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。

User's Manual

ICC コンパイラ・プログラミング・ガイド

（740 ファミリ） 第 2 版

ルネサス エレクトロニクス
www.renesas.com

Rev.2.00

Second edition: June 1997
Part no: ICC740–2

本書は、株式会社ソフトボートが IAR Systems AB の許可を得て作成した ICC 740 の日本語マニュアルです。本書の一部または全部を無断で複製、複写、転記することは、法律で禁じられていますので御注意ください。

株式会社ソフトボートは、IAR Systems AB **の日本における総代理店です。**

ウエルカム

ICC コンパイラ・プログラミング・ガイド（三菱 740 ファミリ）へようこそ。

本ガイド書は、Embedded Workbench (EW) 740 のコンパイラに関するリファレンス情報を提供します。そしてこの情報は、組み込みワークベンチ版とコマンドライン版の両方のツールに適用されます。

組み込みワークベンチ用

組み込みワークベンチを使用する場合、「組み込みワークベンチ・インターフェース・ガイド」を読んで、組み込みワークベンチのコマンドとダイアログ・ボックスについて習熟するようにしてください。

コマンドライン用

EW 740 には、コマンドラインから起動する EXE 版のコンパイラも同梱されています。

740 アセンブラでプログラミングする場合には、「IMA アセンブラ・プログラミング・ガイド（三菱 740 ファミリ）」を参照してください。

製品にオプションの Spy Workbench（SW または C-SPY）740 も含まれている場合、これを用いてデバッグする際には、「SW ユーザ・ガイド」を参照してください。

本書について

本ガイド書は、次の章から構成されています。

第 1 章「チュートリアル」では、C コンパイラを使用して一連の一般的なプログラムを開発する方法を説明し、コンパイラの最も重要な機能をいくつか解説します。また、C コンパイラを使用した一般的な開発サイクルについても触れます。

第 2 章「C コンパイラ・オプションの要約」では、C コンパイラ・オプションの設定方法を説明し、各オプションの要約を示します。

第 3 章「C コンパイラ・オプションのリファレンス」では、各コンパイラ・オプションに関する内容を示します。

第 4 章「構成」では、さまざまな要件に合わせた C コンパイラの構成方法を説明します。

第 5 章「データ表示」では、コンパイラが各 C データ型を表記する方法を説明し、効率的なコード化の推奨事項を示します。

第 6 章「一般的な C ライブラリの定義」では、C ライブラリ関数の概要を説明し、ヘッダー・ファイルに従ってこれらの関数を要約します。

第 7 章「C ライブラリ関数のリファレンス」では、各ライブラリ関数に関するリファレンス情報を示します。

第 8 章「言語拡張機能」では、拡張キーワード、**#pragma** キーワード、定義済みシンボル、および 740 C コンパイラに固有なインライン関数を要約します。

第 9 章「拡張キーワードのリファレンス」では、各拡張キーワードに関するリファレンス情報を示します。

第 10 章「**#pragma** 疑似命令のリファレンス」では、**#pragma** キーワードに関するリファレンス情報を示します。

第 11 章「定義済みシンボルのリファレンス」では、定義済みシンボルに関するリファレンス情報を示します。

第 12 章「インライン関数のリファレンス」では、インライン関数に関するリファレンス情報を示します。

第 13 章「37600 インライン関数のリファレンス」では、37600 ファミリ・マイクロプロセッサでのみ利用できるインライン関数に関するリファレンス情報を示します。

第 14 章「アセンブリ言語インタフェース」では、C プログラムとアセンブリ言語ルーチン間のインタフェースを説明します。

第 15 章「アセンブラ・インタフェース・キーワードのリファレンス」では、C 言語へのインタフェースを提供するアセンブラ演算子と疑似命令に関するリファレンスを示します。

第 16 章「セグメント・リファレンス」では、C コンパイラのセグメントの使用に関するリファレンス情報を示します。

第 17 章「ICC740 制限事項」では、740 C コンパイラのランタイム・モデルでの制限事項を説明します。

第 18 章「K & R および ANSI C 言語の定義」では、K & R 準拠の C 言語と ANSI 規格 C 間の違いについて説明します。

第 19 章「診断」では、コンパイラの警告メッセージとエラー・メッセージをリストします。

第 20 章「コマンドライン版用環境変数」では、コマンドラインから EW 740 の各ツールを起動する場合の環境変数についての情報を提供します。

前提条件

本ガイド書では、既に次の項目の操作知識があるものと想定して説明が行われます。

- 740 プロセッサ
- C プログラミング言語
- ホスト・システムに応じた Windows の基本知識

本書では、C 言語自体の説明は行いません。C 言語の説明については、Kernigham と Richie 共著の C Programming Language（プログラミング言語 C）を推奨します。この本の最新版は、ANSI C についても記載しています。

表記規則

本書では、次の表示規則を使用します。

形式	目的
computer	入力する、あるいは画面に表示されるテキストです。
parameter	コマンドの一部として入力しなければならない実際の値を表したラベルです。
[option]	コマンドの任意指定の部分です。
{a | b | c}	コマンドの選択肢です。
参照	本ユーザ・ガイドの別の箇所、あるいは別のガイド書の参照を意味します。
組み込みワークベンチ用	組み込みワークベンチのインタフェースに固有な命令です。
コマンドライン用	コマンドライン版に固有な命令です。
共通用	上記の **2** つのツールに共通な操作です。

本ガイド書では、Kernigham、Richie 著の「プログラミング言語 C」の省略表現として K & R を使用しています。

本書中で、

- **SW とは、Spy Workbench の略で、シミュレータデバッガ C-SPY の Windows 95/NT 環境対応版です。**
- **その他、特に「コマンドライン版」との断わりがない「C-SPY」は、上記の「SW」のことを指します。**

目次

第 1 章　チュートリアル

本章では、740 C コンパイラを使った一般的なプログラムの開発手順を説明するとともに、C コンパイラが提供する機能をいくつか紹介します。

この章を読む前に、次の点を確認してください。

- C コンパイラソフトウェアを既にインストールしていること。
- 740 プロセッサのアーキテクチャおよび命令セットを熟知していること。詳細は、メーカーのデータブックを参照してください。

また『EW インタフェース・ガイド』にある入門チュートリアルを完了し、ご使用のインターフェースについて習熟されることをお勧めします。

チュートリアル・ファイルの要約

下表に、この章で使用するチュートリアル・ファイルの概要を示します。

ファイル名	デモ内容
tutor1	簡単な **C** プログラムのコンパイルと実行
tutor2	シリアル **I/O** の利用
tutor3	割り込み処理

サンプルプログラムの実行

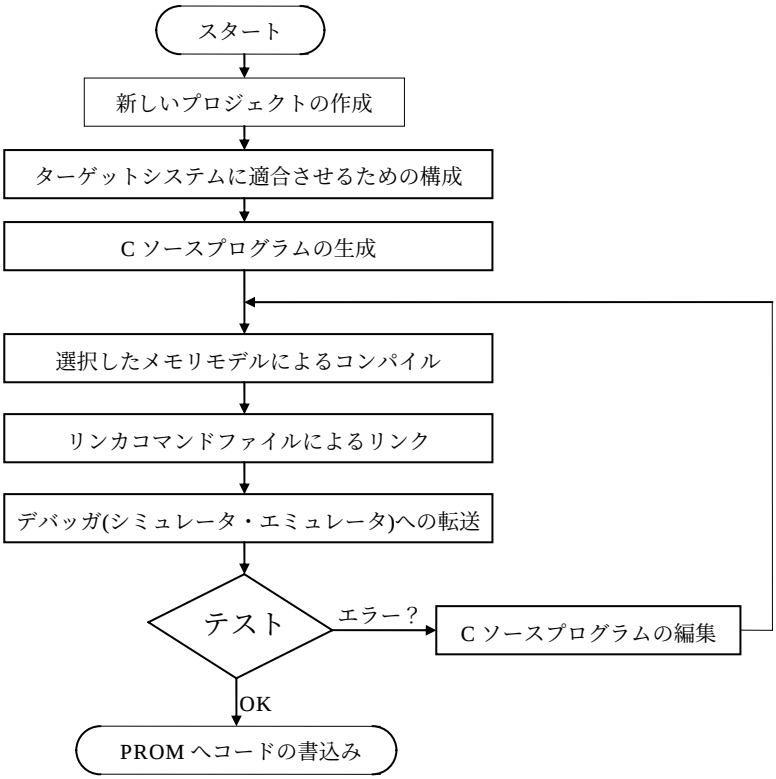
このチュートリアルでは、組み込みワークベンチまたは DOS C-SPY からオプションである SW（C-SPY シミュレータ）を使って、サンプルプログラムを実行する方法を説明します。

コマンドラインを利用することによって、EPROM エミュレータおよびデバッガを使ってターゲットシステムでサンプルプログラムを実行できます。この場合は、まず I/O ルーチンの設定を行ってください。

作成されるリストファイルをチェックし、このチュートリアルを実行することもできます。**.1st** および **.map** ファイルでは、監視すべきメモリ領域を示します。

一般的な開発サイクル

通常、プログラム開発は下図のようなサイクルをとります。



以降のチュートリアルは、このサイクルに従って説明を進めます。

スタート

C コンパイラを用いるプロジェクト開発の最初のステップは、ターゲットシステムに適合した適切な構成を決めることです。

ターゲットシステムに適合した構成

このチュートリアルプログラムには、少数のコードしか含まれていませんので、タイニイ・メモリモデルで実行できます。デフォルトの**-v0** オプション（MUL および DIV 命令付き 38XXX をサポート）を用います。

各プロジェクトには、ターゲットシステムにおけるメモリマップの詳細情報を持つ XLINK コマンドファイルが必要です。

リンカコマンドファイルの選択

小さなメモリモデルに対応したリンカコマンドファイル **lnk740.xcl** が、**icc740** サブディレクトリにあります。

組み込みワークベンチエディタや MS-DOS **edit** エディタなどのテキストエディタを使って、**lnk740.xcl** ファイルを調べます。

lnk740.xcl ファイルの第 1 行目には、次の XLINK コマンドがあり、CPU タイプが 740 であることを定義しています。

```
-c740
```

次に一連の**-Z** コマンドが続き、コンパイラが使用するセグメントを定義しています。キーセグメントは以下の通りです。

セグメント型	セグメント名	アドレス範囲
ZPAGE	ZPAGE, C_ARGZ, Z_UDATA, Z_IDATA, EXPR_STACK	0x00〜0xFF
DATA	CSTACK	0x100〜0x1FF
NPAGE	C_ARGN, N_UDATA, N_IDATA, ECSTR	0x1000〜0x7FFF
CODE	RCODE, CODE, Z_CDATA, N_CDATA, C_ICALL, C_RECFN, CSTR, CCSTR, CONST	0x8000〜0xFDFF
CODE	INTVEC	0xFFE0〜0xFFFF

ファイルは次に、**printf** および **scanf** に使用されるルーチンを定義します。

最終行には次のコマンドがあり、適切な C ライブラリをロードします。

c1740t

他の C ライブラリの詳細は、54 ページの「ランタイム・ライブラリ」を参照してください。これらの定義はあくまでも一時的なものです。自分のプロジェクトに合わない、あるいは最適でない場合は、あとで変更することができます。

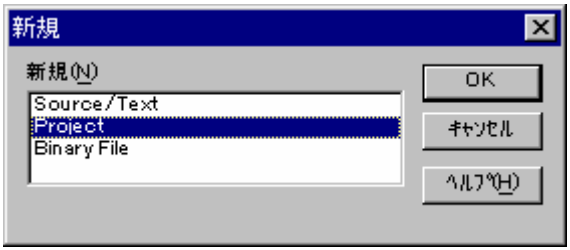
ターゲットメモリに適した設定の詳細は、56 ページの「記憶場所」を参照してください。スタックサイズ選択の詳細は、57 ページの「スタックサイズ」を参照してください。

新しいプロジェクトの作成

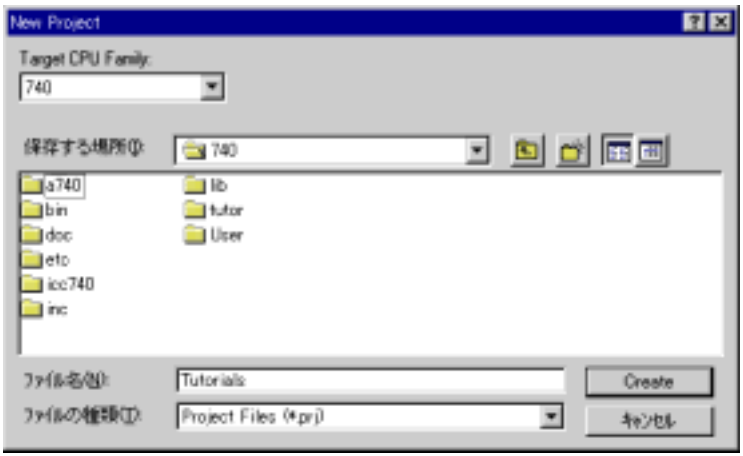
組み込みワークベンチ用新しいプロジェクトの作成

まず組み込みワークベンチを起動し、次の手順に従ってチュートリアル用のプロジェクトを作成します。

File メニューから **New...**を選択すると、次のダイアログボックスが表示されます。



Project を選択し **OK** をクリックすると、**New Project** ダイアログボックスが表示されます。**Project Filename** ボックスに **Tutorials** と入力し、**Target CPU Family** を **740** に設定します。

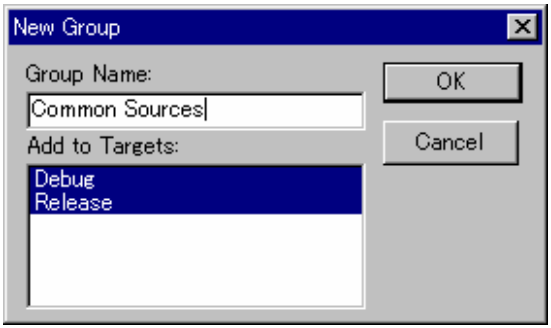


Create をクリックすると、新規プロジェクトが作成されます。

新規プロジェクトの **Project** ウィンドウが表示されます。必要であれば、**Targets** ドロップダウンリストボックスから **Debug** を選択し、**Debug** ターゲットを表示します。

次の手順に従って、チュートリアル用ソースファイルを含むグループを作成します。

Project メニューから **New Group...**を選択し、グループ名に **Common Sources** と入力します。デフォルトでは両方のターゲットが選択されていますので、このグループには両方のターゲットが追加されます。

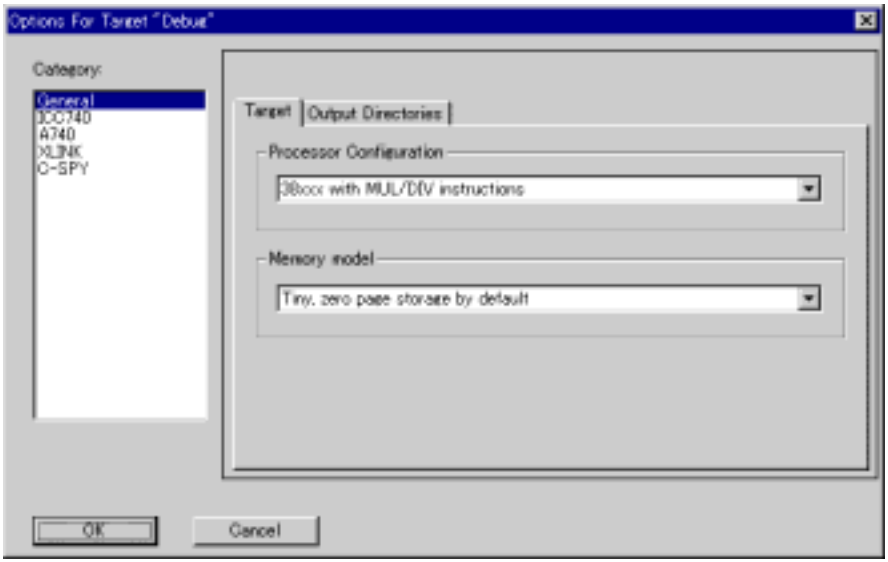


OK をクリックして、グループを作成します。グループ名がプロジェクト・ウィンドウに表示されます。

次に選択したプロセッサとメモリモデルに合うように、ターゲット・オプションを設定します。

プロジェクト・ウィンドウ内の **Debug** フォルダ・アイコンを選択し、**Project** メニューから **Options...**を選び、さらに **Category** リスト内の **General** を選択して、ターゲット・オプション・ページを表示させます。

Processor Configuration を **38XXX with MUL and DIV instructions** に設定して **Tiny** メモリモデルを選択します。



そして **OK** を選択して、ターゲット・オプションを保存します。

コマンドライン用新しいプロジェクトの作成

特定のプロジェクト用のファイルはすべて 1 つのディレクトリに保存し、他のプロジェクトやシステムファイルと分けておくとよいでしょう。

チュートリアル・ファイルは、**icc740** ディレクトリにインストールされています。デフォルトでは次のコマンドを入力し、このディレクトリを選択します。

```
cd \iar\ewxx\icc740
```

チュートリアルの実行中は、このディレクトリを使いますので、作成したファイルはここに保存されます。

チュートリアル 1　プログラムの作成

最初のチュートリアルでは、プログラムのコンパイル、リンクおよび実行方法を説明します。

プログラムの入力

第 1 プログラムは、標準 C 機能のみを使用した簡単なプログラムです。このプログラムでは、変数をインクレメントする関数を繰り返し呼出します。

```
#include <stdio.h>
zpage int call_count;
zpage unsigned char my_char;
const char con_char='a';

void do_foreground_process(void)
{
  call_count++;
  putchar(my_char);
}


void main(void)
{
  int my_int=0;
  call_count=0;
  my_char=con_char;
  while (my_int<100)
  {
   do_foreground_process();
   my_int++;
  }
}
```

組み込みワークベンチ用プログラムの入力

File メニューから **New** を選択し、**New** ダイアログボックスを表示します。

Source/Text を選択し、**OK** をクリックして、新しいテキストドキュメントを開きます。

上記のプログラムを入力し、**tutor1.c** ファイルとして保存します。

プログラムを入力するかわりに、**icc740\tutor** ディレクトリにある **tutor1.c** をコピーすることもできます。

コマンドライン用プログラムの入力

MS-DOS **edit** エディタなどの標準テキストエディタを使って、上記のプログラムを入力し、**tutor1.c** ファイルとして保存します。プログラムを入力するかわりに、C コンパイラファイルディレクトリにある **tutor1.c** をコピーすることもできます。

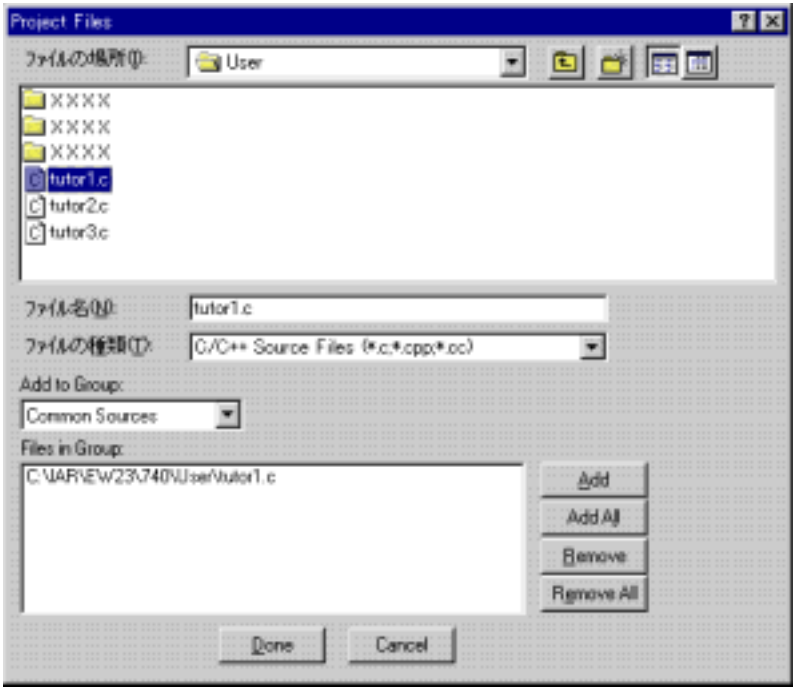
では、このソースファイルをコンパイルしてみましょう。

プログラムのコンパイル

組み込みワークベンチ用プログラムのコンパイル

プログラムをコンパイルするには、まず次の手順に従ってこのプログラムを **Tutorials** プロジェクトに追加します。

Project メニューから **Files...**を選択し、**Project Files** ダイアログボックスを表示します。ダイアログボックス上部にあるファイル一覧から **tutor1.c** を選択し、Add をクリックして **Common Sources** グループに **tutor1.c** ファイルを追加します。

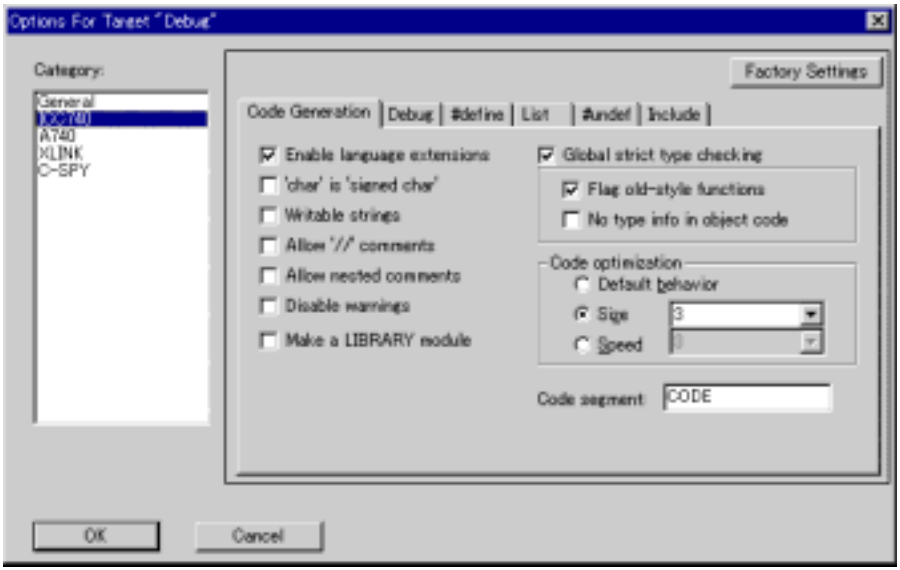


次に **Done** をクリックして、**Project Files** ダイアログボックスを閉じます。

+シンボルをクリックし、Project ウィンドウツリー内のファイルを表示します。

次の手順に従って、プロジェクトに合ったコンパイラオプションを設定します。

Project ウィンドウの **Debug** フォルダをクリックし、**Project** メニューから **Options...**を選択します。**Category** リストから **ICC740** を選択し、C コンパイラのオプションページを表示します。

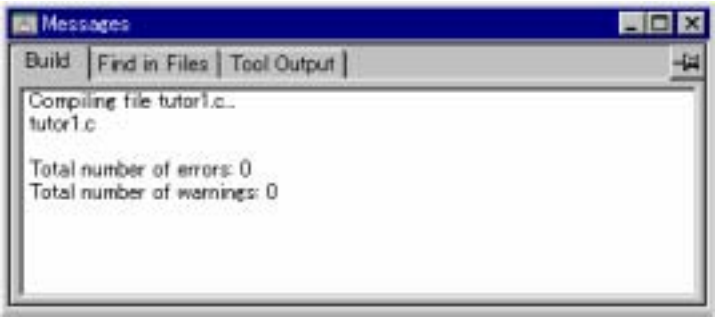


Options...ダイアログボックスの適当なページ上に、以下のオプションが選択されていることを確認します。

ページ	オプション
Code generation	Enable language extensions
Debug	Generate debug information
List	Generate list file
	Insert mnemonics

これらの変更を行ったのち、**OK** を選択して指定したオプションを設定します。

ファイルをコンパイルするには、プロジェクト・ウィンドウでファイルを選択し、**Project** メニューから **Compile** を選択します。進捗状況がメッセージ・ウィンドウに表示されます。



コマンドライン用ファイルのコンパイル

プログラムをコンパイルするには、次のコマンドを入力します。

```
icc740 tutor1 –v0 -mt -r -L -e -q –I\iar\ewxx\inc\
```

ここで使用するコンパイルオプションを、次に示します。

ｵﾌﾟｼｮﾝ	内容
-v0	**MUL および DIV 命令付きの 38XXX を選択する**
-mt	**タイニイ・メモリモデルを選択する**
-r	**コードをデバッグ可能にする**
-L	**リストファイルを作成する**
-e	**拡張コマンド（このチュートリアルでは使用しない）を可能にする**
-q	**リスト内の C にアセンブラコードを含む**
-I	**インクルード・ファイル用のパス名を指定する**

このコマンドを実行すると、オブジェクトモジュール **tutor1.r31** とリストファイル **tutor1.lst** が作成されます。

共通用リストの表示

作成されたリストファイルを調べ、変数がそれぞれのセグメントにどのように割当てられているかを確認します。

```
#############################################################################
#                                                                           #
# IAR 740 C-Compiler V2.11A/WIN                                             #
#                                                                           #
#       Compile time  =  09/Feb/2000  17:00:52                              #
#       Target option =  740 with MUL/DIV                                   #
#       Memory model  =  tiny                                               #
#       Source file   =  c:\iar\ew23\740\user\tutor1.c                      #
#       List file     =  c:\iar\ew23\740\user\debug\list\tutor1.lst         #
#       Object file   =  c:\iar\ew23\740\user\debug\obj\tutor1.r31          #
#       Command line  =  -v0 -mt -OC:\IAR\EW23\740\User\Debug\Obj\ -e -gA    #
#                        -z3 -RCODE -r0 -LC:\IAR\EW23\740\User\Debug\List\   #
#                        -q -t8 -IC:\IAR\EW23\740\inc\                      #
#                        C:\IAR\EW23\740\User\tutor1.c                      #
#                                                                           #
#                           Copyright 1999 IAR Systems. All rights reserved. #
#############################################################################

   \                            NAME       tutor1(16)
   \                            RSEG       CODE(0)
   \                            RSEG       CONST(0)
   \                            RSEG       Z_UDATA(0)
   \                            PUBLIC     call_count
   \                            PUBLIC     con_char
   \                            PUBLIC     do_foreground_process
   \                            DEFFN
         do_foreground_process(0,0,0,0,32768,0,0,0),putchar
   \                            PUBLIC     main
   \                            DEFFN
         main(2,0,0,0,32768,0,0,0),do_foreground_process
   \                            PUBLIC     my_char
   \                            EXTERN     putchar
   \                            DEFFN      putchar(32770,0,0,0)
   \                            EXTERN     ?CL740MDT_2_11_L00
   \                            EXTERN     ?SS_CMP_L02
   \                            RSEG       CODE
   \                        do_foreground_process:
      1          #include <stdio.h>
      2          zpage int call_count;
      3          zpage unsigned char my_char;
      4          const char con_char='a';
```

```
    5
    6          void do_foreground_process(void)
    7          {
    8            call_count++;
\   000000   E6..                 INC     zp:call_count
\   000002   D002                 BNE     ?0013
\   000004   E6..                 INC     zp:call_count+1
\                    ?0013:
    9            putchar(my_char);
\   000006   A5..                 LDA     zp:my_char
\   000008   85..                 STA     zp:PRMBZ putchar
\   00000A   A900                 LDA     #0
\   00000C   85..                 STA     zp:PRMBZ putchar+1
\   00000E   20....               JSR     np:REFFN putchar
\   000011   E8                   INX
\   000012   E8                   INX
   10          }
\   000013   60                   RTS
\                    main:
   11
   12          void main(void)
   13          {
   14            int my_int=0;
\   000014   3C00..               LDM     #0,zp:LOCBZ main
\   000017   3C00..               LDM     #0,zp:LOCBZ main+1
   15            call_count=0;
\   00001A   3C00..               LDM     #0,zp:call_count
\   00001D   3C00..               LDM     #0,zp:call_count+1
   16            my_char=con_char;
\   000020   AD....               LDA     np:con_char
\   000023   85..                 STA     zp:my_char
\                    ?0011:
   17            while (my_int<100)
\   000025   32                   SET
\   000026   CA                   DEX
\   000027   A5..                 LDA     zp:LOCBZ main+1
\   000029   CA                   DEX
\   00002A   A5..                 LDA     zp:LOCBZ main
\   00002C   12                   CLT
\   00002D   32                   SET
```

```
\   00002E   CA                  DEX
\   00002F   A900                LDA     #0
\   000031   CA                  DEX
\   000032   A964                LDA     #100
\   000034   12                  CLT
\   000035   20....              JSR     np:?SS_CMP_L02
\   000038   B00B                BCS     ?0010
\                     ?0012:
  18            {
  19              do_foreground_process();
\   00003A   20....              JSR     np:REFFN do_foreground_process
  20              my_int++;
\   00003D   E6..                INC     zp:LOCBZ main
\   00003F   D002                BNE     ?0014
\   000041   E6..                INC     zp:LOCBZ main+1
\                     ?0014:
  21            }
  22          }
\   000043   80E0                BRA     ?0011
\                     ?0010:
\   000045   60                  RTS
\                           RSEG      CONST
\                     con_char:
\                           BYTE      'a'
\                           RSEG      Z_UDATA
\                     call_count:
\                           BLKB      2
\                     my_char:
\                           BLKB      1
\                           END

Source file: c:\iar\ew23\740\user\tutor1.c
Errors: none
Warnings: none
Code size: 70
Constant size: 1
Static variable size: ZPage(3) NPage(0) Bit(0)
```

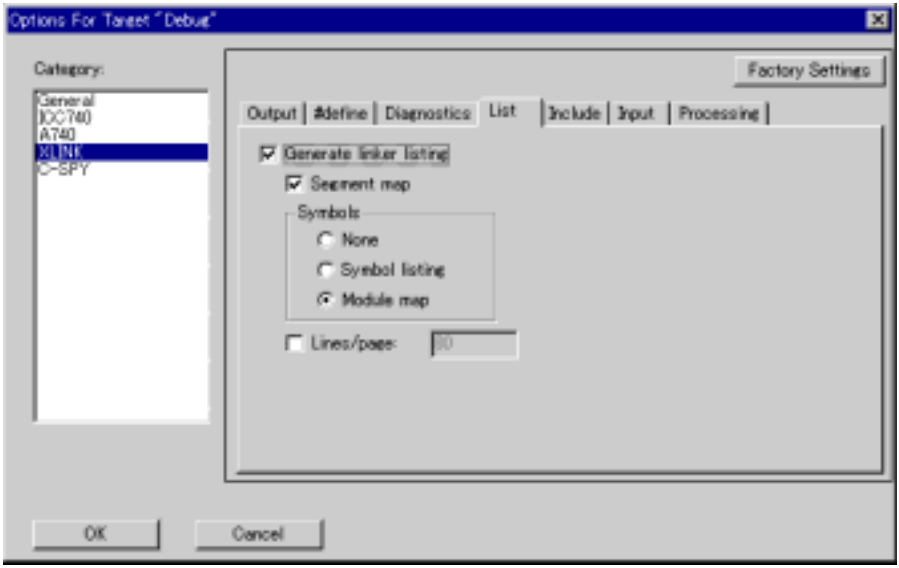
プログラムのリンク

組み込みワークベンチ用プログラムのリンク

まず、XLINK リンカのオプションを設定します。プロジェクト・ウィンドウの **Debug** フォルダをクリックし、**Project** メニューから **Options..**を選択します。**Category** リストから **XLINK** を選択し、XLINK のオプションページを表示します。

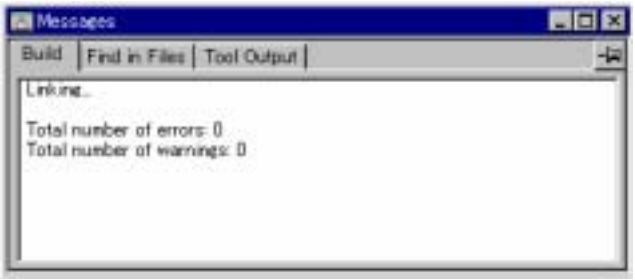
次に、**List** タブをクリックし、リスト・オプションのページを表示します。

Generate linker listing と **Segment map** を選択し、マップファイルを **tutorials.map** に生成します。



OK をクリックし、XLINK のオプションを保存します。

オブジェクトファイルをリンクし、デバッグ可能なコードを生成するには、**Project** メニューから **Link** を選択します。進捗状況がメッセージ・ウィンドウに表示されます。



リンクの結果、コードファイル **tutorials.d31** とマップファイル **tutorials.map** が作成されま

す。

コマンドライン用プログラムのリンク

プログラムをリンクするには、次のコマンドを入力します。

```
xlink tutor1 -f lnk740 -rt -x -l tutor1.map
```

-f オプションでは、XLINK コマンドファイル **lnk740** を指定します。また**-r** オプションを使用すると、デバッグ用のコードを生成することができます。

-x はマップファイルを生成し、**-l** ***filename*** はファイル名を指定します。

リンクの結果、コードファイル **aout.d31** とマップファイル **tutor1.map** が作成されます。

共通用マップファイルの表示

マップファイルを調べ、セグメント定義とコードがどのように物理アドレスに置かれているかを確認します。マップファイルの主なポイントを次に示します。

```
################################################################################
#                                                                              #
#      IAR Universal Linker V4.51N/WIN                                         #
#                                                                              #
#           Link time     =  09/Feb/2000  17:08:05                            #
#           Target CPU    =  740                                              #
#           List file     =  tutorials.map                                    #
#           Output file 1 =  tutorials.d31                                    #
#                            Format: debug                                     #
#                            UBROF version 6.0.0                               #
#                            Using library modules for C-SPY (-rt)             #
#           Command line  =  tutor1.r31 -o Tutorials.d31 -rt -l Tutorials.map  #
#                            -xms -IC:\IAR\EW23\740\LIB\ -f                    #
#                            lnk740t.xcl (-c740                                #
#                            -Z(BIT)BITVARS=200                                #
#                            -Z(ZPAGE)ZPAGE,C_ARGZ,Z_UDATA,Z_IDATA=41-FF       #
#                            -Z(ZPAGE)EXPR_STACK+20                            #
#                            -Z(ZPAGE)INT_EXPR_STACK+0 -Z(ZPAGE)CSTACK+40      #
```

