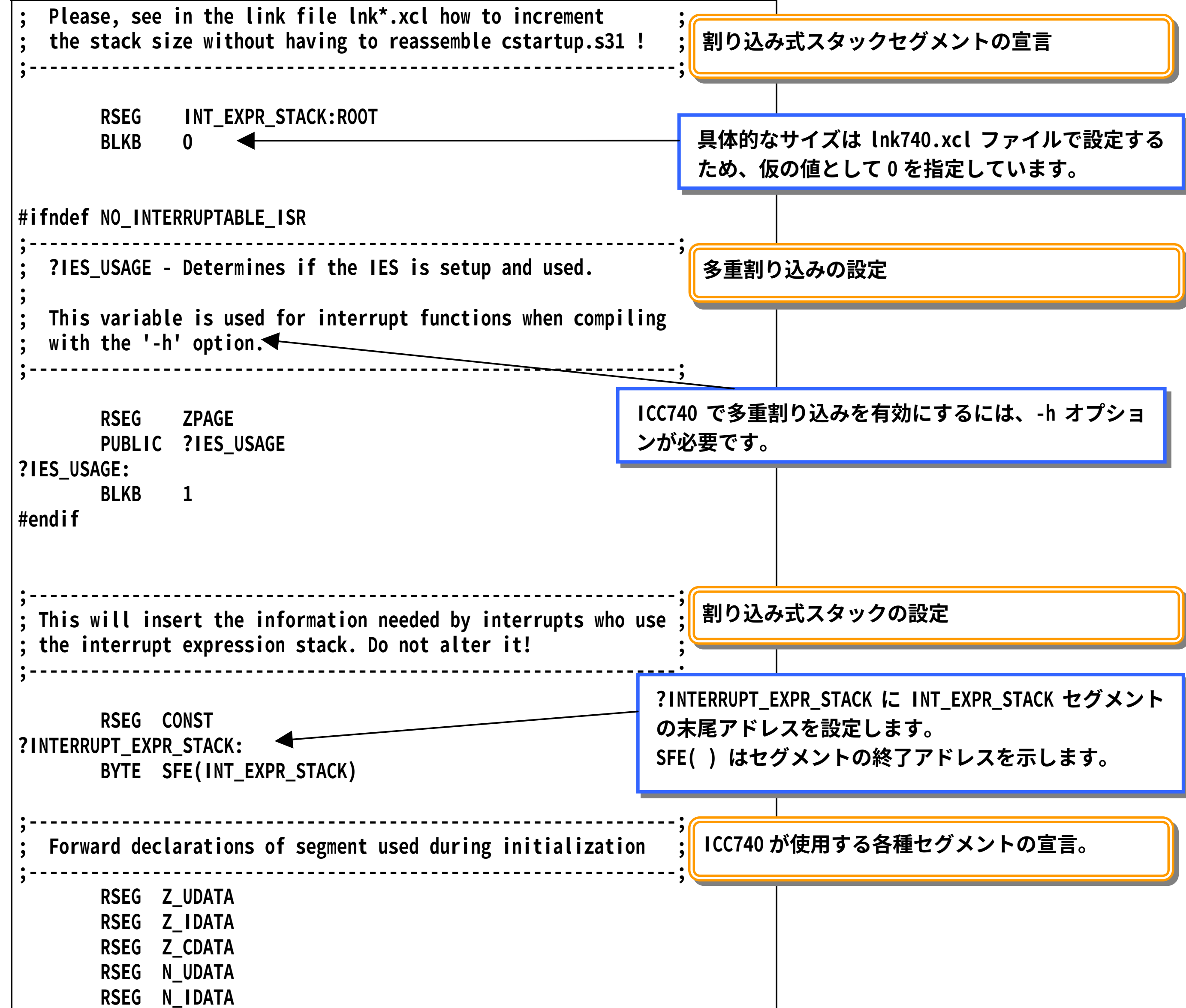
M3T-ICC740 では、この部分は使用しません。

多重割り込みを使用する場合は、先頭の ; を削除しないでください。
多重割り込みを使用しない場合、先頭の ; を削除することで、RAM 1byte, ROM 4bytes 削減できます。

cstartup.s31：41～80 行

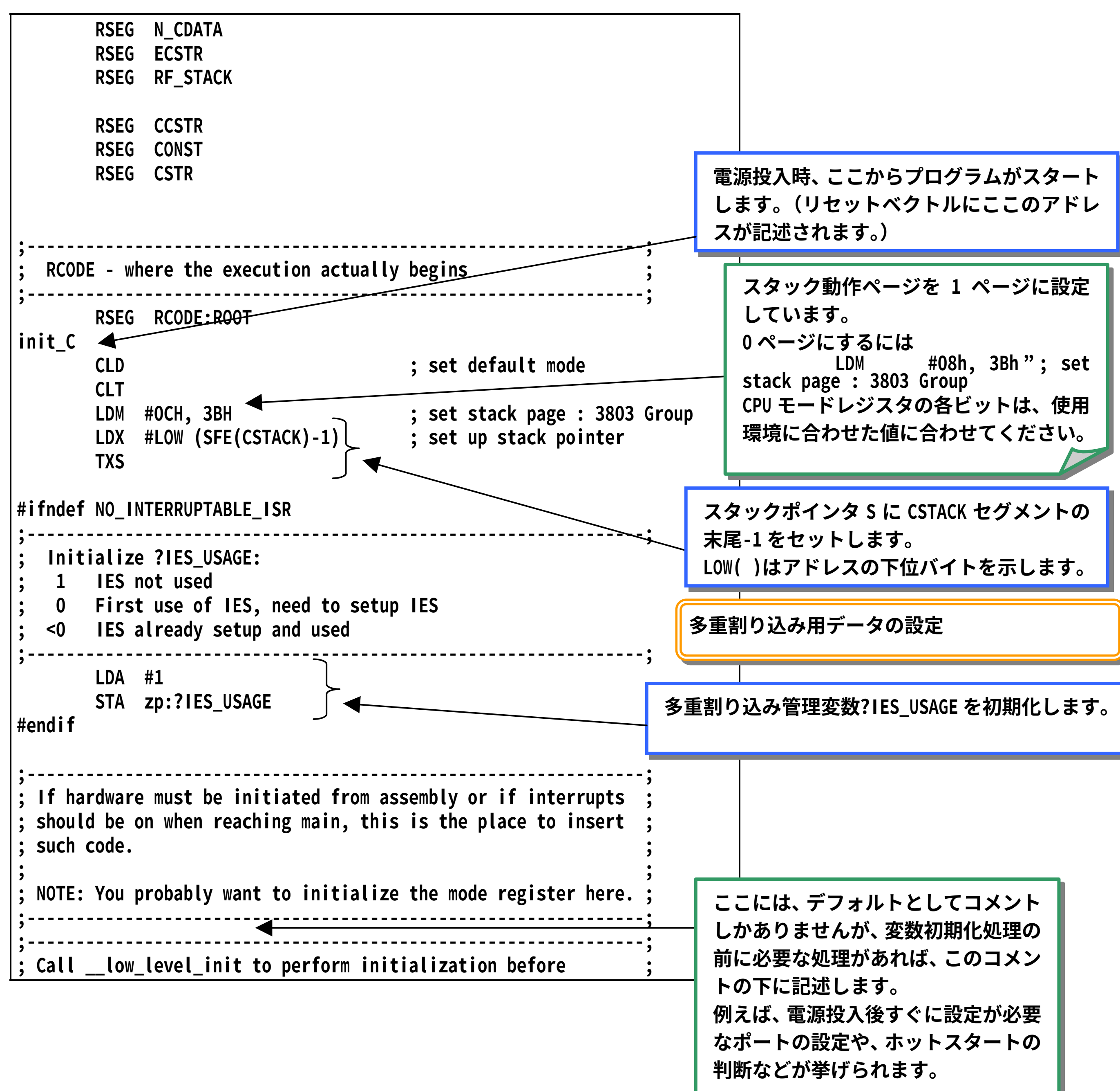


cstartup.s31：81～120 行



```
;  Please, see in the link file lnk*.xcl how to increment          ;
;  the stack size without having to reassemble cstartup.s31 !      ;
;------------------------------------------------------------------;

        RSEG    INT_EXPR_STACK:ROOT
        BLKB    0

#ifndef NO_INTERRUPTABLE_ISR
;------------------------------------------------------------------;
;  ?IES_USAGE - Determines if the IES is setup and used.
;
;  This variable is used for interrupt functions when compiling
;  with the '-h' option.
;------------------------------------------------------------------;

        RSEG    ZPAGE
        PUBLIC  ?IES_USAGE
?IES_USAGE:
        BLKB    1
#endif


;------------------------------------------------------------------;
; This will insert the information needed by interrupts who use   ;
; the interrupt expression stack. Do not alter it!                 ;
;------------------------------------------------------------------;

        RSEG  CONST
?INTERRUPT_EXPR_STACK:
        BYTE  SFE(INT_EXPR_STACK)

;------------------------------------------------------------------;
;  Forward declarations of segment used during initialization      ;
;------------------------------------------------------------------;
        RSEG  Z_UDATA
        RSEG  Z_IDATA
        RSEG  Z_CDATA
        RSEG  N_UDATA
        RSEG  N_IDATA
```

cstartup.s31：121～160 行



cstartup.s31：161～200 行

```
; initializing segments and calling main.                          ;
; If the function returns 0 no segment initialization should       ;
; take place.                                                      ;
;                                                                  ;
; Link with your own version of __low_level_init to override       ;
; the default action: to do nothing but return 1.                  ;
;------------------------------------------------------------------;

        LDX   #SFE(EXPR_STACK)        ; set up expression stack
        JSR   __low_level_init
        TAY                           ; test return value
        BEQ   skip_seg_init


;------------------------------------------------------------------;
; If it is not a requirement that static/global data is set        ;
; to zero or to some explicit value at startup, the following      ;
; line refering to seg_init can be deleted, or commented.          ;
;------------------------------------------------------------------;

        JSR   seg_init                ; initialize data segments
        LDX      #SFE(EXPR_STACK)     ; set up expression stack (again)
                                      ; as seg_init destroys it
skip_seg_init

;------------------------------------------------------------------;
; Set up expression stack                                          ;
;------------------------------------------------------------------;

expr_stack_start EQU  SFE(EXPR_STACK)

        LIMIT  expr_stack_start,0,100h,"Expression stack out of range"

        LDX      #expr_stack_start & 255  ; load initial expr stack pointer

        JSR   main       ; execute main()

;------------------------------------------------------------------;
; Now when we are ready with our C program we must perform a       ;
; system-dependent action.  In this case we just stop.             ;
```

グローバル変数の初期化（セグメントの初期化）、および Main 関数呼出しの前に __low_level_init ルーチンを呼出します。デフォルトでは、 __low_level_init は何も実行をせず、ただ、戻り値 1 を返すように記述されているだけです（403 行目～）。__low_level_init の戻り値が 0 なら（=初期化すべきセグメントがなければ）、グローバル変数の初期化処理ルーチンを飛ばします。

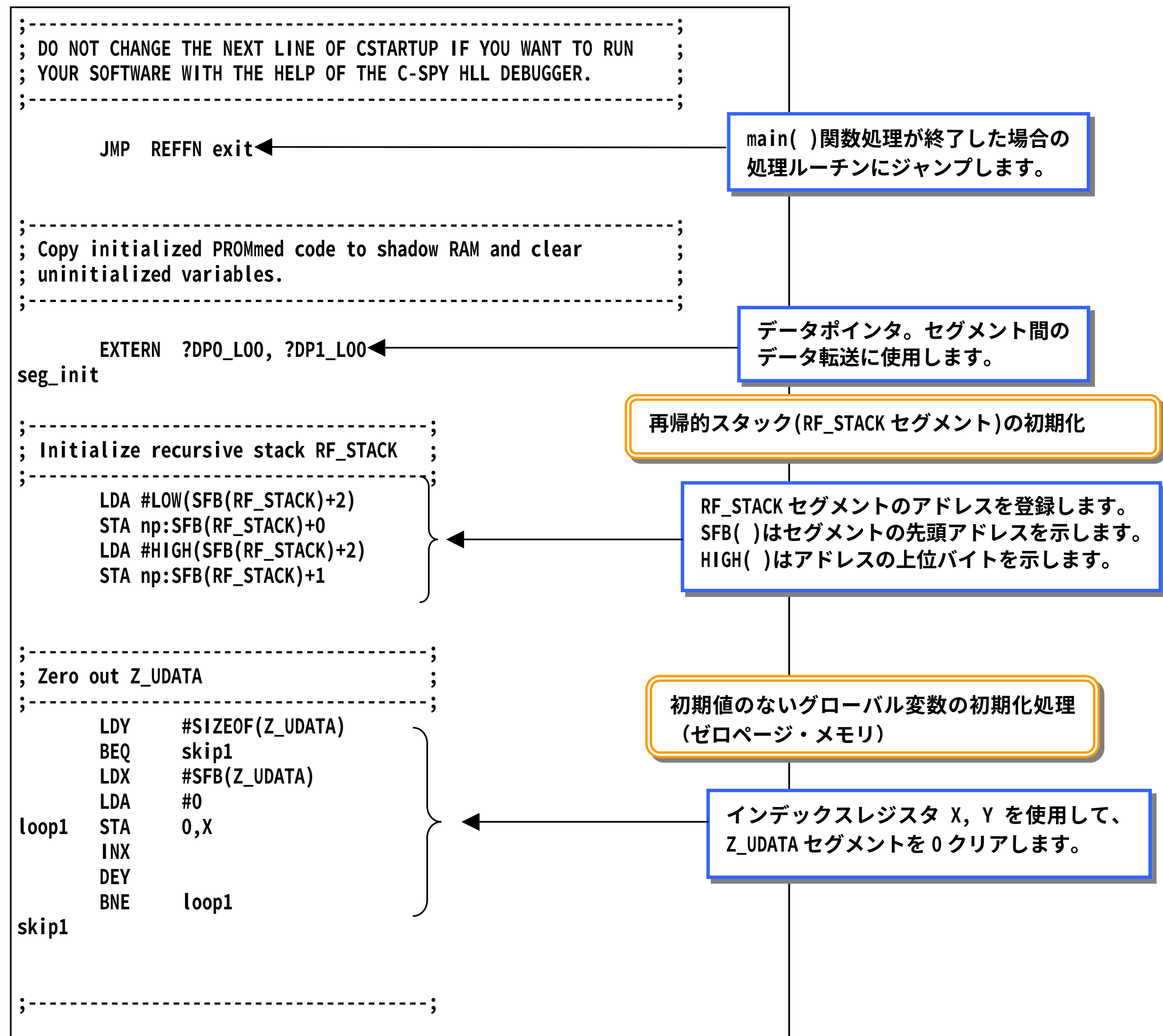
グローバル変数の初期化ルーチン seg_init を呼出します。

インデックスレジスタ X に EXPR_STACK セグメントのエンドアドレスを再度割り当てます。
インデックスレジスタ X は EXPR_STACK セグメントのデータ処理に使用します。

LIMIT はシンボルが指定範囲内にあるかどうかをチェックする擬似命令で、LIMIT ラベル,最小,最大,メッセージという書式で用い、
シンボルに指定範囲外の値が割り当てられている場合、エラーメッセージが出力されます。
expr_stack_start が 0～100h の間、つまり、ゼロページに正しく割り付けられているかをチェックしています。
なお、この部分は、オブジェクトファイルに出力されません。

