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

■本ドキュメントに記載されているURLは、以下のとおり読み替えをお願いいたします。
http://www.necel.com/
http://www2.renesas.com/
↓
開発環境トップページ http://japan.renesas.com/tools
ダウンロードポータル http://japan.renesas.com/tool_download

■技術問合せについては、以下のページをご覧ください。
http://japan.renesas.com/tech_inquiry

■ツールユーザ登録については、以下のページをご覧ください。
http://japan.renesas.com/myrenesas

RENESAS

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。

ユーザーズ・マニュアル

CC78K0R Ver.1.00

Cコンパイラ

言語編

対象デバイス
78K0Rマイクロコントローラ

資料番号　U17837JJ1V0UM00（第1版）
発行年月　June 2006　CP(K)

© NEC Electronics Corporation 2006

〔メ　モ〕

目　次　要　約

第1章　概　　説　…　15

第2章　C言語の基本構成　…　29

第3章　型，記憶域クラスの宣言　…　54

第4章　型の変換　…　72

第5章　演算子と式　…　77

第6章　C言語の制御構造　…　128

第7章　構造体と共用体　…　148

第8章　外部定義　…　154

第9章　前処理指令（コンパイラに対する指令）　…　158

第10章　ライブラリ関数　…　181

第11章　拡張機能　…　319

第12章　アセンブラとの相互参照　…　432

第13章　効率の良いコンパイラの活用法　…　442

付録A　saddr領域のラベル一覧　…　446

付録B　セグメント名一覧　…　449

付録C　ランタイム・ライブラリ一覧　…　462

付録D　ライブラリ消費スタック一覧　…　466

付録E　ライブラリ最大割り込み禁止時間一覧　…　476

総合索引　…　477

- 本資料に記載されている内容は2006年6月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品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，一切その責を負いません。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。
- 本資料に記載された回路，ソフトウエアおよびこれらに関する情報は，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，ソフトウエアおよびこれらに関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，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。当社製品の不具合により生じた生命，身体および財産に対する損害の危険を最小限度にするために，冗長設計，延焼対策設計，誤動作防止設計等安全設計を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「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。

  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

  特別水準：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

  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，生命維持のための装置またはシステム等

  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。意図されていない用途で当社製品の使用をお客様が希望する場合には，事前に当社販売窓口までお問い合わせください。

（注）

（1）本事項において使用されている「当社」とは，NECエレクトロニクス株式会社およびNEC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う。

（2）本事項において使用されている「当社製品」とは，（1）において定義された当社の開発，製造製品をいう。

M8E　02.11

は　じ　め　に

CC78K0R Cコンパイラ（以下本Cコンパイラとする）は，Draft Proposed American National Standard for Information SystemsーProgramming Language C（December 7, 1988）中の2. ENVIRONMENTと3. LANGUAGEを元に作成されています。したがって，ANSIに準拠しているCソース・プログラムであれば，本Cコンパイラによってコンパイルすることにより，78K0Rマイクロコントローラ応用製品の開発が可能となります。

CC78K0R　Cコンパイラ　言語編（本マニュアル）は，本Cコンパイラを用いてソフトウエアの開発を行われる方に，本Cコンパイラの基本機能と，言語仕様を理解していただくことを目的として書かれています。

本マニュアルでは，本Cコンパイラの操作に関する解説はいたしません。したがって，本マニュアルをご理解されたあと，本Cコンパイラをお使いの際には，CC78K0R　Cコンパイラ　操作編（U17838J）をお読みください。

また，78K0Rマイクロコントローラのアーキテクチャについては，78K0Rマイクロコントローラの各ユーザーズ・マニュアルを参照してください。

【ターゲット・デバイス】

本Cコンパイラでは，78K0Rマイクロコントローラのマイクロコンピュータのソフトウエア開発が可能です。各ターゲット・デバイスに対応させるためには，ターゲットの種類に応じたデバイス・ファイルが必要となります。

【対 象 者】

本マニュアルは，開発対象となるマイクロコンピュータのユーザーズ・マニュアルを一読された方で，ソフトウエア・プログラミングの経験がある方を対象とします。Cコンパイラや C言語の知識は特に必要ありませんが，ソフトウエアに関する用語は理解されているものとして説明します。

【構　　成】

本マニュアルの構成を次に示します。

第1章　概　　説

Cコンパイラの一般的な機能と，本Cコンパイラの性能および特徴を説明します。

第2章　C言語の基本構成

C言語プログラムの構成と，その構成要素を説明します。

第3章　型，記憶域クラスの宣言

C言語で使用される型とその宣言，および記憶クラスについて説明します。

第4章　型の変換

本Cコンパイラによって自動的に行われる型の変換について説明します。

第5章　演算子と式

C言語で使用可能な演算子の記述方法や優先度を説明します。

第6章　C言語の制御構造

C言語の制御構造を，制御の流れや使用方法をあげて説明します。

第7章　構造体と共用体

構造体と共用体の概念と使用方法を説明します。

第8章　外部定義

外部定義の種類とその使用方法を説明します。

第9章　前処理指令（コンパイラに対する指令）

前処理指令の種類と使用方法を説明します。

第10章　ライブラリ関数

C言語におけるライブラリ関数の使い方と個々のライブラリ関数を説明します。

第11章　拡張機能

ターゲット・デバイスを活用するための拡張機能を説明します。

第12章　アセンブラとの相互参照

C言語プログラムからアセンブラ・プログラムを呼び出す方法などを説明します。

第13章　効率の良いコンパイラの活用法

本Cコンパイラを効率良く使用するための手段を説明します。

付　録

saddr領域のラベル一覧，セグメント名一覧，ランタイム・ライブラリ一覧，ライブラリ消費スタック一覧，ライブラリ最大割り込み禁止時間一覧があります。

【読 み 方】

本書の読み方を説明します。

■Cコンパイラおよび C言語初心者の方

Cコンパイラおよび C言語について初心者の方は，第1章から順を追ってご覧ください。本マニュアルでは，C言語プログラムの制御構造から拡張機能にいたるまでを順を追って説明しています。なお，**第1章　概　説**ではCソース・プログラム例を使い，本マニュアルの参照箇所を示していますのであわせてご利用ください。

■Cコンパイラおよび C言語経験者の方

本Cコンパイラの言語仕様は，ANSIに準拠しています。したがって，Cコンパイラおよび C言語経験者の方は，本Cコンパイラ特有の機能を示した**第11章　拡張機能**からお読みください。なお，**第11章　拡張機能**をお読みになる場合には，ターゲット・デバイスである78K0Rマイクロコントローラ付属のユーザーズ・マニュアルを参照してください。

【関連資料】

本マニュアルに関連する資料（ユーザーズ・マニュアル）を紹介します。

開発ツールの資料（ユーザーズ・マニュアル）

資　料　名		資料番号	
		和　　文	英　　文
CC78K0R Ver.1.00　Cコンパイラ	操作編	U17838J	U17838E
	言語編	このマニュアル	U17837E
RA78K0R Ver.1.00　アセンブラ・パッケージ	操作編	U17836J	U17836E
	言語編	U17835J	U17835E
SM+　システム・シミュレータ	操作編	U18010J	U18010E（作成予定）
PM+ Ver.6.20		U17990J	U17990E
ID78K0R-QB Ver.3.20　統合ディバッガ	操作編	U17839J	U17839E

【参考資料】

「Draft Proposed American National Standard for Information Systems - Programming Language C（December 7, 1988）」

【用　　語】

RTOS = 78K0マイクロコントローラ用 リアルタイムOS　RX78K0

【凡　　例】

本マニュアル中で共通に使用される記号などの意味を示します。

…　　：同一の形式を繰り返す
“　”　：“　”で囲まれた文字そのもの
‘　’　：‘　’で囲まれた文字そのもの
⋮　　：プログラム記述の省略形
／　　：区切り記号
＼　　：バック・スラッシュ
[ ]　　：かっこ内は省略可能

目次

第 1 章　概説 … 15
1.1 C 言語とアセンブリ言語 … 15
1.2 C コンパイラによる開発手順 … 17
1.2.1 必要となるソフトウエア … 17
1.2.2 製品開発手順 … 18
1.3 C ソース・プログラムの基本構成 … 20
1.3.1 プログラム形式 … 20
1.4 C コンパイラの最大値 … 23
1.5 C コンパイラの特長 … 25

第 2 章　C 言語の基本構成 … 29
2.1 文字セット … 30
2.1.1 文字集合 … 30
2.1.2 マルチバイト文字 … 30
2.1.3 英字拡張表記（エスケープ・シーケンス）… 31
2.1.4 3 文字表記（トライグラフ・シーケンス）… 31
2.2 キーワード … 32
2.2.1 ANSI-C キーワード … 32
2.2.2 CC78K0R 用に追加されたキーワード … 33
2.3 識別子 … 34
2.3.1 識別子の有効範囲 … 35
2.3.2 識別子の結合 … 36
2.3.3 識別子の名前空間 … 37
2.3.4 オブジェクトの記憶域期間 … 37
2.4 型 … 38
2.4.1 基本型 … 39
2.4.2 文字型 … 43
2.4.3 不完全型 … 43
2.4.4 派生型 … 43
2.4.5 スカラ型 … 44
2.4.6 適合型 … 44
2.4.7 合成型 … 45
2.5 定数 … 46
2.5.1 浮動小数点定数 … 46
2.5.2 整数定数 … 46
2.5.3 列挙定数 … 48
2.5.4 文字定数 … 48
2.6 文字列リテラル … 49
2.7 演算子 … 50
2.8 区切り子 … 51
2.9 ヘッダ名 … 52
2.10 コメント … 53

第 3 章　型，記憶域クラスの宣言 … 54
3.1 記憶域クラス指定子 … 55
3.2 型指定子 … 56

3.2.1 構造体指定子と共用体指定子 … 58
3.2.2 列挙型指定子 … 60
3.2.3 タグ … 61
3.3 型修飾子 … 62
3.4 宣言子 … 63
3.4.1 ポインタ宣言子 … 63
3.4.2 配列宣言子 … 64
3.4.3 関数宣言子（プロトタイプ宣言を含む）… 64
3.5 型名 … 65
3.6 typedef … 66
3.7 初期化 … 68
3.7.1 静的記憶域期間を持つオブジェクトの初期化 … 68
3.7.2 自動記憶域期間を持つオブジェクトの初期化 … 68
3.7.3 文字配列の初期化 … 69
3.7.4 集成体，共用体オブジェクトの初期化 … 69

第 4 章　型の変換 … 72
4.1 算術オペランド … 74
4.2 他のオペランド … 76

第 5 章　演算子と式 … 77
5.1 一次式 … 79
5.2 後置演算子 … 80
5.3 単項演算子 … 87
5.4 キャスト演算子 … 94
5.5 算術演算子 … 96
5.6 ビット単位のシフト演算子 … 102
5.7 関係演算子 … 105
5.8 ビット単位の論理演算子 … 112
5.9 論理演算子 … 116
5.10 条件演算子 … 119
5.11 代入演算子 … 121
5.12 カンマ演算子 … 124
5.13 定数式 … 126

第 6 章　C 言語の制御構造 … 128
6.1 ラベル付き文 … 130
6.2 複合文（ブロック）… 134
6.3 式文と空文 … 135
6.4 選択文 … 136
6.5 繰り返し文 … 139
6.6 分岐文 … 143

第 7 章　構造体と共用体 … 148
7.1 構造体 … 148
7.2 共用体 … 151

第 8 章　外部定義 … 154
8.1 関数定義 … 155
8.2 外部オブジェクト定義 … 157

第 9 章　前処理指令（コンパイラに対する指令）… 158
9.1 条件付きコンパイル … 158
9.2 ソース・ファイルの取り込み … 166
9.3 マクロ置換 … 170
9.4 行制御 … 175
9.5 #error 前処理指令 … 176
9.6 #pragma（プラグマ）指令 … 177
9.7 空指令（Null 指令）… 178
9.8 コンパイラ定義のマクロ名 … 179

第 10 章　ライブラリ関数 … 181
10.1 関数間のインタフェース … 181
10.1.1 引数 … 181
10.1.2 返り値 … 182
10.1.3 個々のライブラリによる使用レジスタの保存 … 183
10.2 ヘッダ・ファイル … 185
10.3 リエントラント性 … 192
10.4 標準ライブラリ関数 … 193
10.4.1 引数／返り値に適したライブラリの使用 … 197
10.5 文字／文字列関数 … 198
10.6 プログラム制御関数 … 203
10.7 特殊関数 … 205
10.8 入出力関数 … 208
10.9 ユーティリティ関数 … 227
10.10 文字列／メモリ関数 … 250
10.11 数学関数 … 267
10.12 診断関数 … 314
10.13 スタートアップ・ルーチン，ライブラリ関数更新用バッチ・ファイル … 316
10.13.1 バッチ・ファイルの使用法 … 317

第 11 章　拡張機能 … 319
11.1 マクロ名 … 319
11.2 キーワード … 320
11.3 メモリ … 322
11.4 #pragma 指令 … 324
11.5 拡張機能の使用方法 … 326
11.6 C ソースの修正 … 426
11.7 関数呼び出しインタフェース … 427
11.7.1 返り値 … 427
11.7.2 通常関数呼び出しインタフェース … 428
11.7.3 norec 関数呼び出しインタフェース … 431

第 12 章　アセンブラとの相互参照 … 432
12.1 引数・自動変数のアクセス方法 … 432
12.2 返り値の格納方法 … 433
12.3 C 言語からアセンブリ言語ルーチンの呼び出し … 434
12.3.1 C 言語の関数呼び出し手順 … 434
12.3.2 アセンブリ言語ルーチンの情報退避とリターン … 435
12.4 アセンブリ言語から C 言語ルーチンの呼び出し … 437
12.4.1 アセンブリ言語の関数呼び出し … 437

12.5 他言語で定義された変数の参照 … 439
12.5.1 C 言語で定義した変数を参照する方法 … 439
12.5.2 アセンブリ言語で定義した変数を C 言語側で参照する方法 … 440
12.6 その他注意事項 … 441

第 13 章　効率の良いコンパイラの活用法 … 442
13.1 効率の良いコーディング … 442

付録 A　saddr 領域のラベル一覧 … 446

付録 B　セグメント名一覧 … 449
B.1 セグメント名一覧 … 450
B.1.1 プログラム領域，データ領域 … 450
B.1.2 フラッシュ・メモリ領域 … 451
B.2 セグメントの配置 … 452
B.3 C ソース例 … 453
B.4 出力アセンブラ・モジュール例 … 454

付録 C　ランタイム・ライブラリ一覧 … 462

付録 D　ライブラリ消費スタック一覧 … 466

付録 E　ライブラリ最大割り込み禁止時間一覧 … 476

総合索引 … 477

図の目次

図番号　タイトル, ページ

1-1　翻訳の流れ … 16
1-2　CC78K0R によるプログラム開発手順 … 19
2-1　型分類 … 38
6-1　選択文の制御の流れ … 136
6-2　繰り返し文の制御の流れ … 139
6-3　分岐文の制御の流れ … 143
11-1　メモリ空間の利用 … 323

表の目次

表番号　タイトル, ページ

1-1　C コンパイラの最大値 … 23
1-2　実行速度の向上方法一覧 … 25
1-3　拡張機能一覧 … 26
2-1　文字集合中で使用可能な文字一覧 … 30
2-2　英字拡張表記一覧 … 31
2-3　3 文字表記一覧 … 31
2-4　ANSI-C キーワード一覧 … 32
2-5　CC78K0R 用に追加されたキーワード一覧 … 33
2-6　識別子一覧 … 34
2-7　整数定数 … 47
2-8　整数定数と表現可能な型 … 47
2-9　演算子一覧 … 50
3-1　記憶域クラス指定子 … 55
3-2　型指定子 … 57
3-3　型名の例 … 65
4-1　型変換一覧 … 72
5-1　演算子の評価順序 … 78
5-2　除算／剰余算の演算結果の符号 … 96
5-3　シフト演算 … 102
8-1　外部オブジェクト定義の例 … 157
9-1　マクロ名一覧 … 179
10-1　第 1 引数受け渡し一覧 … 182
10-2　返り値格納一覧 … 182
10-3　標準ライブラリ関数一覧 … 193
10-4　ライブラリ関数更新用バッチ・ファイル … 316
11-1　追加キーワード一覧 … 320
11-2　#pragma 指令リスト … 325
11-3　返り値の格納場所 … 427
11-4　第 1 引数の渡し場所（関数呼び出し側）… 428
B-1　セグメントの配置 … 452
C-1　ランタイム・ライブラリ … 462
D-1　標準ライブラリのスタック消費量一覧 … 466
D-2　ランタイム・ライブラリのスタック消費量一覧 … 472
E-1　ライブラリの最大割り込み禁止時間（クロック数）… 476

第 1 章 概説

この章では，システム開発時における CC78K0R の役割，および，機能概要について説明します。

78K0R シリーズ　C コンパイラは，78K0R シリーズの C 言語，または ANSI-C で記述されたソース・プログラムを機械語に変換する言語処理プログラムです。78K0R シリーズ　C コンパイラにより，78K0R シリーズのオブジェクト・ファイル，またはアセンブラ・ソース・ファイルを得ることができます。

1.1 C 言語とアセンブリ言語

マイクロコンピュータに仕事をさせるには，プログラムやデータが必要です。これをユーザがプログラミングして，マイクロコンピュータのメモリ部に記憶させます。マイクロコンピュータが扱えるプログラムやデータは 2 進数の集まりで，これを機械語といいます。

この機械語に英語の略記号を 1 対 1 で対応させたものがアセンブリ言語です。アセンブリ言語は，機械語と 1 対 1 で対応しているため，マイクロコンピュータに対して詳細な指示を与えることができます（たとえば，入出力時の処理速度の向上など）。しかし，このことはマイクロコンピュータのあらゆる動作を 1 つ 1 つ指示しなければならないことを意味しています。そのために，プログラムの論理構造が一目見ただけでは理解しにくく，また，エラーなども発生しやすいものです。

このようなアセンブリ言語に代わるものとして，高級言語が開発されました。その中の 1 つに C 言語があります。これにより，ユーザは，マイクロコンピュータのアーキテクチャを気にせずにプログラミングすることができ，プログラム自体もアセンブリ言語に比べて論理構造などが理解しやすくなったといえます。

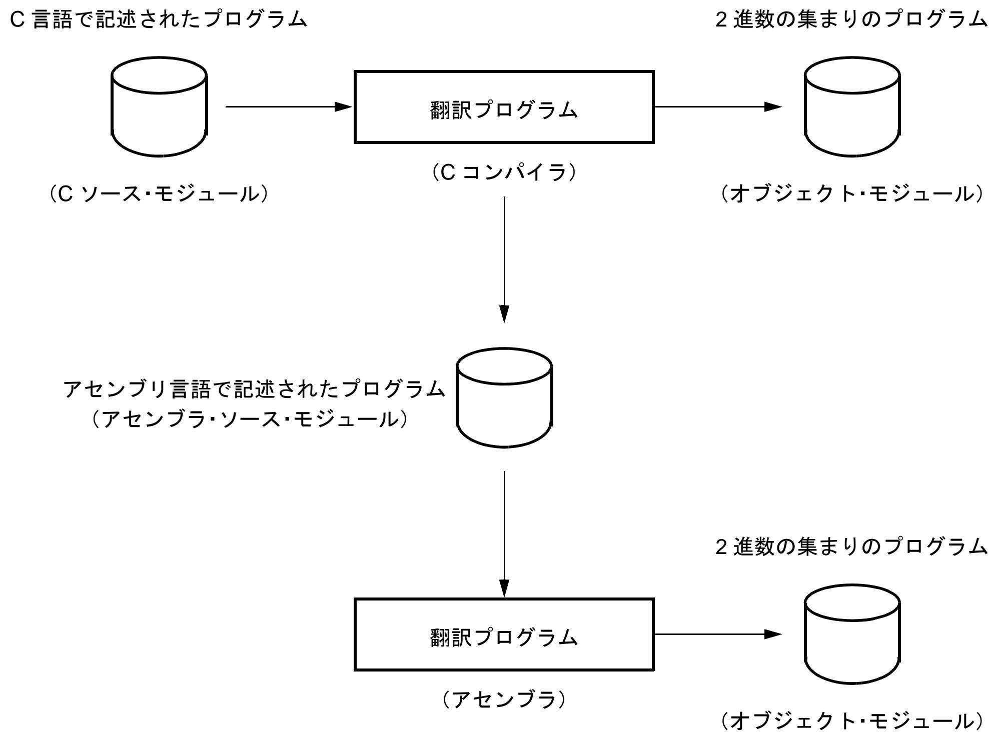
また，C 言語ではプログラムを作成するための多くの部品（関数）が用意されているので，ユーザはこれらを組み合わせてプログラムを作成することができます。

C 言語は，ユーザにとって理解しやすいという特徴を持っています。しかし，C 言語で書かれたプログラムのままでは，マイクロコンピュータは理解することができません。C 言語を理解させるには，それに相当する機械語に翻訳するプログラムが必要となります。この C 言語を機械語に翻訳する翻訳プログラムを C コンパイラと呼びます。

C コンパイラは，C ソース・モジュールを入力し，オブジェクト・モジュールとアセンブラ・ソース・モジュールを出力します。したがって，ユーザは C 言語を用いてプログラムを作成し，プログラムの実行の細部まで指示したい場合には，アセンブリ言語でプログラムを修正することができます。

C コンパイラの翻訳の流れを，次に示します。

図 1-1 翻訳の流れ



1.2　C コンパイラによる開発手順

C コンパイラによる製品開発には，C コンパイラによって生成されたオブジェクト・モジュール・ファイルを連結するためのリンカや，ライブラリ・ファイルの作成を行うライブラリアン，また，プログラムのバグ取りのためのデバッガが必要になります。

1.2.1　必要となるソフトウエア

C コンパイラに関連して必要となるソフトウエアを次に示します。

- エディタ
  ソース・モジュール・ファイルの作成
- RA78K0R　アセンブラ・パッケージ

プログラム名	機能
アセンブラ	アセンブラ・ソース・モジュール・ファイルのアセンブル
リンカ	オブジェクト・モジュール・ファイルの結合，リロケータブル・セグメントの配置アドレス決定
オブジェクト・コンバータ	HEX 形式オブジェクト・モジュール・ファイルへの変換
ライブラリアン	ライブラリ・ファイルの作成
リスト・コンバータ	アブソリュート・アセンブル・リスト・ファイルの生成
PM+	統合開発環境

- 統合デバッガ（78K0R 用）
  C ソース・モジュール・ファイルのデバッグ

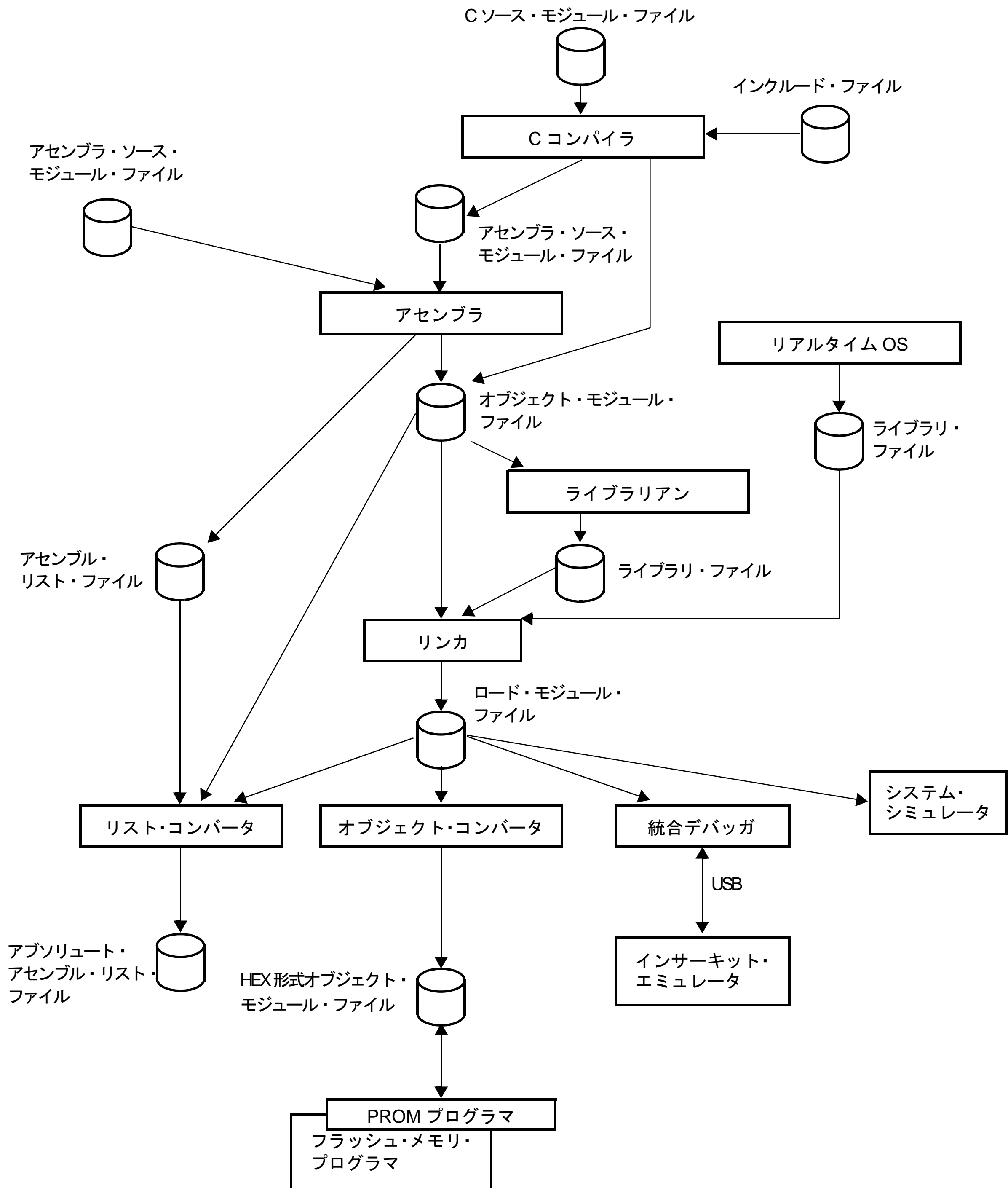
1.2.2 製品開発手順

C コンパイラによる製品開発手順は，次のようになります。

1. 製品の機能分けを行う
2. 機能ごとに C ソース・モジュール・ファイルを作成する
3. 各 C ソース・モジュール・ファイルをコンパイルする
4. 使用頻度の高いオブジェクト・モジュールをライブラリ化する
5. 各オブジェクト・モジュールをリンクする
6. オブジェクト・モジュールのデバッグを行う
7. オブジェクト・コンバータにより HEX 形式オブジェクト・モジュール・ファイルに変換する

C コンパイラは，C ソース・モジュール・ファイルをコンパイルしてオブジェクト・モジュール・ファイル，またはアセンブラ・ソース・モジュール・ファイルを生成します。生成されたアセンブラ・ソース・モジュール・ファイルにより，手作業による最適化（ハンド・オプティマイズ）を行うことができ，効率のよいオブジェクト・モジュールを作成することができます。特に，高速な処理を必要とする場合，またはオブジェクト・モジュールをコンパクトにしたい場合などに有効です。

図 1-2 CC78K0R によるプログラム開発手順

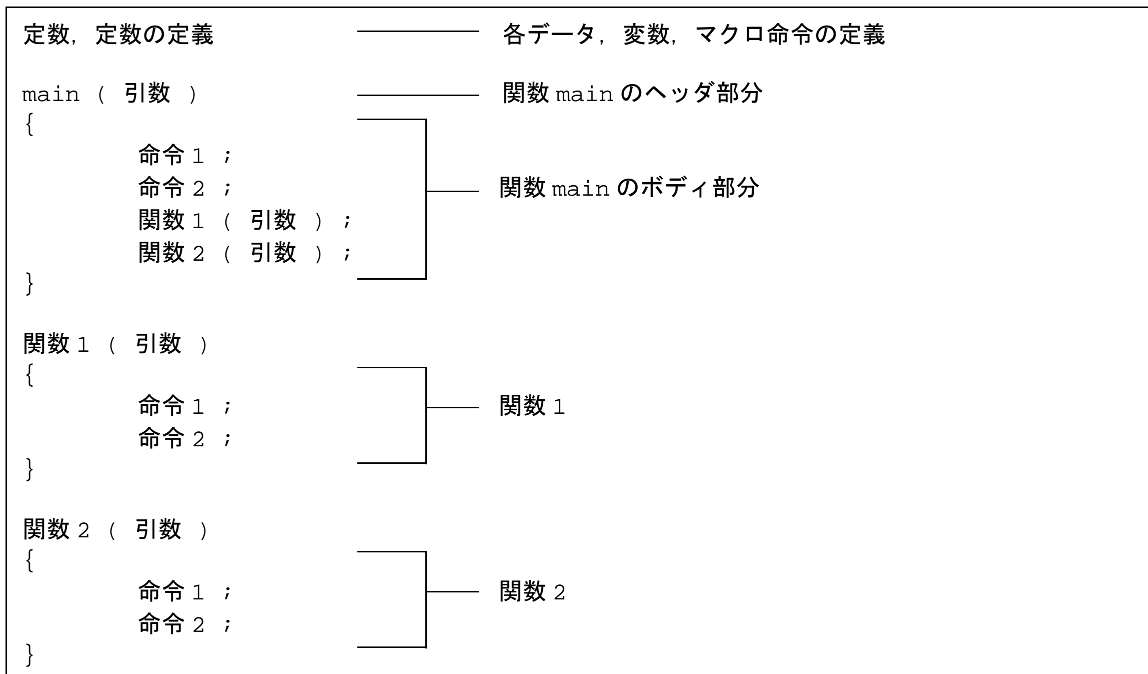


1.3　C ソース・プログラムの基本構成

1.3.1　プログラム形式

C 言語のプログラムは，関数の集まりです。関数は，それぞれ独立した機能を持つように作成します。そして，関数“main”によって 1 つのプログラムにまとめます。C 言語のメイン・ルーチンは，関数“main”になります。

関数は，関数名と引数を定義するヘッダ部分と，プログラムの本体を示すボディ部分からなります。次に C 言語のプログラム形式を示します。



実際の C ソース・プログラムでは，次のようになります。

```
#define TRUE    1                 ← #define xxx xxx        前処理指令（マクロ定義）
#define FALSE   0                 ← #define xxx xxx        前処理指令（マクロ定義）
#define SIZE    200               ← #define xxx xxx        前処理指令（マクロ定義）

void    printf ( char* , int ) ;  ← xxx xxxx ( xxx , xxx ) 関数プロトタイプ宣言
void    putchar ( char ) ;        ← xxx xxxx ( xxx )       関数プロトタイプ宣言

char    mark [ SIZE + 1 ] ;       ← char xxx               型宣言，外部定義
                                    xx [ xx ]              演算子
main ( )
{
    int     i , prime , k , count ;       ← int xxx                 型宣言

    count = 0 ;                           ← xx = xx                 演算子

    for ( i = 0 ; i <= SIZE ; i ++ )      ← for ( xx ; xx ; xx ) xxx ; 制御構造
        mark [ i ] = TRUE ;

    for ( i = 0 ; i <= SIZE ; i ++ ) {
        if ( mark [ i ] ) {
            prime = i + i + 3 ;           ← xxx = xxx + xxx + xxx   演算子
            printf ( "%6d" , prime ) ;    ← xxx ( xxx ) ;           演算子

            count ++ ;
           if ( ( count%8 ) == 0 ) putchar ( '¥n' ) ;  ← if ( xxx ) xxx ; 制御構造
            for ( k = i + prime ; k <= SIZE ; k += prime )
                mark [ k ] = FALSE ;
        }
    }
    printf ( "¥n%d primes found." , count ) ;   ← xxx ( xxx ) ;     演算子
}

void    printf ( char *s , int i )
{
    int     j ;
    char    *ss ;

    j = i ;
    ss = s ;
}

void    putchar ( char c )
{
    char    d ;

    d = c ;
}
```

(1) 型，記憶クラスの宣言

オブジェクトを示す識別子の型，および記憶クラスの宣言です。

型，記憶クラスの詳細については，「第 3 章 型，記憶域クラスの宣言」を参照してください。

(2) 演算子，式

算術演算，論理演算，代入などを行います。

演算子と式の詳細については，「第 5 章 演算子と式」を参照してください。

(3) 制御構造

プログラムの流れを指定します。C 言語の制御構造には，選択，繰り返し，分岐それぞれ数個の命令が用意されています。

制御構造の詳細については，「第 6 章 C 言語の制御構造」を参照してください。

(4) 構造体，共用体

構造体，または共用体を宣言します。構造体は，異なる型の連続した領域を持つオブジェクトで，共用体は，異なる型の重なり合う領域を持つオブジェクトです。

構造体と共用体の詳細については，「第 7 章 構造体と共用体」を参照してください。

(5) 外部定義

関数，または外部オブジェクトを定義します。関数は，C 言語プログラムを機能別に分けたときの 1 つの要素です。C 言語のプログラムは，関数の集まりによって構成されます。

外部定義の詳細については，「第 8 章 外部定義」を参照してください。

(6) 前処理指令

コンパイラに対する命令です。“#define”は，C コンパイラに対してプログラム中に第 1 オペランドと同じものが現れたら第 2 オペランドに置き換えることを指令します。

前処理指令の詳細は，「第 9 章 前処理指令（コンパイラに対する指令）」を参照してください。

(7) 関数プロトタイプ宣言

関数の戻り値と引数の型を宣言します。

1.4 C コンパイラの最大値

実際にプログラム開発をはじめる前に，次のことを参照してください。

表 1-1 C コンパイラの最大値

項目	最大値
複文，繰り返し制御文，選択制御文のネスト	45
条件コンパイルのネスト	255
1 つの宣言中の 1 つの算術型，構造体型，共用体型，または不完全型を修飾するポインタ，配列，および関数宣言子（の任意の組み合わせ）の個数	12
式中のかっこのネスト	32
マクロ名で意味を持つ文字数	256
内部，外部シンボル名で意味を持つ文字数	249
1 ソース・モジュール・ファイル中のシンボル数	1,024 注 1
1 ブロックでブロック・スコープを持つシンボル数	255 注 1
1 ソース・モジュール・ファイル中のマクロ数	10,000
関数定義，関数呼び出しのパラメータ	39
1 つのマクロ定義，マクロ呼び出しのパラメータ	31
1 つの論理ソース行の文字数	2,048
結合後の文字列リテラル内の文字数	509
1 つのオブジェクト・サイズ（データを示します）	65,535 バイト
#include のネスト	8
switch 文の case ラベル数	257
1 コンパイル単位のソース行数	約 30,000
テンポラリ・ファイルを作成せずにコンパイル可能なソース行数	約 300
関数コールのネスト	40
1 関数内のラベル数	33
1 オブジェクト・モジュールあたりのコード，データ，スタック・セグメントのトータル・サイズ	メモリ・モデルによる注 2
1 つの構造体，または共用体のメンバ数	256
1 つの列挙の列挙定数の数	255
1 つの構造体，共用体における構造体，または共用体のネスト	15
初期化子要素のネスト	15
1 ソース・モジュール・ファイル中の関数定義数	1,000
1 つの完全宣言子におけるかっこで囲まれた宣言子の入れ子のレベル	591
マクロのネスト	200

項目	最大値
-i インクルード・ファイル・パス指定数	64

注 1 テンポラリ・ファイルを使用せずに，メモリ・スペースのみで処理可能な最大値を示します。メモリ・スペースで処理しきれない場合は，テンポラリ・ファイルを使用し，そのときの最大値はファイル・サイズにより変わります。

注 2 メモリ・モデルにより，最大値は以下のようになります。

メモリ・モデル	最大値
スモール・モデル	コード部 64K バイト，データ部 64K バイトの合計 128K バイト
ミディアム・モデル	コード部 1M バイト，データ部 64K バイト
コンパクト・モデル	コード部 64K バイト，データ部 1M バイト
ラージ・モデル	コード部 1M バイト，データ部 1M バイト

1.5 C コンパイラの特長

CC78K0R は，ANSI にない CPU のコードを生成する次の拡張機能を備えています。78K0R シリーズの特殊機能用レジスタを C 言語レベルで記述可能にする機能，オブジェクト・コードを短縮し，実行速度の向上を図る機能があります。

オブジェクト・コードを短縮し，実行速度を向上させる方法としては，次のものがあります。

表 1-2 実行速度の向上方法一覧

方法	拡張機能
callt 領域を利用して関数を呼び出す	callt/__callt 関数
変数をレジスタに割り当てる	レジスタ変数
saddr 領域に変数を割り当てる	sreg/__sreg
sfr 名を使用できる	sfr 領域
前後処理（スタック・フレーム）のない関数を生成する	norec/__leaf 関数
C ソース・プログラム中にアセンブリ言語を記述する	ASM 文
saddr，sfr 領域へのビット・アクセスを行う	bit 型変数，boolean/__boolean 型変数
ビット・フィールドを unsigned char 型で指定できる	ビット・フィールド宣言
乗算するコードを直接インライン展開して出力する	乗算関数
CC78K0 と互換性があり，実行スピードが速く，かつサイズが小さいコードを生成する	除算関数
ローテートするコードを直接インライン展開して出力する	ローテート関数
特定のデータや命令を直接コード領域に埋め込む	データ挿入関数
memcpy，memset を直接インライン展開して出力する	メモリ操作関数

次に CC78K0R の拡張機能の概要を示します。

各拡張機能の詳細については，「第 11 章 拡張機能」を参照してください。

表 1-3 拡張機能一覧

拡張機能	概要
callt 関数（callt/__callt）	呼び出される関数のアドレスが callt テーブル領域に置かれ，関数が呼び出されます。 通常の呼び出し命令 call に比べ，オブジェクト・コードを短縮することができます。
レジスタ変数（register）	レジスタ，または saddr 領域に変数をとることができ，通常の変数を使用した場合と比べ，実行速度が向上します。 また，オブジェクト・コードを短縮することができます。
saddr 領域利用（sreg/__sreg）	変数を saddr 領域に割り当てることができ，通常の変数を使用した場合と比べ，実行速度が向上します。 また，オブジェクト・コードを短縮することができます。変数は，オプションによっても saddr 領域に割り当てることができます。
sfr 領域利用（sfr）	特殊機能レジスタ（sfr）を，sfr の略号（sfr 名）によって C ソース・ファイル中で使用することができます。
norec 関数（norec）	前後処理（スタック・フレーム）のない関数を生成します。 norec/__leaf 関数の呼び出しで，引数はレジスタ渡しになります。 また，norec/__leaf 関数内で使用するオートマティック変数は，レジスタ，あるいは saddr 領域に割り当てられます。これにより，実行速度が向上し，オブジェクト・コードを短縮することができます。 この関数は，引数，オートマティック変数に制限があります。また，この関数から関数呼び出しはできません。 詳細については，「11.5　norec 関数（norec）」を参照してください。
bit 型変数，boolean 型変数（bit/boolean/__boolean）	1 ビットの記憶領域を持つ変数を生成します。 bit 型変数，boolean/__boolean 型変数を使用することにより，saddr 領域へビット・アクセスすることができます。 なお，boolean/__boolean 型変数は，機能，使用方法とも，bit 型変数と同じです。
ASM 文（#asm ～ #endasm/__asm）	C コンパイラが出力したアセンブラ・ソース・ファイルに，ユーザが記述したアセンブラ・ソースが埋め込まれます。
漢字（/* 漢字 */, // 漢字）	C ソース・ファイルのコメント文中に，漢字を記述することができます。 漢字コードには，シフト JIS コード，EUC コードを選択することができます。また，漢字コードなしも選択することができます。
割り込み関数（#pragma vect/#pragma interrupt）	ベクタ・テーブルを生成し，割り込みに対応したオブジェクト・コードを出力します。 これにより，C ソース・レベルで割り込み関数の記述が可能となります。
割り込み関数修飾子（__interrupt, __interrupt_brk）	この修飾子により，ベクタ・テーブルの設定と割り込み関数定義を別ファイルに記述することができます。
割り込み機能（#pragma DI, #pragma EI）	オブジェクトに割り込み禁止命令，割り込み許可命令を埋め込みます。

拡張機能	概要
CPU 制御命令（#pragma HALT/STOP/BRK/NOP）	オブジェクトに次の各命令を埋め込みます。 - halt 命令 - stop 命令 - brk 命令 - nop 命令
ビット・フィールド宣言	ビット・フィールドを unsigned char 型で指定することにより，メモリの節約，オブジェクト・コードの短縮，実行速度の向上を図ることができます。
コンパイラ出力セクション名の変更（#pragma section …）	コンパイラ出力セクション名を変更することにより，リンカでそのセクションを独立に配置することができます。
2 進定数（2 進定数 0bxxx）	C ソース中で，2 進数を記述することができます。
モジュール名変更機能（#pragma name）	C ソース中で，オブジェクト・モジュール名を任意に変更することができます。
ローテート関数（#pragma rot）	オブジェクトに式の値をローテートするコードを，直接インライン展開して出力します。
乗算関数（#pragma mul）	オブジェクトに式の値を乗算するコードを，直接インライン展開して出力します。 この関数により，オブジェクト・コードを短縮し，実行速度を向上することができます。
除算関数（#pragma div）	CC78K0 と互換性があり，除算命令の入出力のデータ・サイズを生かしたコードを出力します。 除算式の記述より，実行スピードが速く，かつサイズが小さいコードを生成することができます。
データ挿入関数（#pragma opc）	カレント・アドレスに定数データを挿入します。 アセンブラ記述を使用せずに，特定のデータや命令をコード領域に埋め込めます。
RTOS 用割り込みハンドラ（#pragma rtos_interrupt …）	RX78K0R（リアルタイム OS）用の割り込みハンドラを記述することができます。
RTOS 用割り込みハンドラ修飾子（__rtos_interrupt）	RX78K0R（リアルタイム OS）用の割り込みハンドラ記述とベクタ設定を別ファイルにするための修飾子です。
RTOS 用タスク関数（#pragma rtos_task）	#pragma 指令により，指定された関数を RX78K0R（リアルタイム OS）用のタスクと解釈します。 これにより，C ソース・レベルで，コード効率の良いリアルタイム OS 用タスク関数の記述が可能となります。
フラッシュ領域配置方法（-zf）	コンパイル時に -zf オプションを指定することにより，プログラムをフラッシュ領域に配置したり，-zf オプションを指定せずに作成したブート領域のオブジェクトと結合して使用できるようになります。
フラッシュ領域分岐テーブル（#pragma ext_table）	フラッシュ領域分岐テーブルの先頭アドレスを #pragma 指令により指定することにより，スタートアップ・ルーチン，割り込み関数をフラッシュ領域に配置したり，ブート領域からフラッシュ領域への関数呼び出しを行うことができます。
ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）	ブート領域から呼び出すフラッシュ領域中の関数名，および ID 値を #pragma 指令により指定することにより，ブート領域からフラッシュ領域中の関数を呼び出せるようになります。
ファーム ROM 関数（__flash）	インタフェース・ライブラリのプロトタイプ宣言時に，__flash 属性を先頭に追加することにより，ファーム ROM 関数に関する操作を C ソース・レベルで記述することができます。

拡張機能	概要
引数／返り値の int 拡張抑制方法（-zb）	コンパイル時に -zb オプションを指定することにより，オブジェクト・コードの短縮，実行速度の向上を図ることができます。
メモリ操作関数（#pragma inline）	#pragma inline 指令により，標準ライブラリ関数 memcpy，memset を関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。 これにより，実行速度の向上を図ることができます。
絶対番地配置指定（__directmap）	絶対番地に配置する変数を定義したいモジュール中で __directmap 宣言を行うことにより，任意のアドレスに変数を配置することができ，同じアドレスに複数の変数を重ねて配置することができます。
near/far 領域指定	関数，変数の宣言時に，__near/__far 型修飾子を追加することにより，関数，変数の配置場所を明示的に指定することができます。
メモリ・モデル指定	コンパイル時に，-ms/-mm/-mc/-ml オプションを指定することにより，関数，変数の配置場所をメモリ・モデルで指定することができます。

第 2 章　C 言語の基本構成

この章では，C ソース・モジュール・ファイルの構成要素の説明を行います。

C ソース・モジュール・ファイルは，次の字句から構成されます。

キーワード	識別子	定数
文字列リテラル	演算子	区切り子
ヘッダ名	前処理数	コメント

次に，C プログラム記述例で使用されている字句を示します。

```
#include    "expand.h"

extern      void    testb ( void ) ;          ← extern          キーワード
extern      void    chgb ( void ) ;
extern      bit     data1 ;                   ← data1, data2    識別子
extern      bit     data2 ;

void    main ( )                              ← void            キーワード
{
        data1 = 1 ;                           ← 1               定数
        data2 = 0 ;                           ← 0               定数

        while ( data1 ) {                     ← while           キーワード
                data1 = data2 ;               ← { }             区切り子
                testb ( ) ;                   ← =               演算子
        }

        if ( data1 && data2 ) {               ← if              キーワード
                chgb ( ) ;                    ← &&              演算子
        }                                     ← ( )             演算子
}

void    lprintf ( char *s , int i )           ← lprintf         識別子
{                                             ← char, int       キーワード
        int     j ;                           ← s, i            識別子
        char    *ss ;                         ← *               演算子

        j = i ;
        ss = s ;
}
          :
```

2.1 文字セット

2.1.1 文字集合

C プログラムで使用する文字集合には，ソース・ファイルを記述するソース文字の集合と，実行環境で解釈される実行文字の集合があります。

実行文字集合中の文字の値は JIS コードです。

ソース文字集合，および実行文字集合中では，次の文字を使用することができます。

コメント中では，さらに全角文字のすべて，半角カタカナ（半角の句読点等を含む）を使用することができます。

表 2-1 文字集合中で使用可能な文字一覧

26 個の英大文字	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
26 個の英小文字	a b c d e f g h i j k l m n o p q r s t u v w x y z
10 個の数字	0 1 2 3 4 5 6 7 8 9
29 個の図形文字	! " # % & ' ( ) * + , - . / : ; < = > ? [¥] ^ _ { | } ~
スペース，水平タブ，垂直タブ，改ページなどを示す制御文字	

2.1.2 マルチバイト文字

コメントに記述することができるのは，シフト JIS コード，EUC コードです。

コメント以外で記述することができるのは，ASCII コードが 0x7F 以下の文字です。

具体的には，全角文字のすべて，および半角カタカナ（半角の句読点等を含む）をコメント以外には記述することができません。

2.1.3 英字拡張表記（エスケープ・シーケンス）

警報や改ページなどの非図形文字は，英字拡張表記によって表現します。

英字拡張表記は，円記号“¥”とアルファベット 1 文字からなります。

非図形文字を表現する英字拡張表記を，次に示します。

表 2-2 英字拡張表記一覧

英字拡張表記	意味	文字コード
¥a	警報	07H
¥b	バック・スペース	08H
¥f	改ページ	0CH
¥n	改行	0AH
¥r	復帰	0DH
¥t	水平タブ	09H
¥v	垂直タブ	0BH