Command line
コマンドラインと同等

含まれている XCL ファイル
リンカコマンドファイルの内容も含む

（途中略）

```
#                                                                              #
#                        Copyright 1987-1999 IAR Systems. All rights reserved. #
################################################################################
```

```
                ****************************************
                *                                      *
                *           CROSS REFERENCE            *
                *                                      *
                ****************************************

        Program entry at :          8000  Relocatable, from module : CSTARTUP

                ***********************************

                                MODULE MAP
                *
                ***************************

  DEFINED ABSOLUTE ENTRIES
  PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD

  SEGMENTS IN THE MODULE
  ======================
C_ARGZ
  Relative segment, address: 0045 - 004A (6 bytes)
  Segment part 0.
    ***************                                  ******************

  FILE NAME : c:\iar\ew23\740\user\debug\obj\tutor1.r31
  PROGRAM MODULE, NAME : tutor1

  SEGMENTS IN THE MODULE
  ======================
CODE
  Relative segment, address: 80CF - 8114 (46 bytes)
          ENTRY                   ADDRESS         REF BY
          =====                   =======         ======
          do_foreground_process   80CF
              calls direct
          main                    80E3            Segment part 14 (CSTARTUP)
              calls direct
              C_ARGZ = 0047 ( 0002 , 0000 )
          LOCAL                   ADDRESS
          =====                   =======
          ?0013                   80D5
          ?0011                   80F4
          ?0012                   8109
          ?0014                   8112
```

Program entry
プログラムエントリポイントのアドレスを示す

Module map
プログラムの一部としてロードされたそれぞれのモジュールについての情報

Segments in the module
指定されたモジュール内のセグメント情報一覧

File name
モジュールがリンクされたファイル名

Module 種類と名称

Entries
セグメント内で宣言されたシンボル

```
          ?0010                    8114
    ---------------------------------------------------------------------------
CONST
  Relative segment, address: 8120 - 8120 (1 bytes)
  Segment part 4. ROOT.       Intra module refs:   do_foreground_process
          ENTRY                    ADDRESS         REF BY
          =====                    =======         ======
          con_char                 8120
    ---------------------------------------------------------------------------
Z_UDATA
  Relative segment, address: 004B - 004D (3 bytes)
  Segment part 10. ROOT.      Intra module refs:   do_foreground_process
          ENTRY                    ADDRESS         REF BY
          =====                    =======         ======
          call_count               004B
          my_char                  004D

    ***************************************************************************

  FILE NAME : c:\iar\ew23\740\lib\cl7400t.r31     次のファイル
  PROGRAM MODULE, NAME : CSTARTUP

  SEGMENTS IN THE MODULE
  ======================
CSTACK
  Relative segment, address: 006E
  Segment part 0. ROOT.       Intra module refs:   Segment part 14
    ---------------------------------------------------------------------------
EXPR_STACK
  Relative segment, address: 004E
  Segment part 1. ROOT.       Intra module refs:   Segment part 14
    ---------------------------------------------------------------------------
```

（途中略）

```
****************************************
*                                      *
*      SEGMENTS IN ADDRESS ORDER       *
*                                      *
****************************************
```

Segments in address order

プログラムを形成する全セグメントを
リンクされた順に一覧表示する

```
SEGMENT              SPACE   START ADDRESS    END ADDRESS      SIZE  TYPE  ALIGN
=======              =====   =============    ===========      ====  ====  =====
BITVARS                             0040.0                          dse    0
ZPAGE                            0041 - 0044                   4    rel    0
C_ARGZ                           0045 - 004A                   6    rel    0
Z_UDATA                          004B - 004D                   3    rel    0
EXPR_STACK                       004E - 006D                  20    rel    0
Z_IDATA                             004E                          rel    0
INT_EXPR_STACK                      006E                          rel    0
CSTACK                           006E - 00AD                  40    rel    0
C_ARGN                              1000                          dse    0
N_UDATA                             1000                          rel    0
N_IDATA                             1000                          rel    0
NPAGE                               1000                          dse    0
ECSTR                               1000                          rel    0
RF_STACK                            1000                          rel    0
RCODE                            8000 - 80CE                  CF    rel    0
N_CDATA                             80CF                          rel    0
C_ICALL                             80CF                          dse    0
C_RECFN                             80CF                          dse    0
Z_CDATA                             80CF                          rel    0
CSTR                                80CF                          dse    0
CCSTR                               80CF                          rel    0
CODE                             80CF - 811F                  51    rel    0
CONST                            8120 - 8121                   2    rel    0
C_FNT                               FF00                          dse    0
INTVEC                           FFE0 - FFFD                  1E    com    0

                ****************************************
                *                                      *
                *        END OF CROSS REFERENCE        *
                *                                      *
                ****************************************

   320 bytes of CODE memory
   109 bytes of DATA memory

Errors: none
```

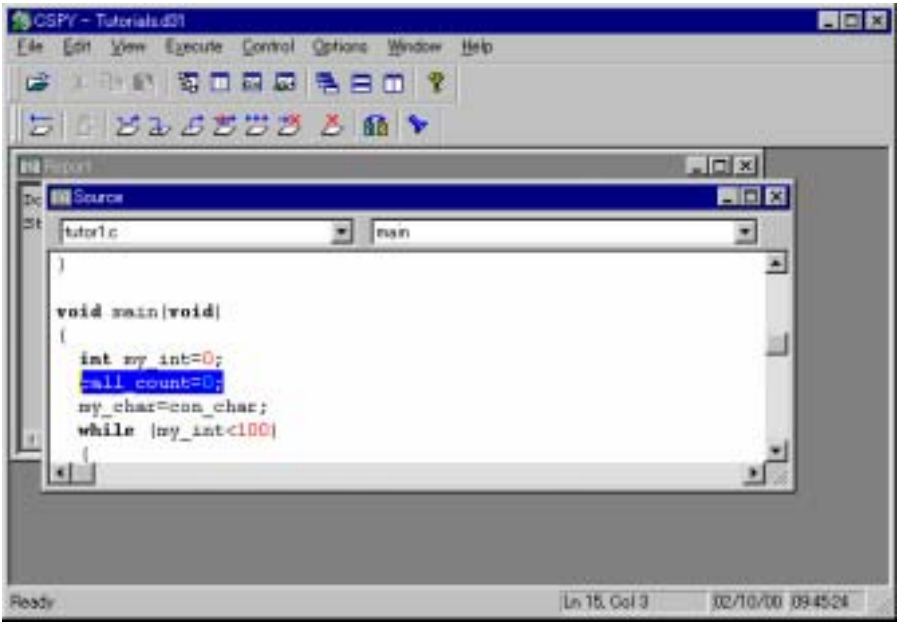
```
Warnings: none
```

プログラムの実行

組み込みワークベンチ用プログラムの実行

SW (C-SPY)デバッガを使ってプログラムを実行するには、まず **Project** メニューから **Debugger** を選択します。C-SPY ウィンドウが表示されます。

Execute メニューから **Step** を選択するか、ツールバーの **Step** ボタンをクリックし、ソース・ウィンドウにソースプログラムを表示します。



次にウォッチ・ウィンドウを使って、**call_count** 変数の値を監視します。**Window** メニューから **Watch** を選択し、ウォッチ・ウィンドウのツールバーにある **Watch** ボタンをクリックしてください。

次にウォッチ・ウィンドウ内でマウス右ボタンをクリックして表示されるポップアップ・メニューから **Add** を選択し、表示されるボックスで **call_count** と入力し、この変数をウォッチ・ウィンドウに追加します。

Execute メニューから **Step** を選択し、**do_foreground_process();**の行までプログラムを 1 ステップずつ実行していきます。ここで **call_count** 変数の値をウォッチ・ウィンドウで調べます。この変数は、初期化されたままインクレメントされていませんので、値は 0 になっているはずです。

現在の行を実行し、ループになっている次の行に進みます。**call_count** の値を再度調べると今度は 1 が表示され、この変数が **do_foreground_process** によってインクレメントされたことがわかります。

コンパイルとリンクオプションの変更

コンパイルやリンクで異なったオプションを利用すると、アウトプットにはあまり差が出ませんが、メモリの配置を変えることができます。いくつかのオプションは他のチュートリアルで紹介しますが、興味のある方は次の方法でコンパイル、リンクを実行してみてください。

- **Tiny** メモリモデルオプション（**-mt**）のかわりに、**Large** オプション（**-ml**）を使ってコンパイルします。マップファイルを調べることにより、変数がどこに配置されたかを確認します。
- **cl740l** にロードされるライブラリを修正することにより、ラージ・メモリモデルへの変更に合うようにリンクファイルを編集します。

チュートリアル 2

プログラムの拡張

次にプログラムを拡張し、740 ファミリーのマイクロプロセッサに内蔵されたシリアル I/O チャネルにアクセスしてみましょう。拡張したプログラムは、シリアルポート番号 0 からの入力を受け付け、バッファに文字を格納します。このシリアルプログラムでは、**#pragma** 疑似命令、およびヘッダファイルを使います。

以下にプログラムリストを示します。テキストエディタを使って入力し、**tutor2.c** として保存してください。プログラムを入力するかわりに、**tutor** サブディレクトリにある **tutor2.c** をコピーすることもできます。

```
#pragma language=extended  /* enable use of extended keywords */
#define Chip_37600         /* define what chip to use */
#include "io740.h"         /* include I/O register definitions */
#define ClockDivider (3)   /* gives 19200 baud with 20 MHz processor */
#define EnableRx (2 | 8)
#define InitMode ((1 << 6) | (1 << 1))    /* Select 8-bit characters,
                                             and division of clock by 16 */
#include <stdio.h>

/*********************************
 *           Variables           *
 *********************************/
char my_char;
```

```
int call_count = 0;
bit Rx_Set = USTS.2;


/*********************************
 *           Start of code          *
 *********************************/
char read_char(void)
{
   while (!Rx_Set);  /* wait for data */
   my_char = URBR1;  /* read data */
   return (my_char);
}

void do_foreground_process(void)
{
   call_count++;
   putchar(my_char);
}

void main(void)
{
       /* Initialize comms channel */
  UBRG = ClockDivider;          /* Set baud rate */
  UMOD = InitMode;
  UCON = EnableRx;   /* set mode */

       /* now loop forever, taking input when ready */
  while (1)
  {
    if (read_char())
        do_foreground_process();
  }
}
```

このプログラムの第 1 行目は、次のようになっています。

```
#pragma language=extended       /* enable use of extended keywords */
```

デフォルトでは拡張キーワードが利用できないことになっていますので、利用したい場合は必ず事前にこの疑似命令を記述する必要があります。**#pragma** 疑似命令の詳細は、「**#pragma** 疑似命令のリファレンス」を参照してください。

2 行目のコードを見てください。

```
#define Chip_37600
```

```
#include “io740.h”                /* include definitions for IO registers */
```

io740.h ファイルには、740 ファミリーのマイクロプロセッサで使用される全 I/O レジスタの定義が含まれています。

以下の行は、シリアルポート入力を設定します。

```
      /* Initialize comms channel */
  UBRG = ClockDivider;          /* Set baud rate */
  UMOD = InitMode;
  UCON = EnableRx;    /* set mode */
```

ClockDriver で設定されているビットは、**UBRG** コントロール・レジスタで設定される場合には、ボーレートを 19200 にセットします。

以下の行は、シリアル入力をテストします。

```
char read_char(void)
{
   while (!Rx_Set);  /* wait for data */
   my_char = URBR1;  /* read data */
   return (my_char);
}
```

Rx_Set は、インクルードされるヘッダファイルで定義されている I/O レジスタの一つである **USTS** のビット 2 をセットします。

シリアルプログラムのコンパイルとリンク

組み込みワークベンチ用コンパイルとリンク

tutor2.c ファイルを含む新規プロジェクトを作成します。

Project メニューから **Files...**を選択し、**Project Files** ダイアログボックスを使用して元の **tutor1.c** ファイルを **Tutorials** プロジェクトから取り除き、代わりに **tutor2.c** を追加します。

そして **Project** メニューから **Make** を選択し、チュートリアル・ファイルのコンパイルとリンクを実行します。

コマンドライン用コンパイルとリンク

タイニイ・メモリモデルおよび標準リンクファイルを使って、次のようにプログラムのコンパイルとリンクを実行します。

```
icc740 tutor2 –v0 -mt -r -L -e -q –I\iar\ewxx\740\inc\
xlink tutor2 -f lnk740 -r
```

シリアルプログラムの実行

組み込みワークベンチ用プログラムの実行

このプログラムを正常に実行するには、特別なハードウェア環境が必要ですが、SW を使ってコードを調べることができます。前述のように、**Project** メニューから **Debugger** を選択して、

プログラムを実行します。

SW の **View** メニューから **Toggle Source/Disassembly** を選択し、アセンブラレベルに切り替えます。**Execute** メニューから **Step** を選択し、プロンプトが出なくなるまで繰返します。プログラムはここでは、**while** ループで外部入力を待っています。

Execute メニューから **Stop** を選択するか、ツールバーの **Stop** ボタンをクリックします。 

次に **Window** メニューから **Memory** を選択してメモリ・ウィンドウを開き、アドレス 0x23 の位置をダブルクリックして表示されるボックスで 0x23 の値を **FF** に、同様に 0x26 の位置の値をたとえば **41**('A')に変更します。

そして再度 **Execute** メニューから **Step** を選択し、ステップを実行します。

シリアル入力ポート番号がトランスミッタに接続された実際のターゲットプロセッサでは、受取った文字に「バイト受信」フラグがセットされ、データレジスタに転送されます。上記のステップを実行することによって、Ready ビットが 1 になります。

次に **Step** を数回選択し、コードおよびシリアル・レジスタから読み込んだキャラクタ転送を端末 I/0 ウィンドウで確認します。

C コードではなくアセンブラ・コードがステップ・スルーされます。これはコンパイラがループを全てアセンブラ･コードに置換えたからです。独自のテストプログラムを設計する場合は、このようなきっちりしたループを避けるようにしてください。

最後に、**File** メニューから **Exit** を選択し、SW を終了します。

チュートリアル 3

割り込みハンドラの追加

今度は、最初のチュートリアルプログラムに割り込みハンドラを追加してみましょう。740 C コンパイラでは、**interrupt** キーワードを利用して割り込みハンドラを C で直接記述することができます。今回処理する割り込みは、BRK 割り込みです。

次に、割り込みプログラムのコーディングを示します。このプログラムは、**tutor3.c** という名前でサンプルチュートリアルに入っています。

```
#pragma language=extended    /* enable use of extended keywords */
#include <stdio.h>
#include "intr740.h"         /* include intrinsics */

/**********************************
 *           Variables            *
 **********************************/
```

```
char my_char = '*';
int call_count = 0;


/*********************************
 *           Start of code          *
 *********************************/

void interrupt [0x1E] brk_interrupt(void)
{
   putchar('I');
   my_char='.';
}

void execute_brk(void)
{
  break_instruction();   /* Use intrinsic function */
}

void do_foreground_process(void)
{
   call_count++;
   putchar(my_char);
}

void main(void)
{
   while (1)
   {
      do_foreground_process();
      if (my_char=='i') execute_brk();
   }
}
```

インライン関数 **brk_interrupt** を定義するには、インライン・インクルードファイルがなければなりません。

プロセッサが **BRK** 命令を実行したときはつねに、この割り込み関数が呼出されます。プログラムでは、イベントの識別を容易にするため、文字を出力します。診断出力チャネルを持つ実際のターゲットシステムで作業している場合は、何らかの診断情報が出力された後、プログラムが終了するでしょう。

interrupt キーワードについては、「拡張キーワードのリファレンス」を参照してください。

プログラムのコンパイルとリンク

組み込みワークベンチ用コンパイルとリンク

前述の例に習って、**tutor3.c** を **Tutorials** プロジェクトに追加します。

次に **Project** メニューから **Make** を選択し、プログラムのコンパイルとリンクを実行します。

コマンドライン用コンパイルとリンク

前述の例と同様に、次のコマンドでプログラムのコンパイルとリンクを実行します。

```
icc740 tutor3 –v0 –mt -r -L -q
xlink tutor3 -f lnk740 -r
```

割り込みプログラムの表示

組み込みワークベンチ用プログラムの表示

このプログラムを正常に実行するにはハードウェアが必要ですが、SW でコードを調べることができます。前回と同様、**Project** メニューから **Debugger** を選択して、SW を起動します。そして変数 **my_char** をメモリ・ウィンドウに追加し、このウィンドウ上でこれを右クリックして表示されるポップアップ・メニューで **Properties...**を選択します。そして **Symbol Properties** ダイアログボックスの **Display Format** を **Char**、**Value** を **i** にします。**OK** をクリックしてこのダイアログボックスを閉じ、**Execute** メニューから **Step Into** を **brk_interrupt** 関数に入るまで入力します。

第 2 章　コンパイラ・オプションの要約

この章では、組み込みワークベンチまたはコマンドラインからの、C コンパイラ・オプションの設定方法を説明します。

オプションは､組み込みワークベンチにおける **ICC740** または **General** のページに対応して、次のように分けられます。

Code generation　　#undef

Debug　　Include

#define　　Target

List

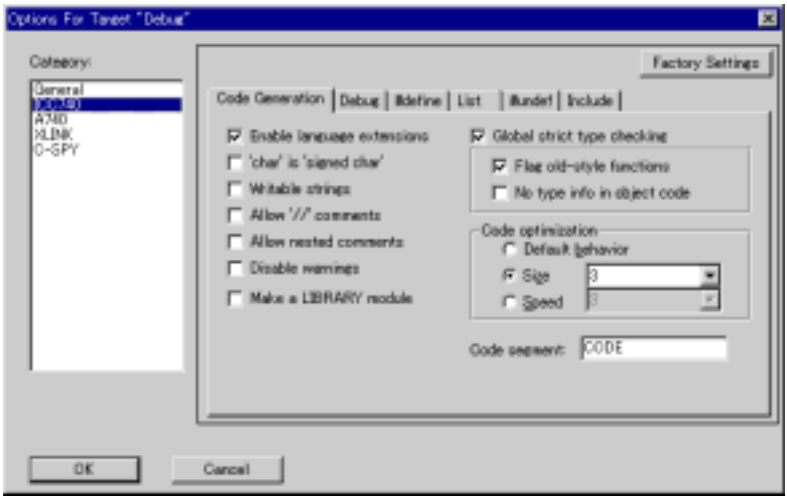
コマンドライン版でのみ利用できるオプションについての「Command line」のセクションもあります。

それぞれのオプションの完全な記述については、以下「C コンパイラ・オプション・リファレンス」の章を参照してください。

コンパイラ・オプションの設定

組み込みワークベンチ用コンパイラ・オプションの設定

組み込みワークベンチで C コンパイラ・オプションを設定するには **Project** メニューより **Options...**を選び、**Category** リストから **ICC740** を選んでコンパイラ・オプション・ページを表示させてください。



次に、参照または変更したいオプションの分類に対応するタブをクリックしてください。

コマンドライン用コンパイラ・オプションの設定

コンパイラ・オプションを使用するには、オプションをコマンドラインの **icc740** コマンドの後、ソース・ファイル名の前か後に入れます。例えば、ソース・ファイル **prog** をコンパイルする場合、デフォルトのリスト・ファイル名（**prog.lst**）にリストを生成するには、次のようにします。

```
icc740 prog -L
```

コンパイラ・オプションの中には、オプション文字の後に分離空白をつけて入力されたファイル名を受け付けるものもあります。例えば、**list.lst** ファイルにリストを生成するには、次のようにします。

```
icc740 prog -l list.lst
```

また、オプションの中には、ファイル名ではない文字列を受け付けるものもあります。ファイル名でない文字列は、オプション文字の後に空白をつけないで入れます。例えば、サブディレクトリ list にある省略時のファイル名にリストを生成するには、次のようにします。

```
icc740 prog -Llist
```

通常、コマンドラインのオプションは相互に、またソース・ファイル名と関連がありますが、オプションの順序は重要ではありません。例外として、**-l** オプションを 2 つ以上使用する場合の順序は重要です。

QCC740 環境変数を用いてのオプションの指定

コンパイラ・オプションは、**QCC740** 環境変数でも指定することができます。コンパイラは、この変数の値を自動的に各コマンドラインに付加します。したがって、この環境変数は、コンパイルごとに要求されるオプションを指定する場合に便利な方法となります。

たとえば、**autoexec.bat** ファイルに次の行を含ませると、つねにリストをファイル **temp.lst** に生成します。

```
set QCC740=-l temp.lst
```

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。各オプションの詳細説明については、第 3 章「コンパイラ・オプションのリファレンス」をオプションの分類名で参照してください。

オプション	内容	分類項目
-A*prefix*	接頭語付きファイル名でアセンブラソースを生成	List
-a *file*name	ファイル名で指定したアセンブラソースを生成	List
-b	オブジェクトをライブラリ・モジュールにする。	Command line
-C	ネスト・コメントを使用可能にする。	Code generation
-c	char 型を singed char にする。	Code generation
-D*symb[xx]*	シンボルを定義する。	#define
-e	ターゲット依存拡張子を使用可能にする。	Code generation
-F	新しいページに各関数をリストする。	List
-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。	Command line
-G	標準入力をソースとして開く。	Command line
-g[O][A]	グローバル型チェックを可能にする。	Code generation
-H*name*	オブジェクト・モジュール名を設定する。	Command line
-I*prefix*	**#include** 探索接頭語を追加する。	Include
-i	**#include** ファイルをリストする。	List
-K	C++コメントを使用可能にする。	Code generation
-L[*prefix*]	接頭語つきソース名のリストを生成する。	List
-l *filename*	ファイル名指定のリストを生成する。	List
-m[tl]	メモリ・モデルを選択する。	Target
-N*prefix*	接頭語付きのプリプロセッサ出力ファイル。	List
-n *filename*	プリプロセッサ出力ファイル名を指定。	List

オプション	内容	分類項目
-O*prefix*	オブジェクト・ファイル名の接頭語を設定する。	Command line
-o *filename*	オブジェクト・ファイル名を設定する。	Command line
-P	PROM 化可能なコードを生成する。	Command line
-p*lines*	リストをページに書式化する。	List
-q	リストにニモニックを入れる。	List
-R*name*	コード・セグメント名を設定する。	Code Generation
-r[012inre]	デバッグ情報を生成する。	Debug
-S	コンパイラのサイレント動作を設定する。	Command line
-s[0-9]	実行速度に対する最適化	Code Generation
-T	使用可能な行だけをリストする。	List
-t*n*	タブ間隔を設定する。	List
-U*symb*	シンボルを未定義にする。	#undef
-v*n*	プロセッサ・グループを選択する。	Target
-w	警告メッセージを表示しないようにする。	Code generation
-X	C 宣言を記述する。	List
-x[D][F][T][2]	相互参照リストを生成する。	List
-y	文字列を変数として初期化する。	Code generation
-z[0-9]	コードサイズによる最適化。	Code Generation

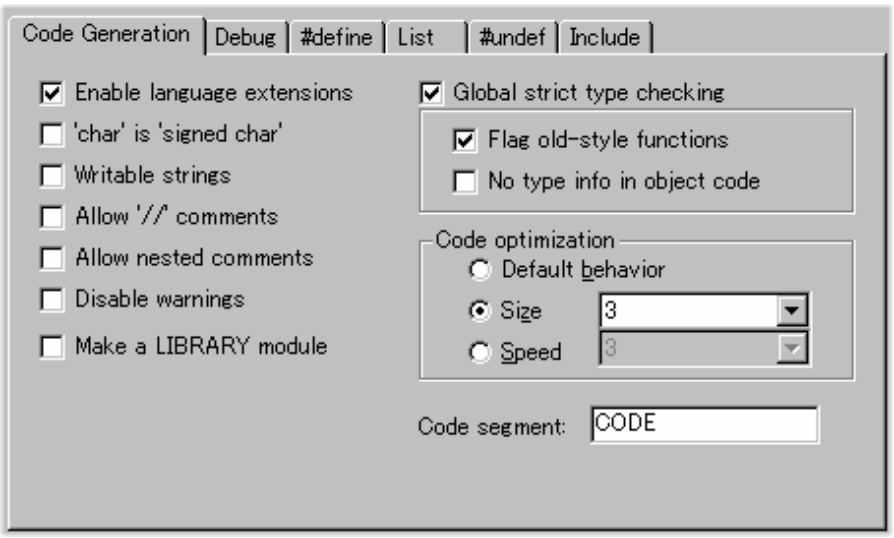
第 3 章　コンパイラ・オプションのリファレンス

この章では、機能分類項目に分けたコンパイラ・オプションをそれぞれ詳細に説明します。

Code Generation

Code generation オプションは、ソース・プログラムを解釈し、オブジェクト・コードを生成します。

組み込みワークベンチ用



Enable language extensions (-e)

構文： -e

ターゲット依存拡張機能をＣ言語で使用可能にします。
通常、言語拡張機能は互換性を確保するために認識できないようにされます。ソースで言語拡張機能を使用している場合は、このオプションによりそれらの機能を使用可能にしてください。
言語拡張機能の詳細については、「言語拡張機能」を参照してください。

‘char’ is ‘signed char’ (-c)

構文： -c

char 型を **signed char** と等価にします。
通常、コンパイラは **char** 型を **unsigned char** と解釈します。そこで例えば、別のコンパイラ

との互換性を考えてコンパイラに **char** 型を **signed char** として解釈させるには、**-c** オプションを使用します。
注記：ランタイム・ライブラリはこの **'char' is 'signed char' (-c)**オプションなしでコンパイルされますので、プログラムに**-c** オプションを使用し、**Global strict type check (-g)**や **Generate debug information (-r)**オプションで型チェックを可能にすると、リンカが型不一致の警告を発することがあります。

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。

Allow '//' comments (-K)

構文： `-K`

C++形式のコメントを使用可能にします。すなわち、コメントは// で導入され、行の終りまで延長できます。

通常、コンパイラは互換性を維持するために、C++形式のコメントを受け付けません。ソースに C++形式のコメントがある場合は、 **Allow '//' comments (-K)** オプションを使用して、それらを受け付けます。

Allow nested comments (-C)

構文： `-C`

ネスト・コメントを可能にします。
通常、コンパイラはネストされたコメントを誤りとして処理しますので、ネスト・コメントを発見すると、コメントのクローズ不良によるなどの警告を発行します。したがって、例えば、コメントを含むコードのコメントアウト部にネスト・コメントを使用したい場合などは、**Allow nested comments (-C)**オプションを使用してこの警告を禁止します。

Disable warnings (-w)

構文： `-w`

コンパイラの警告メッセージを表示しないようにします。

通常、コンパイラは標準警告メッセージ、および **Global strict type checking (-g)** で使用可能にされた追加警告メッセージを発行します。これらの警告メッセージをすべて表示しないようにするには、**Disable warnings (-w)**オプションを使用します。

Make a LIBRARY module (-b)

構文： `-b`

オブジェクトを通常のプログラム・モジュールではなくライブラリ・モジュールにします。

通常、コンパイラは XLINK でいつでもリンクされるプログラム・モジュールを生成します。XLIB を使用したライブラリへの組込みのためにライブラリ・モジュールが必要な場合は、**Make a LIBRARY module (-b)** オプションを使用します。

Global strict type checking (-g)

構文： `-g[A][0]`

ソース全体の型情報のチェックを可能にします。

ソースにはプログラミングの誤りを示す状態等級があります。通常、コンパイラやリンカは互換性を維持するためにこれを無視します。コンパイラやリンカに、このような状態を検出するたびに警告メッセージを発行させるようにするには、**Global strict type checking (-g)** オプションを使用します。

Flag old-style functions (-gA)

構文： `-gA`

通常、**Global strict type checking (-g)** オプションは、旧形式 K&R 関数の警告を行いません。これを可能にするには、**Flag old-style functions (-gA)**を使用します。

No type info in object code (-g0)

構文： `-g0`

通常、**Global strict type checking (-g)** オプションはオブジェクト・モジュールに型情報を含み、モジュールのサイズとリンク時間を増加させます。これによって、リンカは型チェックの警告を発行することができます。リンカの型チェック警告を禁止にするが、この情報を排除しサイズやリンク時間の増加を防止するには、**No type info in object code (-g0)** オプションを使用します。

複数のモジュールをリンクする場合、型情報なしに、すなわち**-g** オプションを付けずに、あるいは 0 修飾子を付けて **-g** オプションを入力してコンパイルされたモジュールのオブジェクトは、型なしと見なされることに注意してください。したがって、対応する宣言をもつモジュールが型情報付きでコンパイルされていても、型情報なしでコンパイルされたモジュールからの宣言に対する型不一致の警告は決して発行されません。

Global stricts type checking (-g) オプションでチェックされる状態は、次のとおりです。

- 未宣言の関数に対する**呼出し**
- 未宣言の K&R 式パラメータ
- 非 **void** 関数の戻り値の抜け
- 未参照のローカル・パラメータや仮パラメータ、あるいは **goto** ラベル
- 到達不能なコード
- K&R 関数に対する不一致パラメータや可変パラメータ
- 未知のシンボルの**#undef**
- 有効ではあるが、あいまいな初期化演算子
- 範囲を超えた定数配列の索引付け

例

次の例では、これらのエラーの型を 1 つ 1 つ説明します。

未宣言の関数に対する呼出し

```
void my_fun(void) { }
int main(void)
{
   my_func( );  /* my_fun のスペルが間違っていると、未宣言関数の警告が発せられる */
   return 0;
}
```

未宣言の K&R 式パラメータ

```
int my_fun(parameter)            /* パラメータの型が宣言されていない。*/
{
   return parameter+1;
}
```

非ボイド関数の戻り値の抜け

```
int my_fun(void)
{
   /* ... 関数本体 ... */
}
```

未参照のローカル・パラメータや仮パラメータ

```
void my_fun(int parameter)      /* 未参照の仮パラメータ */
{
   int localvar; /* 未参照のローカル・パラメータ */
   /* どちらの変数も参照しないで終了 */
}
```

未参照の goto ラベル

```
int main(void)
{
   /* ... 関数本体 ... */
   exit: /* 未参照のラベル */
   return 0;
}
```

到達不能なコード

```
#include <stdio.h>
int main (void)
{
  goto exit;
  puts (“This code is unreachable”);
  exit:
  return 0;
}
```

K&R 関数に対する不一致パラメータや可変パラメータ

```
int my_fun (len,str)
int len;
char *str;
{
   str[0]= ‘ a ’ ;
  return len;
}
char buffer[99];
int main (void)
{
  my-fun (buffer,99);/* パラメータの順序の間違い */
  my-fun (99);    /* パラメータの抜け */
  return  0;
}
```

未確認のシンボルの#undef

```
#define my_macro 99
/* 名前のスペルが間違っていると、シンボルが未確認である旨の警告が発せられる。*/
#undef my_macor
int main (void)
{
```

```
  return 0;
}
```

有効ではあるが、あいまいな初期化演算子

```
typedef struct t1 {int f1; int f2;} type 1;
typedef struct t2 {int f3; type1 f4; type1 f5;} type2;
typedef struct t3 {int f6; type2 f7; int f8;} type3;
type3 example = {99, {42,1,2}, 37};
```

範囲を超えた定数配列の索引付け

```
char buffer[99] ;
int main(void)
{
  buffer[500] = ‘a’;          /* 範囲を超えた定数の索引付け */
  return 0;
}
```

Code optimization (-s, -z)

サイズの最適化 (Size)

構文： -z[0-9]

コンパイラに、コードを最適化して最小サイズを得るようにさせます。

通常、コンパイラはレベル 3（下表参照）で最小サイズのための最適化を図ります。**-z** オプションを使用して、最小サイズのための最適化レベルを以下のように変更できます。

修飾子	レベル
0	最適化なし
1－3	完全にデバッグ可能
4－6	一部の構成体はデバッグ不能
7－9	完全に最適化

-z オプションは、**-s** オプションと同時には使用できません。

実行速度の最適化 (Speed)

構文： -s[0-9]

コンパイラに、コードを最適化させて最高実行速度を得るようにさせます。

通常、コンパイラはレベル 3（下表参照）で最大実行速度のための最適化を図ります。**-s** オプションを使用して、最大実行速度のための最適化レベルを以下のように変更できます。

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。

Code segment (-R)

構文： **-R***name*

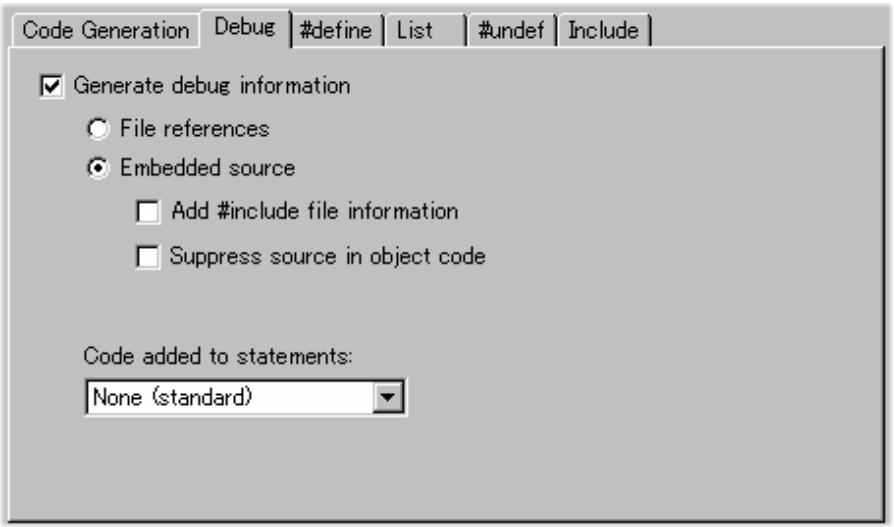
コード・セグメント名を設定します。

通常、コンパイラは、**CODE** と名前が付けられたセグメントに実行可能コードを入れます。あるソースファイルのコードを特別なアドレスを指定することができるようにするには、**-R** オプションを使用して特殊なコード・セグメント名(リンカ・コマンド・ファイルは、このセグメント名を使用しての固定アドレスに割り当てることのできます)を指定します。

Debug

Debug オプションは、オブジェクト・コードに含まれるデバッグ情報のレベルを決定します。

組み込みワークベンチ用



Generate debug information (-r)

構文： `-r[012][i][n][r][e]`

コンパイラに、SW(C-SPY)やその他のデバッガで要求される追加情報をオブジェクト・モジュールに入れるようにさせます。

通常コンパイラはコードの効率化を図るためにデバッギング情報を含みません。SW でコードのデバッギングを可能にするには、修飾子を付けないで **Generate debug information (-r)** オプションを使用し、オブジェクト・コード内にソースファイル参照を入れるようにします。完全なソースファイル情報をオブジェクト・コード内に含めるには、**e** 修飾子を使用します。