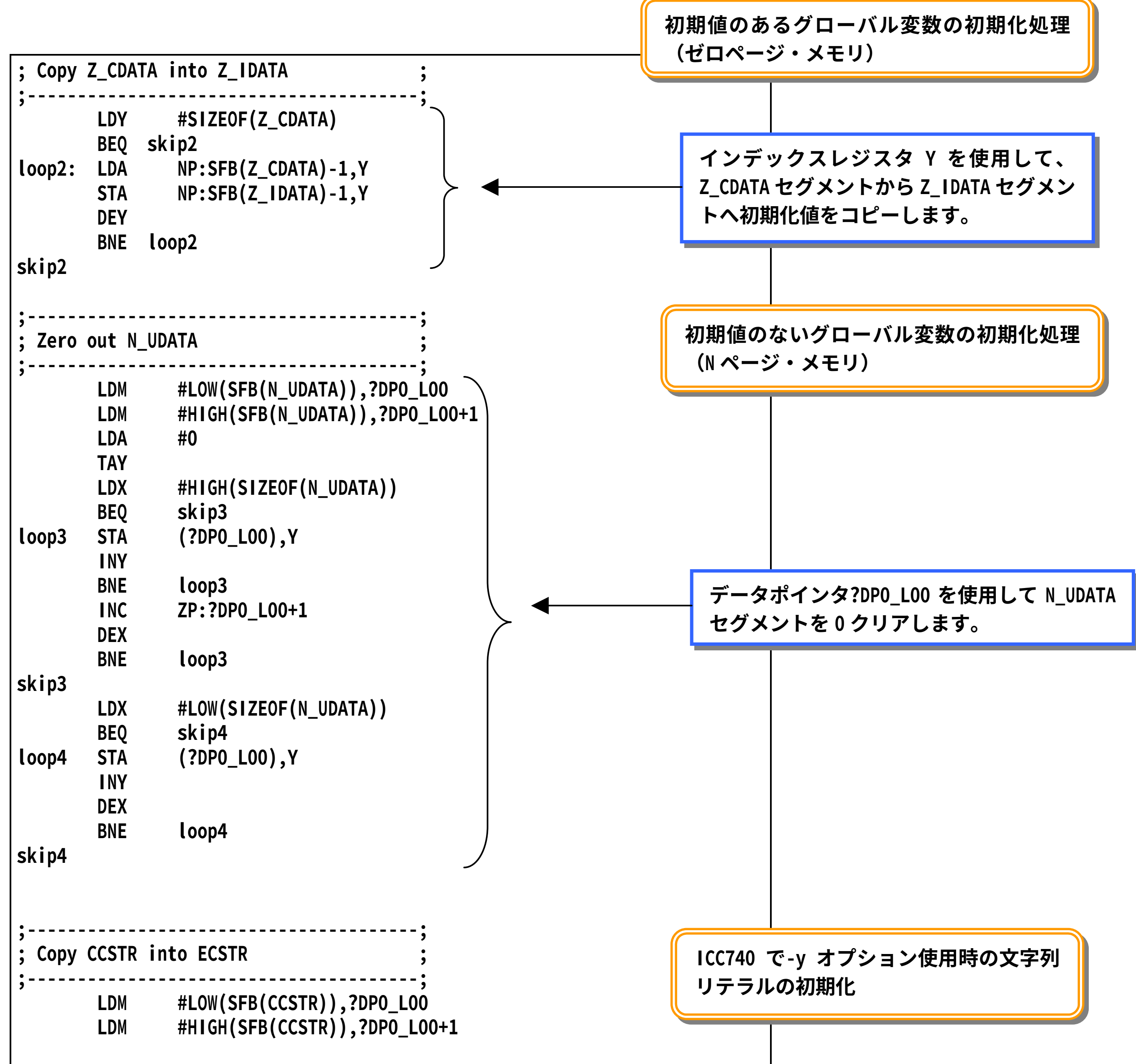
main( )関数を呼出します。

cstartup.s31：201～240 行



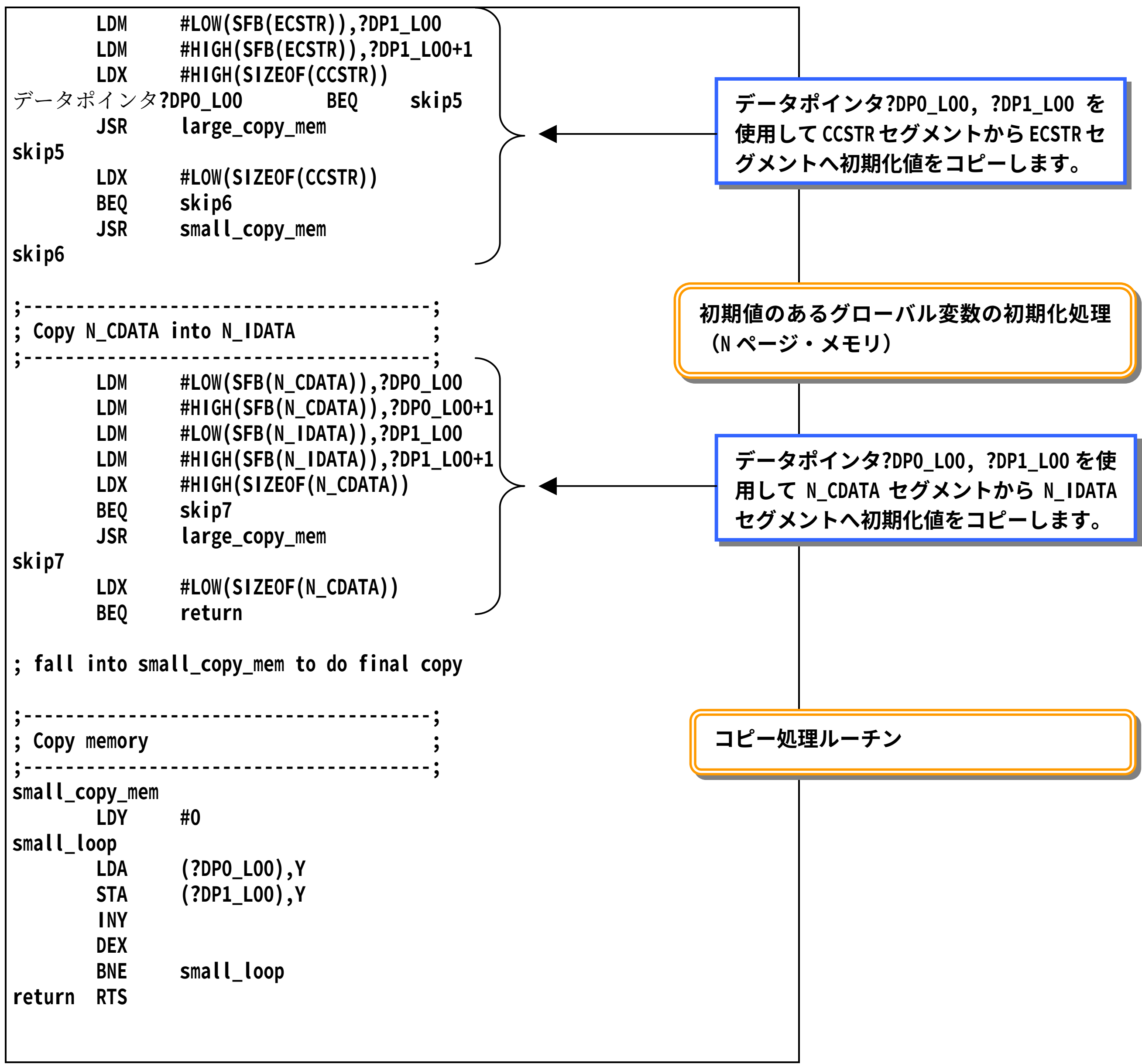
```
;--------------------------------------------------------------;
; DO NOT CHANGE THE NEXT LINE OF CSTARTUP IF YOU WANT TO RUN   ;
; YOUR SOFTWARE WITH THE HELP OF THE C-SPY HLL DEBUGGER.       ;
;--------------------------------------------------------------;

        JMP   REFFN exit


;--------------------------------------------------------------;
; Copy initialized PROMmed code to shadow RAM and clear        ;
; uninitialized variables.                                     ;
;--------------------------------------------------------------;

        EXTERN  ?DP0_L00, ?DP1_L00
seg_init

;---------------------------------------;
; Initialize recursive stack RF_STACK   ;
;---------------------------------------;
        LDA #LOW(SFB(RF_STACK)+2)
        STA np:SFB(RF_STACK)+0
        LDA #HIGH(SFB(RF_STACK)+2)
        STA np:SFB(RF_STACK)+1


;---------------------------------------;
; Zero out Z_UDATA                      ;
;---------------------------------------;
        LDY     #SIZEOF(Z_UDATA)
        BEQ     skip1
        LDX     #SFB(Z_UDATA)
        LDA     #0
loop1   STA     0,X
        INX
        DEY
        BNE     loop1
skip1


;---------------------------------------;
```

cstartup.s31：241～280 行



```
; Copy Z_CDATA into Z_IDATA                ;
;-----------------------------------------;
        LDY     #SIZEOF(Z_CDATA)
        BEQ   skip2
loop2:  LDA     NP:SFB(Z_CDATA)-1,Y
        STA     NP:SFB(Z_IDATA)-1,Y
        DEY
        BNE   loop2
skip2

;-----------------------------------------;
; Zero out N_UDATA                          ;
;-----------------------------------------;
        LDM     #LOW(SFB(N_UDATA)),?DPO_LOO
        LDM     #HIGH(SFB(N_UDATA)),?DPO_LOO+1
        LDA     #0
        TAY
        LDX     #HIGH(SIZEOF(N_UDATA))
        BEQ     skip3
loop3   STA     (?DPO_LOO),Y
        INY
        BNE     loop3
        INC     ZP:?DPO_LOO+1
        DEX
        BNE     loop3
skip3
        LDX     #LOW(SIZEOF(N_UDATA))
        BEQ     skip4
loop4   STA     (?DPO_LOO),Y
        INY
        DEX
        BNE     loop4
skip4


;-----------------------------------------;
; Copy CCSTR into ECSTR                     ;
;-----------------------------------------;
        LDM     #LOW(SFB(CCSTR)),?DPO_LOO
        LDM     #HIGH(SFB(CCSTR)),?DPO_LOO+1
```

cstartup.s31：281～320 行



```
        LDM     #LOW(SFB(ECSTR)),?DP1_L00
        LDM     #HIGH(SFB(ECSTR)),?DP1_L00+1
        LDX     #HIGH(SIZEOF(CCSTR))
データポインタ?DP0_L00        BEQ     skip5
        JSR     large_copy_mem
skip5
        LDX     #LOW(SIZEOF(CCSTR))
        BEQ     skip6
        JSR     small_copy_mem
skip6

;---------------------------------------;
; Copy N_CDATA into N_IDATA             ;
;---------------------------------------;
        LDM     #LOW(SFB(N_CDATA)),?DP0_L00
        LDM     #HIGH(SFB(N_CDATA)),?DP0_L00+1
        LDM     #LOW(SFB(N_IDATA)),?DP1_L00
        LDM     #HIGH(SFB(N_IDATA)),?DP1_L00+1
        LDX     #HIGH(SIZEOF(N_CDATA))
        BEQ     skip7
        JSR     large_copy_mem
skip7
        LDX     #LOW(SIZEOF(N_CDATA))
        BEQ     return

; fall into small_copy_mem to do final copy

;---------------------------------------;
; Copy memory                           ;
;---------------------------------------;
small_copy_mem
        LDY     #0
small_loop
        LDA     (?DP0_L00),Y
        STA     (?DP1_L00),Y
        INY
        DEX
        BNE     small_loop
return  RTS
```

cstartup.s31：321～360 行

```
large_copy_mem
        LDY     #0
large_loop
        LDA     (?DP0_L00),Y
        STA     (?DP1_L00),Y
        INY
        BNE     large_loop
        INC     ?DP0_L00+1              ; update high pointers
        INC     ?DP1_L00+1
        DEX
        BNE     large_loop              ; no, move next block
        RTS


;---------------------------------------------------------------;
; Interrupt vectors must be inserted here by the user.          ;
;                                                               ;
; It is assumed that the interrupt vector segment starts        ;
; at address xxE0 on all chips except 37600 where it is         ;
; starts at xxC0.  The reset vector is assumed to be located    ;
; at xxF?.  We simply skip to xxF? and insert the reset vector. ;
;                                                               ;
; Chip group    Default reset vector                            ;
; ----------------------------------------                      ;
;    -v0      FFFC                                              ;
;    -v1      FFFE                                              ;
;    -v2      FFFA                                              ;
;                                                               ;
; If this does not match your specific chip derivative, you     ;
; have to make changes below.                                   ;
;---------------------------------------------------------------;

        COMMON  INTVEC

?CSTARTUP_INTVEC:
        BLKB    0FFFEH - 0FFDCH -2      ; 3803 Group
#if 0
#if defined(MELPS_37600)
        BLKB    40H - 6      ; FFFA  ( FFC0 + 40 - 6)  (-v2)
```

INTVEC セグメントの設定

INTVEC セグメントをコモン・セグメント宣言しています。
コモン・セグメントは上書きされます。
INTVEC セグメントの先頭アドレスは lnk740.xcl に記述します。

割り込みベクトルからリセット割り込みベクトル分を削除したサイズを設定します。
ここでは以下のように計算しています。

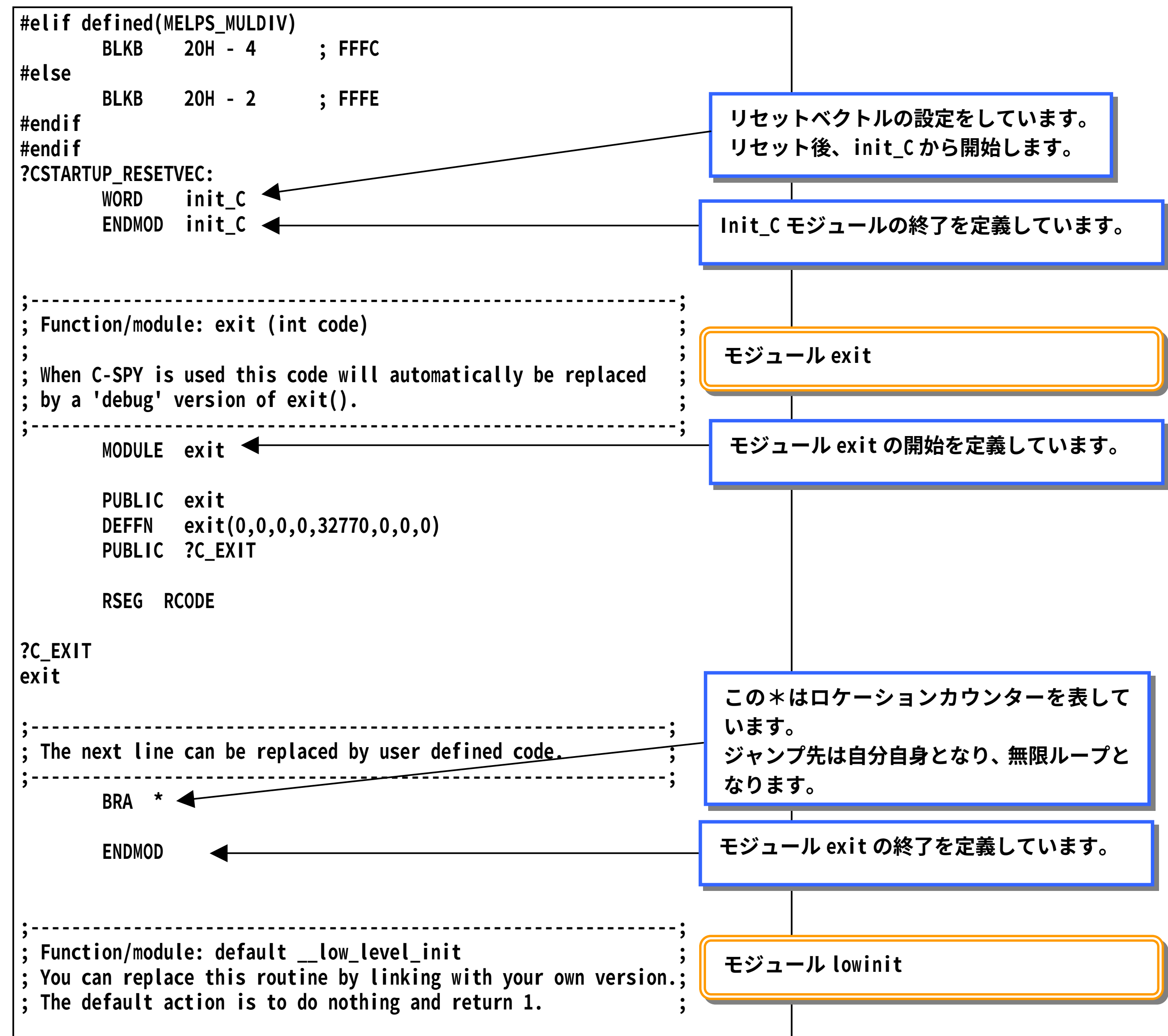
割り込みベクトルの終端アドレス － 割り込みベクトルの先頭アドレス － リセット割り込みベクトルのサイズ