2.1.4 3 文字表記（トライグラフ・シーケンス）

次に示す左側の三つの文字の並び（“3 文字表記”という）がソース・ファイル中にある場合，その 3 つの文字の並びを右側の対応する 1 文字に置き換えます。

コンパイラ・オプション -za（ANSI 規定外の機能を無効とし，ANSI 規定の一部の機能を有効とするオプション）を指定することで有効になります。

表 2-3 3 文字表記一覧

3 文字表記	意味
??=	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!		
??>	}
??-	~

2.2 キーワード

2.2.1 ANSI-C キーワード

次の字句は，コンパイラによってキーワードとして使用されるので，ラベルや変数名として使用することはできません。

表 2-4 ANSI-C キーワード一覧

auto	break	case	char	const	continue	default
do	double	else	enum	extern	for	float
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

2.2.2 CC78K0R 用に追加されたキーワード

CC78K0R では，拡張機能を実現するために次の字句をキーワードとして追加しています。

これらの字句も ANSI キーワードと同様，ラベルや変数名として使用することはできません（大文字が含まれる場合は，キーワードとみなされません）。

ANSI-C 言語仕様のみを許可するオプション（-za）指定により，“__”で始まらないキーワードを無効にすることができます。

callf，__callf，noauto，__banked，__non_banked，__BANK0 ～ 15，__pascal，__temp，__mxcall は，CC78K0 との互換性を考慮して，キーワードとします。

表 2-5 CC78K0R 用に追加されたキーワード一覧

キーワード	用途
__callt/callt	callt 関数の宣言
__callf/callf	callf 関数の宣言
__sreg/sreg	sreg 変数の宣言
noauto	noauto 関数の宣言
__leaf/norec	norec/leaf 関数の宣言
bit	bit 型変数の宣言
__boolean/boolean	boolean 型変数の宣言
__interrupt	ハードウエア割り込み関数
__interrupt_brk	ソフトウエア割り込み関数
__banked，__non_banked	バンク・インタフェース注 1
__BANK0 ～ 15	定数番地のバンク関数
__asm	asm 文
__rtos_interrupt	リアルタイム OS 用割り込みハンドラ
__pascal	パスカル関数
__flash	ファーム ROM 関数
__flashf	__flashf 関数注 2
__directmap	絶対番地配置指定
__temp	テンポラリ変数
__near，__far	メモリ配置領域指定
__mxcall	__mxcall 関数注 3

注 1 関数情報ファイル用に予約しているキーワードです。C ソース中には記述しないでください。

注 2 CC78K0R で予約しているキーワードです。ユーザは使用しないでください。

注 3 MX とのインタフェース用に予約しているキーワードです。ユーザは使用しないでください。

2.3 識別子

識別子は，次のものを示します。

表 2-6 識別子一覧

関数 オブジェクト 構造体，共用体，および列挙のタグ 構造体，共用体，および列挙のメンバ typedef 名 ラベル名 マクロ名 マクロ仮引数

識別子は，アンダースコアを含めた英大文字と英小文字，および数字で表します。

識別子として使用可能な文字を次に示します。

なお，識別子の最大の長さに関して制限はありません。ただし，CC78K0R で認識することができるのは，最初の 249 文字です。

```
_（アンダースコア）

a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z

0  1  2  3  4  5  6  7  8  9
```

識別子の先頭に数字を使用することはできません。また，識別子はキーワードと同じ名前にしてはなりません。

2.3.1 識別子の有効範囲

識別子は，宣言された場所により，その識別子が使用可能な有効範囲が決まります。

識別子の有効範囲には，次のものがあります。

- 関数有効範囲
- ファイル有効範囲
- ブロック有効範囲
- 関数プロトタイプ有効範囲

```
extern   __boolean       data1 , data2 ; ← data1, data2    ファイル有効範囲

void     testb ( int x ) ;               ← x               関数プロトタイプ有効範囲

void     main ( void )
{
         int      cot ;                  ← cot             ブロック有効範囲
         data1 = 1 ;
         data2 = 0 ;

         while ( data1 ) {
                  data1 = data2 ;
             j1 :                        ← j1              関数有効範囲
                  testb ( cot ) ;
         }
}

void     testb ( int x )                 ← x               ブロック有効範囲
{
   :
}
```

(1) 関数有効範囲

関数有効範囲は，関数内全体を指します。

関数有効範囲を持つ識別子は，指定された関数内のどこからでも参照することができます。関数有効範囲を持つ識別子は，ラベル名だけです。

(2) ファイル有効範囲

ファイル有効範囲は，コンパイル単位全体を指します。

ブロック，またはパラメータ・リストの外で宣言された識別子は，ファイル有効範囲を持ちます。ファイル有効範囲を持つ識別子は，プログラム中のどこからでも参照することができます。

(3) ブロック有効範囲

ブロック有効範囲は，対になったブロック（中かっこ“{ }”で囲まれたところ）を閉じるまでの範囲を指します。

ブロック，またはパラメータ・リストの中で宣言された識別子は，ブロック有効範囲を持ちます。ブロック有効範囲を持つ識別子は，指定したブロック内で有効です。

(4) 関数プロトタイプ有効範囲

宣言された関数の終わりまでの範囲を指します。

関数プロトタイプ内のパラメータ・リストの中で宣言された識別子は，関数プロトタイプ有効範囲を持ちます。関数プロトタイプ有効範囲を持つ識別子は，指定された関数内で有効です。

2.3.2 識別子の結合

異なった，または同一の有効範囲内で 2 回以上宣言された識別子が，同じオブジェクト，あるいは関数として参照できるようになることを識別子の結合といいます。

識別子は，結合されることにより，同一のものであるとみなされます。

識別子の結合には，外部結合と内部結合，および無結合があります。

(1) 外部結合

外部結合は，プログラム全体を構成するコンパイル単位，およびライブラリの集まりで結合されるものです。

外部結合の例を次に示します。

- 記憶クラスを指定せずに宣言された関数
- extern 宣言されたオブジェクト，あるいは関数で，参照する識別子に記憶クラスの指定がない場合
- ファイル有効範囲を持ち，記憶クラスの指定がないオブジェクト

(2) 内部結合

内部結合は，1 つのコンパイル単位内で結合されるものです。

内部結合の例を次に示します。

- ファイル有効範囲を持ち，記憶クラス指定子 static を含むオブジェクト，または関数

(3) 無結合

無結合は，固有な実体です。

無結合の例を次に示します。

- オブジェクト，あるいは関数以外の識別子
- 関数のパラメータを宣言する識別子
- ブロック内で記憶クラス指定子 extern を持たないオブジェクトの識別子

2.3.3　　識別子の名前空間

すべての識別子は，次に示す“名前空間”に分類されます。

名前空間	説明
ラベル名	ラベルの宣言により区別されます。
構造体，共用体，列挙のタグ名	キーワード struct，union，enum によって区別されます。
構造体，共用体のメンバ名	演算子“ . ”,“ -> ”によって式中で区別されます。
普通の識別子（上記以外の識別子）	通常の宣言子，または列挙定数として宣言されます。

2.3.4　　オブジェクトの記憶域期間

各オブジェクトは，そのライフタイムを決定する“記憶域期間”を持っています。

記憶域期間には，静的（static）記憶域期間と自動（automatic）記憶域期間の 2 つがあります。

(1)　静的記憶域期間

静的記憶域期間を持つオブジェクトは，実行前に領域が確保されます。

確保される領域は，1 回だけ初期化されます。静的オブジェクトは，プログラムの実行中に存在して，最後に格納された値を保持します。

静的記憶域期間を持つオブジェクトを次に示します。

- 外部結合を持つオブジェクト
- 内部結合を持つオブジェクト
- 記憶クラス指定子 static で宣言されたオブジェクト

(2)　自動記憶域期間

自動記憶域期間を持つオブジェクトは，宣言されるブロック内に入るときにオブジェクトの領域が確保されます。

ブロックに頭から入るときに初期化の指定があると，オブジェクトの初期化が行われます。ブロック内のラベルにジャンプして入った場合は，初期化されません。自動記憶域期間を持つオブジェクトの領域は，宣言されたブロックの実行が終わると，保証されません。

自動記憶域期間を持つオブジェクトを次に示します。

- 無結合のオブジェクト
- 記憶クラス指定子 static で宣言されていないオブジェクト

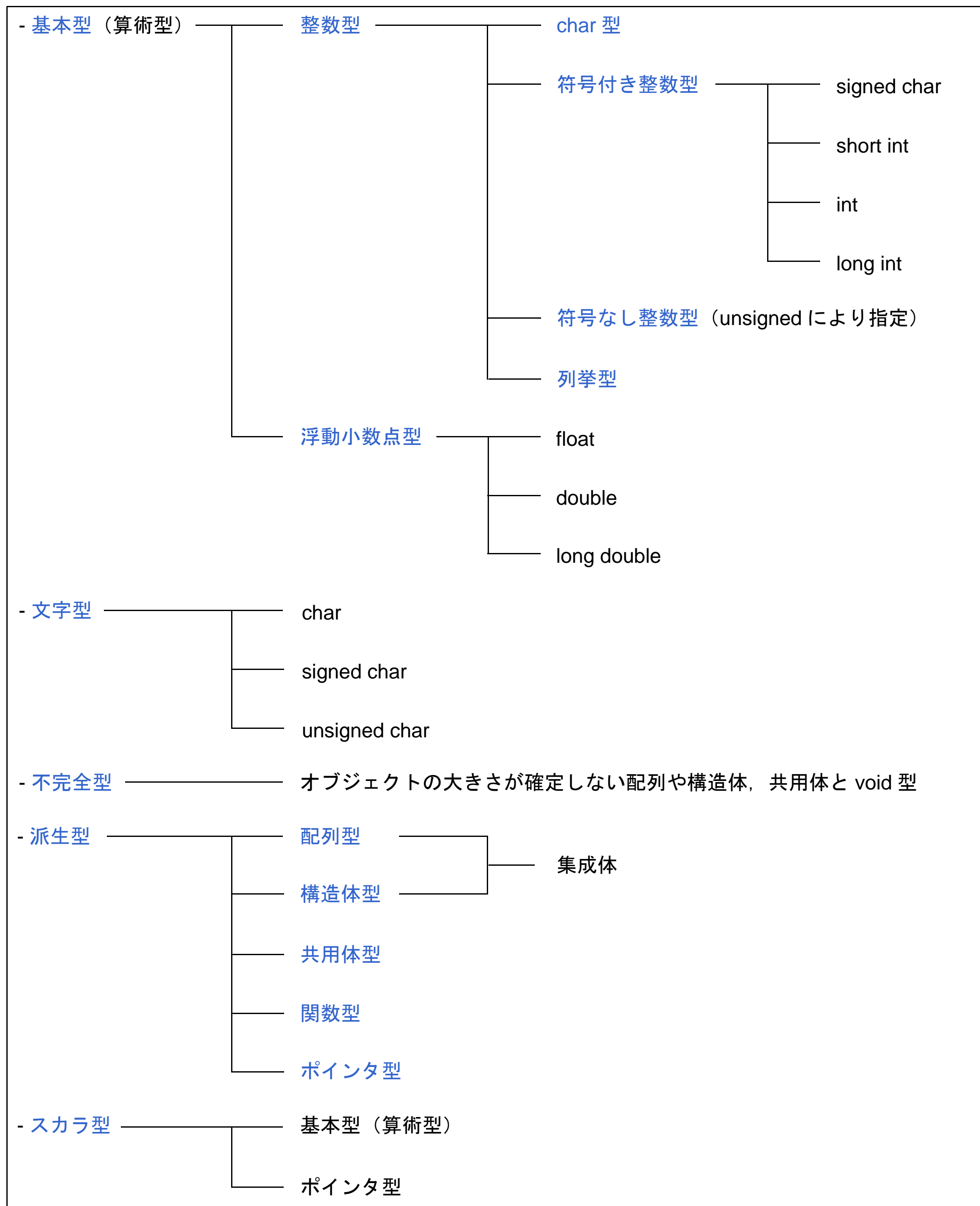
2.4 型

型は，オブジェクトに格納される値の性質を決定し，次の 3 種類に分けられます。

- オブジェクト型　：オブジェクトを表す型
- 関数型　：関数を表す型
- 不完全型　：サイズに関する情報を持たないオブジェクトを表す型

型の分類を，次に示します。

図 2-1 型分類



2.4.1 基本型

基本型は，算術型とも呼ばれ，整数型と浮動小数点型からなります。

また，整数型は，char 型，符号付き整数型，符号なし整数型，列挙型に分類されます。

(1) 整数型

整数型には，次の 4 種類の型があります。整数型の値は，2 進数 0 と 1 によって表現されます。

- char 型
- 符号付き整数型
- 符号なし整数型
- 列挙型

(a) char 型

char 型は，実行文字集合の任意の文字を格納するのに十分な大きさを持っています。

char オブジェクトに格納される文字の値は，正になります。

文字以外のものは，符号付き整数として扱われます。

格納する際，オーバーフローが生じるとオーバーフローした部分は無視されます。

(b) 符号付き整数型

符号付き整数型には，次の 4 種類の型があります。

- signed char
- short int
- int
- long int

signed char 型で宣言されるオブジェクトは，修飾子がない char と同じ大きさの領域を持ちます。

修飾子がない int オブジェクトは，実行環境の CPU アーキテクチャにとって自然な大きさを持ちます。

符号付き整数型には，それに対応する符号なし整数型があり，ともに同じ大きさの領域を使用します。

符号付き整数型の正の数は，符号なし整数型の部分集合です。

(c) 符号なし整数型

符号なし整数型は，キーワード unsigned で示されるものです。

符号なし整数型を含む計算ではオーバーフローしません。符号なし整数型を含む計算の場合，整数型で表現できない値になると，計算結果は符号なし整数型で表現できる最大数に 1 を加算した値で割った余りに置き換わるからです。

(d) 列挙型

列挙は，名付られた整数定数の集合です。

列挙の並びにより，構成されます。

(2) 浮動小数点型

浮動小数点型には，次の 3 種類の型があります。

- float
- double
- long double

なお，CC78K0R では，double，long double 型は，float と同様に ANSI/IEEE 754-1985 で規定されている，単精度正規化数に対する浮動小数点表現としてサポートします。つまり，float，double，long double 型の値の範囲は同じとなります。

型	値の範囲
（signed）char	-128 ～ +127
unsigned char	0 ～ 255
（signed）short int	-32768 ～ +32767
unsigned short int	0 ～ 65535
（signed）int	-32768 ～ +32767
unsigned int	0 ～ 65535
（signed）long int	-2147483648 ～ +2147483647
unsigned long int	0 ～ 4294967295
float	1.17549435E - 38F ～ 3.40282347E + 38F
double	1.17549435E - 38F ～ 3.40282347E + 38F
long double	1.17549435E - 38F ～ 3.40282347E + 38F

備考 1　signed は省略可能です。ただし，char 型の signed を省略した場合，コンパイル時の条件（オプション）により signed char，または unsigned char と判断されます。

備考 2　short int と int は，同じ値の範囲を持ちますが，異なる型として扱われます。

備考 3　unsigned short int と unsigned int は，同じ値の範囲を持ちますが，異なる型として扱われます。

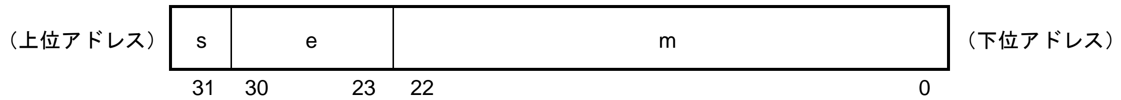
備考 4　float，double，long double は，同じ値の範囲を持ちますが，異なる型として扱われます。

備考 5　float，double，long double の値の範囲は，絶対値の範囲です。

[浮動小数点数（float 型）の仕様]

(a)　フォーマット

浮動小数点数のフォーマットを次に示します。



この形式の数値は，次のようになります。

$$(-1)^{(\text{サイン部値})} * (\text{仮数部値}) * 2^{(\text{指数部値})}$$

<table>
<tr><td>s</td><td>サイン部（1 ビット）
正数の場合は 0，負数の場合は 1 をとります。</td></tr>
<tr><td>e</td><td>指数部（8 ビット）
底 2 の指数を 1 バイトの整数型（負の場合 2 の補数表現）で表し，この値にさらに 7FH のバイアスを加えた値を用いています。これらの関係を，次に示します。
<table>
<tr><th>指数部（16 進）</th><th>指数部の値</th></tr>
<tr><td>FE
:
81
80
7F
7E
:
01</td><td>127
:
2
1
0
-1
:
-126</td></tr>
</table></td></tr>
<tr><td>m</td><td>仮数部（23 ビット）
仮数部は絶対値で表現され，仮数部のビット位置 22 ～ 0 が 2 進数の小数点第 1 位～第 23 位に相当します。
仮数部値は浮動小数点値が 0 になる場合を除いて，常に 1 ～ 2 の範囲になるように指数部の値を調整します（正規化）。そのため，1 の位（1 の値を意味する）は常に 1 となり，この形式では省略した形で表現しています。</td></tr>
</table>

(b)　ゼロの表現

指数部 = 0，かつ仮数部 = 0 のとき，次のように± 0 を表現します。

$$(-1)^{(\text{サイン部値})} * 0$$

(c)　無限大の表現

指数部 = FFH，かつ仮数部 = 0 のとき，次のように±∞を表現します。

$$(-1)^{(サイン部値)} * \infty$$

(d)　非正規化数

指数部 = 0，かつ仮数部 ≠ 0 のとき，次のように非正規化数を表現します。

$$(-1)^{(サイン部値)} * (仮数部値) * 2^{-126}$$

備考　ここでの仮数部値は，1 未満の数値であり，仮数部のビット位置 22 〜 0 がそのまま小数点第 1 位〜 23 位を表現します。

(e)　非数（NaN）の表現

指数部 = FFH，かつ仮数部 ≠ 0 のとき，サイン部にかかわらず非数を表現します。

(f)　演算結果の丸め処理

最近偶数への不偏丸めを行います。演算結果が上記の浮動小数点のフォーマットで表現できない場合，表現可能な最も近い値に丸めます。
丸め以前の値に対して等差の表現可能な値が 2 つある場合，偶数（2 進表現の最下位ビットが 0 となる数）に丸めます。

(g)　演算例外

演算例外には，次の 5 種類があります。

例外	返り値
アンダーフロー	非正規化数
消滅（INEXACT）	± 0
オーバーフロー	±∞
ゼロ除算	±∞
演算不能	非数（NaN）

各例外発生時の警告は，matherr 関数を呼び出すことによって行います。

2.4.2 文字型

文字型には，次の 3 種類の型があります。

- char
- signed char
- unsigned char

2.4.3 不完全型

不完全型には，次の 4 つがあります。

- オブジェクトの大きさが確定しない配列
- 構造体
- 共用体
- void 型

2.4.4 派生型

派生型には，次の 5 種類の型があります。

- 配列型
- 構造体型
- 共用体型
- 関数型
- ポインタ型

(1) 配列型

配列型は，要素型と呼ばれるメンバ・オブジェクトの集まりを連続して割り付けることを示します。
メンバ・オブジェクトは，すべて同じ大きさの領域を持ちます。要素型，および配列の要素の個数を指定します。なお，不完全型の配列を作ることはできません。

(2) 構造体型

構造体型は，大きさの異なるメンバ・オブジェクトの集まりを連続して割り付けることを示します。
個々のメンバ・オブジェクトは，名前によって指定することができます。

備考　配列型と構造体型を総称して集成体型と呼びます。集成体型は，メンバ・オブジェクトが連続して取られます。

(3) 共用体型

共用体型は，重なり合うメンバ・オブジェクトの集まりを示します。
個々のメンバ・オブジェクトは，異なる大きさと名前を持ち，個別に指定することができます。

(4) 関数型

関数型は，指定された型の返り値を持つ関数を示します。

返り値の型，パラメータの数，およびパラメータの型を指定します。

返り値の型が T であれば，その関数は T を返す関数と呼ばれます。

(5) ポインタ型

ポインタ型は，被参照型と呼ばれる関数型オブジェクト型，および不完全型から作られます。

ポインタ型は，オブジェクトを表します。オブジェクトが示す値は，被参照型の実体を参照するために使用されます。

被参照型 T から作られるポインタ型は，T へのポインタと呼ばれます。

2.4.5 スカラ型

基本型（算術型）とポインタ型を総称してスカラ型といいます。

スカラ型には，次のものがあります。

- char 型
- 符号付き整数型
- 符号なし整数型
- 列挙型
- 浮動小数点型
- ポインタ型

2.4.6 適合型

2 つの型が同じものであれば，それは適合型と呼ばれます。

たとえば，別々のコンパイル単位で宣言された 2 つの構造体，共用体，または列挙型は，メンバ数，メンバ名が同じで，メンバの型が一致すれば適合型です。このとき，2 つの構造体，共用体は個々のメンバが同じ順序で並び，2 つの列挙では個々のメンバは同じ値を持たなければなりません。

同じオブジェクト，または関数に関係するすべての宣言は，適合型を持たなければなりません。

2.4.7 合成型

合成型は，適合する２つの型から作られます。

合成型では次のことが成り立ちます。

- 片方が型の大きさが決まった配列であれば，その合成型は同じ大きさを持つ配列です。
- 片方だけがパラメータ型リスト（関数プロトタイプ）を持つ関数型であれば，その合成型はパラメータ型リストを持つ関数プロトタイプです。
- 両方の型がパラメータ型リストを持てば，合成パラメータ型リストのパラメータの型は対応するパラメータの合成型です。

＜合成型の例＞

ファイル有効範囲を持つ２つの宣言が次のようであると仮定します。

```
        int f ( int ( * ) ( ) , double ( * ) [ 3 ] ) ;
        int f ( int ( * ) ( char * ) , double ( * ) [ ] ) ;
```

このとき関数の合成型は，次のようになります。

```
        int f ( int ( * ) ( char * ) , double ( * ) [ 3 ] ) ;
```

2.5 定数

定数は，あらかじめ設定しておく値です。個々の定数は，指定した形式，および値によって型が決定されます。

定数には，次の 4 種類があります。

- 浮動小数点定数
- 整数定数
- 列挙定数
- 文字定数

2.5.1 浮動小数点定数

浮動小数点定数は，有効数字部，指数部，浮動小数点接尾語から構成されます。

有効数字部	整数部，小数点，小数部
指数部	e，または E，符号付き指数
浮動小数点接尾語	f / F（float） l / L（long double） **備考**　省略時は，（double）となります。

指数部の符号付き指数と浮動小数点接尾語は，省略可能です。

なお，有効数字部は，整数部，または小数部のいずれかがなくてはなりません。また，小数点，または指数部のいずれかはなくてはなりません（例：1.23F，2e3）。

2.5.2 整数定数

整数定数は，数字ではじまり，小数点，および指数部を持ちません。

整数定数が符号なし（unsigned）であることを示すのに符号なし接尾語を，long であることを示すのに長語接尾語を整数定数の後ろに付けることができます。

```
符号なし接尾語
        u  U
長語接尾語
        l  L
```

整数定数には，次の 3 種類があります。

- 10 進定数
- 8 進定数
- 16 進定数

各整数定数の詳細を，次に示します。

表 2-7　整数定数

整数定数	説明
10 進定数	10 進定数は，10 を基数とする整数値です。 [指定方法] 0 以外の数字を先頭にして，その後ろに 0 〜 9 の数字を続けます（例：56UL）。 0 以外の数字から始まる 10 進数字[注] **注**　10 進数字 = 1 2 3 4 5 6 7 8 9
8 進定数	8 進定数は，8 を基数とする整数値です。 [指定方法] 0 を先頭にして，その後ろに 0 〜 7 の数字を続けます（例：034U）。 整数接尾語 0 + 8 進数字[注] **注**　8 進数字 = 0 1 2 3 4 5 6 7
16 進定数	16 進定数は，16 を基数とする整数値です。 [指定方法] 0x，または 0X を先頭にして，その後ろに 10 進数字，および 10 から 15 の値を表す a（または A）から f（または F）を続けます（例：0xF3）。 整数接尾語 0x，または 0X +16 進数字[注] **注**　16 進数字 = 0 1 2 3 4 5 6 7 8 9　a b c d e f　A B C D E F

整数定数の型は，次に示す“表現可能な型”の最初のものとみなされます。

CC78K0R では，添字なしの定数の型は，コンパイル時の条件（オプション）により，char，または unsigned char に変更することができます。

表 2-8　整数定数と表現可能な型

整数定数	表現可能な型
接尾語なし 10 進数	int，long int，unsigned long int
接尾語なし 8 進数，16 進数	int，unsiged int，long int，unsigned long int
u または U の接尾語付き	unsiged int，unsigned long int
l または L の接尾語付き	long int，unsigned long int
u または U の接尾語，および l または L の接尾語付き	unsigned long int

2.5.3 列挙定数

列挙定数は，列挙型変数の要素，つまり列挙型変数の値を示すために使用します。

列挙型変数は，識別子で示される特定の値のみを持つことができます。

列挙型（enum）は，次の型の中ですべての列挙定数を表現可能な最初のものとなります。列挙定数は，識別子で示します。

- signed char
- unsigned char
- signed int

記述方法は，“enum 列挙型 {列挙型定数の並び}”となります。

＜例＞

```
enum months { January = 1 , February , March , April , May } ;
```

= と整数の指定がある場合は，列挙型変数はその整数値を持ち，その後ろに続く列挙型変数の値は，その値 +1 の値を持ちます。この例では，それぞれ順に 1，2，3，4，5 の値を持ちます。“ = 1”がない場合は，0，1，2，3，4 の値を持ちます。

2.5.4 文字定数

文字定数は，' x '，または ' ab ' のように，シングル・クォートで囲まれる 1 文字以上の文字列です。

文字定数には，シングル・クォート（ ' ），バック・スラッシュ（ \，あるいは ¥），および改行文字（¥n）は含まれません。それらの文字を表す場合は，拡張表記（エスケープ・シーケンス）を使用します。

拡張表記には，次の 3 種類があります。

- 単純拡張表記

  ¥' ¥" ¥? ¥¥ ¥a ¥b ¥f ¥n ¥r ¥t ¥v

- 8 進拡張表記

  ¥8 進数字 [8 進数字 8 進数字]

  （例：¥012，¥0 注 1）

- 16 進拡張表記

  ¥x 16 進数字

  （例：¥xFF 注 2）

注 1 ヌル文字を表します。

注 2 CC78K0R では，-1 の値を表します。ただし，char を unsigned char とみなす条件（オプション）を付けた場合は，+255 の値を表します。

2.6 文字列リテラル

文字列リテラルは，"xxx" のようにダブル・クォートで囲まれる 0 個以上の文字の並びです（例："xyz"）。

シングル・クォート（ ' ）は，それ自身，または拡張表記（ \' ）で表現します。また，ダブル・クォート（ " ）は，拡張表記（ \" ）で表現します。

配列要素は，char 型を持ち，与えられた文字で初期化されます（例：char array [ ] = "abc" ;）。

2.7 演算子

演算子を次に示します。

表 2-9 演算子一覧

```
[ ]  ( )  .  ->
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=
^  |  &&  ||
?  :
=  *=  /=  %=  +=  -=  <<=  >>=
&=  ^=  |=
,  #  ##
```

“[ ]”，“( )”，および“? :”演算子は，必ずペアで使用します。また，この間には式を書くことも可能です。

“#”，および“##”は，前処理指令のマクロ定義にのみ使用することができます。記述方法については，「第 5 章 演算子と式」を参照してください。

2.8　区切り子

区切り子は，独立した文法，または意味を持つ記号ですが，値の生成は行いません。

区切り子を次に示します。

```
[ ]  ( )  { }  *  , :  =  ;  ...  #
```

区切り子“[ ]”，“( )”，“{ }”は，間に式，宣言，または文を書くことができます。ただし，これらは必ずペアで使用します。

区切り子“#”は，前処理指令だけに使用します。

2.9 ヘッダ名

ヘッダ名は，“#include”前処理指令でのみ使用されます。ヘッダ名の #include 指令により，外部ソース・ファイルとの対応付けが行われます。

ヘッダ名の #include 指令の例を次に示します。#include 各指令の詳細については，「9.2 ソース・ファイルの取り込み」を参照してください。

```
#include        < ヘッダ名 >
#include        " ヘッダ名 "
```

2.10 コメント

コメントとは，C ソース・モジュールに入れる注釈文のことです。

コメント文は，先頭を“ /* ”で示し最後を“ */ ”で閉じます。

なお，CC78K0R はマルチバイト文字を識別することができるため，漢字を使用することが可能です。オプション，または環境変数により，漢字コードを指定することができます。

また，-zp オプションにより，“ // ”以降改行までをコメント文として認識することができます。

＜例＞

```
/* コメント文 */
// コメント文
```

第 3 章　型，記憶域クラスの宣言

この章では，C 言語で使用されるデータや関数の型と宣言，またその有効範囲について説明します。

宣言とは，識別子，または識別子の集まりに，解釈，および属性を指定することです。識別子によって名付けられたオブジェクトや関数に対して記憶域も確保する宣言は，“定義”といいます。

宣言の例を次に示します。

＜型，記憶域クラスの宣言の例＞

```
#define TRUE     1
#define FALSE    0
#define SIZE     200

void     main ( void )
{
        auto    int     i , prime , k ;          /* オートマティック変数の宣言 */

        for ( i = 0 ; i <= SIZE ; i++ )
                mark [ i ] = TRUE ;
                  :
}
```

宣言は，記憶域クラス指定子，型指定子，初期化宣言子などで構成されます。記憶域クラス指定子，型指定子は，結合，記憶の持続期間，および宣言子が示す実体の型を指定します。初期化宣言子は，カンマで区切られる宣言子で，各宣言子は付加的な型情報，初期化子，またはその両方を持つことができます。

あるオブジェクトを表す識別子が無結合で宣言されている場合，そのオブジェクトの型はその宣言子の終わりまでに，または初期化子を持っていればその初期化宣言子の終わりまでに完全（サイズに関する情報を持つオブジェクト）になっていなければなりません。

3.1 記憶域クラス指定子

記憶域クラス指定子は，オブジェクトの記憶域クラスを指定するものです。

記憶域クラスは，オブジェクトが持つ値の格納場所や，オブジェクトのスコープ（有効範囲）を示します。1 つの宣言中で，記憶域クラス指定子は 1 つまでしか記述することができません。

記憶域クラス指定子には，次の 5 つがあります。

表 3-1 記憶域クラス指定子

指定子の種類	意味
typedef	typedef 指定子は，指定した型に対する同義語を宣言します。 typedef 指定子の詳細は，「3.6 typedef」を参照してください。
extern	extern 指定子は，外部変数であることを示します。
static	static 指定子は，オブジェクトが静的持続期間を持つことを示します。 静的持続期間を持つオブジェクトは，プログラムの実行前に領域を確保され，格納される値は 1 回だけ初期化されます。オブジェクトは，プログラムの実行中存在して，最後に格納された値を保持します。
auto	auto 指定子は，オブジェクトが動的持続期間を持つことを示します。 動的持続期間を持つオブジェクトは，実行時に領域が確保されます。ブロックに頭から入るときに，初期化の指定があるとオブジェクトの初期化が行われます。ブロック内のラベルにジャンプして入った場合は，初期化されません。 動的持続期間を持つオブジェクトの領域は，宣言されたブロックの実行が終わると保証はされません。
register	register 指定子は，オブジェクトが CPU のレジスタに割り当てられることを示します。 CC78K0R では，レジスタ，saddr 領域に割り当てられます。 レジスタ変数の詳細については，「第 11 章 拡張機能」を参照してください。

3.2　型指定子

型指定子は，オブジェクトの型を指定します。

型指定子には，次のものがあります。

- void
- char
- short
- int
- long
- float
- double
- long double
- signed
- unsigned
- 構造体指定子と共用体指定子
- 列挙型指定子
- typedef 名

また，CC78K0R では，次の型指定子が追加されています。

```
bit/boolean/__boolean
```

各型指定子の意味と，CC78K0R で表現可能な値の限界値について，次に示します。

CC78K0R は，浮動小数点演算について IEEE Std 754-1985 の単精度のみをサポートするため，double，long double は，float と同じフォーマットを持つものとします。

スラッシュで区切られた型指定子は，同じ大きさです。

表 3-2 型指定子

指定子の種類	意味	限界値
void	空の値の集合	－
char	基本文字セットを格納することができる大きさ	－
signed char	符号付き整数	-128 ～ +127
unsigned char	符号なし整数	0 ～ 255
short/signed short/ short int /signed short int	符号付き整数	-32,768 ～ +32,767
unsigned short/unsigned short int	符号なし整数	0 ～ 65,535
int/signed/signed int	符号付き整数	-32,768 ～ +32,767
unsigned/unsigned int	符号なし整数	0 ～ 65,535
long/signed long/ long int/signed long int	符号付き整数	-2,147,483,648 ～ +2,147,483,647
unsigned long/ unsigned long int	符号なし整数	0 ～ 4,294,967,295
float	単精度の浮動小数点数	1.17,549,435E - 38F ～ 3.40,282,347E + 38F 注
double	倍精度の浮動小数点数	1.17,549,435E - 38F ～ 3.40,282,347E + 38F 注
long double	拡張精度の浮動小数点数	1.17,549,435E - 38F ～ 3.40,282,347E + 38F 注
構造体／共用体指定子	メンバ・オブジェクトの集合	－
列挙指定子	int 型定数の集合	－
typedef	名指定した型の同義語	－
bit/boolean/__boolean	1 ビットで表現可能な整数	0 ～ 1

注 絶対値の範囲です。

3.2.1　構造体指定子と共用体指定子

構造体指定子と共用体指定子は，名前付きメンバ（オブジェクト）の集まりを示します。
個々のメンバは，それぞれ異なった型を持つことができます。

(1) 構造体指定子

構造体指定子は，複数の異なった型の集まりを 1 つのオブジェクトとして宣言します。
個々の型のオブジェクトはメンバと呼ばれ，それぞれに名前を付けられます。また，宣言された順に，メンバ用の連続した領域が確保されます。
ただし，78K0R シリーズは，奇数番地からワード・データのリード／ライトができない制約があるため，デフォルトではコード・サイズを優先して，2 バイト以上のメンバが偶数番地に配置されるよう，アライン・データを挿入します。したがって，メンバ間に関しては，アライン・データによる隙間が生じる場合があります。
-rc オプションを指定することによって，アライン・データの挿入を抑制し，構造体をパッキングすることができます。この場合，データ・サイズは小さくなりますが，奇数番地に配置された 2 バイト以上のメンバのリード／ライトが 1 バイト単位のリード／ライトのコードに展開されるため，コード・サイズが増加します。
構造体は，次のように宣言します。この宣言は，後ろに構造体変数の並びがないため，まだメモリ割り当てを行いません。構造体変数の定義については，「第 7 章 構造体と共用体」を参照してください。

```
struct   識別子 { メンバ宣言並び} ;
```

＜構造体宣言の例＞

```
struct   tnode   {
         int     count ;
         struct  tnode   *left , *right ;
} ;
```

(2) 共用体指定子

共用体指定子は，複数の異なった型の集まりを 1 つのオブジェクトとして宣言します。
個々の型のオブジェクトはメンバと呼ばれ，それぞれに名前を付けられます。
共用体のメンバ用に確保された領域は，すべて重なり合った領域です。
共用体は，次のように宣言します。この宣言は，後ろに共用体変数の並びがないため，まだメモリ割り当てを行いません。共用体変数の定義については，「第 7 章 構造体と共用体」を参照してください。

```
union    識別子 { メンバ宣言並び} ;
```

<共用体宣言の例>

```
union   u_tag  {
        int     var1 ;
        long    var2 ;
} ;
```

メンバの型は，不完全型，または関数型以外であればどのような型でもかまいません。また，メンバはビット数の指定付きで宣言することができます。ビット数が指定されたメンバをビット・フィールドと呼びます。

また，CC78K0R では，ビット・フィールド宣言に関する拡張機能を加えています。詳細については，「11.5 ビット・フィールド宣言」を参照してください。

(3) ビット・フィールド

ビット・フィールドは，指定したビット数からなる整数型の領域です。

ビット・フィールドには，int 型，unsigned int 型，および signed int 型を指定することができます[注 1]。

修飾子のない int ビット・フィールドの最上位ビットを符号ビットとみなすか否かは，シリーズにより異なります。また，シリーズにより，signed 型ビット・フィールドを unsigned 型として扱います。[注 2]。

ビット・フィールドが複数ある場合，同じメモリ単位中に十分な空きが残っていれば，続くビット・フィールドは隣接するビットに詰めて入れられます。幅が 0 で無名のビット・フィールドを置いた場合は，同じメモリ単位中に次のビット・フィールドは詰め込まれません。無名のビット・フィールドは宣言子を持たず，コロン，および幅のみで宣言します。

ビット・フィールド・オブジェクトに単項＆演算子（アドレス）を適用することはできません。

注 1　CC78K0R の場合は，char 型，unsigned char 型，signed char 型も指定可能です。
ただし，CC78K0R は，signed 型ビット・フィールドをサポートしていないので，すべて unsigned 型とみなします。

注 2　CC78K0R では，コンパイラ・オプション -rb により，ビット・フィールドの割り付け方向を変更することができます。詳細については，「第 11 章 拡張機能」を参照してください。

<ビット・フィールドの例>

```
struct  data {
        unsigned int    a : 2 ;
        unsigned int    b : 3 ;
        unsigned int    c : 1 ;
} no1 ;
```

3.2.2　　列挙型指定子

列挙型指定子は，順番付けされるオブジェクトを示します。

列挙型指定子によって宣言されるオブジェクトは，int 型を持つ定数として宣言されます。

列挙型指定子は，次のように宣言します。

```
enum    識別子 { 列挙子並び }
```

オブジェクトは，列挙子並びによって宣言します。

オブジェクトは，宣言された順に先頭を 0 で，それに続くオブジェクトを順に 1 ずつ加えた値に定義します。また，“ = ”によって定数値を指定することができます。

次の例では，“hue”を列挙の識別子（タグ）にし，“col”をこの型を持つオブジェクト，“cp”をこの型を持つオブジェクトへのポインタとしています。この宣言で列挙された値は，“{ 0 , 1 , 20 , 21 }”になります。

```
enum    hue  {
        chartreuse ,
        burgundy ,
        claret = 20 ,
        winedark
} ;

enum    hue     col , *cp ;

void    main ( void )  {
        col = claret ;
        cp = &col ;
        if ( *cp != burgundy ) {
                :
        } else {
                :
        }
          :
}
```

3.2.3 タグ

タグは，構造体／共用体，および列挙型に付ける名前です。

タグは宣言された型を持ち，タグにより同じ型のオブジェクトを宣言することができます。

次の宣言の識別子がタグ名です。

```
構造体／共用体      識別子  { メンバ宣言並び }
または
enum                識別子  { 列挙子並び }
```

タグは，メンバ宣言並びによって定義される構造体／共用体，および列挙の内容を持ちます。一度タグを指定し，構造体／共用体，および列挙を指定すると，中かっこ“{ }”で囲まれた並びを省略することができます。次からの宣言は，タグが持つ内容と同じ構造を持ちます。同じ有効範囲中のそれ以後の宣言は，中かっこで囲まれる並びを省略しなければなりません。

次の型指定子は，内容が未定義なので構造体，または共用体は不完全型です。

```
構造体／共用体      識別子
```

この場合のタグは，オブジェクトの大きさが必要ないときにのみ使うことができます。これは，同じ有効範囲の中でタグの内容を定義することにより，型が完全になります。

次の例では，タグ“tnode”は，整数，および 2 つの同じ型のオブジェクトへのポインタを含む構造体を指定しています。

```
struct  tnode   {
        int     count ;
        struct  tnode   *left , *right ;
} ;
```

次の例は“s”をタグで示される型のオブジェクトとして宣言し，“sp”をタグで示される型のオブジェクトへのポインタとして宣言します。この宣言により，式“sp -> left”は“sp”が指すオブジェクトの左の“struct tnode”へのポインタを指すことを示します。

また，式“s.right -> count”は，“s”の右の“struct tnode”のメンバである“count”を指すことを示します。

```
typedef struct  tnode   TNODE ;

struct  tnode   {
        int     count ;
        struct  tnode   *left , *right ;
} ;

TNODE   s , *sp ;

void    main ( void )  {
        sp -> left = sp -> right ;
        s.right -> count = 2 ;
}
```

3.3　型修飾子

型修飾子には，“const”と“volatile”の 2 つがあります。

これらは，左辺値に対してのみ影響します。

const 修飾型で定義されたオブジェクトを，非 const 修飾型を持つ左辺値を使って変更することはできません。また，volatile 修飾型で定義されたオブジェクトを，非 volatile 修飾型を持つ左辺値を使って参照することはできません。

volatile 修飾型を持つオブジェクトは，コンパイラからは認識されない方法で変更することができ，また，他の認識されない副作用を持つことができます。したがって，このオブジェクトを参照する式は，C 言語で書かれたプログラムがどのように実行されるかを順序規則に従って厳密に評価しなければなりません。さらに，そのオブジェクトに最後に格納された値は，未知の要因による変更点を除き，すべての副作用完了点で，プログラムによって定められたものと一致する必要があります。

配列型の指定に型修飾子がある場合，型修飾子は配列ではなく，配列の要素を修飾します。関数型の指定に型修飾子を含めることはできませんが，「2.2 キーワード」で示した CC78K0R 独自の型修飾子，callt，__callt，callf，__callf，noauto，norec，__leaf，__interrupt，__interrupt_brk，__rtos_interrupt，__near，__far を型修飾子として含めることはできます。

なお，sreg，__sreg，__directmap，__temp，__near，__far も型修飾子です。

次の例では，“real_time_clock”はハードウエアによって変更することができますが，代入や，インクリメント，デクリメントなどの操作を行うことはできません。

```
extern  const   volatile        int     real_time_clock ;
```

型修飾子が集成体型を修飾する場合を次に示します。

```
const   struct  s  { int mem ; } cs = { 1 } ;
struct  s       ncs ;                          /* オブジェクト ncs は変更可能である */
typedef int     A [ 2 ] [ 3 ] ;
const   A a = { { 4 , 5 , 6 } , { 7 , 8 , 9 } } ;    /* const int の配列の配列 */
int     *pi ;
const   int     *pci ;

ncs = cs ;               /* 正しい */
cs = ncs ;               /* 代入演算子の変更可能な左辺値の制約に違反している */
pi = &ncs.mem ;          /* 正しい */
pi = &cs.mem ;           /* 代入演算子の型の制約に違反している */
pci = &cs.mem ;          /* 正しい */
pi = a [ 0 ] ;           /* 正しくない : a [ 0 ] は“const int *”型を持つ */
```

3.4　宣言子

宣言子は，1 つの識別子を宣言します。

ここでは，特にポインタ宣言子，配列宣言子，および関数宣言子を説明します。

宣言子により，識別子の有効範囲，記憶域期間，および型を持つ関数，またはオブジェクトが決まります。

次に各宣言子の記述方法を示します。

3.4.1　ポインタ宣言子

ポインタ宣言子は，宣言する識別子がポインタであることを示します。

ポインタは，値が格納されているところをポイントする（指す）ものです。

ポインタ宣言は，次のように行います。

```
* 型修飾子並び    識別子
```

この宣言により識別子は，型修飾子で修飾されたポインタになります。

次の 2 つの宣言は，“定数値への可変ポインタ”，および“変数値への不変ポインタ”を示しています。

```
const   int     *ptr_to_constant ;
int     *const  constant_ptr ;
```

1 行目の宣言は，ptr_to_constant が指す const int の内容は変更されてはならないが，ptr_to_constant 自身は他の const int を指すように変更されてもよいことを表しています。同様に，2 行目の宣言では，constant_ptr が指す int の内容は変更されてもよいが，constant_ptr 自身は常に同じ位置を指すことを表しています。

不変ポインタ constant_ptr の宣言は，“int 型へのポインタ”型に対する定義を含むことによって明確にすることができます。

次の例は，constant_ptr を“int への const 修飾ポインタ”型を持つオブジェクトとして宣言しています。

```
typedef int     *int_ptr ;
const   int_ptr constant_ptr ;
```

3.4.2　配列宣言子

配列宣言子は，宣言する識別子が配列型を持つオブジェクトであることを宣言します。
配列宣言は，次のように行います。

```
型          識別子  [ 定数式 ]
```

この宣言により識別子は，宣言された型の大きさを持つ配列になります。定数式の値が配列の要素数になります。定数式は，0 より大きい値を持つ整数定数式です。配列の宣言において定数式の指定がないと，配列は不完全型になります。

次の例は，11 の要素数からなる char 型の配列“a [ ]”と，17 の要素数からなる char 型のポインタの配列“ap [ ]”を宣言しています。

```
char    a [ 11 ] ,  *ap [ 17 ] ;
```

次の例で，最初の宣言の x は，int 型へのポインタであることを宣言しています。2 番目の宣言では，y が他のところで宣言される大きさの指定のない int 型の配列であることを宣言しています。

```
extern  int     *x ;
extern  int     y [ ] ;
```

3.4.3　関数宣言子（プロトタイプ宣言を含む）

関数宣言子は，関数の返り値，および引数の型を宣言します。
関数宣言は，次のように行います。

```
型            識別子  （ 仮引数型並び，または識別子並び ）
```

この宣言により，仮引数型並びで指定した型の仮引数を持ち，識別子の前に宣言された型の値を返す関数になります。関数の仮引数は，仮引数識別子並びによって指定します。これらの並びにより，引数を示す識別子とその型が決まります。ヘッダ・ファイル“stdarg.h”で定義されているマクロは，省略記法（ , …）で書かれた並びを仮引数に変換します。また，仮引数がない関数は，仮引数型並びを“void”にします。

3.5　型名

型名は，関数，またはオブジェクトの大きさを示す型の名前です。

文法的にみた型名は，関数，またはオブジェクトに対する宣言から識別子を除いたものです。

型名の例を，次に示します。

表 3-3　型名の例

型名の例	説明
int	int 型を指定します。
int *	int 型へのポインタを指定します。
int * [3]	int 型へのポインタを要素とする配列（要素数 3）を指定します。
int (*) [3]	int 型を要素とする配列（要素数 3）へのポインタを指定します。
int * ( )	仮引数指定を持たない int 型へのポインタを返り値とする関数を指定します。
int (*) (void)	仮引数を持たず，int 型を返り値とする関数へのポインタを指定します。
int (*const [ ]) (unsigned int , …)	unsigned int 型の仮引数，および，その他の不定個の仮引数を持ち，int 型を返り値とする個々の関数への不変ポインタを要素とする配列（要素数不定）を指定します。