他のデバッガでコードをデバッギング可能にするには、次のように 1 つ以上の修飾子を入れます。

修飾子	組み込みワークベンチ	内容
i	Add #include file information	**#include** ファイル情報を含む。
n	Suppress source in object code	C ソース行の組込みを抑制する。
0, 1, 2	Code added to statements	ステートメント追加コード。
r		デバッグ可能なレジスタ操作。
e	Embedded source	ソースファイルのリファレンスを生成する。

通常、**Generate debug information (-r)** オプションは、**#include** ファイル・デバッギング情報を含みません。それは、一般的にこれへの関心が薄いこと、C-SPY 以外のデバッガは **#include** ファイル内部のデバッギングをサポートしていないことによります。**#include** ファイル内部のデバッグを行いたい場合、例えば**#include** ファイルがより一般的な関数宣言ではなく、関数定義を格納しているなどの場合は、**Add #include file information (-ri)** を使用します。副作用は、ソース行レコードが、C-SPY 以外の一部のデバッガでソース行の表示に影響を与えるグローバル（=合計）行カウントを含んでいることです。

通常、**Generate debug information (-r)**オプションはデフォルトで **File reference** が選択されていて、外部のソースファイルを参照し、オブジェクト・ファイルに C プログラムのソースを含みません。**Embedded source (e)**を選択するとオブジェクトに完全なソース情報が含まれます。これを抑制させて、オブジェクト・ファイルのサイズを削減したい場合は、**Suppress source in object code (n)**を使用します。

通常コンパイラは、ローカル変数をレジスタに置こうとします。しかしデバッグの目的のためには、**(r)** 修飾子を使用してこの最適化を阻止すると有益な場合があります。

ICC コンパイラの使用法に関する特定の情報を含んでいない他の多くのデバッガについては、**-rn** オプションを使用してください。

n 修飾子	意味
0,1,2	異なったデバッガ・ハードウエアをサポートします。ソースコード・デバッガについては、この情報は適当なデバッガ・マニュアル中で指定されているべきです。C ソース行表示をサポートしていないデバッガについては、デフォルト(0)で十分です。

#define

#define オプションは、C コンパイラで用いられるシンボルを定義します。

組み込みワークベンチ用

シンボルの定義（-D）

構文： -D*symb*[=*xx*]

シンボルを名前 ***symb*** と値 ***xx*** で定義します。値が指定されていない場合は、1 が使用されます。

Defined symbols (-D) は、ソースファイルの先頭で**#define** 文と同じ効果をもちます。

-D*symb* は、**#define *symb*** に相当します。

Defined symbols (-D) オプションは、コマンド行でより便宜よく値や選択項目を指定する場合に有用です。そうでなければ、これらはソースファイルで指定することになります。例えば、シンボル **testver** が定義されているかどうかに応じて検査バージョンか生産バージョンかのプログラムを生成するように、ソースをアレンジすることができます。このためには、次のようなインクルード部を使用します。

```
#ifdef   testver
         ...                    ; テスト用余分のコード
#endif
```

その後、次のようにコマンド行で必要なバージョンを選択します。

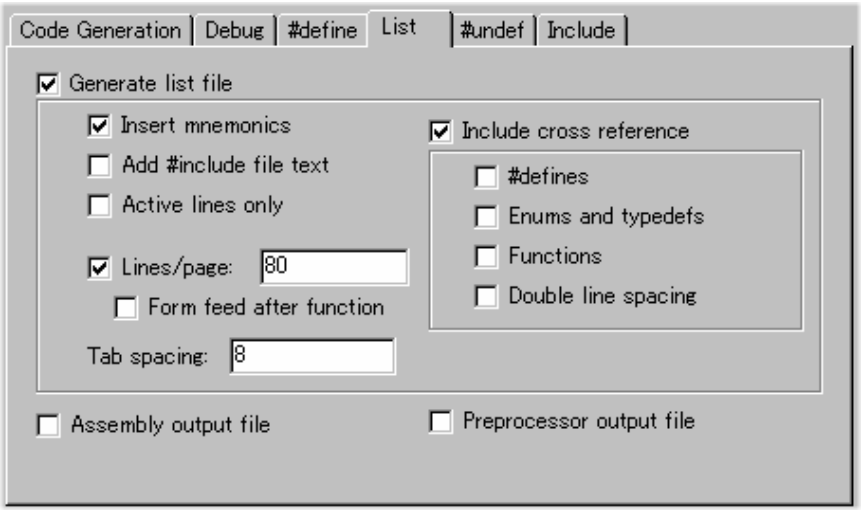
生産バージョン：**icc740 prog**

テストバージョン：**icc740 prog -Dtestver**

List

List オプションは、リストを生成するかどうかと、リストに含まれる情報を決定します。

組み込みワークベンチ用



リストファイル

名前付きファイルへのリストの生成（-l）

構文： **-l filename**

デフォルトの拡張子**.lst** が付いた名前付きファイルにリストを生成します。

通常、コンパイラはリストを生成しません。名前付きファイルにリストを生成するには、**-l** オプションを使用します。例えば **list.lst** ファイルにリストを生成するには、次のようにします。

icc740 prog -l list

一般的に、特定のファイル名を指定する必要はありません。このような場合は、代わりに**-L** オプションを使用します。

このオプションは、**-L** オプションと同時に使用することはできません。

定義済みソース名へのリストの生成（-L）

構文： **-L[prefix]**

ソースと同じ名前をもつが、拡張子**.lst** が付いたファイルにリストを生成します。接頭語がある場合は、引数で付けます。

通常、コンパイラはリストを生成しません。単にリストを生成するには、接頭語を付けないで **-L** オプションを使用します。例えば、**prog.lst** ファイルにリストを生成するには、次のようにします。

```
icc740 prog -L
```

別のディレクトリにリストを生成するには、**-L** オプションを使用し、その後にディレクトリ名を続けます。例えばディレクトリ**\list** の対応ファイル名にリストを生成するには、次のようにします。

```
icc740 prog -Llist\
```

このコマンドは、デフォルトの **prog.lst** ではなく**\list\prog.lst** にファイルを生成します。

-L は、**-l** と同時に使用することができません。

Insert mnemonics (-q)

構文： **-q**

リストに、生成されたアセンブリ行を入れます。

通常、コンパイラはリストに生成アセンブリ行を入れません。アセンブリ行を含んで、例えば、特定の文で生成されたコードの効率をチェックしたい場合は、**Insert mnemonics (-q)** オプションを使用します。

-a、-A、-l および **-L** オプションも参照してください。

Add #include file text (-i)

構文： **-i**

リストに**#include** ファイルを含むようにさせます。

通常、**#include** ファイルはヘッダー情報しか格納しておらず、リストのスペースを無駄にしますので、一般的にはリストは**#include** ファイルを含みません。例えば、**#include** ファイルに関数定義や処理済み行が含まれているためにこのファイルを含むようしたい場合は、**Add #include file text (-i)** オプションをいれます。

Active lines only (-T)

構文： `-T`

コンパイラに、使用可能なソース行のみをリストさせます。

通常、コンパイラは全ソース行をリストします。偽の**#if** 構造体にある行などの使用可能状態にない行を省いてリストのスペースを節約するには、**Active lines only (-T)** オプションを使用します。

Form feed after function (-F)

構文： `-F`

アセンブリ・リストにリストされた各関数の後にフォームフィードを生成します。

通常、リストは単に次の行から各関数を開始します。各関数が新しいページの最上行にくるようにするには、このオプションを入れます。

リストされていない関数(例えば、**#include** ファイル中の関数の場合)については、フォームフィードは決して生成されません。

Lines/page (-p)

構文： `-p`*lines*

リストをページに書式化し、10～150 行の範囲で 1 ページの行数を指定します。

通常、リストはページに書式化されません。各ページに書式送りを設けてリストをページに書式化するには、**Lines/page (-p)**オプションを使用します。例えば、1 ページ当り 50 行のプリンタの場合は、次のようにします。

```
icc740 prog -p50 ‑ L
```

Tab spacing (-t)

構文： -tn

1 タブ・ストップ当りの文字位置の数を n に設定します。n は 2～9 の範囲になければなりません。

通常、リストは 8 文字のタブ間隔で書式化されます。別のタブ間隔を指定したい場合は、Tab spacing (-t) オプションで設定します。

Include cross reference (-x)

構文： **-x[D][F][T][2]**

リストファイルに相互参照リストを含みます。

通常、コンパイラはリストにグローバル・シンボルを含みません。変数オブジェクト全部のリスト、および参照された全ての関数、**#define** 文、**enum** 文、そして **typedef** 文を作表の最後に含むには、修飾子を指定しないで **Cross reference (-x)** オプションを使用します。

追加情報を含むには、次の 1 つ以上の修飾子を追加します。

修飾子	内容（組み込みワークベンチ Include undefined のオプション）	
D	参照されていない**#define** シンボル	(#define)
F	参照されていない関数宣言	(Functions)
T	参照されていない **enum** 定数および **typedef**	(Enums and typedefs)
2	2 行スペーシングを行う	(Double line spacing)

Assembly output file

ファイル名で指定したアセンブラ・ソースを出力します（-a）

構文： **-a *file***

ファイル名 ***file*.s31** のアセンブラ・ソースを生成します。

デフォルトでは、コンパイラはアセンブラ・ソースを生成しません。このオプションは、ファイル名で指定したアセンブラ・ソースを生成します。

ファイル名は、リーフ名、そのリーフ名の前に任意に指定されるパス名、およびリーフ名の後に任意に指定される拡張子から構成されます。拡張子が指定されない場合は、ターゲットプロセッサに固有なアセンブラ・ソース拡張子が使用されます。

生成されるアセンブラ・ソースは、EW のアセンブラでアセンブルすることができます。

-q オプションも使用した場合は、C ソース行はコメントとしてアセンブラ・ソース・ファイルに含まれます。

このオプションは、**-A** オプションと同時に使用することはできません。

接頭語付きソース名に対するアセンブラ・ソースを出力します（-A）

構文： **-A*prefix***

***prefix source*.s31** にアセンブラ・ソースを生成します。

デフォルトでは、コンパイラはアセンブラ・ソースを生成しません。ソースファイルと同じリ

ーフ名をもつ、拡張子**.s31** の付いたファイル名のアセンブラ・ソースを生成するには、引数を付けないで-A オプションを使用します。例えば、

```
icc740 prog -A
```

は、**prog.s31** ァイルにアセンブラ・ソースを生成します。

別のディレクトリの同じ名前をもつファイルにアセンブラ・ソースを送信するには、ディレクトリを引数として付けた**-A** オプションを使用します。例えば、

```
icc740 prog -Aasm\
```

は、**asm\prog.s31** ファイルにアセンブリ・ソースを生成します。

-A オプションの後には、コンパイラがファイル名に追加する接頭語を続けることができます。これによって、ユーザは別のディレクトリにアセンブラ・ソースを生成することができます。

アセンブラ・ソースは、EW のアセンブラでアセンブルすることができます。

-q オプションも使用した場合、C ソース行はコメントとしてアセンブラ・ソース・ファイルに含まれます。

-A オプションは、**-a** オプションと同時に使用することはできません。

Preprocessor output file

ファイル名で指定したプリプロセッサ出力ファイルを出力します（-n）

構文： -n *file*

ファイル名 ***file*.i** のプリプロセッサ出力ファイルを生成します。

デフォルトでは、コンパイラはプリプロセッサ出力ファイルを生成しません。このオプションは、ファイル名で指定したプリプロセッサ出力ファイルを生成します。

ファイル名は、リーフ名、そのリーフ名の前に任意に指定されるパス名、およびリーフ名の後に任意に指定される拡張子から構成されます。拡張子が指定されない場合は、拡張子**.i** が使用されます。

このオプションは、**-N** オプションと同時に使用することはできません。

接頭語付きソース名に対するプリプロセッサを生成します（-N）

構文： -N*prefix*

プリプロセッサ出力を ***prefix source*.i** に行います。

デフォルトでは、コンパイラはプリプロセッサ出力を行いません。ソースファイルと同じリーフ名をもつ、拡張子**.i** の付いたファイル名のプリプロセッサ出力ファイルを生成するには、引数を付けないで-N オプションを使用します。例えば、

icc740 prog -N

は、**prog.i** ファイルにプリプロセッサ出力ファイルを生成します。

別のディレクトリの同じ名前をもつファイルにプリプロセッサ出力ファイルを送信するには、ディレクトリを引数として付けた**-N** オプションを使用します。例えば、

icc740 prog -Npreproc

は、**preproc\prog.i** ファイルにプリプロセッサ出力ファイルを生成します。

-N オプションは、**-n** オプションと同時に使用することはできません。

C 宣言の記述（-X）

構文： **-X**

ファイルの各 C 宣言の英文記述を表示します。

例えば、エラー・メッセージの調査に役立つように C 宣言の英文記述を得るには、-X オプションを使用します。

たとえば、

void (* signal(int _sig, void (* func) ( ))) npage (int);

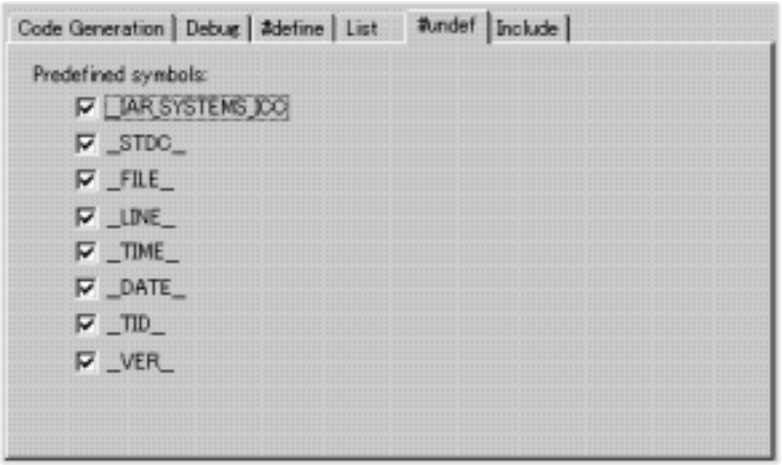
宣言は、次の記述を表示します。

```
Identifier: signal
storage class: extern
  prototyped function returning
    zpage – func pointer to
      prototyped function returning
              zpage - void
      and having following parameter(s):
              storage class: auto
              zpage - int
     and having following parameter(s):
         storage class: auto
         npage - int
         storage class: auto
         zpage – func pointer to
               function returning
                zpage – void
```

#undef

#undef オプションで、以前に宣言したシンボルを未定義にすることができます。

組み込みワークベンチ用



シンボルの未定義（-U）

構文： -U*symb*

名前つきシンボルの定義を取り外します。

通常、コンパイラはさまざまな定義済みシンボルを提供します。これらのシンボルの 1 つを取り外し、例えば同じ名前をもつユーザ自身のシンボルとの矛盾を回避するには、**Undefine symbol (-U)** オプションを使用します。

定義済みシンボルの一覧については、「定義済みシンボルのリファレンス」を参照してください。

組み込みワークベンチ用

シンボルを未定義にするには、**Predefined symbols** のリストを非選択にしてください。

コマンドライン用

例えば、シンボル__VER__を取り外すには、次のようにします。

```
icc740 prog -U__VER__
```

Include

Include オプションは、C コンパイラの include パスを決定します。

組み込みワークベンチ用

#INCLUDE 探索接頭語の追加（-I）

構文： **-I*prefix***

#include ファイル接頭語のリストに接頭語を追加します。

通常、コンパイラはソース・ディレクトリ（ファイル名が山カッコではなく引用符で囲まれている場合）、**C_INCLUDE** パス、そして最後にカレント・ディレクトリでのみインクルード・ファイルを探索します。**#include** ファイルを別のディレクトリに入れている場合は、**Include paths (-I)** オプションを使用してそのディレクトリをコンパイラに知らせます。例えば、次のようにします。

```
icc740 prog -I\mylib\
```

コンパイラは単にインクルード・ファイル名の先頭に**-I** 接頭語を追加するだけですので、必要に応じて最終バックスラッシュを含むことが大切です。

シングル・コマンド行で使用できる**-I** オプションの数に制限はありません。多数の**-I** オプションを使用する場合は、コマンド行がオペレーティング・システムのコマンド行制限を超えないようにするために、コマンド・ファイルを使用します。コマンド・ファイルについては、-f オプションを参照してください。

注記：コンパイラの**#include** ファイル探索手順の完全な説明は、次のとおりです。

コンパイラが

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

などの山カッコの中にインクルード・ファイル名を見つけると、次の探索シーケンスを実行します。

- 各 **-I*prefix*** で指定される接頭語が付いたファイル名
- **C_INCLUDE** 環境変数の各パスが接頭語として付いたファイル名
- ファイル名自体

コンパイラが、

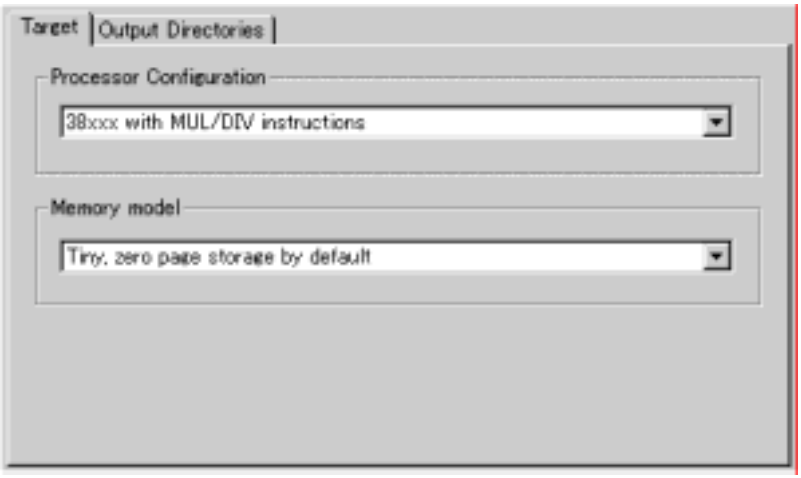
```
#include "vars.h"
```

などの 2 重引用符の中にインクルード・ファイル名を検出した場合は、ソース・ファイル・パスが接頭語として付いたファイル名を探索し、その後、山カッコで囲われたファイルのように順番に探索します。

Target

General カテゴリの **Target** オプションは、アセンブラと C コンパイラ用のプロセッサとメモリモデルを指定します。

組み込みワークベンチ用



Processor configuration (-v)

構文： **-v*n***

以下のプロセッサ・バージョンの一つを選択します。

オプション	コマンドライン
38XXX with MUL and DIV instructions	**-v0**
38XXX without MUL and DIV instructions	**-v1**
37XXX except 37600	**-v1**
37600	**-v2**

Memory model (-m)

構文： **-m[tl]**

生成されるコードのメモリモデルを以下のように選択します。

オプション	コマンドライン	説明
Tiny	**t**	デフォルトでゼロ・ページ・ストレージ
Large	**l**	デフォルトでゼロ・ページ・ストレージの外側

メモリモデルはゼロ・ページ・ストレージデフォルトで使用されるかどうかを決定します。

コマンドライン用

通常、コンパイラはタイニイ・メモリモデル用のコードを生成します。異なったメモリモデル用のコードを希望する場合、適当なオプションを使用してください。

例えば、ラージ・メモリモデル用の 37600 のコードを生成するには、以下のコマンドを指定します。

icc740 prog -ml -v2

Command Line

つぎの追加オプションがコマンドラインより指定できます。

-f *filename* コマンドラインを拡張する。

-G 標準入力をソースとして開く。

-H*name* オブジェクト・モジュール名を設定する。

-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。

-S コンパイラのサイレント動作を設定する。

コマンドライン用 ファイルからのコマンド行オプションの読み取り（-f）

構文： **-f** *file*

デフォルトの拡張子**.xcl** をもつ名前付きファイルからコマンド行オプションを読み取ります。

通常コンパイラは、コマンド行および **QCC740** 環境変数からのみコマンド・パラメータを受け付けます。長いコマンド行をより管理し易くし、オペレーティング・システムのコマンド行長さ制限を回避するには、-f オプションを使用して、コマンド・ファイルを指定します。この場合、コンパイラは、あたかもオプションの位置で入力されたかのようにコマンド行の項目を読み取ります。

コマンド・ファイルでは、改行文字は空白やタブ文字のように作用するので複数の行を使用できるという点を除けば、コマンド行の項目はあたかもコマンド行にあるかのように正確に書式化します。

例えば、コマンド行

```
icc740 prog -r -L -Dtestver "-Dusername=John Smith"  -Duserid=463760
```

を、

```
icc740 prog -r -L -Dtestver -fuserinfo
```

および

```
"-Dusername=John Smith"
-Duserid=463760
```

を格納した **userinfo.xcl** ファイルで置換することができます。

コマンドライン用標準入力をソースとして開く（-G）

構文： **-G**

ファイルからソースを読み取るのではなく、標準入力をソースとして開きます。

通常、コンパイラはコマンド行で命名されたファイルからソースを読み取ります。代わりに、コンパイラに標準入力（通常、キーボード）からソースを読み取らせたい場合は、-G オプションを使用し、ソースファイル名を省略します。

ソースファイル名は、**stdin.c** に設定されます。

コマンドライン用オブジェクト・モジュール名の設定（-H）

構文： **-H***name*

通常、オブジェクト・モジュールの内部名は、ディレクトリ名や拡張子をもたないソースファイルの名前です。オブジェクト・モジュール名を明示的に設定するには、-H オプションを使用します。例えば、次のようにします。

icc740 prog -Hmain

これは、いくつかのモジュールが同じファイル名をもっている場合に特に有用です。というのは、通常は結果のモジュール名が重複していると、リンカ・エラーが生じるからです。例としては、ソースファイルがプリプロセッサで生成された一時ファイルの場合がそうです。次の場合（%1 が、ソースファイル名を含んだオペレーティング・システムの変数となっています）は、リンカから重複名エラーが発行されます。

preproc %1.c temp.c ; **temp.c** を生成する前処理ソース

icc740 temp.c ; モジュール名は常に**"temp"**である。

これを回避するには、**-H** を使用してオリジナル名を得ます。

preproc %1.c temp.c ; **temp.c** を生成する前処理ソース

icc740 temp.c -H%1 ; オリジナル・ソース名をモジュール名として使用する。

コマンドライン用オブジェクト・ファイル名の接頭語の設定（-O）

構文： **-O*prefix***

オブジェクトのファイル名に、使用する接頭語を設定します。

通常（および**-o** オプションを使用していない限り）、オブジェクトは、ソースファイル名に対応する、拡張子**.r31** が付いたファイル名で保存されます。別のディレクトリにオブジェクトを保存するには、**-O** オプションを使用します。

例えば、**\obj** ディレクトリにオブジェクトを保存するには、次のコマンドを使用します。

icc740 prog -O\obj

-O オプションは、**-o** オプションと同時に使用することはできません。

コマンドライン用オブジェクト・ファイル名の設定（-o）

構文： **-o *filename***

オブジェクト・モジュールが保存されるファイル名を設定します。ファイル名は任意指定のパス名、必須のリーフ名、および任意指定の拡張子（デフォルトでは、**.r31**）から構成されます。

通常、コンパイラは次の名前のファイルにオブジェクト・コードを保存します。

- **-O** で指定された接頭語、
- ソースのリーフ名、および
- 拡張子 **.r31**

別のファイル名にオブジェクト・コードを保存するには、**-o** オプションを使用します。例えば、**obj.r31** ファイルにオブジェクト・コードを保存するには、次のコマンドを使用します。

icc740 prog -o obj

代わりに、別のディレクトリの対応ファイル名でオブジェクト・コードを保存したい場合は、**-O** オプションの使用がより便利です。

-o オプションは、**-O** オプションと同時に使用することはできません。

コマンドライン用 PROM 化可能なコードの生成（-P）

構文： **-P**

コンパイラに、読み取り専用メモリ（PROM）での実行に適したコードを生成させます。

このオプションは他の IAR コンパイラとの互換用に含まれていますが、740 C コンパイラではつねにアクティブになっています。

コマンドライン用コンパイラのサイレント動作の設定（-S）

構文： **-S**

コンパイラに、標準出力先（通常、画面）に不必要なメッセージを送信しないで動作するようにさせます。

通常、コンパイラはコピーライトメッセージや最終の統計値レポートを発行します。この出力を禁止するには、**-S** オプションを使用します。この設定を行っても、エラー・メッセージや警告メッセージの表示にはなんら影響ありません。

第 4 章　構成

この章では、さまざまな要件に応じて C コンパイラを構成する方法を解説します。

概要

740 マイクロプロセッサを使用したシステムは、その内部 ROM、RAM および外部 ROM、RAM の使用とスタックの要件がかなり異なっています。また再帰、大容量のライブラリ、あるいはタイム・クリティカル関数に対するシステムの必要性についても異なります。この章では、指定されたアプリケーションに対する 740 C コンパイラの構成方法を説明します。

メモリ・モデルやリンク・オプションは、次のものを指定します。

- ROM 領域：関数、定数、初期値に使用されます。
- RAM 領域：内部メモリ、外部メモリ、外部不揮発性メモリに使用されます。

環境や使用の各機能は、次のようなコンパイラ・パッケージの 1 つ以上の構成可能要素で処理されます。

機能	構成可能要素	参照ページ数
プロセッサ・グループ	XLINK コマンド・ファイル、ランタイム・ライブラリ・モジュール	54
メモリ・モデル	コンパイラ・オプション、XLINK オプション	55
記憶位置	XLINK コマンド・ファイル	56
不揮発性 RAM	XLINK コマンド・ファイル	56
スタック・サイズ	XLINK コマンド・ファイル	57
putchar 関数、getchar 関数	ランタイム・ライブラリ・モジュール	58
printf/scanf 機能	XLINK コマンド	60
ヒープ・サイズ	ヒープ・ライブラリ・モジュール	62
ハードウェアとメモリの初期化	CSTARTUP モジュール	62

以降の項では、これらの機能を 1 つ 1 つ説明します。多くの構成手順は標準ファイルの編集を伴いますので、開始前にオリジナルのコピーをとっておくことを勧めます。

ランタイム・ライブラリ

ライブラリ・ファイルは、システムの構成可能な機能を多く制御します。

メモリ・モデルやプロセッサ・タイプは、下表に示す組合せのすべてにおいても代替ランタイム・ライブラリを必要とします。

	プロセッサ・オプション		
メモリ・モデル	-v0	-v1	-v2
タイニー	**cl7400t.r31**	**cl40lt.r31**	**cl7402t.r31**
ラージ	**cl7400l.r31**	**cl7401l.r31**	**cl7402l.r31**

省略時には、ライブラリ・ファイルは c:\iar\lib ディレクトリにあります。

XLINK コマンド・ファイル

特定のプロジェクト用の XLINK コマンド・ファイルを生成するには、まず配布テンプレート c:\iar740\lnk740.xcl のコピーをとってください。その後、このファイルをファイルの中の説明に従って変更し、目的システムのメモリ・マップの詳細を指定します。

プロセッサ・グループ

740 ファミリのマイクロプロセッサには多数の可変部があり、740 C コンパイラはこれを 3 グループに分類します。

プロセッサ・グループの指定

プログラムは一度に 1 プロセッサ・グループしか使用することができませんので、全てのユーザ・モジュールと全てのライブラリ・モジュールで同じプロセッサ・グループを使用してください。

ユーザ・モジュールがコンパイルされたときにコンパイラにプロセッサ・グループを指定するには、次のコマンド行オプションの 1 つを使用します。

オプション	プロセッサ・グループ
-v0	MUL/DIV 命令をもつ 38XXX
-v1	MUL/DIV 命令をもたない 38XXX、あるいは 37600 以外の 37XXX
-v2	37600

例えば、myprog を 37600 で使用するためにコンパイルを行うには、次のコマンドを使用します。

```
icc740 myprog -v2
```

メモリ・モデル

740 マイクロプロセッサには、アドレス指定速度を増加させるために使用するアドレス指定モードがあります。740 C コンパイラは、2 つのメモリ・モデルをサポートすることによってこれらの機能を活用します。この選択を行うと、プログラム・コード／データの最大サイズに対する実行速度やコード／データのサイズに影響がでます。

メモリ・モデルの選択

タイニー・メモリ・モデルは、シングルチップ（あるいは制限付き外部メモリ）システム用に、またラージ・メモリ・モデルは外部メモリ付きシステム用に設計されています。タイニー・メモリ・モデルは省略時に全ての変数をゼロ・ページに置くのに対し、ラージ・メモリ・モデルはこれらをゼロ・ページの外に置きます。

メモリ・モデルの指定

プログラムは一度に 1 メモリ・モデルしか使用することができませんので、全てのユーザ・モジュールと全てのライブラリ・モジュールで同じメモリ・モデルを使用してください。

メモリ・モデルは、コンパイラと XLINK の両方に指定してください。

ユーザ・モジュールがコンパイルされたときにコンパイラにメモリ・モデルを指定するには、

次のコマンド行オプションの 1 つを使用します。

オプション	メモリ・モデル
-mt	タイニー・メモリ・モデル
-ml	ラージ・メモリ・モデル

例えば、ラージ・メモリ・モデルで myprog をコンパイルするには、次のコマンドを使用します。

```
icc740 myprog -ml
```

メモリ・モデル・オプションを何も含まない場合、コンパイラはラージ・メモリ・モデルを使用します。

XLINK にメモリ・モデルを指定するには、リンカ・コマンド・ファイルを編集します。

例えば、モジュール myprog（事前にラージ・メモリ・モデル用にコンパイルされたもの）をリンクするには、次のコマンドを使用してください。

```
xlink myprog -f lnk740
```

-f オプションは、コマンド・ファイル名を指定します。詳細については、「740 アセンブラ、リンカ、ライブラリアン・プログラム・ガイド」を参照してください。

記憶場所

XLINK には、ハードウェア環境の ROM と RAM のアドレス範囲指定が必要です。通常、この指定は XLINK コマンド・ファイル・テンプレートのコピーで行います。

メモリ・アドレス範囲の指定に関する詳細については、「740 アセンブラ、リンカ、ライブラリアン・プログラミング・ガイド」の第 9 章「XLINK リンカ」や XLINK コマンド・ファイル・テンプレートの内容を参照してください。

不揮発性 RAM

コンパイラは、不揮発性 RAM に存在しなければならない変数の宣言を no_init 型修飾子や #pragma メモリによってサポートします。コンパイラはこのような変数を個別セグメント NO_INIT にいれます。このセグメントは、ハードウェア環境の不揮発性 RAM のアドレス範囲に割り付けてください。ランタイム・システムは、これらの変数の初期化を行いません。

NO_INIT セグメントを不揮発性 RAM のアドレス範囲に割り当てるには、XLINK コマンド・ファイルの変更が必要です。指定アドレスへのセグメントの指定に関する詳細については、「740 アセンブラ、リンカ、ライブラリアン・プログラミング・ガイド」を参照してください。

スタック・サイズ

コンパイラは、各種のユーザ・プログラムの操作にスタックを使用しており、要求されるスタック・サイズはこれらの操作の詳細に大きく依存しています。指定されたスタック・サイズが小さすぎると、通常、スタックは変数記憶を上書きできるようになっています。この場合、プログラム不良となってしまいます。また、指定されたスタック・サイズが大きすぎると、RAM 容量を無駄にします。

コンパイラは、4 つの個別スタックを活用します。

- 式スタック
- 割込み式スタック
- プロセッサ・スタック
- 再帰的スタック

要求スタック・サイズの見積

スタックは、次のように使用されます。

式スタック

- 通常の処理中に式の一時的結果を保存します。X レジスタは、スタック・ロケーションの先頭を示します。

割込み式スタック

- 割込み処理中に式の一時的結果を保存します。X レジスタは、スタック・ロケーションの先頭を示します。

プロセッサ・スタック

- 関数呼出しの戻りアドレスを保存します。
- 割込み中にプロセッサ状態を保存します。
- 関数呼出し中に CPU レジスタとランタイム・システムを保存します。

再帰的スタック

- 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存します。

それぞれのケースで必要とされるスタック・サイズの合計は、最悪な場合を基準に上記の 1 つ 1 つに必要なサイズを合計して出した値です。

コードを検査することによって必要なスタック・サイズを見積るのは、簡単ではありません。このような場合、次の手順でスタックの使用量を検査することができます。

- スタートアップ時に既知の値でスタックを埋めます。
- C−SPY シミュレータの下でプログラムを実行します。
- スタック・メモリを検査して、使用量を確認します。

スタック・サイズの変更

省略時のスタック・サイズについては、式スタックは 32（0x20）バイトに、割込み式スタックは 4 バイトに、プロセッサ・スタックは 64（0x40）バイトに、そして再帰的スタックは 256（0x100）バイトに設定されます。

スタック・サイズの一部を変更するには、リンカ・コマンド・ファイルを編集し、オリジナル・サイズを希望するスタック・サイズで置換します。

入出力

PUTCHAR と GETCHAR

putchar 関数、getchar 関数は、C 言語が全てのキャラクタ・ベースの I/O を実行するための基本関数です。キャラクタ・ベースの I/O を利用可能にするには、ハードウェア環境で提供されるなんらかの機能を使用して、これらの 2 つの関数に定義を行ってください。

新しい I/O ルーチンを生成する場合の出発点は、c:＼iar＼icc740＼putchar.c と c:＼iar＼icc740＼getchar.c です。putchar のカスタマイズ・バージョンを生成するための方法は次のとおりです。

- putchar.c ソースに必要な追加を行い、同じ名前で保存します（あるいは、putchar.c をモデルとして使用し、独自のルーチンを生成します）。次のコードはメモリ・マップ I/O を使用して、LCD ディスプレイに書込みを行います。

```
#include <stdio.h>
int putchar(int outchar)
{
  unsigned char *LCD_IO;
  LCD_IO= (unsigned char *) 0x8000;
```

```
    * LC_IO=otchar;
    return(outchar);
}
```

◆ 適切なメモリ・モデルを使用して、変更した putchar をコンパイルします。例えば、プログラムがタイニー・メモリ・モデルを使用する場合は、次のコマンドでスモール・メモリ・モデル用に putchar.c をコンパイルします。

```
icc740 putchar -mt -z9
```

このコマンドは、putchar.r31 と命名された最適置換オブジェクト・モジュール・ファイルを生成します

◆ 適切なランタイム・ライブラリ・モジュールに新しい putchar モジュールを追加し、オリジナル・モジュールを置換します。例えば、標準のタイニー・メモリ・モデル・ライブラリに新しい putchar モジュールを追加するには、次のコマンドを使用します。

```
xlib
def-cpu 740
rep-mod putchar cl7400t
exit
```

これで、ライブラリ・モジュール cl7400t は、オリジナル・モジュールの代わりに変更した putchar をもちます。(putchar モジュールを上書きする前に、オリジナルの cl7400t.r31 ファイルを保存することを忘れないでください。)