各割り込みベクトルの値は ICC740 の interrupt [ ] 記述で設定します。

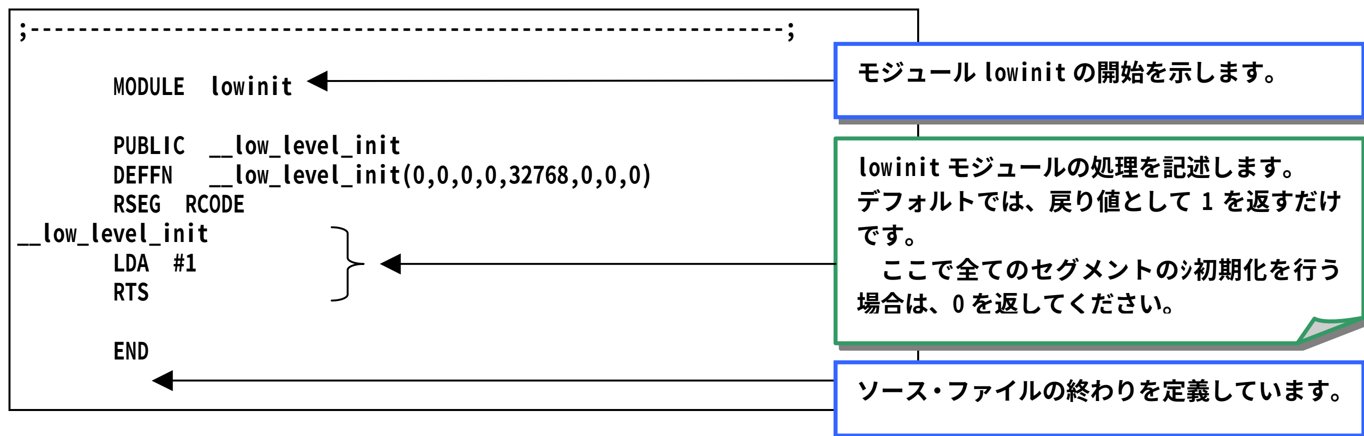
interrupt [16] void intr_timer2(void)

上記では、リンク時に INTVEC セグメントの先頭＋16 番地に intr_timer2 のアドレスが上書きされます。

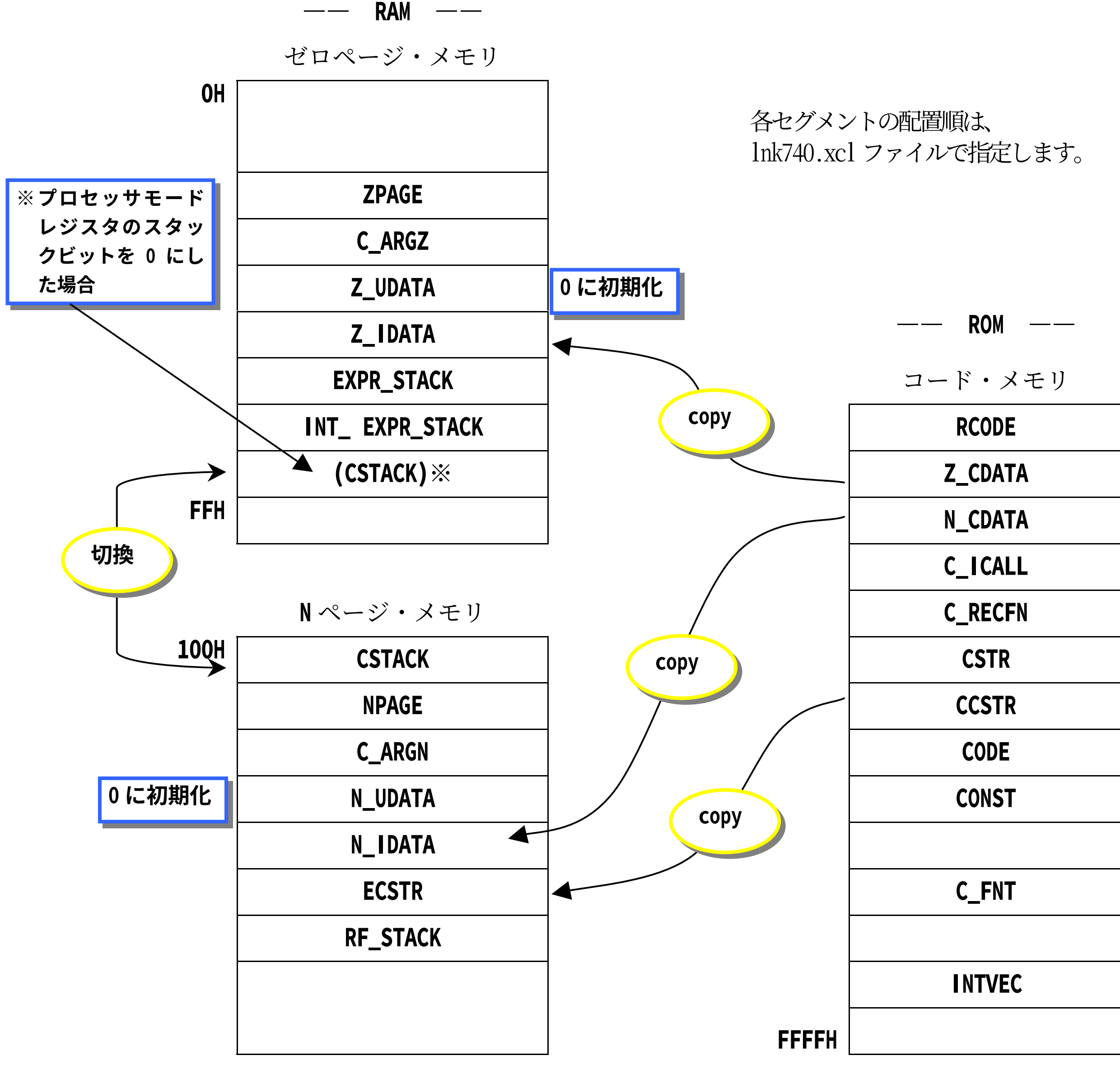
cstartup.s31：361～400 行


```
#elif defined(MELPS_MULDIV)
        BLKB    20H - 4     ; FFFC
#else
        BLKB    20H - 2     ; FFFE
#endif
#endif
?CSTARTUP_RESETVEC:
        WORD    init_C
        ENDMOD  init_C


;---------------------------------------------------------------;
; Function/module: exit (int code)                              ;
;                                                               ;
; When C-SPY is used this code will automatically be replaced   ;
; by a 'debug' version of exit().                               ;
;---------------------------------------------------------------;
        MODULE  exit

        PUBLIC  exit
        DEFFN   exit(0,0,0,0,32770,0,0,0)
        PUBLIC  ?C_EXIT

        RSEG  RCODE

?C_EXIT
exit

;--------------------------------------------------------------;
; The next line can be replaced by user defined code.          ;
;--------------------------------------------------------------;
        BRA   *

        ENDMOD


;---------------------------------------------------------------;
; Function/module: default __low_level_init                     ;
; You can replace this routine by linking with your own version.;
; The default action is to do nothing and return 1.             ;
```

cstartup.s31：401～413 行



Ｃスタートアップ・モジュールの処理を図示すると以下のようになります。



2.7 特殊な領域への値の設定方法

2.7.1 特殊な領域への値の設定方法

IDコード等の特殊な用途に使用する領域への値の設定は、この領域用にlnk740.xclファイル内でセグメントを作成し、値の設定は、アセンブリ言語ファイル内で行ってください。
この領域は、製品によって異なります。各MCUのデータシートでご確認ください。

アセンブリ言語ファイル、lnk740.xclの設定例を以下に示します。

・7542グループ(フラッシュメモリ版)の例

FFD4h	ID1
FFD5h	ID2
FFD6h	ID3
FFD7h	ID4
FFD8h	ID5
FFD9h	ID6
FFDAh	ID7
FFDBh	ROM コードプロテクト

lnk740.xcl ファイル

```
 -Z(CODE)RCODE,.... (この行の次行)
 -Z(CODE)ID_CODE=FFD4-FFDB
```

アセンブリ言語ファイル：

```
    RSEG        ID_CODE
    BYTE        0FFH
    BYTE        0FFH
    BYTE        0FFH
    BYTE        0FFH
    BYTE        0FFH
    BYTE        0FFH
    BYTE        0FFH
ROMCP:
    BYTE        0FFH              ; ROM Code Protect
```

ID コード、ROM コードプロテクトの値を設定してください。

・7545グループ(QzROM版)の例

FFD4h	Renesas 出荷検査用領域
FFD5h	Renesas 出荷検査用領域
FFD6h	Renesas 出荷検査用領域
FFD7h	Renesas 出荷検査用領域
FFD8h	Renesas 出荷検査用領域
FFD9h	Renesas 出荷検査用領域
FFDAh	機能設定 ROM データ
FFDBh	ROM コードプロテクト

lnk740.xcl ファイル

```
    -Z(CODE)RCODE,.... (この行の次行)
    -Z(CODE)RESERVE1,FUNCTION_SET_ROM,RESERVE2=FFD4-FFDB
```

アセンブリ言語ファイル：

```
    RSEG        RESERVE1
    BLKB        01H
    BLKB        01H
    BLKB        01H
    BLKB        01H
    BLKB        01H
    BLKB        01H
    RSEG        FUNCTION_SET_ROM
    BYTE        12H               ; Function Set Rom Data
    RSEG        RESERVE2
    BLKB        01H               ; ROM Code Protect
```

Renesas 出荷検査用領域には、何も書き込まないでください。
機能設定 ROM データの値を設定してください。
ROM コードプロテクトの値は、ROM ライタ上で設定してください。

第　3　章
Cコンパイラ：ICC740

3.1 基本的なオプションの説明

3.2 拡張機能について

この章では、Cコンパイラ：ICC740のオプションや機能について説明します。

3.1　基本的なオプションの説明

3.1.1 コンパイラ・オプションの要約

下表に、全コンパイラ・オプションの要約を示します。

オプション	内容
-A*prefix*	接頭語付きファイル名でアセンブラソースを生成。
-a *filename*	ファイル名で指定したアセンブラソースを生成。
-b	オブジェクトをライブラリ・モジュールにする。
-C	ネスト・コメントを使用可能にする。
-c	char 型を singed char にする。
-D*symb[xx]*	シンボルを定義する。
-e	ターゲット依存拡張子を使用可能にする。
-F	新しいページに各関数をリストする。
-f *filename*	ファイルからコマンドラインオプションを読み取る。
-G	標準入力をソースとして開く。
-g[O][A]	グローバル型チェックを可能にする。
-h	多重割り込み時の INT_EXPR_STACK セグメントの管理を行う。
-H*name*	オブジェクト・モジュール名を設定する。
-I*prefix*	#include 探索接頭語を追加する。
-i	#include ファイルをリストする。
-K	C++コメントを使用可能にする。
-L[*prefix*]	接頭語つきソース名のリストを生成する。
-l *filename*	ファイル名指定のリストを生成する。
-m[tl]	メモリ・モデルを選択する。
-N*prefix*	接頭語付きのプリプロセッサ出力ファイル。
-n *filename*	プリプロセッサ出力ファイル名を指定。
-O*prefix*	オブジェクト・ファイル名の接頭語を設定する。
-o *filename*	オブジェクト・ファイル名を設定する。
-P	PROM 化可能なコードを生成する。
-p*lines*	リストをページに書式化する。
-q	リストにニモニックを入れる。
-R*name*	コード・セグメント名を設定する。
-r[012inre]	デバッグ情報を生成する。
-S	コンパイラのサイレント動作を設定する。
-s[0-9]	実行速度に対する最適化
-T	使用可能な行だけをリストする。
-t*n*	タブ間隔を設定する。
-U*symb*	シンボルを未定義にする。
-v*n*	プロセッサ・グループを選択する。
-w	警告メッセージを表示しないようにする。
-X	C 宣言を記述する。
-x[D][F][T][2]	相互参照リストを生成する。
-y	文字列を変数として初期化する。
-z[0-9]	コードサイズによる最適化。

3.2 拡張機能について

740マイクロプロセッサに固有な機能をサポートするために740Cコンパイラで提供されている拡張機能を説明します。

概要

拡張機能は、次の3つの方法で提供されます。

- ◆ 拡張キーワードとして：省略時、740CコンパイラはANSI仕様に準拠し、740拡張機能を使用することはできません。コマンド行オプション -e を使用すると、拡張キーワードが利用可能になります。よって、キーワードは変数名として使用されないように予約されます。
- ◆ #pragma キーワードとして：これらのキーワードは、コンパイラのメモリ割当方法、コンパイラで拡張キーワードが使用可能かどうか、あるいはコンパイラが警告メッセージを出力するかどうかを制御する#pragma 疑似命令を提供します。
- ◆ 組込み関数として：これらの関数は、非常に低レベルのプロセッサの詳細に至るまで直接的なアクセスを提供しています。

3.2.1 拡張キーワードの要約

拡張キーワードは、次の機能を提供します。

アドレス指定の制御

変数は、zpageでゼロ・ページ領域内に、あるいはnpageでゼロ・ページの外に強制的に指定することができます。

また、bitによって単一ビット・ゼロ・ページ変数として宣言することもできます。

zpage　npage　bit

不揮発性 RAM

変数は、次のデータ型修飾子を使用して不揮発性RAMに置くことができます。

no_init

I/O アクセス

バイトI/O識別子を宣言するには、sfrデータ型を使用することができます。

sfr

割込みルーチン

次のキーワードを使用すると、C言語で割り込み処理ルーチンや非割込みルーチンを記述することができます。

interrupt monitor

呼出し手順

関数は、次のキーワードを使用して変更された呼出しシーケンスをもつことができます。

tiny-func

3.2.2　#PRAGMA疑似命令の要約

#pragam 疑似命令は、標準言語構文内で拡張機能の制御を提供します。
#pragam 疑似命令は、-e オプションと無関係に利用することができます。
次の種類の#pragma 関数が利用できます。

ビットフィールド・オリエンテーション

```
#pragma bitfields=default
#pragma bitfields=reversed
```

拡張機能の制御

```
#pragma language=default
#pragma language=extended
```

関数属性

```
#pragma function=default
#pragma function=interrupt
#pragma function=intrinsic
#pragma function=monitor
#pragma function=tiny_func
```

メモリ使用

```
#pragma codeseg
#pragma memory=constseg
#pragma memory=dataseg
#pragma memory=default
#pragma memory=no_init
#pragma memory=zpage
#pragma memory=npage
```

警告メッセージ制御

```
#pragma warnings=default
#pragma warnings=off
#pragma warnings=on
```

3.2.3　定義済みシンボルの要約

定義済みシンボルを使用すると、コンパイル時の環境を検査することができます。

関数	内容
DATE	Mmm dd yyyy 形式の現在の日付
FILE	現在のソース・ファイル名
_IAR_SYSTEMS_ICC_	IAR C コンパイラ識別子
LINE	現在のソース行番号
STDC	IAR C コンパイラ識別子
TID	目的の識別子
TIME	hh:mm:ss 形式の現在の時間
VER	int としてバージョン番号を返却する。