3.6 typedef

typedef は，識別子が指定した型と同義語であることを定義します。定義された識別子が typedef 名となります。

typedef 名の定義は，次のように行います。

```
typedef 型        識別子 ;
```

次の例では，distance は int 型であり，metricp はパラメータ指定を持たず，int 型を返す関数へのポインタです。また，z の型は指定された構造体であり，zp はこの構造体へのポインタです。

```
typedef int      MILES , KLICKSP ( ) ;
typedef struct  { long re , im ; }      complex ;
  :
MILES   distance ;
extern  KLICKSP *metricp ;
complex z , *zp ;
```

次に示す例は，typedef 名 t を signed int 型で，typedef 名 plain を int 型でそれぞれ宣言し，3 つのビット・フィールド・メンバを持つ構造体を宣言しています。ビット・フィールド・メンバは，次のとおりです。

- 名前が t で，0 ～ 15 の範囲の値をとるもの
- 名前がなく const 修飾された，（アクセスされたならば）-16 ～ +15 の範囲の値をとるもの
- 名前が r で，-16 ～ +15 の範囲の値をとるもの

```
typedef signed  int      t ;
typedef int              plain ;
struct  tag {
        unsigned         t : 4 ;
        const            t : 5 ;
        plain            r : 5 ;
} ;
```

この例で，1 番目のビット・フィールド宣言は unsigned を型指定子とする（このため t を構造体メンバの名前となる）のに対し，2 番目のビット・フィールド宣言は const を型修飾子とする（typedef 名として参照できる t を修飾する）という点で異なっています。この宣言のあとで，内側の有効範囲中で，次の記述があれば，関数 f は“仮引数を 1 つ持ち，signed int を返す関数”として宣言され，その仮引数は，“仮引数を 1 つ持ち，signed int を返す関数へのポインタ型”として宣言されています。また，識別子 t は long 型として宣言されます。

```
t       f ( t ( t ) ) ;
long    t ;
```

typedef 名は，プログラムを読みやすくするために使用することができます。次に示す signal 関数の 3 つの宣言は，すべて 1 番目に定義された typedef 名を使用しないものと同じ型を指定します。

```
typedef void    fv ( int ) ;
typedef void    ( *pfv ) ( int ) ;

void    ( *signal ( int , void ( * ) ( int ) ) ) ( int ) ;
fv      *signal ( int , fv * ) ;
pfv     signal ( int , pfv ) ;
```

3.7　初期化

初期化は，オブジェクトに前もって値を設定することです。

オブジェクトの初期化は，初期化子によって行います。

初期化は，次のように行います。

オブジェクト = ｛ *初期化子並び* ｝ ;

初期化子並びには，初期化するオブジェクトの数だけ初期化子を指定します。

静的記憶域期間を持つオブジェクトと集成体／共用体型を持つオブジェクトに対する初期化子，または初期化子並び中のすべての式は定数式によって指定します。

識別子の宣言がブロック有効範囲を持ち，外部結合，または内部結合を持つ場合，初期化できません。

3.7.1　静的記憶域期間を持つオブジェクトの初期化

静的記憶域期間を持つ算術型のオブジェクトの初期化を行わなかった場合は，暗黙的に 0 に初期化されます。同様に，静的記憶域期間を持つポインタ型のオブジェクトは，NULL ポインタ定数に初期化されます。

＜例＞

```
unsigned        int     gval1 ;         /* 0 で初期化される */
static          int     gval2 ;         /* 0 で初期化される */
void    func ( void )  {
        static   char   aval ;          /* 0 で初期化される */
}
```

3.7.2　自動記憶域期間を持つオブジェクトの初期化

自動記憶域期間を持つオブジェクトの初期化を行わなかった場合は，その値は不定となり保証されません。

＜例＞

```
void    func ( void )  {
        char    aval ;                  /* この時点では不定 */
          :
        aval = 1 ;                      /* 1 に初期化 */
}
```

3.7.3 文字配列の初期化

文字配列は，文字列リテラル（" " で囲まれた文字列）によって初期化することができます。同様に，文字列リテラルの連続した文字は，配列の要素を初期化します。

- 次の例では，“型修飾子なし”の配列オブジェクト s，t が定義され，各配列の要素は文字列リテラルで初期化されます。

```
char    s [ ] = "abc" , t [ 3 ] = "abc" ;
```

- 次の例は，前に示した初期化と同じです。

```
char    s [ ] = { 'a' , 'b' , 'c' , '¥0' } ,
        t [ ] = { 'a' , 'b' , 'c' } ;
```

- 次の例は，p を“char へのポインタ”型として定義し，要素（メンバ）が文字列リテラルで初期化され，長さが 4 の“char 配列”型オブジェクトを指すように初期化しています。

```
char    *p = "abc" ;
```

3.7.4 集成体，共用体オブジェクトの初期化

(1) 集成体

集成体型のオブジェクトの初期化は，添字の昇順，またはメンバの順に書かれた初期化子の並びで行います。指定する初期化子の並びは，中かっこで囲みます。
その並び中の初期化子が集成体のメンバ数より少ない場合，残りのメンバは静的記憶域期間を持つオブジェクトと同じように暗黙的に初期化されます。
大きさのわからない配列は，初期化子の数によって配列の要素数が決まり，不完全型でなくなります。

(2) 共用体

共用体型オブジェクトの初期化は，共用体の最初のメンバに対する中かっこで囲んだ初期化子です。

- 次の例では，大きさがわからない配列 x が初期化によって 3 つのメンバを持つ int 型の 1 次元配列となります。

```
int     x [ ] = { 1 , 3 , 5 } ;
```

- 次の例は，中かっこで囲まれた初期化子を持つ完全な定義です。
“{１,３,５}”は，配列オブジェクト“y［０］”の第１行目“y［０］［０］”，“y［０］［１］”，“y［０］［２］”を初期化します。同様に，次の２行は“y［１］”，“y［２］”を初期化します。“y［３］”の初期値は，指定されていないので０となります。配列の初期化は，次のように指定することができます。

```
char    y [ 4 ] [ 3 ] = {
        { 1 , 3 , 5 } ,
        { 2 , 4 , 6 } ,
        { 3 , 5 , 7 } ,
} ;
```

- 次の例は，前にあげた例と同じ結果になります。

```
char    z [ 4 ] [ 3 ] = {
        1 , 3 , 5 , 2 , 4 , 6 , 3 , 5 , 7
} ;
```

- 次の例は，z の第１列が指定した値に初期化され，残りの要素は０になります。

```
char    z [ 4 ] [ 3 ] = {
        { 1 } , { 2 } , { 3 } , { 4 }
} ;
```

- 次の例は，３次元配列を初期化しています。
q［０］［０］［０］は１に，q［１］［０］［０］は２に，q［１］［０］［１］は３に初期化され，さらに，4，5，および６はそれぞれq［２］［０］［０］，q［２］［０］［１］，およびq［２］［１］［０］を初期化します。そして残りはすべて０になります。

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        { 1 } ,
        { 2 , 3 } ,
        { 4 , 5 , 6 }
} ;
```

- 次の例は，前に挙げた３次元配列の初期化と同じ値に初期化されます。

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        1 , 0 , 0 , 0 , 0 , 0 ,
        2 , 3 , 0 , 0 , 0 , 0 ,
        4 , 5 , 6
} ;
```

- 次の例は，前に挙げた初期化を中かっこを使って完全にしたものです。

```
short    q [ 4 ] [ 3 ] [ 2 ] = {
         {
                  { 1 } ,
         } ,
         {
                  { 2 , 3 } ,
         } ,
         {
                  { 4 , 5 , 6 } ,
         }
} ;
```

第 4 章　型の変換

式中で，2 つの演算数（オペランド）の型が違う場合，自動的に型変換が行われます。これは，キャスト演算子によって得られる変換と同じようなものです。この自動的な型変換は，暗黙の型変換といわれます。この章では，この暗黙の型変換について説明します。

型変換には，正常に変換されるものと，切り捨て，または四捨五入されるもの，また符号が変わるものがあります。型変換の一覧を，次に示します。

表 4-1　型変換一覧

変換前		変換後										
		(signed) char	unsigned char	(signed) short int	unsigned short int	(signed) int	unsigned int	(signed) long int	unsigned long int	float	double	long double
(signed) char	+	―	○	○	○	○	○	○	○	○	○	○
	-	―	N	○	N	○	N	○	N	○	○	○
unsigned char		△	―	○	○	○	○	○	○	○	○	○
(signed) short int	+			―	○	―	○	○	○	○	○	○
	-			―	N	―	N	○	N	○	○	○
unsigned short int				△	―	△	―	○	○	○	○	○
(signed) int	+			―	○	―	○	○	○	○	○	○
	-			―	N	―	N	○	N	○	○	○
unsigned int				△	―	△	―	○	○	○	○	○
(signed) long int	+							―	○	○	○	○
	-							―	N	○	○	○
unsigned long int								△	―	○	○	○
float										―	○	○
double											―	○
long double												―

○	：	正しく変換されます。
―	：	型変換は，行われません。
N	：	正しい値になりません（符号なし整数とみなされます）。
△	：	ビット・イメージ的には変わりませんが，正の数で表現しきれない場合は，正しい値になりません（符号付き整数とみなされます）。
空欄	：	変換時にオーバーフローした部分は，切り捨てられます。 符号も変換後の型によって変わる場合もあります。

備考 signed は省略できます。ただし，char 型の場合にかぎり，コンパイル時の条件（オプション）によって signed char，または unsigned char とみなされます。

4.1 算術オペランド

(1) 文字型と整数型（汎整数拡張）

char，short int，int のビット・フィールドと，これらの符号付き，または符号なしのもの，または列挙型を持つオブジェクトは，いずれの場合も，int 型で表現可能な範囲内にあれば int 型に変換されます。int 型で表現できない場合は，unsigned int 型に変換されます。これらを“汎整数拡張”と呼びます。

その他のすべての算術型は，汎整数拡張によって変わることはありません。汎整数拡張は，符号も含めて値を保持します。

CC78K0R では，修飾子なしの char を符号付きとして扱います。なお，オプションにより，unsigned char として扱うこともできます。

(2) 符号付き整数型と符号なし整数型

整数型を持つ値が他の整数型に変換される場合，値が変換後の整数型で表現できればその値は変わりません。

符号付き整数がそれと等しいか，より大きいサイズを持つ符号なし整数に変換される場合，符号付き整数の値は負でなければ変わりません。符号付き整数の値が負で，符号なし整数が符号付き整数より大きいサイズを持つ場合は，まず，符号付き整数が符号なし整数と同じ大きさの符号付き整数に拡張され，次に，符号なし整数型で表現できる最大数に 1 を加えた値を足して変換前の符号付き整数の値が符号なしの値に変換されます。

整数型を持つ値がより小さいサイズを持つ符号なし整数に変換される場合，その結果は変換後の符号なし整数型で表現できる最大の符号なし数に 1 を加えた値で割った非負の余りとなります。整数型を持つ値がより小さいサイズを持つ符号付き整数に変換される場合，または符号なし整数が同じ大きさの符号付き整数に変換される場合，変換後の値を表現できなければ，オーバーフローした部分は無視されます。変換パターンについては，表 4-1 を参照してください。

符号付き整数から符号なし整数への変換には，次の場合があります。

		unsigned	
		値の範囲小	値の範囲大
signed	+	／	○
	-	／	+

○　：正しく変換されます。

+　：正の整数に変換されます。

／　：変換される型の最大値に 1 を加えた値で割った余り（剰余）となります。

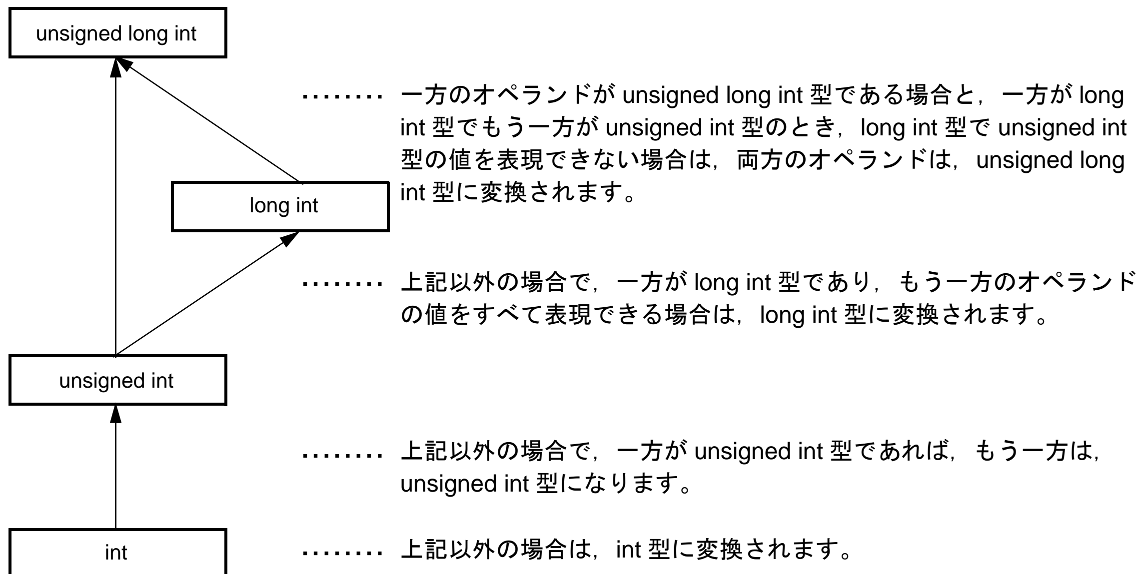
(3) 通常の算術型変換

算術型の演算結果の型は，広い値の範囲を持つ方の型になります。

演算結果の型変換は次のとおり行われます。

- 一方のオペランドが long double 型を持てば，もう一方のオペランドは long double 型に変換される。
- 一方のオペランドが double 型を持てば，もう一方のオペランドは double 型に変換される。
- 一方のオペランドが float 型を持てば，もう一方のオペランドは float 型に変換される。

上記以外は，次の規則に従って汎整数拡張を両方のオペランドに対して行います。



CC78K0R では，コンパイル条件（最適化オプション）により，意図的に int 型に変換させないようにすることができます。詳細については，「CC78K0R C コンパイラ 操作編」のユーザーズ・マニュアルを参照してください。

4.2 他のオペランド

(1) 左辺値と関数指示子

“左辺値”とは，オブジェクトを指定する（オブジェクト型，または void を除く不完全型を持つ）式です。配列型，不完全型，または const 修飾型を持たない左辺値，および const 修飾型のメンバを持たない構造体，または共用体は“変換可能な左辺値”です。

sizeof 演算子，単項＆演算子，++ 演算子，-- 演算子のオペランド，演算子，または代入演算子の左オペランドである場合を除いて，配列型を持たない左辺値は，指定されるオブジェクトに格納されている値に変換されます。変換されることにより左辺値ではなくなります。

不完全型を持ち，配列型を持たない左辺値は保証されません。

文字配列を除き，“ …の配列”型を持つ左辺値は，配列オブジェクトの先頭のメンバを指す“ …へのポインタ”型を持つ式に変換されます。これは左辺値ではありません。

“関数指示子”は，関数型を持つ式です。sizeof 演算子，または単項＆演算子のオペランドを除いて，“ …を返す関数”型を持つ関数指示子は“ …を返す関数へのポインタ”型を持つ式に変換されます。

(2) void

void 式（void 型を持つ式）の（存在しない）値は，どのような方法でも使うことはできません。そして，この式に対して，void を除く暗黙的な，または明示的な変換は適用されません。他の型の式が void 式を必要とする文脈中に現れると，その値，または指示子はないものとされます。

(3) ポインタ

void ポインタは，任意の不完全型，またはオブジェクト型へのポインタに変換することができます。任意の不完全型，またはオブジェクト型へのポインタは，void ポインタに変換することができます。結果の値は，もとのポインタと等しくなければなりません。

値 0 を持つ整数定数式が void * 型にキャストされた式は，“NULL ポインタ定数”と呼ばれます。NULL ポインタ定数があるポインタに代入されるか，またはあるポインタと等しいか，または比較されれば，NULL ポインタ定数はそのポインタに変換されます。

第 5 章　演算子と式

この章では，C 言語で使用される演算子と定数式について説明します。

C 言語には，算術演算，論理演算を行うための豊富な演算子が用意されています。また，ビット演算やアドレス演算を行う演算子も用意されています。

式は，値の計算，オブジェクト，または関数の指示，副作用の生成とこれらの組み合わせを実行する演算子，およびオペランドの列です。

次に演算子の例を示します。

```
#define TRUE    1
#define FALSE   0
#define SIZE    200

void    lprintf ( char * , int ) ;
void    putchar ( char c ) ;
char    mark [ SIZE + 1 ] ;                                ← +     算術演算子

void    main ( void )  {
        int     i , prime , k , count ;
        count  = 0 ;                                       ← =     代入演算子
        for ( i = 0 ; i <= SIZE ; i++ )                    ← ++    後置演算子
            mark [ i ] = TRUE ;                            ← <=    関係演算子

        for ( i = 0 ; i <= SIZE ; i++ ) {
            if ( mark [ i ] ) {
                prime = i + i + 3 ;                        ← +     算術演算子
                lprintf ( "%d" , prime ) ;
                count++ ;                                  ← ++    後置演算子
                if ( ( count%8 ) == 0 )                    ← ==    関係演算子
                    putchar ( '¥n' ) ;
                for ( k = i + prime ; k <= SIZE ; k += prime )  ← += 代入演算子
                    mark [ k ] = FALSE ;
            }
        }

        lprintf ( "Total %d¥n" , count ) ;
loop1 : ;
        goto    loop1 ;
}

lprintf ( char *s , int i ) {
        int     j ;
        char    *ss ;
        j = i ;
        ss = s ;
}

void    putchar ( char c )  {
        char    d ;
        d = c ;
}
```

C 言語で使用される演算子とその優先順位を，次に示します。

表 5-1　演算子の評価順序

<table>
<tr><th>分類</th><th>演算子</th><th>結合[注]</th><th>優先順位</th></tr>
<tr><td>後置演算子</td><td>[ ]，( )，.，->，++，--</td><td>→</td><td rowspan="18">高い

低い</td></tr>
<tr><td>単項演算子</td><td>++，--，&，*，+，-，~，!，sizeof</td><td>←</td></tr>
<tr><td>キャスト演算子</td><td>（型名）</td><td>←</td></tr>
<tr><td rowspan="2">算術演算子</td><td>*，/，%</td><td>→</td></tr>
<tr><td>+，－</td><td>→</td></tr>
<tr><td>ビット単位のシフト演算子</td><td><<，>></td><td>→</td></tr>
<tr><td rowspan="2">関係演算子</td><td><，>，<=，>=</td><td>→</td></tr>
<tr><td>==，!=</td><td>→</td></tr>
<tr><td rowspan="3">ビット単位の論理演算子</td><td>&</td><td>→</td></tr>
<tr><td>^</td><td>→</td></tr>
<tr><td>|</td><td>→</td></tr>
<tr><td rowspan="2">論理演算子</td><td>&&</td><td>→</td></tr>
<tr><td>||</td><td>→</td></tr>
<tr><td>条件演算子</td><td>? :</td><td>←</td></tr>
<tr><td>代入演算子</td><td>=，*=，/=，%=，+=，-=，<<=，>>=，&=，^=，|=</td><td>←</td></tr>
<tr><td>カンマ演算子</td><td>，</td><td>→</td></tr>
</table>

注　同一行の演算子は，同じ優先順位を持ちます。式中に同じ優先順位を持つ演算子が 2 つ以上ある場合，矢印で示される方向に評価されていきます。

5.1 一次式

一次式には，次のものがあります。

- オブジェクト，または関数として宣言されている識別子
- 定数
- 文字列リテラル
- かっこで囲まれる式

一次式となる識別子は，オブジェクトの場合左辺値で，関数の場合は関数指示子です。定数は，「2.5 定数」で説明するように指定する値によって，型が決まります。

文字列リテラルは，「2.6 文字列リテラル」で説明する型を持つ左辺値です。

5.2　後置演算子

後置演算子は，オブジェクトの後ろに置かれる演算子です。

後置演算子には，次のものがあります。

- [ ] 添字演算子
- ( ) 関数呼び出し
- 構造体と共用体のメンバ（ . ）
- 構造体と共用体のメンバ（->）
- 後置インクリメント演算子（++）
- 後置デクリメント演算子（--）

[ ] 添字演算子

【記述形式】

後置式 [*添字式*]

【機能】

- 添字演算子“[ ]”は，配列オブジェクトのメンバを指定します。
  配列“E1 [E2]”は，“(* (E1 + (E2)))”と同じであると定義されています。つまり，E1 の値は配列の先頭メンバへのポインタであり，E2（整数なら）は E1 の E2 番目（0 から数えて）のメンバを示します。多次元配列の場合，配列の次元数の分，添字演算子を連結します。
  次の例で，x は 3 * 5 の int 型配列になります。x は 3 つのメンバを持ち，各メンバは 5 つの int 型メンバを持つ配列です。

```
int      x [ 3 ] [ 5 ] ;
```

  添字演算子を連続して指定することにより，多次元配列を指定することができます。
  E が i * j * … * k の n 次元配列（n ≧ 2）であるとき，E は n 個の添字演算子によって表すことができます。
  このとき，E は j * … * k の（n - 1）次元配列へのポインタになります。

【注意】

- 後置式は，“……のオブジェクトへのポインタ”を持たなければなりません。添字式は，整数型で指定します。結果は，“……”型になります。

( ) 関数呼び出し

【記述形式】

```
後置式 （ 引数式並び ） ;
```

【機能】

- 関数を呼び出します。
  関数呼び出しは，後置演算子“( )”によって行われます。後置式によって，呼び出す関数を指定し，かっこ内に呼び出す関数へ渡す引数を示します。
- 関数に関する記述は，関数プロトタイプ宣言と，関数定義（関数本体），関数呼び出しがあります。関数プロトタイプ宣言は，関数が返す値と引数の型，および記憶クラスを指定します。
- 関数呼び出しで関数プロトタイプ宣言が参照されなければ各引数は，汎整数拡張されます。これは，" デフォルトの実引数拡張 " と呼ばれます。関数プロトタイプ宣言を行うことにより，デフォルトの実引数拡張を避け，引数の型や数，返り値の型のミスを検出することができます。
- “識別子 ( ) ;”のように記憶クラス指定，および型指定がない関数呼び出しは，外部オブジェクトを持ち，引数に関する情報がなく int を返す関数呼び出しであると解釈されます。つまり，次の宣言が暗黙的に行われます。

```
extern  int     識別子 （ ） ;
```

【関数呼び出し例】

```
int     func ( char , int ) ;           /* 関数プロトタイプ宣言 */
char    a ;
int     b , ret ;

void    main ( void ) {
        ret = func ( a , b ) ;          /* 関数呼び出し */
}

int     func ( char c , int i )  {      /* 関数定義 */
          :
        return  i ;
}
```

【注意】

- 呼び出す関数は，配列を除くオブジェクトを返し，後置式はこの関数へのポインタ型です。
- プロトタイプを含む関数呼び出しでは，引数の型を対応する仮引数に代入可能な型にします。また，引数の数も合わせなければなりません。

構造体と共用体のメンバ（ . ）

【記述形式】

後置式 **.** *識別子*

【機能】

- "**.**"は，構造体，または共用体のメンバを指定します。

  後置式は指定する構造体，または共用体オブジェクトの名前であり，識別子はそのメンバの名前です。

構造体と共用体のメンバ（->）

【記述形式】

後置式 -> *識別子*

【機能】

- “ -> ”は，構造体，または共用体のメンバを指定します。
  後置式は，指定する構造体，または共用体オブジェクトへのポインタの名前であり，識別子はそのメンバの名前です。

  ＜. -> 演算子の例＞

```
#include < stdlib.h >

union {
        struct  {
                int     type ;
        } n ;
        struct  {
                int     type ;
                int     intnode ;
        } ni ;
        struct  {
                int     type ;
                struct  {
                        long    longnode ;
                } *nl_p ;
        } nl ;
} u ;

void    func ( void )  {
        u.nl.type = 1 ;
        u.nl.nl_p -> longnode = -31415L ;
          :
        if ( u.n.type == 1 )
                u.nl.nl_p -> longnode = labs ( u.nl.nl_p -> longnode ) ;
}
```

後置インクリメント演算子（++）

【記述形式】

後置式 ++

【機能】

- 後置インクリメント演算子はオブジェクトの値を 1 加算します。

  この演算は，オブジェクトの型を考慮して行われます。

後置デクリメント演算子（--）

【記述形式】

```
後置式 --
```

【機能】

- 後置デクリメント演算子はオブジェクトの値を 1 減算します。
  この演算は，オブジェクトの型を考慮して行われます。

【注意】

- 後置インクリメントと後置デクリメント演算子のオペランドは，修飾された，または，されていない変換可能な左辺値です。

5.3 単項演算子

単項演算子は，1 つのオブジェクト，および項目に対して演算を行います。

単項演算子には，次のものがあります。

- 前置インクリメント演算子（++）
- 前置デクリメント演算子（--）
- 単項 & 演算子（&）
- 単項 * 演算子（ * ）
- 単項算術演算子（+ - ~ !）
- sizeof 演算子

前置インクリメント演算子（++）

【記述形式】

```
++ 単項式
```

【機能】

- 前置インクリメント演算子はオブジェクトの値を 1 加算します。

  前置インクリメント演算子の式“++E”は，次の式と同じ結果になります。

```
E = E + 1
または
E += 1
```

前置デクリメント演算子（--）

【記述形式】

```
-- 単項式
```

【機能】

- 前置デクリメント演算子はオブジェクトの値を 1 減算します。

  前置デクリメント演算子の式“--E”は，次の式と同じ結果になります。

```
E = E - 1
または
E -= 1
```

単項 & 演算子（&）

【記述形式】

```
& オペランド
```

【機能】

- 指定したオブジェクトのアドレスを返します。

単項 * 演算子（*）

【記述形式】

```
* オペランド
```

【機能】

- 指定したポインタが指す値を返します。

【注意】

- 単項＆演算子のオペランドは，register 記憶域クラス指定子で宣言されていないオブジェクトを指す左辺値です。関数指示子，またはビット・フィールドは，単項＆演算子のオペランドに使用することはできません。
  単項 * 演算子のオペランドは，ポインタ型です。

単項算術演算子（+ - ～ !）

【記述形式】

```
+ オペランド
－ オペランド
~ オペランド
！オペランド
```

【機能】

- 単項 + 演算子は，オペランドに対して正の整数拡張を行います。
- 単項 - 演算子は，オペランドに対して負の整数拡張を行います。
- 単項 ~ 演算子は，オペランドのビットごとの補数を返します。
- 単項 ! 演算子は，論理否定演算子と呼ばれます。論理否定演算子は，オペランドの値が“0”のとき“1”を返します。それ以外のときは“0”を返します。

sizeof 演算子

【記述形式】

```
sizeof 単項式
sizeof ( 型名 )
```

【機能】

- 指定したオブジェクトの大きさをバイト単位で返します。
  返り値はオブジェクトの型で決まり，オブジェクトの値自体は評価しません。
- sizeof 演算した char 型，unsigned char 型，または signed char 型（それらの修飾型も含める）のオブジェクトが返す値は 1 です。配列型のオブジェクトでは，配列の総バイト数になります。また，構造体，または共用体型のオブジェクトの場合，結果の値は領域を保持するために入れられた内部的な詰めものを含めたオブジェクトの総バイト数です。
- 結果は整数定数であり，その型は size_t です。これはヘッダ“stddef.h”で定義されています。sizeof 演算子は，主に記憶域の割り当て，および入出力システムとのやり取りに使用します。

【使用例】

- 次の例は，配列の総バイト数をメンバの大きさで割ることにより，配列のメンバ数を求めています。num には，5 が入ります。

```
int     num ;
char    array [ ] = { 0 , 1 , 2 , 3 , 4 } ;

void    func ( void ) {
                num = sizeof array / sizeof array [ 0 ] ;
}
```

【注意】

- sizeof 演算子のオペランドには，関数型，または不完全型を持つ式とビット・フィールド・オブジェクトを指すものを使用することはできません。

5.4 キャスト演算子

キャスト演算子は，データの型を変更します。

主に，ポインタ型の変換を行う場合にキャスト演算子を使用します。

キャスト演算子には，次のものがあります。

- キャスト演算子（型名）

キャスト演算子（型名）

【記述形式】

```
( 型名 ) 式
```

【機能】

- オブジェクトの型を，かっこ内で示した型に変換します。

【使用例】

```
void    func ( void )  {
        int     val ;
        float   f ;

        f = 3.14F ;
        val = ( int ) f ;               /* キャストにより，3 が val に入る */
        val = * ( int * ) 0x10000 ;     /* 定数をキャスト */
}
```

5.5　算術演算子

算術演算子には，次のものがあります。

- * 演算子
- / 演算子
- % 演算子
- + 演算子
- - 演算子

算術演算子は，優先順位により乗除演算子と加減演算子に分かれます。

乗除演算子は，2 つのオペランドの積，商，余りを求め，加減演算子は，2 つのオペランドの和と差を求めます。

表 5-2　除算／剰余算の演算結果の符号

a / b		b	
		+	-
a	+	+	-
	-	-	+

a % b		b	
		+	-
a	+	+	+
	-	-	-

備考　a，b は各オペランドを示します。

除算は符号を取った数値によって行い，小数点以下は切り捨てます。剰余算も同じように符号を取った数値によって行います。

除算／剰余算の演算結果は，符号を取って計算された値に表 5-2 の符号を付けたものです。表 5-2 は，2 つのオペランドの符号のみの計算結果を示しています。

* 演算子

【記述形式】

```
E1 * E2
```

【機能】

- * 演算子は，2 つのオペランドの積を求めます。

/ 演算子

【記述形式】

```
E1 / E2
```

【機能】

- / 演算子は，左オペランドを右オペランドで除算した商を求めます。

% 演算子

【記述形式】

```
E1 % E2
```

【機能】

- % 演算子は，左オペランドを右オペランドで除算した余りを求めます。

+ 演算子

【記述形式】

```
E1 + E2
```

【機能】

- 2 つのオペランドの和を求めます。

- 演算子

【記述形式】

```
E1 - E2
```

【機能】

- 左オペランドから右オペランドを引いた差を求めます。

5.6 ビット単位のシフト演算子

シフト演算子は，演算子のオペランドを記号で示された方向へ移動します。

シフト演算子には，次のものがあります。

- << 演算子
- >> 演算子

表 5-3 シフト演算

<table>
<tr><th colspan="2">a << b</th><th>b 注</th></tr>
<tr><td rowspan="2">a</td><td>+</td><td>0</td></tr>
<tr><td>-</td><td>0</td></tr>
</table>

<table>
<tr><th colspan="2">a >> b</th><th>b 注</th></tr>
<tr><td rowspan="2">a</td><td>+</td><td>0</td></tr>
<tr><td>-</td><td>−1</td></tr>
</table>

注 b に a のビット幅以上の数値が指定され，シフト演算によりオーバーフローが起こった場合の結果を表に示します。b に負の数が指定された場合は符号なし型とし，正の数として処理します。

備考 a，b は各オペランドを示します。

<< 演算子

【記述形式】

```
E1 << E2
```

【機能】

- 左オペランドを右オペランドが示す値（ビット）分左にシフトし，空いたビットに 0 を入れます。“E1 << E2”で，“E1”が符号なし型であれば結果の値は，“E1”に 2 の“E2”乗をかけた値になります。

>> 演算子

【記述形式】

```
E1 >> E2
```

【機能】

- 左オペランドを右オペランドが示す値（ビット）分右にシフトします。
- “E1”が符号なし型の場合，シフトして空いたビットに 0 を入れます。
- “E1”が符号付き型の場合，空いたビットに符号ビットと同じものを入れます。
- “E1 >> E2”で，“E1”が符号なし型，または符号付き型でかつ非負の値を持つ場合，結果の値は，“E1”を 2 の“E2”乗で割った値です。

5.7　関係演算子

関係を示す演算子には，2 つのオペランドの大小関係を示す“関係演算子”と，等しい／等しくないを示す“等値演算子”があります。

関係演算子と等値演算子には，次のものがあります。

- < 演算子
- > 演算子
- <= 演算子
- >= 演算子
- == 演算子
- != 演算子

関係演算子で，2 つのポインタを比較した場合の大小関係は，ポインタで指されるオブジェクトのアドレス空間内での相対位置によって決まります。

CC78K0R では，関係演算子，等値演算子は，指定された関係が真であれば“1”を，偽であれば“0”を生成し，それらの結果は int 型を持ちます。

< 演算子

【記述形式】

```
E1 < E2
```

【機能】

- 左オペランドが右オペランドより小さいときに“1”を返します。それ以外の場合には，“0”を返します。

> 演算子

【記述形式】

```
E1 > E2
```

【機能】

- 左オペランドが右オペランドより大きいときに“1”を返します。それ以外の場合には，“0”を返します。

<= 演算子

【記述形式】

```
E1 <= E2
```

【機能】

- 左オペランドが右オペランドより小さいか，等しいときに“1”を返します。それ以外の場合には，“0”を返します。

>= 演算子

【記述形式】

```
E1 >= E2
```

【機能】

- 左オペランドが右オペランドより大きいか，等しいときに“1”を返します。それ以外の場合には，“0”を返します。

== 演算子

【記述形式】

```
E1 == E2
```

【機能】

- 2 つのオペランドが等しい場合“1”を返し，等しくない場合に“0”を返します。

!= 演算子

【記述形式】

```
E1 != E2
```

【機能】

- 2 つのオペランドが等しくない場合“1”を返し，等しい場合に“0”を返します。

5.8　ビット単位の論理演算子

ビット単位の論理演算子は，オブジェクトの値をビット単位で論理演算します。

ビット単位の論理演算には，次のものがあります。

- ビット単位の AND 演算子（&）
- ビット単位の排他 OR 演算子（ ^ ）
- ビット単位の OR 演算子（ | ）

ビット単位の AND 演算子（&）

【記述形式】

```
E1 & E2
```

【機能】

- “&”はビットごとの論理積を返す，ビット単位の AND 演算子です。
  ビット単位の AND 演算子は，それぞれ対応するビットが“1”のときのみ“1”を返します。
- ビット単位の AND 演算子は，“& 演算子”によって指定します。

		左オペランドの 1 ビットの値	
		1	0
右オペランドの 1 ビットの値	1	1	0
	0	0	0

ビット単位の排他 OR 演算子（ ^ ）

【記述形式】

```
E1 ^ E2
```

【機能】

- “ ^ ”はビットごとの排他的論理和を返す，ビット単位の排他 OR 演算子です。
  ビット単位の排他 OR 演算子は，それぞれ対応するビットが異なるときのみ“1”を返します。

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">左オペランドの 1 ビットの値</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="2">右オペランドの 1 ビットの値</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
</table>

ビット単位の OR 演算子（ | ）

【記述形式】

```
E1 | E2
```

【機能】

- “ | ”はビットごとの論理和を返す，ビット単位の OR 演算子です。
  ビットごとの OR 演算子は，それぞれ対応するビットが“0”のときのみ“0”を返します。

		左オペランドの 1 ビットの値	
		1	0
右オペランドの 1 ビットの値	1	1	1
	0	1	0

5.9　論理演算子

論理演算子は，2 つのオペランドの論理積と論理和を行います。

論理積は，論理 AND 演算子によって，論理和は論理 OR 演算子によって指定します。

論理演算子には，次のものがあります。

- 論理 AND 演算子（&&）
- 論理 OR 演算子（ || ）

両論理演算子の各オペランドは，ともに int 型の値“0”，または“1”を返します。

論理 AND 演算子（&&）

【記述形式】

```
E1 && E2
```

【機能】

- && 演算子は，2 つのオペランドの論理 AND 演算を行います。
  論理 AND 演算は，2 つのオペランドが“0”以外のときのみ“1”を返します。これ以外の場合“0”を返します。結果の型は，int 型です。

		左オペランドの値	
		0	0 以外
右オペランドの値	0	0	0
	0 以外	0	1

【注意】

- && 演算子は，オペランドを左から右に評価します。左オペランドの値が“0”であれば，右オペランドの評価を行いません。

論理 OR 演算子（ || ）

【記述形式】

```
E1 || E2
```

【機能】

- || 演算子は，2 つのオペランドの論理 OR を行います。
  論理 OR 演算は，2 つのオペランドが“0”のときのみ“0”を返します。これ以外のときは，“1”を返します。結果の型は，int 型です。

		左オペランドの値	
		0	0 以外
右オペランドの値	0	0	1
	0 以外	1	1

【注意】

- || 演算子は，オペランドを左から右に評価します。左オペランドの値が“0”以外であれば，右オペランドの評価を行いません。

5.10　条件演算子

条件演算子は第 1 オペランドの値によって次に行う処理を判断します。条件演算子は，“?”と“ : ”によって判断します。

条件演算子には，次のものがあります。

- 条件演算子（? :）

条件演算子（? :）

【記述形式】

第 1 オペランド ? 第 2 オペランド : 第 3 オペランド

【機能】

- 条件演算は第 1 オペランドを評価して，値が“0”以外であれば第 2 オペランドを評価し，“0”であれば第 3 オペランドを評価します。
  条件演算子の結果の値は，第 2，または第 3 オペランドの値になります。

【使用例】

```
#define TRUE    1
#define FALSE   0

char    flag ;
int     ret ;

int     func ( ) {
        ret = flag ? TRUE : FALSE ;
        return  ret ;
}
```

【注意】

- 第 2，および第 3 オペランドの型がともに算術型ならば，それらを共通の型にするために通常の算術型変換を行います。結果の型は，その共通の型とします。
  両オペランドの型がともに構造体型，または共用体型ならば，結果の型はその型とします。
  また，両オペランドが void 型ならば結果の型は void 型とします。

5.11　代入演算子

代入演算子には，右オペランドそのものを左のオブジェクトに格納する単純代入と，両オペランドの演算結果を左のオブジェクトに格納する複合代入があります。

代入演算子は，次のものがあります。

- 単純代入（=）
- 複合代入（*= /= %= += -= <<= >>= &= ^= |=）

単純代入（=）

【記述形式】

```
E1 = E2
```

【機能】

- 単純代入は，右オペランドを左オペランドの型に変換し，左のオブジェクトに格納します。
  次の例では，単純代入の型変換によって関数から返される int 型の値は char 型に変換され，オーバーフローした部分は切り捨てられます。そして，“-1”との比較は，再び int 型に変換されてから行われます。修飾子なしで宣言されている変数“c”が unsigned char とみなされれば，変換の結果は負にならず，“-1”との比較は決して等しくなりません。このような場合，移植性を完全にするために，変数“c”は int 型で宣言します。

```
int      f ( void ) ;

char     c ;
if ( ( c = f ( ) ) == -1 ) {
    :
} else {
    :
}
```

複合代入（*= /= %= += -= <<= >>= &= ^= |=）

【記述形式】

```
E1 *= E2
E1 /= E2
E1 %= E2
E1 += E2
E1 -= E2
E1 <<= E2
E1 >>= E2
E1 &= E2
E1 ^= E2
E1 |= E2
```

【機能】

- 複合代入演算子は，左右のオペランドの演算を行い，結果を左のオブジェクトに格納します。格納される値は，左オペランドの型に変換されます。
- “E1 op = E2”の複合代入は，左オペランド“E1”が一度しか評価されないことを除き，単純代入式“E1 = E1 op (E2)”と同じです。
  次の複合代入演算の結果は，右の単純代入式の結果と同じになります。

```
a *= b ;        a = a * b ;
a /= b ;        a = a / b ;
a %= b ;        a = a % b ;
a += b ;        a = a + b ;
a -= b ;        a = a - b ;
a <<= b ;       a = a << b ;
a >>= b ;       a = a >> b ;
a &= b ;        a = a & b ;
a ^= b ;        a = a ^ b ;
a |= b ;        a = a | b ;
```

5.12 カンマ演算子

カンマ演算子には，次のものがあります。

- カンマ演算子（ , ）

カンマ演算子（ , ）

【記述形式】

```
E1 , E2
```

【機能】

- カンマ演算子は，左のオペランドを void 型として評価します。
それから右のオペランドを評価し，その値を返します。
- 記述形式によって示されるように，カンマを区切り子として使用するところ（関数の引数並び，および初期化子並び中）では，この章で示すカンマ演算子は現れません。
- 次の例では，カンマ演算子によって関数 f ( ) に渡す第 2 引数の値を求めています。カンマ演算子により，第 2 引数の値は 5 になります。

```
int     a , c , t ;

void    main ( void )  {
        f ( a , ( t = 3 , t + 2 ) , c ) ;
}
```

5.13 定数式

定数式には，汎整数定数式と算術定数式，アドレス定数式，および初期化子中の定数式があります。

定数式の評価は実行中でなく，ほとんどコンパイル中に行われます。

定数式では，sizeof 演算子の中で使用する以外の，次のような演算子は使用することはできません。

- 代入演算子
- インクリメント演算子
- デクリメント演算子
- 関数呼び出し演算子
- カンマ演算子

(1) 汎整数定数式

汎整数定数式は汎整数型です。

汎整数定数式のオペランドには，次のものが使用できます。

- 整数定数
- 列挙定数
- 文字定数
- sizeof 式
- 浮動小数点定数

(2) 算術定数式

算術定数式は算術型です。

算術定数式のオペランドには，次のものを使用することができます。

- 整数定数
- 列挙定数
- 文字定数
- sizeof 式
- 浮動小数点定数

(3) アドレス定数

アドレス定数は，静的記憶域期間を持つオブジェクトへのポインタ，または関数指示子へのポインタです。
アドレス定数は，配列型，または関数型の式を用いることにより，暗黙的に指定することができます。明示的に指定するときは，単項＆演算子を用います。
アドレス定数は，次の演算子を用いて指定することができます。しかし，これを利用してもオブジェクトの値を参照することはできません。

- 配列添字演算子“[ ]”
- メンバ・アクセス演算子“ . ”
- メンバ・アクセス演算子“->”
- アドレス単項演算子“&”
- 間接単項演算子“* ”
- ポインタへのキャスト

第 6 章　C 言語の制御構造

この章では，C 言語の制御構造と C で実行される文について説明します。

一般的に，どのような複雑な処理でも基本的な 3 つの制御構造で表すことができます。この 3 つの制御構造は，順次，選択，および繰り返しです。また強制的にプログラムの流れを変える場合，分岐を使用します。

- 順次処理

  順次処理は，プログラムに記述された順に上から下へ実行します。

  順次実行される文は，特に指定する必要はなく，順次実行されます。

- 選択処理

  選択処理は，実行中のプログラムの状態により次に実行する文が選択され，実行します。

  選択の条件は，制御文として指示します。制御文により 2 つ，または多岐にわたる文の 1 つが選択され，実行されます。

- 繰り返し処理

  繰り返し処理は，同じ処理を複数回実行します。

  制御される文は，制御文で示した状態の間，または指定した回数の間繰り返し実行されます。

- 分岐処理

  分岐処理は，強制的に現在のプログラムの流れから抜け出し，指定したラベルに制御を移します。

  分岐により指定したラベル名の次の文から実行されます。

C 言語で実行される文には，次の 6 つがあります。

- ラベル付き文

  switch 文の取る値と goto 文の分岐先により，分岐を引き起こします。

- 複合文（ブロック）

  複数の文の集まりを 1 つの構文単位としてまとめます。

- 式文と空文

  1 つの式とセミコロンからなる文が式文，セミコロンのみからなる文が空文です。

- 選択文

  式の値に応じていくつかの文から 1 つの文を選択します。

- 繰り返し文

  ループ本体と呼ばれる文を制御式が 0 と比較して等しくなるまでの間，繰り返して実行します。

- 分岐文

  別の場所への無条件分岐を引き起こします。

これらの文の記述例を次に示します。

```
#define SIZE    10
#define TRUE    1
#define FALSE   0

extern  void    putchar ( char ) ;
extern  void    lprintf ( char * , int ) ;

charmark [ SIZE + 1 ] ;

void    main ( void ) {
        int     i , prime , k , count ;

        count = 0 ;
        for ( i = 0 ; i <= SIZE ; i++ )         ← for         繰り返し文
            mark [ i ] = TRUE ;
        for ( i = 0 ; i <= SIZE ; i++ )  {      ← for         繰り返し文
            if ( mark [ i ] )  {                ← if          選択文
                prime = i + i + 3 ;
                lprintf ( "%d" , prime ) ;
                if ( ( count%8 ) == 0 )         ← if          選択文
                      putchar ( '¥n' ) ;
                for ( k = i + prime ; k <= SIZE ; k += prime )
                      mark [ k ] = FALSE ;
            }
        }
        lprintf ( "Total %d¥n" , count ) ;

loop1 :                                         ← loop1 :     ラベル付き文
        goto    loop1 ;                         ← goto        分岐文
}
```

6.1 ラベル付き文

ラベル付き文は，switch 文と goto 文の分岐先を指定します。

switch 文は，複数の選択肢がある文から制御式で指定した文を選択し，実行します。ラベル付き文は，switch 文で実行される文のラベルになります。

goto 文は，通常の処理の流れから対応するラベルへ無条件に分岐します。

ラベル付き文には，次のものがあります。

- case ラベル
- default ラベル

case ラベル

【記述形式】

```
case 定数式 : 文
```

【機能】

- case は，switch 文中にのみ使用します。switch 文の制御式のとる値を列挙します。

【使用例 1】

```
int     f ( void ) , i ;

void    main ( void )  {
          :
        switch ( f ( ) ) {
                case 1 :
                        i = i + 4 ;
                        break ;
                case 2 :
                        i = i + 3 ;
                        break ;
                case 3 :
                        i = i + 2 ;
        }
          :
}
```

- この例では，f ( ) の返り値が 1 のとき最初の case 文が選択され，“i = i + 4”の式が実行されます。同様に，値が 2 のときは 2 番目の case が，3 のとき 3 番目の case が選択されます。使用例の break 文は途中で switch 文から抜け出すためのものです。
  このように，case は複数の選択肢がある場合に使用します。

【使用例 2】

```
int     i ;

void    main ( void ) {
          :
        i = 2 ;
        switch ( i ) {
                case 1 :
                        i = i + 4 ;
                case 2 :
                        i = i + 3 ;
                case 3 :
                        i = i + 2 ;
        }
          :
}
```

- この例では，i に 2 が入っているので，2 番目の case 文から実行されますが，case 文の中に break 文を含まないため，続けて 3 番目の case 文も実行されます。
  このように，case 文の定数式と制御式が一致した場合，それ以降のプログラムを順次実行します。途中で switch 文から抜けたい場合は，break 文を使用します。

default ラベル

【記述形式】

```
default : 文
```

【機能】

- default は，switch 文中にのみ使用します。default は，switch 文中に対応する case がない場合の処理を指定します。

【使用例】

```
int      f ( void ) ,  i ;

switch ( f ( ) ) {
        case 1 :
                i = i + 4 ;
                break ;
        case 2 :
                i = i + 3 ;
                break ;
        case 3 :
                i = i + 2 ;
        default :
                i = 1 ;
}
```