XLINK を使用すると、変更モジュールをライブラリへインストールする前に -A オプションを使ってこのモジュールを検査することができます。.xcl リンク・ファイルに次の行を入れます。

```
-A putchar
cl7400t
```

これによって、cl7400t ライブラリにあるファイルの代わりに putchar.r3 のユーザ・バージョンがロードされます。詳細は、「740 アセンブラ、リンカ、ライブラリアン・プログラミング・ガイド」を参照してください。

getchar についても、同じ方法を使用します。

putchar は printf 関数の低レベル部としての役目を果たすことに注意してください。

PRINTF と SPRINTF

printf と sprintf 関数は、_formatted_write と呼ばれる共通フォーマッタを使用します。ANSI 規格バージョンの_formatted_write は非常に容量が大きく、多くのアプリケーションでは必要としない機能を提供しています。そこで、標準 C ライブラリでは、メモリの消費を抑えるために容量の小さな次の代替バージョンも用意しています。

-medium_write

_formatted_write との違いは、浮動小数点型数値がサポートされていないという点です。また%f, %g, %G, %e、あるいは%E 指定子を使用しようとすると、エラーが発生します。

```
FLOATS? wrong formatter installed!
```

_medium_write は、_formatted_write よりもかなり小さな容量になっています。

_small_write

_medium_write との違いは、int オブジェクトのための%%, %d, %o, %c, %s、%x 指定子のみサポートし、field width や precision 引数はサポートしていないという点です。_small_write のサイズは、_formatted_write のサイズの 10～15%です。

省略時のバージョンは、_small_write です。

書込みフォーマッタ・バージョンの選択

書込みフォーマッタの選択は、XLINK 制御ファイルで行われます。省略時の選択肢 _small_write は、次の行で実行されます。

```
-e_small_write=_formatted_write
```

完全な ANSI バージョンを選択するには、この行を取り外します。

_medium_write を選択するには、この行を次の行で置き換えます。

```
-e_medium_write=_formatted_write
```

縮小 PRINTF

多くのアプリケーションでは、sprintf は必要ありません。_small_formatter 付きの printf でさえも、消費されるメモリからみると、必要以上の機能を提供しています。あるいは、特定の書式化ニーズおよび／または非標準出力装置をサポートするには、特注出力ルーチンが必要になることがあります。

そのようなアプリケーションでは、全ての printf 関数（sprintf 以外）の非常に縮小されたバージョンを、ソース形式で intwri.c ファイルに配布しています。このファイルは、ユーザの要件に合わせて変更することができ、コンパイルしたモジュールは上記の putchar で説明した方法でオリジナル・ファイルの代わりにライブラリに挿入することができます。

SCANF と SSCANF

printf や sprintf 関数と同じように、scanf と sscanf 関数は _formatted_read と呼ばれる共通フォーマッタを使用します。ANSI 規格バージョンの_formatted_read は非常に容量が大きく、多くのアプリケーションでは必要としない機能を提供しています。そこで、標準 C ライブラリでは、メモリの消費を抑えるために容量の小さな代替バージョンも用意しています。

_medium_read

_formatted_read との違いは、浮動小数点数値がサポートされていないという点です。_medium_read は、_formatted_read よりもかなり容量が小さいです。

省略時のバージョンは、_medium_read です。

読み込みフォーマッタ・バージョンの選択

読み込みフォーマッタの選択は、XLINK 制御ファイルで行われます。省略時の選択肢 _medium_read は、次の行で実行されます。

```
-e_medium_read=_formatted_read
```

完全な ANSI バージョンを選択するには、この行を取り外します。

レジスタ I/O

プログラムは、SFR レジスタを使用して 740 I/O システムをアクセスすることがあります。

io740.h は、次のような 740 SFR レジスタを定義します。

- nsigned char に相当するもの
- 直接アドレス指定できるもの

単項&（アドレス）演算子を除く整数型に適用される演算子はすべて、SFR レジスタに使用できます。740 ファミリ・メンバー用の定義済み define 宣言を配布しています。詳細については、io740.h ファイルを参照してください。

例：

次の例は、（io740.h に定義された）ロケーション P1 の I/O ポートのビット 0 を操作します。

```
#define Chip_37600
#include <io740.h>          /* レジスタ・アドレスを定義する。*/
#define BIT0_SET 0X01
#define BIT0_CLR 0xFE
void func( )
{
   P1 = 4;                  /* I/0 P1 = 00000100 を設定する。*/
   P1 | BIT0_SET;           /* 1 ビットのみに影響を与える。*/
   if (P1 & BIT0_SET) printf("ON");        /* ポート全体を読み取り、ビット 1 を
                                           マスクする。*/
5  P1&=BIT0_CLR; /* ポートのビット 0 をクリアする。*/
```

ヒープ・サイズ

ライブラリ関数 malloc や calloc がプログラムで使用された場合、C コンパイラは、関数の割当が行われるメモリのヒープを生成します。省略時のヒープ・サイズは、2000 バイトです。

ヒープ・サイズを変更する方法は、c:\iar\etc\heap.c ファイルに説明しています。

変更したヒープ・モジュールは、.xcl リンク・ファイルに次の行を入れることによって検査できます。

```
-A heap
cl7400l
```

これは、cl7400l ライブラリにあるヒープ・モジュールの代わりにユーザ・バージョンの heap.r31 をロードします。

初期化

プロセッサのリセット時、CSTARTUP と呼ばれるランタイム・システム・ルーチンに実行が渡されます。通常、このルーチンは次の内容を実行します。

- 10 進モードのクリア
- T フラグのクリア
- 式スタック・ポインタの初期化

- スタック・ポインタの初期化
- C ファイル・レベルおよび静的変数の初期化
- ユーザ・プログラム関数 main の呼出し

ユーザ・プログラムが exit あるいは abort 処理にかかわらず終了した場合、CSTARTUP は制御を受け取り、確保する責任もあります。

CSTARTUP を変更して、例えば main へのエントリ前に特殊なハードウェアを初期化したり、変数の希望しない初期化を取り外すことができます。

CSTARTUP を変更するための全体的な手順は、次のとおりです。

- 次のファイルに省略値として提供されている CSTRATUP のアセンブラ・ソースに必要な変更を行います。

  c:＼iar＼icc740＼cstartup.s31

  それを同じ名前で保存します。

- 適切なメモリ・モデルを使用して、変更した putchar をアセンブルします。例えば、プログラムがスモール・メモリ・モデルを使用する場合は、次のコマンドでスモール・メモリ・モデルの CSTARTUP を再アセンブルします。

  a740 cstartup.s31

  このコマンドは、cstartup.r31 と命名された置換オブジェクト・モジュール・ファイルを生成します

- 適切なランタイム・ライブラリ・モジュールに新しい CSTARTUP モジュールを追加して、オリジナルを置換します。

例えば、標準メモリ・モデル・ライブラリに新しい CSTARTUP モジュールを追加するには、次のコマンドを使用します。

```
xlib
def-cpu 740
rep-mod CSTARTUP c174001
exit
```

これで、ライブラリ・モジュール c17400l はオリジナルの代わりに、変更された CSTARTUP をもちます。

XLINK を使用すると、変更モジュールをライブラリへインストールする前に、-C オプションを使ってこのモジュールを検査することができます。.xcl リンク・ファイルに次の行を入れます。

```
mycstart
-C c17400l
```

これによって、c17400l ライブラリにある CSTARTUP プログラム・モジュールの代わりに mycstart の CSTARTUP がロードされます。詳細は、「740 アセンブラ、リンカ、ライブラリアン・プログラミング・ガイド」を参照してください。

第 5 章　データ表示

この章では、740 C コンパイラが各 C データ型を表示する方法を説明し、効率的なコーディングのための推奨事項を示します。

データ型

740 C コンパイラは、ANSI C 基本構成要素をすべてサポートしています。変数には、低メモリ・アドレスにある最下位部が格納されます。

下表に、各 C データ型のサイズと範囲を示します。

データ型	バイト数	範囲	注記
char（省略時）	1	0 to 255	unsigned char に相当する。
char（-c オプションを使用）	1	-128 to 127	signed char に相当する。
signed char	1	-128 to 127	
unsigned char	1	0 to 255	
short, int	2	-2^{15} to $2^{15}-1$	-32768 to 32767
usigned short, unsigned int	2	0 to $2^{16}-1$	0 to 65535
long	4	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
unsigned long	4	0 to $2^{32}-1$	0 to 4294967295
pointer	2		
float, double, long double	4	±1.18E-38 to ±3.39E+38	

ENUM 型

enum キーワードは、値を格納するために必要な最短整数型（char, int または long）をもつ各オブジェクトを生成します。

浮動小数点

浮動小数点値は標準 IEEE 形式で 4 バイト数で表示されますので、単精度値と倍精度値は同じ表示となります。浮動小数点値が最小リミット未満の場合はゼロと見なされ、オーバーフローは未定義になっています。

4 バイト浮動小数点数のメモリ・レイアウトは、次のとおりです。

1）指数部　2）仮数部

数の値は次のとおりです。

（-1）S * 2（指数-127）* 1.仮数

ゼロは 4 バイトの 0 で表示されます。

float 型演算子（+, -, *, /）の精度は、約 7 桁です。

効率的なコーディング

非効率的な言語構成体の使用を避けるために、740 アーキテクチャの制限項目を知っておくことが大切です。以下に、740 を最適に使用するための推奨事項を列挙します。

- できるだけ 8 ビット・データ型を使用します。より大きなデータ型（int, long, float）は、740 アーキテクチャでは直接サポートしていません。また、ANSI C 規格に従って、int 型より短いデータ型はすべて、算術演算で使用された場合には汎整数拡張を、すなわち暗黙の型宣言を受けなければなりません。これは、明示的キャストを使用して回避することができます。
- できるだけ unsigned データ型を使用します。通常、740 は符号付き演算よりも符号なし演算をより効率的に実行します。これは、特に型変換、比較、配列索引付け、＞＞や／などの一部の算術演算に適用されます。

  可能であれば、最適化オプションを使用します。48 ページの「実行速度の最適化（-s）」、49 ページの「サイズの最適化（-z）」を参照してください。
- メモリの属性を適切に使用すると、重要なアプリケーションで実行速度とコード・サイズの両方が強化されます。151 ページの第 9 章「拡張キーワードのリファレンス」を参照してください。

第 6 章　汎用 C ライブラリの定義

この章では C ライブラリ関数の概要を説明し、ヘッダー・ファイルに従って関数を要約します。

概要

ICC C コンパイラ・パッケージは、PROM ベースの埋め込み型システムに適用される重要な C ライブラリ定義をほとんど提供します。これには、3 種類あります。

- ユーザ・プログラムのための標準 C ライブラリ定義。これは、この章で説明します。
- **CSTARTUP**、すなわちスタートアップ・コードをもつ単一のプログラム・モジュール
- 組込み関数。740 機能の低レベルまでの使用を可能にします。

ライブラリ・オブジェクト・ファイル

構成とモードの各組合わせには、全てのライブラリ定義をもつ単一ライブラリ・オブジェクト・ファイルがあります。リンカは、ユーザ・プログラムで（直接的に、あるいは間接的に）要求されるルーチンのみを取り込みます。

ライブラリ定義のほとんどは、変更を行わずに、すなわち配布されたライブラリ・オブジェクト・ファイルから直接使用することができます。この多くについて、ソースは任意指定で利用できます。残りは、目的アプリケーションに合わせてカスタマイズが必要になることがある I/O 向けルーチン（**putchar**, **getchar** など）です。これらについては、ソースは標準インストレーションの一部として提供されます。

ライブラリ・オブジェクト・ファイルは、グローバル型チェック・オプションをオン（**-gA**）にしてコンパイルさせて、提供されます。

ヘッダー・ファイル

ユーザ・プログラムは、**#include** 疑似命令を使用して取り込まれるヘッダー・ファイルによってライブラリ定義にアクセスできます。コンパイル時に時間を浪費しないために、ライブラリ定義は、それぞれが特定の関数部分を網羅した多数のさまざまなヘッダー・ファイルに分割されます。これによって、必要なライブラリ定義だけを含むことができます。

この定義を参照する前に、適切なヘッダー・ファイルを取り込むことが必要です。これを怠ると、実行中の呼出し不良、コンパイル時やリンク時のエラー・メッセージ、警告メッセージの発生につながります。

ライブラリ定義の要約

この項では、ヘッダー・ファイルの一覧を示し、それぞれのファイルに包含された関数を要約します。ヘッダー・ファイルには、目的プロセッサに固有な定義を追加することができます。これについては、145 ページの第 8 章「言語拡張機能」に説明しています。

ライブラリ関数はすべて、特に記述した場合を除き並列して再使用できます(再エントラント)。

文字操作 - ctype.h

isalnum	int isalnum(int c)	英数字の判定
isalpha	int isalpha(int c)	英字の判定
iscntrl	int iscntrl(int c)	制御コードの判定
isdigit	int isdigit(int c)	数字の判定
isgraph	int isgraph(int c)	非空白印字文字の判定
islower	int islower(int c)	小文字の判定
isprint	int isprint(int c)	印字文字の判定
ispunct	int ispunct(int c)	句切り文字の判定
isspace	int isspace(int c)	空白文字の判定
isupper	int isupper(int c)	大文字の判定
isxdigit	int isxdigit(int c)	16 進数の判定
tolower	int tolower(int c)	小文字化
toupper	int toupper(int c)	大文字化

低水準ルーチン - icclbutl.h

int _formatted_read (const char **line, const char **format, va_list ap)	定様式データを読み取ります。

`int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)`	データを書式化し、書き出します。
`int_ formatted_read (const char **line, const chart **format, va_list ap)`	浮動小数点数を除く定様式データを読み取ります。
`int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)`	浮動小数点数を除く定様式データを書き出します。
`int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)`	小定様式データ書込みルーチン

数学 - math.h

`double acos(double arg)`	逆余弦
`double asin(double arg)`	逆正弦
`double atan(double arg)`	逆正接
`double atan2(double arg1,) double arg2)`	象限逆正接
`double ceil(double arg)`	arg より大きい、あるいは等しい最小の整数
`double cos(double arg)`	余弦
`double cosh(double arg)`	双曲線余弦
`double exp(double arg)`	指数
`double fabs(double arg)`	倍精度浮動小数点の絶対値
`double floor(double arg)`	arg より小さい、あるいは等しい最大の整数
`double fmod(double arg1,) double arg2)`	浮動小数点剰余
`double frexp(double arg1,) int *arg2)`	浮動小数点数を 2 つの部分に分割します。

ldexp	double ldexp(double arg1, int arg2)	2 の累乗で乗算します。
log	double log(double arg)	自然対数
log10	double log10(double arg)	常用対数
modf	double modf(double value, double *iptr)	小数部と整数部
pow	double pow(double arg1, doule arg2)	累乗
sin	double sin(double arg)	正弦
sinh	double sinh(double arg)	双曲線正弦
sqrt	double sqrt(double arg)	平方根
tan	double tan(double x)	正接
tanh	double tanh(double arg)	双曲線正接

非局所分岐 - setjmp.h

longjmp	void longjmp(jmp_buf env, int val)	環境回復
setjmp	int setjmp(jmp_buf env)	環境の退避設定

可変個引数- stdarg.h

va_arg	type va_arg(va_list ap, mode)	関数呼出しの次の引数
va_end	void va_end(va_list ap)	関数呼出し引数の読取りを終了します。
va_list	char *va_list[1]	引数リストの型
va_sart	void va_start(va_list ap, parmN)	関数呼出し引数の読取りを開始します。

入出力 - stdio.h

getchar	int getchar(void)	文字を取得します。
gets	char *gets(char *s)	文字列を取得します。

`int printf(const char *format, ...)`	定様式データの書き出し
`int putchar(int value)`	文字の表示
`int puts(const char *s)`	文字列の表示
`int scanf(const char *format, ...)`	定様式データの読取り
`int sprintf(char *s, const char *format,)`	文字列に定様式データを書き出します。
`int sscanf(const char *s, const char *format, ...)`	文字列から定様式データを読み取ります。

一般実用 - stdlib.h

`void abort(void)`	プログラムを異常終了させます。
`int abs(int j)`	絶対値
`double atof(const char *nptr)`	ASCII 形式を double に変換します。
`int atoi(const char *nptr)`	ASCII 形式を int に変換します。
`long atol(const char *nptr)`	ASCII 形式を long int に変換します。
`void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base))`	配列の中で一般的な検索を行います。
`void *calloc(size_t nelem, size_t elsize)`	メモリをオブジェクトの配列に割り付けます。
`div_t div(int numer, int denom)`	除算
`void exit(in status)`	プログラムの終了
`void free(void *ptr)`	メモリの解放
`long int labs(long int j)`	長整数絶対値
`ldiv_t ldiv(long int numer, long int denom)`	長整数の除算

malloc	void *malloc(size_t size)	メモリの割付
qsort	void qsort(const void base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base))	配列の一般的な整列を行います。
rand	int rand(void)	乱数
realloc	void *realloc(void *ptr, size_t size)	メモリの再割付
srand	void srand(unsigned int seed)	乱数列の設定
strtod	double strtod(const char *nptr, char *:endptr)	文字列を double に変換します。
strtol	long int strtol(const char *nptr, char **endptr, int base)	文字列を長整数に変換します。
strtoul	unsigned long int strtoul const char *nptr, char **endptr, base int)	文字列を符号なし長整数に変換します。

文字列操作 - string.h

memchr	void *memchr(const void s, int c, size_t n)	メモリの中の文字を検索します。
memcmp	int memcmp(const void *s1, const void *s2, size_t n)	メモリの比較
memcpy	void *memcpy(void *s1, const void *s2, size_t n)	メモリの複写
memmove	void *memmove(void *s1, const void *s2, size_t n)	メモリの移動
memset	void *memset(void *s, int c, size_t n)	メモリの設定
strcat	char *strcat(char *s1, const char *s2)	文字列の連結
strchr	char *strchr(const char *s, int c)	文字列の中で文字の有無を検索します。

int strcmp(const char *s1, const char *s2)	2 つの文字列を比較します。
int strcoll(const char *s1, const char *s2)	文字列の比較
char *strcpy(char *s1, const char *s2)	文字列の複写
size_t strcspn(const char *s1, const cha *s2)	文字列の中の排除文字の長さを求めます。
char *strerror(int errnum)	エラー・メッセージ文字列を与えます。
size_t strlen(const char *s)	文字列長さ
char *strncat(char *s1, const char *s2, size_t n)	指定数の文字を文字列と連結します。
int strncmp(const char *s1, const char *s2, size_t n)	指定数の文字を文字列と比較します。
char *strncpy(char *s1, const char *s2, size_t n)	文字列から指定数の文字を複写します。
char *strpbrk(const char *s1, const char *s2)	文字列の中の指定文字の 1 つを探索します。
char *strrchr(const char s, int c)	文字列の右側から文字を探索します。
size_t strspn(const char *s1, const char *s2)	文字列の文字の長さを求めます。
char *strsr(const char s1, const char *s2)	サブストリングの有無を検索します。
char *strtok(char *s1, const char *s2)	文字列をトークンに分割します。
size_t strxfrm8char *s1, const char *s2, size_t n)	文字列を変形し、長さを返します。

共通定義 - stddef.h

関数はありません（**size_t**, **NULL**, **ptrdiff_t**, **offsetof** などを含む各種の定義）。

整数型 - limits.h

関数はありません（整数型の各種のリミットとサイズ）。

浮動小数点型 - float.h

関数はありません（浮動小数点型の各種のリミットとサイズ）。

エラー - errno.h

関数はありません（各種のエラー返却値）。

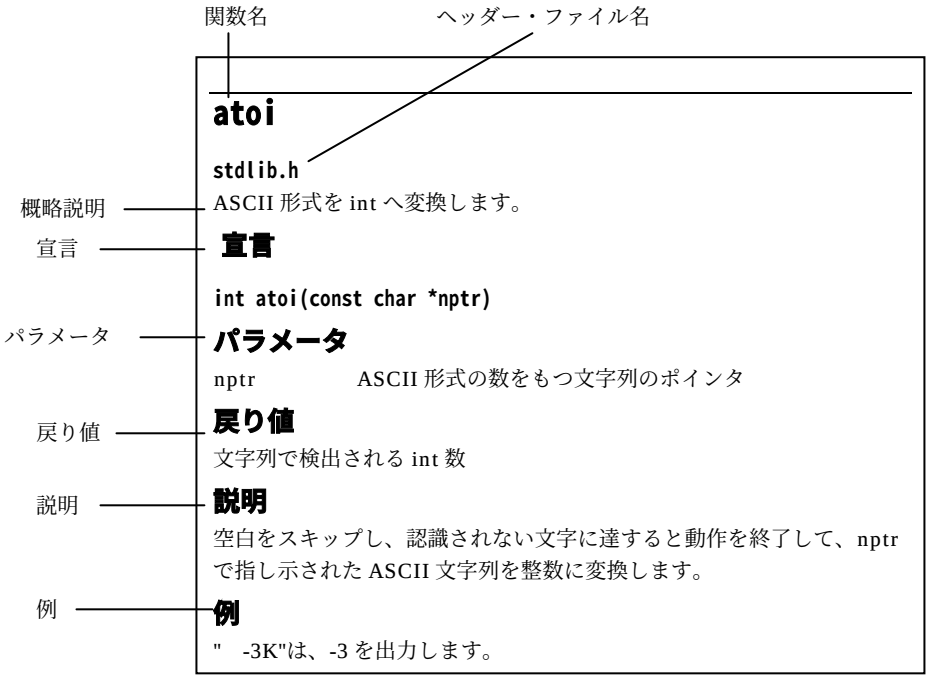
アサート - assert.h

assert void assert(int expression) 式をチェックします。

第 7 章　C ライブラリ関数のリファレンス

この章では、C ライブラリ関数をアルファベット順に示し、その動作を詳細に説明するとともに各関数で利用可能なオプションを示します。

各ライブラリ関数の説明の形式は、次のとおりです。



関数名

C ライブラリ関数名です。

ヘッダー・ファイル名

関数ヘッダー・ファイル名です。

概略説明

関数の要約です。

宣言

C ライブラリの宣言です。

パラメータ

宣言の中の各パラメータの詳細説明です。

戻り値

戻り値が存在する場合に、関数が返す値です。

説明

ライブラリ関数の最も一般的な使用を網羅した詳細説明です。これには、関数の用途に関する内容、特殊な条件の説明、一般的に犯しやすい誤りを記述しています。

例

ライブラリ関数の使用を説明した 1 つ以上の例です。

abort

stdlib.h

プログラムを異常終了させます。

宣言

```
void abort(void)
```

パラメータ

なし

戻り値

なし

説明

プログラムを異常終了させ、呼出し元には戻りません。この関数は、exit 関数を呼び出します。省略時、この関数のエントリは CSTARTUP に存在します。

abs

stdlib.h

絶対値

宣言

```
int abs(int j)
```

パラメータ

j　　int 値

戻り値

絶対値 j をもつ int

説明

絶対値 j を計算します。

acos

math.h

逆余弦

宣言

```
double acos(double arg)
```

パラメータ

arg　　範囲[–1,+1]の double 型です。

戻り値

範囲 [–pi/2, +pi/2]の arg の double 逆余弦

説明

arg の逆余弦の主値をラジアンで計算します。

asin

math.h

宣言

double asin(double arg)

パラメータ

arg　　arg の逆余弦の主値をラジアンで計算します。

戻り値

範囲[-pi/2,+pi/2]の arg の double 逆余弦

説明

arg の逆余弦の主値をラジアンで計算します。

assert

assert.h

式をチェックします。

宣言

void assert (int expression)

パラメータ

expression　チェックする式

戻り値

なし

説明

これは、式をチェックするマクロです。結果が偽の場合は、stderr にメッセージを出力し、abort を呼び出します。

メッセージは次の形式で出力されます。

File ファイル名; line 行番号 Assertion failure "式"

assert 呼出しを無視するには、#include <assert.h>文の前に#define NDEBUG 文を入れます。

atan

math.h

逆正接

宣言

```
double atan(double arg)
```

パラメータ

arg　　double 値

戻り値

範囲[-pi/2,pi/2]の arg の double 逆正接

説明

arg の逆正接を計算します。

atan2

math.h

象限逆正接

宣言

```
double atan2(double arg1, double arg2)
```

パラメータ

arg1　　double 値

arg2　　double 値

戻り値

範囲[-pi,pi]の arg1/arg2 の double 逆正接

説明

両方の引数の符号を使用して arg1/arg2 の逆正接を計算し、戻り値の象限を決定します。

atof

stdlib.h

ASCII 形式を double へ変換します。

宣言

```
double atof(const char *nptr)
```

パラメータ

nptr　　ASCII 形式の数をもつ文字列のポインタ

戻り値

文字列で検出される double 数

説明

空白をスキップし、認識されない文字に達すると動作を終了して、nptr で指し示された文字列を倍精度浮動小数点数に変換します。

例

" -3K"は、-3.00 を出力します。

".0006"は、0.0006 を出力します。

"1e-4"は、0.0001 を出力します。

atoi

stdlib.h

ASCII 形式を int へ変換します。

宣言

```
int atoi(const char *nptr)
```

パラメータ

nptr　　ASCII 形式の数をもつ文字列のポインタ

戻り値

文字列で検出される int 数

説明

空白をスキップし、認識されない文字に達すると動作を終了して、nptr で指し示された ASCII 文字列を整数に変換します。

例

" -3K"は、-3 を出力します。

"6"は、6 を出力します。

"149"は、149 を出力します。

atol

stdlib.h

ASCII 形式を long int へ変換します。

宣言

```
long atol(const char *nptr)
```

パラメータ

nptr　　ASCII 形式の数をもつ文字列のポインタ

戻り値

文字列で検出される long 数

説明

空白をスキップし、認識されない文字に達すると動作を終了して、nptr で指し示された ASCII 文字列で検出された数を長整数値に変換します。

例

" -3K"は、-3 を出力します。

"6"は、6 を出力します。

"149"は、149 を出力します。

bsearch

stdlib.h

配列の中で一般的な検索を行います。

宣言

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size,
int (*compare) (const void *_key, const void *_base));
```

パラメータ

key	検索の対象となるオブジェクトのポインタ
base	検索する配列のポインタ
nmemb	base で指し示された配列の次元
size	配列要素のサイズ
compare	戻り値
	＜0（負の値　_key が_base 未満の場合
	0　_key が_base と等しい場合
	＞0（正の値　_key が_base を超える場合

説明

base で指し示された nmemb オブジェクトの配列を検索し、key で指し示されたオブジェクトと一致する要素の有無を調べます。

calloc

stdlib.h

オブジェクトの配列にメモリを割り当てます。

宣言

```
void *calloc(size_t nelem, size_t elsize)
```

パラメータ

nelem	オブジェクト数
elsize	各オブジェクトのサイズを指定する size_t 型の値

戻り値

結果	値
割当てられた	メモリ・ブロックの始まり（最低アドレス）のポインタ
割当てられなかった	必要サイズ以上のメモリ・ブロックが利用できない場合は、ゼロ

説明

指定されたサイズのオブジェクトの配列にメモリ・ブロックを割り当てます。移植性を確実にするために、サイズは、バイト数などのメモリの絶対単位ではなく sizeof 関数で返却される 1 サイズあるいは複数のサイズで与えられます。

メモリの利用度は、省略時のヒープ・サイズによって異なります。

ceil

math.h

arg より大きい、あるいは等しい最小の整数

宣言

```
double ceil(double arg)
```

パラメータ

arg　　double 値

戻り値

arg より大きい、あるいは等しい最小の整数値をもつ double

説明

arg より大きい、あるいは等しい最小の整数値を計算します。

cos

math.h

余弦

宣言

```
double cos(double arg)
```

パラメータ

arg　　ラジアンで表した double 値

戻り値

arg の double 余弦

説明

ラジアン arg の余弦を計算します。

cosh

math.h

双曲線余弦

宣言

```
double cosh(double arg)
```

パラメータ

arg	ラジアンで表した double 値

戻り値

arg の double 双曲線余弦

説明

ラジアン arg の双曲線余弦を計算します。

div

stdlib.h

除算

宣言

```
div_t div(int numer, int denom)
```

パラメータ

numer	int 分子
demon	nt 分母

戻り値

除算の商および剰余結果を保持する div_t 型の構成体

説明

分子 numer を分母 denom で除算します。型 div_t は stdlib.h に定義されています。

除算が不正確な場合、商は代数学上の商より小さな最大の整数になります。結果は次のように定義されます。

```
quot * denom + rem == numer
```

exit

stdlib.h

プログラムの終了

宣言

```
void exit(int status)
```

パラメータ

status　　int ステータス値

戻り値

なし

説明

プログラムを通常どおりに終了します。この関数は、呼出し元には戻りません。この関数のエントリは、省略時には CSTARTUP に存在します。

exp

math.h

指数

宣言

```
double exp(double arg)
```

パラメータ

arg　　double 値

戻り値
arg の指数関数の値をもつ double

説明
arg の指数関数を計算します。

fabs

math.h

倍精度浮動小数点絶対値

宣言
double fabs(double arg)

パラメータ
arg　　double 値

戻り値
arg の double 絶対値

説明
浮動小数点数 arg の絶対値を計算します。

floor

math.h

arg より小さい、あるいは等しい最大の整数

宣言
double floor(double arg)

パラメータ
arg　　double 値

戻り値
arg より小さい、あるいは等しい最大の整数値をもつ double

説明

arg より小さい、あるいは等しい最大の整数値を計算します。

fmod

math.h

浮動小数点の剰余

宣言

double fmod(double arg1, double arg2)

パラメータ

arg1	double 分子
arg2	double 分母

戻り値

arg1/arg2 除算の double 剰余

説明

arg1/arg2 の剰余を、すなわち一部の整数 i について、arg2 が 0 以外の場合に結果が arg1 と同じ符号をもち arg2 の大きさよりも小さくなるように、arg1-i*arg2 の値を計算します。

free

stdlib.h

メモリの解放

宣言

void free(void *ptr)

パラメータ

ptr	malloc, calloc、あるいは realloc で割り当てられた以前のメモリ・ブロックのポインタ

戻り値

なし

説明

ptr.ptr で指し示されたオブジェクトによって使用され、malloc, calloc、あるいは realloc から値を割り付けられた以前のメモリを解放します。

frexp

math.h

浮動小数点数を 2 つの部分に分割します。

宣言

double frexp(double arg1, int *arg2)

パラメータ

arg1	分割する浮動小数点数
arg2	arg1 の指数を入れるための、整数のポインタ

戻り値

範囲 0.5〜1.0 の arg1 の double 仮数

説明

浮動小数点数 arg1 を、*arg2 に保存する指数と、関数の値として返される仮数とに分割します。

値は次のようになります。

仮数＊2 指数＝値

getchar

stdio.h

文字を取得します。

宣言

int getchar(void)

パラメータ

なし

戻り値

標準入力ストリームからの次の文字の ASCII 値をもつ int

説明

標準入力ストリームから次の文字を取得します。

この関数は、特定の目的ハードウェア構成に合わせてカスタマイズしてください。getchar 関数は、getchar.c ファイルにソース形式で提供されます。

gets

stdio.h

文字列を取得します。

宣言

```
char *gets(char *s)
```

パラメータ

s　　入力を受け取る文字列のポインタ

戻り値

結果	値
取得できた	s に等しいポインタ
取得できなかった	ヌル

説明

標準入力から次の文字列を取得し、指し示された文字列に入れます。文字列は、行の終り、あるいはファイルの終りで終了します。行の終了文字は、ゼロで置き換えられます。

この関数は、特定の目的ハードウェア構成に適合しなければならない getchar を呼び出します。

isalnum

ctype.h

英数字の判定

宣言

int isalnum(int c)

パラメータ

c　　文字を表した int

戻り値

c が英数字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が英数字であるかどうかを検査します。

isalpha

ctype.h

英字の判定

宣言

int isalpha(int c)

パラメータ

c　　文字を表した int

戻り値

c が英字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が英字かどうかを検査します。

iscntrl

ctype.h

制御コードの判定

宣言

```
int iscntrl(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が制御コードの場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が制御コードかどうかを検査します。

isdigit

ctype.h

数字の判定

宣言

```
int isdigit(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が数字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が 10 進数かどうかを検査します。

isgraph

ctype.h

非空白印字文字の判定

宣言

```
int isgraph(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が非空白印字文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が非空白印字文字かどうかを検査します。

islower

ctype.h

小文字の判定

宣言

```
int islower(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が小文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が小文字かどうかを検査します。

isprint

ctype.h

印字文字の判定

宣言

```
int isprint(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が、空白を含み印字文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が、空白を含み印字文字かどうかを検査します。

ispunct

ctype.h

句切り文字の判定

宣言

```
int ispunct(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が空白、数字、英字以外の印字文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が、空白、数字、英字以外の印字文字かどうかを検査します。

isspace

ctype.h

空白文字の判定

宣言

```
int isspace(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が空白文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が空白文字、すなわち次のうちの 1 つであるかどうかを検査します。

文字	シンボル
空白	" "
書式送り	＼f
改行	＼n
復帰	＼r
水平タブ	＼t
垂直タブ	＼v

isupper

ctype.h

大文字の判定

宣言

```
int isupper(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が大文字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が大文字かどうかを検査します。

isxdigit

ctype.h

16 進数の判定

宣言

```
int isxdigit(int c)
```

パラメータ

c　　文字を表した int

戻り値

c が大文字や小文字の数字の場合は 0 以外の値、そうでない場合は 0 となる int

説明

文字が大文字や小文字の 16 進数、すなわち 0～9、a～f、A～F の中の 1 つであるかどうかを検査します。

labs

stdlib.h

長整数絶対値

宣言

```
long int labs(long int j)
```

パラメータ

j　　long int 値

戻り値

j の long int 絶対値

説明

長整数 j の絶対値を計算します。

ldexp

math.h

2 の累乗で乗算します。

宣言

```
double ldexp(double arg1, int arg2)
```

パラメータ

arg1	double 乗数値
arg2	int 累乗値

戻り値

arg2 乗した 2 で乗算された arg1 の double 値

説明

累乗した 2 で乗算された浮動小数点数の値を計算します。

ldiv

stdlib.h

長整数の除算

宣言

```
ldiv_t ldiv(long int numer, long int denom)
```

パラメータ

numer　　long int 分子

denom　　long int 分母

戻り値

除算の商および剰余を保持する ldiv_t 型の struct

説明

分子 numer を分母 denom で除算します。型 ldiv_t は、stdlib.h に定義されています。

除算が不正確な場合、商は代数学上の商より小さい最大の整数になります。結果は次のように定義されます。