3.2.4　その他の拡張機能

＄文字

DEC/VMS C との互換性を確保するために、識別子の有効文字のセットに＄文字を追加しています。

コンパイル時の SIZEOF の使用

ANSI 規定では、#if 式や#elif 式は sizeof 演算子を使用できないという制限がありますが、これは撤廃しています。

第　４　章

アセンブラ：A740

4.1 基本的なオプションの説明

4.2 アセンブリ言語インターフェース

この章では、アセンブラ：A740のオプション等について説明します。

4.1　基本的なオプションの説明

4.1.1 アセンブラ・オプションの概要

下表に、全アセンブラ・オプションの要約を示します。

オプション	内容
-B	マクロ実行情報を出力する。
-b	オブジェクトをライブラリ・モジュールにする。
-c{DMEAO}	リスト・オプションを設定する。
-D*symb*[=*xx*]	オプション値でシンボルを定義する。
-d	#ifdef / #endif がそろっているかをチェックしない。
-E*number*	エラーの最大数を設定する。
-f *filename*	コマンド行を拡張する。
-G	標準入力をソースとして開く。
-I*prefix*	インクルード検索接頭語を追加する。
-i	インクルード・ファイルをリストする。
-L[*prefix*]	前置ソース名にリストを生成する。
-l *filename*	命名ファイルにリストを生成する。
-Mab	マクロ引数引用文字を設定する。
-N	リストにヘッダーをつけません。
-O*prefix*	オブジェクト・ファイル名接頭語を設定する。
-o *filename*	オブジェクト・ファイル名を設定する。
-p*nn*	各ページの行数を設定する。
-r[en]	オブジェクトでデバッガ出力を使用可能にする。
-S	アセンブラのサイレント演算を設定する。
-s{+|-}	ユーザー・シンボルの大文字／小文字の区別を設定する。
-T	アクティブ行だけをリストする。
-t*n*	タブ・スペーシングを設定する。
-v*n*	プロセッサ・コンフィギュレーション
-U*symb*	シンボルを未定義にする。
-uN	16 ビットアドレッシングを設定する。
-w[*string*]	警告を不能にする。
-x{DI2}	相互参照リストを生成する。

4.2 アセンブリ言語インターフェース

C 言語関数からアセンブリ言語サブルーチンをコールする場合、またはアセンブリ言語サブルーチンから C 言語関数をコールする場合のインターフェースを説明します。

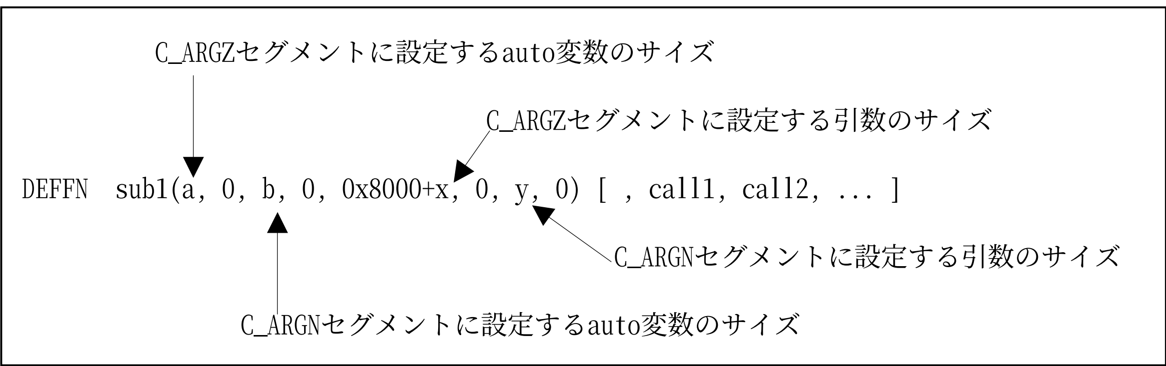
4.2.1 関数宣言

C 言語、アセンブリ言語のどちら側から呼び出す場合も、アセンブリ言語側で呼び出される関数の関数宣言を記述する必要があります。

関数宣言は、アセンブリ言語ソースファイルに DEFFN アセンブラ疑似命令を記述します。DEFFN は、C_ARGZ，C_ARGN セグメントのサイズ計算に必要となります。

1) C 言語からアセンブリ言語のサブルーチンをコールする場合

DEFFN アセンブラ疑似命令には、auto 変数と引数のサイズを設定します。
また、関数呼び出しがある場合は、call1，... に呼び出し関数を記述してください。

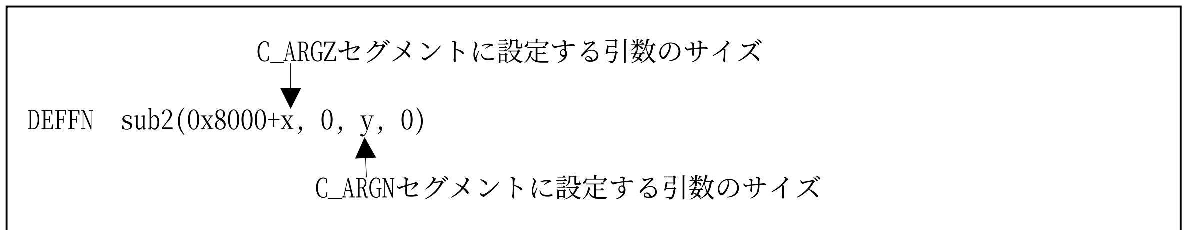


なお、割り込み処理で使用する関数の場合は DEFFN の最初のパラメータを以下のように設定してください。

```
DEFFN   sub1(0x200+a, 0, b, 0, 0x8000+x, 0, y, 0) [ , call1, call2, ... ]
```

2) アセンブリ言語から C 言語の関数をコールする場合

DEFFN アセンブラ疑似命令には、引数のサイズを設定をしてください。



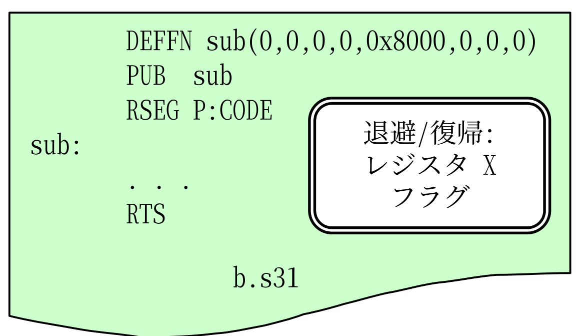
4.2.2　C 言語からアセンブリ言語サブルーチンをコール

1) C 言語から引数無し、戻り値無しのアセンブリ言語サブルーチンを呼び出す場合

C 言語から引数無し、戻り値無しのアセンブリ言語サブルーチンを呼び出す場合の記述例を示します。

```
extern void sub(void);
func5(void)
{
        sub();
}
```

a.c

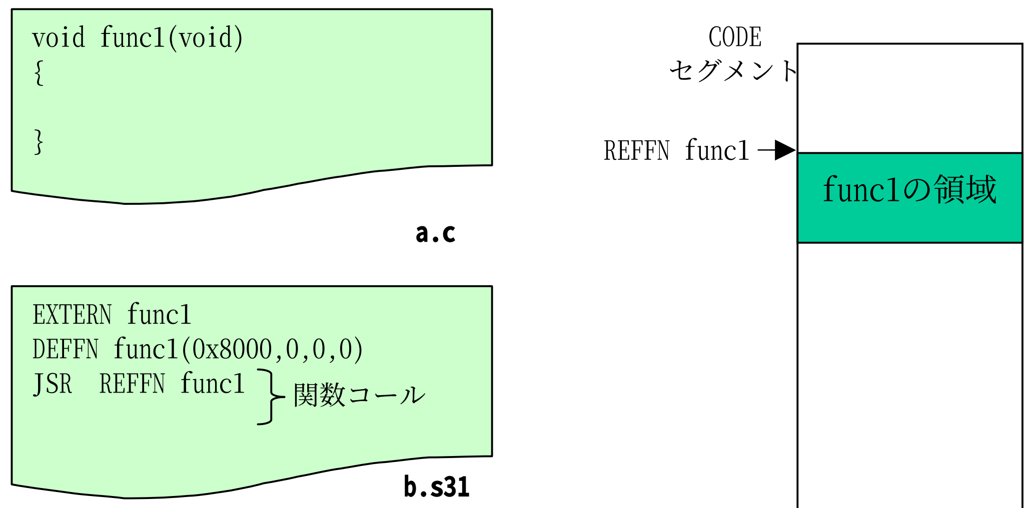


C 言語ソース内では、アセンブリ言語サブルーチンを extern 宣言してコールしてください。
コールされたアセンブリ言語サブルーチンでは、インデクスレジスタ X とプロセッサステータスレジスタの退避および復帰を行ってください。

4.2.3　アセンブリ言語から C 言語関数をコール

1) アセンブリ言語から引数無し、戻り値無しの C 言語関数を呼び出す場合

アセンブリ言語から引数無し、戻り値無しの C 言語関数を呼び出す場合の記述例を示します。

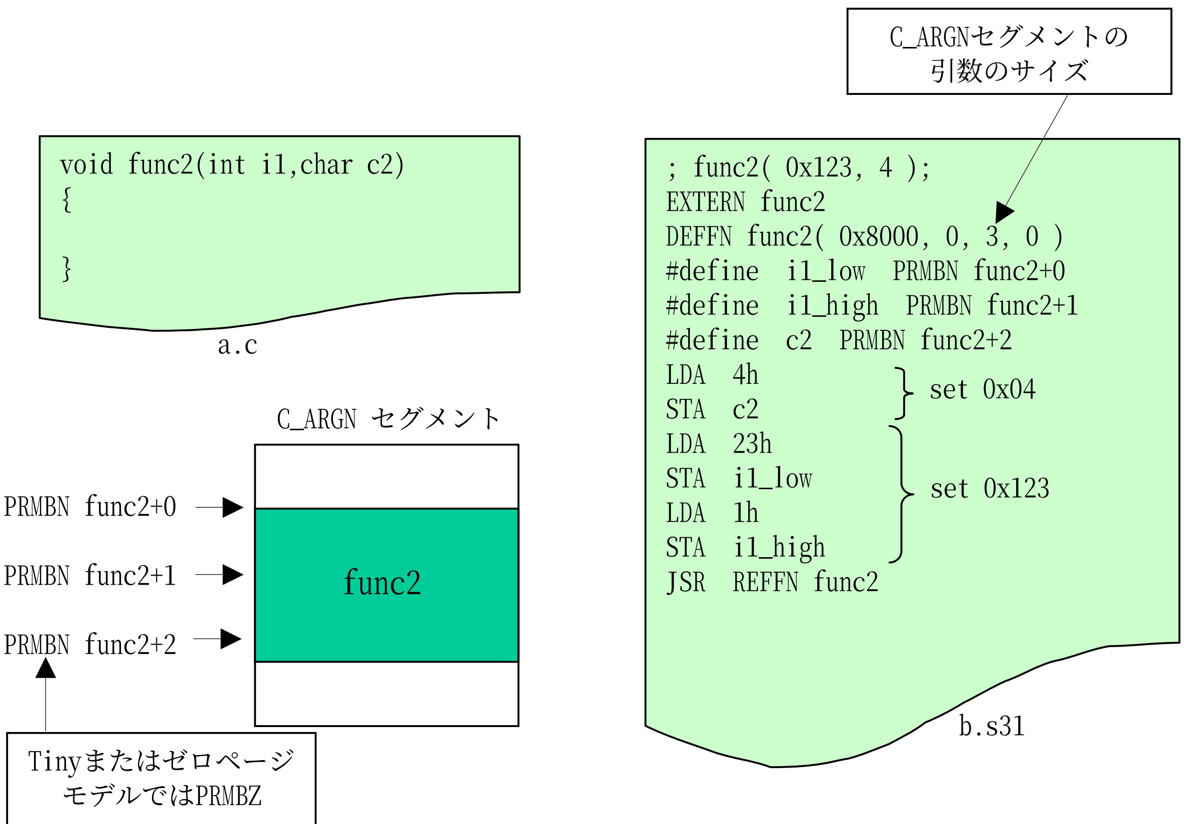


アセンブリ言語サブルーチンで C 言語関数をコールする場合は、REFFN を付けてコールしてください。
C 言語関数では C ランタイム関数コール等に EXPR_STACK セグメント（割り込み発生時は INT_EXPR_STACK セグメント）を使用します。
このセグメントはインデックスレジスタ X で操作します。
インデックスレジスタ X が EXPR_STACK セグメントを示していない場合は、関数コールの前にインデックスレジスタ X を設定してください。

例）　LDX　#LOW(SFE(EXPR_STACK))

2）アセンブリ言語から引数を持つ C 言語関数を呼び出す場合（Large モデル）

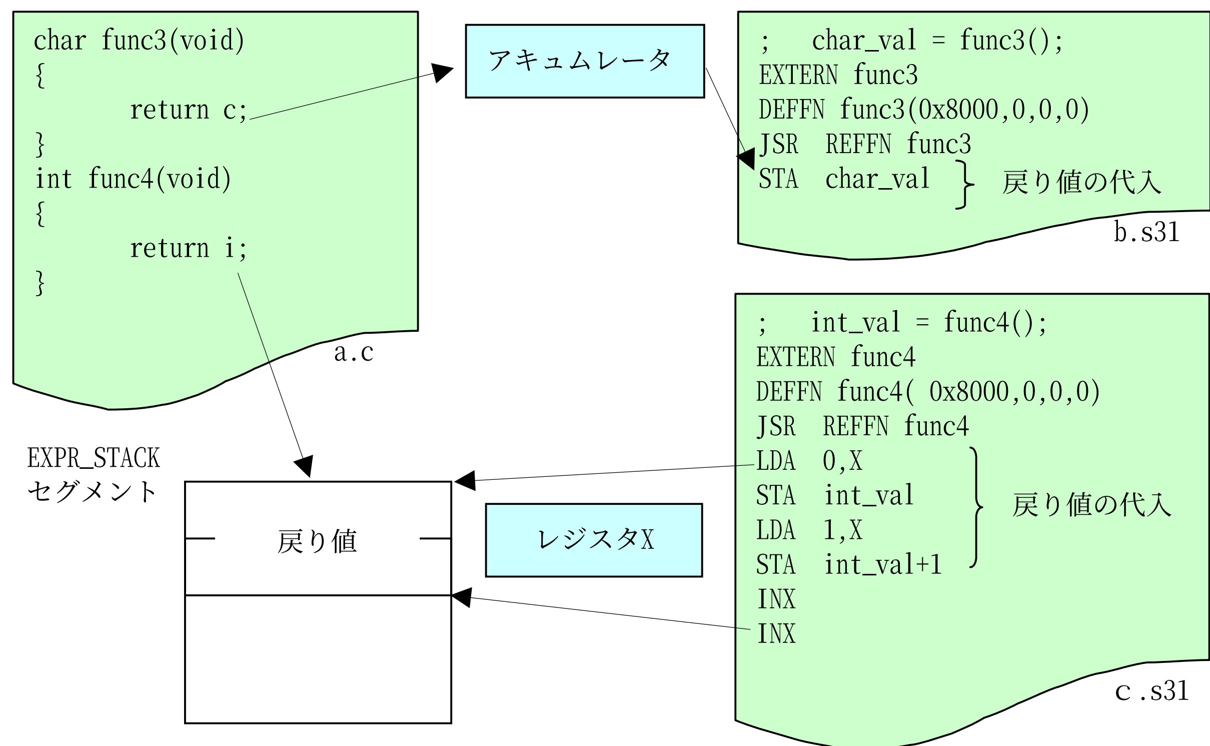
アセンブリ言語から引数を持つ C 言語関数を呼び出す場合の記述例を示します。



アセンブリ言語サブルーチンで C 言語関数の引数を設定する場合は、PRMBN を使用します。
"PRMBN 関数名 + オフセット"で引数の位置を示します。

3) アセンブリ言語から戻り値を持つ C 言語関数を呼び出す場合

アセンブリ言語から戻り値を持つ C 言語関数を呼び出す場合の記述例を示します。



C 言語関数からの戻り値は、型が char 型など 1 バイトデータの場合は、アキュムレータに設定されます。その他は EXPR_STACK セグメントに格納されます。
アセンブリ言語サブルーチンで C 言語関数を呼び出した後は、戻り値の型に応じてアセンブリ言語データへの代入を行ってください。
リターン値が EXPR_STACK セグメントに格納される場合は、インデックスレジスタ X が戻り値の位置を示していますので、インデックスレジスタ X を使用して代入を行ってください。代入後、リターン値のサイズ分、インデックスレジスタ X を加算してください。