- この例では，f ( ) の返り値が 1 ～ 3 のときは対応する case が選択され，それに続く文が実行されます。この例の break 文は，途中で switch 文から抜け出すためのものです。f ( ) の返り値が 1 ～ 3 以外の場合，default に続く文が実行され，i の値は 1 になります。

6.2 複合文（ブロック）

複合文は，複数の文を 1 つの構文単位とします。複数の文は，中かっこ“{ }”で囲まれることにより，複合文となります。

たとえば，複合文は，ある状態のときに行わせる処理が複数ある場合，その文を中かっこ“{ }”で囲み処理させます。

6.3 式文と空文

1 つの文とセミコロンからなる文を式文といいます。

また，セミコロンのみからなる文を空文といいます。空文は，空のループ本体やラベルを置くために使用します。

式文と空文の記述例を次に示します。

次の例のように，式文として副作用を得るためだけに呼ばれる関数は，キャスト式を用いて明示的に返り値の値を捨てることができます。

```
int     p ( int ) ;

void    main ( void )  {
          :
        ( void ) p ( 0 ) ;
}
```

空文は，繰り返し文のループ本体として使用することができます。

```
char    *s ;

void    main ( void )  {
          :
        while ( *s++ != ' 0 ' ) ;
          :
}
```

また複合文を閉じる“}”の前にラベルを置くためにも使用することができます。

```
void    func ( void )  {
          :
        while ( loop1 )  {
                  :
                while ( loop2 )  {
                          :
                        if ( want_out )
                                goto    end_loop1 ;
                          :
                }
end_loop1 : ;
        }
}
```

6.4 選択文

選択文には，if 文，switch 文があります。

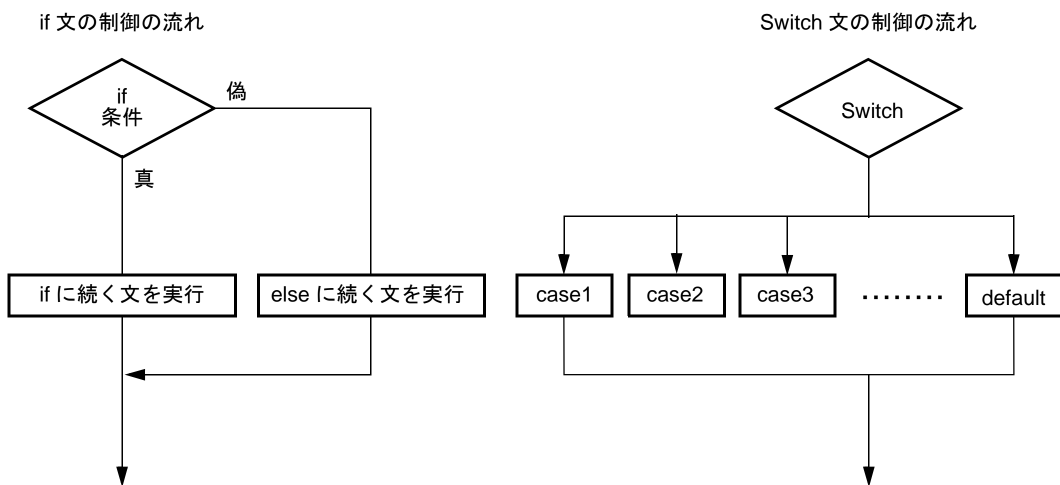
選択文は，“( )”で囲まれた制御式の値によって，文の集まりの中から行う処理を選びます。

選択文には，次のものがあります。

- if 文，if ～ else 文
- switch 文

if 文，switch 文の制御の流れを，次に示します。

図 6-1 選択文の制御の流れ



if 文，if ～ else 文

【記述形式】

```
if ( 式 ) 文
if ( 式 ) 文 else 文
```

【機能】

- if 文，if ～ else 文は，“( )”で囲まれた制御式の値が 0 でなければ，次に続く文を実行します。if ～ else 文の場合，式の値が 0 になると else 文の文を実行します。

【使用例】

```
unsigned char   uc ;

void    func ( void ) {
        if ( uc < 10 ) {
                /* 111 */
        } else {
                /* 222 */
        }
}
```

- この例では，if 文中の制御式により，uc の値が 10 より小さい場合は“ { /* 111 */ } ”のブロックが実行され，10 以上の場合は“{ /* 222 */ }”のブロックが実行されます。

【注意】

- if 文，if ～ else 文のあとに，“{ }”で処理が囲まれていない場合は，if 文／ else 文の次の 1 行の処理のみを本体とみなし，実行します。

switch 文

【記述形式】

```
switch ( 式 ) 文
```

【機能】

- switch 文は，“( )”で囲った制御式に対応する case のスイッチ本体に制御を移します。
  制御式に対応する case がないときは，default に続く文が実行され，また default がないときはどの文も実行されません。

【使用例】

```
extern void     func ( void ) ;

unsigned char   mode ;

void    main ( void ) {
        switch ( mode ) {
                case 2 :
                        mode = 8 ;
                        break ;
                case 4 :
                        mode = 2 ;
                        break ;
                case 8 :
                        func ( ) ;
        }
}
```

【注意】

- 1 つの switch 文の各 case は，同じ値を設定することはできません。また，default は 1 つの switch 文中で一度しか使用することができません。

6.5 繰り返し文

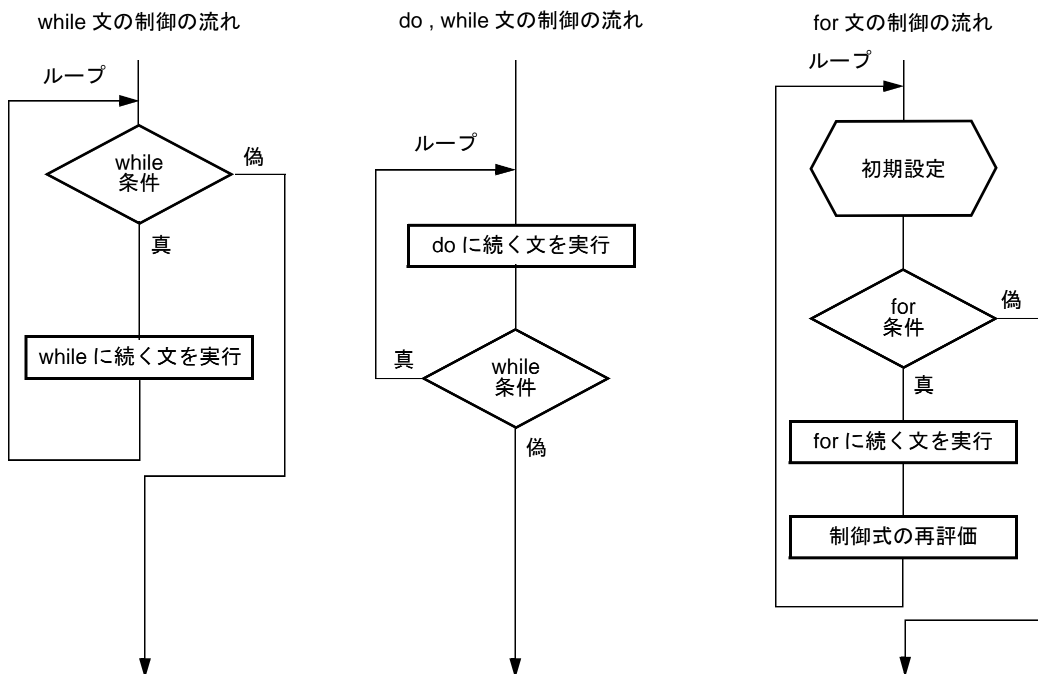
繰り返し文は,“( )”で囲まれた制御式が正しい間（“0”以外のとき）ループ本体を繰り返し実行します。

繰り返し文には，次の 3 つがあります。

- while 文
- do 文
- for 文

繰り返し文の制御の流れを，次に示します。

図 6-2 繰り返し文の制御の流れ



while 文

【記述形式】

```
while ( 式 ) 文
```

【機能】

- while 文は，“( )”で囲まれた制御式が正しい間（0 以外のとき）ループ本体を繰り返し実行します。
  while 文は，制御式をループ本体の実行前に評価します。

【使用例】

```
int     i , x ;

void    main ( void )  {
        i = 1 ,  x = 0 ;

        while ( i < 11 )  {
                x += i ;
                i++ ;
        }
}
```

- 使用例は，x に 1 から 10 までの整数の総和を求めるものです。この while 文のループ本体は，中かっこで囲まれた部分です。
  制御式“i < 11”は，i が 11 になると 0 を返します。このため，i が 1 から 10 になる間，ループ本体が繰り返し実行されます。
- “while (1) { 文 }”は，永久にループ文を実行するために使用します。

do 文

【記述形式】

```
do 文 while ( 式 ) ;
```

【機能】

- do 文は，“( )”で囲まれた制御式が正しい間（0 以外のとき）ループ本体を繰り返し実行します。
  do 文は，制御式をループ本体の実行後に評価します。

【使用例】

```
int     i , x ;

void    main ( void ) {
        i = 1 ,  x = 0 ;

        do  {
                x += i ;
                i++ ;
        } while ( i < 11 ) ;
}
```

- 使用例は，x に 1 から 10 までの整数の総和を求めるものです。この do 文のループ本体は，中かっこで囲まれた部分です。
  制御式“i < 11”は，i が 11 になると 0 を返します。このため，i が 1 から 10 になる間，ループ本体が繰り返し実行されます。do 文の制御式は実行後に評価されるので，ループ本体は必ず 1 回以上実行されます。

for 文

【記述形式】

```
for ( 第1の式 ; 第2の式 ; 第3の式 ) 文
```

【機能】

- for 文は，“( )”で囲まれた中の第 2 の式が正しい間（0 以外のとき）ループ本体を繰り返し実行します。第 1 の式は，カウンタとして使用する変数の初期化を行い，ループの最初に 1 回だけ実行します。第 2 の式でカウンタの判断を行います。第 3 の式は，ループごとに最後に実行する式で，この式の実行後，変数の再評価を行います。

【使用例】

```
int     i , x = 0 ;

for ( i = 1 ; i < 11 ; ++i )
        x += i ;
```

- 使用例は，x に 1 から 10 までの整数の総和を求めるものです。この for ループの本体は，“x += i”です。制御式“ i < 11 ”は，i が 11 になると 0 を返します。このため，i が 1 から 10 になる間，ループ本体が繰り返し実行されます。

【注意】

- for 文のあとに，“{ }”で処理が囲まれていない場合は，for 文の次の 1 行の処理のみを for 文のループ本体とみなします。
- for 文の第 1 の式と第 3 の式は，省略可能です。第 2 の式を省略した場合は，0 でない定数によって置き換えます。“for (; ;) 文”の記述は，永久にループ本体を実行する場合に用います。

6.6 分岐文

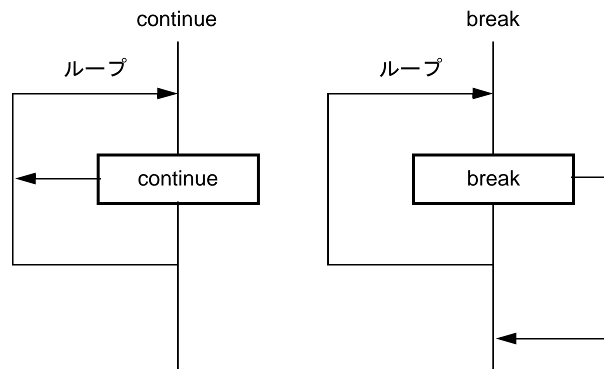
分岐文は，現在の制御の流れから抜け出し，任意の場所へ無条件に制御を移すものです。

分岐文には，次の 4 つがあります。

- goto 文
- continue 文
- break 文
- return 文

分岐文の制御の流れを，次に示します。

図 6-3 分岐文の制御の流れ



goto 文

【記述形式】

```
goto 識別子 ;
```

【機能】

- goto 文は，現在の関数中に指定したラベル名へ無条件にジャンプします。

【使用例】

```
do  {
          :
        goto point ;
          :
} while ( i < 11 ) ;
          :
point : ;
```

- この例では，goto 文に制御が移るとループ処理から無条件に抜け出し，point の次の文に制御が移ります。

【注意】

- goto 文で示される分岐先（ラベル名）は，その goto 文を含む関数中のどこかに必ず示します。

continue 文

【記述形式】

```
continue ;
```

【機能】

- continue 文は，繰り返し文のループ本体中で使用します。continue 文により制御の流れは，ループ本体の最後に無条件に分岐します。continue 文は，これを囲む最も内側の繰り返し文に作用します。

【使用例】

```
while ( i < 11 ) {
          :
        continue ;
          :
contin : ;
}
```

- この例では，ループ本体中の処理が continue 文にくると，制御はラベル“contin”に無条件に分岐します。ラベル“contin”は分岐先を示したもので，特に付ける必要はありません。この例は，goto 文を使用して，continue 文を“goto contin ;”に替えても同じ動作をします。

【注意】

- continue 文は，ループ本体，またはループ本体中にのみ使用することができます。

break 文

【記述形式】

```
break ;
```

【機能】

- break 文は，繰り返し文，または switch 文中から抜け出し，繰り返し文，または switch 文の次の文へ制御を移します。

【使用例】

```
int             i ;
unsigned char   count   , flag ;

void    main ( void ) {
          :
        for ( i = 0 ; i < 20 ; i++ )  {
                switch ( count )  {
                        case 10 :
                                flag = 1 ;
                                break ;         /* switch 文を抜ける */
                        default :
                                func ( ) ;
                }
                if ( flag )
                        break ;                 /* for ループから抜ける */
        }
}
```

- この例では，break 文は switch 文中で必要以上の評価を行わないように使用されています。switch 文の評価で適合する case ラベルがあると，続く break 文により switch 文から抜け出します。

【注意】

- break 文は，スイッチ本体として使用するか，ループ本体としてのみ使用することができます。

return 文

【記述形式】

```
return 式 ;
```

【機能】

- return 文は，return を含む関数から抜け出し，これを呼び出した関数に制御を移します。
  また，return 文の式の値を，関数呼び出し式の値として呼び出し元に返します。
- 1 つの関数中に複数の return 文を使用することができます。
- 関数の最後を“ } ”で閉じることは，式を持たない return 文を実行することと同じです。

【使用例】

```
int     f ( int ) ;

void    main ( void )  {
          :
        int     i = 0 ,  y = 0 ;

        y = f ( i ) ;
          :
}

int     f ( int  i ) {
        int x = 0 ;
          :
        return ( x ) ;
}
```

- この例では，関数“f ( )”は，return 文に制御が移ると main 関数へリターンします。return 文では，返り値として変数“x”の値を返しているので，代入演算子により変数“y”に変数“x”の値が代入されます。

【注意】

- void 型の関数では，返り値を示す式を return 文に使用することはできません。

第 7 章　構造体と共用体

構造体，共用体は，1 つの名前でまとまった，異なった型を持つメンバ・オブジェクトの集まりです。

構造体は，メンバ・オブジェクトが連続的に領域に割り付けられ，共用体は重なり合う領域を割り付けられます。

7.1　構造体

構造体は，連続的に割り付けられるメンバ・オブジェクトの集まりです。

(1) 構造体と構造体変数の宣言

構造体は，“struct”のキーワードによって，構造体宣言リスト，および構造体変数を宣言します。

構造体宣言リストにはタグ名と呼ばれる任意の名前を付けられ，以降このタグ名によって同一構造の構造体変数を宣言することができます。

【構造体の宣言】

```
struct タグ名 { 構造体宣言リスト } 変数名 ;
```

次の例では，最初の struct で data というタグ名を持つ int 型の code，char 型の name，addr，tel 配列を宣言し，no1 をその変数として宣言しています。次の struct では，タグ名により no1 と同じ構造の構造体変数 no2，no3，no4，no5 を宣言しています。

```
struct  data {
        int     code ;
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
struct  data    no2 , no3 , no4 , no5 ;
```

(2) 構造体宣言リスト

構造体宣言リストは，宣言する構造体型の構造を示します。

構造体宣言リスト中の個々の要素をメンバといい，宣言されたメンバの順に領域が確保されます。次の“構造体宣言リストの例”では，変数 a，配列 b，二次元配列 c の順に領域が確保されます。

メンバの型は，不完全型（大きさがわからない配列），関数型であってはなりません。したがって，構造体宣言リスト中に自分自身を含んではいけません。以上の型を除き，メンバはどんなオブジェクト型でも持つことができます。さらに，メンバをビット数で指定するビット・フィールドも指定することができます。

ビット・フィールドは，変数のとる値が 0 か 1 の 2 値である場合，必要最小限のビット数の指定である 1 ビットを指定します。ビット・フィールドにより，必要最小限のビット数の指定で，複数のメンバを 1 個の整数領域に格納することができます。

【構造体宣言リストの例】

```
int     a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

【ビット・フィールド宣言の例】

```
struct  bf_tag {
        unsigned        int     a : 2 ;   ┐
        unsigned        int     b : 3 ;   │  ビット・フィールド
        unsigned        int     c : 1 ;   ┘
} bit_field ;
```

(3) 配列，ポインタ

構造体変数も，他のオブジェクトと同様に配列にしたり，ポインタをとることができます。

構造体の配列では，配列の要素も構造体となります。

【構造体の配列】

構造体の配列宣言は，他のオブジェクトと同じように行います。

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
struct  data no [ 5 ] ;
```

【構造体のポインタ】

構造体のポインタは，ポインタが示す構造体の特徴を持ちます。つまり，構造体のポインタがインクリメントされると，ポインタは構造体の大きさの分加算され，次の構造体を指すようになります。

次の例では，“dt_p”は“struct data”型の値に対するポインタであることを示しています。

ここで，“dt_p”をインクリメントすると，“&no [1]”と同じ値になります。

```
struct  data no [ 5 ] ;
struct  data *dt_p = no ;
```

(4) 構造体メンバの参照方法

構造体メンバを参照するには，構造体変数と変数へのポインタを使う 2 通りの方法があります。

構造体変数による参照には“ . ”演算子を，ポインタによる参照には“->”演算子を使います。

【構造体変数による参照】

構造体変数によるメンバの参照には，“ . ”演算子を使います。

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } , *data_ptr = no ;

void    main ( ) {
        char    c ;
        c = no [ 0 ].name [ 1 ] ;
}
```

【ポインタによる参照】

ポインタ変数によるメンバの参照には，“->”演算子を使います。

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } , *data_ptr = no ;

void    main ( ) {
        char    c ;

        data_ptr -> tel [ 3 ] = 'E' ;
}
```

7.2 共用体

共用体は，同じ領域に割り付けられるメンバの集まりです。

(1) 共用体と共用体変数の宣言

共用体は，“union”のキーワードによって，共用体宣言リスト，および共用体変数を宣言します。共用体宣言リストにはタグ名と呼ばれる任意の名前を付けられ，以降このタグ名によって同一構造の共用体変数を宣言できます。

【共用体の宣言】

```
union タグ名 { 共用体宣言リスト } 変数名 ;
```

次の例では，最初の union で data というタグ名を持つ char 型の name，addr，tel 配列を宣言し，no1 をその変数として宣言しています。次の union ではタグ名により no1 と同じ構造の共用体変数 no2，no3，no4，no5 を宣言しています。

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
union   data    no2 , no3 , no4 , no5 ;
```

(2) 共用体宣言リスト

共用体宣言リストは，宣言する共用体型の構造を示します。共用体宣言リスト中の個々の要素をメンバといい，宣言されたメンバは同じ領域に確保されます。次の“共用体宣言リストの例”では，メンバの中で一番大きな領域となる“c”について領域が確保され，他のメンバは，新たに領域をとることはせず，同じ領域を使用します。

メンバの型は，構造体宣言リストと同様，不完全型（大きさがわからない配列），関数型であってはなりません。以上の型を除き，メンバはどんなオブジェクト型でも持てます。

【共用体宣言リストの例】

```
int     a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

(3) 配列，ポインタ

共用体変数も他のオブジェクトと同様，配列やポインタをとれます。

【共用体の配列】

共用体の配列宣言は他のオブジェクトと同じように行います。

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
union   data    no [ 5 ] ;
```

【共用体のポインタ】

共用体のポインタは，ポインタが示す共用体の特徴を持ちます。つまり，共用体のポインタがインクリメントされるとポインタは共用体の大きさ分加算され，次の共用体を指すようになります。
次の例で“dt_p”は，union data 型の値に対するポインタです。

```
union   data    no [ 5 ] ;
union   data    *dt_p = no ;
```

(4) 共用体メンバの参照方法

共用体メンバを参照するには，共用体変数と変数へのポインタを使う 2 通りの方法があります。共用体変数による参照には“ . ”演算子を，ポインタによる参照には“->”演算子を使います。

【共用体変数による参照】

共用体変数によるメンバの参照には，“ . ”演算子を使います。

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } ;

void    main ( void ) {
        no [ 0 ].addr [ 10 ] = 'B' ;
          :
}
```

【ポインタによる参照】

ポインタによるメンバの参照には，“->”演算子を使います。

```
union   data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} *data_ptr ;

void    main ( void ) {
        data_ptr -> name [ 1 ] = 'N' ;
          :
}
```

第 8 章　外部定義

プログラムの中で，前処理のあとには外部宣言の並びがあります。これらは，関数の外で現れ，ファイル有効範囲を持つので，“外部”と呼ばれます。

外部オブジェクトに対し，識別子で名前付けを行う宣言，または関数のために記憶域の確保を行う宣言を，外部定義と呼びます。外部結合を持って宣言される識別子が式中（sizeof 演算子の演算数の部分を除く）で使われる場合，プログラム全体のどこかに，その識別子に対する外部定義が 1 つ必要です。

```
#define TRUE    1
#define FALSE   0
#define SIZE    200

void    printf ( char * , int ) ;
void    putchar ( char c ) ;

char    mark [ SIZE + 1 ] ;             /* 外部オブジェクト定義 */

main ( ) {
        int     i , prime , k , count ;

        count = 0 ;

        for ( i = 0 ; i <= SIZE ; i++ )
                mark  [ i ] = TRUE ;
        for ( i = 0 ; i <= SIZE ; i++ ) {
                if ( mark [ i ] ) {
                        prime = i + i + 3 ;
                        printf ( "%d" , prime ) ;
                        count++ ;
                        if ( ( count%8 ) == 0 ) putchar ( '¥n' ) ;
                        for ( k = i + prime ; k <= SIZE ; k += prime )
                                mark [ k ] = FALSE ;
                }
        }
        printf ( "Total %d¥n" , count ) ;

loop1 :
        goto    loop1 ;
}
```

8.1 関数定義

関数の定義は，外部定義です。

関数定義は，記憶域クラス指定子を省略した場合でも“extern”で定義されたとみなされます。外部関数定義は，定義された関数が他のファイルから参照できることを示しています。たとえば，複数のファイルからなるプログラムにおいて，他のファイルにある関数を使用する場合，この関数は外部定義にします。

関数の記憶域クラス指定子は，extern，または static です。extern と定義した場合，他の関数から参照することができますが，static で定義すると他のファイルから参照することはできません。

次の例では，extern は記憶域クラス指定子，int は型指定子です。これらは，デフォルトの値なので省略可能です。“max (int a , int b) ”は関数宣言子です。そして，“{ return a > b ? a : b ; }”が関数本体になります。

【関数定義の例】

```
extern int      max ( int a , int b ) {
        return a > b ? a : b ;
}
```

この関数定義は，関数宣言中で仮引数の型を指定しているので，強制的に引数の型変換が行われます。仮引数に対して識別子並びの形を用いても記述することができます。次に，この例を示します。

```
extern int      max ( a , b )

int     a , b ;
{
        return a > b ? a : b ;
}
```

関数呼び出しの引数として，関数のアドレスを渡すことができます。関数名を式中に使うことによって，その関数のポインタが生成されます。

```
int     f ( void ) ;

void    main ( )  {
  :
        g ( f ) ;
}
```

この例では，関数 f を指すポインタにより関数 g に関数 f を渡しています。関数 g では，たとえば次のように定義します。

```
void    g ( int ( *funcp ) ( void ) ) {
        ( *funcp ) ( ) ;                /* または funcp ( ) ; */
}
```

あるいは

```
void    g ( int func ( void ) ) {
        func ( ) ;                      /* または ( *func ) ( ) ; */
}
```

8.2 外部オブジェクト定義

外部オブジェクト定義は，オブジェクトに対する識別子の宣言がファイル有効範囲，および初期化子を持つものです。また，ファイル有効範囲を持つオブジェクトに対する識別子の宣言で，記憶域クラス指定子がなく初期化子を持たないか，記憶域クラス指定子が static である場合は，仮の定義です。この場合，初期化子 0 のファイル有効範囲を持つ宣言とみなされます。

外部オブジェクト定義の例を，次に示します。

表 8-1 外部オブジェクト定義の例

```
int             i1 = 1 ;        /* 外部結合を持つ定義 */
static  int     i2 = 2 ;        /* 内部結合を持つ定義 */
extern  int     i3 = 3 ;        /* 外部結合を持つ定義 */
int             i4 ;            /* 外部結合を持つ仮の定義 */
static  int     i5 ;            /* 内部結合を持つ仮の定義 */

int             i1 ;            /* 前のものを参照する正しい仮の定義 */
int             i2 ;            /* 結合規則違反 */
int             i3 ;            /* 前のものを参照する正しい仮の定義 */
int             i4 ;            /* 前のものを参照する正しい仮の定義 */
int             i5 ;            /* 結合規則違反 */

extern  int     i1 ;            /* 外部結合された前のオブジェクトの参照 */
extern  int     i2 ;            /* 内部結合された前のオブジェクトの参照 */
extern  int     i3 ;            /* 外部結合された前のオブジェクトの参照 */
extern  int     i4 ;            /* 外部結合された前のオブジェクトの参照 */
extern  int     i5 ;            /* 内部結合された前のオブジェクトの参照 */
```

第 9 章　前処理指令（コンパイラに対する指令）

前処理指令は，“#”前処理字句から改行文字までの前処理字句の列です。

前処理字句列の間で使用可能な空白文字は，スペース，および水平タブだけです。

前処理指令は，ソース・ファイルのコンパイル前に行う処理を指定します。前処理には，ソース・ファイルの一部を条件によって処理，またはスキップさせる指令や，他のソース・ファイルを取り込む指令，マクロに置き換える指令などがあります。

次に，それぞれの前処理指令について説明します。

9.1　条件付きコンパイル

条件付きコンパイルは，定数式の値によりソース・ファイルの一部分のコンパイルをスキップします。

条件付きコンパイル指令で指定された定数式の値が偽（0）のとき，続く文はコンパイルされません。定数式には，sizeof 演算子，キャスト，列挙定数を使用することはできません。

条件付きコンパイルの指令には，次のものがあります。

- #if 指令
- #elif 指令
- #ifdef 指令
- #ifndef 指令
- #else 指令
- #endif 指令

条件付きコンパイルでは，次の単項式を指定することができます。

```
defined 識別子
または
defined ( 識別子 )
```

この単項式は，識別子が前処理指令 #define で定義されていれば 1 を返します。定義されていないか，定義を取り消してある場合は 0 を返します。

【使用例】

この例では，SYM が定義されているので，1 を返し，#if ～ #endif の間をコンパイルします（#if ～ #endif の説明については，次頁以降の説明を参照してください）。

```
#define         SYM  0

#if  defined    SYM
  :
#endif
```

#if 指令

【記述形式】

```
#if 定数式 改行 グループ
```

【機能】

- 定数式の値が偽であれば，ソース・ファイルの一部分のコンパイルをスキップします。

【使用例】

```
#if   FLAG == 0
   :
#endif
```

- この例では，“FLAG == 0”によって，後ろに続く文をコンパイルするかどうかを判断しています。
  “FLAG”の値が 0 以外であれば，#if 指令と #endif 指令間のプログラムはコンパイルされず，0 の場合コンパイルされます。

#elif 指令

【記述形式】

```
#elif 定数式 改行 グループ
```

【機能】

- この指令は，通常 #if 指令の後ろにきます。#if 指令の定数式が偽のとき，後ろに続く #elif の定数式が評価され，偽であれば #elif の後ろのプログラムはコンパイルをスキップされます。

【使用例】

```
#if   FLAG == 0
  :
#elif  FLAG != 0
  :
#endif
```

- この例では，“FLAG”の値によって，後ろに続く文をコンパイルするかどうかを判断しています。
  “FLAG”の値が 0 の場合，#if 指令と #elif 指令間のプログラムがコンパイルされます。そして，0 以外の場合，#elif 指令と #endif 指令間のプログラムがコンパイルされます。

#ifdef 指令

【記述形式】

```
#ifdef 識別子 改行 グループ
```

【機能】

- #ifdef 指令は，#if 指令の定数式が defined 識別子になったものです。
- 識別子が #define 指令で定義されていれば，後ろに続くプログラムをコンパイルし，定義されていないか，定義を取り消してある場合にはコンパイルをスキップします。

【使用例】

```
#define ON
#ifdef  ON
  :
#endif
```

- この例では，#define 指令によって“ON”が定義されているので，#ifdef と #endif の間のプログラムはコンパイルされます。“ON”が定義されていなければ，#ifdef と #endif の間のプログラムはコンパイルされません。

#ifndef 指令

【記述形式】

```
#ifndef 識別子 改行 グループ
```

【機能】

- #ifndef 指令は，#if 指令の定数式が !defined 識別子となったものと同じです。この指令は，識別子が前に定義されていれば，後ろに続くプログラムをコンパイルしません。

【使用例】

```
#define ON
#ifndef ON
  :
#endif
```

- この例では，#define 指令によって“ON”が定義されているので，#ifndef と #endif の間のプログラムはコンパイルされません。“ON”が定義されていなければ，#ifndef と #endif の間のプログラムはコンパイルされます。

#else 指令

【記述形式】

```
#else 改行 グループ
```

【機能】

- #else 指令は，前にある条件付きコンパイル指令の識別子が偽の場合にのみ，後ろに続くプログラムをコンパイルします。#else 指令の前にくる指令は，#if，#elif，#ifdef，#ifndef 指令があります。

【使用例】

```
#define ON
#ifdef  ON
  :
#else
  :
#endif
```

- この例では，#define 指令によって“ON”が定義されているので，#ifdef と #else の間のプログラムがコンパイルされます。“ON”が定義されていなければ，#else と #endif の間のプログラムがコンパイルされます。

#endif 指令

【記述形式】

```
#endif 改行
```

【機能】

- #endif 指令は，前にある条件付きコンパイル指令の有効範囲が終わったことを示します。

【使用例】

```
#define ON
#ifdef  ON
  :
#endif
```

- この例では，“#endif”は，条件付きコンパイル ifdef 指令の有効範囲の終わりを示しています。

9.2 ソース・ファイルの取り込み

前処理指令 #include は，指定したヘッダの検索を行い，#include 指令とヘッダの内容全部を置き換えます。

#include によるソース・ファイルの取り込みには，次の 3 つの方法があります。

- #include < > 指令
- #include " " 指令
- #include 前処理字句列指令

#include により取り込まれるソースの中で，#include 指令が現れてもよいですが，CC78K0R では，#include 指令のネストの制限があります。制限については，表 1-1 を参照してください。

備考　前処理字句列：#define 指令で定義された文字列

#include < > 指令

【記述形式】

```
#include          < ファイル名 > 改行
```

【機能】

- 指定されたファイルを， -i コンパイラ・オプションで指定したフォルダ，INC78K0R 環境変数で指定されているフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）から順に検索し，#include 指令をファイルの内容すべてに置き換えます。

【使用例】

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

- INC78K0R 環境変数により指定されたフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）の中から“stdio.h”を検索し，前処理指令“#include < stdio.h >”を“stdio.h”の内容に置き換えます。

#include " " 指令

【記述形式】

```
#include          " ファイル名 "  改行
```

【機能】

- この前処理指令によって取り込まれるソース・ファイルは，はじめにカレント・フォルダの中から検索します。そして，目的のファイルがないと，次に -i コンパイラ・オプションで指定されたフォルダ，INC78K0R 環境変数で指定されているフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）から順に検索します。このようにして検索されたファイルは，#include 指令と置き換えられます。

【使用例】

```
#include          "myprog.h"
```

- カレント・フォルダ，INC78K0R 環境変数により指定されたフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）の中から“myprog.h”を検索し，前処理指令“#include " myprog.h "”を“myprog.h”の内容に置き換えます。

#include 前処理字句列指令

【記述形式】

```
#include        前処理字句列 改行
```

【機能】

- 前処理字句列の置き換えにより，ヘッダ・ファイルが示されます。そして，ヘッダ・ファイルが検索され，#include 指令と置き換わります。

【使用例】

```
#define         INCFILE "myprog.h"
#include        INCFILE
```

- “#include 前処理字句列 改行”によるソース・ファイルの取り込みでは，指定された前処理字句列がマクロ置換により < ファイル名 >，または “ファイル名 ”に置き換わらなければなりません。< ファイル名 > に置き換わった場合，ソース・ファイルは -i コンパイラ・オプションにより指定されたフォルダ，INC78K0R 環境変数で指定されているフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）から順に検索します。“ファイル名 ”の場合はカレント・フォルダから検索し，なければ -i コンパイラ・オプションにより指定されたフォルダ，INC78K0R 環境変数で指定されているフォルダ，..¥inc78k0r（CC78K0R の起動されたパスに対して）から順に検索します。

9.3 マクロ置換

マクロ置換は，識別子で指定した文字列（マクロ名）を“置換要素並び”に置き換えます。

マクロ置換には，オブジェクト形式と関数形式の 2 つがあります。

- オブジェクト形式

  #define 指令

- 関数形式

  #define ( ) 指令

(1) 実引数置換

実引数の置き換えは，関数形式マクロの呼び出しの引数が識別されたあとに行われます。

置換要素並びの仮引数に #，または ## 前処理字句を前に付けずに，## 前処理字句が後ろに続かなければ，並び中に含まれるマクロがすべて展開されたあとに，対応する引数に置き換えられます。

(2) # 演算子

前処理字句は，対応する引数を char 文字列処理字句に置き換えます。

置換要素並び中の仮引数の前にこれを付けると，対応する引数は文字，または文字列になります。

(3) ## 演算子

前処理字句は，前後にある字句を結合します。

結合は，次のマクロ展開が行われる前に実行され，## 前処理字句は削除されます。この結果，生成される字句にマクロ名があれば，さらにマクロ展開されます。

【## 演算子の例】

この例では，次のようにマクロ展開されます。

printf (" x " " 1 " " = %d , x " " 2 " " = %s " , x1 , x2) ;

さらに，char 文字列が結合され，次のようになります。

printf ("x1 = %d , x2 = %s" , x1 , x2) ;

```
#include <stdio.h>

#define debug ( s , t ) printf ( "x" #s "= %d , x" #t "= %s" , x##s , x##t ) ;

void    main ( )  {
        int     x1 ,  x2 ;
        debug ( 1 , 2 ) ;
}
```

(4) 再走査とそれ以上の置き換え

マクロ置換によって置き換えられた結果の前処理字句，およびソース・ファイルの残りの前処理字句の中にマクロ名がある場合，マクロ置換を行います。

現在置き換え中のマクロ名（ソース・ファイルの残りの前処理字句は含まない）が置換要素並びの走査中に見つけられても，置き換えられません。

(5) マクロ定義の有効範囲

マクロ定義は，対応する #undef 指令が現れるまで置き換え続けます。

#define 指令

【記述形式】

```
#define 識別子 置換要素並び 改行
```

【機能】

- #define 指令は，指定した識別子を置換要素並びに置き換えます。
  この指令以降の同じ識別子は，置換要素並びに置き換えられます。

【使用例】

```
#define PAI      3.1415
```

- この例では，ソース・リスト中“PAI”が現れると，すべて“3.1415”に置き換えられます。

#define ( ) 指令

【記述形式】

```
#define 識別子 （ 識別子リスト ） 置換要素並び 改行
```

【機能】

- 関数形式のマクロ指令は，関数形式で指定した識別子を置換要素並びに置き換えます。
  この指令以降の同じ識別子は，置換要素並びに置き換えられます。また，関数形式のマクロ置換では引数を含む置き換えを行うことができます。

【使用例】

```
#define F ( n ) ( n * n )

void    main ( ) {
        int     i ;

        i = F ( 2 ) ;
}
```

- この例の F (2) は，#define 指令により，“(2*2)”に置き換えられます。したがって，i の値は 4 となります。
- 関数形式のマクロは，関数定義と違い単なる文字の置き換えです。したがって，安全のために，#define 指令の置換要素並びは ( ) で囲んでおきます。

#undef 指令

【記述形式】

```
#undef 識別子 改行
```

【機能】

- 対応するマクロ置換指令を終わらせます。

【使用例】

```
#define F ( n ) ( n * n )
  :
#undef  F
```

この例では，“#undef”は前に指定されていた“#define　F (n) (n*n)”を無効にします。

9.4 行制御

行制御は，コンパイラがコンパイル時に使用する行番号を“#line”によって指定された番号に置き換えます。また，文字列を指定した場合，コンパイラが持つソース・ファイル名を指定した文字列に置き換えます。

[行番号を変更する場合]

行番号を変更する場合，次のように指定します。

数字列には，0，および 32767 より大きい数を指定することはできません。

```
#line    数字列 改行
```

＜例＞

```
#line   10
```

[行番号とファイル名を変更する場合]

行番号とファイル名を変更する場合，次のように指定します。

```
#line    数字列    " 文字列 "  改行
```

＜例＞

```
#line   10       "file1.c"
```

[前処理字句列を使用して変更する場合]

上記の指定のほかに，次のように指定することができます。この場合には，指定した前処理字句列は，すべての置き換えのあとに，前記の 2 つの例のいずれかになるようにします。

```
#line    前処理字句列 改行
```

＜例＞

```
#define LINE_NUM        100
#line   LINE_NUM
```

9.5　#error 前処理指令

#error 前処理指令は，指定した前処理字句を含むメッセージを出力し，コンパイルを不成功に終わらせる指定です。

この前処理により，コンパイルを終了させたい場合に使用します。

次のように指定します。

```
#error  "前処理字句列"  改行
```

【使用例】

```
#if __K0R__
  :
#else
#error  "not for 78K0R"
  :
#endif
```

- この例では，CC78K0R が持つ，デバイスのシリーズを示すマクロ名“__K0R__”を使用しています。デバイスが 78K0R シリーズであれば，#if ～ #else 間のプログラムをコンパイルします。そうでない場合は，#else ～ #endif 間のプログラムをコンパイルしますが，#error 指令により，“not for 78K0R”というメッセージをエラーとして出力し，コンパイルを終了します。

9.6 #pragma（プラグマ）指令

#pragma 指令は，コンパイラに対し，コンパイラ定義の方法で動作することを指示する指令です。

CC78K0R では，78K0R シリーズ用のコードを生成するために #pragma 指令が何種類か用意されています。#pragma 指令の詳細については，「第 11 章 拡張機能」を参照してください。

【使用例】

```
#pragma NOP
```

- この例では，#pragma NOP 指令により，C ソースで NOP 命令を直接出力するように記述することができます。

9.7 空指令（Null 指令）

空指令は，コンパイラに対して何の影響も与えません。

改行

9.8 コンパイラ定義のマクロ名

コンパイラには，次のマクロ名があらかじめ定義されています。

表 9-1 マクロ名一覧

マクロ名	説明
__LINE__	カレント・ソース行の行番号（10 進定数）
__FILE__	ソース・ファイル名（文字列リテラル）
__DATE__	ソース・ファイルのコンパイル日付（“M*mm dd yyyy*”の形をした文字列リテラル）
__TIME__	ソース・ファイルのコンパイル時刻（“*hh* : *mm* : *ss*”の形をした文字列リテラル）
__STDC__	ANSI [注]の規格に合致していることを意味する 10 進定数“1”

注 ANSI とは，American National Standards Institute の略称です。

これらのマクロ名，および defined 識別子は，#define，または #undef 前処理指令の適用を受けてはなりません。また，すべてのコンパイラ定義のマクロ名は，アンダースコアではじめます。その後ろには，英大文字，または 2 番目のアンダースコアが続きます。

CC78K0R では，上記のほかに，応用製品の開発対象となるデバイスにより，デバイスのシリーズ名を示すマクロ名と，デバイス名を示すマクロ名を持ちます。これらは，ターゲット・デバイス用のオブジェクト・コードを出力するために，コンパイル時のオプション，または C ソース中のデバイス種別によって指定します。

- デバイスのシリーズ名を示すマクロ名

```
__K0R__
```

- デバイス名を示すマクロ名

デバイス種別名の前に“__”，後ろに“_”を付与したもの

＜例＞

```
__F1166A0_ __F1166A0Y_
```

注意 英字は大文字で記述してください。

備考 デバイス種別名は，-c オプションで指定するものと同じです。デバイス種別名については，デバイス・ファイルに関する資料を参照してください。

また，CC78K0R は，メモリ・モデルを示すマクロ名を持ちます。

メモリ・モデル指定時に，次のように定義します。

＜スモール・モデルの場合＞

```
#define __K0R_SMALL__    1
```

＜ミディアム・モデルの場合＞

```
#define __K0R_MEDIUM__   1
```

＜コンパクト・モデルの場合＞

```
#define __K0R_COMPACT__ 1
```

＜ラージ・モデルの場合＞

```
#define __K0R_LARGE__    1
```

コンパイル時のデバイス種別の指定は，次のものをコマンド・ラインに追加することにより行います。

“-c デバイス種別”

＜例＞

```
cc78k0r -cF1166A0Y prime.c
```

次のように，C ソース・プログラムの先頭にデバイス種別を指定することにより，コンパイル時に指定する必要がなくなります。

“#pragma PC (種別)”

＜例＞

```
#pragma PC      ( F1166A0Y )
  :
```

ただし，次のものは“#pragma PC (種別)”の前に記述することができます。

- コメント文
- 変数の定義または参照，および関数の定義または参照を生成しない前処理指令

第 10 章 ライブラリ関数

C 言語には，外部（周辺）装置，機器との入出力を行う命令がありません。これは，C 言語の設計者が，C 言語の機能を最小限度に抑えるように設計したためです。しかし，実際にシステムを開発するには，入出力操作が必要となります。このため，CC78K0R には，入出力操作を行うためのライブラリ関数が用意されています。

CC78K0R には，入出力，文字／メモリ操作，プログラム制御，数学関数などのライブラリ関数があります。この章では，CC78K0R が持つライブラリ関数について説明します。

10.1 関数間のインタフェース

ライブラリ関数は，関数呼び出しで利用します。関数の呼び出しは，call 命令により行います。引数はスタック，返り値はレジスタにより受け渡しが行われます。

ただし，可能であれば，第 1 引数もレジスタにより受け渡します。

10.1.1 引数

引数をスタックへ積むことと取り去ることは，呼び出す側が行います。呼び出される側はその値の参照だけを行います。

ただし，引数をレジスタにより受け渡した場合は，呼び出された側が直接レジスタを参照し，必要に応じ，別のレジスタに引数の値のコピーを行います。

引数をスタック渡しする場合は，引数は最後から先頭に向かう順番でスタックに積まれます。

スタックに積まれる最小単位は 16 ビットであり，16 ビットより大きい型は上位から順番に 16 ビット単位で積まれます。8 ビットの型は，16 ビットに拡張されます。

次に，引数の受け渡し一覧を示します。

標準ライブラリの関数インタフェース（引数の受け渡し，返り値の格納）は，通常関数と同じです。

表 10-1 第 1 引数受け渡し一覧

第 1 引数の型	受け渡し方法
1 バイト，2 バイト整数	AX
near データ・ポインタ	AX
3 バイト整数	AX，BC
4 バイト整数	AX，BC
関数ポインタ， far データ・ポインタ	AX，BC
浮動小数点数（float 型）	AX，BC
浮動小数点数（double 型）	float 型として扱う
その他	スタック渡し

備考 上記の型で，1 ～ 4 バイト整数には，構造体，共用体を含みます。

10.1.2 返り値

返り値は，最小単位を 16 ビットとして，レジスタ BC から DE まで下位から 16 ビット単位で格納します。構造体を返す場合は，構造体の先頭アドレスを BC，DE に格納します。

返り値格納一覧を，次に示します。返り値の格納方法は，通常関数の場合と同じです。

表 10-2 返り値格納一覧

返り値の型	格納方法
1 ビット	CY
1 バイト，2 バイト整数	BC
near ポインタ	BC
4 バイト整数	BC（下位），DE（上位）
far ポインタ	BC（下位），DE（上位）
浮動小数点数（float 型）	BC（下位），DE（上位）
浮動小数点数（double 型）	float 型として扱う
構造体	返却する構造体を関数固有の領域にコピーし，アドレスを BC，DE に格納します。

10.1.3 個々のライブラリによる使用レジスタの保存

HL を使用するライブラリは，それらの使用するレジスタをスタックに保存します。

saddr 領域を使用するライブラリは，使用する saddr 領域をスタックに保存します。

また，ライブラリが使用するワーク・エリアは，スタック領域を使用します。

引数と返り値の受け渡し手順の例（スモール・モデル，ミディアム・モデルの場合）を次に示します。

＜呼び出す関数＞

```
"long func ( int a , long b , char *c ) ;"
```

(1) 引数をスタックに積む（呼び出す側）

c，b の上位 16 ビット，b の下位 16 ビットの順にスタックに積まれます。a は AX レジスタ渡しとなります。

(2) call 命令により func を呼び出す（呼び出す側）

b の下位 16 ビットの次に戻り番地が積まれ，関数 func に制御が移ります。

(3) 関数内で使用するレジスタを保存する（呼び出される側）

HL を使用する場合，HL がスタックに積まれます。

(4) レジスタで渡された第 1 引数をスタックに積む（呼び出される側）

(5) 関数 func の処理を行い，返り値をレジスタに格納する（呼び出される側）

返り値“long”の下位 16 ビットが BC に，上位 16 ビットが DE に格納されます。

(6) 格納した第 1 引数を復帰する（呼び出される側）

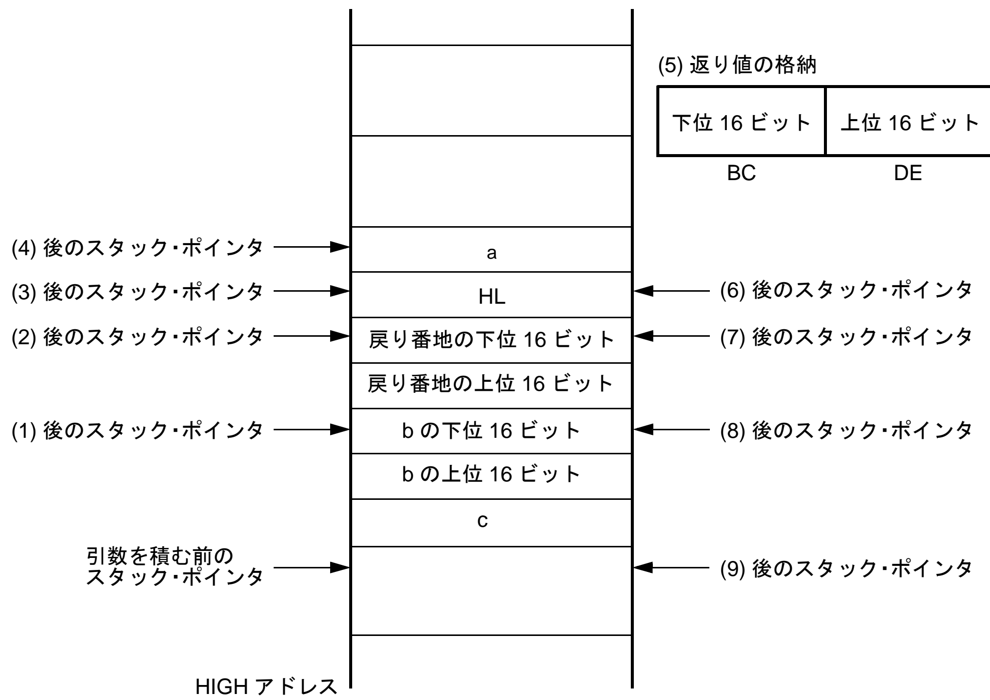
(7) 退避したレジスタを復帰する（呼び出される側）

(8) ret 命令で呼び出した関数に制御を戻す（呼び出される側）

(9) 引数をスタックから取り除く（呼び出す側）

引数のバイト数（2 バイト単位）がスタック・ポインタに加えられます。

スタック・ポインタに 6 が加えられます。



10.2 ヘッダ・ファイル

CC78K0R には，13 個のヘッダ・ファイルがあり，標準ライブラリ関数，型名，マクロ名を定義，または宣言しています。

CC78K0R のヘッダ・ファイルを次に示します。

```
ctype.h     setjmp.h    stdarg.h    stdio.h     stdlib.h
string.h    error.h     errno.h     limits.h    stddef.h
math.h      float.h     assert.h
```