```
quot * denom + rem == numer
```

log

math.h

自然対数

宣言

```
double log(double arg)
```

パラメータ

arg　　double 値

戻り値

arg の double 自然対数

説明

数値の自然対数を計算します。

log10

math.h

常用対数

宣言

double log10(double arg)

パラメータ

arg	double 値

戻り値

arg の double 常用対数

説明

数の常用対数を計算します。

longjmp

setjmp.h

環境回復

宣言

void longjmp(jmp_buf env, int val)

パラメータ

env	setjmp で設定された環境を保持する jmp_buf 型の struct
val	対応する setjmp で返却される int 値

戻り値

なし

説明

setjmp で保存された以前の環境を回復します。これによって、対応する setjmp からのリターンとしてプログラムの実行が継続され、val 値を返します。

malloc

stdlib.h

メモリの割付

宣言

```
void *malloc(size_t size)
```

パラメータ

size	オブジェクトのサイズを指定する size_t オブジェクト

戻り値

結果	値
割付けられた	メモリ・ブロックの始まり（最低アドレス）のポインタ
割付けられなかった	必要サイズ以上のメモリ・ブロックが利用できない場合は、ヌル

説明

指定されたサイズのオブジェクトにメモリ・ブロックを割り当てます。

メモリの利用度は、省略時のヒープ・サイズによって異なります。

memchr

string.h

メモリの中で文字を検索します。

宣言

```
void *memchr(const void *s, int c, size_t n)
```

パラメータ

s	オブジェクトのポインタ
c	文字を表した int
n	各オブジェクトのサイズを指定する size_t 型の値

戻り値

結果	値
検索できた	s で指し示された n 文字の中で最初に出現する c のポインタ
検索できなかった	ヌル

説明

指定されたサイズのメモリの指し示した領域の中で最初に出現する文字を検索します。

オブジェクトの中の 1 文字も複数の文字も、無符号型として処理されます。

memcmp

string.h

メモリの比較

宣言

```
int *memcmp(const void *s1, const void *s2, size_t n)
```

パラメータ

s1	最初のオブジェクトのポインタ
s2	2 番目のオブジェクトのポインタ
n	各オブジェクトのサイズを指定する size_t 型の値

戻り値

s1 で指し示されたオブジェクトの最初の n 文字と s2 で指し示されたオブジェクトの最初の n 文字との比較の結果を示す整数

戻り値	意味
>0	**s1 < s2**
=0	**s1 = s2**
<0	**s1 < s2**

説明

2 つのオブジェクトの最初の n 文字を比較します。

memcpy

string.h

メモリの複写

宣言

```
void *memcpy(void *s1, const void *s2, size_t n)
```

パラメータ

s1	複写先のオブジェクトのポインタ
s2	複写元のオブジェクトのポインタ
n	複写する文字数

戻り値

s1

説明

複写元のオブジェクトから複写先のオブジェクトに指定数の文字を複写します。

オブジェクトが重なり合う場合、結果は未定義になりますので、代わって memmove を使用してください。

memmove

string.h

メモリの移動

宣言

```
void *memmove(void *s1, const void *s2, size_t n)
```

パラメータ

s1	複写先のオブジェクトのポインタ
s2	複写元のオブジェクトのポインタ
n	複写する文字数

戻り値

s1

説明

複写元のオブジェクトから複写先のオブジェクトに指定数の文字を複写します。

複写は、あたかも複写元の文字がどちらのオブジェクトにも重なり合わない一時配列にまず複写され、その後一時配列からの文字が複写先のオブジェクトに複写されるかのようにして行われます。

memset

string.h

メモリの設定

宣言

```
void *memset(void *s, int c, size_t n)
```

パラメータ

s	複写先のオブジェクトのポインタ
c	文字を表した int
n	オブジェクトのサイズ

戻り値

s

説明

文字（unsigned char 型に変換されたもの）を複写先のオブジェクトの最初の指定文字数の 1 つ 1 つに複写します。

modf

math.h

小数部と整数部

宣言

```
double modf(double value, double *iptr)
```

パラメータ

value	double 値
iptr	value の整数部を受け取らなければならない double のポインタ

戻り値

value の小数部

説明

value の小数部と整数部を計算します。小数部と整数部の符号は、value の符号と同じです。

pow

math.h

累乗

宣言

```
double pow(double arg1, double arg2)
```

パラメータ

arg1	double の数
arg2	double 累乗

戻り値

arg2 乗された arg1

説明

数の累乗を計算します。

printf

stdio.h

定様式データの書き出し

宣言

```
int printf(const char *format, ...)
```

パラメータ

format	書式文字列のポインタ
...	format の制御下で出力されなければならない任意指定の値

戻り値

結果	値
書き出された	書き出された文字数
書き出されなかった	エラーが発生した場合は、負の値

説明

標準出力ストリームに定様式データを書き出し、書き出された文字数を、あるいはエラーが発生した場合は、負の値を返します。

完全なフォーマッタは多くの領域を要求しますので、数種類のフォーマッタの中から選択できるようになっています。詳細については、53 ページの第 4 章「構成」を参照してください。

format は、出力される文字の並びおよび変換仕様から成る文字列です。各変換仕様によって、format 文字列の後に続く次の引数が評価、変換され、書き出されます。

変換仕様の形式は、次のとおりです。

% **[flags] [field_width] [.precision] [length_modifier]conversion**

[ ]内の項目は、任意指定です。

フラグ

flags は、次のとおりです。

フラグ	意味
-	左詰めフィールド
+	符号付きの値は、常に＋あるいは－符号から始まります。
space	値は、必ず－あるいは空白から始まります。
#	代替形式： **指定子** 意味 **octal** 第 1 桁は常に 0 である。 **G g** 10 進小数点は出力され、後続の 0 は保持される。 **E e f** 10 進小数点は出力される。 **X** 0 以外の値は、結果の前に 0X がつけられる。
x	0 以外の値は、結果の前に 0X がつけられる。
0	フィールド幅を 0 で埋める（d, i, o, u, x, X, e, E, f, g, G 指定子の場合）。

フィールド幅

field_width とは、フィールドに出力される文字数のことです。必要であれば、空白でフィールドを埋めます。負の値は、左詰めフィールドを示します。フィールド幅に＊があると、次の後続引数の値を意味します。この値は整数でなければなりません。

精度

precision とは、整数（d, i, o, u, x, X）の場合は出力桁数、浮動小数点値（e, E, f）の場合は出力された小数点以下の桁数、g や G 変換の場合は有効桁数のことをいいます。フィールド幅が＊の場合は、次の後続引数の値を意味します。この値は整数でなければなりません。

長さ修飾子

各 length_modifier の意味は、下表のとおりです。

長さ修飾子	内容
h	d, i, u, x, X、あるいは o 指定子の前に使用され、short int あるいは unsigned short int 値を意味する。
l	d, i, u, x, X、あるいは o 指定子の前に使用され、long 整数、あるいは unsigned long 値を意味する。
L	e, E, f, g、あるいは G 指定子の前に使用され、long double 値を意味する。

変換

conversion の各値の結果は、下表のとおりです。

変換指定子	結果
d	符号つき 10 進数
I	符号つき 10 進数
o	符号なし 8 進数
u	符号なし 10 進数
x	小文字の符号なし 16 進数（0～9、a～f）
X	大文字の符号なし 16 進数（0～9、A～F）
e	[-]d.ddde+dd 形式の double 値
E	[-]d.dddE+dd 形式の double 値
f	[-]ddd.ddd 形式の double 値
g	f または e 形式の double 値のより適切な方
G	F または E 形式の double 値のより適切な方
C	単一の文字定数
s	文字列定数
p	ポインタ値（アドレス）
n	出力しない。ただし、次の引数で指し示された整数に、それまでに書き出された文字数を格納する。
%	%文字

拡張規則は全ての char および short int 引数を int に変換するのに対し、floats は double に変換されることに注意してください。

printf は、目的ハードウェア構成に適用されなければならない putchar ライブラリ関数を呼び出します。

printf のソースは、printf.c ファイルに提供されています。より少ないプログラム領域とスタックしか使用しない縮小バージョンのソースは、intwri.c ファイルに提供されています。

例

C 文

```
int i=6, j=-6;
char *p = “ABC”;
long l=100000;
float f1= 0.0000001;
f2 = 750000;
double d = 2.2;
```

の後では、さまざまな printf 関数の呼出しの意味は下表に示すとおりです。Δは空白を示します。

文	出力	出力文字
printf("%c",p[1])	B	1
printf("%d",i)	6	1
printf("%3d",i)	ΔΔ6	3
printf("%.3d",i)	006	3
printf("%-10.3d",i)	006ΔΔΔΔΔΔΔ	10
printf("Value=%+3d",i)	Value=Δ+6	9
printf("%10.*d",i,j)	ΔΔΔ-000006	10
printf("String=[%s]",p)	String=[ABC]	12
printf("Value=%1X",l)	Value=186A0	11
printf("%f",f1)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",f1)	1.000000e-07	12
printf("%16e",d)	ΔΔΔΔ2.20000e+00	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",f1)	1e-07	5
printf("%g",f2)	750000	6
printf("%g",d)	2.2	3

putchar

stdio.h

文字の表示

宣言

int putchar(int value)

パラメータ

value　　表示する文字を表した int

戻り値

結果	値
書き出しできた	value
書き出しできなかった	EOF マクロ

説明

標準出力ストリームに文字を書き出します。

この関数は、特定の目的ハードウェア構成に合わせてカスタマイズしてください。この関数は、putchar.c ファイルにソース形式で提供されています。

この関数は、printf で呼び出されます。

puts

stdio.h

文字列の表示

宣言

```
int puts(const char *s)
```

パラメータ

s　　表示する文字列のポインタ

戻り値

結果	値
書き出しできた	非負の値
書き出しできなかった	エラーが発生した場合は、-1

説明

改行文字が続く文字列を標準出力ストリームに書き出します。

qsort

stdlib.h

配列の一般的な整列を行います。

宣言

```
void qsort(const void *base, size_t nmemb, size_t size, int (*compare)
(const void *_key, const void *_base));
```

パラメータ

base	整列する配列のポインタ
nmemb	base で指し示された配列の次元
size	配列要素のサイズ
compare	返却値
	＜0（負の値）_key が_base 未満の場合
	0 _key が_base と等しい場合
	＞0（正の値）_key が_base を超える場合

説明

base で指し示された nmemb オブジェクトの配列を整列します。

rand

stdlib.h

乱数

宣言

```
int rand(void)
```

パラメータ

なし

返却値

乱数列の次の int

説明

範囲[0,RAND_MAX]に入るように変換された現在の疑似乱数整数列の次の int を計算します。疑似乱数列のシード方法の説明については、srand を参照してください。

realloc

stdlib.h

メモリの再割付

宣言

```
void *realloc(void *ptr, size_t size)
```

パラメータ

ptr	メモリ・ブロックの始まりを示すポインタ
size	オブジェクトのサイズを指定する型 size_t の値

返却値

結果	値
再割付できた	メモリ・ブロックの始まり（最低アドレス）のポインタ
再割付できなかった	必要サイズ以上のメモリ・ブロックが利用できない場合は、ヌル

説明

メモリ・ブロック（malloc, calloc、あるいは realloc で割り付けられます）のサイズを変更します。

scanf

stdio.h

定様式データの読取り

宣言

```
int scanf(const char *format, ...)
```

パラメータ

format	書式文字列のポインタ
...	値を受け取る任意指定のポインタ

返却値

結果	値
読取りできた	完了した変換の数
読取りできなかった	入力が尽きてしまった場合は、-1

説明

標準入力ストリームから定様式データを読み取ります。

完全なフォーマッタは多くの領域を要求しますので、数種類のフォーマッタの中から選択できるようになっています。詳細については、58 ページの「入出力」を参照してください。

format は、一般文字の並びと変換仕様から成る文字列です。各一般文字は、入力から突合せ文字を読み取ります。各変換仕様は、変換仕様を満たす入力を受け付け、変換し、format の後に続く次の引数によって指し示されたオブジェクトにそれを割り付けます。

書式文字列に空白文字があると、入力は非空白文字が検出されるまでスキャンされます。

変換仕様の形式は、次のとおりです。

```
% [assign_suppress] [field_width] [.precision] [length_modifier]conversion
```

[ ]内の項目は、任意指定です。

代入抑止

この位置に＊があると、フィールドのスキャンは行われますが、割付は実行されません。

フィールド幅

field_width とは、スキャンする最大フィールドのことです。省略時は、文字が一致しなくなるまでスキャンされます。

長さ修飾子

各 length_modifier の意味は、下表のとおりです。

長さ修飾子	使用箇所	意味
l	**d，i、あるいは n の前**	int に対する long int
	o，u、あるいは x の前	unsigned int に対する unsigned long int
	e，E，g，G、あるいは f の前	float に対する double オペランド
h	**d，i、あるいは n の前**	int に対する short int
	o，u、あるいは x の前	unsigned int に対する unsigned short int
L	**e，E，g，G、あるいは f の前**	float に対する long double オペランド

変換

各変換の意味は、下表のとおりです。

変換指定子	結果
d	任意指定の符号付き 10 進整数値
I	標準 C 表記法で表された任意指定の符号付き整数値。すなわち、8 進数（0n）か 16 進数（0xn,0Xn）
o	任意指定の符号付き 8 進整数
u	符号なし 10 進整数
x	任意指定の符号付き 16 進整数
X	任意指定の符号付き 16 進整数（x と同等）
f	浮動小数点定数

変換指定子	結果
e E g G	浮動小数点定数（f と同等）
s	文字列
c	1 文字あるいは field_width 文字
n	読取りなし。ただし、次の引数で指し示された整数に、それまでに読み込まれた文字数を格納する。
p	ポインタ値（アドレス）
[	終端] の前の文字と一致するあらゆる数の文字。例えば、[abc]は、a, b、または c を意味する。
[]	] と一致するあらゆる数の文字、あるいは別の終端] の前の文字。例えば、[]abc]は、], a, b、あるいは c を意味する。
[^	終端] の前のいずれの文字とも一致しないあらゆる数の文字。例えば、[^abc]は、a, b、あるいは c でもないという意味である。
[^]	] と一致しないあらゆる数の文字、あるいは別の終端] の前の文字。例えば、[^]abc]は、], a, b、あるいは c でもないという意味である。
%	%文字

c, n、および全形式の [を除く全ての変換では、先頭の空白文字はスキップされます。

scanf は、実際の目的ハードウェア構成に適用されなければならない getchar ライブラリ関数を間接的に呼び出します。

例

例えば、プログラム

```
int n, I;
char name [50];
float x;
n = scanf ("%d%f%s",&1,&x,name)
```

の後では、入力行

```
25 54.32E-1 Hello World
```

は、次のように変数を設定します。

```
n = 3, i = 25, x = 5.432, name="Hello World"
```

関数

```
scanf("%2d%f%*d %[0123456789]", &i, &x, name)
```

に入力行

```
56789 0123 56a72
```

を設定すると、次のように変数が設定されます。

```
i = 56, x = 789.0, name="56" (0123 unassigned)
```

setjmp

setjmp.h

環境の退避設定

宣言

```
int setjmp(jmp_buf env)
```

パラメータ

env　　setjmp による環境の格納先の型 jmp_buf のオブジェクト

返却値

ゼロ

対応する longjmp が実行されると、あたかもそれが setjmp からのリターンであるかのように実行が継続されます。この場合、longjmp に指定された int の値が返されます。

説明

後に longjmp で使用されるように env に環境を保存します。

setjmp は、常に同じ関数で、あるいは longjmp に対する対応呼出しよりもより高いネスト・レベルで使用されなければなりません。

sin

math.h

正弦

宣言

```
double sin(double arg)
```

パラメータ

arg　　ラジアンで表した double 値

返却値

arg の double 正弦

説明

数の正弦を計算します。

sinh

math.h

双曲線正弦

宣言

```
double sinh(double arg)
```

パラメータ

arg　　ラジアンで表した double 値

返却値

arg の double 双曲線正弦

説明

arg ラジアンの双曲線正弦を計算します。

sprintf

stdio.h

定様式データを文字列に書き出します。

宣言

```
int sprintf(char *s, const char *format, ...)
```

パラメータ

s	定様式のデータを受け取る文字列のポインタ
format	書式文字列のポインタ
...	format の制御下で出力される任意指定の値

返却値

結果	値
書き出された	書き出された文字数
書き出されなかった	エラーが発生した場合は、負の値

説明

出力が文字列に向けて送信されるという点以外は、printf と同じように動作します。

sprintf は、putchar 関数を使用しません。したがって、目的構成で putchar が利用できない場合でも使用することができます。

完全なフォーマッタは多くの領域を要求しますので、数種類のフォーマッタの中から選択できるようになっています。詳細については、58 ページの「入出力」を参照してください。

sqrt

math.h

平方根

宣言

```
double sqrt(double arg)
```

パラメータ

arg　　double 値

返却値

arg の double 平方根

説明

数の平方根を計算します。

srand

stdlib.h

乱数列を設定します。

宣言

```
void srand(unsigned int seed)
```

パラメータ

seed　　特定の乱数列を識別する unsigned int 値

返却値

なし

説明

繰返し可能な疑似乱数列を選択します。

rand 関数は、乱数列から連続した乱数を取得するのに使用されます。srand に対する呼出しが行われる前に rand が呼び出された場合、生成される乱数列は srand(1)の後に生成されます。

sscanf

stdio.h

文字列から定様式データを読み取ります。

宣言

```
int sscanf(const char *s, const char *format, ...)
```

パラメータ

s	定様式のデータをもつ文字列のポインタ
format	書式文字列のポインタ
...	値を受け付ける変数を示す任意指定のポインタ

返却値

結果	値
読み取られた	完了した変換の数
読み取られなかった	入力が尽きた場合は-1

説明

入力が文字列 s から取得されるという点以外は、scanf と同じように動作します。詳細は、scanf を参照してください。

sscanf 関数は、getchar を使用しません。したがって、目的構成で getchar が利用できない場合でも使用することができます。

完全なフォーマッタは多くの領域を要求しますので、数種類のフォーマッタの中から選択できるようになっています。詳細については、53 ページの第 4 章「構成」を参照してください。

strcat

string.h

文字列の連結

宣言

```
char *strcat(char *s1, const char *s2)
```

パラメータ

s1　　第 1 文字列のポインタ

s2　　第 2 文字列のポインタ

返却値

s1

説明

第 2 文字列のコピーを第 1 文字列の終りに付加します。第 2 文字列の初期文字は、第 1 文字列の終端ヌル文字を上書きします。

strchr

string.h

文字列の中で文字の有無を検索します。

宣言

```
char *strchr(const char *s, int c)
```

パラメータ

c　　文字の int 表現

s　　文字列のポインタ

返却値

検索が完了すると、s で差し示された文字列の中で最初に出現する c（char 型に変換されたもの）のポインタ

c の検出不良のために失敗した場合は、ヌル

説明

文字列の中で最初に出現する文字（char 型に変換されたもの）を検出します。終端ヌル文字は、文字列の一部と見なされます。

strcmp

string.h

2 つの文字列を比較します。

宣言

```
int strcmp(const char *s1, const char *s2)
```

パラメータ

s1	第 1 文字列のポインタ
s2	第 2 文字列のポインタ

返却値

2 つの文字列を比較した int 結果

返却値	意味
>0	s1 < s2
=0	s1 = s2
<0	s1 < s2

説明

2 つの文字列を比較します。

strcoll

string.h

文字列の比較

宣言

```
int strcoll(const char *s1, const char *s2)
```

パラメータ

s1	第 1 文字列のポインタ
s2	第 2 文字列のポインタ

返却値

2 つの文字列を比較した int 結果

返却値	意味
>0	**s1 < s2**
=0	**s1 = s2**
<0	**s1 < s2**

説明

2 つの文字列を比較します。この関数は、strcmp と同じ動作をします。互換性のためだけに提供されています。

strcpy

string.h

文字列の複写

宣言

```
char *strcpy(char *s1, const char *s2)
```

パラメータ

s1	複写先オブジェクトのポインタ
s2	複写元文字列のポインタ

返却値

s1

説明

文字列をオブジェクトに複写します。

strcspn

string.h

文字列の中の排除文字の長さを求めます。

宣言

```
size_t strcspn(const char *s1, const char *s2)
```

パラメータ

s1	主体文字列のポインタ
s2	目的文字列のポインタ

返却値

s2 で指し示された文字列からの文字以外で構成される、s1 で指し示された文字列の最大先頭セグメントの int 長さ。

説明

目的文字列からの文字以外で構成された主体文字列の最大先頭セグメントを見つけ出します。

strerror

string.h

エラー・メッセージ文字列を出します。

宣言

```
char *strerror(int errnum)
```

パラメータ

errnum	返却するエラー・メッセージ

返却値

strerror は、処理系定義関数です。740 C コンパイラでは、この関数は次の文字列を返します。

返却文字列	他の全ての番号
EZERO	“no error”
EDOM	“domain error”
ERANGE	“range error”
errnum <0		errnum> Max_err_num	“unknown error”
All other numbers	“error No. errnum”

strlen

string.h

文字列の長さ

宣言

```
size_t strlen(const char *s)
```

パラメータ

s　　文字列のポインタ

返却値

文字列の長さを示す型 size_t のオブジェクト

説明

文字列の中で、終端ヌル文字を含まない文字数を見つけ出します。

strncat

string.h

指定数の文字を文字列に連結します。

宣言

```
char *strncat(char *s1, const char *s2, size_t n)
```

パラメータ

s1	複写先文字列のポインタ
s2	複写元文字列のポインタ
n	使用する複写元文字列の文字数

返却値

s1

説明

複写元の文字列から複写先の文字列に n 先頭文字未満の文字を付加します。

strncmp

string.h

指定数の文字を文字列と比較します。

宣言

```
int strncmp(const char *s1, const char *s2, size_t n)
```

パラメータ

s1	第 1 文字列のポインタ
s2	第 2 文字列のポインタ
n	比較するソース文字列の文字数

返却値

2 つの文字列の n 先頭文字未満の文字を比較した int 結果

返却値	意味
>0	**s1 < s2**
=0	**s1 = s2**
<0	**s1 < s2**

説明

2 つの文字列の n 先頭文字未満の文字を比較します。

strncpy

string.h

文字列から指定数の文字を複写します。

宣言

```
char *strncpy(char *s1, const char *s2, size_t n)
```

パラメータ

s1	複写先文字列のポインタ
s2	複写元文字列のポインタ
n	複写する複写元文字列の文字数

返却値

s1

説明

複写元の文字列から複写先のオブジェクトに n 先頭文字未満の文字を複写します。

strpbrk

string.h

文字列の中で、指定文字の 1 つを見つけ出します。

宣言

```
char *strpbrk(const char *s1, const char *s2)
```

パラメータ

s1	主体文字列のポインタ
s2	目的文字列のポインタ

返却値

結果	意味
検出できた	主体文字列の中で、目的文字列のいずれかの文字の最初の出現を示すポインタ
検出できなかった	文字を見つけられなかった場合はヌル

説明

ある文字列の中で第 2 文字列からの文字を検索します。

strrchr

string.h

文字列の右側から文字を見つけ出します。

宣言

```
char *strrchr(const char *s, int c)
```

パラメータ

s	文字列のポインタ
c	文字を表した int

返却値

検出された場合は、s で指し示された文字列の中で最後に出現した c のポインタ

説明

文字列の中で（char 型に変換された）文字の最後の出現を検索します。終端ヌル文字は、文字列の一部と見なされます。

strspn

string.h

文字列の中の文字の長さを求めます。

宣言

```
size_t strspn(const char *s1, const char *s2)
```

パラメータ

s1	主体文字列のポインタ
s2	目的文字列のポインタ

返却値

s2 で指し示された文字列からの文字だけで構成される、s1 で指し示された文字列の最大先頭セグメントの長さ。

説明

完全に目的文字列からの文字で構成された主体文字列の最大先頭セグメントを見つけ出します。

strstr

string.h

サブ文字列を検索します。

宣言

```
char *strstr(const char *s1, const char *s2)
```

パラメータ

s1	主体文字列のポインタ
s2	目的文字列のポインタ

返却値

結果	値
検出できた	s1 で指し示された文字列の中で、s2 で指し示された文字列の（終端ヌル文字を含まない）文字の並びの最初の出現を示すポインタ
検出できなかった	文字列を見つけられなかった場合は、ヌルを返します。s2 が長さゼロの文字列を指し示している場合は、s1 を返します。

説明

ある文字列の中で 2 番目の文字列の出現を検索します。

strtod

stdlib.h

文字列を double へ変換します。

宣言

```
double strtod(const char *nptr, char **endptr)
```

パラメータ

nptr	文字列のポインタ
endptr	文字列のポインタを示すポインタ

返却値

結果	値
変換できた	nptr で指し示された文字列の中の ASCII 表現の浮動小数点定数を変換した double 結果を返し、endptr を定数の後の最初の文字を指し示したままにします。
変換できなかった	ゼロを返し、endptr を最初の非空白文字を示したままにします。

説明

先頭の空白文字を取り去って、数の ASCII 表現を double 表現に変換します。

strtok

string.h

文字列をトークンに分割します。

宣言

```
char *strtok(char *s1, const char *s2)
```

パラメータ

s1	トークンに分割される文字列のポインタ
s2	デリミタの文字列のポインタ

返却値

結果	値
分割できた	トークンのポインタ
分割できなかった	ゼロ

説明

デリミタ s2 の文字列からの 1 つ以上の文字列で分離される、文字列 s1 の中の次のトークンを見つけます。

初めて strtok を呼び出すと、トークンに分割したい文字列は s1 となります。strtok は、この文字列を保存します。後続の各呼出しでは、s1 はヌルとなります。strtok は、それが保存した文字列の中で次のトークンの有無を検索します。s2 は呼出しごとに異なります。

strtok がトークンを発見すると、トークンの中の第 1 文字を示すポインタを返します。それ以外の場合は、ヌルを返します。文字列の最後にトークンがない場合、strtok はデリミタをヌル文字（＼0）で置き換えます。

strtol

stdlib.h

文字列を長整数に変換します。

宣言

```
long int strtol(const char *nptr, char **endptr, int base)
```

パラメータ

nptr	文字列のポインタ
endptr	文字列のポインタを示すポインタ
base	ベースを指定する int 値

返却値

結果	値
変換できた	nptr で指し示された文字列の中の ASCII 表現の整数定数を変換した long int 結果を返し、endptr を定数の後の最初の文字を指し示したままにします。
変換できなかった	ゼロを返し、endptr を最初の非空白文字を示したままにします。

説明

指定されたベースを使用し、かつ先頭の空白文字を取り去って、ASCII 表現の数を long int 表現に変換します。

ベースが 0 の場合、予測される列は一般整数となります。それ以外の場合、予測される列は、基数（2～36 です）を base で指定して整数を表した英数字から構成されます。英字[a, z]、[A, Z]は、10～35 の値に割り当てられます。ベースが 16 の場合、16 進整数の 0x 部分は先頭列でも構いません。

strtoul

stdlib.h

文字列を符号なし長整数に変換します。

宣言

```
unsigned long int strtoul(const char *nptr, char **endptr, base int)
```

パラメータ

nptr	文字列のポインタ
endptr	文字列のポインタを示すポインタ
base	ベースを指定する int 値

返却値

結果	値
変換できた	nptr で指し示された文字列の中の ASCII 表現の整数定数を変換した unsigned long int 結果を返し、endptr を定数の後の最初の文字を指し示したままにします。
変換できなかった	ゼロを返し、endptr を最初の非空白文字を示したままにします。

説明

指定されたベースを使用し、かつ先頭の空白文字を取り去って、ASCII 表現の数を unsigned long int 表現に変換します。

ベースが 0 の場合、予測される列は一般整数となります。それ以外の場合、予測される列は、基数（2～36 です）を base で指定して整数を表した英数字から構成されます。英字[a, z]、[A, Z]は、10～35 の値に割り当てられます。ベースが 16 の場合、16 進整数の 0x 部分は先頭の列でも構いません。

strxfrm

string.h

文字列を変形し、長さを返します。

宣言

```
size_t strxfrm(char *s1, const char *s2, size_t n)
```

パラメータ

s1	変形文字列の戻り位置
s2	変形する文字列
n	s1 に置かれる最大文字数

返却値

終端ヌル文字を含まない変形文字列の長さ

説明

変形は、strcmp 関数が 2 つの変形文字列に適用された場合に、2 つの同じオリジナル文字列に適用された strcoll 関数の結果に対応する値を strcmp 関数が返すように行われます。

tan

math.h

正接

宣言

```
double tan(double arg)
```

パラメータ

arg	ラジアンで表した double 値

返却値

arg の double 正接

説明

arg ラジアンの正接を計算します。

tanh

math.h

双曲線正接

宣言

```
double tanh(double arg)
```

パラメータ

arg　　ラジアンで表した double 値

返却値

arg の double 双曲線正接

説明

arg ラジアンの双曲線正接を計算します。

tolower

ctype.h

小文字化

宣言

```
int tolower(int c)
```

パラメータ

c　　文字の int 表現

返却値

c に対応した小文字の int 表現

説明

文字を小文字に変換します。

toupper

ctype.h

大文字化

宣言

```
int toupper(int c)
```

パラメータ

c	文字の int 表現

返却値

c に対応した大文字の int 表現

説明

文字を大文字に変換します。

va_arg

stdarg.h

関数呼出しの中の次の引数

宣言

```
type va_arg(va_list ap, mode)
```

パラメータ

ap	型 va_list の値
mode	指定された型をもつオブジェクトのポインタの型が、単に type の後に＊を置くことによって得られるような型名

返却値

説明を参照してください。

説明

関数呼出しの中の次の引数の型と値をもつ式に展開されるマクロです。va_start による初期化後、これは parmN によって指定されたものの後にくる引数です。va_arg は、ap を更新して後続の引数を順に配布します。

va_arg と関連マクロの使用例については、printf.c、intwri.c ファイルを参照してください。

va_end

stdarg.h

関数呼出引数の読取りを終了します。

宣言

```
void va_end(va_list ap)
```

パラメータ

ap　　可変個引数リストを指し示す型 va_list のポインタ

返却値

説明を参照してください。

説明

va_list ap を初期化した展開 va_start が参照していた可変個引数リストをもつ関数からの通常のリターンを容易にするマクロです。

va_list

stdarg.h

引数リストの型

宣言

```
char *va_list[1]
```

パラメータ

なし

返却値

説明を参照してください。

説明

va_arg や va_end で要求される情報の保持に適した配列型です。

va_start

stdarg.h

関数呼出引数の読取りを開始します。

宣言

```
void va_start(va_list ap, parmN)
```

パラメータ

ap	可変個引数リストを指し示す型 va_list のポインタ
parmN	関数定義内の変数パラメータ・リストの最右端パラメータの識別子

返却値

説明を参照してください。

説明

va_arg や va_end での使用のために ap を初期化するマクロです。

_formatted_read

icclbutl.h

定様式データの読み込み

宣言

```
int _formatted_read (const char **line, const char **format, va_list ap)
```

パラメータ

line	スキャンするデータのポインタを示すポインタ
format	標準の scanf 形式仕様文字列のポインタを示すポインタ
ap	可変個引数リストを指し示す型 va_list のポインタ

返却値

完了した変換の数

説明

定様式データを読み込みます。この関数は、scanf の基本フォーマッタです。

_formatted_read は、並行して再使用（リエントラント）することはできません。

_formatted_read を使用する場合、stdarg.h ファイルに、上述した特殊な ANSI 定義マクロが要求されます。特に、次の項目が要求されます。

- 型 va_list の変数 ap があること。
- formatted_read を呼び出す前に、va_start の呼出しがあること。
- 現在の文脈を抜ける前に、va_end の呼出しがあること。
- va_start の引数は、可変個引数リストの左に直に接した形式パラメータであること。

_formatted_write

icclbutl.h

データの書式化と書き出し

宣言

```
int _formatted_write (const char *format, void outputf (char, void *),
void *sp, va_list ap)
```

パラメータ

format 標準の printf/sprintf 形式仕様文字列のポインタ

outputf _formatted_write で生成された単一文字を実際に書き出すルーチンの関数ポインタです。この関数に対する第 1 パラメータは実際の文字値を、第 2 パラメータは常に_formatted_writeの第3パラメータと同等の値をもつポインタをもっています。

sp 低水準出力関数が必要とする場合があるデータ構造体の一部の型を示すポインタです。文字値以外に指定の必要がない場合でも、このパラメータは(void*) 0 で指定し、出力関数に定義してください。

ap 可変個引数リストを指し示す型 va_list のポインタ

返却値

書き出された文字数

説明

書き出しデータを書式化します。この関数は、printf や sprintf の基本フォーマッタですが、その汎用インタフェースによって、非標準表示装置への出力に容易に適用させることができます。

完全なフォーマッタは多くの領域を要求しますので、数種類のフォーマッタの中から選択できるようになっています。詳細については、53 ページの第 4 章「構成」を参照してください。

_formatted_write は、並行して再使用（リエントラント）することはできません。

_formatted_write を使用する場合は、stdarg.h ファイルに、上述した特殊な ANSI 定義マクロが要求されます。特に、次の項目が要求されます。

- 型 va_list の変数 ap があること。
- _formatted_write を呼び出す前に、va_start の呼出しがあること。
- 現在の文脈を抜ける前に、va_end の呼出しがあること。
- va_start の引数は、可変個引数リストの左に直に接した形式パラメータであること。

_formatted_write の使用例については、printf.c ファイルを参照してください。

_medium_read

icclbutl.h

浮動小数点数以外の定様式データを読み込みます。

宣言

```
int _medium_read (const char **line, const char **format, va_list ap)
```

パラメータ

line	スキャンするデータのポインタを示すポインタ
format	標準の scanf 形式仕様文字列のポインタを示すポインタ
ap	可変個引数リストを示す型 va_list のポインタ

返却値

完了した変換の数

説明

ハーフサイズで、浮動小数点数をサポートしていない_formatted_read の縮小バージョンです。詳細については、_formatted_read を参照してください。

_medium_write

icclbutl.h

浮動小数点数を除く、定様式データを書き出します。

宣言