第　5　章
リンカ：XLINK

5.1　基本的なオプションの説明

5.2　オプションファイルの説明

この章では、リンカ：XLINKのオプション等について説明します。

5.1　基本的なオプションの説明

5.1.1　オプションの概要

下表に、740 ファミリで使用する XLINK オプションの概要を示します。

オプション	内容
-! ｺﾒﾝﾄ -!	コメント・デリミタ。
-C *ﾌｧｲﾙ*,...	ライブラリとしてロードする。 補足：cstartup.s31を使用の場合は-Cを指定してください。 (理由：ライブラリの中にcstartupモジュールがあり、-Cを使用しないとcstartup.s31と二重モジュールのエラーとなります。)
-c*cpu*	プロセッサ・タイプを指定する。
-D *ｼﾝﾎﾞﾙ=値*	シンボルを定義する。
-d	コードの生成を不能にする。
-e *新=旧*[,*旧*]...	外部シンボルをリネームする。
-F *ﾌｫｰﾏｯﾄ*	出力形式を選択する。
-f *ﾌｧｲﾙ*	XCL ファイル名。
-G	グローバル型チェックを不能にする。
-H *ﾍｷｻ値*	未使用コードメモリを充填する。
-I *ﾊﾟｽ*	インクルード・パス。
-J *ｻｲｽﾞ*,*方法*[,*補数*]	チェックサムを生成する。
-L *ﾃﾞｨﾚｸﾄﾘ*	リスト・ファイルのディレクトリを指定する。
-l *ﾌｧｲﾙ*	リスト・ファイル名を指定する。
-o *ﾌｧｲﾙ*	出力ファイル名。
-p *行数*	ページ毎の行数を設定する。
-R[w]	アドレス範囲のチェック。
-S	サイレント操作を選択する。
-w[*番号*|s|t]	警告メッセージを不能にする。
-x[e][h][i][m][n][s][o]	相互参照リストを生成する。 (注意：xlink.pdf 参照)
-Y[*文字列*]	出力形式の可変部を選択する。
-y[*文字列*]	出力形式の可変部を選択する。
-Z[@] *ｾｸﾞﾒﾝﾄ*	セグメントを定義する。 (注意：xlink.pdf 参照)
-z	セグメント・オーバーラップのエラーを警告にする。

5.2　オプションファイルの説明

5.2.1　リンク・コマンド・ファイルの解説

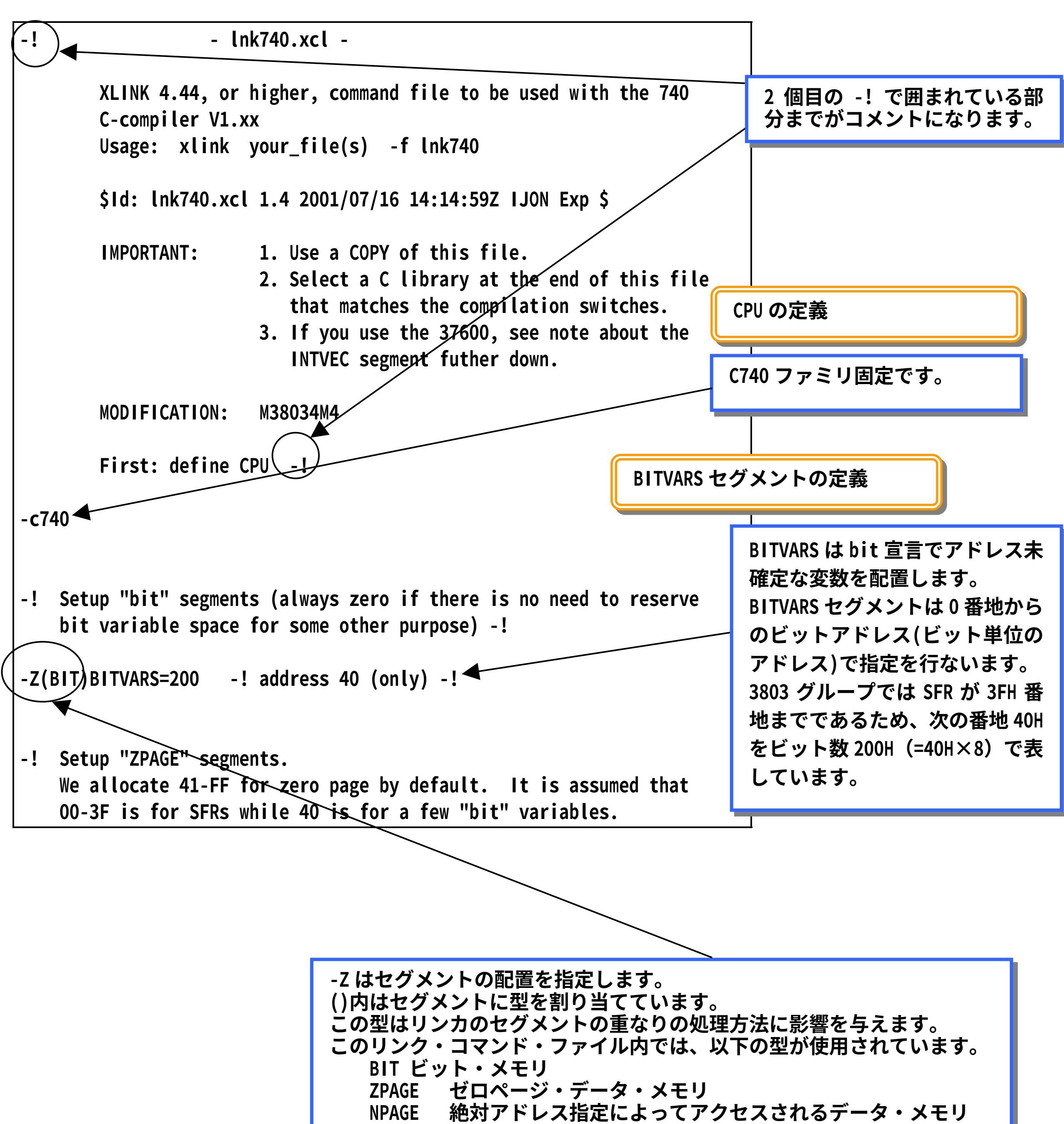
リンク・コマンド・ファイルはテンプレートが用意されていますが、ターゲットシステムに適応させるために、いくつかの変更が必要となってきます。
ここでは、M3T-ICC740 で用意されているリンク・コマンド・ファイル(lnk740.xcl)に解説を加えながら、必要に応じて変更する個所や方法を説明していきます。

リンク・コマンド・ファイルは以下の場合に変更が必要となります。
1.　デフォルトのマイコンと異なる、割り込みベクトル、メモリ配置を持つマイコンを使用します。
2.　C スタック領域を 1 ページ・メモリからゼロページ・メモリへ切り替えます。
3.　各セグメントの配置を変更します。

なお、リンク・コマンド・ファイル内では数値を 16 進数で処理しています。

lnk740.xcl ファイルの解説

lnk740.xcl：1～30 行



lnk740.xcl：31～60 行

```
    The following segment defintions (EXPR_STACK, INT_EXPR_STACK and
    CSTACK) that do not have an address given must fit inside the
    "41-FF" address range.
    If you have the CSTACK (processor stack) segment outside zero page,
    you have to give it an address and XLINK will no longer try to
    fit it within zero page.              -!

-Z(ZPAGE)ZPAGE,C_ARGZ,Z_UDATA,Z_IDATA=41-FF


-!  Setup "EXPR_STACK" segment. This zero page located stack is used
    to hold temporary when evaluating complex expressions.
    It is set to 20(hex) below. -!

-Z(ZPAGE)EXPR_STACK+20


-!  Setup "INT_EXPR_STACK" segment. This zero page located stack is used
    to hold temporary when evaluating complex expressions for interrupt
    routines written in C.  It is set to 0 below.
    You must give this stack space if you have C written interrupts that
    need an expression stack.  -!

-Z(ZPAGE)INT_EXPR_STACK+0


-!  Setup "CSTACK" segment. This is the CPU stack. Note that this can
    either reside in page 0 or 1  -!

-Z(NPAGE)CSTACK+40=100
```

ゼロページ RAM の設定

セグメント名 ：YY（－ZZ)：セグメントが指定した範囲の先頭(YYH 番地) から始まるようにセグメントを割り付けます。
範囲指定が必要ない場合は、“＝ 41 ”のように、先頭アドレスのみ記述します。

カンマで区切られたセグメントは、記述順でメモリに割り付けられます。

式スタックセグメントの設定

“＋20 ”としてサイズを 20H と指定しています。必要に応じて変更します。

セグメント名 ＋YY　：セグメントが指定したメモリ容量(YYH バイト)となるようセグメントを割り付けます。

割り込み式スタックセグメントの設定

“＋0 ”としてサイズを 0 と指定しています。必要に応じて変更します。

C スタックセグメントの設定

C スタック領域を 100H から 13FH まで指定しています。
C スタック領域をゼロページに配置する場合は以下のように変更してください。
　　-Z(ZPAGE)+40
　　INT_EXPR_STACK の後に 40H バイト配置されます。
　　※cstartup.s31 の変更も必要です。

※注意
以下のセグメントはゼロページに配置されなければなりません。
　　BITVARS, ZPAGE, C_ARGZ, Z_UDATA, Z_IDATA
　　EXPR_STACK, INT_EXPR_STACK

lnk740.xcl：61～90 行

```
-!  Setup "NPAGE" segments at address 1000-7FFF   -!

-Z(NPAGE)NPAGE,C_ARGN,N_UDATA,N_IDATA,ECSTR=100-43F

-! Setup "RF_STACK" segment.  This stack is used for recursive function.
   It is by default given a size of 256 bytes by the library.  By giving
   a non-zero size below, you _expand_ the stack by that amount.  -!

-Z(NPAGE)RF_STACK+0


-!  Setup all read-only segments (PROM) at address 8000  -!

-Z(CODE)RCODE,Z_CDATA,N_CDATA,C_ICALL,C_RECFN,CSTR,CCSTR,CODE,CONST=C080-FEFF

-!  Setup the "INTVEC" interrupt segment.
    If you are using the 37600 (chip group -v2) and the default cstartup
    reset vector, you must change the INTVEC line below to:
        -Z(CODE)INTVEC=FFC0-FFFF
    If you have a tiny chip derivative that does not have the interrupt
    vectors in page FF, you can change the page of the addresses below.
    CSTARTUP inserts the reset vector relative to INTVEC start which
    means that you can change the page without any problems:
        -Z(CODE)INTVEC=1FE0-1FFF
        -Z(CODE)C_FNT=1F00                      -!
```

ゼロページ以外の RAM の設定

N ページ・メモリに配置するセグメントを指定します。
100H 番地から指定していますが、CSTACK が 13FH まで使用するため、NPAGE は 140H となります。
2 バイトは RF_STACK のアドレスを保持します。

再帰的関数を使用しない場合は+0 としてください。
再帰的関数使用時に、0 以外のサイズを指定することによって、256+2 のサイズが与えられます。2 バイトは RF_STACK の管理用アドレスを保持します。

ROM の設定

ROM に配置するセグメントとアドレスを設定します。
アドレスには予約領域を持つマイコンでは、予約領域を外してください。
また、割り込みベクトル領域も外してください。

割り込みベクトル INTVEC の設定（次ページ）

割り込みベクトルの先頭からリセットベクトルの終了までを指定します。
記述例は 3803 グループです。マイコンに応じて変更してください。

スペシャルページ C_FNT の設定（次ページ）

拡張記述 tiny_func を使用した関数の領域を設定します。
CONST 等が FF00H を越える場合は、その後に続きます。

lnk740.xcl：91～122 行

```
-Z(CODE)INTVEC=FFDC-FFFD
-Z(CODE)C_FNT=FF00-FFDB


-!  See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write


-!  See configuration section concerning scanf/sscanf -!
-e_medium_read=_formatted_read


-! This example files selects the default library which is
   tiny memory model and a 740 with MUL/DIV.
   This corresponds to option -mt and -v0 to the compiler.
   If you want to use another library, you can do it by
   removing the comments around it and adding comments around
   the default library.                  -!

-C cl7400l

-! -C cl7400t -!                -!   -v0  -mt  -!
-! -C cl7400l -!                -!   -v0  -ml  -!
-! -C cl7401t -!                -!   -v1  -mt  -!
-! -C cl7401l -!                -!   -v1  -ml  -!
-! -C cl7402t -!                -!   -v2  -mt  -!
-! -C cl7402l -!                -!   -v2  -ml  -!


-!  Code will now reside on file aout.a31 in INTEL-STANDARD format -!
```

printf，scanf 等のフォーマット変更

printf 等フォーマットを使用するライブラリ関数はサイズが大きいため、表示する変数の型を変更することにより小さくできます。
printf 等を使用しない場合は、変更の必要がありません。

-e A=B：既存の外部シンボル名 B を新しい名前 A にリネームします。

_medium_write	浮動小数点数値を未サポート
_small_write	%%,%d,%o,%c,%s,%x のみサポート
_medium_read	浮動小数点数値を未サポート

lnk740.xcl ファイルでのセグメント配置は以下のようになります。

ゼロページ RAM
（Z ページ・メモリ）

アドレス	セグメント	サイズ
0x00		
0x40	BITBARS	(0x1)
0x41	ZPAGE	(可変)
	C_ARGZ	(可変)
	Z_UDATA	(可変)
	Z_IDATA	(可変)
	EXPR_STACK	(0x20)
	INT_ EXPR_STACK	(0x20)
0xFF		

ゼロページ以外の RAM
（N ページ・メモリ）

アドレス	セグメント	サイズ
0x100	CSTACK	(0x40)
0x140	NPAGE	(可変)
	C_ARGN	(可変)
	N_UDATA	(可変)
	N_IDATA	(可変)
	ECSTR	(可変)
	RF_STACK	(0x0)
0x43F		

ROM
（コード・メモリ）

アドレス	セグメント	サイズ
0xC080	RCODE	(可変)
	Z_CDATA	(可変)
	N_CDATA	(可変)
	C_ICALL	(可変)
	C_RECFN	(可変)
	CSTR	(可変)
	CCSTR	(可変)
	CODE	(可変)
	CONST	(可変)
0xFFxx	C_FNT	(可変)
0xFFDC – 0xFFFD	INTVEC	(0x22)
0xFFFF		

可変部分はリンカ XLINK により設定されます。
C_FNT は 0xFF00 から配置指定ですが、CONST セグメントが 0xFF00 を越えた場合、その後に配置されます。

第　6　章
デバッガ

6.1 デバッガの起動

6.2 シミュレータのセットアップ

6.3 シミュレータ用ＭＣＵファイルの作成

この章では、High-performance Embedded Workshop(HEW) の主に「デバッガ」としての機能について説明しています。

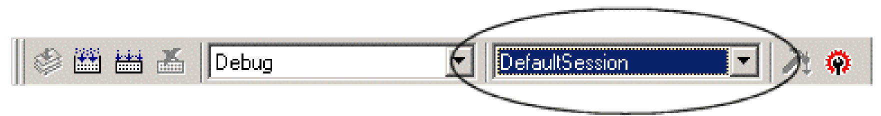
6.1 デバッガの起動

ここでは、デバッガの中で740シミュレータの接続及び終了方法について説明します。
シミュレータ に接続することで、デバッグを開始することができます。

6.1.1 シミュレータの接続

シミュレータ を使用する設定があらかじめ登録されているセッションファイルに切り替えることにより、シミュレータを簡単に接続できます。

下記ツールバーのドロップダウンリストから、"Session740_Simulator"を選択してください。



選択すると、デバッガのセットアップを行うためのダイアログが表示されます。
セットアップ方法は6.2,6.3の項目を参照してください。
このセットアップが終了すると、接続は完了です。