(1) ctype.h

ctype.h は，文字・文字列関数を定義します。

ctype.h では，次の関数が定義されています。

ただし，コンパイラ・オプション -za（ANSI 規定外の機能を無効とし，ANSI 規定の一部の機能を有効とするオプション）を指定した場合は，_toupper，_tolower の定義を行わず，代わりに tolow，toup の定義を行います。-za を指定しない場合は，tolow，toup の定義は行われません。また，オプション，および指定モデルにより，宣言する関数が異なります。

```
isalnum     isalpha     iscntrl     isdigit     isgraph
islower     isprint     ispunct     isspace     isupper
isxdigit    tolower     toupper     isascii     toascii
_tolower    _toupper    tolow       toup
```

(2) setjmp.h

setjmp.h は，プログラム制御関数を定義します。

setjmp.h では，次の関数が定義されています。なお，オプション，および指定モデルにより，宣言する関数が異なります。

```
setjmp      longjmp
```

setjmp.h では，次のオブジェクトが宣言されています。

[int 型配列の型 “ jmp_buf ” の宣言]

```
typedef int     jmp_buf [ 12 ]
```

(3) stdarg.h

stdarg.h は，特殊関数を定義します。

stdarg.h では，次の関数が定義されています。

```
va_arg          va_start        va_starttop     va_end
```

stdarg.h では，次のオブジェクトが定義されています。

[char へのポインタ型 “ va_list ” の宣言]

```
typedef char    *va_list  ;
```

(4) stdio.h

stdio.h は，入出力関数を定義します。stdio.h では，次の関数が定義されています。

ただし，オプション，および指定モデルにより，宣言する関数が異なります。

```
sprintf         sscanf          printf          scanf           vprintf
vsprintf        getchar         gets            putchar         puts
__putc
```

次のマクロ名を宣言しています。

```
#define  EOF     ( -1 )
```

(5) stdlib.h

stdlib.h は，文字・文字列関数，メモリ関数，プログラム制御，および数学関数，特殊関数を定義します。

stdlib.h では，次の関数が定義されています。

ただし，コンパイラ・オプション -za（ANSI 規定外の機能を無効とし，ANSI 規定の機能を有効とするオプション）を指定した場合は，brk，sbrk，itoa，ltoa，ultoa の定義は行わず，代わりに strbrk，strsbrk，stritoa，strltoa，strultoa の定義を行います。-za を指定しない場合は，これらの関数の定義は行われません。

```
atoi            atol            strtol          strtoul         calloc
free            malloc          realloc         abort           atexit
exit            abs             div             labs            ldiv
brk             sbrk            atof            strtod          itoa
ltoa            ultoa           rand            srand           bsearch
qsort           strbrk          strsbrk         stritoa         strltoa
strultoa
```

stdlib.h では，次のオブジェクトが宣言されています。

[int 型のメンバ“quot”，“rem”を持つ構造体型“div_t”の宣言]

```
typedef struct {
        int     quot ;
        int     rem ;
} div_t ;
```

[マクロ名“RAND_MAX”の定義]

```
#define RAND_MAX        32767
```

[マクロ名の宣言]

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

(6) string.h

string.h は，文字・文字列関数，メモリ関数，および特殊関数を定義します。

string.h では，次の関数が定義されています。

ただし，オプション，および指定モデルにより，宣言する関数が異なります。

```
memcpy          memmove         strcpy          strncpy         strcat
strncat         memcmp          strcmp          strncmp         memchr
strchr          strcspn         strpbrk         strrchr         strspn
strstr          strtok          memset          strerror        strlen
strcoll         strxfrm
```

(7) error.h

error.h は，errno.h をインクルードしています。

(8) errno.h

次のオブジェクトが定義されています。

[マクロ名“EDOM”，“ERANGE”，“ENOMEM”の定義]

```
#define EDOM    1
#define ERANGE  2
#define ENOMEM  3
```

[volatile int 型の外部変数“errno”の宣言]

```
extern volatile int errno ;
```

(9) limits.h

limits.h では，次のマクロ名が定義されています。

```
#define CHAR_BIT        8
#define CHAR_MAX        +127
#define CHAR_MIN        -128
#define INT_MAX         +32767
#define INT_MIN         -32768
#define LONG_MAX        +2147483647
#define LONG_MIN        -2147483648

#define SCHAR_MAX       +127
#define SCHAR_MIN       -128
#define SHRT_MAX        +32767
#define SHRT_MIN        -32768
#define UCHAR_MAX       255U
#define UINT_MAX        65535U
#define ULONG_MAX       4294967295U
#define USHRT_MAX       65535U

#define SINT_MAX        +32767
#define SINT_MIN        -32768
#define SSHRT_MAX       +32767
#define SSHRT_MIN       -32768
```

ただし，修飾子なし char を unsigned char とみなす -qu オプションを指定した場合は，コンパイラが宣言するマクロ __CHAR_UNSIGNED__ により，CHAR_MAX，CHAR_MIN を次のように宣言します。

```
#define CHAR_MAX        ( 255U )
#define CHAR_MIN        ( 0 )
```

(10)stddef.h

stddef.h では，次のオブジェクトが宣言，定義されています。

[int 型の型“ptrdiff_t”の宣言]

```
typedef int     ptrdiff_t ;
```

[unsigned int 型の型“size_t ”の宣言]

```
typedef unsigned int    size_t ;
```

[マクロ名“NULL”の定義]

```
#define NULL    ( void * ) 0 ;
```

[マクロ名“offsetof”の定義]

```
#define offsetof ( type , member )  ( (size_t) & (((type*)0) -> member) )
```

備考 offsetof (*型* , *メンバ指示子*)

型 size_t を持つ汎整数定数式に展開し，その値は，（型が指示する）構造体の先頭から（メンバ指示子が指示する）構造体メンバまでのバイト単位でのオフセット値とします。

メンバ指示子は，static 型 t ; という宣言があった場合，式＆（t. メンバ指示子）を評価した結果がアドレス定数になるものでなければなりません。指定されたメンバがビット・フィールドの場合，その動作は保証しません。

(11)math.h

math.h では，次の関数が定義されています。

```
acos        asin        atan        atan2       cos
sin         tan         cosh        sinh        tanh
exp         frexp       ldexp       log         log10
modf        pow         sqrt        ceil        fabs
floor       fmod        matherr     acosf       asinf
atanf       atan2f      cosf        sinf        tanf
coshf       sinhf       tanhf       expf        frexpf
ldexpf      logf        log10f      modff       powf
sqrtf       ceilf       fabsf       floorf      fmodf
```

次のオブジェクトが，定義されています。

[マクロ名“HUGE_VAL”の定義]

```
#define HUGE_VAL        DBL_MAX
```

(12)float.h

float.h では，次のオブジェクトが定義されています。

double 型の大きさが 32 ビットのときコンパイラが宣言するマクロ，__DOUBLE_IS_32BITS__ により，定義するマクロを切り分けます。

```
#ifndef _FLOAT_H

#define FLT_ROUNDS      1
#define FLT_RADIX       2

#ifdef  __DOUBLE_IS_32BITS__
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    24
#define LDBL_MANT_DIG   24

#define FLT_DIG         6
#define DBL_DIG         6
#define LDBL_DIG        6

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -125
#define LDBL_MIN_EXP    -125

#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -37
#define LDBL_MIN_10_EXP -37

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +128
#define LDBL_MAX_EXP    +128

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +38
#define LDBL_MAX_10_EXP +38

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         3.40282347E+38F
#define LDBL_MAX        3.40282347E+38F

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     1.19209290E-07F
#define LDBL_EPSILON    1.19209290E-07F

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         1.17549435E-38F
#define LDBL_MIN        1.17549435E-38F

#else   /* __DOUBLE_IS_32BITS__ */
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    53
#define LDBL_MANT_DIG   53

#define FLT_DIG         6
#define DBL_DIG         15
#define LDBL_DIG        15

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -1021
#define LDBL_MIN_EXP    -1021
```

```
#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -307
#define LDBL_MIN_10_EXP -307

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +1024
#define LDBL_MAX_EXP    +1024

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +308
#define LDBL_MAX_10_EXP +308

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         1.7976931348623157E+308
#define LDBL_MAX        1.7976931348623157E+308

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     2.2204460492503131E-016
#define LDBL_EPSILON    2.2204460492503131E-016

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         2.225073858507201E-308
#define LDBL_MIN        2.225073858507201E-308
#endif  /* __DOUBLE_IS_32BITS__ */

#define _FLOAT_H
#endif  /* !_FLOAT_H */
```

(13)assert.h

assert.h では，次の関数が定義されています。

```
__assertfail
```

assert.h では，次のオブジェクトが定義されています。

```
#ifdef  NDEBUG
#define assert ( p )    ( ( void ) 0 )
#else
extern  int     __assertfail ( char *__msg , char *__cond , char *__file ,
int__line ) ;
#define assert ( p )    ( ( p ) ? ( void ) 0 : ( void ) __assertfail (
                            "Assertion failed : %s , file %s , line %d¥n" ,
                        #p , __FILE__ , __LINE__ ) )
#endif  /* NDEBUG */
```

ただし，assert.h ヘッダ・ファイルは，assert.h ヘッダ・ファイルでは定義しないもう 1 つのマクロ NDEBUG を参照し，ソース・ファイル中に assert.h を取り込む時点で，NDEBUG がマクロとして定義されている場合，assert マクロを単に，次のように宣言し，__assertfail の定義も行いません。

```
#define assert ( p )    ( ( void ) 0 )
```

10.3 リエントラント性

リエントラントとは，あるプログラムから呼び出されている関数が，続けて他のプログラムによって呼び出し可能である状態です。

CC78K0R の標準ライブラリは，リエントラント性を考慮し，静的領域を使用していません。したがって，関数が使用する記憶域のデータが，他プログラムからの呼び出しによって破壊されることはありません。

ただし，次の関数は，リエントラントでないので注意してください。

- リエントラント化できない関数

```
setjmp , longjmp , atexit , exit
```

- スタートアップ・ルーチンで確保している領域を使用する関数

```
div , ldiv , brk , sbrk , rand , srand , strtok
```

- 浮動小数点を扱う関数

```
sprintf , sscanf , printf , scanf , vprintf , vsprintf 注
atof , strtod , すべての数学関数
```

注 sprintf，sscanf，printf , scanf , vprintf , vsprintf のうち，浮動小数点未対応のものは，リエントラントです。

10.4 標準ライブラリ関数

CC78K0R の標準ライブラリ関数を機能別に分けて説明します。

標準ライブラリは，すべて -zf オプション指定時もサポートしています。

表 10-3 標準ライブラリ関数一覧

関数の種類	関数名
文字／文字列関数	is ～
	toupper，tolower
	toascii
	_toupper/toup，_tolower/tolow
プログラム制御関数	setjmp，longjmp
特殊関数	va_start，va_starttop，va_arg，va_end
入出力関数	sprintf
	sscanf
	printf
	scanf
	vprintf
	vsprintf
	getchar
	gets
	putchar
	puts
	__putc

関数の種類	関数名
ユーティリティ関数	atoi，atol
	strtol，strtoul
	calloc
	free
	malloc
	realloc
	abort
	atexit，exit
	abs，labs
	div，ldiv
	brk，sbrk
	atof，strtod
	itoa，ltoa，ultoa
	rand，srand
	bsearch
	qsort
	strbrk
	strsbrk
	stritoa，strltoa，strultoa

関数の種類	関数名
文字列／メモリ関数	memcpy，memmove
	strcpy，strncpy
	strcat，strncat
	memcmp
	strcmp，strncmp
	memchr
	strchr，strrchr
	strspn，strcspn
	strpbrk
	strstr
	strtok
	memset
	strerror
	strlen
	strcoll
	strxfrm
数学関数	acos
	asin
	atan
	atan2
	cos
	sin
	tan
	cosh
	sinh
	tanh
	exp
	frexp
	ldexp
	log
	log10
	modf
	pow

関数の種類	関数名
数学関数	sqrt
	ceil
	fabs
	floor
	fmod
	matherr
	acosf
	asinf
	atanf
	atan2f
	cosf
	sinf
	tanf
	coshf
	sinhf
	tanhf
	expf
	frexpf
	ldexpf
	logf
	log10f
	modff
	powf
	sqrtf
	ceilf
	fabsf
	floorf
	fmodf
診断関数	__assertfail

10.4.1 引数／返り値に適したライブラリの使用

標準ライブラリの引数／返り値にポインタを指定するものは，メモリ・モデルに応じて適切なライブラリがリンクされます。

メモリ・モデルのデフォルトでないポインタを扱いたい場合，以下の標準関数名で呼び出すことで，そのポインタに適切なライブラリをリンクすることが可能です。

< 関数名 >_n　：ポインタを常に near として扱う

< 関数名 >_f　：ポインタを常に far として扱う

例えば，スモール・モデル選択時に，strcmp 関数の引数として far ポインタを指定することができます。

＜例＞

```
#include <string.h>

__far char * sf1;
__far char * sf2;

func ( ) {
          :
        r = strcmp_f ( sf1 , sf2 );
          :
}
```

【注意事項】

- スモール・モデル，ミディアム・モデル指定時の可変個引数を取り扱う入出力関数 sprintf/printf/vprintf/vsprintf/sscanf/scanf のポインタ引数は，near ポインタとして扱います。関数ポインタは使用することができません。
  関数ポインタ使用時，あるいは far ポインタ使用時は，printf_f を使用し，可変個引数のポインタをすべて far ポインタにキャストする必要があります。
  コンパクト・モデル，ラージ・モデル指定時の可変個引数を取り扱う入出力関数 sprintf/printf/vprintf/vsprintf/sscanf/scanf のポインタ引数は，far ポインタとして扱います。
- スモール・モデル，ミディアム・モデル指定時の可変個引数を取り扱う特殊関数 va_start/va_starttop/va_arg/va_end のポインタ引数は，near ポインタとして扱います。関数ポインタは使用することができません。

10.5　文字／文字列関数

文字／文字列関数には，次のものがあります。

- is ～
- toupper，　tolower
- toascii
- _toupper/toup，　_tolower/tolow

is ～

【機能】

- is ～は，文字種の判定を行います。

【ヘッダ・ファイル】

- すべて ctype.h

【関数プロトタイプ】

- int is ～ (int *c*) ;

関数名	引数	返り値
is ～	*c* ： 判定する文字	文字 *c* が目的の文字である場合 ： 1 文字 *c* が目的の文字でない場合 ： 0

【説明】

関数名	範囲
isalpha	*c* が英文字（A ～ Z，a ～ z）であるかを判定します。
isupper	*c* が英大文字（A ～ Z）であるかを判定します。
islower	*c* が英小文字（a ～ z）であるかを判定します。
isdigit	*c* が数字（0 ～ 9）であるかを判定します。
isalnum	*c* が英数字（0 ～ 9，A ～ Z，a ～ z）であるかを判定します。
isxdigit	*c* が 16 進数字（0 ～ 9，A ～ F，a ～ f）であるかを判定します。
isspace	*c* が空白文字（空白，タブ，復帰，改行，垂直，タブ，改ページ）であるかを判定します。
ispunct	*c* が空白文字と英数字以外の表示可能文字であるかを判定します。
isprint	*c* が表示可能文字であるかを判定します。
isgraph	*c* が空白以外の表示可能文字であるかを判定します。
iscntrl	*c* がコントロール文字であるかを判定します。
isascii	*c* が ASCII コードであるかを判定します。

toupper, tolower

【機能】

- 文字種の変換を行います。
- toupper は，英小文字を英大文字に変換します。
- tolower は，英大文字を英小文字に変換します。

【ヘッダ・ファイル】

- ctype.h

【関数プロトタイプ】

- int toupper (int *c*) ;
- int tolower (int *c*) ;

関数名	引数	返り値
toupper, tolower	*c* : 変換される文字	*c* が変換可能な場合 : 文字 *c* に対応した変換後の文字 *c* が変換不可能な場合 : *c*

【説明】

toupper

- toupper は，引数が英小文字であることを確認したうえで，英大文字に変換します。

tolower

- tolower は，引数が英大文字であることを確認したうえで，英小文字に変換します。

toascii

【機能】

- toascii は，ASCII コードへの変換を行います。

【ヘッダ・ファイル】

- ctype.h

【関数プロトタイプ】

- int toascii (int *c*) ;

関数名	引数	返り値
toascii	*c* : 変換される文字	*c* の ASCII コードの範囲以外のビットを 0 にした値 *c*

【説明】

- *c* の ASCII コードに変換します。ASCII コードの範囲（ビット 0 ～ 6）以外のビット（ビット 7 ～ 15）は 0 にします。

_toupper/toup，_tolower/tolow

【機能】

- _toupper/toup は，*c* から“a”を引き，“A”を加えます。
  _tolower/tolow は，*c* から“A”を引き，“a”を加えます。
  （_toupper と toup，_tolower と tolow はまったく同じです。）

  備考　a：英小文字，A：英大文字

【ヘッダ・ファイル】

- ctype.h

【関数プロトタイプ】

- int _toupper/toup (int *c*) ;
- int _tolower/tolow (int *c*) ;

関数名	引数	返り値
_toupper/toup	*c* ： 変換される文字	*c* から“a”を引き，“A”を加えた値
_tolower/tolow		*c* から“A”を引き，“a”を加えた値

備考　a：英小文字，A：英大文字

【説明】

_toupper

- _toupper は，toupper と似ていますが，引数が英小文字であることを確認しません。

_tolower

- _tolower は，tolower と似ていますが，引数が英大文字であることを確認しません。

10.6 プログラム制御関数

プログラム制御関数には，次のものがあります。

- setjmp， longjmp

setjmp，longjmp

【機能】

- setjmp は，呼び出し時の環境をセーブします。
- longjmp は，setjmp でセーブされた環境を復帰します。

【ヘッダ・ファイル】

- setjmp.h

【関数プロトタイプ】

- int setjmp (jmp_buf *env*) ;
- void longjmp (jmp_buf *env* , int *val*) ;

関数名	引数	返り値
setjmp	*env* ： 環境をセーブする配列	直接呼び出された場合 ： 0 対応する longjmp の呼び出しから返る場合 ： 対応する longjmp の呼び出し時の *val* の値，ただし *val* が 0 の場合は 1
longjmp	*env* ： setjmp でセーブした環境の配列 *val* ： setjmp に返す値	*env* に環境をセーブした setjmp の次に実行を移すので，longjmp には戻りません。

【説明】

setjmp

- setjmp は，直接呼び出された場合，HL レジスタ，レジスタ変数として使用する saddr 領域，SP，および関数のリターン・アドレスを *env* にセーブし，0 を返します。

longjmp

- longjmp は，*env* に保存された環境（HL レジスタ，およびレジスタ変数として使用する saddr 領域，SP）を復帰し，対応する setjmp が *val*（ただし，*val* が 0 の場合は 1）を返したかのようにプログラムの実行が続きます。

10.7 特殊関数

特殊関数には，次のものがあります。

- va_start， va_starttop，va_arg， va_end

va_start，va_starttop， va_arg，va_end

【機能】

- va_start は，可変個の引数の処理のための設定を行います（マクロ）。
- va_starttop は，可変個の引数の処理のための設定を行います（マクロ）。
- va_arg は，可変個の引数の処理を行います（マクロ）。
- va_end は，可変個の引数の処理の終了を知らせます（マクロ）。

【ヘッダ・ファイル】

- stdarg.h

【関数プロトタイプ】

- void va_start (va_list *ap* , *parmN*) ;
- void va_starttop (va_list *ap* , *parmN*) ;
- type va_arg (va_list *ap* , type) ;
- void va_end (va_list *ap*) ;

備考 va_list は，stdarg.h で typedef 定義されています。

関数名	引数	返り値
va_start, va_starttop	*ap* ： va_arg，va_end で使えるように初期化される変数 *parmN* ： 可変引数の 1 個前の引数	なし
va_arg	*ap* ： 引数リストの処理のための変数 type ： 変引数の該当箇所をポイントするための型（type は可変長の型で，たとえば va_arg (va_list *ap* , int) と記述すれば int 型，va_arg (va_list *ap*, long) と記述すれば long 型となる）	正常の場合 ： 可変引数の該当箇所の値 *ap* が NULL ポインタの場合 ： 0
va_end	*ap* ： 可変個の引数の処理のための変数	なし

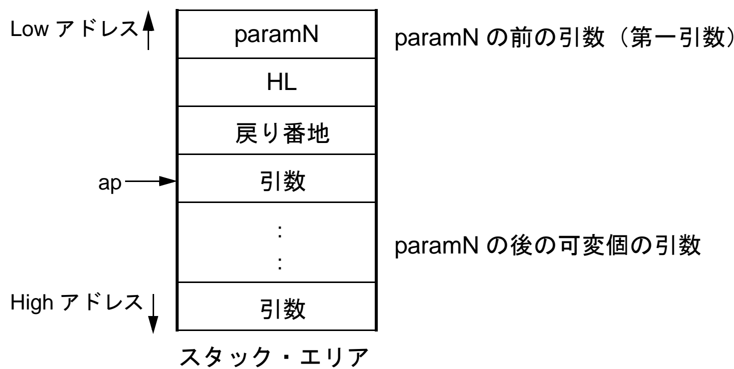
【説明】

va_start

- va_start で，引数 ap は va_list 型（char * 型）のオブジェクトです。
- *ap* に *parmN* の次の引数を指すポインタを格納します。
- *parmN* は，関数定義上での右端のパラメータの名前です。
- *parmN* がレジスタ記憶クラスで宣言されている場合は，正常動作は保証されません。
- *parmN* が第一引数の場合は，正常動作は保証されません（代わりに va_starttop を使用してください）。

va_starttop

- *ap* は，va_list 型のオブジェクトです。
- *ap* に，*parmN* の次の引数を指すポインタをストアします。
- *parmN* は，関数定義上での右端，かつ 1 番目のパラメータの名前です。
- *parmN* がレジスタ記憶クラスで宣言されている場合は，正常動作は保証されません。
- *parmN* が第一引数以外の場合は，正常動作は保証されません。



va_arg

- va_arg では，引数 *ap* は va_start で初期化された va_list 型の *ap* と同じでなければなりません（それ以外の正常動作は保証されません）。
- 可変引数の該当箇所（va_start の直後は可変引数の先頭，その後は va_arg ごとに進めます）の値を type 型で返します。
- *ap* が NULL ポインタの場合は，type 型の 0 を返します。
- CC78K0R では，引数リストとしてポインタを指定する場合，ミディアム・モデルでは near データ・ポインタ（2 バイト長）を，ラージ・モデルでは far データ・ポインタ（4 バイト長）とする必要があります。また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数リストとして指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

va_end

- va_end は，すべての可変引数を処理し終わったことをマクロ系に知らせるために，*ap* に NULL ポインタをセットします。

10.8 入出力関数

入出力関数には，次のものがあります。

- sprintf
- sscanf
- printf
- scanf
- vprintf
- vsprintf
- getchar
- gets
- putchar
- puts
- __putc

sprintf

【機能】

- sprintf は，フォーマットに従ってデータを文字列に書きます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int sprintf (char **s* , const char **format* , …) ;

関数名	引数	返り値
sprintf	*s* ： 出力する文字列へのポインタ *format* ： 出力変換仕様を示す文字列へのポインタ … ： 変換される 0 個以上の引数	*s* に書かれた文字数（終端のヌル文字は数えません）

【説明】

- 書式に対して実引数が不足しているときの動作は保証されません。実引数が残っているにもかかわらず，書式が尽きてしまう場合，余分の実引数は評価するだけで無視します。
- *format* で指定された出力変換仕様に従い，*format* の後ろに続く（0 個以上の）引数を変換して *s* で示された文字列に書き出します。
- 出力変換仕様は，0 個以上の指令です。通常の文字（%で始まる変換仕様以外）は，そのまま文字列 *s* に出力します。変換仕様は，（0 個以上の）後続の引数を取り出し，変換して文字列 *s* に出力します。
- 各変換仕様は % で始まり，次のものが順に続きます（変換指定が不正な場合には，その文字を出力します。この際，フラグと最小フィールド幅は有効です）。

  (1) 0 個以上のフラグ（後述）は，変換仕様の意味を修飾します。

  (2) 最小フィールド幅を指定するオプションの 10 進整数

  もし，変換後の幅が，このフィールド幅よりも小さい場合，左にパッドを入れます（左寄せのフラグ（ - ）が指定されていれば，右にパッドが入ります）。パッドは，フィールド幅整数が 0 で始まり，右寄せの場合は 0，その他はスペース文字です。変換後の幅がフィールド幅より多くても切り捨てません。

- オプションの精度指定（ . 整数）

  d，i，o，u，x， X 変換の場合は，最小の桁数を指定します。

  s 変換では，最大文字数を指定します。

  e，E，および f 変換については，小数点文字の後ろに出力すべき桁数を，g，および G 変換については最大の有効桁数を指定します。

  この精度指定は，（ . 整数）の形をしています。整数部が省略されたときは 0 とみなします。

  この精度指定から生ずるパッドの量は，フィールド幅指定のパッドに優先します。

- オプションの h，l，または L

  h は，引き続く d，i，o，u，x， X 変換を short int，または unsigned short int に対して行うように指定します。また，h は引き続く n 変換を short int へのポインタに対して行うように指定します。

  l は，引き続く d，i，o，u，x， X 変換を long int，または unsigned long int に対して行うように指定します。また，l は引き続く n 変換を long int へのポインタに対して行うように指定します。

  その他の変換に対しては，h，l，または L は無視します。

- 変換を指定する文字（後述の変換指定）

  フィールド幅，または精度指定は，整数文字列の代わりに * を指定することができます。このとき，int 引数が整数値を与えます（変換される引数の前）。

  この結果生じる負のフィールド幅は，- フラグのあとに正のフィールドが続いたものと解釈します。負の精度は無視されます。

フラグは次のとおりです。

フラグ	内容
-	変換した結果をフィールド内で左寄せします。
+	符号付き変換の結果に，+，または - の符号を付けます。
スペース	符号付き変換の結果に符号がない場合，スペースを頭に付けます。 スペースと + フラグを同時に指定すると，スペース・フラグは無視されます。
#	結果を“代替形式”に変換します。 o 変換では，最初の桁が 0 になるように精度を上げます。 x， X 変換では，非ゼロの結果 には 0x（または 0X）が頭に付きます。 e，E，f 変換では，すべての場合に出力値に強制的に小数点が入ります（# なしのデフォルトでは，後続の数値がある場合にのみ小数点が表示されます）。 g，G 変換では，すべての場合に出力値に強制的に小数点が入り，後続する 0 の切り捨てを許しません（# なしのデフォルトでは，後続の数値がある場合にのみ小数点が表示されます。後続の 0 は切り捨てられます）。 その他の変換では，# フラグを無視します。
0	先頭フィールドを 0 で埋めます。 - フラグと同時に指定した場合は，0 フラグは無視されます。 d，i，o，u，x，X 変換で精度を指定している場合は，0 フラグは無視されます。

変換指定は次のとおりです。

変換指定	内容
d，i	int 引数を符号付き 10 進（d または i）表記に変換します。
o	int 引数を符号付き無符号 8 進（o）表記に変換します。
u	int 引数を符号付き無符号 10 進（u）表記に変換します。
x，X	int 引数を符号付き無符号 16 進（x または X）表記に変換します。 x 変換は a ～ f，X 変換は A ～ F の文字を 16 進文字として使います。

精度指定は，結果の最小桁数を指定し，結果が足りないときには頭の不足分の 0 を付けます。

精度指定の省略時は，1 とします。

0 を精度指定 0 で変換すると，何も現れません。

精度指定	内容
f	double 引数を [-]*dddd.dddd* の形式を持つ符号付きの値として変換します。 *dddd* は，1 個，または複数の 10 進数です。小数点の前の桁数はその数の絶対値によって決定され，小数点のあとの桁数は要求された精度によって決定されます。 精度が省略された場合は，精度を 6 として解釈します。
e	double 引数を [-]*d.dddd* e [sign] *ddd* の形式を持つ符号対の値として変換します。 d は 1 個の 10 進数，*dddd* は 1 個，または複数の 10 進数です。*ddd* は正確に 3 桁の 10 進数で，sign は +，または - です。 精度が省略された場合は，精度を 6 として解釈します。
E	指数の前に付くのが e ではなく，E である点を除いて，e のフォーマットと同様です。
g	double 引数を f，または e のフォーマットのうち，指定された精度に基づいて変換したときに，より短くなる方式を用います。 e フォーマットは，値の指数部が，-4 より小さいか，精度で指定された数よりも大きい場合にのみ用います。 あとに続く 0 は切り捨てられ，小数点は 1 個，または複数の数字が続く場合にのみ表示されます。
G	指数の前にあるのが e ではなく，E である点を除いて，g のフォーマットと同様です。
c	int 引数を unsigned char に変換し，結果の文字が書かれます。
s	引数は文字列へのポインタで，その文字列からの各文字は終端のヌル文字（出力には含みません）まで書かれます。 精度指定があれば，それより多くの文字は書きません。 精度が指定されない場合，または精度が配列の大きさよりも大きい場合，配列はヌル文字を含まなければなりません。
p	引数は void へのポインタの値を無符号 16 進 4 桁で表記（4 桁未満は頭に 0 を付けます）します。 ラージ・モデルは，無符号 16 進 8 桁で表記（上位 2 桁 0 でパディングし，6 桁未満は頭に 0 を付ける）します。 精度指定は無視します。
n	引数は整数へのポインタで，そこに対してこれまでに文字列 *s* に書き出した文字数を入れます。 変換は行いません。
%	% が書かれます。 引数は変換しません（フラグと最小フィールド幅は有効です）。

- 無効な変換指定子に対する動作は，保証しません。
- 実引数が共用体，または集成体であるか，またはそれを指すポインタである場合（% s 変換のときの文字型配列，または %p 変換のときのポインタを除きます），動作は保証されません。
- フィールド幅が存在しないとき，または小さいときでも，変換結果を切り捨てることはありません。すなわち，変換結果の文字数がフィールド幅より大きい場合，その変換結果を含む幅までフィールドを拡張します。
- %f，%e，%E，%g，%G 変換時の特別の出力文字列の形式を次に示します。

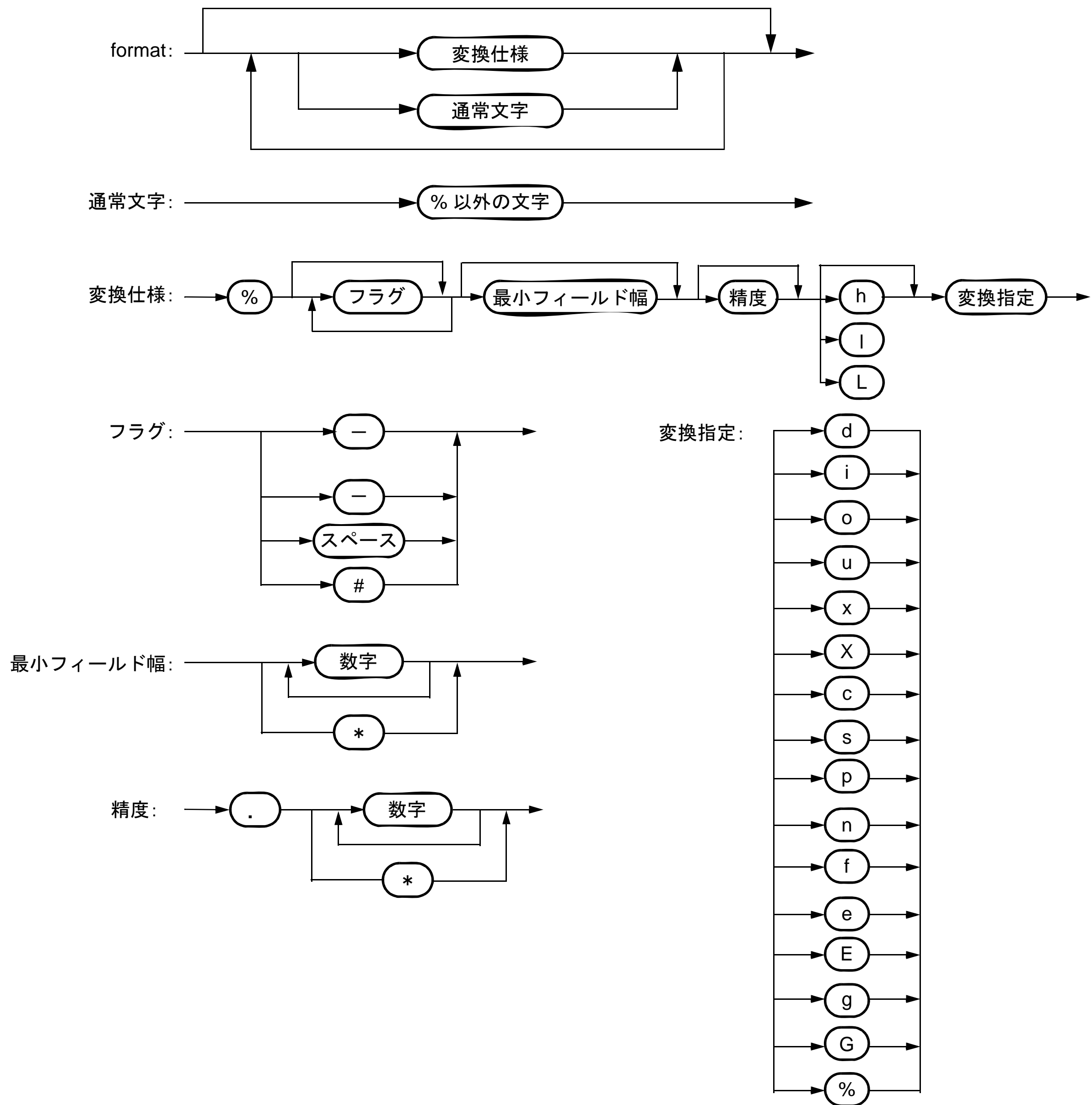
  非数　→　“(NaN)”

  + ∞　→　“(+INF)”

  - ∞　→　“(-INF)”

  文字列 *s* の末尾にヌル文字（返り値のカウントには含まない）を書きます。

format の構文図を，次に示します。



- CC78K0R では，引数としてポインタを指定する変換指定 s/p/n について，ミディアム・モデルでは near データ・ポインタ（2 バイト長）を，ラージ・モデルでは far データ・ポインタ（4 バイト長）を指定する必要があります。

  また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数として指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

sscanf

【機能】

- 入力文字列からフォーマットに従ってデータを読みます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int sscanf (const char **s* , const char **format* , …) ;

関数名	引数	返り値
sscanf	*s* : 入力文字列へのポインタ *format* : 入力変換仕様を示す文字列へのポインタ … : 変換された値を入れるオブジェクトへのポインタ（0 個以上の）引数	文字列 *s* が空の場合 : -1 文字列 *s* が空でない場合 : 代入された入力項目の数

【説明】

- *s* が指す文字列から入力します。*format* が指す文字列により許される入力列を指定します。
format 以降の引数をオブジェクトへのポインタとして用います。*format* は入力列から，どのように変換するかを指定します。
- *format* に対して引数が足りない場合の正常動作は保証されません。過剰な引数の場合，式の評価は行いますが入力はされません。
- *format* は 0 以上の指令からなります。指令は次のとおりです。

  1 : 1 個以上の空白文字（isspace が真となる文字）

  2 : 通常文字（% 以外）

  3 : 変換指示

- 変換指示は % から始まり，% の後ろに次のものが順に続きます。

  (1) オプションの代入禁止文字 *（引数へは代入しないことを示します）

  (2) オプションの最大フィールド幅を指定する 10 進整数（0 の場合，指定のないものとします）

  (3) オプションの h，l，または L（受信する側のオブジェクトのサイズを示します）
  変換指示子 d，i，n，o，x に h が先行すれば，引数は int でなく short int へのポインタです。l がこれらに先行した場合は long int へのポインタです。
  同様に，変換指示子 u に h が先行すれば，引数は unsigned short int へのポインタです。l が先行した場合は，unsigned long int へのポインタです。

変換指示子 e，E，f，g，G に l が先行すれば，引数は double へのポインタです（l なしのデフォルトでは引数は float へのポインタ）。また，L が先行した場合，無視します。

備考 変換指示子：対応する変換の種類を示す文字（後述）

sscanf は *format* 中の指令を順に実行します。指令が失敗すれば，sscanf は戻ります。

(1) 空白文字からなる指令は，最初の非空白文字（これは読み込みません）までか，読む文字がなくなるまで入力を読むことで実行されます。空白文字指令は，非空白文字が発見できなければ失敗します。

(2) 通常文字の指令は，次の文字を読むことで実行されます。その文字と指令文字が異なるとき，指令は失敗します。

(3) 変換指示の指令は，各変換指示子（後述）ごとに一致する入力列の集合を定義します。
変換指示は，次のステップ順に実行されます。

(a) 入力空白文字（isspace で指定される）はスキップされます。ただし，変換指示子が [，c，n の場合を除きます。

(b) 入力項目が文字列 *s* から読まれます。ただし，n 変換指示子のときは除きます。
入力項目とは，変換指示子で指示される文字列の最初の部分列のうち，最長の入力列（ただし，最大フィールド幅が指定されている場合は，その長さで打ち切ります）と定義します。入力項目の次の文字は，まだ読まれていないとみなします。
入力項目の長さが 0 のとき，指令の実行は失敗します。

(c) % 変換指示子を除いて，入力項目（%n 指令の場合は，入力文字数）が変換指示子により定まる型に変換されます。
入力項目が指示する形式と合わない場合は指令の実行は失敗します。
* によって入力禁止が指定されないかぎり，変換の結果は，*format* に続く変換結果を受け取っていない最初の引数に指されるオブジェクトに格納されます。

変換指示子は次のとおりです。

変換指示子	内容
d	10 進整数（符号が付いてもよい）に変換します。 対応する引数は整数へのポインタです。
I	整数（符号が付いてもよい）に変換します。 数値部の先頭が 0x，または 0X の場合 16 進整数，0 の場合は 8 進整数，その他は 10 進整数とみなします。 対応する引数は整数へのポインタです。
o	8 進整数（符号が付いてもよい）に変換します。 対応する引数は整数へのポインタです。
u	無符号の 10 進整数に変換します。 対応する引数は無符号整数へのポインタです。
x	16 進整数（符号が付いてもよい）に変換します。

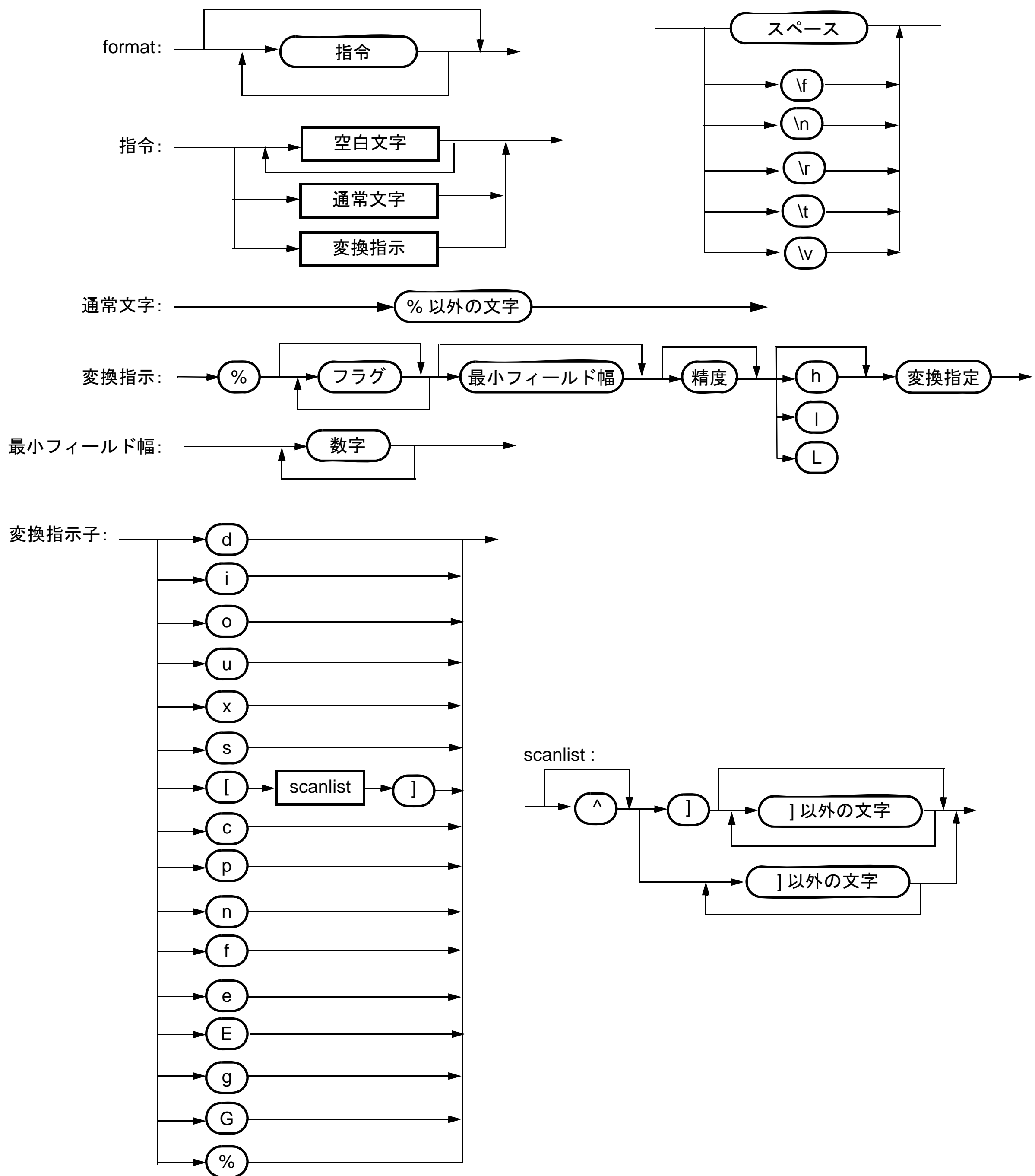
変換指示子	内容
e，E，f，g，G	オプションの符号（+，または -），小数点を含む 1 個，または複数個の連続する 10 進数，およびオプションの指数（“ e ” または “ E ”）とそれに続くオプションの符号付き整数値から構成される浮動小数点値。 変換の結果，オーバーフローとなった場合，±∞を変換結果とし，アンダーフローとなった場合，非正規化数，または，± 0 を変換結果とします。 対応する引数は，float へのポインタです。
s	非空白文字列からなる文字列を入力します。 対応する引数は整数へのポインタです。16 進整数の先頭には 0x，または 0X を付けることができます。 対応する引数は，この文字列と終端のヌル文字を収容するのに十分な大きさを持つ配列へのポインタです。 終端のヌル文字は，自動的に付加されます。
[	期待する文字群（scanset という）からなる文字列を入力します。 対応する引数は，この文字列と終端のヌル文字を収容するのに十分な大きさを持つ配列へのポインタです。 終端のヌル文字は自動的に付加されます。 変換指示は，この文字以降から右角かっこ ( ] ）まで続きます。角かっこにはさまれた文字列（scanlist という）は，左角かっこの直後の文字がサーカム・フレックス（ ^ ）の場合を除き，scanset を構成します。^ の場合は，このサーカム・フレックスから右角かっこの間の scanlist 以外のすべての文字が scanset を構成します。ただし，[ ]，または [^] で始まる場合は，この右角かっこは scanlist に入り，次の右角かっこが，scanlist の終端になります。 scanlist の左端，右端以外のハイフン（ - ）は範囲指定です。- の左の文字が右の文字より ASCII コードが小さくない場合は，ハイフンは “ - ” そのものの文字とします。
c	フィールド幅（指定のないときは 1）で指定された個数の文字からなる文字列を入力します。 対応する引数は，この文字列を収容するのに十分な大きさを持つ配列へのポインタです。 終端のヌル文字は，追加しません。
p	無符号の 16 進整数として変換します。 対応する引数は，void へのポインタのポインタです。
n	文字列 *s* からは入力しません。 対応する引数は整数へのポインタであり，これまでこの関数で文字列 *s* から読み出された文字数がそのポインタの指すオブジェクトに格納されます。 %n 指令は，返り値の代入カウントには含めません。
%	% を読みます。 いかなる変換も代入も起こりません。

変換指示が不正な場合，指令は失敗します。

入力文字列に終端のヌル文字が出現したら，sscanf は戻ります。

整数変換の場合（d，i，o，u，x，p）は，オーバーフローした場合，変換後の型のビット数より上位は切り捨てます。

format の構文図を，次に示します。



- CC78K0R では，引数としてポインタを指定する変換指定 s/p/n について，ミディアム・モデルでは near データ・ポインタ（2 バイト長）を，ラージ・モデルでは far データ・ポインタ（4 バイト長）を指定する必要があります。
  また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数として指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

printf

【機能】

- printf は，フォーマットに従ってデータを SFR に出力します。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int printf (const char **format* , …) ;

関数名	引数	返り値
printf	*format* : 出力変換仕様を示す文字列へのポインタ … : 変換される 0 個以上の引数	*s* に出力された文字数（終端のヌル文字は数えません）