```
int _medium_write (const char *format, void outputf (char, void *), void *sp,
va_list ap)
```

パラメータ

format	標準の printf/sprintf 形式仕様文字列のポインタ
outputf	_formatted_write で生成された単一文字を実際に書き出すルーチンの関数ポインタです。この関数に対する第 1 パラメータは実際の文字値を、第 2 パラメータは常に_formatted_writeの第3パラメータと同等の値をもつポインタをもっています。
sp	低水準出力関数が必要とする場合があるデータ構造体の一部の型を示すポインタです。文字値以外に指定の必要がない場合でも、このパラメータは(void*) 0 で指定し、出力関数に定義してください。
ap	可変個引数リストを指し示す型 va_list のポインタ

返却値

書き出された文字数

説明

ハーフサイズで、浮動小数点数をサポートしていない_formatted_write の縮小バージョンです。

詳細については、_formatted_write を参照してください。

_small_write

icclbutl.h

小定様式データ書き出しルーチン

宣言

int _small_write (const char *format, void outputf (char, void *), void *sp, va_list ap)

パラメータ

format	標準の printf/sprintf 形式仕様文字列のポインタ
outputf	_formatted_write で生成された単一文字を実際に書き出すルーチンの関数ポインタです。この関数に対する第 1 パラメータは実際の文字値を、第 2 パラメータは常に_formatted_writeの第3パラメータと同等の値をもつポインタをもっています。
sp	低水準出力関数が必要とする場合があるデータ構造体の一部の型を示すポインタです。文字値以外に指定の必要がない場合でも、このパラメータは(void*) 0 で指定し、出力関数に定義してください。
ap	可変個引数リストを指し示す型 va_list のポインタ

返却値

書き出された文字数

説明

約 1／4 のサイズで、RAM を約 15 バイトしか使用しない_formatted_write の小型バージョンです。

_small_write フォーマッタは、int オブジェクトについて次の指定子のみをサポートします。

%%, %d, %o, %c, %s, %x

フィールド幅や精度引数はサポートしていません。サポートしていない指定子や修飾子が使用された場合、診断は行われません。

詳細については、_formatted_write を参照してください。

第 8 章　言語拡張機能

この章では、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。

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
```

#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
```

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。

組込み関数の要約

組込み（インライン）関数を使用すると、740 マイクロプロセッサを非常に低レベルまで制御することができます。

関数	内容
_args$	パラメータの配列を関数に返却する。
_argt$	パラメータの型を返却する。
break_instruction	BRK 命令を挿入する。
cld_instruction	CLD 命令を挿入する。
disable_interrupt	割込みをオフにする（SEI）。
enable_interrupt	割込みをオンにする（CLI）。
enter_stop_mode	STP 命令を挿入する。
enter_wait_mode	WAIT 命令を実行する。
nop_instruction	NOP 命令を挿入する。

37600 組込み関数の要約

下表の組込み関数は、37600 ファミリ・マイクロプロセッサのみでサポートされています。

関数	内容
void copy_string_from_bank (char *to, char *from);	第 2 バンクから通常メモリに文字列をコピーする。
void copy_memory_from_bank (char *to, char *from, size_t size);	第 2 バンクから通常メモリにメモリの 1 ブロックをコピーする。
char *read_pointer_from_bank (char **p);	第 2 バンクからポインタを読み取る。
void put_string_from_bank (char *p);	第 2 バンクから putchar を介して文字列を書き出す。
char read_byte_from_bank (char *p)	第 2 バンクから単一バイトを読み取る。

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 9 章　拡張キーワードのリファレンス

この章では、拡張キーワードをアルファベット順に説明します。
一部の定義では、次の一般パラメータが使用されています。

パラメータ	意味
storage-class	任意指定のキーワード extern または static を示します。
declarator	標準 C 変数や関数の宣言子を示します。

bit

単一ビット変数を宣言します。

構文－再配置可能アドレス

```
storage-class bit identifier
```

構文－固定アドレス

```
bit identifier = constant-expression-bit-selector
bit identifier = constant-expression
```

構文－SFR

```
bit identifier = sfr-identifier.bit-selector
```

説明

bit 変数とは、記憶域がシングル・ビットの変数のことです。0 と 1 の値のみをもつことができ、ゼロ・ページのみに格納することができます。ビット変数を C 標準ビット・フィールドと混同しないでください。

ビット変数は、次の 3 種類の 1 つになります。

ビット変数の種類	内容
再配置可能アドレス	変数は、通常の再配置可能変数の 1 ビットです。
固定アドレス	変数は、固定アドレスにある記憶場所の 1 ビットです。アドレスは、0〜7 の範囲のビット番号と一緒に 0〜0xFF の範囲のバイト・アドレスとして、あるいは 0〜0x7FF の範囲のビット・アドレスとして指定されます。
sfr	変数は、sfr 変数の 1 ビットです。

ビット変数は、次の場合を除いて、その他の整数型が使用できる場所であればどこでも使用できます。

- 単項&（アドレス）演算子に対するオペランドとして
- 仮関数パラメータとして
- struct/union 要素として
- 関数の返却値として

interrupt

割込み関数を宣言します。

構文

```
storage-class interrupt function-declarator
storage-class interrupt [vector] function-declarator
```

パラメータ

function-declarator	引数をもたない虚空関数の宣言子
[vector]	ベクトル・アドレスをもたらす大カッコで囲まれた定数式

説明

interrupt キーワードは、プロセッサ割込み発生時に呼び出される関数を宣言します。この関数は空で、引数をもっていてはいけません。

ベクトルを指定した場合、この関数を呼び出す割込み処理ルーチンのアドレスは、そのベクトルに挿入されます。ベクトル・アドレスとは、割込みベクトル・ブロックの始まりからのベクトルのオフセットのことです。ベクトルを指定しない場合は、割込み関数のベクトル・テーブル（CSTARTUP モジュールにあると思われます）に適切なエントリを提供してください。

ランタイム割込み処理ルーチンは、プロセッサ・レジスタの保存や復元、RTI 命令を介した復帰を取り扱います。

[vector]を指定して順方向宣言を行うと、ベクトル情報は以降の関数定義に転送されます。

INTVEC セグメントは、リンカ・ファイルの -Z オプションを使用して割込みベクトル部の開始アドレスに位置付けしてください。例えば、次の行は link740l.xcl リンカ・ファイルにあります。

```
-ZINTVEC=FFE0
```

コンパイラは、プログラム自体から割込み関数を呼び出すことを禁止します。コンパイラでは、割込み関数アドレスを、割込み属性をもたない関数ポインタに渡すことができます。この機能は、オペレーティング・システムと一緒に割込み処理ルーチンをインストールする場合に有用です。

例

以下の例では、interupt 修飾子の 3 種類の使用方法を示しています。

次の interrupt 関数宣言には、ベクトル・オフセットとコードの両方が含まれます。

```
interrupt [2] void int2( )      /* INT2 割込み処理ルーチン */
   *.
{
   P0 = 6;
}
```

次の例では、オフセット 8 の割込みが宣言されていますが、コードは含まれていません。

```
interrupt [8] void timer_2int( );      /* 宣言のみ */
```

この例では、前の例で宣言された関数の割込みコードが提供されます。

```
interrupt void timer_2int( );          /* タイマ 2 処理ルーチン */
{
   if (P4 & 1) startup( );
}
```

23 ページの「割込み処理ルーチンの追加」も参照してください。

monitor

関数を最小化します。

構文

```
storage-class moinitor function-declarator
```

説明

monitor キーワードによって、関数の実行中は割込みが不能になります。これによって、複数の処理によるリソースへのアクセスを制御するセマフォの操作などのアトミック・オペレーションが実行されます。

その他の点については、monitor で宣言された関数は通常の関数と変わりません。

例

次の例は、フラグの検査中割込みを不能にします。フラグが設定されていない場合は、設定されます。関数が存在する場合、割込みはそれ以前の状態に設定されます。

```
char printer_free;                  /* プリンタ・フリーのセマフォ */
monitor int got_flag(char *flag)    /* 割込みの危険性がない・・・  */
{
  if (!*flag)                       /* 利用可能な場合は、検査を行う。*/
  {
    return (*flag = 1);             /* yes - 取る */
  {
  return (0);                       /* no - 取らない */
{
```

```
void f (void)
{
  if (got_fla(&printer_free)        /* プリンタが空いているならば実行する。*/
  ...; action code ...
}
```

no_init

不揮発性変数の型修飾子

構文

```
storage=class no_init declarator
```

説明

省略時、コンパイラは変数を揮発性メイン RAM に置き、スタートアップ時にその RAM を初期化します。no_init 型修飾子は、コンパイラに変数を不揮発性 RAM（あるいは EEPROM）に置かせ、スタートアップ時にそれを初期化させないようにします。

no_init 変数宣言は、初期化子を含まないことがあります。

不揮発性メモリが使用された場合は、プログラムをリンクさせて、不揮発性 RAM 領域を参照する必要があります。詳細については、56 ページの「不揮発性 RAM」を参照してください。

例

次の例は、no_init 修飾子の使用の有効化、無効化を示します。

```
no_init int settings[50];          /* 不揮発性設定の配列 */
no_init zpage I ;                  * 型修飾子が矛盾している ― 無効 */
no_init int I = 1 ;                /* 初期化子が含まれている ― 無効 */
```

npage

変数を NPAGE 領域に入れます。

構文

```
npage declarator
```

説明

このキーワードは、変数を、2 バイト・アドレスによってアドレス指定される N_UDATA や N_IDATA セグメントの NPAGE 領域に向けて送信します。

例

次のコードは、NPAGE メモリに 1000 バイトのバッファを生成します。

```
npage char buffer[1000];
```

sfr

1 バイト I/O データ型のオブジェクトを宣言します。

構文

```
sfr identifier = constant-expression
```

説明

sfr は、次の I/O レジスタを示します。

- unsigned char に相当するもの
- 直接アドレス指定できるもの
- 0～0xFF 範囲の固定記憶位置に存在するもの

sfr 変数の値は、アドレス constant-expression にある SFR レジスタのコンテンツです。単項 &（アドレス）演算子を除き、整数型に適用される演算子はすべて、sfr 変数に適用できます。

式の中では、sfr 変数はまた、ピリオドによって付加することができます。ピリオドの後には、ビット選択子を続けます。

例

```
sfr P2 = 0xC;                       /* P2 の定義 */
void func(void)
{
  P2 = 4;                           /* 変数 P2 = 00000100 全体を設定する。*/
  P2,2 = 1;                         /* P2 = xxxxx1xx の 1 ビットのみに作用する。*/
  if (P2 & 4) printf ("ON");        /* P2 全体を読み取り、ビット 2 をマスクする。*/
  if (P2.2) printf ("ON");          /* 同じ効果であるが、ビット・アクセスだけを実行
                                       する。*/
}
```

tiny_func

関数をより小さな呼出し手順で呼び出します。

構文

```
storage-class tiny-func function-declarator
```

説明

tiny_func キーワードは、関数が特殊ページの C_FNT セグメントのエントリを介して呼び出されることを宣言します。これは、呼出しが 3 バイト JSR ではなく、2 バイト JSR であることを意味します。これによって、最終的にコード・サイズが削減されます。

例

次の例は、tiny_func 呼出し手順で test が呼び出されなければならないことを宣言します。testcaller 呼出し関数は、どのような型でも構いません。

```
tiny_func void test(void)
{
  ...
}

void testcaller (void)
{
```

```
  ...
  test( );              /* 2 バイト JSR で"test"を呼び出す。*/
  ...
}
```

zpage

変数を ZPAGE 領域に入れます。

構文

```
zpage declarator
```

説明

このキーワードは、変数を、高速ゼロ・ページ・アドレス指定を使用することができる Z_UDATA や Z_IDATA セグメントの ZPAGE 領域に向けて送信します。

例

次のコードは、ZPAGE メモリに 10 バイト・バッファを生成します。

```
zpage char buffer[10];
```

第 10 章　#PRAGMA 疑似命令のリファレンス

この章では、**#pragma** 疑似命令をアルファベット順に説明します。

bitfields=default

デフォルトのビットフィールドの記憶順序を復元します。

構文

```
#pragma bitfields=default
```

説明

コンパイラに、通常の順序でビットフィールドを割り当てるようにさせます。
bitfields=reversed を参照してください。

bitfields=reversed

ビットフィールドの記憶順序を逆にします。

構文

```
#pragma bitfields=reversed
```

説明

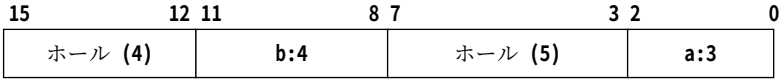
コンパイラに、フィールドの最下位ビットからではなく最上位ビットからビットフィールドを割り当てるようにさせます。ANSI 規格では、記憶順序は処理系に依存できるようになっていますので、このキーワードを使用すれば、移植性の問題を回避することができます。

例

次のメモリ構造体のデフォルトのレイアウトを下図に示します。

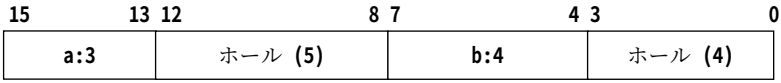
```
struct
{
 short a:3;                  /* a は 3 ビットである。*/
 short :5;                   /*  これによって、ビット 5 のホールが逆になる。*/
 short b:4;                  /* b は 4 ビットである。*/
```

```
} bits;                    /* bits は 16 ビットである。*/
```



比較のために、下図に示すレイアウトをもった構造体を次に示します。

```
#pragma bitfields=reversed
struct
{
 short a:3;                /* a は 3 ビットである。*/
 short :5;                 /* これによって、ビット 5 のホールが逆になる。*/
 short b:4;                /* b は 4 ビットである。*/
} bits;                    /* bits は 16 ビットである。*/
```



codeseg

コード・セグメント名を設定します。

構文

```
#pragma codeseg (seg_name)
```

ここで、seg_name はセグメント名を指定します。セグメント名は、データ・セグメントと矛盾してはいけません。

説明

後続のコードを名前付きセグメントに入れます。この疑似命令は、-R オプションの使用に相当します。48 ページの「コード・セグメント名（-R）」を参照してください。

function=default

関数定義をデフォルトの型に復元します。

構文

```
#pragma function=default
```

説明

関数定義を通常の型に戻します。

例

次の例では、外部関数 f1 を tiny_func 関数として呼び出すのに対して、f3 は通常型とするように指定されます。

```
#pragma function = tiny_func        /* extern tiny_func void f1()と同じ */
extern void f1( );
#pragma function = default          /* デフォルトの関数型 */
extern int f3( );
```

function=interrupt

関数定義を interrupt 型にします。

構文

```
#pragma function=interrupt
```

説明

この疑似命令は、以後の関数定義を interrupt 型にします。これは、関数属性 interrupt を代替するものです。**#pragma function=interrupt** には、ベクトル・オプションがないことに注意してください。

例

次の例は、割込み関数 **process_int**（この関数のアドレスは、INTVEC テーブルに入れてください）を示しています。

```
#pragma function=interrupt
void process_int()                  /* 割込み関数 an interrupt function */
{
      ...
}
#pragma function=default
```

function=intrinsic

関数をインライン・コードシーケンスで置き換えます。

構文

```
#pragma function=intrinsic
```

パラメータ

n　　インライン関数をインライン・コードとして生成する時を決めます。

n	条件
0	C コンパイラの-s オプションで制御される。
1	常にオンになっている。
2	C コンパイラの-s または-z オプションで制御される。
3	C コンパイラの-s オプションで制御される。

48 ページの「実行速度の最適化（-s）」、49 ページの「サイズの最適化（-z）」を参照してください。

説明

特定の C ライブラリ関数に対する呼出しは、コード生成部が実行するインライン・シーケンスで置換することができます。関数の特性が望ましくない場合、これはその特性を変更することができます。この最適化は、次の項目によって制御できます。

- パーサにインライン関数を認識させるようにする#pragma
- 呼出しの前にある ANSI 宣言

呼出しがインラインに書き直されると、コードは関数呼出しではなく別の演算子のように見えます。これは、関数呼出しでしばしば要求される管理を必要としないためです。

175 ページの第 11 章「インライン関数のリファレンス」も参照してください。

例

次のコードは、**string.h** インラインファイルからのものです。

```
#if __TID__ & 0x8000    /* このコンパイラは、インライン関数を認識している */
#pragma function=intrinsic(0)        /* "n" = 0（下記参照）*/
#endif
```

```
/* ここからはインライン関数の宣言です */
extern char *strcpy(char *s1, const char *s32);
extern size_t strlen(const char *s);
#if __TID__ & 0x8000
#pragma function=intrinsic(0)        /*通常の型に戻る */
#endif
```

__TID____の上位ビットは、コンパイラがこの機能をサポートしていることを示しています。

インライン関数が見つけられると、外部宣言は生成されません。しかしながら、ANSI では標準関数の明示的な指定も要求しますので、ライブラリには ” 実 ” 関数も必要です。

```
extern char *strcpy(char *s1, const char *s2);       /* 明示的な宣言 */
char arr[80];
void main(void)
{
       strcpy(arr, “hey”);    /*  ”実 ” 関数の呼び出し */
}
```

また、関数を間接的に参照することもできますので、これがインライン関数で実行された場合は、外部宣言も存在します。しかし、直接呼出しはインライン型となります。

```
#include <string.h>
char (*fp)(char *, const char *) = strcpy;           /* lib の間接参照 */
char arr[80];
{
       strcpy(arr, “hey”);    /* インライン呼出し／コード */
}
```

インライン関数宣言（本体）が見つけ出された場合、これは標準関数として扱われますが、呼出しはインラインのままです。これは、最小（そして高速）ANSI 対応ライブラリの生成に使用されます。

```
#include <string.h>
char *strcpy(char *d, const char *s);      /*  ”実 ” 関数のエントリ */
{
   return (strcpy(d, s));        /* インライン呼出し／コード */
}
```

function=monitor

関数定義をアトミック（割込み禁止）にします。

構文

```
#pragma function=monitor
```

説明

以後の関数定義を **monitor** 型にします。この疑似命令は、関数属性 **monitor** を代替するものです。

例

次の関数 f2 は、割込みを一時的に禁止にして実行されます。

```
#pragma function=monitor
void f2()                          /* f2 関数を monitor 関数にする */
{
   ...
}
#pragma function=default
```

function=tiny_func

関数定義を **tiny_func** 型にします。

構文

```
#pragma function=tiny_func
```

説明

この疑似命令は、以後の関数定義を **tiny_func** 型にします。これは、関数属性 **tiny_func** を代替するものです。

例

次の例は、外部関数 **f1** を **tiny_func** 関数として呼び出すように指定します。

```
#pragma function=tiny_func
extern void f1(void);           /* extern tiny_func f1(void)と同じ */
#pragma function=default
```

language=default

拡張キーワードの使用をデフォルトの状態に復元します。

構文

```
#pragma language=default
```

説明

拡張キーワードの利用度を C コンパイラの -e オプションで設定されたデフォルトの状態に戻します。**language=extended** を参照してください。

例

次の **language=extended** の例を参照してください。

language=extended

拡張キーワードを使用可能にします。

構文

```
#pragma language=extended
```

説明

C コンパイラの **-e** オプションの状態にかかわらず、拡張キーワードを使用可能にします。40 ページの「ターゲット依存拡張機能の使用可能化（**-e**）」を参照してください。

例

次の例では、**zpage** 拡張言語修飾子が変数 **ccount** の定義のために使用可能になっています。**mycount** は、標準的な方法で定義されます。

```
#pragma language=extended
zpage int ccount;                    /*単一バイト・アドレス指定の使用 */
#pragma language=default
int mycount;
```

memory=constseg

定数を名前付きセグメントに配置します。

構文

```
#pragma memory=constseg (seg_name)
```

説明

定数を名前付きセグメントに配置します。この疑似命令は、メモリ属性キーワードを代替するものです。この疑似命令で指定された定数セグメント名のデフォルト値は、メモリ属性拡張キーワードで上書きすることができます。

セグメントは、コンパイラの予約セグメント名の 1 つであってはいけません。

例

次の例は、定数配列 **arr** を ROM セグメント TABLE に入れます。

```
#pragma memory=constseg(TABLE)
char ar[ ] = {6, 9, 2, -5, 0};
#pragma memory = default
```

別のモジュールが定数配列をアクセスする場合、そのモジュールは同等の宣言を使用しなければなりません。

```
pragma memory=constseg(TABLE)
extern char * arr;
```

memory=dataseg

変数を名前付きセグメントに配置します。

構文

```
#pragma memory=dataseg (seg_name)
```

説明

デフォルトで変数を名前付きセグメントに配置します。この疑似命令で指定された変数セグメント名のデフォルト値は、メモリ属性拡張キーワードで上書きすることができます。

変数定義には、初期値を提供することはできません。指定モジュールには、最大 10 種類の代替データ・セグメントを定義できます。事前に定義したものであれば、プログラムのどの時点でも、どのデータ・セグメント名にも切り替えることができます。

例

次の例は、USART と言われる読込み／書込み領域に 3 つの変数を入れます。

```
#pragma memory = dataseg(USART)
char USART_data;       /* オフセット 0offset 0 */
char USART_control; /* オフセット 1offset 1 */
int USART_rate; /* オフセット 2、3offset 2, 3 */
#pragma memory = default
```

別のモジュールがこれらのシンボルをアクセスしたい場合は、同等の **extern** 宣言を使用しなければなりません。

```
#pragma memory=detaseg(USART)
extern char USART_data;
```

memory=default

オブジェクトのメモリ割当をデフォルトの領域に復元します。

構文

```
#pragma memory=default
```

説明

オブジェクトのメモリ割当を、使用中のメモリ・モデルで指定されるように、デフォルトの領域に復元します。

memory=no_init

変数を NO_INIT セグメントに配置します。

構文

```
#pragma memory=no_init
```

説明

変数を NO_INIT セグメントに配置し、変数が初期化されずに、不揮発性 RAM に存在するようにします。この疑似命令は、メモリ属性 no_init を代替するものです。デフォルトの属性は、このメモリ属性で上書きすることができます。

NO_INIT セグメントは、不揮発性 RAM の物理アドレスと一致するようにリンクします。詳細は、53 ページの第 4 章「構成」を参照してください。

例

次の例は、変数 buffer を非初期化メモリに入れます。変数 i と j は、DATA 領域に置かれます。

```
#pragma memory=no_init
char buffer[1000];    /* 非初期化メモリにある。 */]
#pragma memory=default
int i, j;             /* デフォルトのメモリ・タイプ */
```

非デフォルト・メモリ**#pragma** の状態で関数宣言が検出された場合、エラー・メッセージを発生しますので、関数宣言の前に必ず**#pragma memory=default** 設定を行ってください。これは、ローカル変数やパラメータは、そのデフォルトのセグメント、すなわちスタック以外のセグメントに存在することはできないからです。

memory=npage

変数をゼロページの外側に配置します。

構文

```
#pragma memory=zpage
```

説明

デフォルトで変数をゼロページの外側に配置し、ゼロページ領域を節約します。デフォルト状態はこのメモリ属性で上書きすることができます。

例

次の例は、変数 buffer と d をゼロページの外側に配置します。strings の no_init 属性は、それを強制的に n ページではなく NO_INIT メモリに入れます。

```
#pragma memory=zpage
int buffer[100];      /* ゼロページの外側 */
extern double d;      /* ゼロページの外側 */
no_init char *strings[5];       /* no_init で上書き */
#pragma memory=default
int i;      /* デフォルトのメモリ・タイプ*/
```

memory=zpage

変数をゼロページ（00-FF）領域に配置します。

構文

```
#pragma memory=zpage
```

説明

変数をゼロページ領域（高速ゼロページアドレス指定を使用します）に配置します。デフォルト状態はこのメモリ属性で上書きすることができます。

例

次の例は、変数 buffer と d をゼロページメモリに入れます。strings の no_init 属性は、それを強制的にゼロページではなく NO_INIT メモリに入れます。

```
#pragma memory=zpage
int buffer[10];       /* 0-FF メモリ */
extern double d;      /* 0-FF メモリ */
no_init char *strings[5];       /* no_init で上書き */
#pragma memory=default
int i;      /* デフォルトのメモリ・タイプ*/
```

warnings=default

コンパイラ警告出力をデフォルトの状態に復元します。

構文

```
#pragma warnings=default
```

説明

コンパイラ警告メッセージを C コンパイラの -w オプションで設定されたデフォルトの状態に復元します。#pragma warnings=on、#pragma warnings=off を参照してください。

warnings=off

コンパイラの警告出力をオフにします。

構文

```
#pragma warnings=off
```

説明

C コンパイラの -w オプションの状態にかかわらず、コンパイラ警告メッセージの出力を不能にします。39 ページの「警告の不能化（-w）」参照してください。

warnings=on

コンパイラの警告出力をオンにします。

構文

```
#pragma warnings=on
```

説明

C コンパイラの -w オプションの状態にかかわらず、コンパイラ警告メッセージの出力を使用可能にします。39 ページの「警告の不能化（-w）」参照してください。

第 11 章　定義済みシンボルのリファレンス

この章では、コンパイラで事前に定義されたシンボルに関するリファレンス情報を解説します。

__DATE__

現在の日付

構文

```
__DATE__
```

説明

コンパイルの日付は、`Mmm dd yyyy` 形式で返却されます。

__FILE__

現在のソース・ファイル名

構文

```
__FILE__
```

説明

現在コンパイル中のファイル名を返却します。

__IAR_SYSTEMS_ICC__

IAR C コンパイラ識別子

構文

```
__IAR_SYSTEMS_ICC__
```

説明

番号 1 が返却されます。このシンボルは`#ifdef` で検査され、IAR Systmes C コンパイラによってコンパイルされているかどうか検出できます。

__LINE__

現在のソース行番号

構文

```
__LINE__
```

説明

現在コンパイル中のファイルの現在の行番号を返却します。

__STDC__

IAR C コンパイラ識別子

構文

```
__STDC__
```

説明

番号 1 が返却されます。このシンボルは**#ifdef** で検査され、IAR Systmes C コンパイラによってコンパイルされているかどうか検出できます。

__TID__

ターゲット識別子

構文

```
__TID__
```

説明

ターゲット識別子は、各 IAR SystemsC コンパイラに一意（すなわち、各目的プロセッサに一意）の番号、組込みフラグ、**-v** オプションの値、そして**-m** オプションに対応した値をもっています。

31　16	15	14　8	7　4	3　0
（未使用）	インライン コード サポート	各ターゲットプロセッサの識別子値 **_IDENT**	**-v** オプション値がサポートされている場合は、その値	**-m** オプション値がサポートされている場合は、その値

740 C コンパイラについては、**__TID__**値は、次のように構成されています。

```
(0x8000 | (t << 8) | (v << 4) |m)
```

値は、次のように抽出できます。

```
f = (__TID__) & 0x8000;
t = (__TID__>> 8) & 0x7F;
v = (__TID__>> 4) & 0xF;
m = __TID__ & 0x0F;
```

マクロ名の各終りには、2 本の下線が引かれているので注意してください。

カレント・コンパイラのターゲット**_IDENT** の値を見つけるには、次のように実行します。

```
printf("%ld",(__TID__>>8)&0x7F)
```

__TID__の使用例については、**stdarg.h** ファイルを参照してください。

最上位ビット 0x8000 は 740 C コンパイラに設定され、コンパイラが組込み関数を認識することを示します。これは、ヘッダー・ファイルの記述方法に影響を与えることがあります。162 ページの「**function=intrinsic**」を参照してください。

__TIME__

現在の時間

構文

```
__TIME__
```

説明

コンパイルの時間は、**hh:mm:ss** 形式で返却されます。

__VER__

コンパイラのバージョン番号を返します。

構文

```
__TIME__
```

説明

コンパイラのバージョン番号を **int** として返します。

例

次の例は、バージョン 3.34 のメッセージを出力します。

```
#if __VER__ ==334
#message “Compiler version 3.34”
#endif
```

第 12 章　インライン関数のリファレンス

この章では、インライン関数に関するリファレンス情報を説明します。

_args$

パラメータの配列を関数に返します。

構文

_args$

説明

args$は、現在の関数の仮パラメータの記述リストを含んだ char 配列（char *）を返却する予約語です。

オフセット	内容
0	_argt$形式のパラメータ 1 の型
1	バイト単位のパラメータ 1 のサイズ
2	_argt$形式のパラメータ 2 の型
3	バイト単位のパラメータ 2 のサイズ
2n-2	_argt$形式のパラメータ n の型
2n-1	バイト単位のパラメータ n のサイズ
2n	¥0

127 を超えるサイズは、127 として通知されます。

_args$**は、関数定義内でのみ使用することができます。**_args$の使用例については、**stdarg.h** ファイルを参照してください。

変数長さ（**varargs**）パラメータ・リストが指定された場合、パラメータ・リストは最終明示パラメータで終わるものと見なされます。したがって、任意指定のパラメータの型やサイズを安易に決定することはできません。

_argt$

パラメータの型を返します。

構文

_argt$ (v)

説明

下表に、返却値およびそれに対応する意味を示します。

値	型	値	型
1	unsigned char	8	long
2	char	9	float
3	unsigned short	10	double
4	short	11	long double
5	unsigned int	12	pointer/address
6	int	13	union
7	unsigned long	14	struct

例

次の例は **_argt$**を使用して **integer** パラメータ型か、**long** パラメータ型かを検査します。

```
switch (_argt$(i))
{
          case 6:
                    printf("int %d\n", i);
                    break;
          case 8:
                    printf("long %ld\n", i);
                    break;
          default:
                    printf("int or long expected\n");
                    break;
}
```

break_instruction

BRK 命令を出します。

構文

```
void break_instruction (void)
```

説明

この関数は、プログラム・コードに BRK 命令、続いて NOP 命令を挿入して、BRK ベクトルを呼び出します。

BRK 命令は PC + 2 をスタックに押し込みますので、ブレーク・ハンドラの終りにある RTI は制御を NOP に続く命令に返します。したがって、NOP は実行されません。

cld_instruction

CLD 命令を出します。

構文

```
void cld_instruction (void)
```

説明

アプリケーション内で 10 進モードを使用した場合、この関数をインラインルーチン内の最初の命令として使用し、10 進フラグが割込みの中で誤りのないようにしてください。

disable_interrupt

SEI 命令を出します。

構文

```
void disable_interrupt (void)
```

説明

この関数は、SEI 命令をプログラム・コードに挿入することによって割込みを不能にします。

enable_interrupt

CLI 命令を出します。

構文

```
void enable_interrupt (void)
```

説明

この関数は、CLI 命令をプログラム・コードに挿入することによって割込みを使用可能にします。

enter_stop_mode

STP 命令を出します。

構文

```
void enter_stop_mode (void)
```

説明

この関数は、STP 命令をプログラム・コードに挿入することによって割込みを待ちます。

enter_wait_mode

WIT 命令を出します。

構文

```
void enter_wait_mode (void)
```

説明

この関数は、WIT 命令をプログラム・コードに挿入することによって割込みを待ちます。

nop_instruction

NOP 命令を出します。

構文

```
void nop_instruction (void)
```

説明

この関数は、プログラム・コードに NOP 命令を挿入します。

第 13 章　37600 インライン関数のリファレンス

この章では、37600 ファミリ・マイクロプロセッサでのみ利用できるインライン関数に関するリファレンス情報を説明します。

copy_memory_from_bank

第 2 バンクから正規メモリにメモリの 1 ブロックをコピーします。

構文

```
void copy_memory_from_bank (char *to, char *from, size_t size)
```

説明

size バイトは、第 2 バンクの 16 ビット・オフセット **from** から正規メモリの目的アドレス **to** にコピーされます。

copy_string_from_bank

第 2 バンクから正規メモリに文字列をコピーします。

構文

```
void copy_string_from_bank (char *to, char *from)
```

説明

ヌル終了文字列は、第 2 バンクの 16 ビット・オフセット **from** から正規メモリの目的アドレス **to** にコピーされます。

read_byte_from_bank

第 2 バンクからバイトを読み取ります。

構文

```
char read_byte_from_bank (char *p)
```

説明

第 2 バンクの 16 ビット・オフセット **p** から読み取ったバイトを返却します。

read_pointer_from_bank

第 2 バンクからポインタを読み取ります。

構文

```
char *read_pointer_from_bank (char **p)
```

説明

第 2 バンクの 16 ビット・オフセット **p** から読み取ったポインタを返却します。

put_string_from_bank

第 2 バンクからの文字列を putchar を介して書き出します。

構文

```
void put_string_from_bank (char *p)
```

説明

第 2 バンクの 16 ビット・オフセット **p** にある文字列を **putchar** を介して標準出力に書き出します。

第 14 章　アセンブリ言語インタフェース

740 C コンパイラを使用すると、アセンブリ言語モジュールと、コンパイルされた C モジュールを組み合わせることができます。これは、特に、アセンブリ言語で記述され、C メイン・プログラムから呼び出される小さなタイム・クリティカル・ルーチンに使用されます。この章では、C メイン・プログラムとアセンブリ言語ルーチン間のインタフェースについて記述します。

シェルの生成

アセンブリ言語ルーチンを適切なインタフェースで生成する場合、C コンパイラで生成したアセンブリ言語ソースから始める方法を推奨します。このシェルに対して、ルーチンの関数本体を容易に追加することができます。

シェル・ソースは必要な変数を宣言し、その変数への簡単なアクセスを実行すればよいだけです。例えば、次のとおりです。

```
char assem(char pc1,char pc2,zpage int pi1,int pi2)
{
          int my_a;
          int my_b;
          my_a=pc1;
          my_b=pc2;
          return (pi1+pi2);
}
void main(void)
{
          int main_x=255;
          assem('x','y',main_x,2);
}
```

次に、このプログラムは次のようにコンパイルされます。