6.1.2 シミュレータの終了

以下の方法があります。

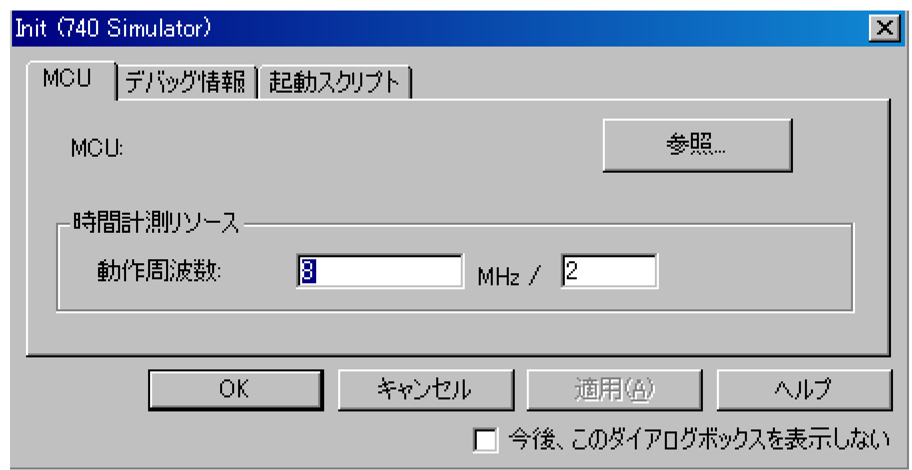
1. セッションを"DefaultSession" に切り替える
   シミュレータ 接続時に使用したドロップダウンリストで、"DefaultSession" を選択してください。
2. HEW自体を終了する
   [ファイル->アプリケーションの終了]を選択してください。HEWは終了します。

セッション切り替え、および、HEW 終了前には、セッション保存確認のメッセージボックスが表示されます。 セッション保存が必要な場合は、[はい]ボタンをクリックしてください。不要なら、[いいえ]ボタンをクリックしてください。

6.2 シミュレータのセットアップ

6.2.1 740のInitダイアログ

Init ダイアログは、シミュレータ起動時に設定が必要な項目を設定するためのダイアログです。
このダイアログで設定した内容は、次回起動時も有効となります。
Init ダイアログでは MCU ファイルを指定してください。「参照...」ボタンで MCU ファイルを選択することができます。
ご使用のマイコンの MCU ファイルが見つからない場合は、MCU ファイルを作成してください。



なお、Init ダイアログは、以下のいずれかの方法で再表示できます。

- シミュレータ起動後、メニュー[基本設定]→[シミュレータ]→[システム...]を選択する。
- Ctrl キーを押しながらデバッグセッションに切り替える。

6.3 シミュレータ用MCUファイルの作成

ご使用のマイコンの MCU ファイルが見つからない場合、MCU ファイルを作成します。
MCU ファイルには、以下の内容を順番に記述します。 ファイル名は、MCU 名を指定してください("m3xxxx.mcu")。拡張子は、"mcu"と指定してください。 各アドレスは 16 進数で記述してください。また、基数を示すプレフィックスは付けないでください。
3〜6 の情報は、ご使用になられる MCU のデータブック等をご参照の上、記述してください。

1. MCU 名
2. 予約番号
3. CPU モードレジスタアドレスとスタックページ選択ビット番号
4. リセットベクトルアドレス
5. BRK ベクトルアドレス
6. 割り込みベクトル情報

- MCU 名、予約番号
  先頭には、必ず';' (セミコロン)を付けて下さい。

- CPU モードレジスタのアドレスとスタックページ選択ビット番号
  CPU モードレジスタのアドレスとスタックページ選択ビット番号は':' (コロン)で区切って下さい。

- リセットベクトルアドレス
  アドレスの後ろに":RST"を付けて下さい。

- BRK ベクトルアドレス
  アドレスの後ろに":BRK"を付けて下さい。

- 割り込みベクトル情報
  割り込みベクトルアドレス、対応する割り込み制御レジスタアドレス、割り込み制御ビット番号の情報を記述します。
  各情報は、':' (コロン)で区切って下さい。割り込みベクトル情報は最大 32 点まで記述可能です。

記述例

以下に例(**m38000.mcu**)を示します。

```
;M38000
;1
3B:2
FFFC:RST
FFDC:BRK
FFFA:3E:0
FFF8:3E:1
FFF6:3E:2
FFF4:3E:3
FFF2:3E:4
FFF0:3E:5
FFEE:3E:6
FFEC:3E:7
FFEA:3F:0
FFE8:3F:1
FFE6:3F:2
FFE4:3F:3
FFE2:3F:4
FFE0:3F:5
```

第　7　章

コーディングのコツ

C言語でコーディングする時にちょっと気を付けておくことで、コード効率の良いコーディングとなる場合がありますので、この章では、コーディングのコツについて説明します。

1)　最適化オプションを使用する

　コーディングのコツではありませんが、最適化オプションを指定することで簡単にコードサイズまたは実行スピードの改善を実現することが出来ます。

　最適化オプションには、コードサイズを小さくすることを目的とした最適化と、実行スピードを速くすることを目的とした最適化の２種類があります。

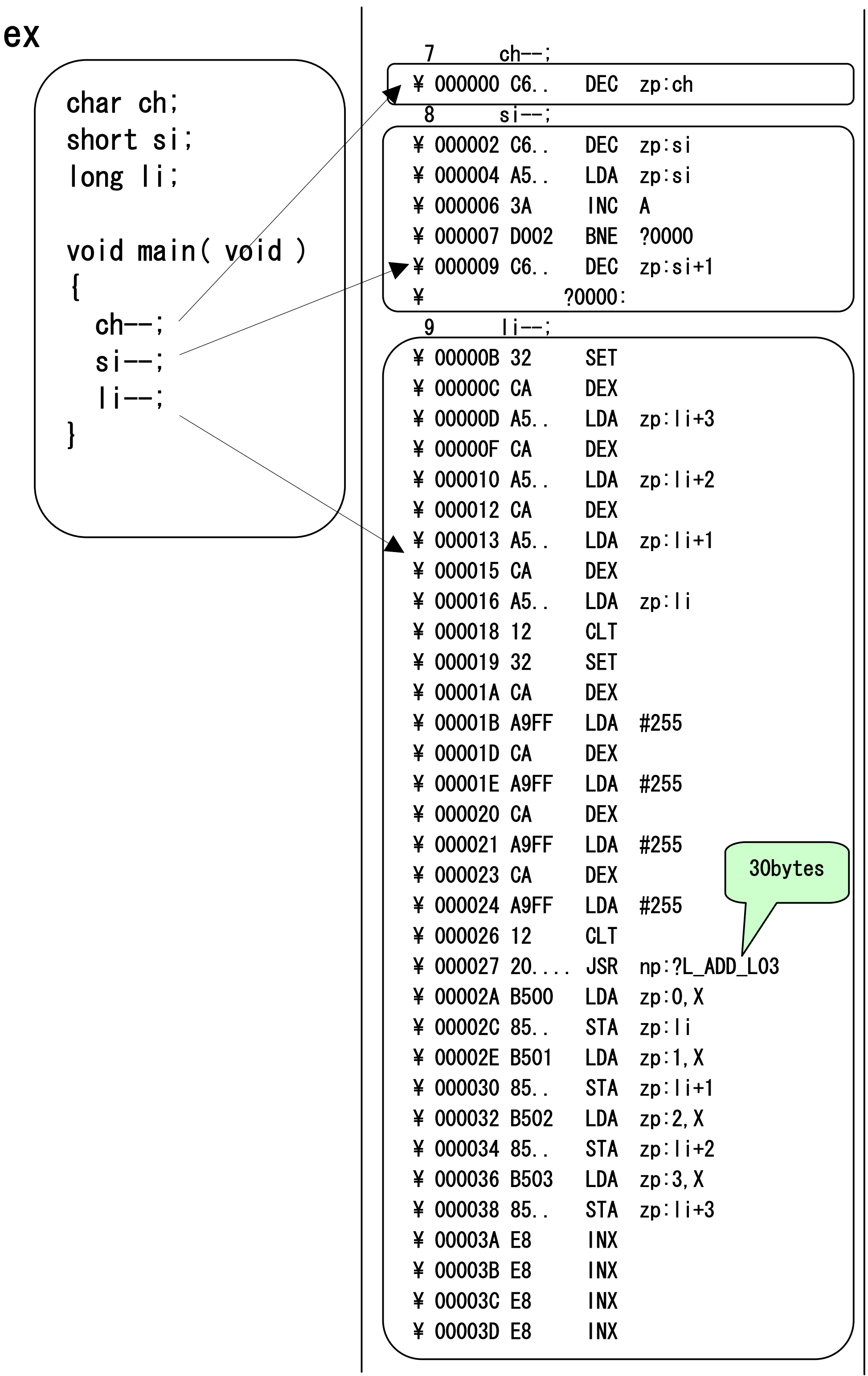
```
-z1 ～ -z9 : サイズ優先最適化

-s1 ～ -s9 : スピード優先最適化
```

値	レベル
0	最適化なし
1–3	完全にデバッグ可能
4–6	一部の構成体がデバッグ不可.
7–9	完全な最適化.

-s オプションは、-z オプションと同時には使用できません。

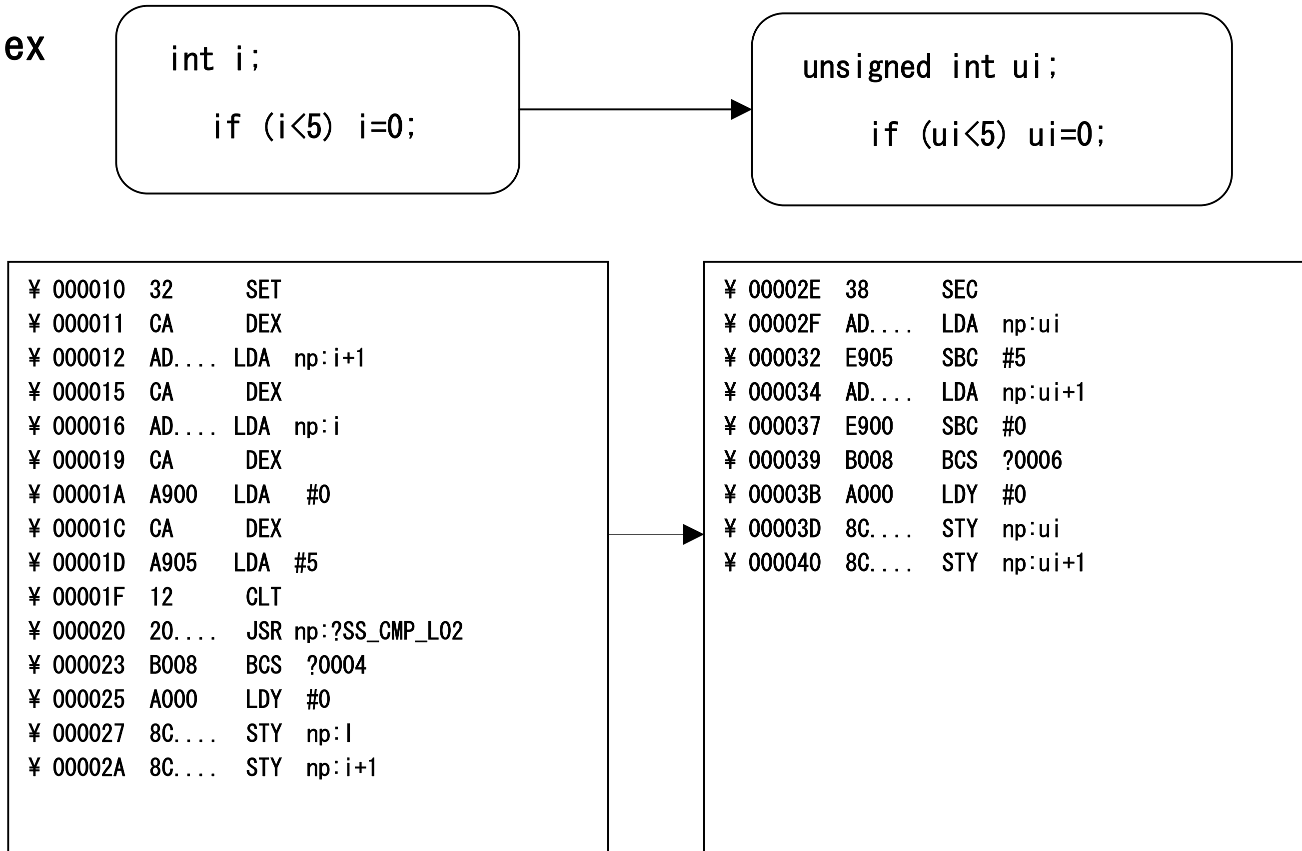
2) できるだけ小さい整数型を使用する

変数を定義する場合は、使用目的に合わせてできるだけ小さい整数型を使うようにしましょう。コードサイズおよび実行スピードの両方に効果的なコードが出力されます。



3)　unsigned int 型 変数を使用する

740 ファミリは、signed int 型変数を使うより unsigned int 型変数を使う方がＲＯＭ効率がよくなります。特に、型の変換、比較、配列の索引付け、シフト、除算などをおこなう場合にＲＯＭ効率がよくなります。



4)　ビット処理には、ビットフィールドを使用する

ビットの判定やon/offを行う時にANDやORを使うより、ビットフィールドを使った方が、効率の良いコードとなります。

ex

```
typedef union {
  unsigned char byte;
  struct {
    unsigned char b0:1;
    unsigned char b1:1;
    unsigned char b2:1;
    unsigned char b3:1;
    unsigned char b4:1;
    unsigned char b5:1;
    unsigned char b6:1;
    unsigned char b7:1;
  } bitf;
} BYTE_BIT;
unsigned char uc;
BYTE_BIT data;
if ( (uc & 0x04) == 0 ) {
   uc |= 0x04;
}
if ( data.bitf.b2 == 0 ) {
   data.bitf.b2 = 1;
}
```

```
000063  A904    LDA  #4
000065  2D....  AND  np:uc
000068  1A      DEC  A
000069  D007    BNE  ?0005
00006B  AD....  LDA  np:uc
00006E  4B      SEB  2,A
00006F  8D....  STA  np:uc
                ?0005:
```

```
000072  AD....  LDA  np:data
000075  4304    BBS  2,A,?0007
000077  4B      SEB  2,A
000078  8D....  STA  np:data
                ?0007:
```

5) Switch 文は条件判定式の型、case ラベルの数に注意

switch 文では条件判定式の型、case ラベルの数により、ジャンプテーブルを用いた C ランタイムライブラリを使用します。
サイズの小さい C ランタイムライブラリを使用することで、ROM 効率がよくなります。

ex

```
switch( type ) {
case 1:
        ...
        ...
case xx:
}
```

型	case ラベルの数	C ランタイムライブラリ	ライブラリのサイズ
signed char	4	―	small
	5	?C_S_SWITCH_L06	↓
unsigned char	4	―	↓
	5	?C_S_SWITCH_L06	↓
signed short	1	?S_V_SWITCH_L06	↓
unsigned short	1	?S_V_SWITCH_L06	↓
signed int	1	?S_V_SWITCH_L06	↓
unsigned int	1	?S_V_SWITCH_L06	↓
signed long	1	?L_V_SWITCH_L06	big
unsigned long	1	?L_V_SWITCH_L06	

6）関数はプロトタイプ宣言を行う

　プロトタイプ宣言を行わないで関数呼び出しを行うと、コンパイラは int 型の戻り値を返す関数を呼び出したと判断します。
その為、無駄なコードが出力されますので、必ずプロトタイプ宣言を行ってから関数の呼び出しを記述しましょう。

ex

```
Warning[52]:

740 specific: 'No prototype for function "func1",

assuming that it returns int'
```

```
¥ 000000  20....  JSR  np:REFFN func1
¥ 000003  E8      INX
¥ 000004  E8      INX
```

```
¥ 000000  20....  JSR  np:REFFN func1
```