【説明】

- *format* で指定された出力変換仕様に従い，*format* のあとに続く（0 個以上の）引数を変換して putchar 関数を使用して出力します。
- 出力変換仕様は，0 個以上の指令です。通常の文字（%で始まる変換仕様以外）は，そのまま putchar 関数を使用して出力します。変換仕様は（0 個以上の）後続の引数を取り出し変換して putchar 関数を使用して出力します。
- 各変換仕様は，sprintf 関数と同じです。

scanf

【機能】

- SFR からフォーマットに従ってデータを読みます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int scanf (const char **format* , …) ;

関数名	引数	返り値
scanf	*format* ： 入力変換仕様を示す文字列へのポインタ … ： 変換された値を入れるオブジェクトへのポインタ（0 個以上の）引数	文字列 *s* が空でない場合 ： 代入された入力項目の数

【説明】

- getchar 関数を使用し，入力を行います。 *format* が指す文字列により許される入力列を指定します。*format* 以降の引数をオブジェクトへのポインタとして使用します。*format* は入力列からどのように変換するかを指定します。
- *format* に対して引数が足りない場合の正常動作は保証されません。過剰な引数の場合，式の評価は行いますが入力はされません。
- *format* は 0 以上の指令からなります。指令は次のとおりです。
  - 1　：1 個以上の空白文字（isspace が真となる文字）
  - 2　：通常文字（% 以外）
  - 3　：変換指示
- 指令と矛盾する入力文字によって変換が終了した場合，その矛盾した入力文字は切り捨てます。変換指示は，sscanf 関数と同じです。

vprintf

【機能】

- vprintf は，フォーマットに従ってデータを SFR に出力します。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int vprintf (const char **format* , va_list *p*) ;

関数名	引数	返り値
vprintf	*format* : 出力変換仕様を示す文字列へのポインタ *p* : 引数並びへのポインタ	出力された文字数（終端のヌル文字は数えません）

【説明】

- *format* で指定された出力変換仕様に従い，引数並びのポインタが指す引数を変換して putchar 関数を使用し出力します。
- 各変換仕様は，sprintf 関数と同じです。

vsprintf

【機能】

- vsprintf は，フォーマットに従ってデータを文字列に書きます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int vsprintf (char **s* , const char **format* , va_list *p*) ;

関数名	引数	返り値
vsprintf	*s* : 出力を書く文字列へのポインタ *format* : 出力変換仕様を示す文字列へのポインタ *p* : 引数並びへのポインタ	*s* に出力された文字数（終端のヌル文字は数えません）

【説明】

- *format* で指定された出力変換仕様に従い，引数並びのポインタが指す引数を *s* が指す文字列に書き出します。
- 出力変換仕様は，sprintf 関数と同じです。

getchar

【機能】

- SFR から，1 文字読み込みます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int getchar (void) ;

関数名	引数	返り値
getchar	なし	SFR から読み込んだ 1 文字

【説明】

- SFR シンボル P0（ポート 0）から読み込んだ値を返します。
- 読み込みに関して，エラー・チェックは行いません。
- 読み込む SFR の変更を行う場合は，ソースを変更しライブラリに登録し直すか，ユーザが新たに getchar 関数を作成する必要があります。

gets

【機能】

- 文字列を読み取ります。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- char *gets (char **s*) ;

関数名	引数	返り値
gets	*s* : 入力文字列へのポインタ	正常な場合 : *s* 1 文字も読み取らずファイルの終わりを検出した場合 : NULL ポインタ

【説明】

- getchar 関数を使用して文字列を読み取り，*s* が指す配列に格納します。
- ファイルの終わりを検出したとき（getchar 関数が -1 を返したとき），または改行文字を読み取ったときに，文字列の読み取りは終了します。そして読み取った改行文字を捨て，最後に配列に格納した文字の最後にヌル文字を書きます。
- 正常の場合は，*s* を返します。
- ファイルの終わりを検出し，かつ配列に 1 文字も読み取っていなかった場合は，配列の内容は変化せずに残し，NULL ポインタを返します。

putchar

【機能】

- SFR に 1 文字出力します。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int putchar (int *c*) ;

関数名	引数	返り値
putchar	*c* : 出力する文字	出力した文字

【説明】

- SFR シンボル P0（ポート 0）に *c* で指定された文字を（unsigned char 型に変換して）書き込みます。
- 書き込みに関して，エラー・チェックは行いません。
- 書き込む SFR の変更を行う場合は，ソースを変更しライブラリに登録し直すか，ユーザが新たに putchar 関数を作成する必要があります。

puts

【機能】

- 文字列を出力します。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int puts (const char **s*) ;

関数名	引数	返り値
puts	*s* : 出力文字列へのポインタ	正常な場合 : 0 putchar 関数が -1 を返したとき : -1

【説明】

- putchar 関数を使用し，*s* が指す文字列を書き込みます。そして出力の最後に改行文字を追加します。
- 文字列の終端のヌル文字の書き込みは行いません。
- 正常の場合 0 を返し，putchar 関数が -1 を返したとき，-1 を返します。

__putc

【機能】

- opaque に 1 文字出力します。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- int __putc (int c , void **opaque*) ;

関数名	引数	返り値
__putc	*c* : 出力する文字 *opaque* : 文字出力先へのポインタ	出力した文字

【説明】

- *opaque* で示される文字出力先に，*c* で指定した文字を（unsigned char 型に変換して）書き込みます。また，*opaque* で示される文字出力先を 1 バイト進めます。
- *opaque* が 0 である場合，putchar 関数を呼び出し，putchar 関数の返り値を返します。

10.9 ユーティリティ関数

ユーティリティ関数には，次のものがあります。

- atoi，atol
- strtol，strtoul
- calloc
- free
- malloc
- realloc
- abort
- atexit，exit
- abs，labs
- div，ldiv
- brk，sbrk
- atof，strtod
- itoa，ltoa，ultoa
- rand，srand
- bsearch
- qsort
- strbrk
- strsbrk
- stritoa，strltoa，strultoa

atoi，atol

【機能】

- atoi は，10 進整数文字列を int に変換します。
- atol は，10 進整数文字列を long に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int atoi (const char **nptr*) ;
- long int atol (const char **nptr*) ;

関数名	引数	返り値
atoi	*nptr* : 変換する文字列	正常の場合 : 変換された値 正のオーバーフローの場合 : INT_MAX (32,767) 負のオーバーフローの場合 : INT_MIN (-32,768) 不正文字列の場合 : 0
atol		正常の場合 : 変換された値 正のオーバーフローの場合 : LONG_MAX (2,147,483,647) 負のオーバーフローの場合 : LONG_MIN (-2,147,483,648) 不正文字列の場合 : 0

【説明】

atoi

- *nptr* が指す文字列の最初の部分を int に変換します。

つまり，先頭から 0 個以上の空白文字（isspace が真となる文字）の列をスキップし，次の文字からの省略可能な符号と引き続く 10 進数字の列（10 進数字以外か終端のヌル文字が現れるまで）を整数に変換します。10 進数字がない場合は，0 を返します。

オーバーフローが起こった場合は，正のときは INT_MAX（32,767），負のときは INT_MIN（-32,768）を返します。

atol

- *nptr* が指す文字列の最初の部分を long に変換します。

つまり，先頭から 0 個以上の空白文字（isspace が真となる文字）の列をスキップし，次の文字からの省略可能な符号と引き続く 10 進数字の列（10 進数字以外か終端のヌル文字が現れるまで）を整数に変換します。10 進数字がない場合は，0 を返します。

オーバーフローが起こった場合は，正のときは LONG_MAX（2,147,483,647），負のときは LONG_MIN（-2,147,483,648）を返します。

strtol，strtoul

【機能】

- strtol は，文字列を long に変換します。
- strtoul は，文字列を unsigned long に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- long int strtol (const char **nptr* , char ***endptr* , int *base*) ;
- unsigned long int strtoul (const char **nptr* , char ***endptr* , int *base*) ;

関数名	引数	返り値
strtol	*nptr* ： 変換する文字列 *endptr* ： 認識不可能部へのポインタを格納するポインタ *base* ： 指定する基数	正常の場合 ： 変換した値 正のオーバーフローの場合 ： LONG_MAX (2,147,483,647) 負のオーバーフローの場合 ： LONG_MIN (-2,147,483,648) 変換が行われない場合 ： 0
strtoul		正常の場合 ： 変換した値 オーバーフローの場合 ： ULONG_MAX (4,294,967,295U) 変換が行われない場合 ： 0

【説明】

strtol

- *nptr* が指す文字列を次の 3 部分に分解します。

  (1) 空であってもよい空白文字列（isspace で指定される）

  (2) *base* の値により決定される基数による整数表現

  (3) 1 文字以上の認識できない文字（終端のヌル文字を含む）の列

  備考 (2) の文字列を整数に変換し，その結果を返します。

- *base* が 0 ならば，c の数値表現と解釈します（数値は，0x ～，または 0X ～（16 進数），0 ～（8 進数），0 以外の数字～（10 進数）で符号が前にあってもよい）。
- *base* が 2 ～ 36 のときは，それを基数とします（符号が前にあってもよい）。
  a（A）から z（Z）は 10 から 35 までを表します。
  base が 16 のときは，（あれば）符号の次に 0x，または 0X がついてもかまいません。
- （*endptr* が NULL ポインタでなければ）(3) の文字列へのポインタを endptr が指すオブジェクトへ格納します。
- オーバーフローの場合，正は LONG_MAX（2,147,483,647），負は LONG_MIN（-2,47,483,648）を返し，errno に ERANGE (2) を入れます。
- (2) の文字列が空，あるいは期待する型式に反する場合，変換は行わず（*endptr* が NULL ポインタでなければ）*endptr* が指すオブジェクトに文字列へのポインタを格納し，0 を返します。*base* が 0，2 ～ 36 以外の場合も同様です。

strtoul

- *nptr* が指す文字列を次の 3 部分に分解します。

  (1) 空であってもよい空白文字列（isspace で指定される）

  (2) *base* の値により決定される基数による整数表現

  (3) 1 文字以上の認識できない文字（終端のヌル文字を含む）の列

  備考 (2) の文字列を無符号整数に変換し，その結果を返します。

- *base* が 0 ならば C の数値表現（0x ～，または 0X ～（16 進数），0 ～（8 進数），0 以外の数字～（10 進数））と解釈します。
- *base* が 2 ～ 36 のときは，それを基数とします。a（A）から z（Z）は 10 から 35 までを表します。base が 16 のときは，0x，または 0X がついてもかまいません。
- （*endptr* が NULL ポインタでなければ）(3) の文字列へのポインタを endptr が指すオブジェクトへ格納します。
- オーバーフローの場合，ULONG_MAX（4,294,967,295U）を返し，errno に ERANGE (2) を入れます。
- (2) の文字列が空，あるいは期待する型式に反する場合，変換は行わず（*endptr* が NULL ポインタでなければ）*endptr* が指すオブジェクトに文字列へのポインタを格納し，0 を返します。*base* が 0，2 ～ 36 以外の場合も同様です。

calloc

【機能】

- calloc は，配列の領域を割り付けて 0 で初期化します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void * calloc (size_t *nmemb* , size_t *size*) ;

関数名	引数	返り値
calloc	*nmemb* ： 配列の個数 *size* ： 配列のサイズ	割り付けられる場合 ： 割り付けられた領域の先頭へのポインタ 割り付けられない場合 ： NULL ポインタ

【説明】

- *size* バイトの配列 *nmemb* 個分の領域を割り付け，その領域を 0 で初期化します。
- 割り付けられた領域の先頭へのポインタを返します。
- 割り付けられない場合には，NULL ポインタを返します。
- 割り付けは，ブレーク値から割り付け，割り付けられた領域の次のアドレスを新たなブレーク値とします。新たなブレーク値が奇数の場合には，偶数に補正します。ブレーク値は，brk で設定します。brk については「brk， sbrk」を参照してください。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，calloc_n/calloc_f 関数は存在しません。

free

【機能】

- 割り付けられているブロックを解放します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void free (void **ptr*) ;

関数名	引数	返り値
free	*ptr* ： 解放するブロックの先頭へのポインタ	なし

【説明】

- *ptr* が指す領域からの割り付け済みの領域（ブレーク値の前まで）を解放します（free の後で呼ばれる malloc，calloc，realloc は，*ptr* からの領域を割り付けます）。
- *ptr* が割り付け済みの領域を指していなければ何もしません（解放は，*ptr* を新たなブレーク値とすることで行います）。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，free_n/free_f 関数は存在しません。

malloc

【機能】

- malloc は，ブロックを割り付けます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void *malloc (size_t *size*) ;

関数名	引数	返り値
malloc	*size* ： 割り付けるブロックの大きさ	割り付けられる場合 ： 割り付けられた領域の先頭へのポインタ 割り付けられない場合 ： NULL ポインタ

【説明】

- *size* バイト分の領域を割り付け，割り付けられた領域の先頭へのポインタを返します。
- 割り付けられない場合は，NULL ポインタを返します。
- 割り付けは，ブレーク値から割り付け，割り付けられた領域の次のアドレスを新たなブレーク値とします。新たなブレーク値が奇数の場合には，偶数に補正します。ブレーク値は，brk で設定します。brk については「 brk， sbrk」を参照してください。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，malloc_n/malloc_f 関数は存在しません。

realloc

【機能】

- realloc は，ブロックの再割り付けを行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void * realloc (void **ptr* , size_t *size*) ;

関数名	引数	返り値
realloc	*ptr* ： 再割り付けされるブロックの先頭へのポインタ *size* ： 再割り付けするブロックの大きさ	再割り付けされる場合 ： 再割り付けした領域の先頭へのポインタ *ptr* が NULL ポインタで割り付けられる場合 ： 割り付けられた領域の先頭へのポインタ 再割り付け，割り付けできない場合 ： NULL ポインタ

【説明】

- *ptr* が指す領域からの割り付け済みの領域（ブレーク値の前まで）の大きさを size に変更します。再割り付けする領域と，再割り付けされる割り付け済みの領域の小さい方の大きさまでの内容は変化しません。大きさが増加する場合は，増加分の割り付けを行い，減少する場合は減少分を解放します。
- *ptr* が NULL ポインタの場合は，*size* 分の領域を新たに割り付けます（malloc と同じ）。
- *ptr* が割り付け済みの領域を指していない場合，または割り付けられない場合は，何もせずに NULL ポインタを返します。
- 再割り付けは，*ptr* に size バイトを加えたアドレスを新たなブレーク値として行います。新たなブレーク値が奇数の場合には，偶数に補正します。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，realloc_n/realloc_f 関数は存在しません。

abort

【機能】

- abort は，プログラムを異常終了させます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void abort (void) ;

関数名	引数	返り値
abort	なし	戻りません。

【説明】

- ループして戻りません。
- ユーザは abort の処理を作成します。

atexit，exit

【機能】

- atexit は，正常終了時に呼び出される関数を登録します。
- exit は，プログラムを終了させます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int atexit (void (**func*) (void)) ;
- void exit (int *status*) ;

関数名	引数	返り値
atexit	*func* ： 登録する関数へのポインタ	関数の登録が成功した場合 ： 0 関数が登録できない場合 ： 1
exit	*status* ： 終了状態を示す値	戻りません。

【説明】

atexit

- atexit は，プログラムの正常終了時に *func* の指す関数が引数なしで呼ばれるように登録します。
- 関数は 32 個まで登録することができます。登録することができた場合は，0 を返します。登録されている関数が 32 個あり，これ以上登録することができない場合は，登録せずに 1 を返します。

exit

- exit は，プログラムを正常終了させます。
- 最初に atexit で登録した関数を登録と逆の順に呼びます。
- 内容はループになっており，exit 関数からは戻りません。
- ユーザは，exit の処理を作成します。

abs，labs

【機能】

- abs は，int 型の値の絶対値を求めます。
- labs は，long 型の値の絶対値を求めます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int abs (int *j*) ;
- long int labs (long int *j*) ;

関数名	引数	返り値
abs	*j* ： 　絶対値を求める値	-32,767 ≦ *j* ≦ 32,767 の場合 ： 　*j* の絶対値 *j* が -32,768 の場合 ： 　-32,768（0x8000）
labs		-2,147,483,647 ≦ *j* ≦ 2,147,483,647 の場合 ： 　*j* の絶対値 *j* が -2,147,483,648 の場合 ： 　-2147483,648（0x80000000）

【説明】

abs

- abs は，*j* の値（int 型）の絶対値を求めます。
- *j* が -32768 の場合は，-32,768 を返します。

labs

- labs は，*j* の値（long 型）の絶対値を求めます。
- *j* が -2,147,483,648 の場合は，-2,147,483,648 を返します。

div，ldiv

【機能】

- div は，int 型の除算を行い，商と剰余を求めます。
- ldiv は，long 型の除算を行い，商と剰余を求めます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- div_t div (int *numer* , int *denom*) ;
- ldiv_t ldiv (long int *numer* , long int *denom*) ;

関数名	引数	返り値
div	*numer* ： 被除数 *denom* ： 除数	div_t 型のメンバ quot に商，rem に剰余を返します。
ldiv		ldiv_t 型のメンバ quot に商，rem に剰余を返します。

【説明】

div

- div は，*numer* を *denom* で割った商と剰余を求めます。
- 商の絶対値は，*numer* の絶対値を *denom* の絶対値で割った値以下の最大の整数です。符号は数学と同じです（*numer* と *denom* が同符号の場合は正，異符号の場合は負）。
- 剰余は，*numer* - *denom* * 商の値です。
- *denom* が 0 の場合，商は 0，剰余は *numer* です。
- *numer* が -32,768，*denom* が -1 の場合，商は -32,768，剰余は 0 です。

ldiv

- ldiv は，*numer* を *denom* で割った商と剰余を求めます。
- 商の絶対値は，*numer* の絶対値を *denom* の絶対値で割った値以下の最大の（long int 型）整数です。符号は数学と同じです（*numer* と *denom* が同符号の場合は正，異符号の場合は負）。
- 剰余は，*numer* - *denom* * 商の値です。
- *denom* が 0 の場合，商は 0，剰余は *numer* です。
- *numer* が -2,147,483,648，*denom* が -1 の場合は，商は -2,147,483,648，剰余は 0 です。

brk，sbrk

【機能】

- brk は，ブレーク値をセットします。
- sbrk は，ブレーク値を増減します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int brk (char **endds*) ;
- char *sbrk (int *incr*) ;

関数名	引数	返り値
brk	*endds* : 設定するブレーク値	正常の場合 : 0 ブレーク値を変更できない場合 : -1
sbrk	*incr* : ブレーク値を増減する量	正常の場合 : 旧ブレーク値 ブレーク値が増減できない場合 : -1

【説明】

brk

- brk は，*endds* で与えられた値をブレーク値に設定します。
- *endds* が許容範囲外の場合は，ブレーク値を変更せず，errno に ENOMEM (3) をセットし，-1 を返します。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，brk_n/brk_f 関数は存在しません。

sbrk

- sbrk は，ブレーク値を *incr* バイト増減（*incr* の符号による）します。
- *incr* に指定された増減量が奇数の場合には，偶数に補正します。
- 増減したあとのブレーク値が許容範囲外になる場合は，ブレーク値を変更せず，errno に ENOMEM (3) をセットし，-1 を返します。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，sbrk_n/sbrk_f 関数は存在しません。

atof，strtod

【機能】

- atof は，10 進整数文字列を double に変換します。
- strtod は，文字列を double に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- double atof (const char **nptr*) ;
- double strtod (const char **nptr* , char ***endptr*) ;

関数名	引数	返り値
atof	*nptr* ： 変換する文字列	正常の場合 ： 変換された値 正のオーバーフローの場合 ： HUGE_VAL（オーバーフローした値の符号を持つ） 負のオーバーフローの場合 ： 0 不正文字列の場合 ： 0
strtod	*nptr* ： 変換する文字列 *endptr* ： 認識不可能部へのポインタを格納するポインタ	正常の場合 ： 変換された値 正のオーバーフローの場合 ： HUGE_VAL（オーバーフローした値の符号を持つ） 負のオーバーフローの場合 ： 0 不正文字列の場合 ： 0

【説明】

atof

- *nptr* が指す文字列を double に変換します。
  つまり，先頭から 0 個以上の空白文字（isspace が真となる文字）の列をスキップし，次の文字からの文字列（10 進数字以外か終端のヌル文字が現れるまで）を浮動小数点数に変換します。
- 変換が正常に行われると，浮動小数点数を返します。
- 変換でオーバーフローが生じた場合には，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- アンダーフロー，またはオーバーフローによる有効桁数の消滅が生じた場合には，それぞれ非正規化数，± 0 を返し，errno に ERANGE をセットします。
- 変換が行えない場合には，0 を返します。

strtod

- *nptr* が指す文字列を double に変換します。
  つまり，先頭から 0 個以上の空白文字（isspace が真となる文字）の列をスキップし，次の文字からの文字列（10 進数字以外か終端のヌル文字が現れるまで）を浮動小数点数に変換します。
  この書式を満たさない先頭文字が現れた場合には，スキャンを終了し，*endptr* が空ポインタでなければ，空白を含む書式先頭のポインタを *endptr* に格納します。
- 変換が正常に行われると，浮動小数点数を返します。
- 変換でオーバーフローが生じた場合には，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- アンダーフロー，またはオーバーフローによる有効桁数の消滅が生じた場合には，それぞれ非正規化数，± 0 を返し，errno に ERANGE をセットします。またこのとき *endptr* は，次の文字列へのポインタを格納します。
- 変換が行えない場合には，0 を返します。

itoa，ltoa，ultoa

【機能】

- itoa は，int を文字列に変換します。
- ltoa は，long を文字列に変換します。
- ultoa は，unsigned long を文字列に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- char *itoa (int *value* , char **string* , int *radix*) ;
- char *ltoa (long *value* , char **string* , int *radix*) ;
- char *ultoa (unsigned long *value* , char **string* , int *radix*) ;

関数名	引数	返り値
itoa, ltoa, ultoa	*value* ： 変換する数値 *string* ： 変換結果へのポインタ *radix* ： 指定する基数	正常な場合 ： 変換した文字列へのポインタ それ以外の場合 ： NULL ポインタ

【説明】

- 指定した数値 *value* をヌル文字で終了する文字列に変換し，結果を *string* で指される領域に格納します。変換は，指定された基数 *radix* で行い，変換した文字列へのポインタを返します。
- *radix* は 2 〜 36 の範囲でなければなりません。それ以外の場合には，変換を行わず，NULL ポインタを返します。

rand，srand

【機能】

- rand は，疑似乱数を発生させます。
- srand は，疑似乱数の発生状態の初期化を行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int rand (void) ;
- void srand (unsigned int *seed*) ;

関数名	引数	返り値
rand	なし	0 から RAND_MAX の範囲の疑似乱数
srand	*seed* ： 疑似乱数の発生状態の初期値	なし

【説明】

rand

- rand は，0 から RAND_MAX の範囲の疑似乱数を発生させます。

srand

- srand は，疑似乱数の発生状態の初期化を行います。rand 関数が呼ばれたときの返り値である疑似乱数列の基となる値として *seed* を使います。*seed* の値が同じであれば，再び srand 関数が呼ばれても，疑似乱数の列は変わりません。
- srand 関数をコールせずに rand 関数をコールすることは，*seed* = 1 で srand 関数をコールしたあとに rand 関数をコールするのと同じです。

bsearch

【機能】

- bsearch は，バイナリ・サーチを行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void *bsearch (const void **key* , const void **base* , size_t *nmemb* , size_t *size* ,
int (**compare*) (const void *, const void *)) ;

関数名	引数	返り値
bsearch	*key* ： サーチする値へのポインタ *base* ： サーチする配列へのポインタ *nmemb* ： 配列要素の数 *size* ： 配列の 1 要素のサイズ *compare* ： *key* と配列の要素を比較し，その関係を返す関数	マッチする配列要素がある場合 ： 最初にマッチした配列要素へのポインタ マッチする配列要素がない場合 ： NULL ポインタ

【説明】

- ポインタ *base* の指す配列から *key* の指すものをバイナリ・サーチします。ポインタ *base* の指す配列は，*size* の大きさの *nmemb* 個の昇順にソートされた配列です。
- *compare* 関数は *key* によって指されるものと配列要素を比較し，その関係を次の値により返します。*compare* 関数の第 1 引数は *key*，第 2 引数は配列要素です。

0 より小さい ： *key* によって指されるものの方が小さい
0 ： 両者は等しい
0 より大きい ： *key* によって指されるものの方が大きい

qsort

【機能】

- qsort は，クイック・ソートを行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- void qsort (void **base* , size_t *nmemb* , size_t *size* , int (**compare*) (const void * , const void *)) ;

関数名	引数	返り値
qsort	*base* : ソートする配列へのポインタ *nmemb* : 配列要素の数 *size* : 配列の 1 要素のサイズ *compare* : 配列の 2 つの要素を比較し，その関係を 返す関数	なし

【説明】

- ポインタ *base* の指す配列を昇順になるようにクイック・ソートします。
  ポインタ *base* の指す配列は，*size* の大きさの *nmemb* 個の配列です。
- *compare* 関数は，2 つの配列要素（配列要素 1 と 2）を比較し，その関係を次の値により返します。
- *compare* 関数の第 1 引数は配列要素 1，第 2 引数は配列要素 2 です。

0 より小さい	:	配列要素 1 の方が小さい
0	:	両者は等しい
0 より大きい	:	配列要素 1 の方が大きい

- 等しい配列要素であった場合には，配列の先頭に近い方にあったものが先になります。

strbrk

【機能】

- ブレーク値をセットします。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- int strbrk (char **endds*) ;

関数名	引数	返り値
strbrk	*endds* : 設定するブレーク値	正常な場合 : 0 ブレーク値を変更できない場合 : -1

【説明】

- *endds* で与える値をブレーク値（割り当てられる領域の終わりのアドレスの次のアドレス）に設定します。
- *endds* が許容範囲外の場合はブレーク値を変更せず，errno に ENOMEM (3) をセットし -1 を返します。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。
  したがって，strbrk_n/strbrk_f 関数は存在しません。

strsbrk

【機能】

- ブレーク値を増減します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- char *strsbrk (int *incr*) ;

関数名	引数	返り値
strsbrk	*incr* ： 　ブレーク値を増減する量	正常な場合 ： 　旧ブレーク値 ブレーク値が増減できない場合 ： 　-1

【説明】

- ブレーク値を *incr* バイト増減（*incr* の符号によります）します。
- 増減した後のブレーク値が許容範囲外になる場合は，ブレーク値を変更せず errno に ENOMEM (3) をセットし -1 を返します。
- CC78K0R では，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，strsbrk_n/strsbrk_f 関数は存在しません。

stritoa，strltoa，strultoa

【機能】

- stritoa は，int を文字列に変換します。
- strltoa は，long を文字列に変換します。
- strultoa は，unsigned long を文字列に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- char *stritoa (int *value* , char **string* , int *radix*) ;
- char *strltoa (long *value* , char **string* , int *radix*) ;
- char *strultoa (unsigned long *value* , char **string* , int *radix*) ;

関数名	引数	返り値
stritoa, strltoa, strultoa	*value* ： 変換する文字列 *string* ： 変換結果へのポインタ *radix* ： 指定する基数	正常な場合 ： 変換した文字列へのポインタ それ以外の場合 ： NULL ポインタ

【説明】

- 指定した数値 value をヌル文字で終了する文字列に変換し，結果を *string* で指される領域に格納します。変換は，指定された基数 *radix* で行い，変換した文字列へのポインタを返します。
- *radix* は 2 ～ 36 の範囲でなければなりません。それ以外の場合には，変換を行わず，NULL ポインタを返します。

10.10 文字列／メモリ関数

文字列／メモリ関数には，次のものがあります。

- memcpy,　memmove
- strcpy,　strncpy
- strcat,　strncat
- memcmp
- strcmp,　strncmp
- memchr
- strchr,　strrchr
- strspn,　strcspn
- strpbrk
- strstr
- strtok
- memset
- strerror
- strlen
- strcoll
- strxfrm

memcpy，memmove

【機能】

- memcpy は，バッファを指定文字数分コピーします。
- memmove は，バッファを指定文字数分コピーします（バッファが重なっても正常に動作します）。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- void *memcpy (void **s1*, const void **s2* , size_t *n*) ;
- void *memmove (void **s1* , const void **s2* , size_t *n*) ;

関数名	引数	返り値
memcpy, memmove	*s1* ： コピー先のオブジェクトへのポインタ *s2* ： コピー元のオブジェクトへのポインタ *n* ： 指定文字数	*s1* の値

【説明】

memcpy

- memcpy は，*s2* が指すオブジェクトの n 文字を *s1* が指すオブジェクトへコピーします。
- *s2* ＜ *s1* ＜ *s2* + *n* の場合，正常動作は保証されません（先頭から順にコピーするため）。

memmove

- memmove は，*s2* が指すオブジェクトの n 文字を *s1* が指すオブジェクトへコピーします。
- *s1* と *s2* の指すオブジェクトが重なった場合も正常に動作します。

strcpy，strncpy

【機能】

- strcpy は，文字列をコピーします。
- strncpy は，文字列の先頭から指定の文字数分コピーします。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strcpy (char **s1* , const char **s2*) ;
- char *strncpy (char **s1* , const char **s2* , size_t *n*) ;

関数名	引数	返り値
strcpy	*s1* ： コピー先文字列へのポインタ *s2* ： コピー元文字列へのポインタ	*s1* の値
strncpy	*s1* ： コピー先文字列へのポインタ *s2* ： コピー元文字列へのポインタ *n* ： コピーする文字数	*s1* の値

【説明】

strcpy

- strcpy は，*s2* が指す文字列（終端のヌル文字を含みます）を *s1* が指す文字列へコピーします。
- *s2* ＜ *s1* ≦（*s2* + コピーする文字列の長さ）の場合，正常動作は保証されません（先頭から順にコピーするため）。

strncpy

- strncpy は，*s2* が指す文字列の n 文字以内を *s1* が指す配列へコピーします。
- *s2* ＜ *s1* ≦（*s2* + コピーする文字列の長さ，または *s2* + *n* - 1 の最小値）の場合，正常動作は保証されません（先頭から順にコピーするため）。
- *s2* が指す文字列が *n* 文字未満の場合，終端の NULL 文字までをコピーし，*n* 文字に達するまで NULL 文字を付加します。

strcat，strncat

【機能】

- strcat は，文字列に文字列を追加します。
- strncat は，文字列に指定文字数分の文字列を追加します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strcat (char **s1* , const char **s2*) ;
- char *strncat (char **s1* , const char **s2* , size_t *n*) ;

関数名	引数	返り値
strcat	*s1* ： 追加される文字列へのポインタ *s2* ： 追加する文字列へのポインタ	*s1* の値
strncat	*s1* ： 追加される文字列へのポインタ *s2* ： 追加する文字列へのポインタ *n* ： 追加する文字数	*s1* の値

【説明】

strcat

- strcat は，*s1* が指す文字列の終わりに *s2* が指す文字列（終端のヌル文字を含みます）をコピーして追加します。*s2* の初めの文字を *s1* の終端のヌル文字に上書きします。
- 重なり合うオブジェクト間で複写を行う場合，その動作は保証されません。

strncat

- strncat は，*s1* が指す文字列の終わりに，*s2* が指す文字列（終端のヌル文字を含みません）のうち *n* 文字分を追加します。*s2* の初めの文字を *s1* の終端の文字に上書きします。
- *s2* が指す文字列が *n* 文字未満の場合には，終端のヌル文字までを追加します。*n* 文字以上の場合には，先頭から *n* 文字分追加します。
- 終端のヌル文字は必ず追加します。
- 重なり合うオブジェクト間で複写を行う場合，その動作は保証されません。

memcmp

【機能】

- memcmp は，2 つのバッファの指定文字数分を比較します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- int memcmp (const void *_s1_ , const void *_s2_ , size_t _n_) ;

関数名	引数	返り値
memcmp	_s1_ : 比較するオブジェクトへのポインタ _s2_ : 比較するオブジェクトへのポインタ _n_ : 比較する文字数	_s1_ と _s2_ が _n_ 文字分等しい場合 : 0 _s1_ と _s2_ が _n_ 文字以内で異なる場合 : 最初の異なる文字を int に変換した値の差（_s1_ の文字 - _s2_ の文字）

【説明】

- _s1_ の指すオブジェクトと _s2_ の指すオブジェクトを n 文字分比較します。
- _s1_ と _s2_ が _n_ 文字分等しい場合，0 を返します。
- _s1_ と _s2_ が _n_ 文字以内で異なる場合，最初の異なる文字を int に変換した値の差（_s1_ の文字 - _s2_ の文字）を返します。

strcmp，strncmp

【機能】

- strcmp は，2 つの文字列を比較します。
- strncmp は，2 つの文字列の指定文字数分を比較します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- int strcmp (const char **s1* , const char **s2*) ;
- int strncmp (const char **s1* , const char **s2* , size_t *n*) ;

関数名	引数	返り値
strcmp	*s1* ： 比較文字列へのポインタ *s2* ： 比較文字列へのポインタ	文字列 *s1* と文字列 *s2* が等しい場合 ： 0 文字列 *s1* と文字列 *s2* が異なる場合 ： 最初の異なる文字を int に変換した値の差（*s1* の文字 - *s2* の文字）
strncmp	*s1* ： 比較文字列へのポインタ *s2* ： 比較文字列へのポインタ *n* ： 比較する文字数	文字列 *s1* と文字列 *s2* が *n* 文字分等しい場合 ： 0 文字列 *s1* と文字列 *s2* が *n* 文字分異なる場合 ： 最初の異なる文字を int に変換した値の差（*s1* の文字 - *s2* の文字）

【説明】

strcmp

- strcmp は，*s1* の指す文字列と *s2* の指す文字列を比較します。
- 文字列 *s1* と *s2* が等しい場合，0 を返します。文字列 *s1* と *s2* が異なる場合には，最初の異なる文字を int に変換した値の差（*s1* の文字 - *s2* の文字）を返します。

strncmp

- strncmp は，*s1* の指す文字列と *s2* の指す文字列の *n* 文字分を比較します。
- 文字列 *s1* と *s2* が *n* 文字以内で等しい場合，0 を返します。文字列 *s1* と *s2* が *n* 文字以内で異なる場合には，最初の異なる文字を int に変換した値の差（*s1* の文字 - *s2* の文字）を返します。

memchr

【機能】

- memchr は，指定文字数分のバッファから指定文字を探します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

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

関数名	引数	返り値
memchr	*s* ： 検索されるオブジェクトへのポインタ *c* ： 指定文字 *n* ： 検索するオブジェクトの文字数	文字 *c* がある場合 ： 最初に出現した文字 *c* へのポインタ 文字 *c* がない場合 ： NULL ポインタ

【説明】

- *s* が指すオブジェクトの先頭から *n* 文字以内で最初に出現する（unsigned char に変換した）*c* の位置へのポインタを返します。
- 出現しない場合は，NULL ポインタを返します。

strchr，strrchr

【機能】

- strchr は，文字列中から指定された文字を探し，最初の出現位置を返します。
- strrchr は，文字列中から指定された文字を探し，最後の出現位置を返します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strchr (const char **s* , int *c*) ;
- char *strrchr (const char **s* , int *c*) ;

関数名	引数	返り値
strchr, strrchr	*s* ： 検索される文字列へのポインタ *c* ： 指定文字	文字列 *s* 中に文字 *c* がある場合 ： 文字列 *s* 中に最初／最後に出現した 文字 *c* を指すポインタ 文字列 *s* 中に文字 *c* がない場合 ： NULL ポインタ

【説明】

trchr

- strchr は，*s* が指す文字列中の（char 型へ変換した）*c* の最初の出現位置を求め，そのポインタを返します。
- 終端のヌル文字は，文字列の一部とみなします。
- 文字列 *s* 中に文字 *c* がない場合は，NULL ポインタを返します。

strrchr

- strrchr は，*s* が指す文字列中の（char 型へ変換した）*c* の最後の出現位置を求め，そのポインタを返します。
- 終端のヌル文字は，文字列の一部とみなします。
- 文字列 *s* 中に文字 *c* がない場合は，NULL ポインタを返します。

strspn，strcspn

【機能】

- strspn は，検索される文字列の中で指定文字列中に含まれる文字だけで構成されている部分の先頭からの長さを求めます。
- strcspn は，検索される文字列の中で指定文字列中に含まれる文字以外で構成されている部分の先頭からの長さを求めます。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- size_t strspn (const char **s1* , const char **s2*) ;
- size_t strcspn (const char **s1* , const char **s2*) ;

関数名	引数	返り値
strspn	*s1* ： 検索される文字列へのポインタ *s2* ： 指定文字列を示す文字列へのポインタ	文字列 *s1* 中の *s2* で指定される文字で構成される部分の長さ
strcspn		文字列 *s1* 中の *s2* で指定される文字以外で構成される部分の長さ

【説明】

strspn

- strspn は，*s1* が指す文字列中で *s2* が指す文字列中に含まれる，文字だけで構成されている部分の長さを返します。
- *s2* の終端のヌル文字は *s2* の一部とはみなしません。

strcspn

- strcspn は，*s1* が指す文字列中で *s2* が指す文字列中に含まれる，文字以外で構成されている部分の長さを返します。
- *s2* の終端のヌル文字は *s2* の一部とはみなしません。

strpbrk

【機能】

- strpbrk は，指定された文字列のどれかの文字が，検索される文字列中で最初に現れる位置を求めます。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strpbrk (const char **s1* , const char **s2*) ;

関数名	引数	返り値
strpbrk	*s1* ： 検索される文字列へのポインタ *s2* ： 指定文字を示す文字列へのポインタ	文字列 *s1* 中に文字列 *s2* 内のどれかの文字がある場合 ： 文字列 *s2* 内のどれかの文字が文字列 *s1* 中で最初に現れる文字へのポインタ 文字列 *s1* 中に文字列 *s2* 内の文字がない場合 ： NULL ポインタ

【説明】

- *s2* が指す文字列内のどれかの文字が *s1* が指す文字列中で最初に現れる位置を求め，そのポインタを返します。
- 文字列 *s1* 中に文字列 *s2* 内の文字がない場合，NULL ポインタを返します。

strstr

【機能】

- strstr は，指定文字列が，検索される文字列中に最初に現れる位置を求めます。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strstr (const char **s1* , const char **s2*) ;

関数名	引数	返り値
strstr	*s1* ： 検索される文字列へのポインタ *s2* ： 指定文字列へのポインタ	文字列 *s1* 中に文字列 *s2* がある場合 ： 文字列 *s2* が文字列 *s1* 中で最初に現れる位置の先頭へのポインタ 文字列 *s1* 中に文字列 *s2* がない場合 ： NULL ポインタ *s2* が空文字列の場合 ： *s1* の値

【説明】

- *s1* が指す文字列中で *s2* が指す文字列（終端のヌル文字を除く）と全文字が一致する最初の位置の先頭へのポインタを返します。
- 文字列 *s1* 中に文字列 *s2* がない場合，NULL ポインタを返します。
- *s2* が空文字列を指す場合，*s1* の値を返します。

strtok

【機能】

- 文字列を区切り文字以外からなる文字列に分解する。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strtok (char **s1* , const char **s2*) ;

関数名	引数	返り値
strtok	*s1* ： 分解される文字列へのポインタ，または NULL ポインタ *s2* ： 字句の区切り文字を示す文字列へのポインタ	字句がある場合 ： 字句の第 1 文字へのポインタ 字句がない場合 ： NULL ポインタ

【説明】

- 字句とは，指定される文字列中の区切り文字以外の文字からなる文字列です。
- *s1* が NULL ポインタの場合は，前回の strtok の呼び出しでの保存ポインタが指す文字列を分解される文字列とします。ただし，保存ポインタが NULL ポインタの場合は何もせずに NULL ポインタを返します。
- *s1* が NULL ポインタでない場合は，*s1* が指す文字列を分解される文字列とします。
- *s2* が指す文字列に含まれない文字を分解される文字列から探し，見つからなければ保存ポインタを NULL ポインタにして，NULL ポインタを返します。見つかれば，その文字を字句の第 1 文字とします。
- 字句の第 1 文字が見つかった場合，文字列 *s2* に含まれる文字を字句の第 1 文字以降から探します。見つからなければ，保存ポインタを NULL ポインタにします。見つかれば，その文字の位置にヌル文字を上書きし，その次の文字へのポインタを保存ポインタにします。
- 字句の第 1 文字へのポインタを返します。

memset

【機能】

- memset は，バッファの指定文字数分を指定文字で初期化します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- void *memset (void **s* , int *c* , size_t *n*) ;

関数名	引数	返り値
memset	*s* : 初期化するオブジェクトへのポインタ *c* : 指定文字 *n* : 指定文字数	*s* の値

【説明】

- *s* が指すオブジェクトの先頭から *n* 文字分に（unsigned char 型に変換された）*c* の値をコピーします。

strerror

【機能】

- strerror は，指定されたエラー番号に対応するエラー・メッセージの文字列を格納する領域へのポインタを返します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- char *strerror (int *errnum*) ;

関数名	引数	返り値
strerror	*errnum* ： エラー番号	エラー番号に対応するエラーがある場合 ： エラー・メッセージの文字列へのポインタ エラー番号に対応するエラーがない場合 ： NULL ポインタ

【説明】

- *errnum* の値に対応して，次の値を返します。

errnum の値	返り値
0	文字列“Error 0”へのポインタ
1（EDOM）	文字列“Argument too large”へのポインタ
2（ERANGE）	文字列“Result too large”へのポインタ
3（ENOMEM）	文字列“Not enough memory”へのポインタ
その他	NULL ポインタ

strlen

【機能】

- 文字列の長さを求めます。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- size_t strlen (const char **s*) ;

関数名	引数	返り値
strlen	*s* : 文字列へのポインタ	文字列 *s* の長さ

【説明】

- *s* が指す文字列の文字数を返します。

  文字数は，文字列の先頭から終端を示すヌル文字の前までの文字数です。

strcoll

【機能】

- 地域特有の情報に基づいて 2 つの文字列を比較します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- int strcoll (const char **s1* , const char **s2*) ;

関数名	引数	返り値
strcoll	*s1* ： 比較文字列へのポインタ *s2* ： 比較文字列へのポインタ	文字列 *s1* と文字列 *s2* が等しい場合 ： 0 文字列 *s1* と文字列 *s2* が異なる場合 ： 最初の異なる文字を int に変換した値の差（*s1* の文字 - *s2* の文字）

【説明】

- CC78K0R は，文化圏固有操作はサポートしていません。
  strcmp と同じ動作をします。

strxfrm

【機能】

- 地域特有の情報に基づいて文字列を変換します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- size_t　strxfrm (char　**s1* , const char　**s2* , size_t　*n*) ;

関数名	引数	返り値
strxfrm	*s1* ： 比較文字列へのポインタ *s2* ： 比較文字列へのポインタ *n* ： *s1* に入る最大文字数	変換した結果の文字列（終端を示す文字列を含みません）の長さを返します。返却された値が n 以上の場合，*s1* で示される配列の内容は不定とします。

【説明】

- CC78K0R は，文化圏固有操作はサポートしていません。

次の関数と同じ動作をします。

strncpy (*s1* , *s2* , *c*) ;

return (strlen (*s2*)) ;

10.11 数学関数

数学関数には，次のものがあります。

- acos
- asin
- atan
- atan2
- cos
- sin
- tan
- cosh
- sinh
- tanh
- exp
- frexp
- ldexp
- log
- log10
- modf
- pow
- sqrt
- ceil
- fabs
- floor
- fmod
- matherr
- acosf
- asinf
- atanf
- atan2f
- cosf
- sinf
- tanf
- coshf
- sinhf
- tanhf
- expf
- frexpf

- ldexpf
- logf
- log10f
- modff
- powf
- sqrtf
- ceilf
- fabsf
- floorf
- fmodf

acos

【機能】

- acos を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double acos (double *x*) ;

関数名	引数	返り値
acos	*x* : 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 : *x* の acos *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 : NaN

【説明】

- *x* の acos（0 から π の範囲内）を計算します。
- *x* ＜ -1，1 ＜ *x* の定義域エラーの場合は，NaN を返し errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。

asin

【機能】

- asin を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double asin (double *x*) ;

関数名	引数	返り値
asin	*x* : 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 : *x* の asin *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 : NaN *x* = -0 の場合 : -0 アンダーフロー時 : 非正規化数

【説明】

- *x* の asin（- π/2 から + π/2 の範囲内）を計算します。
- *x* ＜ -1，1 ＜ *x* の領域エラーの場合は，NaN を返し errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が -0 の場合は，-0 を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

atan

【機能】

- atan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double atan (double *x*) ;

関数名	引数	返り値
atan	*x* : 演算を行う数値	正常時 : *x* の atan *x* が非数の場合 : NaN *x* = -0 の場合 : -0 アンダーフロー時 : 非正規化数

【説明】

- *x* の atan（- π/2 から + π/2 の範囲内）を計算します。

- *x* が非数の場合は，NaN を返します。

- *x* が -0 の場合は，-0 を返します。

- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

atan2

【機能】

- y/x の atan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double atan2 (double *y* , double *x*) ;

関数名	引数	返り値
atan2	*x* : 演算を行う数値 *y* : 演算を行う数値	正常時 : *y*/*x* の atan *x* と y がともに 0 か，y/x が表現できない値の場合，*x*，*y* がともに±∞の場合，*x*，*y* のどちらかが非数の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *y*/*x* の atan（- π から + π の範囲内）を計算します。
- *x* と y がともに 0 か，*y*/*x* が表現できない値の場合，あるいは，*x*，*y* がともに無限大の場合には，NaN を返し errno に EDOM をセットします。
- *x*，*y* のどちらかが非数の場合は，NaN を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

cos

【機能】

- cos を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double cos (double *x*) ;

関数名	引数	返り値
cos	*x* : 演算を行う数値	正常時 : *x* の cos *x* が非数，*x* が無限大の場合 : NaN

【説明】

- *x* の cos を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

sin

【機能】

- sin を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double sin (double *x*) ;

関数名	引数	返り値
sin	*x* : 演算を行う数値	正常時 : *x* の sin *x* が非数，*x* が無限大の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *x* の sin を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

tan

【機能】

- tan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double tan (double *x*) ;

関数名	引数	返り値
tan	*x* : 演算を行う数値	正常時 : *x* の tan *x* が非数，*x* = ±∞の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *x* の tan を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

cosh

【機能】

- cosh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double cosh (double *x*) ;

関数名	引数	返り値
cosh	*x* : 演算を行う数値	正常時 : *x* の cosh *x* が非数の場合 : NaN *x* = ±∞の場合 : + ∞ オーバーフロー時 : HUGE_VAL（正の符号を持ちます）

【説明】

- *x* の cosh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，+ ∞を返します 。
- 演算の結果，オーバーフローが生じた場合は，正の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。

sinh

【機能】

- sinh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double　sinh (double *x*) ;