```
icc740 shell -A -q -L -e -ml
```

-A オプションはアセンブリ言語出力を行い、**-q** オプションは C ソース行をアセンブラ・コメントとして取込み、**-L** オプションはリストを生成します。

コンパイルの結果、宣言、関数呼出し、関数返却、変数アクセス、そしてリスト・ファイル **shell.lst** をもつアセンブラ・ソース **shell.s31** が生成されます。

以降の項で、アセンブリ言語インタフェースを詳細に説明します。

呼出し規則

C コンパイラは、各関数のパラメータやローカル変数の記憶に最大 4 ブロックのメモリを使用します。これらの 4 ブロックは、パラメータをゼロページに、パラメータを正規メモリに、ローカル変数をゼロページに、そしてローカル変数を正規メモリに保持します。どの関数が相互に呼び出し合っているかを検査することによって、リンカは可能ならばこれらのブロックをオーバレイします。

関数の宣言

関数は、宣言された関数で呼び出された全ての関数のリストを取り込んだ **DEFFN** アセンブラ疑似命令で宣言されなければなりません。このリストは、リンカに、どの関数が直接、あるいは間接に再帰的なものであるか、どの関数が割込み関数で呼び出されるか、そしてどのパラメータやローカル変数ブロックをオーバレイできるのかを判断させるものです。詳細については、「アセンブラ・サポート演算子、疑似命令」を参照してください。

パラメータの渡し

パラメータは、容量が十分に小さければ A レジスタに渡される第 1 パラメータを除いて、静的メモリに渡されます。特定の関数のパラメータ領域のベースを見つけるには、ゼロページや正規パラメータではそれぞれ **PRMBZ** 演算子や **PRMBN** 演算子を使用します。詳細については、188 ページの「PRMBZ および PRMBN」を参照してください。

関数の呼出し

呼出しのアドレスを見つけて関数を実行するには、**REFEN** および **REFFNT** アセンブラ疑似命令を使用します。関数の始まりを直接呼び出すべきではありません。それは、リンカが、代わって呼び出されなければならない“ ラッパ ”関数を取り込んでいることがあるからです。

呼び出された関数は、A や Y レジスタ、Z、N および C フラグのエントリ値を破壊することがあります。他の全てのレジスタ、フラグ、システム・ポインタは保護しておいてください。

ローカル変数

特定の関数のローカル変数記憶域のベースを見つけるには、ゼロページや正規変数ではそれぞれ **LOCBZ** 演算子や **LOCBN** 演算子を使用します。詳細については、「740 アセンブラ、リンカ、ライブラリアン・プログラミング・ガイド」を参照してください。

返却値

返却値は、容量が十分に小さければ A レジスタに渡されます。そうでない場合は、式スタックで渡されます。式スタックの上部は、C コード実行中、X レジスタで指し示されています。

再帰的関数

再帰的関数とは、それ自体を直接的に、あるいは間接的に呼び出すことができる関数のことです。

パラメータやローカル変数はメモリの固定領域に保存されていますので、再帰的関数の呼出しはいずれも、まずはじめにパラメータやローカル変数ブロックのカレント・コンテンツをソフトウェア・スタックに格納し、関数が完了したらそれらを復元するようにしなければなりません。

リンカは、リンク時にどの関数が実際に再帰的かを判断し、ユーザに代わって格納、呼出し、復元を実行する "ラッパー "関数を取り込みます。このようなラッパー関数が取り込まれると、リンカは全てのリファレンスを **REFFN functionname** に設定し、代わりにラッパー・エントリ点を指し示します。

リエントラント関数

C コンパイラは、リエントラント関数をサポートしていません。

リエントラント関数とは、パラメータが書き込まれている間に同じ関数に対する呼出しによって割り込まれることのある関数をいいます。

例えば、関数 **foo** が実行可能な状態にある間に、割込みや別のタスクによってこの関数を再度呼び出すことができる場合、この関数はリエントランス関数となります。

次に、リエントラントになっている関数（**foo**）の例を示します。それは、この関数がそのパラメータ・リストから再度呼び出されているからです。

```
void bar(void)
{

        foo(1, 2);  /* この呼出しは、...を破壊する */
}
void main(void)
{
        foo(bar(), 3);          /* 値は"3" */
}
```

アセンブラ・インタフェース・キーワードの使用法

関数のアドレスを生成､参照するために使用されるアセンブラ疑似命令や演算子が 5 種類用意されています。

次章「アセンブラ・インタフェース・キーワードのリファレンス」では、これらの疑似命令や演算子に関するリファレンス情報を説明します。

DEFFN

DEFFN とは関数定義疑似命令のことであり、次の 3 種類のパラメータをとります。

- 関数名
- 使用するデータ領域のバイト数を指定する次の 8 つのサイズ

  ローカル ZPAGE
  未使用
  ローカル NPAGE
  未使用
  ZPAGE パラメータ、およびマーカ値 0x8000
  未使用
  NPAGE パラメータ
  未使用
- 定義済み関数で呼び出される関数のリスト

ローカル ZPAGE サイズの上位バイトは、次のように関数の使用状況を識別するためのフラグを 1 セットもっています。

ビット	関数の使用状況
0	間接呼出しを行う。
1	割込み
2	オーバレイ不可
3−7	未使用

外部関数については、ローカルサイズは省略されます。よって、サイズのリストは ZPAGE パラメータ・サイズから始まります。

REFFN および REFFNT

REFFN、**REFFNT** は、関数名をオペランドとみなし、関数を入力するために呼び出すアドレスを返す演算子です。**REFFN** は正規関数の呼出しに、一方、**REFFNT** は **tiny_func** 関数の呼出しに使用されます。

IFREF

IFREF は、関数名をオペランドとみなし、関数への（ポインタを介した）間接呼出しに使用するアドレスを返す演算子です。

アドレスの下位バイトや上位バイトを提供するには、**LOW**、**HIGH** アセンブラ演算子を **IFREF** と一緒に使用します。以下の例は、アドレスの上位バイトをアキュムレータに移します。

```
        LDA     #HIGH IFREF my_func
```

LOCBZ および LOCBN

LOCBZ、**LOCBN** は、関数名をオペランドとみなし、関数のオートとパラメータをもつローカル変数記憶ブロックの開始アドレスを返す演算子です。

LOCBZ は **ZPAGE** ブロックの 8 ビット・アドレスを返すのに対し、**LOCBN** は **NPAGE** ブロックの 16 ビット・アドレスを返します。

アドレスの下位バイトや上位バイトを提供するには、**LOW**、**HIGH** アセンブラ演算子を **LOCB** 演算子と一緒に使用します。

PRMBZ および PRMBN

PRMBZ、**PRMBN** は、関数名をオペランドとみなし、パラメータ・メモリ・ブロックの開始アドレスを返す演算子です。

PRMBZ は ZPAGE ブロックの 8 ビット・アドレスを返すのに対し、**PRMBN** は NPAGE ブロックの 16 ビット・アドレスを返します。

アドレスの下位バイトや上位バイトを提供するには、**LOW**、**HIGH** アセンブラ演算子を **PRMB** 演算子と一緒に使用します。

例

shell プログラムは、次のアセンブラ・ソースを **shell.s31** として生成します。このファイルは、一部のアセンブリ言語インタフェースの機能を説明します。

```
        NAME    shell(18)
        RSEG    CODE(0)
        PUBLIC  assem
        DEFFN   assem(0,0,4,0,32770,0,4,0)
        PUBLIC  main
        DEFFN   main(0,0,2,0,32768,0,0,0),assem
```

DEFFN アセンブラ疑似命令は、このモジュールで 2 つの関数を宣言するために使用されます。3 種類の引数は、次のとおりです。

- 関数名。この場合は、**assem** と **main** です。
- 使用するデータ領域のバイト数を指定する 8 つのサイズは、次のとおりです。

  ローカル NPAGE（**assem** のための 2 つの **int** と **main** のための **1 つの int**）

  ZPAGE パラメータ（**assem** のための **1 つの int**）

  NPAGE パラメータ（**assem** のための 1 つの int と 2 つの **char**）
- 定義済み関数で呼び出される関数。この例では、**main** は **assem** を呼び出し、**assem** は呼出しを行いません。

```
        EXTERN  ?CL740MDL_1_00_L00
```

この宣言は、コンパイラ・ライブラリのバージョン・チェックです。

```
        RSEG CODE
; 1.    char assem(char pc1,char pc2,zpage int pi1,int pi2)
; 2.    {
assem:
        STA np:LOCBN assem+4
```

A レジスタに渡された **pc1** パラメータが格納されます。

```
; 3.       int my_a;
; 4.       int my_b;
; 5.       my_a=pc1;
          LDA       np:LOCBN assem+4
          STA       np:LOCBN assem
          LDA       #0
          STA       np:LOCBN assem+1
; 6.       my_b=pc2;
          LDA       np:LOCBN assem+5
          STA       np:LOCBN assem+2
          LDA       #0
          STA       np:LOCBN assem+3
```

ゼロ拡張文字パラメータは **LOCBN** 演算子で参照され、ローカル変数に割り付けられます。また、**LOCBN** 演算子によっても参照されます。コンパイラは、割当オフセットを使用して各パラメータや変数をアクセスしていることに注意してください。

これらのメモリ・アクセスは静的変数を参照しますが、これらの変数はリンカによってオーバレイされることがあります。関数 **other_func** の実行中には、関数のローカル変数は必要ない、すなわち **assem** を呼び出すことができない、あるいは既に終了しているとリンカが判断した場合、**other_func** は **assem** のローカル変数領域を自己のローカル変数のために再使用することができます。

```
; 7.       return (pi1+pi2);
          LDA       np:LOCBN assem+6
          CLC
          ADC       zp:LOCBZ assem
; 8.       }
          RTS
```

A レジスタは戻り値に設定され、関数は終了します。

```
; 9.
; 10.      void main(void)
; 11.      {
main:
; 12.      int main_x=255;
          LDY       #255
          STY       np:LOCBN main
          LDY       #0
          STY       np:LOCBN main+1
```

main のローカル・データは初期化されます。

```
; 13.      assem('x','y',main_x,2);
           LDA       #2
           STA       np:PRMBN assem+2
           LDA       #0
           STA       np:PRMBN assem+3
```

パラメータ **pi2** の **assem** のパラメータ領域は、2 に設定されます。

```
           LDA       np:LOCBN main
           STA       zp:PRMBZ assem
           LDA       np:LOCBN main+1
           STA       zp:PRMBZ assem+1
```

assem のゼロページの **tiny pil** パラメータは、**main** の **main_x** に設定されます。

```
           LDA       #121
           STA       np:PRMBN assem+1
```

assem の **pc2** パラメータは、文字定数 **y** に設定されます。

```
           LDA    #120
```

第 1 パラメータは十分に A レジスタに適合できる容量であるため、A レジスタに渡されます。

```
           JSR       np:REFFN assem
```

assem が呼び出されます。**assem** は呼出しを行いませんので、再帰的関数と判断されることは絶対にありません。しかしながら、リンカが **assem** のより複雑なバージョンを再帰的と判断した場合、呼出しは、リンカが提供するラッパーを介して進み、再帰的スタックへの古いローカル変数のコピーを処理します。

```
; 14.   }
       RTS
       END
```

第 15 章　アセンブラ・インタフェース・キーワードのリファレンス

この章では、C 言語へのインタフェースを提供するアセンブラ演算子と疑似命令を解説します。

アセンブラ・インタフェース・キーワードの要約

DEFFN name...	関数を宣言します。
IFREF name	名前付き関数に対する間接呼出しのエントリ・ポイント
LOCBN name	名前付き関数の NPAGE ローカル変数とパラメータの開始アドレス
LOCBZ name	名前付き関数のゼロ・ページ・ローカル変数とパラメータの開始アドレス
PRMBN name	名前付き関数の NPAGE パラメータ・ブロックの開始アドレス
PRMBZ name	名前付き関数のゼロ・ページ・パラメータ・ブロックの開始アドレス
REFFN name	名前付き関数のエントリ・ポイント
REFFNT name	名前付き tiny_func 関数のエントリ・ポイント

DEFFN

関数を宣言します。

構文

```
DEFFN name([autos ZPAGE,0,autos NPAGE,0],ZPAGE
parameters,0,NPAGE parameters,0)[,functionlist]
```

パラメータ

name	関数名
autos ZPAGE	関数のゼロ・ページ・ローカル変数とパラメータ記憶域のサイズ
autos NPAGE	関数の通常にアドレス指定されるローカル変数とパラメータ記憶域のサイズ
ZPAGE parameters	関数のゼロ・ページ・パラメータ・ブロックのサイズおよびマーカ値 0x8000
NPAGE parameters	関数の通常にアドレス指定されるパラメータ・ブロックのサイズ
functionlist	名前付き関数で呼び出される関数をカンマで区切ったリスト

説明

C から呼び出される関数の定義

C の環境内から呼び出される関数を宣言するには、**DEFFN** を使用します。また、C プログラムから関数を参照するには、関数名を **PUBLIC** で宣言することが必要です。詳細については、本ガイド書の第 14 章「アセンブリ言語インタフェース」を参照してください。

ローカル ZPAGE サイズの上位バイトは、次のように関数の使用状況を識別するためのフラグを 1 セットもっています。

ビット	関数の使用状況
0	間接呼出しを行う。
1	割込み
2	オーバレイ不可
3−7	未使用

外部関数については、ローカルサイズは省略されます。よって、サイズのリストは ZPAGE パラメータ・サイズから始まります。

例

C 定義

```
char assem(char pc1,char pc2,zpage int pi1,int pi2)
{
          int my_a;
          int my_b;
          my_a = pc1;
          my_b = pc2;
          foo();
          return (pi1+pi2);
}
```

は、次のアセンブラ関数宣言を生成します。

```
          DEFFN     assem(0,0,4,0,32770,0,4,0),foo
```

この宣言は、関数 **assem** が正規メモリに 4 バイトのローカル変数記憶域を、ゼロページに 2 バイトのパラメータ記憶域を、また正規メモリに 4 バイトのパラメータ記憶域を必要とすることを示しています。また、関数 **assem** が関数 **foo** を呼び出すことも宣言しています。

C 宣言

```
extern void foo(void)
```

は、次のアセンブラ関数宣言を生成します。

```
          DEFFN foo(32768,0,0,0)
```

この宣言は、外部関数 **foo** がパラメータ記憶域を必要としないことを示しています。また関数 foo は、次の別の疑似命令で **EXTERN** として宣言されることにも注意してください。

```
          EXTERN foo
```

foo への外部呼出しが関数呼出しのパラメータをもっている場合、この呼出しは DEFFN 疑似命令にも加えられます。例えば、

```
          foo(bar(),1)
```

は、次のようになります。

```
          DEFFN foo(numbers),bar
```

IFREF

名前付き関数に対する間接呼出しのエントリ・ポイント

構文

```
IFREF name
```

説明

IFREF は、関数名をオペランドとみなし、関数への（ポインタを介した）間接呼出しに使用するアドレスを返す演算子です。

例

アドレスの下位バイトや上位バイトを提供するには、IFREF と一緒に LOW、HIGH アセンブラ演算子を使用します。次の例は、アドレスの上位バイトをアキュムレータに移します。

```
        LDA     #HIGH IFREF my_func
```

LOCBN

名前付き関数の NPAGE ローカル変数（オート変数の後には、パラメータが続きます）の開始アドレス

構文

```
LOCBN name
```

説明

LOCBN は関数名をオペランドとみなし、関数のオート変数とパラメータをもつローカル変数記憶ブロックの 16 ビット開始アドレスを返します。

例

次の例は、関数 foo のローカル変数記憶域のオフセット 4 からデータをロードします。

```
        LDA     LOCBN foo +4
```

アドレスの下位バイトや上位バイトを提供するには、**LOW**、**HIGH** アセンブラ演算子を **LOCBN** 演算子と一緒に使用します。

次の例は、**foo** 関数のローカル変数記憶ブロック内のオフセット 2 のアドレスの上位バイトをロードします。

```
LDA     #HIGH (LOCBN foo +2)
```

LOCBZ

名前付き関数のゼロ・ページ・ローカル変数（オート変数の後にパラメータが続きます）の開始アドレス

構文

```
LOCBZ name
```

説明

LOCBZ は関数名をオペランドとみなし、関数のオート変数とパラメータをもつローカル変数ゼロページ記憶ブロックの 8 ビット開始アドレスを返します。

例

次の例は、関数 **foo** のゼロ・ページ・ローカル変数記憶域のオフセット 4 からデータをロードします。

```
LDA     LOCBZ foo+4
```

PRMBN

名前付き関数の NPAGE パラメータ・ブロックの開始アドレス

構文

```
PRMBN name
```

説明

PRMBN は関数名をオペランドとみなし、パラメータ・ブロックの 16 ビット開始アドレスを返します。パラメータは、関数の呼出し前にこのブロックに書き込まれなければなりません。

例

次の例は、関数 **foo** のパラメータ・ブロックのオフセット 1 にパラメータを設定します。

```
STA     PRMBN foo+1
```

PRMBZ

名前付き関数のゼロ・ページ・パラメータ・ブロックの開始アドレス

構文

```
PRMBZ name
```

説明

PRMBZ は関数名をオペランドとみなし、ゼロ・ページ・パラメータ・ブロックの 8 ビット開始アドレスを返します。**zpage** パラメータは、関数の呼出し前にこのブロックに書き込まれなければなりません。

例

次の例は、関数 **foo** のゼロ・ページ・パラメータ・ブロックのオフセット 3 にパラメータを設定します。

```
        STA     PRMBZ foo+3
```

REFFN

名前付き関数のエントリ・ポイント

構文

```
REFFN name
```

説明

REFFN は関数名をオペランドとみなし、関数を入力するために呼び出すアドレスを返します。

例

関数 **foo** を呼び出すには、次のようにします。

```
        JSR     REFFN foo
```

REFFNT

名前付き **tiny_func** 関数のエントリ・ポイント

構文

REFFNT name

説明

REFFNT は **tiny_func** 関数名をオペランドとみなし、関数を入力するために呼び出す特殊ページ・アドレスを返します。

例

tiny_func 関数 **foo** を呼び出すには、次のようにします。

```
        JSR     \REFFNT foo
```

第 16 章　セグメントのリファレンス

740 C コンパイラは、XLINK が参照する名前付きセグメントにコードやデータを入れます。セグメントの詳細はアセンブリ言語モジュールのプログラミングで要求され、コンパイラのアセンブリ言語出力を解釈する場合にも有用です。

この章では、セグメントをアルファベット順に紹介します。各セグメントについては、次の項目を説明します。

- セグメント名
- 内容の概説
- セグメントが読み書き用か、あるいは読出し専用か。
- セグメントがアセンブリ言語からアクセスできるのか（アセンブリ・アクセス可能）、あるいはコンパイラのみからアクセスできるのか。
- セグメントのコンテンツと使用例の詳細説明

記憶域割当図

次ページの記憶域割当図は、740 マイクロプロセッサの記憶域割当と各記憶域内の ROM、RAM セグメントの割当を示しています。

上位	**INTVEC**	割込みベクトル・テーブル
	C_FNT	特殊ページ・ジャンプ・テーブル
	C_ICALL	間接関数呼出しのテーブル
	C_RECFN	再帰的関数のテーブル
	CONST	**const** や **code** で宣言された定数データ
	CCSTR	**-y** コンパイラ・オプションを使用した場合の C 文字列リテラルの初期化子（**ECSTR** を参照）
	CSTR	定数 C 文字列（**WSTR** を参照）
	N_CDATA **Z_CDATA**	**DATA** メモリ内の初期化変数の初期値
	CODE	実行可能コード
下位	**RCODE**	C ランタイム・ライブラリ・コード 割込みサービス・ルーチン・コード 終了コード、スタートアップ・コード（**init_C** から）

ROM セグメント

上位	ECSTR WCSTR	書込み可能文字列（-y、-P オプションを参照）
	NPAGE	通常ページ・アセンブラ・ライブラリ・データ
	RF_STACK	再帰的スタック
	DATA N_UDATA N_IDATA	データ領域。-p オプションを使用すると、DATA は UDATA、IDATA で置き換えられる
	CSTACK	ハードウェア・スタック
	EXPR_STACK	式スタック
	INT_EXPR_STACK	割込み式スタック
	BITVAR	ビット変数記憶域
	ZIDATA Z_UDATA	ゼロページ RAM
下位	ZPAGE	

RAM セグメント

BITVAR

ビット変数

タイプ

読み書き用

説明

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。

CCSTR

文字列リテラル

タイプ

読出し専用

説明

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、C コンパイラの -y オプションを参照してください。-y オプションについては、49 ページの「文字列を変数として初期化（-y）」を参照してください。また、203 ページの「CSTR」、205 ページの「ECSTR」、208 ページの「WCSTR」も参照してください。

CODE

コード

タイプ

読出し専用

説明

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

タイプ

読出し専用

説明

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。

CSTACK

スタック

タイプ

読み書き用

説明

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、nn は長さ、start は位置を示します。

CSTR

文字列リテラル

タイプ

読出し専用

説明

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、49 ページの「文字列を変数として初期化（-y）」を参照してください。また、202 ページの「CCSTR」、205 ページの「ECSTR」、208 ページの「WCSTR」も参照してください。

C_FNT

特殊ページ分岐テーブル

タイプ

読出し専用

説明

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、特定のプロセッサ・オプションの特殊ページのアドレスに置かれなければなりません。

C_ICALL

間接関数呼出しのテーブル

タイプ

読出し専用

説明

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

タイプ

読出し専用

説明

コンパイラ専用

再帰的関数テーブルを保持します。

ECSTR

書込み可能な文字列リテラルのコピー

タイプ

読み書き用

説明

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、C コンパイラの -y オプションを参照してください。-y オプションについては、49 ページの「文字列を変数として初期化（-y）」を参照してください。また、202 ページの「CCSTR」、203 ページの「CSTR」、208 ページの「WCSTR」も参照してください。

EXPR_STACK

式スタック

タイプ

読み書き用

説明

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。

INTVEC

割込みベクトル

タイプ

読出し専用

説明

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、特定のプロセッサ・オプションのベクトルの開始アドレスでなければなりません。

INT_EXPR_STACK

割込み式スタック

タイプ

読み書き用

説明

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。

NO_INIT

不揮発性変数

タイプ

読み書き用

説明

アセンブリ・アクセス可能

不揮発性メモリに入れられる変数を保持します。これらの変数は、コンパイラで割り当てられたり、メモリ#pragma の使用で no_init と宣言されたり、no_init で生成されたり、あるいはアセンブリ言語ソースからマニュアル操作で生成されたりします。

NPAGE

正規ページ・アセンブラのライブラリ・データ

タイプ

読み書き用

説明

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。

N_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 N_IDATA セグメントへ初期化値をコピーします。

N_IDATA

初期化済み静的データ

タイプ

読み書き用

説明

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。上記の N_CDATA も参照してください。

N_UDATA

非初期化静的データ

タイプ

読み書き用

説明

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。

RCODE

スタートアップ・コード

タイプ

読出し専用

説明

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。

RF_STACK

再帰的スタック

タイプ

読み書き用

説明

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。

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。

ZPAGE

ゼロ・ページ・アセンブラのライブラリ・データ

タイプ

読み書き用

説明

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。

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。

Z_IDATA

初期化済み静的データ

タイプ

読み書き用

説明

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。上記の Z_CDATA 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。

Z_UDATA

非初期化静的データ

タイプ

読み書き用

説明

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。

第 17 章　ICC740 制限事項

740 プロセッサ命令セットの制限のため、ICC コンパイラの仕様もそれによる多少の制限事項があります。

パラメータ領域サイズ

1 つの関数で、合計 256 バイトを超えるサイズのパラメータをとることはできません。パラメータのサイズが 256 バイトを超えようとすると、コンパイラはエラーを発生します。

ANSI C 規格との SETJMP の相違点

ANSI C 規格によると、setjmp は限られた数の方法でしか使用することはできません。これには、次のような式が含まれます。

```
if (setjmp(buf1));
if (!setjmp(buf2));
setjmp(buf3);
if (setjmp(buf4)>2);
```

許容可能な最も複雑な式は、定数との比較です。

一般的に犯す“誤り”は、setjmp からの返却値を変数に格納することです。これは許容されていません。

残念なことに、ICC740 は、結果を 0 以外の定数と比較することはできません。これが、ANSI C 規格からはずれている点です。この制限の理由は、式スタックによります。場合によっては、定数は setjmp が呼び出される前にスタックにプッシュされることがありますが、これは関係演算子のインストレーションによって異なります。すなわち、＞や＜＝は最初に setjmp を呼び出すこともありますが、＜や＞＝はオペランドを逆にし、＜＝や＞の比較を行うことによって実現されます。

定数が初めにプッシュされると、それに続く比較は setjmp からの初期リターンでそれを消費します。よって、longjmp を介する後続のリターンは、スタックからランダム・ガーベッジをとってしまいます。

ビット・データ型

静的 bit 変数やオート bit 変数のみが生成されます。関数パラメータや返却値は、ビット型にすることはできません。

アドレスへの直接呼出し

明示的アドレスへの関数呼出しは、どのようなパラメータもとることができません。これは、パラメータ領域がメモリ・アドレスに割り当てられていないためです。明示的アドレスへの関数呼出しを行おうとすると、コンパイラはエラーを発生します。

例：

```
((void *(void))0x2000)();     /* 有効 */
((void *(int))0x2100)(42);     /* 受け付けられない */
```

再帰的関数や間接関数のパラメータ位置

再帰的関数や間接関数は、デフォルトのメモリ領域でのみパラメータをもつことができます。これは、コンパイラには関数が再帰的に呼び出されるか、間接的に呼び出されるかを認識する必要がないために、XLINK でチェックされます。

関数はリエントラントに未対応

ICC 740 コンパイラは、リエントラント関数をサポートしていません。割込み処理ルーチンは、割込み時にフォアグランドで実行される場合がある関数を呼び出してはいません。

直接的な、および間接的な再帰的関数は、完全にサポートされています。

アトミックな C 割込み関数

割込み関数がエントリごとにベースから使用する割込み式スタックは 1 つだけですので、C 言語で記述された割込み関数は、C 関数で処理される別の割込みによって中断することはできません。これを可能にすると、初期割込みコードはその式のガーベッジ・データを使用します。

バンク 1 は読出し専用

37600 プロセッサの特殊バンクは、単にバンクからの読出しを提供する組込み関数でサポートされているのみです。詳細については、175 ページの第 12 章「組込み関数のリファレンス」を参照してください。

特殊バンクは、アセンブラ・コードから書き込まれます。

ネストされた間接関数呼出しは不可

ネストされた間接関数呼出しをもつ式の評価を行うことはできません。これを試みると、コンパイラはエラーを発生します。

例えば、次のプログラムは受け付けられません。

```
int (*fred)(int,int);
void main(void)
{
        (*fred)(3,(*fred)(4,5));              /* ネストされた間接関数呼出し */
}
```

第 18 章　K&R および ANSI C 言語の定義

この章では、K&R 記述 C 言語と ANSI 規格 C 言語間の違いを解説します。

概要

標準 C 言語定義には、主に次の 2 種類があります。

◆　Kernighan & Richie、通常 K&R と省略されます。

これは C 言語の著者によるオリジナル定義であり、著書 The C Programming Language（プログラミング言語 C）に解説されています。

◆　ANSI

ANSI 定義は、オリジナルの K&R 定義を発展させたものです。ANSI 定義は、移植性やパラメータのチェックを強化するための機能を追加し、多少の冗長キーワードをオリジナルから取り外したものです。IAR Systems C コンパイラは、ANSI 認定規格 X3.159-1989 に準拠しています。

両定義とも、カーニハン、リッチー著の「プログラミング言語 C」の最新版に詳細に説明されています。この章では、両定義間の違いを概説します。特に、K&R C に精通し、新しく ANSI 機能を使用したいと願うプログラマに有用です。

定義

ENTRY キーワード

ANSI C では、entry キーワードは取り外されています。よって、entry はユーザ定義のシンボルになります。

CONST キーワード

ANSI C は、const を追加しています。これは、宣言されたオブジェクトが変更不可能であり、よって、読出し専用メモリ・セグメントにコンパイルすることができることを示した属性です。

```
const int i;                    /* 定数 int */
const int *ip;                  /* 定数 int の変数ポインタ */
int *const ip;                  /* 変数 int への定数ポインタ */
```

```
typedef struct                          /*構造体'cmd_entry'を定義する */
{
    char *command;
    void (*function)(void);
} cmd_entry

const cmd_entry table[]=      /*型'cmd_entry'の定数オブジェクトを定義する*/
{
    "help", do_help,
    "reset", do_reset,
    "quit", do_quit
};
```

VOLATILE キーワード

ANSI C は、volatile を追加しています。これは、ハードウェアでオブジェクトの変更ができ、いかなるアクセスをも最適化によって取り外すべきではないことを示した属性です。

SIGNED キーワード

ANSI C は、整数型が符号付きであることを示した属性である signed を追加しています。これは unsigned に対応するもので、整数型指定子の前で使用することができます。

VOID キーワード

ANSI C は、関数返却値、関数パラメータ、汎用ポインタを宣言するために使用することができる型指定子である void を追加しています。例えば、次のとおりです。

```
void f();                               /* 返却値のない関数 */
type_spec f(void);                      /* パラメータのない関数 */
void *p;                                /* 別のポインタにキャストすることができ、どの
                                           ポインタ型とも割付互換性をもつ汎用ポインタ */
```

ENUM キーワード

ANSI C は、後続の名前付き整数定数を連続した値で便宜よく定義するキーワード enum を追加しています。例えば、次のとおりです。

enum {zero,one,two,step=6,seven,eight};

データ型

ANSI C では、基本データ型の 1 セットは次のとおりです。

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*                                   /* ポインタ */
```

関数定義パラメータ

K&R C では、関数パラメータは関数本体の前で従来の宣言文で宣言されます。一方、ANSI C では、パラメータ・リストの各パラメータの前にその型識別子がつきます。例えば、次のとおりです。

K&R	ANSI
`long int g(s)`	`long int g(char * s);`
`char * s;`	
`{`	`{`

ANSI 型関数の引数は、常に型チェックされます。IAR Systems C コンパイラは、-g オプションが使用された場合に限り、K&R 型関数の引数をチェックします。41 ページの「グローバル型チェックの使用可能化（-g）」を参照してください。

関数宣言

K&R C では、関数宣言にはパラメータは含まれていません。一方、ANSI C はこれを含みます。例えば、下表のとおりです。

種類	例
K&R	`extern int f( );`
ANSI（名前付き形式）	`extern int(long int val);`
ANSI（名前なし形式）	`extern int(long int);`

K&R の場合、宣言を介した関数の呼出しでは、そのパラメータ型をチェックすることはできません。また、パラメータ型の不一致があった場合、呼出しは失敗します。

ANSI C の場合、関数引数の型は宣言の中のパラメータの型と照合されます。必要であれば、関数呼出しのパラメータは、割付演算子に対する引数の場合と同じように宣言のパラメータの型にキャストされます。パラメータ名は、宣言の中では任意指定です。

また ANSI は、引数の変数名を示すのに、省略記号（3 ドット）を最終仮パラメータとして含むことを指定しています。

ANSI 型関数への外部参照や順方向参照が使用されている場合は、呼出しの前に関数宣言が必要です。ANSI 型宣言と K&R 型宣言は更新パラメータ（char や float）に関して互換性がありませんので、これらの宣言をミックスさせるのは安全ではありません。

IAR Systems C コンパイラでは、-g オプションは、モジュール間を含め関数呼出し間や宣言間の互換性の問題をすべて検出します。41 ページの「グローバル型チェックの使用可能化（-g）」を参照してください。

16 進文字列定数

ANSI では、バックスラッシュの後に x および 16 進数の値を続けて示すことによって、16 進数定数を使用することができます。例えば、次のとおりです。

```
#define Escape_C "\x1b\x43"     /*エスケープ 'C' \0 */
```

\x43 は、直接含まれた場合に 16 進定数の一部として解釈される ASCII C を表しています。

構造体と共用体の割付

K&R C では、関数や割付演算子は、struct や union オブジェクトを指し示すポインタではあるが、struct や union オブジェクト自体ではない引数をもつことがあります。

ANSI C では、関数や割付演算子は、struct や union オブジェクトである、あるいはこれらのオブジェクトを指し示すポインタである引数をもつことができます。関数は、構造体や共用体を返却することもできます。

```
struct s a, b;                  /* 前に宣言された struct s */
struct s f(struct s parm);      /* structs を受け付け、返却する関数の宣言 */
a = f(b);                       /* その関数の呼出し */
```

構造体の利用度をさらに向上させるために、ANSI ではオート構造体を初期化することができます。

共用変数オブジェクト

さまざまな C コンパイラがありますが、これらはモジュール間で共用される変数オブジェクトの取り扱いで異なります。IAR Systems C コンパイラは、ANSI 補足ドキュメント Rationale For C で推奨されている Strict REF/DEF と呼ばれる方式を使用します。この方式では、1 つを除き全てのモジュールが変数宣言の前で extern キーワードを使用することを要求します。例えば、下表のとおりです。

モジュール#1	モジュール#2	モジュール#3
`int i;`	`extern int i;`	`extern int i;`
`int j=4;`	`extern int j;`	`extern int j;`

#elif

ANSI C の新しい#elif 疑似命令を使用すると、よりコンパクトなネスト型 else-if 構造体を使用することができます。

```
#elif expression
...
```

は、次の疑似命令に相当します。

```
#else
#if expression
...
#endif
```

#error

#error 疑似命令は、条件付きコンパイルと連結して使用するために用意されています。#error 疑似命令が検出されると、コンパイラはエラー・メッセージを発行し、動作を終了します。

第 19 章　診断

診断エラー・メッセージ、警告メッセージは、次の 6 種類に分類されます。

- コマンド行エラー・メッセージ
- コンパイル・エラー・メッセージ
- コンパイル警告メッセージ
- コンパイルの致命的エラー・メッセージ
- コンパイルのメモリ・オーバーフロー・メッセージ
- コンパイルの内部エラー・メッセージ

コマンド行エラー・メッセージ

コマンド行エラーは、コマンド行で指定されたパラメータに障害を見つけると発生します。この場合、コンパイラはそのままで明白なメッセージを発行します。

コンパイル・エラー・メッセージ

コンパイル・エラー・メッセージは、コンパイラがコードの生成不能などの C 言語規則を明確に違反する構成体を検出すると生成されます。

ICC コンパイラは、他の多くの C コンパイラよりも互換性の問題に対してより厳密に対応します。特に、ポインタや整数は明示的にキャストされない場合は互換性がないものと見なされます。

コンパイル警告メッセージ

コンパイル警告メッセージは、コンパイラが重要ではあるがコンパイルの完了を妨げるほどではないプログラミング・エラーや脱落を発見した場合に発行されます。

コンパイルの致命的エラー・メッセージ

コンパイルの致命的エラー・メッセージは、コンパイラが単にコードの生成を妨げるだけでなく、以後のソースの処理を意味のないものにしてしまう状態を検出すると生成されます。このメッセージを発行した後、コンパイルは終了します。コンパイルの致命的エラー・メッセージは、この章の「コンパイル・エラー・メッセージ」で解説され、致命的エラーと記されています。

コンパイルのメモリ・オーバーフロー・メッセージ

コンパイラがメモリを使い切ると、次の特別なメッセージを発行します。

```
* * * COMPILER OUT OF MEMORY * * *
Dynamic memory used: nnnnnn bytes
```

このエラーが発生した場合の処置は、システム・メモリを追加するか、あるいはソース・ファイルをより小さなモジュールに分割するかのどちらかです。また、次のオプションを使用すると、コンパイラはより多くのメモリを使用します（**-rn** オプションは除きます）。

オプション	内容
-q	ニモニックをリストに入れます。
-x	相互参照リストを生成します。
-A	アセンブラはソース名に接頭語をつけます。
-P	PROM 可能なコードを生成します。
-r	デバッグ情報を生成します。

コンパイルの内部エラー・メッセージ

コンパイルの内部エラー・メッセージは、例えば、内部一貫性チェックの不良などコンパイラ内部の障害により重大かつ予想外の誤りが発生したことを示しています。コンパイラは、自ら明白なメッセージを発行した後、動作を終了します。

通常、内部エラーは発生しません。発生した場合は、当社のテクニカル・サポート部門に連絡してください。連絡の際は、問題に関する全ての情報を伝えてください。また、できれば内部エラーを発生させた問題のあるディスクを提供してください。