7）明示的なキャストの使用

明示的なキャストを使用します。

ex

```
char c1; int i1,i2;
c1 = i1 + i2;
```

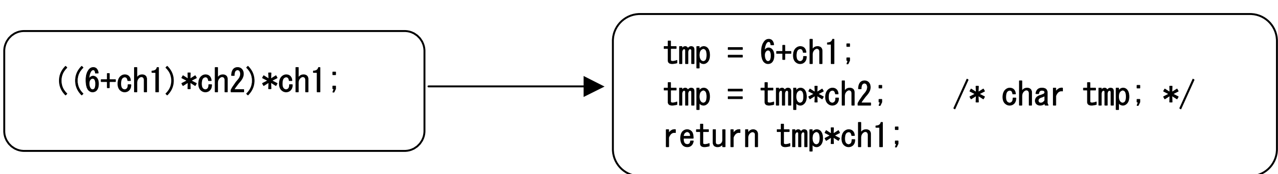
→

```
char c1; int i1,i2;
c1 = (char)i1 + (char)i2;
```

8）式はシンプルに記述

複雑な一つの式にするよりも、分割して、各式をシンプルに記述します。

ex



9）後置演算子の複雑な表現は避ける

後置演算子を使用する場合、演算子の優先度と結合規則を熟知していないと、予期せぬ結果を招くことがあります。

ex

```
char *src ;
char *dest ;
while( *src ){
    *dest++ = *src++ ;
}
```

→

```
char *src ;
char *dest ;
while( *src ){
    *dest = *src ;
     dest++ ;
     src++ ;
}
```

第 8 章
スタックの見積り方法

8.1 デフォルトのスタックサイズ

8.2 EXPR_STACK / INT_EXPR_STACKセグメント

8.3 CSTACKセグメント

8.4 C_ARGN / C_ARGZセグメント

8.5 RF_STACKセグメント

8.6 ICC740のランタム関数のスタック使用量

この章では、スタックの見積り方法について説明します。

8.1　デフォルトのスタックサイズ

740 用 C コンパイラパッケージではプロジェクト作成時のスタックサイズを以下に設定しています。

EXPR_STACK セグメント	32 バイト
INT_EXPR_STACK セグメント	32 バイト(tiny モデルは 16 バイト)
CSTACK セグメント	64 バイト
C_ARGN セグメント	リンカによる設定
C_ARGZ セグメント	リンカによる設定
RF_STACK セグメント	リンカによる設定

スタックサイズを変更する場合は次のように見積もってください。

8.2　EXPR_STACK / INT_EXPR_STACK セグメント

見積もり方法：(a)と(b)のサイズの大きい方を使用

(a) ランタイム関数のスタック使用量
　(a-1) 簡単な見積もり
　　・short 型、int 型の演算時は 4 バイト
　　・long 型の演算時は 8 バイト
　　・float 型、double 型の演算(加算等)時は 16 バイト
　(a-2) 詳細な見積もり
　　・使用ランタイム関数の最大の値

(b) ユーザ定義関数のリターン値のサイズで最大サイズの値

**※ リターン値を変数に代入しないで、そのまま演算に使用している場合は、
リターン値のサイズを加算**

8.3 CSTACK セグメント

見積もり方法：(a)～(f)のサイズを加算

(a) 関数の最大ネスティング数×2 バイト（戻り番地）
(b) 割り込み関数の最大ネスティング数×2 バイト（戻り番地）
(c) 割り込み時のスタック使用量（戻り番地用 2 バイト＋レジスタ用 4 バイト）
(d) MUL/DIV 命令使用時のスタック使用量
(e) アセンブリ言語記述のスタック操作量
(f) ランタイム関数の CSTACK 使用量

※　関数のネスティング構造は、リンカにオプション-xmso で参照可能

8.4　C_ARGN　/　C_ARGZ セグメント

見積もり方法：リンカによる設定のため、設定できません。

8.5　RF_STACK セグメント

見積もり方法：リンカが設定する 256 バイトより多くの領域を設定したい場合、増分を設定

```
-Z(NPAGE)RF_STACK+10
```

8.6　ICC740 の C ランタイム関数のスタック使用量

RunTime	Function	EXPR_STACK	CSTACK	Call
8 ビット整数				
?C_ADD_L01	add	2	2	
?C_SUB_L01	sub	2	2	
?C_MUL_L01	multiplication	2	3	
?C_FIND_SIGN_L01	character sign finding	2	3	
?C_DIVMOD_L01	character division and modulo	3	2	
?SC_DIV_L01	signed character division	3	5	?C_DIVMOD_L01 ?C_FIND_SIGN_L01
?UC_DIV_L01	unsigned division	2	3	
?SC_MOD_L01	signed character modulo	3	6	?C_DIVMOD_L01 ?C_FIND_SIGN_L01
?UC_MOD_L01	unsigned modulo	2	4	
?C_SHL_L01	left_shift_A_Y_steps	0	2	
?SC_SHR_L01	right_signed_shift_A_Y_steps	0	2	
?UC_SHR_L01	right_unsigned_shift_A_Y_steps	0	2	
?UC_CMP_L01	unsigned_compare	2	2	
?SC_CMP_L01	signed_compare	2	2	
16 ビット整数				
?S_ADD_L02	add	4	2	
?S_AND_L02	and	4	2	
?S_ORA_L02	or	4	2	
?S_EOR_L02	xor	4	2	
?S_SUB_L02	sub	4	2	
?S_MUL_L02	multiplication	6	3	
?S_FIND_SIGN_L02	character sign finding	4	3	
?US_DIV_L02	unsigned division	7	2	
?S_DIVMOD_L02	character division and modulo	7	2	
?SS_DIV_L02	signed word division	7	5	?S_DIVMOD_L02 ?S_FIND_SIGN_L02
?SS_MOD_L02	signed word modulo	7	6	?S_DIVMOD_L02 ?S_FIND_SIGN_L02
?US_MOD_L02	unsigned modulo	7	4	?S_DIVMOD_L02
?S_SHL_L02	left_shift_NOS_TOS:8_steps	3	2	
?SS_SHR_L02	right_signed_shift_NOS_TOS:8_steps	3	2	(?US_SHR_L02)
?US_SHR_L02	right_unsigned_shift_NOS_TOS:8_steps	3	2	
?SS_CMP_L02	signed_compare	4	2	
?US_CMP_L02	unsigned_compare	4	2	
?US_ZERO_L02	is_zero	2	2	
32 ビット整数				
?L_ADD_L03	add	8	2	
?L_AND_L03	and	8	2	
?L_OR_L03	or	8	2	
?L_XOR_L03	xor	8	2	
?L_SUB_L03	sub	8	2	
?L_MUL_L03	multiplication	12	3	
?L_FIND_SIGN_L03	character sign finding	8	3	
?UL_DIV_L03	unsigned division	13	2	
?L_DIVMOD_L03	long division and modulo	13	2	

?SL_DIV_L03	signed long division	13	5	?L_DIVMOD_L03 ?L_FIND_SIGN_L03 ?L_NOT_L03 ?L_INC_L03
?SL_MOD_L03	signed long modulo	13	6	?L_DIVMOD_L03 ?L_FIND_SIGN_L03 ?L_NOT_L03 ?L_INC_L03
?UL_MOD_L03	unsigned modulo	13	4	?L_DIVMOD_L03
?L_SHL_L03	left_shift_NOS_TOS:8_steps	5	2	
?SL_SHR_L03	right_signed_shift_NOS_TOS:8_steps	5	2	(?UL_SHR_L03)
?UL_SHR_L03	right_unsigned_shift_NOS_TOS:8_steps	5	2	
?SL_CMP_L03	signed_compare	8	2	
?UL_CMP_L03	unsigned_compare	8	2	
?L_ZERO_L03	is_zero	4	2	
?L_TEST_L03	test	8	4	?L_SUB_L03, (?L_ZERO_L03)
?L_NOT_L03	not	4	2	
?L_INC_L03	increment	4	2	
32 ビット浮動小数点				
?F_MUL_L04	Floating point Multiplication	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_OVERFLOW_TEST_L04) (?F_OVERFLOW_TEST1_L04) (?F_UNDERFLOW_L04) (?F_EXIT_L04)
?F_DIV_L04	Floating point division	12	4	?F_UNPACK_L04 (?F_PACK_L04) (?F_PACK_2_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_TEST1_L04) (?F_UP_ROUND_L04)
?F_ADD_L04	Floating point addition	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_EXIT_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_L04)
?F_SUB_L04	Floating point subtraction	12	4	?F_UNPACK_L04 (?F_PACK_2_L04) (?F_ROUND_L04) (?F_EXIT_L04) (?F_UNDERFLOW_L04) (?F_OVERFLOW_L04)
?SL_TO_F_L04	Cast a signed long integer	4	5	?F_0_SUB_L04
?UL_TO_F_L04	Cast a unsigned long integer	4	3	
?F_TO_L_L04	Cast a floating point	4	3	(?F_0_SUB_L04)
?F_CMP_L04	Float compare	8	2	
?F_UNPACK_L04	Internal entry	0	0	
?F_ROUND_L04	Internal entry	0	0	(?F_PACK_L04) (?F_UP_ROUND_L04)
?F_UP_ROUND_L04	Internal entry	0	0	(?F_PACK_L04)

?F_PACK_L04	Internal entry	0	0	(?F_UNDERFLOW_L04) (?F_OVERFLOW_TEST_L04) (?F_EXIT_L04)
?F_PACK_2_L04	Internal entry	0	0	?F_PACK_L04)
?F_UNDERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_TEST_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_OVERFLOW_TEST1_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_NEG_OVERFLOW_L04	Internal entry	0	0	(?F_EXIT_L04)
?F_0_SUB_L04	Internal entry	0	0	
?F_EXIT_L04	Internal entry	0	0	
switch				
?C_S_SWITCH_L06	switch (char):series	1	2	
?S_S_SWITCH_L06	switch (short):series	2	2	
?L_S_SWITCH_L06	switch (long):series	6	2	
?C_V_SWITCH_L06	switch (char)	2	2	
?S_V_SWITCH_L06	switch (short)	4	2	
?L_V_SWITCH_L06	switch (long)	6	2	
Function enter/leave				
?ENTER_L08	Function enter, save DP0 and DP1	0	4	
?LEAVE_L08	Function leave, restore DP0 and DP1	0	0	
?ENTER_FP_L08	Function enter, save DP0, DP1 and FP	0	6	
?LEAVE_FP_L08	Function leave, restore DP0, DP1 and FP	0	0	
Stack				
?IND_STK_16_DP0_L09	(tos) -> tos	2	3	
?IND_STK_16_DP1_L09	(tos) -> tos	2	3	
?IND_STK_16_L09	(tos) -> tos	2	5	
?IND_DP0_DP1_L09	(dp0) -> dp1	0	3	
?IND_DP1_DP0_L09	(dp1) -> dp0	0	3	
?STK_TO_DP0_L09	TOS -> dp0	2	3	
?STK_TO_DP1_L09	TOS -> dp1	2	3	
?DP0_TO_STK_L09	DP0 -> TOS	0	2	
?DP1_TO_STK_L09	DP1 -> TOS	0	2	
?IND_DP0_L09	(dp0) -> dp1	0	4	
?IND_DP1_L09	(dp1) -> dp0	0	4	
?PUSH_A_16_L09	0 A -> TOS	0	3	
?MOVE_LONG_L09	Block move (dp1) -> (dp0)	2	3	
?PUSH_DP0_L09	dp0 -> cpu stack	0	4	
?POP_DP0_L09	cpu stack --> dp0	0	0	
?PUSH_DP1_L09	dp1 -> cpu stack	0	4	
?POP_DP1_L09	cpu stack --> dp1	0	0	

四則演算等リターン値を持つものは、EXPR_STACKの領域内にリターン値を設定します。
Callの()はjmp命令を使用します。
下位関数のcallがある場合、下位関数のスタックサイズも含みます。
CSTACKでは下位関数1の戻り番地サイズも含みます。
?POP_DP0_L09の0は?PUSH_DP0_L09の2に対する減算値(-2)が含まれます。?POP_DP1_L09も同様です。
「32ビット浮動小数点」項目のInternal entryの値は上位関数に含めているため0としています。
?LEAVE_L08は?ENTER_L08のペア関数です。?LEAVE_FP_L08は?ENTER_FP_L08のペア関数です。

第 9 章
割り込み処理

9.1 割り込み処理

9.2 多重割り込み

この章では、割り込み処理について説明します。

9.1　割り込み処理

ICC740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。

(1) 割り込み処理関数の記述
(2) 割り込み禁止フラグ(Iフラグ)の設定
　　インライン関数により行います。
(3) 割り込みベクトル領域の登録
(4) 割り込みベクトルセグメントの設定

9.1.1　割り込み処理関数の記述例

この項では、3803グループでINT0割り込み(立ち上がりエッジ)が発生するたびに"counter"の内容を0にし、INT2割り込み(立ち下がりエッジ)が発生するたびに"counter"の内容をカウントアップするプログラムの記述例を示します。

割り込み処理関数の記述例

ソースファイルの記述例を示します。

```
#include    <intr740.h>         /* インライン関数用ヘッダファイル */
#include    "sfr_3803h.h"         /* 3803H グループ用 SFR ヘッダファイル */
unsigned char    counter ;

interrupt [30] void INT0_TimerZ( void )    /* 割り込み処理関数 */
{
    cld_instruction() ;         /* 10 進モードフラグの初期化 CLD 命令 */
    counter = 0 ;
}

interrupt [8] void Int2( void )    /* 割り込み処理関数 */
{
    cld_instruction() ;         /* 10 進モードフラグの初期化 CLD 命令 */
    if( counter < 9 ){
        counter++ ;
    } else {
        counter = 0 ;
    }
}

void main( void )
{
    /* ①割り込みエッジ選択ビットや割り込み要因ビットを設定 */
    INTEDGE.0 = 1 ;          /* INT0 立ち上がりエッジアクティブ */
    INTEDGE.3 = 0 ;          /* INT2 立ち下がりエッジアクティブ */
    INTSEL.0 = 0 ;           /* 割り込み要因 → INT0 割り込み */

    /* ②一命令以上おいてから、該当する割り込み要求ビットを 0(要求なし)にする */
    nop_instruction() ;      /* 一命令おく */
    IREQ1.0 = 0 ;            /* INT0 割り込み要求ビット → 要求なし */
    IREQ2.3 = 0 ;            /* INT2 割り込み要求ビット → 要求なし */

    /* ③該当する割り込み許可ビットを 1(許可)にする */
    ICON1.0 = 1 ;            /* INT0 割り込み許可ビット → 許可 */
    ICON2.3 = 1 ;            /* INT2 割り込み許可ビット → 許可 */

    enable_interrupt() ;     /* 割り込み許可 CLI 命令 */
    while( 1 ) ;             /* 割り込み待ちループ */
}
```

プログラム内で 10 進モードフラグを全く使用していなければ、割り込み処理関数内での初期化は不要です。

"sfr_3803h.h"で定義

```
sfr INTSEL   = 0x00039;
sfr INTEDGE  = 0x0003a;
sfr IREQ1    = 0x0003c;
sfr IREQ2    = 0x0003d;
sfr ICON1    = 0x0003e;
sfr ICON2    = 0x0003f;
```

ICON1.0 は、SFR ICON1 のビット 0 を示す。

プロセッサステータスレジスタ
(割り込み時は自動的に退避されます。)