関数名	引数	返り値
sinh	*x* ： 演算を行う数値	正常時 ： *x* の sinh *x* が非数の場合 ： NaN *x* = ±∞の場合 ： ±∞ オーバーフロー時 ： HUGE_VAL（オーバーフローした値の符号を持ちます） アンダーフロー時 ： ± 0

【説明】

- *x* の sinh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が±∞の場合は，±∞を返します。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- 演算の結果，アンダーフローが生じた場合は，± 0 を返します。

tanh

【機能】

- tanh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double tanh (double *x*) ;

関数名	引数	返り値
tanh	*x* : 演算を行う数値	正常時 : *x* の tanh *x* が非数の場合 : NaN *x* = ±∞の場合 : ± 1 アンダーフロー時 : ± 0

【説明】

- *x* の tanh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が±∞の場合は，± 1 を返します。
- 演算の結果，アンダーフローが生じた場合は，± 0 を返します。

exp

【機能】

- 指数関数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double exp (double *x*) ;

関数名	引数	返り値
exp	*x* : 演算を行う数値	正常時 : *x* の指数関数 *x* が非数の場合 : NaN *x* = + ∞の場合 : + ∞ *x* = - ∞の場合 : +0 アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : +0 オーバーフロー時 : HUGE_VAL（正の符号を持ちます）

【説明】

- *x* の指数関数を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が + ∞の場合は，+ ∞を返します。
- *x* が - ∞の場合は，+0 を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダーフローによる有効桁数の消滅が生じた場合は，+0 を返します。
- 演算の結果，オーバーフローが生じた場合は，正の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。

frexp

【機能】

- 仮数部と指数部を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double frexp (double *x* , int **exp*) ;

関数名	引数	返り値
frexp	*x* : 演算を行う数値 *exp* : 指数部を格納するポインタ	正常時 : *x* の仮数 *x* が非数，*x* = ±∞の場合 : NaN *x* = ± 0 の場合 : ± 0

【説明】

- 浮動小数点数 *x* を，*x* = m * 2 ^ n のような仮数 m と指数 n に分け，仮数 m を返します。
- 指数 n は，ポインタ *exp* の指し示すところに格納します。ただし，m の絶対値は 0.5 以上 1.0 未満です。
- *x* が非数の場合，NaN を返し，**exp* の値は 0 とします。
- *x* が無限大の場合は，NaN を返し，**exp* の値を 0 とし，errno に EDOM をセットします。
- *x* が± 0 の場合，± 0 を返し，**exp* の値は 0 とします。

ldexp

【機能】

- x * 2 ^ exp を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double ldexp (double *x* , int *exp*) ;

関数名	引数	返り値
ldexp	*x* : 演算を行う数値 *exp* : べき乗数	正常時 : *x* * 2 ^ *exp* *x* が非数の場合 : NaN *x* = ±∞ の場合 : ±∞ *x* = ± 0 の場合 : ± 0 オーバーフロー時 : HUGE_VAL（オーバーフローした値の符号を持ちます） アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : ± 0

【説明】

- *x* * 2 ^ *exp* を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が±∞の場合は，±∞を返します。
- *x* が± 0 の場合，± 0 を返します。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダーフローによる有効桁数の消滅が生じた場合は，± 0 を返します。

log

【機能】

- 自然対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double log (double *x*) ;

関数名	引数	返り値
log	*x* : 演算を行う数値	正常時 : *x* の自然対数 *x* ＜ 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* が無限大の場合 : + ∞

【説明】

- *x* の自然対数を求めます。
- *x* ＜ 0 の領域エラーの場合は，NaN を返し，errno に EDOM をセットします。
- *x* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が + ∞の場合は，+ ∞を返します。

log10

【機能】

- 10 を底とした対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double log10 (double *x*) ;

関数名	引数	返り値
log10	*x* : 演算を行う数値	正常時 : *x* の 10 を底とした対数 *x* ＜ 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* が無限大の場合 : + ∞

【説明】

- *x* の 10 を底とした対数を求めます。
- *x* ＜ 0 の領域エラーの場合は，NaN を返し，errno に EDOM をセットします。
- *x* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が + ∞の場合は，+ ∞を返します。

modf

【機能】

- 小数部と整数部を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double modf (double *x* , double **iptr*) ;

関数名	引数	返り値
modf	*x* : 演算を行う数値 *iptr* : 整数部へのポインタ	正常時 : *x* の小数部 *x* が非数，または *x* が無限大の場合 : NaN *x* が± 0 の場合 : ± 0

【説明】

- 浮動小数点数 *x* を小数部と整数部に分けます。
- *x* と同じ符号を持つ小数部を返し，整数部はポインタ *iptr* の指し示すところに格納します。
- *x* が非数の場合は，NaN を返し，ポインタ *iptr* の指し示すところに NaN を格納します。
- *x* が無限大の場合は，NaN を返し，ポインタ *iptr* の指し示すところに NaN を格納し，errno に EDOM をセットします。
- *x* = ± 0 の場合は，ポインタ *iptr* の指し示すところに± 0 を格納します。

pow

【機能】

- x の y 乗を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double pow (double *x* , double *y*) ;

関数名	引数	返り値
pow	*x* : 演算を行う数値 *y* : 乗数	正常時 : *x* ^ *y* *x* が非数，または *y* が非数の場合， *x* = + ∞，かつ *y* = 0, *x* ＜ 0，かつ *y* ≠整数, *x* ＜ 0，かつ y = ±∞, *x* = 0，かつ *y* ≦ 0 のいずれかの場合 : NaN オーバーフロー時 : HUGE_VAL（オーバーフローした値の符号を持ちます。） アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : ± 0

【説明】

- *x* ^ *y* を計算します。
- *x* が非数，または *y* が非数の場合は，NaN を返します。
- *x* = + ∞かつ *y* = 0，*x* ＜ 0 かつ *y* ≠整数，*x* ＜ 0 かつ *y* = ±∞，*x* = 0 かつ *y* ≦ 0 のいずれかの場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- アンダーフローが生じた場合は，非正規化数を返します。
- アンダーフローによる有効桁数の消滅が生じた場合は，± 0 を返します。

sqrt

【機能】

- 平方根を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double sqrt (double *x*) ;

関数名	引数	返り値
sqrt	*x* : 演算を行う数値	*x* ≧ 0 の場合 : *x* の平方根 *x* ＜ 0 の場合 : 0 *x* が非数の場合 : NaN *x* = ± 0 の場合 : ± 0

【説明】

- *x* の平方根を計算します。
- *x* ＜ 0 の領域エラーの場合は，0 を返し，errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が± 0 の場合は，± 0 を返します。

ceil

【機能】

- x より小さくない最小の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double　ceil (double *x*) ;

関数名	引数	返り値
ceil	*x* ： 　演算を行う数値	正常時 ： 　*x* より小さくない最小の整数 *x* が非数，*x* が無限大の場合 ： 　NaN *x* = -0 の場合 ： 　+0 *x* より小さくない最小の整数を表現できない場合 ： 　*x*

【説明】

- *x* より小さくない最小の整数を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *x* より小さくない最小の整数を表現できない場合は，*x* を返します。

fabs

【機能】

- 浮動小数点数 x の絶対値を返します。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double fabs (double *x*) ;

関数名	引数	返り値
fabs	*x* : 絶対値を求める値	正常時 : *x* の絶対値 *x* が非数の場合 : NaN *x* = -0 の場合 : +0

【説明】

- *x* の絶対値を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が -0 の場合は，+0 を返します。

floor

【機能】

- x より大きくない最大の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double floor (double *x*) ;

関数名	引数	返り値
floor	*x* : 演算を行う数値	正常時 : *x* より大きくない最大の整数 *x* が非数，*x* が無限大の場合 : NaN *x* = -0 の場合 : +0 *x* より大きくない最大の整数を表現できない場合 : *x*

【説明】

- *x* より大きくない最大の整数を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *x* より大きくない最大の整数を表現できない場合は，*x* を返します。

fmod

【機能】

- x/y の余りを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- double fmod (double *x* , double *y*)

関数名	引数	返り値
fmod	*x* : 演算を行う数値 *y* : 演算を行う数値	正常時 : *x*/*y* の余り *x* が非数，または *y* が非数の場合， *y* が± 0，または *x* が±∞の場合 : NaN *x* ≠∞，かつ *y* = ±∞の場合 : *x*

【説明】

- *x* - i * *y* で表される x/y の余りを計算します。i は整数です。
- *y* ≠ 0 の場合は，返り値は *x* と同じ符号を持ち，その絶対値は *y* の絶対値より小さくなります。
- *x* が非数，または *y* が非数の場合は，NaN を返します。
- *y* が± 0，または *x* = ±∞の場合は，NaN を返し，errno に EDOM をセットします。
- *y* が無限大の場合は，*x* が無限大でなければ x を返します。

matherr

【機能】

- 浮動小数点数を扱うライブラリの例外処理を行います。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- void matherr (struct exception **x*) ;

関数名	引数	返り値

matherr	`struct exception {`
	`        int     type ;`
	`        char    *name ;`
	`}`
	
	type ：
	  演算例外を示す値
	name ：
	  関数名
		なし

【説明】

- 浮動小数点数を扱う，標準ライブラリ，ランタイム・ライブラリにおいて，例外発生時に呼び出されます。
- 標準ライブラリから呼び出された場合は，errno に EDOM，ERANGE を設定します。

次に，演算例外 type と errno の関係を示します。

type	演算例外	errno に設定する値
1	アンダーフロー	ERANGE
2	消滅	ERANGE
3	オーバーフロー	ERANGE
4	ゼロ除算	EDOM
5	演算不能	EDOM

matherr を変更，あるいは作成することで，独自のエラー処理を行うことができます。

acosf

【機能】

- acos を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float acosf (float *x*) ;

関数名	引数	返り値
acosf	*x* ： 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 ： *x* の acos *x* ＜ -1，1 ＜ *x*，*x* が非数 の場合 ： NaN

【説明】

- *x* の acos（0 から π の範囲内）を計算します。
- *x* ＜ -1，1 ＜ *x* の定義域エラーの場合は，NaN を返し，errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。

asinf

【機能】

- asin を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float asinf (float *x*) ;

関数名	引数	返り値
asinf	*x* : 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 : *x* の asin *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 : NaN *x* = -0 の場合 : -0 アンダーフロー時 : 非正規化数

【説明】

- *x* の asin（- π/2 から + π/2 の範囲内）を計算します。
- *x* ＜ -1，1 ＜ *x* の定義域エラーの場合は，NaN を返し，errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* = -0 の場合は，-0 を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

atanf

【機能】

- atan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float atanf (float *x*) ;

関数名	引数	返り値
atanf	*x* : 演算を行う数値	正常時 : *x* の atan *x* が非数の場合 : NaN *x* = -0 の場合 : -0 アンダーフロー時 : 非正規化数

【説明】

- *x* の atan（- π/2 から + π/2 の範囲内）を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* = -0 の場合は，-0 を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

atan2f

【機能】

- y/x の atan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float atan2f (float *y* , float *x*) ;

関数名	引数	返り値
atan2f	*x* : 演算を行う数値 *y* : 演算を行う数値	正常時 : *y*/*x* の atan *x* と y がともに 0 か，*y*/*x* が表現できない値の場合，*x*，*y* がともに無限大の場合，*x*，*y* のどちらかが非数の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *y*/*x* の atan（- π から + π の範囲内）を計算します。
- *x* と y がともに 0 か，*y*/*x* が表現できない値の場合，あるいは *x*，*y* がともに無限大の場合には，NaN を返し，errno に EDOM をセットします。
- *x*，*y* のどちらかが非数の場合は，NaN を返します。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。

cosf

【機能】

- cos を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float cosf (float *x*) ;

関数名	引数	返り値
cosf	*x* : 演算を行う数値	正常時 : *x* の cos *x* が非数，*x* が無限大の場合 : NaN

【説明】

- *x* の cos を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

sinf

【機能】

- sin を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float sinf (float *x*) ;

関数名	引数	返り値
sinf	*x* : 演算を行う数値	正常時 : *x* の sin *x* が非数，*x* が無限大の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *x* の sin を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

tanf

【機能】

- tan を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float tanf (float *x*) ;

関数名	引数	返り値
tanf	*x* : 演算を行う数値	正常時 : *x* の tan *x* が非数，*x* が無限大の場合 : NaN アンダーフロー時 : 非正規化数

【説明】

- *x* の tan を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- *x* の絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

coshf

【機能】

- cosh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float　coshf (float *x*) ;

関数名	引数	返り値
coshf	*x* ： 演算を行う数値	正常時 ： *x* の cosh *x* が非数の場合 ： NaN *x* が無限大の場合 ： + ∞ オーバーフロー時： HUGE_VAL（正の符号を持ちます）

【説明】

- *x* の cosh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，正の無限大の値を返します。
- 演算の結果，オーバーフローが生じた場合は，正の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。

sinhf

【機能】

- sinh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float　sinhf (float *x*) ;

関数名	引数	返り値
sinhf	*x*　: 　演算を行う数値	正常時　: 　*x* の sinh *x* が非数の場合　: 　NaN *x* = ±∞の場合　: 　±∞ オーバーフロー時　: 　HUGE_VAL（オーバーフローした値の符号を持ちます） アンダーフロー時　: 　± 0

【説明】

- *x* の sinh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が±∞の場合は，±∞を返します。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- 演算の結果，アンダーフローが生じた場合は，± 0 を返します。

tanhf

【機能】

- tanh を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float tanhf (float *x*) ;

関数名	引数	返り値
tanhf	*x* : 演算を行う数値	正常時 : *x* の tanh *x* が非数の場合 : NaN *x* = ±∞の場合 : ± 1 アンダーフロー時 : ± 0

【説明】

- *x* の tanh を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が±∞の場合は，± 1 を返します。
- 演算の結果，アンダーフローが生じた場合は，± 0 を返します。

expf

【機能】

- 指数関数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float expf (float *x*) ;

関数名	引数	返り値
expf	*x* : 演算を行う数値	正常時 : *x* の指数関数 *x* が非数の場合 : NaN *x* = + ∞の場合 : + ∞ *x* = - ∞の場合 : +0 オーバーフロー時 : HUGE_VAL（正の符号を持ちます） アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : +0

【説明】

- *x* の指数関数を計算します。
- *x* が非数の場合は，NaN を返します。
- *x* が + ∞の場合は，+ ∞を返します。
- *x* が - ∞の場合は，+0 を返します。
- 演算の結果，オーバーフローが生じた場合は，正の符号を持つ HUGE_VAL を返し， errno に ERANGE をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダーフローによる有効桁数の消滅が生じた場合は，+0 を返します。

frexpf

【機能】

- 仮数部と指数部を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float frexpf (float *x* , int **exp*) ;

関数名	引数	返り値
frexpf	*x* : 演算を行う数値 *exp* : 指数部を格納するポインタ	正常時 : *x* の仮数 *x* が非数，*x* = ±∞の場合 : NaN *x* = ± 0 の場合 : ± 0

【説明】

- 浮動小数点数 *x* を *x* = m * 2 ^ n のような仮数 m と指数 n に分け，仮数 m を返します。
- 指数 n は，ポインタ exp の指し示すところに格納します。ただし，m の絶対値は 0.5 以上 1.0 未満です。
- *x* が非数の場合は，NaN を返し，**exp* の値は 0 とします。
- *x* が±∞の場合は，NaN を返し，**exp* の値は 0 とし，errno に EDOM をセットします。
- *x* が± 0 の場合は，± 0 を返し，**exp* の値は 0 とします。

ldexpf

【機能】

- x * 2 ^ exp を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float ldexpf (float *x* , int *exp*) ;

関数名	引数	返り値
ldexpf	*x* : 演算を行う数値 *exp* : べき乗数	正常時 : *x* * 2 ^ *exp* *x* が非数の場合 : NaN *x* = ±∞の場合 : ±∞ *x* = ± 0 の場合 : ± 0 オーバーフロー時 : HUGE_VAL（オーバーフローした値の符号を持ちます） アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : ± 0

【説明】

- *x* * 2 ^ *exp* を計算します。
- *x* が非数の場合は NaN， ±∞のときは±∞，± 0 のときは，± 0 を返します。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- 演算の結果，アンダーフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダーフローによる有効桁数の消滅が生じた場合は，± 0 を返します。

logf

【機能】

- 自然対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float logf (float *x*) ;

関数名	引数	返り値
logf	*x* : 演算を行う数値	正常時 : *x* の自然対数 *x* ＜ 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* が無限大の場合 : + ∞

【説明】

- *x* の自然対数を求めます。
- *x* ＜ 0 の領域エラーの場合は，NaN を返し，errno に EDOM をセットします。
- *x* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が + ∞の場合は，+ ∞を返します。

log10f

【機能】

- 10 を底とした対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float log10f (float *x*) ;

関数名	引数	返り値
log10f	*x* : 演算を行う数値	正常時 : *x* の 10 を底とした対数 *x* ＜ 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* = + ∞の場合 : + ∞

【説明】

- *x* の 10 を底とした対数を求めます。

- *x* ＜ 0 の領域エラーの場合は，NaN を返し，errno に EDOM をセットします。

- *x* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。

- *x* が非数の場合は，NaN を返します。

- *x* が + ∞の場合は，+ ∞を返します。

modff

【機能】

- 小数部と整数部を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float modff (float *x* , float **iptr*) ;

関数名	引数	返り値
modff	*x* : 演算を行う数値 *iptr* : 整数部へのポインタ	正常時 : *x* の小数部 *x* が非数，*x* が無限大の場合 : NaN *x* = ± 0 の場合 : ± 0

【説明】

- 浮動小数点数 x を小数部と整数部に分けます。
- *x* と同じ符号を持つ小数部を返し，整数部はポインタ *iptr* の指し示すところに格納します。
- *x* が非数の場合は，NaN を返し，ポインタ *iptr* の指し示すところに NaN を格納します。
- *x* が無限大の場合は， NaN を返し，ポインタ *iptr* の指し示すところに NaN を格納し，errno に EDOM をセットします。
- *x* = ± 0 の場合は，± 0 を返し，ポインタ *iptr* の指し示すところに± 0 を格納します。

powf

【機能】

- x の y 乗を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float powf (float *x* , float *y*) ;

関数名	引数	返り値
powf	*x* : 演算を行う数値 *y* : 乗数	正常時 : *x* ^ *y* *x* が非数，または *y* が非数の場合， *x* = + ∞，かつ y = 0， *x* ＜ 0，かつ *y* ≠整数， *x* ＜ 0，かつ *y* = ±∞， *x* = 0，かつ *y* ≦ 0 のいずれかの場合 : NaN オーバーフロー時 : HUGE_VAL（オーバーフローした値の符号を持ちます。） アンダーフロー時 : 非正規化数 アンダーフローによる有効桁数の消滅時 : ± 0

【説明】

- *x* ^ *y* を計算します。
- *x* が非数，または *y* が非数の場合は，NaN を返します。
- *x* = ＋∞ かつ *y* = 0，*x* ＜ 0 かつ *y* ≠整数，*x* ＜ 0 かつ *y* = ±∞，*x* = 0 かつ *y* ≦ 0 のいずれかの場合は，NaN を返し，errno に EDOM をセットします。
- 演算の結果，オーバーフローが生じた場合は，オーバーフローした値の符号を持つ HUGE_VAL を返し，errno に ERANGE をセットします。
- アンダーフローが生じた場合は，非正規化数を返します。
- アンダーフローによる有効桁数の消滅が生じた場合は，± 0 を返します。

sqrtf

【機能】

- 平方根を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float sqrtf (float *x*) ;

関数名	引数	返り値
sqrtf	*x* : 演算を行う数値	*x* ≧ 0 の場合 : *x* の平方根 *x* ＜ 0 の場合 : 0 *x* が非数の場合 : NaN *x* = ± 0 の場合 : ± 0

【説明】

- *x* の平方根を計算します。
- *x* ＜ 0 の領域エラーの場合は，0 を返し，errno に EDOM をセットします。
- *x* が非数の場合は，NaN を返します。
- *x* が± 0 の場合は，± 0 を返します。

ceilf

【機能】

- x より小さくない最小の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float ceilf (float *x*) ;

関数名	引数	返り値
ceilf	*x* : 演算を行う数値	正常時 : *x* より小さくない最小の整数 *x* が非数，*x* が無限大の場合 : NaN *x* = -0 の場合 : +0 *x* より小さくない最小の整数を表現できない場合 : *x*

【説明】

- *x* より小さくない最小の整数を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *x* より小さくない最小の整数を表現できない場合は，*x* を返します。

fabsf

【機能】

- 浮動小数点数 x の絶対値を返します。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float fabsf (float *x*) ;

関数名	引数	返り値
fabsf	*x* : 絶対値を求める値	正常時 : *x* の絶対値 *x* が非数の場合 : NaN *x* = -0 の場合 : +0

【説明】

- *x* の絶対値を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が -0 の場合は，+0 を返します。

floorf

【機能】

- *x* より大きくない最大の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float floorf (float *x*) ;

関数名	引数	返り値
floorf	*x* : 演算を行う数値	正常時 : *x* より大きくない最大の整数 *x* が非数，*x* が無限大の場合 : NaN *x* = -0 の場合 : +0 *x* より大きくない最大の整数を表現できない場合 : *x*

【説明】

- *x* より大きくない最大の整数を求めます。
- *x* が非数の場合は，NaN を返します。
- *x* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *x* より大きくない最大の整数を表現できない場合は，*x* を返します。

fmodf

【機能】

- x/y の余りを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- float fmodf (float *x* , float *y*) ;

関数名	引数	返り値
fmodf	*x* : 演算を行う数値 *y* : 演算を行う数値	正常時 : *x/y* の余り *y* が± 0，または *x* が±∞の場合， *x* が非数，または *y* が非数の場合 : NaN *x* ≠∞，かつ *y* = ±∞の場合 : *x*

【説明】

- *x* - i * *y* で表される *x/y* の余りを計算します。i は整数です。

- *y* ≠ 0 の場合は，返り値は *x* と同じ符号を持ち，その絶対値は *y* の絶対値より小さくなります。

- *y* が± 0，または x = ±∞の場合は，NaN を返し，errno に EDOM をセットします。

- *x* が非数，または *y* が非数の場合は，NaN を返します。

- *y* が無限大の場合は，*x* が無限大でなければ *x* を返します。

10.12 診断関数

診断関数には，次のものがあります。

- __assertfail

__assertfail

【機能】

- assert マクロのサポートをします。

【ヘッダ・ファイル】

- assert.h

【関数プロトタイプ】

- int __assertfail (char *__*msg* , char *__*cond* , char *__*file* , int __*line*) ;

関数名	引数	返り値
__assertfail	__*msg* : printf 関数に渡す出力変換仕様を示す文字列へのポインタ __*cond* : assert マクロの実引数 __*file* : ソース・ファイル名 __*line* : ソース行番号	不定

【説明】

- __assertfail 関数は，assert マクロ（「10.2 (13) assert.h」を参照してください）から情報を受け取り，printf 関数を呼び，情報の出力を行い，さらに abort 関数の呼び出しを行います。
- assert マクロは，プログラム中に診断機能を付け加えます。
  assert マクロを実行するとき，p が偽（0 と等しい）の場合，assert マクロは，偽の値をもたらした特定の呼び出しに関する情報（情報の中には，実引数のテキスト，ソース・ファイル名，およびソース行番号を含みます。あとの 2 つは，それぞれマクロ __FILE__，および __LINE__ の値とします）を __assertfail 関数に渡します。

10.13 スタートアップ・ルーチン，ライブラリ関数更新用バッチ・ファイル

CC78K0R は，一部の標準ライブラリ関数，およびスタートアップ・ルーチンを更新するためのバッチ・ファイルを提供しています。bat フォルダ下にあるバッチ・ファイルについて，次に示します。

表 10-4 ライブラリ関数更新用バッチ・ファイル

バッチ・ファイル	用途
mkstup.bat	スタートアップ・ルーチン（cstart*.asm）を更新します。 スタートアップ・ルーチンを変更した場合は，このバッチ・ファイルを使用してアセンブルを行ってください。
reprom.bat	ROM 化終端ルーチン（rom.asm）を更新します。 rom.asm を更新した場合は，このバッチ・ファイルを使用してライブラリを更新してください。
repgetc.bat	getchar 関数を更新します。 デフォルトでは，SFR の P0 が入力ポートに設定されています。入力ポートを変更したい場合は，getchar.asm 中の PORT の EQU 定義値を変更し，このバッチ・ファイルを使用してライブラリを更新してください。
repputc.bat	putchar 関数を更新します。 デフォルトでは，SFR の P0 が出力ポートに設定されています。 出力ポートを変更したい場合は，putchar.asm 中の PORT の EQU 定義値を変更し，このバッチ・ファイルを使用してライブラリを更新してください。
repputcs.bat	putchar 関数を SM+ for 78K0R 対応に更新します。 SM+ for 78K0R で putchar 関数の出力を確認したい場合は，このバッチ・ファイルを使用してライブラリを更新してください。
repselo.bat	setjmp/longjmp 関数の退避／復帰処理において，コンパイラの予約領域（_@KREGxx）の退避／復帰を行うようにします（デフォルトは退避／復帰を行いません）。 -qr オプションを指定する場合は，このバッチ・ファイルを使用してライブラリを更新してください。
repselon.bat	setjmp/longjmp 関数の退避／復帰処理において，コンパイラの予約領域（_@KREGxx）の退避／復帰を行わないようにします（デフォルトは退避／復帰を行いません）。 -qr オプションを指定しない場合は，このバッチ・ファイルを使用してライブラリを更新してください。
repvect.bat	フラッシュ領域に配置する割り込みベクタ・テーブルへの分岐テーブルのアドレス値の設定処理（vect*.asm）を更新します。 デフォルトでは，フラッシュ領域分岐テーブルの先頭アドレスが 2000H に設定されていますが，フラッシュ領域分岐テーブルの先頭アドレスを変更したい場合は，vect.inc 中の ITBLTOP の EQU 定義値を変更し，このバッチ・ファイルを使用してライブラリを更新してください。

10.13.1 バッチ・ファイルの使用法

サブフォルダ bat の下に置かれたバッチ・ファイルを使用します。

アセンブラ，ライブリアンの起動を行うバッチ・ファイルとなっているため，RA78K0R アセンブラ・パッケージ Ver.1.00 以上が動作する環境が必要です。バッチ・ファイルを使用する前に，RA78K0R の実行形式ファイルがあるフォルダを環境変数 PATH で設定してください。

バッチ・ファイルは，bat と同レベルのサブフォルダ（lib）を作成し，その下にアセンブル後のファイルを置きます。C スタートアップ・ルーチン，およびライブラリが，bat と同レベルのサブフォルダ lib にインストールされている場合は，それらのファイルを上書きします。

バッチ・ファイルの使用方法は，カレント・フォルダをサブフォルダ bat に移動し，各バッチ・ファイルを実行します。その際，次のパラメータが必要です。

品種 = chiptype（ターゲット・チップの種別）

f1166a0 … uPD78F1166_A0 など

次に，各バッチ・ファイルの使用方法を示します。

(1) スタートアップ・ルーチン用

mkstup *品種*

＜例＞

```
mkstup f1166a0
```

(2) ROM 化ルーチン更新用

reprom *品種*

＜例＞

```
reprom f1166a0
```

(3) getchar 関数更新用

regetc *品種*

＜例＞

```
repgetc f1166a0
```

(4) putchar 関数更新用

repputc *品種*

＜例＞

```
repputc f1166a0
```

(5) putchar 関数（SM78K0R 対応）更新用

repputcs *品種*

＜例＞

```
repputcs f1166a0
```

(6) setjmp/longjmp 関数更新用（復帰／退避処理あり）

repselo *品種*

＜例＞

```
repselo f1166a0
```

(7) setjmp/longjmp 関数更新用（復帰／退避処理なし）

repselon *品種*

＜例＞

```
repselon f1166a0
```

(8) 割り込みベクタ・テーブル更新用

repvect *品種*

＜例＞

```
repvect f1166a0
```

第 11 章　拡張機能

この章では，ANSI（American National Standards Institute）規格に規定されていない，CC78K0R 特有の拡張機能について説明します。

CC78K0R の拡張機能は，ターゲット・デバイスである 78K0R シリーズを有効的に利用するためのコードを生成します。この拡張機能すべてが常に有効とはかぎらないので，目的にあわせて有効なもののみ使用することをお勧めします。拡張機能の効率的な使用法を「第 13 章 効率の良いコンパイラの活用法」で説明しているので，この章とあわせて参照してください。

CC78K0R の拡張機能を使った C ソース・プログラムは，マイクロコンピュータに依存した機能を利用しますが，他のマイクロコンピュータへの移植に関しては C 言語レベルで互換性を持っています。このため，拡張機能を使って作成された C ソース・プログラムにおいても，簡単な修正により他のマイクロコンピュータへ移植することができます。

備考　この章の説明において，“RTOS”は，78K0R シリーズ リアルタイム OS の意味です。

11.1 マクロ名

CC78K0R は，ターゲット・デバイスのシリーズ名を示すマクロ名と，デバイス名を示すマクロ名の 2 種類の名前を持ちます。これらは，ターゲット・デバイス用のオブジェクト・コードを出力するためにコンパイル時のオプション，または C ソース中のデバイス種別によって指定されます。例では，__K0R__ と，__F1166A0_ が指定されたことになります。

マクロ名の詳細については，「9.8 コンパイラ定義のマクロ名」を参照してください。

＜例＞

```
コンパイル時のオプション：
        >CC78K0R -cf1166a0 prime.c …

デバイス種別指定：
        #pragma pc ( f1166a0 )
```

11.2 キーワード

CC78K0R では，拡張機能を実現するために次の字句をキーワードとして追加しています。これらの字句も ANSI-C のキーワードと同様，ラベルや変数名として使用することはできません。

キーワードは，すべて英小文字で記述します。このため，英大文字が含まれているとキーワードと判断されません。

次に，CC78K0R で追加されているキーワード一覧を示します。これらのキーワードのうち，“__” で始まらないキーワードは，ANSI-C 言語仕様のみを許可するオプション（-za）指定により，無効にすることができます（ANSI-C キーワードについては，「2.2 キーワード」を参照してください）。

表 11-1 追加キーワード一覧

キーワード		用途
常に有効	-za オプション指定時は無効	
__callt	callt	callt/__callt 関数
__callf	callf	callf/__callf 関数
__sreg	sreg	sreg/__sreg 変数
−	noauto	noauto 関数
__leaf	norec	norec/__leaf 関数
__boolean	boolean	boolean 型 /__boolean 型変数
−	bit	bit 型変数
__interrupt	−	ハードウエア割り込み
__interrupt_brk	−	ソフトウエア割り込み
__banked，__non_banked	−	バンク・インタフェース
__BANK0 ～ 15	−	定数番地のバンク関数
__asm	−	ASM 文
__rtos_interrupt	−	RTOS 用割り込みハンドラ
__pascal	−	パスカル関数
__flash	−	ファーム ROM 関数
__flashf	−	__flashf 関数
__directmap	−	絶対番地配置指定
__temp	−	テンポラリ変数
__near，__far	−	メモリ配置領域指定
__mxcall	−	__mxcall 関数

(1) 関数

callt，__callt，norec，__leaf，__interrupt，__interrupt_brk，__rtos_interrupt，__flash，__flashf は，修飾属性子です。これは，関数の宣言時に先頭に記述します。
修飾宣言子の記述形式を次に示します。

修飾属性子 通常の宣言子 関数名 （ 仮引数型並び／識別子並び ）

<例>

```
__callt  int  func  ( int ) ;
```

修飾属性子の指定は，次のものに限ります。
なお，callt と __callt，norec と __leaf は，同じ指定とみなされます。ただし，“__” が付加されている修飾属性子は，-za オプション指定時でも有効となります。

- callt
- norec
- callt norec
- norec callt
- __interrupt
- __interrupt_brk
- __rtos_interrupt
- __flash
- __flashf

注意 callf，__callf，noauto，__pascal，__mxcall の記述に対しては，ワーニングを出力し無視します。

(2) 変数

- sreg，__sreg の指定は，C 言語の register と同じ規定です（sreg の詳細については，「11.5 saddr 領域利用（sreg/__sreg）」を参照してください）。
- bit，boolean，__boolean 型の指定は，C 言語の char，または int 型指定子と同じ規定です。
  ただし，これらの型は，関数の外で定義された変数（外部変数）にのみ指定することができます。
- __directmap の指定は，C 言語の型修飾子と同じ規定です（詳細については，「11.5 絶対番地配置指定（__directmap）」を参照してください）。
- __near，__far の指定は，C 言語の型修飾子と同じ規定です（詳細については，「11.5 near/far 領域指定」を参照してください）。

注意 __temp の記述に対しては，ワーニングを出力し無視します。

11.3 メモリ

メモリ・モデルは，ターゲット・デバイスのメモリ空間により決定します。

(1) メモリ・モデル

メモリ・モデルには，次のものがあります。

メモリ・モデル	説明
スモール・モデル （-ms オプション指定時）	コード部最大 64K バイト，データ部 64K バイトの合計 128K バイトのモデル
ミディアム・モデル （-mm オプション指定時）	コード部最大 1M バイト，データ部最大 64K バイトのモデル
コンパクト・モデル （-mc オプション指定時）	コード部最大 64K バイト，データ部最大 1M バイトのモデル
ラージ・モデル （-ml オプション指定時）	コード部最大 1M バイト，データ部最大 1M バイトのモデル

なお，データ部には，ROM データを含みます。

(2) レジスタ・バンク

- スタートアップ時に，レジスタ・バンクが“RB0”に設定されます（CC78K0R のスタートアップ・ルーチンの中で設定されています）。この設定により，通常（レジスタ・バンクの変更をしないかぎり）レジスタ・バンク 0 は，常に使用されます。
- レジスタ・バンク変更指定をした割り込み関数の先頭で，指定されたレジスタ・バンクに設定されます。

(3) メモリ空間

CC78K0R は，次のようにメモリ空間を利用します。

図 11-1 メモリ空間の利用

アドレス		用途		サイズ（バイト）
00	080 - 0BFH	CALLT テーブル		64
FF	E20 - EB3H	sreg 変数, boolean 型変数		148
FF	EB4 - EC3H	レジスタ変数		16
FF	EC4 - ECBH	norec 関数の引数		8
FF	ECC - ED3H	norec 関数のオートマティック変数		8
FF	ED4 - ED7H	セグメント情報格納		4
FF	ED8 - EDFH	ランタイム･ライブラリの引数		8
FF	EE0 - EF7H	RB3 - RB1	ワーク･レジスタ[注1]	24
	EF8 - EFFH	RB0	ワーク･レジスタ	8
FF	F00 - FFFH	sfr 変数		256
F0	000 - 7FFH	2nd sfr 変数		最大 2048[注2]

注 1　　レジスタ・バンク指定をしたときに使用します。

注 2　　デバイスにより異なります。

11.4 #pragma 指令

#pragma 指令は，ANSI でサポートされている前処理指令の 1 つです。#pragma に続く文字列により，コンパイラで決められた方法で翻訳するようにコンパイラに指示するものです。

#pragma 指令がコンパイラによってサポートされていなければ，#pragma 指令は無視されコンパイルが続けることができます。指令によりキーワードの追加がある場合は，そのキーワードが C ソース中にある場合にエラーが出力されます。これを避けるためには，C ソース中のキーワードを削除するか，#ifdef で切り分けます。

CC78K0R では，拡張機能を実現するために，次の #pragma 指令をサポートしています。

なお，#pragma の後ろに指定するキーワードは，大文字でも小文字でも記述可能です。

この指令を使用した拡張機能については，「11.5 拡張機能の使用方法」を参照してください。

表 11-2　#pragma 指令リスト

#pragma 指令	用途
#pragma sfr	SFR 名を c で記述する →「11.5　sfr 領域利用（sfr）」
#pragma vect #pragma interrupt	割り込み処理を C で記述する →「11.5　割り込み関数（#pragma vect/#pragma interrupt）」
#pragma di #pragma ei	DI ／ EI 命令を C で記述する →「11.5　割り込み機能（#pragma DI，#pragma EI）」
#pragma halt #pragma stop #pragma brk #pragma nop	CPU 制御命令を C で記述する →「11.5　CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」
#pragma section	コンパイラ出力セクション名を変更し，セクション配置を指定する →「11.5　コンパイラ出力セクション名の変更（#pragma section …）」
#pragma name	モジュール名を変更する →「11.5　モジュール名変更機能（#pragma name）」
#pragma rot	ローテート関数を使用する →「11.5　ローテート関数（#pragma rot）」
#pragma mul	乗算関数を使用する →「11.5　乗算関数（#pragma mul）」
#pragma div	除算関数を使用する →「11.5　除算関数（#pragma div）」
#pragma opc	データ挿入関数を使用する →「11.5　データ挿入関数（#pragma opc）」
#pragma rtos_interrupt	RX78K0R（リアルタイム OS）用割り込みハンドラを使用する →「11.5　RTOS 用割り込みハンドラ（#pragma rtos_interrupt …）」
#pragma rtos_task	RX78K0R（リアルタイム OS）用タスク関数を使用する →「11.5　RTOS 用タスク関数（#pragma rtos_task）」
#pragma ext_table	フラッシュ領域分岐テーブルの先頭アドレスを指定する →「11.5　フラッシュ領域分岐テーブル（#pragma ext_table）」
#pragma ext_func	ブート領域からフラッシュ領域への関数呼び出しを行う →「11.5　ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）」
#pragma inline	標準ライブラリ関数 memcpy，memset をインライン展開する →「11.5　メモリ操作関数（#pragma inline）」

11.5　拡張機能の使用方法

拡張機能には，次のものがあります。

- callt 関数（callt/__callt）
- レジスタ変数（register）
- saddr 領域利用（sreg/__sreg）
- sfr 領域利用（sfr）
- norec 関数（norec）
- bit 型変数，boolean 型変数（bit/boolean/__boolean）
- ASM 文（#asm 〜 #endasm/__asm）
- 漢字（/* 漢字 */，// 漢字）
- 割り込み関数（#pragma vect/#pragma interrupt）
- 割り込み関数修飾子（__interrupt，__interrupt_brk）
- 割り込み機能（#pragma DI，#pragma EI）
- CPU 制御命令（#pragma HALT/STOP/BRK/NOP）
- ビット・フィールド宣言
- コンパイラ出力セクション名の変更（#pragma section …）
- 2 進定数（2 進定数 0bxxx）
- モジュール名変更機能（#pragma name）
- ローテート関数（#pragma rot）
- 乗算関数（#pragma mul）
- 除算関数（#pragma div）
- データ挿入関数（#pragma opc）
- RTOS 用割り込みハンドラ（#pragma rtos_interrupt …）
- RTOS 用割り込みハンドラ修飾子（__rtos_interrupt）
- RTOS 用タスク関数（#pragma rtos_task）
- フラッシュ領域配置方法（-zf）
- フラッシュ領域分岐テーブル（#pragma ext_table）
- ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）
- ファーム ROM 関数（__flash）
- 引数／返り値の int 拡張抑制方法（-zb）
- メモリ操作関数（#pragma inline）
- 絶対番地配置指定（__directmap）
- near/far 領域指定
- メモリ・モデル指定

個々の拡張機能について，次の内容を説明します。

【機能】

拡張機能により実現される機能を説明します。

【効果】

拡張機能により得られる効果を説明します。

【方法】

拡張機能の利用法を説明します。

【使用例】

拡張機能の使用例を示します。

【制限】

拡張機能を利用する場合の制限を説明します。

【説明】

使用例の説明をします。

【互換性】

他の C コンパイラによって開発された C ソース・プログラムを CC78K0R によってコンパイルする場合の C ソース・プログラムの互換性を説明します。

callt 関数（callt/__callt）

【機能】

- callt 命令は，callt テーブルと呼ばれる領域 [80H - BFH] に，呼ぶ関数のアドレスを格納し，直接関数を呼ぶよりも短いコードで関数を呼ぶことを可能にします。
- callt 宣言（あるいは __callt 宣言）された関数（callt 関数と呼ぶ）の呼び出しには，関数名の先頭に？を付加した名前を使用します。呼び出しには，callt 命令を使用します。
- 呼ばれる関数は，通常の関数と変わりません。

【効果】

- オブジェクト・コードを短縮することができます。

【方法】

- 呼び出す関数に callt/__callt 属性を追加します（先頭に記述します）。

```
callt   extern  型名     関数名
__callt extern  型名     関数名
```

【使用例】

```
__callt void    func1 ( void ) ;

__callt void    func1 ( void ) {
            :
        /* 関数本体 */
            :
}
```

【制限】

- callt 関数は，メモリ・モデルによらず，[C0H - 0FFFFH] の領域に配置します。
- callt/__callt 宣言された関数のアドレスは，callt テーブルに配置されます。しかし，callt テーブルへの配置はリンク時に行われるので，アセンブラ・ソース・モジュール中で callt テーブルを利用する場合，作成するルーチンはシンボルを使い，リロケータブルにします。
- callt 関数の数に関するチェックは，リンク時に行います。
- -za オプション指定時は，__callt が有効となり，callt は無効となります。
- -zf オプション指定時は，callt 関数は定義できません。定義した場合は，エラーとなります。
- callt テーブルは 80H - BFH の領域です。
- 許される callt 属性の関数の数を越えて callt テーブルを使用した場合は，コンパイル・エラーとなります。
- -ql オプションの指定により，callt テーブルを使用します。そのため，1 ロード・モジュール当たり，およびリンクするモジュールのトータルで許される callt 属性の数は，次に示すとおりとなります。

オプション	-ql1	-ql2
callt 属性の関数の数	32	30

- -ql オプション未使用時，およびデフォルトの制限値は，次のようになります。

callt 関数	制限値
1 ロード・モジュール当たりの個数	最大 32
リンクするモジュールでトータルの個数	最大 32

【使用例】

<table>
<tr><td>

```
(C ソース)
============ ca1.c ============
__callt extern  int     tsub ( ) ;

void    main ( )
{
        int     ret_val ;
        ret_val = tsub ( ) ;
}
```

</td><td>

```
============ ca2.c ============


__callt int     tsub ( )
{
        int val ;
        return val ;
}
```

</td></tr>
<tr><td colspan="2">

```
(コンパイラの出力オブジェクト)
ca1 のモジュール
        EXTRN   ?tsub                   ; 宣言
        callt   [ ?tsub ]               ; 呼び出し

ca2 のモジュール
        PUBLIC  _tsub                   ; 宣言
        PUBLIC  ?tsub                   ;
@@CALT  CSEG    CALLT0                  ; セグメントへの割り付け
?tsub : DW      _tsub
@@BASE  CSEG    BASE
_tsub :                                 ; 関数定義
                  :
                                        ; 関数本体
                  :
```

</td></tr>
</table>

- 呼ばれる関数“tsub ( )”は callt テーブルにアドレスを格納するために callt 属性を加えてあります。

【互換性】

〈他の C コンパイラから CC78K0R〉

- キーワード callt/__callt を使用していなければ修正する必要はありません。
- callt 関数に変更する場合，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #define によって行います。詳細については，「11.6 C ソースの修正」を参照してください。

レジスタ変数（register）

【機能】

- 宣言した変数（関数引数を含む）をレジスタ（HL），saddr 領域（_@KREG00 ～ _@KREG15）に割り当てます。レジスタ宣言をしたモジュールの前処理・後処理中にレジスタ，あるいは saddr 領域の退避・復帰を行います。
- レジスタ変数の割り当て方法の詳細については，「11.7 関数呼び出しインタフェース」を参照してください。
- レジスタ変数は，参照頻度順にレジスタ HL，saddr 領域 [FFEB4H - FFEC3H] に割り当てます。ただし，レジスタ HL には，スタック・フレームがない場合のみレジスタ変数を割り当てます。saddr 領域には，-qr オプションを指定した場合のみ割り当てます。

【効果】

- レジスタ，saddr 領域に対する命令は，通常メモリに対する命令より短く，オブジェクト・コードの短縮，実行速度の向上を図ることができます。

【方法】

- 記憶域クラス指定子 register で，register クラスであることを宣言します。

```
register        型名      変数名
```

【使用例】

```
void    main ( void ) {
        register unsigned char c ;
           :
}
```

【制限】

- レジスタ変数の使用回数が少ない場合は，逆にオブジェクト・コードが増加することもあります（ソースの規模，内容に依存します）。
- レジスタ変数宣言は，char/int/short/long/float/double/long double，およびポインタに対して使用することができます。
- char は他の型に対して 1/2 の領域を，long/float/double/long double/far ポインタは 2 倍の領域を使用します。char 同士はバイト境界を持ちますが，それ以外の場合はワード境界を持ちます。
- int/short，near ポインタの場合で，1 関数当たり最大 8 変数まで使用可能とします。9 変数目からは通常のメモリに割り当てます。
- スタック・フレームがない関数の場合は，int/short，near ポインタの場合で 1 関数あたり最大 9 変数まで使用可能とし，10 変数目からは通常のメモリに割り当てます。

【使用例】

＜C ソース＞

```
void     func ( ) ;
void     main ( ) {
         register int    i , j ;
         i = 0 ;
         j = 1 ;
         i += j ;
         func ( ) ;
}
```

[レジスタ変数がレジスタ HL と saddr 領域に割り当てられた例]

次のラベルはスタートアップ・ルーチンで宣言されます（「付録 A saddr 領域のラベル一覧」を参照してください）。

＜コンパイラの出力オブジェクト＞

```
         EXTRN    _@KREG00             ; 使用する saddr 領域の参照を行う
@@CODEL CSEG
_main :
         push     hl                   ; 関数の先頭でレジスタの内容を退避する
         movw     ax , _@KREG00        ; 関数の先頭で saddr の内容を退避する
         push     ax
; line   3 :      register int i , j;
; line   4 :      i = 0;   j = 1;
         movw     hl , #00H            ; 関数中では次のようなコードを出力する
         onew     ax
         movw     _@KREG00 , ax        ; j
; line   5 :      i += j;
         addw     ax , hl
         movw     hl , ax
; line   6 :
         pop      ax                   ; 関数の終わりで saddr の内容を復帰する
         movw     _@KREG00 , ax
         pop      hl                   ; 関数の終わりでレジスタの内容を復帰する
         ret
         END
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- register 宣言をサポートしているコンパイラであれば，修正する必要はありません。
- レジスタ変数にしたい場合は，register 宣言を追加します。

〈CC78K0R から他の C コンパイラ〉

- register 宣言をサポートしているコンパイラであれば，修正する必要はありません。
- レジスタ変数がいくつまで，また，どのような領域に割り当てられるかは使用するコンパイラに依存します。

saddr 領域利用（sreg/__sreg）

(1) sreg 宣言による利用

【機能】

- sreg 宣言，あるいは __sreg 宣言された外部変数，および関数内 static 変数（sreg 変数と呼ぶ）は，自動的に saddr 領域 [FFE20H - FFEB3H]，にリロケータブルに割り当てられます。前記領域を越える場合は，コンパイル・エラーとなります。
- C ソース中における sreg 変数は通常の変数と同様に扱います。
- char/short/int/long 型の sreg 変数の各ビットは，自動的に boolean 型変数になります。
- 初期値なしで宣言された sreg 変数は初期値 0 を持ちます。
- アセンブラ・ソース中で宣言した sreg 変数のうち参照できる領域は，saddr 領域 [FFE20H - FFF1FH] です。ただし，[FFEB4H - FFEDFH] はコンパイラが使用するので，注意が必要です（図 11-1 を参照してください）。

【効果】

- saddr 領域に対する命令は，通常メモリに対する命令よりも短く，オブジェクト・コードが短縮し，実行速度が向上します。

【方法】

- 変数を定義するモジュール中，および関数の中で，sreg 宣言あるいは __sreg 宣言を行います。関数の中では，static 記憶域クラス指定子が付いている変数のみ sreg 変数にすることができます。