コンパイル・エラー・メッセージ

No.	エラー・メッセージ	処置
0	**Invalid syntax**	コンパイラが文や宣言をデコードできませんでした。
1	**Too deep #include nesting (max is 10)**	致命的エラー。コンパイラの**#include** ファイルのネスト化の制限を超えています。考えられる原因の 1 つは、不注意に再帰化された**#include** ファイルです。
2	**Failed to open #include file 'name'**	致命的エラー。コンパイラが**#include** ファイルを開くことができませんでした。原因は、指定されたディレクトリにファイルが存在しないか（誤った -I 接頭語か、あるいは **C_INCLUDE** パスによるものと考えられます）、あるいは読出し不能になっています。
3	**Invalid #include file name**	致命的エラー。**#include** ファイル名が無効でした。#include ファイル名は、**<file>**あるいは**"file"**と記述してください。
4	**Unexpected end of file**	致命的エラー。宣言内で、関数定義内で、あるいはマクロ展開中にファイルの終りが検出されました。原因は、( )や{ }のネスティング不良と考えられます。
5	**Too long source line (max is 512 chars); truncated**	ソース行の長さがコンパイラのリミットを超えています。
6	**Hexadecimal constant without digits**	16 進定数の接頭語 **0x** あるいは **0X** が、それに続く 16 進数がない状態で検出されました。
7	**Character constant larger than "long"**	文字定数の文字数が多すぎて、長整数の領域に適合することができません。

No.	エラー・メッセージ	処置
8	**Invalid character encoun-tered: '\xhh'; ignored**	C 文字セットにない文字が検出されました。
9	**Invalid floating point constant**	浮動小数点定数が大きすぎる、あるいは構文が無効になりました。有効な形式については、ANSI 規格を参照してください。
10	**Invalid digits in octal constant**	コンパイルが 8 進定数で非 8 進数を検出しました。有効な 8 進数は 0〜7 です。
11	**Missing delimiter in literal or character constant**	文字、あるいはリテラル定数に、終了デリミタ ' あるいは " を検出できませんでした。
12	**String too long (max is 509)**	1 文字列あるいは連結文字列の長さの制限を超えました。
13	**Argument to #define too long (max is 512)**	¥で終了した行が**#define** 行になりましたが、行が長すぎます。
14	**Too many formal parameters for #define (max is 127)**	致命的エラー。マクロ定義で検出された仮パラメータの数が多すぎます（**#define** 疑似命令）。
15	**',' or ')' expected**	コンパイラが、関数定義ヘッダーあるいはマクロ定義で無効な構文を検出しました。
16	**Identifier expected**	宣言子、**goto** 文、あるいはプリプロセッサ行から識別子が脱落していました。
17	**Space or tab expected**	プリプロセッサ引数は、タブや空白文字で疑似命令と区切ってください。
18	**Macro parameter 'name' redefined**	**#define** 文のシンボルの仮パラメータが繰り返されました。

No.	エラー・メッセージ	処置
19	**Unmatched #else, #endif or #elif**	致命的エラー。**#if, #ifdef**、あるいは**#ifndef** が脱落しています。
20	**No such pre-processor command: 'name'**	＃の後に未確認の識別子が続いていました。
21	**Unexpected token found in pre-processor line**	引数部を読み取った後、プリプロセッサ行が空にはなりませんでした。
22	**Too many nested parame-terized macros(max is 50)**	致命的エラー。プリプロセッサの限界を超えました。
23	**Too many active macro parameters (max is 256)**	致命的エラー。プリプロセッサの限界を超えました。
24	**Too deep macro nesting (max is 100)**	致命的エラー。プリプロセッサの限界を超えました。
25	**Macro 'name' called with too many parameters**	致命的エラー。パラメータ化した**#define** マクロが、宣言された数を超える引数で呼び出されました。
26	**Actual macro parameter too long (max is 512)**	単一のマクロ引数は、ソース行の長さを超えることはできません。
27	**Macro 'name' called with too few parameters**	致命的エラー。パラメータ化した**#define** マクロが、宣言された数より少ない引数で呼び出されました。
28	**Missing #endif**	致命的エラー。偽の条件後のテキストのスキップ中にファイルの終りが検出されました。
29	**Type specifier expected**	型記述が脱落しています。このエラーは、**struct, union,** プロトタイプ型関数定義／宣言、あるいはK&R 関数仮パラメータの宣言で起こることがあります。

No.	エラー・メッセージ	処置
30	**Identifier unexpected**	無効な識別子があります。これは、次のような型名定義の識別子であったり、2 つの連続識別子であったりします。 `sizeof(int*ident);`
31	**Identifier 'name' rede-clared**	宣言子識別子が再度宣言されました。
32	**Invalid declaration syntax**	デコード化できない宣言子がありました。
33	**Unbalanced '(' or ')' in declarator**	宣言子の中にカッコのエラーがありました。
34	**C statement or func-def in #include file, add "i" to the "-r" switch**	C-SPY デバッガが使用されたときに**#include** コードで適切な C ソース行のステップ動作を行うには、**-ri** オプションを指定してください。 他のソース・コード・デバッガ（UBROF 出力形式を使用しないもの）は、**#include** ファイルのコードでは動作しないことがあります。
35	**Invalid declaration of "struct", "union" or "enum" type**	**struct**, **union** あるいは **enum** の後に、無効なトークンが続いていました。
36	**Tag identifier 'name' redeclared**	**struct**, **union** あるいは **enum** タグは、既に現在の有効範囲で定義されています。
37	**Function 'name' declared within "struct" or "union"**	関数が **struct** あるいは **union** のメンバーとして宣言されました。
38	**Invalid width of field (max nn)**	フィールドの宣言幅が整数のサイズを超えています（**nn** は、目的プロセッサによって 16 か 32 です）。

No.	エラー・メッセージ	処置
39	**',' or ';' expected**	宣言子の終りで **,** あるいは **;** が脱落しています。
40	**Array dimension outside of "unsigned" "int" bounds**	配列次元が負、あるいは符号なし整数で表示される範囲を超えています。
41	**Member 'name' of "struct" or "union" redeclared**	**struct** あるいは **union** のメンバーが、再度宣言されました。
42	**Empty "struct" or "union"**	メンバーをもたない **struct** あるいは **union** の宣言があります。
43	**Object cannot be initialized**	**typedef** 宣言子や **struct** あるいは **union** メンバーを初期化しようとしました。
44	**';' expected**	文や宣言には、終了セミコロンが必要です。
45	**']' expected**	不良配列宣言や不良配列式があります。
46	**':' expected**	**default** や **case** ラベルの後で、あるいは**?-**演算子の中でコロンが脱落しています。
47	**'(' expected**	考えられる原因は、**for, if**、あるいは **while** 文の形式不良です。
48	**')' expected**	考えられる原因は、**for, if**、あるいは **while** 文や式の形式不良です。
49	**',' expected**	無効な宣言がありました。
50	**'{' expected**	無効な宣言、あるいは無効な初期化子がありました。
51	**'}' expected**	無効な宣言、あるいは無効な初期化子がありました。

No.	エラー・メッセージ	処置
52	**Too many local variables and formal parameters (max is 1024)**	致命的エラー。コンパイラの制限を超えました。
53	**Declarator too complex (max is 128 '(' and/or '*')**	宣言の中の（**,**)、あるいは*****が多すぎます。
54	**Invalid storage class**	無効なオブジェクトの記憶域クラスが指定されました。
55	**Too deep block nesting (max is 50)**	致命的エラー。関数定義の**{}**ネスティングが深すぎます。
56	**Array of functions**	関数の配列が宣言されようとしました。 有効な形式は、関数を指し示すポインタの配列です。 `int array[5] ();  /* 無効 */` `int (*array[5]) (); /* 有効 */`
57	**Missing array dimension specifier**	指定された次元が脱落している多次元配列宣言子がありました。除外できるのは、最初の次元だけです（**extern** 配列や関数仮パラメータの宣言の場合)。
58	**Identifier 'name' redefined**	宣言子識別子が再度宣言されました。
59	**Function returning array**	関数は配列を返却できません。
60	**Function definition expected**	後続の関数定義がない K&R 関数ヘッダーが検出されました。例： `int f(i);  /* 無効 */`
61	**Missing identifier in declaration**	宣言子に識別子が不足しています。
62	**Simple variable or array of "void" type**	**void** 型にできるのは、ポインタ、関数、仮パラメータだけです。

No.	エラー・メッセージ	処置
63	**Function returning function**	次のような場合、関数は関数を返すことができません。 `int f()(); /* 無効 */`
64	**Unknown size of variable object 'name'**	定義されたオブジェクトに未確認のサイズがあります。これは、次元指定のない外部配列であったり、部分的に（順方向で）宣言された **struct** や **union** のオブジェクトであったりします。
65	**Too many errors encountered (>100)**	致命的エラー。コンパイラは、特定の数の診断メッセージが発行されると中止されます。
66	**Function 'name' redefined**	関数の複数定義が検出されました。
67	**Tag 'name' undefined**	型が未定義のままの **enum** 型の変数の定義、または関数プロトタイプの中で、あるいは **sizeof** 引数として定義されていない **struct** や **union** 型の参照がありました。
68	**"case" outside "switch"**	実行可能状態の **switch** 文のない **case** がありました。
69	**"interrupt" function may not be referred or called**	プログラムに **interrupt** 関数呼び出しが含まれています。割込み関数は、ランタイム・システムでのみ呼び出すことができます。
70	**Duplicated "case" label: nn**	**case** ラベルとして同じ定数値が 2 回以上使用されました。
71	**"default" outside "switch"**	実行可能状態の **switch** 文のない **default** がありました。
72	**Multiple "default" within "switch"**	1 つの **switch** 文に 2 つ以上の **default** があります。

No.	エラー・メッセージ	処置
73	**Missing "while" in "do" - "while" statement**	考えられる原因は、複数文の周囲で**{}**が脱落しています。
74	**Label 'name' redefined**	同じ関数の中で、ラベルが 2 回以上定義されました。
75	**"continue" outside iteration**	実行可能状態の **while**、**do ... while**、あるいは **for** 文の外に **continue** 文がありました。
76	**"break" outside "switch" or iteration statement**	実行可能状態の **switch**、**while**、**do ... while**、あるいは **for** 文の外に **break** 文がありました。
77	**Undefined label "name"**	関数本体に、**label**:定義のない **goto label** があります。
78	**Pointer to a field not allowed**	**struct** あるいは **union** のフィールド・メンバーを指し示すポインタがあります。 **struct** **{** **int *f:6; /* 無効 */** **}**
79	**Argument of binary operator missing**	2 進演算子の第 1 引数、あるいは第 2 引数が脱落しています。
80	**Statement expected**	文が予測される箇所で、**? : , }**、あるいは**}**の 1 つが検出されました。
81	**Declaration after statement**	文の後に宣言が見つかりました。 このエラーは、例えば次のような不必要な；による場合があります。 **int i;;** **char c; /* 無効 */** 2 番目の；は文となりますので、文の後に宣言が発生します。

No.	エラー・メッセージ	処置
82	**"else" without preceding "if"**	考えられる原因は、**{}**ネスティング不良です。
83	**"enum"constant(s) outside "int" or "unsigned" "int" range**	生成された列挙定数が小さすぎたか、あるいは大きすぎました。
84	**Function name not allowed in this context**	関数名を間接アドレスとして使用しようとしました。
85	**Empty "struct", "union" or "enum"**	メンバーをもたない **struct** や **union** の定義が、あるいは列挙定数をもたない **enum** の定義があります。
86	**Invalid formal parameter**	関数宣言の中の仮パラメータに誤りがあります。 考えられる原因は、次のとおりです。 **int f(); /* 有効な K&R 宣言 */** **int f(i); /* 無効な K&R 宣言 */** **int f(int i); /* 有効な ANSI 宣言 */** **int f(i); /* 無効な ANSI 宣言 */**
87	**Redeclared formal parameter: 'name'**	K&R 関数宣言の仮パラメータが 2 回以上宣言されました。
88	**Contradictory function declaration**	**void** が指定子の他の型と一緒に関数パラメータ型のリストに存在します。
89	**"..." without previous parameter(s)**	**...** のみをパラメータ記述として指定することはできません。 例： **int f(...); /* 無効 */** **int f(int, ...); /* 有効 */**

No.	エラー・メッセージ	処置
90	**Formal parameter identifier missing**	プロトタイプ関数定義のヘッダーでパラメータの識別子が脱落しています。 例： `int f( int *p, char, float ff)` `/* 無効 － 第 2 パラメータに名前がない。*/` `{ /* 関数本体 */` `}`
91	**Redeclared number of formal parameters**	最初の宣言とは異なった数のパラメータでプロトタイプ関数が宣言されました。 例： `int f(int,char); /* 第 1 宣言 － 有効 */` `int f(int); /* パラメータの数が少ない － 無効 */` `int f(int,char,float); /*パラメータの数が多い－無効 */`
92	**Prototype appeared after reference**	関数のプロトタイプ宣言が、K&R 関数として定義、あるいは参照された後に現れました。
93	**Initializer to field of width nn (bits) out of range**	ビット・フィールドは大きすぎる定数で初期化されたため、フィールド領域に適合できませんでした。
94	**Fields of width 0 must not be named**	ゼロ長さフィールドは次の **int** 境界にフィールドを合わせるために使用できるのみで、識別子を介してアクセスすることはできません。
95	**Second operand for division or modulo is zero**	0 で除算しようとしました。
96	**Unknown size of object pointed to**	サイズが既知でなければならない式の中で不完全なポインタ型が使用されました。

No.	エラー・メッセージ	処置
97	**Undefined "static" function 'name'**	関数が **static** 記憶域クラスで宣言されましたが、定義されませんでした。
98	**Primary expression expected**	式が脱落しています。
99	**Extended keyword not allowed in this context**	プロセッサに固有な拡張キーワードが違法文脈の中で発生しました。例：**interrupt int i**
100	**Undeclared identifier: 'name'**	宣言されていない識別子に対する参照がありました。
101	**First argument of '.' operator must be of "struct" or "union" type**	**struct** でも **union** でもない引数にドット演算子 **.** が適用されました。
102	**First argument of '->' was not pointer to "struct" or "union"**	**struct** でも **union** のポインタでもない引数に矢印演算子 **->** が適用されました。
103	**Invalid argument of "sizeof" operator**	**sizeof** 演算子が、ビット・フィールド、関数、あるいは未確認のサイズの **extern** 配列に適用されました。
104	**Initializer "string" exceeds arrary dimension**	明示的次元をもつ **char** の配列が、配列のサイズを超える文字列で初期化されました。 **char array [4] ="abcde"; /* 無効 /**
105	**Language feature not implemented: 'name'**	現在、コンパイラは使用された言語機能をサポートしていません。
106	**Too many function parameters (max is 127)**	致命的エラー。関数宣言／定義内のパラメータの数が多すぎます。

No.	エラー・メッセージ	処置
107	**Function parameter 'name' already declared**	関数定義ヘッダーの仮パラメータが 2 回以上宣言されました。 例： `* K&R 関数 */ int myfunc( i, i ) /* 無効 */` `int I;` `{` `}` `/* プロトタイプ関数 */` `int myfunc( int i, int i ) /* 無効 */` `{` `}`
108	**Function parameter 'name' declared but not found in header**	K&R 関数宣言で、パラメータが宣言されましたが、関数ヘッダーで指定されていませんでした。 例： `int myfunc( I )` `int i, j; /* 無効 − j が関数ヘッダーに指定されていない。*/` `{` `}`
109	**';' unexpected**	予想外のデリミタが検出されました。
110	**')' unexpected**	予想外のデリミタが検出されました。
111	**'{' unexpected**	予想外のデリミタが検出されました。
112	**',' unexpected**	予想外のデリミタが検出されました。
113	**':' unexpected**	予想外のデリミタが検出されました。
114	**'[' unexpected**	予想外のデリミタが検出されました。

No.	エラー・メッセージ	処置
115	**'(' unexpected**	予想外のデリミタが検出されました。
116	**Integral expression required**	式が評価された結果、誤った型であると判明しました。
117	**Floating point expression required**	式が評価された結果、誤った型であると判明しました。
118	**Scalar expression required**	式が評価された結果、誤った型であると判明しました。
119	**Pointer expression required**	式が評価された結果、誤った型であると判明しました。
120	**Arithmetic expression required**	式が評価された結果、誤った型であると判明しました。
121	**Lvalue required**	式の結果がメモリ・アドレスではありませんでした。
122	**Modifiable lvalue required**	式の結果が変数オブジェクトや **const** ではありませんでした。
123	**Prototyped function argument number mismatch**	プロトタイプ関数が、宣言された数とは異なる数の引数で呼び出されました。
124	**Unknown "struct" or "union" member: 'name'**	**struct** や **union** の存在しないメンバーを参照しようとしました。
125	**Attempt to take address of field**	**&**演算子は、ビット・フィールドで使用することはできません。
126	**Attempt to take address of "register" variable**	**&**演算子は、**register** 記憶域クラスをもつオブジェクトで使用することはできません。

No.	エラー・メッセージ	処置
127	**Incompatible pointers**	ポインタの互換性がありません。ポインタが指す対象は、完全な互換性を持っていな　ければなりません。 　特に、ポインタが（直接、あるいは間接的に）プロトタイプ関数を指し示した場合、コードは返却値およびパラメータの数と型に関して互換性検査を実行します。これは、非互換性をかなり深く隠せることを意味します。例えば、次のとおりです。 **char (*(*p1)[8])(int);** **char (*(*p2)[8])(float);** **/* p1 と p2 は互換性がない。－ 関数パラメータが互換性のない型をもっている。*/** 　配列次元が、指し示されたオブジェクトの記述に現れる場合、互換性検査は配列次元のチェックも含みます。 **int (*p1)[8];** **int (*p2)[9];** **/* p1 と p2 は互換性がありません。－ 配列次元が異なる。*/**
128	**Function argument incompatible with its declaration**	関数引数が、宣言の引数と互換性がありません。
129	**Incompatible operands of binary operator**	2 進演算子に対する 1 つ以上のオペランドの型がこの演算子と互換性がありませんでした。
130	**Incompatible operands of '=' operator**	=に対する 1 つ以上のオペランドの型が=と互換性がありませんでした。
131	**Incompatible "return" expression**	式の結果が **return** 値の宣言と互換性がありません。
132	**Incompatible initializer**	初期化子の式の結果が、初期化されたオブジェクトと互換性がありません。

No.	エラー・メッセージ	処置
133	**Constant value required**	**case** ラベルの式、**#if, #elif**、ビット・フィールド宣言子、配列宣言子、静的初期化子が定数ではありませんでした。
134	**Unmatching "struct" or "union" arguments to '?' operator**	？演算子の第 2、第 3 引数が違っています。
135	**"pointer + pointer"**	ポインタは加算することはできません。
136	**Redeclaration error**	現在の宣言と、同じオブジェクトの以前の宣言との間には一貫性がありません。
137	**Reference to member of undefined "struct" or "union"**	未定義の**struct**や**union**宣言子を唯一参照できるのはポインタです。
138	**"- pointer" expression**	両方の演算子がポインタ、すなわち **pointer - pointer** の場合に限り、－演算子をこれらのポインタに使用することができます。このエラーは、**non-pointer - pointer** 形式の式が検出されたことを示しています。
139	**Too many "extern" symbols declared (max is 32767)**	致命的エラー。コンパイラの限界を超えました。
140	**"void" pointer not allowed in this context**	索引付け式などのポインタ式に虚空ポインタ（構成要素のサイズが未確認）がありました。
141	**#error 'any message'**	致命的エラー。プリプロセッサ疑似命令**#error** が検出され、このモジュールをコンパイルするためにコマンド行で何かを定義しなければならないことがあるのを示しています。

No.	エラー・メッセージ	処置
142	**"interruput" function can only be "void" and have no arguments**	割込み関数の宣言に結果が **void** にならないもの、や引数が含まれています。どちらも使用できません。
143	**Too large, negative or overlapping "interrput" [value] in name**	宣言された割込み関数の**[vector]**値をチェックしてください。
144	**Bad context for storage modifier (storage-class or function)**	**no_init** キーワードは、静的記憶域クラスをもつ変数の宣言にのみ使用できます。すなわち、**no_init** は **typedef** 文に使用したり、関数のオート変数に適用したりすることはできません。実行可能状態の **#pragma memory=no_init** は、関数宣言が検出されるとこのようなエラーを発生します。
145	**Bad context for function call modifier**	キーワード **interrupt, banked, non_banked**、あるいは **monitor** は、関数宣言にのみ適用できます。
146	**Unknown #pragma identifier**	未確認の **pragma** 識別子が検出されました。このエラーが発生すると、**-g** オプションが使用中の場合に限り、オブジェクト・コードの生成を終了します。
147	**Extension keyword "name" is already defined by user**	**#pragma language=extended** の実行時に、コンパイラは、名前付き識別子が拡張キーワードと同じ名前をもっていることを検出しました。このエラーは、コンパイラが ANSI モードで実行中である場合にのみ発行されます。
148	**'=' expected**	**sfr** 宣言識別子には、**=value** を続けてください。

No.	エラー・メッセージ	処置
149	**Attempt to take address of "sfr" or "bit" variable**	&演算子は、**bit** あるいは **sfr** として宣言された変数に適用することはできません。
150	**Illegal range for "sfr" or "bit" address**	アドレス式が、有効な **bit** でも **sfr** アドレスでもありません。
151	**Too many function defined in a single modules.**	モジュールでは 256 個以上の関数を使用することはできません。ただし、関数の宣言数については制限はありません。
152	**'.' expected**	bit 宣言から **.** が脱落しています。
153	**Illegal context for extended specifier**	

740 に固有なエラー・メッセージ

以下の表は、740 に固有なエラー・メッセージをリストしています。

No.	エラー・メッセージ	処置
154	**Too much local data on zero page (max 255)**	255 バイトを越えるゼロ・ページ・データをもった関数があります。このエラーは、リンク時にこのようなコードを機能させるのが不可能であるために生じます。 キーワード **npage** を関数中のいくつかのサイズが大きい配列に追加すると解決される可能性があります。
154	**Nested indirect calls**	ICC740 はネストされた関数ポインター呼び出しを扱うことができません。
154	**Cannot call memory location with parm/return**	関数呼び出しにおいて、数値アドレスをパラメータにしたり、返り値にしたりすることはできません。
154	**Too large parameter area for indirect function (>255)**	間接呼び出しされた関数に対して 255 バイトを越えるパラメータ・データをもつことはできません。
154	**Too large switch table**	**switch** テーブルが大き過ぎます（>64K）。
154	**No prototype for function "XX", assuming that it returns int**	プロトタイプ宣言のない関数を呼び出しています。 コンパイラはこれを、**int** 型を返し、返り値がスタックにあるものと仮定します。 安全のため、つねにプロトタイプ宣言を行ってください。

コンパイル警告メッセージ

No.	警告メッセージ	処置
0	**Macro 'name' redefined**	**#define** で定義されたシンボルが、別の引数あるいは仮の並びで再宣言されました。
1	**Macro formal parameter 'name' is never referenced**	**#define** 仮パラメータが引数文字列に一度も現れませんでした。
2	**Macro 'name' is already #undef**	**#undef** が、マクロではないシンボルに適用されました。
3	**Macro 'name' called with empty parameter(s)**	**#define** 文の中で宣言されたパラメータ化マクロが、ゼロ長さの引数で呼び出されました。
4	**Macro 'name' is called recursively; not expanded**	再帰的マクロの呼び出しは、プリプロセッサにそのマクロの以降の展開を停止させます。
5	**Undefined symbol 'name' in #if or #elif; assumed zero**	**#if** 式や**#elif** 式では非マクロ・シンボルはゼロとして扱われなければならないと考えるのは、よいプログラム手法とはいえません。**#ifdef symbol** か、**#if defined (symbol)**を使用してください。
6	**Unknown escape sequence ('\c'); assumed 'c'**	文字定数あるいは文字列リテラルで検出されたバックスラッシュ（\）の後に、未確認のエスケープ文字が続いていました。
7	**Nested comment found without using the '-C' option**	コメントの中で文字の列 **/*** が見つけられましたが、無視されました。
8	**Invalid type-specifier for field; assumed "int"**	この処理系では、ビット・フィールドは **int** あるいは **unsigned int** 型として指定することができます。

No.	警告メッセージ	処置
9	**Undeclared function parameter 'name'; assumed "int"**	K&R 関数定義のヘッダーの未宣言識別子は、デフォルトでは **int** 型に指定されます。
10	**Dimension of array ignored;array assumed pointer**	明示的次元をもつ配列が仮パラメータとして指定され、コンパイラはそれをオブジェクトを指し示すポインタとして処理しました。
11	**Storage class "static" ignored; 'name' declared "extern"**	まずオブジェクトあるいは関数が **extern** として(明示的に、あるいはデフォルトで）宣言され、後に **static** として再宣言されました。静的宣言は、無視されます。
12	**Incompletely bracketed initializer**	あいまいさを避けるために、初期化子は {} カッコを 1 レベルのみで使用するか、あるいは {} カッコで完全に囲ってください。
13	**Unreferenced label 'name'**	ラベルが定義されていましたが、一度も参照されませんでした。
14	**Type specifier missing; assumed "int"**	宣言で型指定子が指定されていません。**int** と見なされます。
15	**Wrong usage of string operator ('#' or '##'); ignored**	この処理系では、**#**や**##**演算子の使用はパラメータ化マクロのトークン・フィールドに制限されます。 さらに、**#**演算子は仮パラメータの前に置かれなければなりません。 `#define mac(p1) #pi  /* "p1"になる。 */` `#define mac(p1,p2) p1+p2##add_this` `/* 組合せ p2 */`

No.	警告メッセージ	処置
16	**Non-void function: return with<expression>; expected**	非 **void** 関数定義は、つねに定義された返り値で終了しなければなりません。
17	**Invalid storage class for function; assumed to be "extern"**	関数の記憶域クラスが無効ですので、無視されます。有効なクラスは、**extern, static**、あるいは **typedef** です。
18	**Redeclared parameter's storage class**	関数仮パラメータの記憶域クラスが、後続の宣言／定義で **register** から **auto** に、あるいはその逆に変更されました。
19	**Storage class "extern" ignored; 'name' was first declared as "static"**	静的記憶域クラスとして宣言された識別子が、後に明示的に、あるいは暗黙に **extern** として宣言されました。**extern** 宣言は、無視されます。
20	**Unreachable statement(s)**	1 つ以上の文の前に無条件分岐やリターンが置かれ、そのために文が絶対に実行できないようになっています。 例： **break;** **i = 2; /* 決して実行されない */**
21	**Unreachable statement (s) at unreferenced label 'name'**	1 つ以上のラベル付き文の前に無条件分岐やリターンが置かれていましたが、ラベルは一度も参照されませんでした。そのため、文は決して実行されません。 例： **break;** **here;** **i = 2 /* 決して実行されない */**
22	**Non-void function: explicit "return" <expression>; expected**	非 **void** 関数が暗黙のリターンを行いました。 これは、ループやスイッチを予想外に終了した結果に生じることがあります。**case** 構成体のいかんにかかわらず、デフォルトのない **switch** は常にコンパイラで終了可能なものと見なされます。

No.	警告メッセージ	処置
23	**Undeclared function 'name'; assumed "extern" "int"**	未宣言の関数を参照すると、デフォルトの宣言が使用されます。関数は、K&R 型のもので、**extern** 記憶域クラスをもち、**int** を返すものと想定されます。
24	**Static memory option converts local "auto" or "register" to "static"**	静的メモリ割当のコマンド行オプションを使用すると、**auto** や **register** 宣言は、**static** として扱われます。
25	**Inconsistent use of K&R function - varying number of parameters**	K&R 関数がさまざまな数のパラメータで呼び出されました。
26	**Inconsistent use of K&R function - changing type of parameter**	K&R 関数がさまざまな型のパラメータで呼び出されました。 例： `myfunc ( 34 );  /* int 引数 */` `myfunc ( 34.6 );  /* float 引数 */`
27	**Size of "extern" object 'name' is unknown**	**extern** 配列は、サイズで宣言してください。
28	**Constant [index] outside array bounds**	宣言された配列境界外に定数索引がありました。
29	**Hexadecimal escape sequence larger than "char"**	拡張表記は、char に適合するように短縮されます。

No.	警告メッセージ	処置
30	**Attribute ignored**	**const** や **volatile** はオブジェクトの属性ですので、同時に宣言されたオブジェクトをもたない **structure, union**、あるいは **enumeration** タグ定義で指定された場合、これらの属性は無視されます。また、関数は **const** や **volatile** を変換することができないものと見なされます。 例： **const struct s** **{** **...** **}; /* オブジェクトは宣言されていないので、** **const は無視される。－ 警告 */** **const int myfunc(void);** **/* const int を返却する関数－警告 */** **const int (*fp)(void);** **/* const int を返却する関数を指し示す** **ポインタ － 警告 */** **int (*const fp) (void);** **/* int を返却する関数を指し示す const** **ポインタ － OK、警告なし */**
31	**Incompatible parameters of K&R functions**	関数あるいは K&R 関数宣言を指し示す（恐らく、間接）ポインタが互換性のないパラメータ型をもっています。

No.	警告メッセージ	処置
32	**Incompatible numbers of parameters of K&R functions**	関数あるいは K&R 関数宣言を指し示す（恐らく、間接）ポインタが異なった数のポインタをもっています。
33	**Local or formal 'name' was never referenced**	仮パラメータやローカル変数オブジェクトが関数定義で使用されていません。
34	**Non-printable character '\xhh' found in literal or character constant**	文字列リテラルや文字定数で非印字文字を使用するのは、よいプログラミング手法とは考えられません。**\0xhhh** を使用すると、同じ結果が得られます。
35	**Old-style (K&R) type of function declarator**	旧式の K&R 関数宣言子が見つかりました。この警告は、**-gA** オプションを使用している場合に限り発行されます。
36	**Floating point constant out of range**	浮動小数点値が大きすぎたり、小さすぎたりして、ターゲット・プロセッサの浮動小数点法では表示することができません。
37	**Illegal float operation: division by zero not allowed**	定数演算中に、ゼロ除算が見つかりました。
39	**Dummy statement.**	冗長的なコードが見つかりました。通常、これはプログラムの入力ミスを示しています。 例： **a+b;**
40	**Possible bug! “If” statement terminated**	通常、これはプログラムのエラーを示しています。 例： **if (a == b); /* ;は入力ミス */** **{** **/* if 文の本体 */** **}**

No.	警告メッセージ	処置
41	**Possible bug! Uninitalized variable.**	変数が初期化前に使用されています。 例： ``` void func(p1) { short a; p1 += a; } ```
42		このメッセージは削除されました。
43	**Possible bug! Integer promotion may cause problems. Use cast to avoid it**	ANSI C では、整数演算が **int** よりも小さな精度をもつ場合、全ての整数演算があたかも **int** 型であるかのように結果が引き出されることを要求します。 例： ``` short test(unsinged char a) { if (~a) return (1); else return (-1); } ``` この例では、たとえ **0xff** が指定されても常に値 1 を返します。その理由は、まず変数 **a** を **0x00ff** にキャストし、その後 **bit NOT** を実行する整数拡張によります。整数拡張は、多くのコンパイラでは省略されています。よって、この警告は旧式コードの再コンパイル時によく生成されます。
44	**Possible bug! Single '=' instead of '==' used in "if" statement.**	通常、このメッセージはプログラムのエラーを示しています。 例： ``` if (a = 1) { /* if 文の本体 */ } ```
45	**Redundant expression. Example: Multiply with 1, and with 0.**	このメッセージは、プログラムのエラーを示しています。 例： ``` value = value * 1; ```

No.	警告メッセージ	処置
46	**Possible bug! Strang or faulty expression. Example: Division by zero.**	通常、このメッセージはプログラムにバグがあることを示しています。
47	**Unreachable code deleted by the global optmizer.**	ユーザ・コード中の冗長なコード・ブロックです。おそらくバグの結果と思われますが、通常は単なる不完全なコードであることを示しています。
48	**Unreachable returns. The funtion will never return.**	関数は、決して呼び出し関数に戻ることはできません。これはバグを示していることもありますが、通常はRTOSシステムで決してループを終了させない場合に発生します。
49	**Unsigned compare always true/false.**	これは、プログラムにバグがあることを示しています。一般的な理由は、-c コンパイラ・スイッチが脱落していることです。 例： `for ( uc = 10; uc>=0; uc- )` `{` `/* for 文の本体 */` `}` これは、無符号値が常に 0 より大きいか、あるいは等しいため終りのないループとなります。
51	**Signed compare always true/false.**	これは、プログラムにバグがあることを示しています。

第２０章　コマンドライン版用の環境変数

この章では、コマンドライン・オプションを用いたＣコンパイラ・コンフィギュレーションのカスタマイズについての情報を示します。

以下の環境変数が ICC740 で使用できます。

環境変数	説明
QCC740	たとえば以下のように、コマンドライン・オプションを指定します。 **set QCC740=-v1 -z9**
C_INCLUDE	たとえば以下のように、インクルード用に探索するディレクトリを指定します。 **set C_INCLUDE=c:\iar\inc\;c:\headers**

以下の環境変数が A740 で使用できます。

環境変数	説明
ASM740	たとえば以下のように、コマンドライン・オプションを指定します。 **set A740=-v1**
A740_INC	たとえば以下のように、インクルード用に探索するディレクトリを指定します。 **set A740_INC=c:\myinc**

以下の環境変数が、XLINK リンカで使用できます。

環境変数	説明
XLINK_COLUMNS	1 行当たりのカラム数を設定します。
XLINK_CPU	ターゲット CPU タイプを設定します。
XLINK_DFLTDIR	オブジェクト・ファイル用のデフォルト・ディレクトリへのパスを設定します。
XLINK_ENVPAR	デフォルトの XLINK コマンドラインを生成します。
XLINK_FORMAT	出力形式を設定します。
XLINK_MEMORY	XLINK がファイル境界（0）であるか、またはメモリ境界（非 0）であるかを指定します。
XLINK_PAGE	1 ページ当たりの行数を設定します。
XLINK_TFILE	一時的ファイルを指定します。

以下の環境変数が、XLIB で使用できます。

環境変数	説明
XLIB_COLUMNS	1 行当たりのカラム数を設定します。
XLIB_CPU	ターゲット CPU タイプを設定します。
XLIB_PAGE	1 ページ当たりの行数を設定します。
XLINK_SCROLL_BREAK	指定した行数でのスクロール・ポーズを設定します。

ICC コンパイラ・プログラミング・ガイド
（740 ファミリ） 第 2 版

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