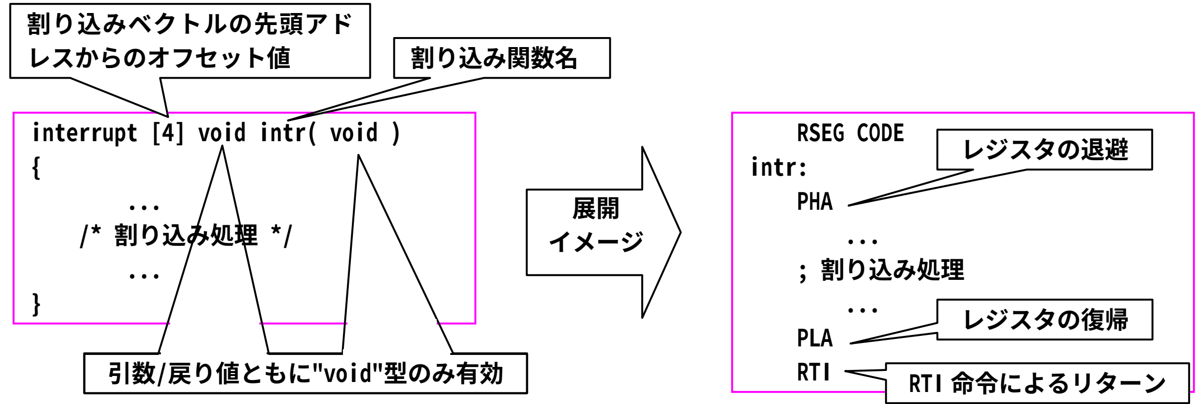
- D、T フラグ
  cstartup.s31 の init_C で初期化しています。
- I フラグ
  リセット直後、1(禁止)になっています。

9.1.2　割り込み処理関数の記述

ICC740では、割り込み処理関数をC言語又はアセンブリ言語で記述することができます。
この項では、基本的なC言語での割り込み処理関数の記述方法を説明します。

基本的な割り込み処理関数の記述(C 言語)

割り込み処理関数の定義は、拡張キーワード **interrupt** を用いて記述します。記述方法と展開イメージを示します。



上記のように記述すると、関数の入口と出口において、通常の関数の手続き以外に 740 ファミリのレジスタの退避・復帰と RTI 命令を生成します。なお、退避・復帰されるレジスタの数は割り込み処理関数の内容により異なります。

また、"interrupt"のすぐあとに、割り込みベクトルの先頭アドレスからのオフセット値をカッコで囲んで指定することにより、割り込み処理ルーチンのアドレスがそのベクトルに挿入され、割り込み発生時にこの関数を呼び出します。オフセット値を指定しない場合は、割り込み関数のベクトル・テーブルを cstartup.s31 ファイルに定義(割り込み処理関数を疑似命令"EXTERN"で外部参照宣言し、割り込みベクトルに登録する)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、コンパイル時にエラーとなります。

割り込み使用時の注意事項

通常時に使用しているローカル変数を持つ関数を、割り込み関数内で呼び出さないでください。
＜理由＞
ローカル変数を静的に配置しているため、通常時で関数呼び出し中に割り込みが発生すると、その関数のローカル変数を書き換えてしまいます。
通常時に呼び出している関数を割り込み関数内で呼び出したい場合は、同じ関数を通常用と割り込み用の2つ準備してください。
間接呼び出しの関数についても、上記と同様の制限があります。

コラム　割り込み処理関数から init_C(スタートアップ)へジャンプする方法

使用しない割り込み関数は、次のように、文を記述しない空の関数にしておきます。

```
interrupt [28] void Int1( void )
{
}
```

このように記述すると、RTI 命令のみ生成されます。そのため、予期しない割り込みが発生した場合、RTI 命令が実行された後、プログラムは続行されます。
　上記とは別の記述方法として、予期しない割り込みが発生した場合、リセット時と同様に、init_C から実行させるには、次のように記述します。

c ソースファイル

```
extern void init_C( void ) ;
      .
      .
      .
interrupt [28] void Int1( void )
{
    __asm( "JMP init_C" ) ;
}
```

- init_C(アセンブリ言語関数)を extern 宣言する
- 本来呼び出されることのない割り込み関数
- asm 関数を使用して init_C へジャンプする命令を記述する

cstartup.s31 ファイル

```
DEFFN init_C( 0, 0, 0, 0, 0x8000, 0, 0, 0 )
PUBLIC init_C
```

- init_C が出て来る前(ファイルの先頭)の部分にこの 2 行を追加

9.1.3　割り込み禁止フラグ(Iフラグ)の設定

割り込み禁止フラグ(Iフラグ)の設定

　割り込みの発生を有効にするには、CLI命令により割り込み禁止フラグ(Iフラグ)を0(許可)にする必要があります。ICC740では、インライン関数により、Iフラグの設定が行えます。インライン関数を使用する場合は、"intr740.h"をインクルードします。

```
#include    <intr740.h>

void main( void )
{
    enable_interrupt() ;
    while( 1 ) ;
    disable_interrupt() ;
}
```

展開イメージ

```
main:
    CLI          ; enable_interrupt()
?0003:
    BRA    ?0003
    SEI          ; disable_interrupt()
    RTS
```

　上記において、"enable_interrupt()"は I フラグを 0 にする"CLI"命令、"disable_interrupt()"は I フラグを 1 にする"SEI"命令にそれぞれ置き換えます。

9.1.4　割り込みベクトル領域の登録

割り込みベクトル領域の登録

付属の cstartup.s31 の INTVEC セグメントの内容を、ご使用のマイコンの割り込み領域に合わせて変更します。

```
    COMMON   INTVEC

?CSTARTUP_INTVEC:
        BLKB    0FFFEH - 0FFDCH -2      ; 3803 Group
#if 0
#if defined(MELPS_37600)
        BLKB    40H - 6      ; FFFA  ( FFC0 + 40 - 6)  (-v2)
#elif defined(MELPS_MULDIV)
        BLKB    20H - 4      ; FFFC
#else
        BLKB    20H - 2      ; FFFE
#endif
#endif
?CSTARTUP_RESETVEC:
        WORD    init_C
        ENDMOD  init_C
```

BLKB 疑似命令で、割り込みベクトル領域のサイズを割り当てます。この時、リセットは、故意に発生させる割り込みではないので、リセットベクトルの 2 バイト分を差し引いたサイズを指定します。

リセットベクトルは最下部に設定してあります。リセット時、プログラムはリセットベクトルに登録してある init_C からスタートします。

9.1.5　割り込みベクトルセグメントの設定

割り込みベクトルセグメントの設定

割り込みベクトルセグメントを設定する場合、マイコンに合わせてリンク・コマンド・ファイル"lnk740.xcl"中の割り込みベクトルセグメント"INTVEC"に、下記に示すようなアドレスを設定します。

※　割り込みベクトル領域の先頭アドレス及び終了アドレスは、各マイコンに合わせた値を設定してください。

9.2　多重割り込み

9.2.1　多重割り込みの使用方法

この項では、多重割り込みを使用する場合に必要な設定と注意点、関数の記述方法を説明します。

多重割り込み使用時の注意事項

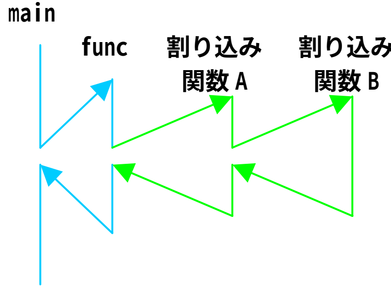
下の図のようなプログラムの場合、割り込み関数 B は、次の点に注意してプログラミングしてください。

- フラグ設定やカウンタの更新程度の簡単な処理にしてください。
- C ランタイムライブラリを呼び出さないでください。
- 多重割り込みを許可する割り込み関数で使用しているローカル変数を持つ関数を、多重割り込み関数で使用しないで下さい。

＜理由＞

C ランタイムライブラリを呼び出したりリターン値が char 型以外の関数を呼び出したりすると、割り込み式スタックを使用します。

割り込み式スタックは、割り込み処理で式の評価を行っている間、一時的に結果を保持する領域です。この領域は全ての割り込みにおいて共通に使用されています。そのため、割り込み関数 B で割り込み式スタックを使用すると、割り込み関数 A で使用している部分を上書きしてしまいます。



※　C ランタイムライブラリは、コンパイラが必要に応じて自動的に呼び出すもので、プログラム上で明示的に呼び出せるものではありませんので、ご注意ください。 C ランタイムライブラリは、signed char 型の割り算等、多くの演算で呼び出されます。

コンパイルオプション

多重割り込みを使用する場合、ICC740のコンパイルオプションに、 -h を追加する必要があります。

コラム　多重割り込みで割り込み式スタックを使用したい場合の応用例

上の図で、割り込み関数 B において割り込み式スタックを使用したい場合、次のような方法があります。

- 割り込み関数 A で割り込み式スタックを使用している間、割り込みを禁止状態にします。 ただし、その間、プログラムのリアルタイム性が損なわれることになります。
- 割り込み関数 B の演算部分をアセンブリ言語で記述して、割り込み式スタックを使用しないようにします。
- 割り込み関数 A で割り込み式スタックを使用していなければ、割り込み関数 B で割り込み式スタックを使用しても問題ありません。
  (割り込み関数 A 又は B のどちらか一方での使用であれば、割り込み式スタックの内容は上書きされません。)

9.2.2　多重割り込みに関する定義

付属のcstartup.s31の次の部分で、多重割り込みを使用するための設定を行っています。

```
;---------------------------------------------------------------
; Turning off 'interruptable ISRs':
; Do this if you need the extra byte(s)
;
; 1. Uncomment the define below
; 2. Assemble this file
; 3. Include the result in your linker command file:
;     -C cstartup.r31
;
; Variable '?IES_USAGE' and its initialization will no longer
; be included.
;---------------------------------------------------------------
;#define NO_INTERRUPTABLE_ISR
```

(多重割り込みを)使用する場合、この行はこのまま。

デフォルトでは、「使用する」になっていますので、使用する場合、ここはそのままにしておきます。使用しない場合は、最終行の先頭にある「;」を削除し、コメント行ではなく有効行にします。

```
#ifndef NO_INTERRUPTABLE_ISR
;---------------------------------------------------------------;
;  ?IES_USAGE - Determines if the IES is setup and used.
;
;  This variable is used for interrupt functions when compiling
;  with the '-h' option.
;---------------------------------------------------------------;

    RSEG    ZPAGE
    PUBLIC  ?IES_USAGE
?IES_USAGE:
    BLKB    1
#endif
```

```
#ifndef NO_INTERRUPTABLE_ISR
;---------------------------------------------------------------;
;  Initialize ?IES_USAGE:
;   1   IES not used
;   0   First use of IES, need to setup IES
;  <0   IES already setup and used
;---------------------------------------------------------------;
        LDA  #1
        STA  zp:?IES_USAGE
#endif
```

この2箇所で、多重割り込みのための設定を行っています。多重割り込みを使用する(NO_INTERRUPTABLE_ISRが定義されていない)場合、この部分がアセンブルされます。使用しない(NO_INTERRUPTABLE_ISRが定義されている)場合は、この部分はアセンブルされません。

9.2.3　多重割り込み処理関数の記述例

前述の注意事項をふまえ、次の仕様でのプログラム記述例を以下に示します。

関数	処理内容	割り込み優先度	割り込み状態
INT1 割り込み処理関数	異常処理(緊急停止)	1	多重なし
シリアル I/O1 受信割り込み処理関数	通信受信	2	1つの割り込みのみ多重 OK
タイマ 2 割り込み処理関数	一般処理	3	多重 OK

main 関数の③のところで、割り込みを発生させたいものだけ許可ビットを許可にしています。タイマ 2 の割り込み処理関数では、冒頭部分で CLI 命令を行うことにより、全ての割り込みを許可しています。これにより、多重割り込みが可能となります。ただし、自分自身への割り込みは禁止にしています。シリアル I/O1 受信の割り込み処理関数では、INT1 以外の割り込みを禁止に設定してから CLI 命令することにより、INT1 の割り込みのみ多重割り込み可能となります。INT1 の割り込み処理関数では、関数実行中、割り込み禁止状態になっているため、多重割り込みは発生しません。

なお、リセット直後は、割り込みは禁止状態(I フラグ=1)になっています。また、割り込み関数に入ると、割り込みは禁止状態(I フラグ=1)になり、割り込み関数を出ると(RTI 命令で)、割り込みは許可状態(I フラグ=0)になります。

```
#include    <intr740.h>      /* インライン関数用ヘッダファイル */
#include    "sfr_3803h.h"    /* 3803H グループ用 SFR ヘッダファイル */

void main( void )
{
        ・
        ・
        ・
    /* ①割り込みエッジ選択ビット(や割り込み要因ビット)を設定 */
    INTEDGE.1 = 1 ;          /* INT1 立ち上がりエッジアクティブ */

    /* ②一命令以上おいてから、該当する割り込み要求ビットを 0(要求なし)にする */
    nop_instruction() ;      /* 一命令おく */
    IREQ1.1 = 0 ;            /* INT1 割り込み要求ビット → クリア */
    IREQ1.2 = 0 ;            /* シリアル I/O1 受信割り込み要求ビット →クリア */
    IREQ1.7 = 0 ;            /* タイマ 2 割り込み要求ビット →クリア */

    /* ③該当する割り込み許可ビットを 1(許可)にする */
    ICON1.1 = 1 ;            /* INT1 割り込み許可ビット → 許可 */
    ICON1.2 = 1 ;            /* シリアル I/O1 受信割り込み許可ビット → 許可 */
    ICON1.7 = 1 ;            /* タイマ 2 割り込み許可ビット → 許可 */

    enable_interrupt() ;     /* 割り込み許可 CLI 命令 */

    while( 1 ) ;             /* 割り込み待ちループ */
}
```

割り込みを発生させたいものだけ許可する

```
#include    <intr740.h>     /* インライン関数用ヘッダファイル */
#include    "sfr_3803h.h"   /* 3803H グループ用 SFR ヘッダファイル */

unsigned char    Cntr ;
unsigned char    T_5msec = 1 ;
unsigned char    T_flg = 0 ;
unsigned char    Val ;

interrupt [28] void Int1( void )        /* INT1 割り込み処理関数[緊急停止] */
{
    Cntr = 0 ;
}

interrupt [26] void SIO1R( void )      /* シリアル I/01 受信割り込み処理関数 */
{
    ICON1.2 = 0 ;              /* シリアル I/01 受信割り込み許可ビット → 禁止 */
    ICON1.7 = 0 ;              /* タイマ 2 割り込み許可ビット → 禁止 */
    enable_interrupt() ;       /* 割り込み許可 CLI 命令 */
            ¦
    T_flg = 1 ;
            ¦
    disable_interrupt() ;      /* 割り込み禁止 SEI 命令 */
    ICON1.2 = 1 ;              /* シリアル I/01 受信割り込み許可ビット → 許可 */
    ICON1.7 = 1 ;              /* タイマ 2 割り込み許可ビット → 許可 */
}

interrupt [16] void Timer2( void )      /* タイマ 2 割り込み処理関数 */
{
    ICON1.7 = 0 ;              /* タイマ 2 割り込み許可ビット → 禁止 */
    enable_interrupt() ;       /* 割り込み許可 CLI 命令 */

    if( T_5msec ){
        T_5msec = 0 ;
        if( Cntr < 9 ){
            Cntr++ ;
        } else {
            Cntr = 0 ;
        }
        if( T_flg ){
            Val = Cntr ;
            T_flg = 0 ;
        }
    } else {
        T_5msec = 1 ;
    }

    disable_interrupt() ;      /* 割り込み禁止 SEI 命令 */
    ICON1.7 = 1 ;              /* タイマ 2 割り込み許可ビット → 許可 */
}
```

・多重割り込み禁止
・割り込み式スタック使用禁止
(注意事項参照)

・緊急停止のみ多重割り込み可能
・割り込み式スタック

・多重割り込み可能
・割り込み式スタック使用可能

自分自身(タイマ 2)への割り込みを禁止

自分自身(タイマ 2)への割り込みを許可

740ファミリ用Cコンパイラパッケージ
M3T-ICC740
アプリケーションノート

発行年月日　2006年7月16日　Rev.1.00

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

© 2006. Renesas Technology Corp. and Renesas Solutions Corp., All rights reserved. Printed in Japan.

740 ファミリ用 C コンパイラパッケージ
M3T-ICC740
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

RJJ06J0005-0100