```
sreg    型名    変数名 / sreg   static  型名    変数名
__sreg  型名    変数名 / __sreg static  型名    変数名
```

- sreg 外部変数を参照するモジュール中では，次の宣言を行います。関数内でも記述することができます。

```
extern  sreg    型名    変数名 / extern  __sreg  型名    変数名
```

【制限】

- const 型，または関数に sreg/__sreg を指定した場合は，ワーニング・メッセージを出力し，sreg 宣言を無視します。
- char 型は，他の型の半分の領域，long/float/double/long double/far ポインタ型は 2 倍の領域を使用します。
- char 同士はバイト境界を持ちますが，それ以外の場合はワード境界を持ちます。
- -za 指定時は，__sreg のみ有効となり，sreg が無効となります。

- int/short，near ポインタの場合で 1 ロード・モジュールあたり 74 変数まで使用可能とします（saddr 領域 [FFE20H - FFEB3H] を使用した場合）。

  ただし bit , boolean 型変数を使用した場合，使用できる数は減ります。

【使用例】

＜C ソース＞

```
extern sreg     int      hsmm0  ;
extern sreg     int      hsmm1  ;
extern sreg     int      *hsptr ;

void     main  (  ) {
                 hsmm0  -= hsmm1  ;
}
```

sreg 変数の定義コードをユーザが作成する場合の例です。ただし，C ソースに extern 宣言をつけない場合は，CC78K0R が次のコードを出力します。この場合，ORG 疑似命令は出力しません。

＜アセンブラ・ソース＞

```
                PUBLIC   _hsmm0   ; 宣言
                PUBLIC   _hsmm1   ;
                PUBLIC   _hsptr   ;

@@DATS          DSEG     SADDRP   ; セグメントに割り付けます。
                ORG      0FE20H   ;
_hsmm0  :       DS       ( 2 )    ;
_hsmm1  :       DS       ( 2 )    ;
_hsptr  :       DS       ( 2 )    ;
```

関数中では，次のようなコードを出力します。

＜コンパイラの出力オブジェクト＞

```
movw     ax  , _hsmm0
subw     ax  , _hsmm1
movw     _hsmm0  , ax
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- キーワード sreg/__sreg を使用していなければ，修正する必要はありません。

  sreg 変数に変更する場合，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #define によって行います。詳細については，「11.6 C ソースの修正」を参照してください。これにより，sreg 変数は通常の変数として扱われます。

(2) 外部変数／外部 static 変数の saddr 自動割り当てオプションによる利用（-rd）

【機能】

- 外部変数／外部 static 変数（const 型を除く）を sreg 宣言あり／なしにかかわらず，自動的に saddr 領域に割り当てます。
- *n* の値と m の指定により，割り当てる外部変数，外部 static 変数を次のように指定することができます。

n，m の指定	saddr 領域に割り当てる変数
n	- *n* = 1 の場合 char，unsigned char 型の変数 - *n* = 2 の場合 *n* = 1 の場合の変数と short，unsigned short，int，unsigned int，enum，near ポインタ型の変数 - *n* = 4 の場合 *n* = 2 の場合の変数と long，unsigned long，float，double，long double，far ポインタ型の変数
m	構造体，共用体，配列
省略した場合	すべての変数

- sreg 宣言された変数は上記の指定にかかわらず，saddr 領域に割り当てます。
- extern 宣言により参照する変数についても上記に従い，saddr 領域に割り当てられているものとして処理します。
- このオプションによって，saddr 領域に割り当てられた変数は，sreg 変数と同じ扱いとなり，機能，制限は「(1) sreg 宣言による利用」で記述したとおりとなります。

【指定方法】

- -rd[*n*][m]（*n* は 1，2，または 4）オプションを指定します。

【制限】

- -rd[*n*][m] オプションで異なる *n*，m を指定したモジュール同士は，リンクすることはできません。

(3) 内部 static 変数の saddr 自動割り当てオプションによる利用（-rs）

【機能】

- 内部 static 変数（const 型を除く）を sreg 宣言あり／なしにかかわらず，自動的に saddr 領域に割り当てます。
- *n* の値と m の指定により，割り当てる内部 static 変数を次のように指定することができます。

n，m の指定	saddr 領域に割り当てる変数
n	- *n* = 1 の場合 char，unsigned char 型の変数 - *n* = 2 の場合 *n* = 1 の場合の変数と short，unsigned short，int，unsigned int，enum，near ポインタ型の変数 - *n* = 4 の場合 *n* = 2 の場合の変数と long，unsigned long，float，double，long double，far ポインタ型の変数
m	構造体，共用体，配列
省略した場合	すべての変数

- sreg 宣言された変数は上記の指定にかかわらず，saddr 領域に割り当てます。
- このオプションによって，saddr 領域に割り当てられた変数は，sreg 変数と同じ扱いとなり，機能，制限は「(1) sreg 宣言による利用」で記述したとおりとなります。

【指定方法】

- -rs[*n*][m]（*n* は 1，2，または 4）オプションを指定します。

備考　-rs[*n*][m] オプションで異なる *n*，m を指定したモジュール同士も，リンクすることができます。

sfr 領域利用（sfr）

【機能】

- sfr 領域は，78K0R シリーズの各種周辺ハードウエアに対するモード・レジスタや制御レジスタなどの特別な機能が割り付けられたレジスタ群の領域です。
- sfr 名の使用を宣言することにより，sfr 領域に関する操作が C ソース・レベルで記述することができます。
- sfr 変数は，初期値なし（不定）の外部変数です。
- 読み出し専用 sfr 変数の書き込みチェックを行います。
- 書き込み専用 sfr 変数の読み出しチェックを行います。
- sfr 変数に不正な定数データを代入した場合，コンパイル・エラーとします。
- 使用できる sfr 名は，[FFF00H - FFFFFH，および F0000H - F07FFH [注]] 中に割り付けてあるものです。

  注　デバイスにより異なります。

【効果】

- sfr 領域に関する操作を，C ソース・レベルで記述することができます。
- sfr に対する命令は，メモリに対する命令よりも短く，オブジェクト・コードの短縮，実行速度の向上を図ることができます。

【方法】

- #pragma 指令により，C ソース中に sfr 名を使用することを宣言します（キーワードの sfr は，大文字でも小文字でも記述可能です）。

```
#pragma sfr
```

- #pragma sfr は，C ソースの先頭に記述します。ただし，#pragma PC（種別）を指定する場合は，それよりも後ろに #pragma sfr を記述します。

  次のものは，#pragma sfr の前に記述することができます。

  (1) コメント

  (2) 前処理指令のうち，変数の定義／参照，関数の定義／参照を生成しないもの
- C ソース中では，デバイスが持つ sfr 名をそのまま記述します。このとき，sfr 名を宣言する必要はありません。

【制限】

- sfr 名は，大文字で記述します。小文字は通常の変数扱いとなります。

【使用例】

＜C ソース＞

```
#ifdef  __K0R__
#pragma sfr
#endif

void    main ( )
{
                PL0 -= ADCR ;
                /* ADCR = 10 ;  ==> error */
}
```

宣言に関するコードは何も出力されず，関数中で次のようなコードを出力します。

＜コンパイラの出力オブジェクト＞

```
mov     a , PL0
sub     a , ADCR
mov     PL0 , a
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- デバイスやコンパイラに依存しない部分であれば，修正する必要はありません。

〈CC78K0R から他の C コンパイラ〉

- “#pragma sfr”文を削除するか，または“#ifdef”により切り分け，sfr 変数であった変数の宣言を追加します。
  次に例を示します。

```
#ifdef  __K0R__
#pragma sfr
#else
/* 変数の宣言 */
unsigned char   P0 ;
#endif

void    main ( void ) {
        P0 = 0 ;
}
```

- sfr，またはそれに代わる機能を持つデバイスの場合，その領域をアクセスするためには専用のライブラリを作成しなければなりません。

norec 関数（norec）

【機能】

- 関数自身から他の関数を呼び出さない関数は，norec 関数にすることができます。
- norec 関数では，関数の前後処理（スタック・フレームの形成）のコードを出力しません。
- norec 関数の引数は，レジスタ，norec 関数の引数用 saddr 領域 [FFEC4H - FFECBH] に割り当てます。
- レジスタ，saddr 領域に割り当てることができない場合は，コンパイル・エラーとなります。
- 引数はレジスタ，あるいは saddr 領域 [FFEC4H - FFECBH] に格納し，norec 関数を呼び出します。
- オートマティック変数は，saddr 領域 [FFECCH - FFED3H] に割り当てます。レジスタ変数も同様です。
- saddr 領域には，コンパイル時に -qr オプションを指定した場合のみ割り当てられます。
- 引数が long/float/double/long double/far ポインタ型以外の場合，第 1 引数をレジスタ AX，第 2 引数をレジスタ DE，第 3 引数以降を saddr 領域に昇順に格納します。
  引数が long/float/double/long double/far ポインタ型の場合，第 1 引数から saddr 領域へ昇順に格納します。
  ただし，レジスタ AX，DE に格納されるのは，引数の型によらず，1 引数ずつのみです。第 1 引数は char/signed char/unsigned char の場合，レジスタ A に格納されます。
- AX，または A に格納された引数は，norec 関数の先頭で，DE に格納された引数がなければ DE，または E にコピーされ，DE に格納された引数があれば _@RTARG6，7 にコピーされます。
- オートマティック変数が long/float/double/long double/far ポインタ型以外の場合，引数の割り当て後，余っていれば参照頻度順に DE，_@RTARG6，7，_@NRARG0，1，…の順に格納していきます。
  オートマティック変数が long/float/double/long double/far ポインタ型の場合，引数の割り当て後，余っていれば参照頻度順に，_@NRARG0，1，…の順に格納していきます。
  @RTARG6，7，@NRARG0，1，…については，「付録 A saddr 領域のラベル一覧」を参照してください）。

【効果】

- オブジェクト・コードを短縮することができ，プログラムの実行速度が向上します。

【方法】

- 関数の宣言時に，norec 属性を宣言します。

```
norec    型名    関数名
```

- norec の代わりに __leaf の記述も可能です。

【制限】

- norec 関数中から，他の関数を呼び出すことはできません。
- norec 関数の引数，およびオートマティック変数には，サイズや数の制限があります。
- -za 指定時は，norec は無効となり，__leaf のみ有効となります。
- 引数，オートマティック変数に関する制限は，コンパイル時にチェックし，エラーとします。
- 引数，およびオートマティック変数にレジスタ宣言した場合は，レジスタ宣言を無視します。
- norec 関数で使用可能な引数，およびオートマティック変数の型を次に示します。
  なお，char/signed char/unsigned char 同士であれば，連続して saddr 領域に割り当てますが，それ以外の型と連続する場合は，2 バイト・アラインで割り当てます。

  (1) ポインタ

  (2) char/signed char/unsigned char

  (3) int/signed int/unsigned int

  (4) short/signed short/unsigned short

  (5) long/signed long/unsigned long

  (6) float/double/long double

（-qr オプション指定なしの場合）

- 使用できる引数の数は，long/float/double/long double/far ポインタ型以外の場合は 2 変数で，long/float/double/long double/far ポインタ型を使用することはできません。
- norec 関数内で使用可能なオートマティック変数は，long/float/double/long double/far ポインタ型以外の場合は引数で使用せず余ったバイト数分で，最大 4 バイトです。long/float/double/long double/far ポインタ型を使用することはできません。

（-qr オプション指定ありの場合）

- 使用できる引数の数は，long/float/double/long double/far ポインタ型以外の場合は 6 変数で，long/float/double/long double/far ポインタ型の場合は 2 変数です。
- norec 関数内で使用可能なオートマティック変数は，引数で使用せず余ったバイト数分と saddr のサイズ分で，long/float/double/long double/far ポインタ型以外の場合は最大 20 バイト，long/float/double/long double/far ポインタ型の場合は最大 16 バイトです。
- これらの制限は，コンパイル時にチェックしエラーとします。

【使用例】

＜C ソース＞

```
norec int       rout ( int  a , int  b , int  c ) ;

int      i ,  j ;

void     main ( ) {
         int      k , l , m ;
         i = l + rout ( k , l , m ) + ++k ;
}

norec int       rout ( int a , int b , int c ) {
         int      x , y ;
         return ( x + ( a << 2 ) ) ;
}
```

[-qr オプション指定ありの場合]

＜コンパイラの出力オブジェクト＞

```
          EXTRN    _@NRARG0               ; 使用する saddr 領域の参照を行う
          EXTRN    _@NRARG6
          EXTRN    _@NRARG1
            :
          _@NRARG0 ←m                     ; 引数を saddr 領域に格納する
            :
          de        ←l                    ; 引数を DE に格納する
            :
          ax        ←k                    ; 引数を AX に格納する
          call     !_rout                 ; norec 関数を呼び出す

 _rout :
          movw     _@RTARG6 , ax          ; saddr 領域から引数を受け取る
          shlw     ax , 2
          addw     ax , _@NRARG1          ; saddr 領域のオートマティック変数を使用する
          movw     bc , ax
          ret
```

【説明】

- rout 関数の定義に，norec 関数であることを示すための norec 属性を付けます。

【互換性】

〈他の C コンパイラから CC78K0R〉

- キーワード norec を使用していなければ，修正する必要はありません。
- norec 関数に変更する場合，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #define によって行います。詳細については，「11.6 C ソースの修正」を参照してください。

bit 型変数，boolean 型変数（bit/boolean/__boolean）

【機能】

- bit，boolean 型変数は，1 ビットのデータとして扱われ，saddr 領域に配置されます。
- bit，boolean 型変数は初期値なし（不定）の外部変数と同様に扱います。
- このビット変数に対してコンパイラは，次のビット操作命令を出力します。

```
MOV1 , AND1 , OR1 , XOR1 , SET1 , CLR1 , NOT1 , BT , BF 命令
```

【効果】

- C 記述でアセンブラ・ソース・レベルのプログラミング，saddr，sfr 領域へのビット・アクセスが可能になります。

【方法】

- bit，boolean 型変数を使用するモジュール中で bit，boolean 型宣言を行います。
- bit の代わりに __boolean を記述することも可能です。

```
bit             変数名
boolean         変数名
__boolean       変数名
```

- bit，boolean 型変数を参照するモジュール中で extern bit (boolean) 宣言を行います。

```
extern  bit             変数名
extern  boolean         変数名
extern  __boolean       変数名
```

- char/int/short/long 型の sreg 変数（配列の要素，構造体のメンバを除く），および 8 ビットの sfr 変数は自動的に bit 型変数としても使用可能になります。

```
変数名 .n  （n は 0 - 31）
```

【制限】

- bit，boolean 型変数同士の演算は，キャリー・フラグを使用して行われます。このため，各ステートメント間でのキャリー・フラグの内容は保証されません。
- 配列の定義／参照を行うことはできません。
- 構造体，共用体のメンバとして使用することはできません。
- 関数の引数の型として使用することはできません。
- オートマティック変数の型として使用することはできません。
- bit 型変数のみで，1 ロード・モジュール当たり最大 1184 変数まで使用することができます（saddr 領域 [FFE20H - FFEB3H] を使用した場合）。
- 初期値ありで宣言することはできません。
- const 宣言とともに記述された場合は，const 宣言を無視します。
- 次に示した演算子による定数との演算は，0，1 のみ可能となります。

分類	演算子
代入	=
ビットごとの AND	&，&=
ビットごとの OR	|，|=
ビットごとの XOR	^，^=
論理 AND	&&
論理 OR	||
等しい	==
等しくない	!=

- *，&（ポインタ参照，アドレス参照），sizeof 演算を行うことはできません。
- -za オプション指定時は，__boolean のみ有効となります。
- sreg 変数を使用した場合と，-rd，-rs（saddr 自動割り当てオプション）指定時には，使用可能な数は減ります。

【使用例】

＜C ソース＞

```
#define ON      1
#define OFF     0

extern  bit     data1 ;
extern  bit     data2 ;

void    main ( )
{
        data1 = ON ;
        data2 = OFF ;
        while ( data1 )  {
                data1 = data2;
                testb ( ) ;
        }

        if ( data1 && data2 ) {
                chgb ( ) ;
        }
}
```

bit 型変数の定義コードをユーザが作成する場合を示します。ただし，extern 宣言を付けない場合は，コンパイラが次のコードを出力します。このときには，ORG 疑似命令は出力しません。

＜アセンブラ・ソース＞

```
PUBLIC  _data1          ; 宣言
PUBLIC  _data2

@@BITS  BSEG            ; セグメントへの割り付け
        ORG     0FE20H
_data1  DBIT
_data2  DBIT
```

関数中では，次のようなコードを出力します。

＜コンパイラの出力オブジェクト＞

```
set1    _data1                  (初期化)
clr1    _data2                  (初期化)
bf      data1 , $?L0004         (判断)
mov1    CY , _data2             (代入)
mov1    _data1 , CY             (代入)
bf      _data1 , $?L0005        (論理 AND 式)
bf      _data2 , $?L0005        (論理 AND 式)
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- キーワード bit，boolean，__boolean を使用していなければ，修正する必要はありません。
- bit，boolean 型変数に変更する場合，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #define によって行います。詳細については，「11.6 C ソースの修正」を参照してください（この変更により，bit , boolean 型変数は通常の変数として扱われます）。

ASM 文（#asm 〜 #endasm/__asm）

【機能】

#asm 〜 #endasm

- CC78K0R が出力するアセンブラ・ソース・ファイル中に，ユーザが記述したアセンブラソースを埋め込みます。
- #asm の行と #endasm の行は出力しません。

__asm

- 文字列リテラルにアセンブリ・コードを記述することで，アセンブリ命令を出力し，アセンブラ・ソース中に挿入します。

【効果】

- C ソースのグローバル変数をアセンブラ・ソースで操作することができます。
- C ソースには記述することができない機能を実現可能です。
- C コンパイラが出力したアセンブラ・ソースをハンド・オプティマイズし，C ソース中に埋め込むことにより，効率の良いオブジェクトを得ることができます。

【方法】

#asm 〜 #endasm

- #asm でアセンブラ・ソースの開始を示し，#endasm でアセンブラ・ソースの終了を示します。アセンブラ・ソースは #asm，#endasm の間に記述します。

```
#asm
  :      /* アセンブラ・ソース */
#endasm
```

__asm

- C ソース中に次の形式で記述します。

```
__asm ( 文字列リテラル ) ;
```

- 文字列リテラルの記述方法は ANSI に準拠し，エスケープ文字列（\n：改行，\t：タブなど）や \ による行の継続，文字列の連結などの記述が可能です。

【制限】

- #asm のネストは許されません。
- ASM 文を使用した場合，オブジェクト・モジュール・ファイルは生成されず，アセンブラ・ソース・ファイルが生成されます。
- __asm は，小文字の記述のみ許します。大文字や大文字小文字混在で記述された場合，ユーザ関数とみなします。
- -za オプション指定時は，__asm のみ有効となります。
- “#asm 〜 #endasm”，および __asm は，C ソースの関数中にしか記述することができません。したがって，アセンブラ・ソースはセグメント名 @@CODE，または @@CODEL の CSEG に出力されます。

【使用例】

#asm 〜 #endasm

＜ C ソース＞

```
void    main ( ) {
#asm
        callt [ init ]
#endasm
}
```

＜コンパイラの出力オブジェクト＞

```
@@CODEL CSEG
_main :
        callt [ init ]
        ret
        END
```

#asm と #endasm の間をアセンブル・ソースとして，アセンブラ・ソース・ファイルへ出力します。

__asm

＜ C ソース＞

```
int     a , b ;

void    main ( ) {
        __asm ( "¥tmovw ax , !_a ¥t ; ax <- a" ) ;
        __asm ( "¥tmovw !_b , ax ¥t ; b <- ax" ) ;
}
```

＜アセンブラ・ソース＞

```
@@CODEL CSEG
_main :
        movw    ax , !_a        ; ax <- a
        movw    !_b , ax        ; b <- ax
        ret
        END
```

【互換性】

- #asm をサポートしている C コンパイラには，その C コンパイラで指定されるフォーマットに従って修正してください。
- ターゲット・デバイスが異なる場合，アセンブラ・ソース部分を修正してください。

漢字（/* 漢字 */, // 漢字）

【機能】

- C ソースのコメント文中に漢字を記述することができます。
- コメント中の漢字はコメントとして扱われ，コンパイルの対象とはしません。
- コメント中で使用される漢字のコードを，オプション，または環境変数により選択することができます。オプションの指定がない場合，環境変数 LANG78K に設定されたものが設定されます。
- オプションと環境変数 LANG78K の両方が指定されている場合は，オプションで指定したものが有効になります。
- 環境変数 LANG78K に SJIS と設定された場合は，コメント中の漢字種別をシフト JIS コードと解釈します。
- 環境変数 LANG78K に EUC と設定された場合は，コメント中の漢字種別を EUC コードと解釈します。
- 環境変数 LANG78K に NONE と設定された場合は，コメント中に漢字コードがないと解釈します。
- デフォルトは，SJIS を指定したものとします。

【効果】

- 理解しやすいコメントを書くことができ，C ソースの管理が容易になります。

【方法】

- コンパイラ・オプション，または環境変数のいずれかにより，漢字コードを設定します（デフォルトの設定でよい場合は，設定の必要はありません）。

(1) コンパイラ・オプションによる設定

次のオプションのうち，いずれかを指定します。

オプション	説明
-zs	SJIS（シフト JIS コード）
-ze	EUC（EUC コード）
-zn	NONE（漢字コードなし）

(2) 環境変数 LANG78K による設定

(a) SJIS，EUC，または NONE のいずれかを設定します（autoexec.bat などのファイルに必要に応じ記述します）。

(b) SJIS，EUC，NONE は，大文字でも小文字でも記述可能です。

(c) C ソースのコメント文中に漢字（環境変数 LANG78K に SJIS を設定した場合はシフト JIS コード，EUC を設定した場合は EUC コード）を記述します。

```
SET     LANG78K = SJIS   （シフト JIS コードの場合）
SET     LANG78K = EUC    （EUC コードの場合）
SET     LANG78K = NONE   （漢字コードなしの場合）
```

【制限】

- コメント文中に記述することができるのは，シフト JIS コード，EUC コードです。
  コメント以外で記述することができるのは，ASCII コードが 0x7F 以下の文字です。
  具体的には，全角文字のすべて，半角カタカナ（半角の句読点等を含む）をコメント以外には記述することができません。
  なお，記述した場合，意図したコードにならない場合があります。

【使用例】

＜C ソース＞

```
// main 関数
void    main ( )
{
        /* コメント文 */
}
```

アセンブラ・ソース中に漢字種別情報を出力します。

＜コンパイラの出力オブジェクト＞

```
$KANJICODE SJIS
```

アセンブラ・ソース中に C ソースを出力する場合，コメント中の漢字も出力します。

```
; line          1 : // main 関数
; line          2 : void    main ( )
; line          3 : {
@@CODEL CSEG
_main :
; line          4 :             /* コメント文 */
; line          5 : }
```

【説明】

- C ソースのコメント文中にのみ漢字を使うことができます。
- “// コメント”を使用する場合は，コンパイラ・オプション -zp を指定してください。

【互換性】

〈他の C コンパイラから CC78K0R〉

- コメント文を書ける以外の場所（“/* … */”，または“// …改行”の外）に漢字がある場合，修正しなければなりません。
- 漢字コードが違う場合は，漢字コードの変換が必要です。

〈CC78K0R から他の C コンパイラ〉

- コメント中に漢字を書くことができる C コンパイラに対しては，C ソースの修正はありません。
- コメント中に漢字を書くことができない C コンパイラの場合は，C ソースの漢字を削除しなければなりません。

割り込み関数（#pragma vect/#pragma interrupt）

【機能】

- 記述された関数名のアドレスを，指定された割り込み要求名に対応する割り込みベクタ・テーブルに登録します。
- 割り込み関数では，次のもののうち，使用しているもの（ASM 文中で使用されているものは除く）をスタックに退避／復帰を行うためのコードを，割り込み関数の先頭（レジスタ・バンク指定の場合は，そのコードの後ろ）と終わりに出力します。

  (1) レジスタ

  (2) レジスタ変数用 saddr 領域

  (3) norec 関数の引数／ auto 変数用 saddr 領域（使用の有無を問わない）

  (4) ランタイム・ライブラリ用 saddr 領域

  (5) セグメント情報格納用 saddr 領域

  (6) ES，CS レジスタ

  ただし，割り込み関数の指定や状況によっては，次のとおり，退避／復帰領域が異なります。

  - 無変更指定時は，レジスタ・バンクの変更，またはレジスタの退避／復帰，および saddr 領域の退避／復帰を行うためのコードを使用の有無にかかわらず，出力しません。
  - レジスタ・バンク指定がある場合は，指定されたレジスタ・バンクに変更するためのコードを割り込み関数の先頭に出力するため，レジスタの退避／復帰は行いません。
  - 無変更指定がない場合で，割り込み関数内に関数呼び出しがある場合は，レジスタに関しては，使用／未使用にかかわらず，全領域を退避／復帰します。
- コンパイル時に -qr オプションを指定しない場合は，レジスタ変数用 saddr 領域，norec 関数の引数／ auto 変数用の saddr 領域は未使用のため，退避／復帰コードを出力しません。

  なお，全退避コードの方がサイズが小さい場合は，全退避コードを出力します。

以上をまとめると，退避／復帰領域は，次のようになります。

退避／復帰領域	NO BANK	関数コールあり				関数コールなし			
		-qr なし		-qr あり		-qr なし		-qr あり	
		スタック	RBn	スタック	RBn	スタック	RBn	スタック	RBn
使用レジスタ	×	×	×	×	×	○	×	○	×
全レジスタ	×	○	×	○	×	×	×	×	×
使用ランタイム・ライブラリ用 saddr 領域，ES，CS レジスタ，セグメント情報格納用 saddr 領域	×	×	×	×	×	○	○	○	○
全ランタイム・ライブラリ用 saddr 領域，ES，CS レジスタ，セグメント情報格納用 saddr 領域	×	○	○	○	○	×	×	×	×
使用レジスタ変数用 saddr 領域	×	×	×	○	○	×	×	○	○
norec 関数の引数／auto 変数用全 saddr 領域	×	×	×	○	○	×	×	×	×

スタック　：スタック使用指定

RBn　：レジスタ・バンク指定

○　：退避する

×　：退避しない

【効果】

- C ソース・レベルで割り込み関数の記述が可能となります。
- レジスタ・バンクを変更できるため，レジスタの退避処理を行うコードを出力せず，オブジェクト・コードを縮小，実行速度を向上することができます。
- 割り込み要求名を認識するため，ベクタ・テーブルのアドレスを意識する必要がありません。

【方法】

- #pragma 指令により割り込み要求名，関数名，スタック切り替え，コンパイラが使用するレジスタ，および saddr 領域の退避／復帰を指定します。なお，#pragma 指令は C ソースの先頭に記述します（割り込み要求名に関しては，デバイスのユーザーズ・マニュアルを参照してください）。ただし，ソフトウエア割り込み BRK の場合は，BRK_I と記述してください。
- #pragma PC (種別) を記述する場合は，それよりも後ろにこの #pragma 指令を記述します。次の項目はこの #pragma 指令の前に記述することができます。

  (1) コメント

  (2) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

```
#pragma △ vect( または interrupt) △割り込み要求名△関数名△

                    [ スタック切り替え指定 ] △ ┌ スタック使用指定      ┐
                                               │ 無変更指定            │
                                               └ レジスタ・バンク指定  ┘
```

- *割り込み要求名*

  大文字で記述します。

  デバイスのユーザーズ・マニュアルを参照してください（例：NMI，INTP0 など）。

  ただし，ソフトウエア割り込み BRK の場合は，BRK_I と記述してください。

- *関数名*

  割り込み処理を記述した関数名

- *スタック切り替え指定*

  SP = 配列名 [+ オフセット位置]（例：SP = buff + 10）

  配列は，unsigned short で定義してください（例：unsigned short buff [5] ;）。

  オフセット位置は，buff のサイズ以下の偶数の値を指定してください（例：unsigned short buff[5] の場合，サイズは 10 バイトとなるので，10 以下の偶数値）。

- *スタック使用指定*

  STACK（デフォルト）

- *無変更指定*

  NOBANK

- *レジスタ・バンク指定*

  RB0/RB1/RB2/RB3

- △

  スペース

注意　CC78K0R のスタートアップ・ルーチンでは，レジスタ・バンク 0 に初期指定されているので，レジスタ・バンク 1-3 を指定するようにしてください。

【制限】

- -zf 無指定時，割り込み関数はメモリ・モデルによらず，[C0H - 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- スタック切り替え指定は，near 領域以外の配列を指定することはできません。指定した場合，エラーとなります。
- オフセット位置の指定は，偶数以外を指定することはできません。奇数を指定した場合，エラーとなります。
- スタック・ポインタを切り替えるために確保する配列は，ほかのシリーズと異なり，unsigned short 型とします。
- 割り込み要求名は，大文字で記述します。
- 1 モジュール単位でのみ，割り込み要求名の重複チェックを行います。
- 以下の 3 つ条件を満たすときに，レジスタの内容を書き換えてしまう可能性がありますが，コンパイラはこれをチェックすることはできません。
  レジスタ・バンク切り換えの設定がある場合は，レジスタ・バンクが重複しないように設定してください。
  また，レジスタ・バンクが重複するような設定を行う場合は，それらの割り込みが重ならないように，制御してください。
  NOBANK（無変更指定）を指定した場合も，レジスタの退避を行わないので，レジスタを破壊しないように制御する必要があります。

  (1) 複数の割り込みが発生

  (2) 発生した割り込みの中に，同じ BANK を使用する割り込みが複数ある

  (3) #pragma interrupt ～ の記述で，NOBANK，またはレジスタ・バンク指定がある
- 割り込み関数は，callt/__callt/norec/__leaf/__rtos_interrupt /__flash /__flashf を指定することができません。
- __far は，-zf オプション指定時のみ指定可能です。
- 割り込み関数は，引数，返り値を持つことができないため，void 型で指定します（例：void func (void) ;）。
- 割り込み関数中に ASM 文が存在しても，全退避のコードは出力しません。したがって，割り込み関数中の ASM 文中でコンパイラ予約領域などを使用する場合，または ASM 文中で関数コールを行う場合の退避はユーザが行う必要があります。
- leafwork1 ～ 16 を指定した場合は，ワーニングを出力し，無視します。
- スタック切り替えを指定した場合，配列名シンボルにオフセットを加算した位置にスタック・ポインタを切り替えます。配列名の領域の確保は #pragma 指令では行わないため，別途グローバルの unsigned short 型配列として定義する必要があります。
- 関数の先頭にスタック・ポインタを切り替えるコードを，関数の最後にスタック・ポインタを元に戻すコードを生成します。
- スタック切り替え用の配列に sreg/__sreg キーワードを付加した場合，属性が違う同名の変数が複数定義されたとみなし，コンパイル・エラーとなります。なお，-rd オプションにより saddr 領域に配列を配置させることは可能ですが，スタックとして使用されるため，コード，およびスピードに関し，効率が良くなることはありません。スタック以外の用途で saddr 領域を使用することをお勧めします。

- スタック切り替え指定は，無変更指定とは同時に指定することはできません。指定した場合は，エラーとなります。
- スタック切り替え指定は，スタック使用指定／レジスタ・バンク指定より先に記述しなければなりません。スタック切り替え指定を後に記述した場合は，エラーとなります。
- #pragma vect/#pragma interrupt 指定で退避先として無変更指定，レジスタ・バンク指定，およびスタック切り替え指定をした関数が同一モジュール内で定義されなかった場合，ワーニングを出力し退避先指定，スタック切り替えを無視します。この場合，デフォルトのスタックが使用されます。

【使用例】

[レジスタ・バンク指定がある場合]

＜C ソース＞

```
#pragma interrupt NMI inter rb1

void    inter ( )
{
        /* NMI 端子入力に対する割り込み処理 */
}
```

＜コンパイラの出力オブジェクト＞

```
@@VECT02        CSEG    AT      0002H ; NMI
_@vect02 :
        DW      _inter
@@BASE          CSEG    BASE
_inter :

        ; レジスタ・バンクの切り替えコード
        ; コンパイラが使用する saddr 領域の退避コード
        ; ES, CS レジスタの退避コード
        ; NMI 端子入力に対する割り込み処理（関数本体）
        ; ES, CS レジスタの復帰コード
        ; コンパイラが使用する saddr 領域の復帰コード
        reti
```

[スタック切り替え指定とレジスタ・バンク指定がある場合]

＜C ソース＞

```
#pragma interrupt INTP0 inter sp = buff + 10 rb2

unsigned short  buff [ 5 ] ;
void    func ( ) ;

void    inter ( )
{
        func ( ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
@@BASE      CSEG    BASE
_inter :
            sel     RB2                     ; レジスタ・バンクの切り替え
            movw    ax , sp                 ; スタック・ポインタの切り替え
            movw    sp , #_buff + 10        ;       〃
            push    ax                      ;       〃
            movw    c , #0CH                ; コンパイラが使用する saddr の退避
            dec     c                       ;       〃
            dec     c                       ;       〃
            movw    ax , _@SEGAX [ c ]      ;       〃
            push    ax                      ;       〃
            bnz     $$ - 6                  ;       〃
            mov     a , ES                  ; ES，CS レジスタの退避
            mov     x , a                   ;       〃
            mov     a , CS                  ;       〃
            push    ax                      ;       〃
            call    !!_func
            pop     ax                      ; ES，CS レジスタのの復帰
            mov     CS , a                  ;       〃
            mov     a , x                   ;       〃
            mov     CS , a                  ;       〃
            movw    de , #_@SEGAX           ; コンパイラが使用する saddr 領域の復帰
            mov     c , #06H                ;       〃
            pop     ax                      ;       〃
            movw    [ de ] , ax             ;       〃
            incw    de                      ;       〃
            incw    de                      ;       〃
            dec     c                       ;       〃
            bnz     $$ - 5                  ;       〃
            pop     ax                      ; スタックポインタを元に戻す
            movw    sp , ax                 ;       〃
            reti

@@VECT06    CSEG    AT      0006H
_@vect06 :
            DW      _inter
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 割り込み関数を使用していなければ，修正する必要はありません。
- 割り込み関数に変更する場合は，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #pragma vect ／ #pragma interrupt 指定を削除すれば，通常の関数として扱われます。
- 割り込み関数として使用する場合は，各コンパイラの仕様により変更が必要です。

割り込み関数修飾子（__interrupt，__interrupt_brk）

【機能】

- 関数を __interrupt 修飾子で宣言することにより，その関数はハードウエア割り込み関数とみなされ，ノンマスカブル／マスカブル割り込み関数のためのリターン命令 RETI により復帰します。
- 関数を __interrupt_brk 修飾子で宣言することにより，その関数はソフトウエア割り込み関数とみなされ，ソフトウエア割り込み関数のためのリターン命令 RETB により復帰します。
- この修飾子で宣言された関数は，（ノンマスカブル／マスカブル／ソフトウエア）割り込み関数とみなされ，次の (1) ～ (6) の内コンパイラの作業領域として使用しているものをスタックに退避／復帰します。
  ただし，この関数中に関数コールの記述がある場合は，全領域をスタックに退避します。

  (1) レジスタ

  (2) レジスタ変数用 saddr 領域

  (3) norec 関数の引数／ auto 変数用 saddr 領域（使用の有無を問わない）

  (4) ランタイム・ライブラリ用 saddr 領域

  (5) セグメント情報格納用 saddr 領域

  (6) ES，CS レジスタ

  備考 コンパイル時に -qr オプションを指定しない場合（デフォルト）は，(2)，(3) の領域は未使用のため，退避／復帰コードを出力しません。

【効果】

- この修飾子で宣言することにより，ベクタ・テーブルの設定と割り込み関数定義を別のファイルに記述することができます。

【方法】

- 割り込み関数の修飾子に __interrupt/__interrupt_brk のいずれかを付加します。

  ＜ノンマスカブル／マスカブル割り込み関数の場合＞

```
__interrupt     void    func ( ) { 処理 }
```

  ＜ソフトウエア割り込み関数の場合＞

```
__interrupt_brk void    func ( ) { 処理 }
```

【制限】

- -zf 無指定時，割り込み関数はメモリ・モデルによらず，[C0H - 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- 割り込み関数は，callt/__callt/norec/__leaf/__rtos_interrupt/__flash/__flashf を指定することができません。

【注意】

- この修飾子を宣言するだけでは，ベクタ・アドレスの設定を行いません。ベクタ・アドレスの設定は #pragma vect/interrupt 指令あるいはアセンブラ記述などにより，別途行う必要があります。
- saddr 領域，レジスタの退避先はスタックとなります。
- #pragma vect（または interrupt）…によりベクタ・アドレスの設定，退避先の変更を行った場合でも，同一ファイル中に関数定義がない場合は，退避先の変更は無視され，デフォルトであるスタックになります。
- #pragma vect（または interrupt）…の指定と同一ファイルに割り込み関数を定義する場合は，この修飾子を記述しなくても，#pragma vect（または interrupt）…で指定された関数名を割り込み関数と判断します。
  #pragma vect/interrupt 指令の詳細については，「割り込み関数（#pragma vect/#pragma interrupt）」を参照してください。

【使用例】

- 次のように，割り込み関数宣言，定義をします。ベクタ・アドレスの設定コードは，#pragma interrupt により生成されます。

```
#pragma interrupt   INTP0   inter   RB1      /* ソフトウエア割り込みの割り込み */
#pragma interrupt   BRK_I   inter_b RB2      /* 要求名は“BRK_I”です。*/

__interrupt     void    inter ( ) ;               /* プロトタイプ宣言 */
__interrupt_brk void    inter_b ( ) ;             /* プロトタイプ宣言 */
__interrupt     void    inter ( ) { 処理 } ;       /* 関数本体 */
__interrupt_brk void    inter_b ( ) { 処理 } ;     /* 関数本体 */
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 割り込み関数をサポートしていなければ，修正は必要ありません。
- 割り込み関数に変更したい場合は，上記の方法に従って変更します。

〈CC78K0R から他の C コンパイラ〉

- #define により可能です。通常の関数として扱えます。
- 割り込み関数として使用する場合は，各コンパイラの仕様により変更が必要です。

割り込み機能（#pragma DI，#pragma EI）

【機能】

- オブジェクトに DI，EI のコードを出力し，オブジェクト・ファイルを作成します。
- #pragma 指令がない場合，DI ( )，EI ( ) は通常の関数とみなされます。
- 関数中の先頭（オートマティック変数の宣言，コメント，プリプロセス指令を除く）に“DI ( ) ;” が記述された場合は，関数の前処理より前（関数名のラベルの直後）に DI のコードを出力します。
- 関数の前処理のあとに DI のコードを出力する場合は，“DI ( ) ;”を記述する前で新たなブロックを開きます（“ { ”で区切ります）。
- 関数中の最後（コメント，プリプロセス指令を除く）に“EI ( ) ;”が記述された場合は，関数の後処理より後ろ（RET のコードの直前）に EI のコードを出力します。
- 関数の後処理の前に EI のコードを出力する場合は，“EI ( ) ;”を記述したあとで新たなブロックを閉じます（“ } ”で区切ります）。

【効果】

- 割り込み禁止の関数を作成できます。

【方法】

- #pragma DI，#pragma EI 指令を C ソースの先頭に記述します。
  次の項目は，#pragma DI，#pragma EI の前に記述することができます。
  (1) コメント
  (2) 他の #pragma 指令
  (3) 前処理指令のうち変数の定義／参照，関数の定義／参照を生成しないもの
- 関数呼び出しと同様の形式で，ソース中に DI ( ) ;，EI ( ) ; と記述します。
- #pragma 以降に記述する DI，EI は，大文字でも小文字でも記述可能です。

【制限】

- この機能を使用する場合は，関数名として DI，EI を使用することはできません。
- DI，EI は大文字で記述します。小文字は通常の関数として扱われます。

【使用例】

```
#ifdef  __K0R__
#pragma DI
#pragma EI
#endif
```

＜ C ソース＞

```
#pragma DI
#pragma EI

void    main ( )
{
        DI ( ) ;
        ; 関数本体
        EI ( ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
_main :
        di
        ; 前処理
        ; 関数本体
        ; 後処理
        ei
        ret
```

[DI，EI を前／後処理の後と前に出力する場合]

＜ C ソース＞

```
#pragma DI
#pragma EI

void    main ( )
{
        {
                DI ( ) ;
                ; 関数本体
                EI ( ) ;
        }
}
```

＜コンパイラの出力オブジェクト＞

```
_main :
        ; 前処理
        di
        ; 関数本体
        ei
        ; 後処理
        ret
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 割り込み機能を使用していなければ，修正する必要はありません。
- 割り込み機能を使用している場合は，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- #pragma DI，#pragma EI 指令を削除するか，あるいは #ifdef で切り分けます。関数名として DI，EI を使用することができます（例：#ifdef __K0R__ ～ #endif）。
- 割り込み機能として使用する場合は，各コンパイラの仕様により変更が必要です。

CPU 制御命令（#pragma HALT/STOP/BRK/NOP）

【機能】

- オブジェクトに次のコードを出力し，オブジェクト・ファイルを作成します。
  (1) HALT 動作用の命令を出力します（HALT）。
  (2) STOP 動作用の命令を出力します（STOP）。
  (3) BRK 命令を出力します。
  (4) NOP 命令を出力します。

【効果】

- マイクロコンピュータのスタンバイ機能を，C プログラムで使用することができます。
- ソフトウエア割り込みを発生することができます。
- CPU を動作させずに，クロックを進めることができます。

【方法】

- #pragma HALT，#pragma STOP，#pragma NOP，#pragma BRK 命令を C ソースの先頭に記述します。
- 次の項目は， #pragma 指令の前に記述することができます。
  (1) コメント
  (2) 他の #pragma 指令
  (3) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの
- #pragma 以降のキーワードは大文字でも小文字でも記述可能です。
- 関数呼び出しと同様の形式で，C ソース中に次のように大文字で記述します。
  (1) HALT ( ) ;
  (2) STOP ( ) ;
  (3) BRK ( ) ;
  (4) NOP ( ) ;

【制限】

- この機能を使用する場合は，関数名として HALT，STOP，BRK，NOP を使用することができません。
- HALT，STOP，BRK，NOP は大文字で記述します。小文字は通常の関数扱いとなります。

【使用例】

＜C ソース＞

```
#pragma HALT
#pragma STOP
#pragma BRK
#pragma NOP

void    main ( )
{
        HALT ( ) ;
        STOP ( ) ;
        BRK ( ) ;
        NOP ( ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
@@CODEL CSEG
_main :
        halt
        stop
        brk
        nop
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- CPU 制御命令を使用していなければ，修正する必要はありません。
- CPU 制御命令を使用したい場合は，前記の方法に従って修正します。

〈CC78K0R から他の C コンパイラ〉

- “#pragma HALT”，“#pragma STOP”，“#pragma BRK”，“#pragma NOP”文を削除，あるいは #ifdef で切り分けると，関数名として HALT，STOP，BRK，NOP を使用できます。
- CPU 制御命令として使用する場合は，各コンパイラの仕様により変更が必要です。

ビット・フィールド宣言

(1) 型指定子の拡張

【機能】

- unsigned char，signed char 型のビット・フィールドは，バイト境界をまたがって割り付けられることはありません。
- unsigned int，signed int 型のビット・フィールドは，ワード境界をまたがって割り付けられることはありません。バイト境界をまたがって割り付けることは可能です。
- 同じ型のビット・フィールドは，同じバイト単位（またはワード単位）に割り付けられます。
  違う型の場合は，違うバイト単位（またはワード単位）に割り付けられます。

【効果】

- メモリの節約，オブジェクト・コードの短縮，実行速度の向上を図ることができます。

【方法】

- ビット・フィールドの型指定子として， unsigned int 型に加え，unsigned char，signed char，signed int 型の指定を行うことができます。
  次のように宣言します。

```
struct  タグ名 {
        unsigned char    フィールド名：ビット幅 ;
        unsigned char    フィールド名：ビット幅 ;
          :
        unsigned int     フィールド名：ビット幅 ;
} ;
```

【使用例】

```
struct tagname {
       unsigned char   A : 1 ;
       unsigned char   B : 1 ;
         :
       unsigned int    C : 2 ;
       unsigned int    D : 1 ;
         :
} ;
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- ソースの修正は必要ありません。
- 型指定子に unsigned char を使用したい場合は，型指定子を変更します。

〈CC78K0R から他の C コンパイラ〉

- 型指定子に unsigned char，signed char，signed int を使用していなければ，修正は必要ありません。
- 型指定子に unsigned char，signed char，signed int を使用している場合は，unsigned int に変更します。

(2) ビット・フィールドの割り付け方向

【機能】

- ビット・フィールドの割り付け方向を，-rb オプション指定により MSB 側 からに変更します。
- -rb オプション指定がない場合は，LSB 側から割り付けられます。

【方法】

- ビット・フィールドを MSB 側から割り付ける場合，コンパイル時に -rb オプションを指定します。
- ビット・フィールドを LSB 側から割り付ける場合，オプションは指定しません。

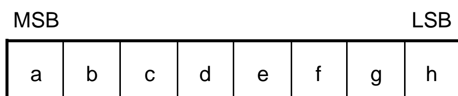
【使用例 1】

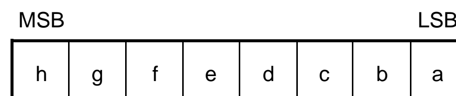
＜ビット・フィールドの宣言＞

```
struct t {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
        unsigned char   g : 1 ;
        unsigned char   h : 1 ;
} ;
```

- a - h は 8 ビット以下なので，1 バイト単位中に割り付けます。

-rb オプション指定時の MSB から割り付けたビット配置	-rb オプション無指定時の LSB から割り付けたビット配置



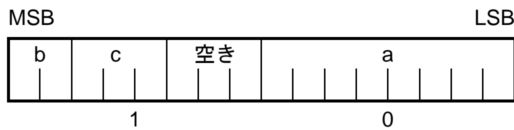


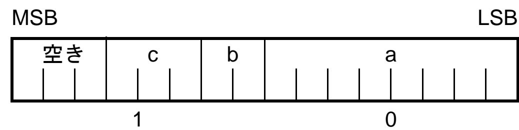
【使用例 2】

＜ビット・フィールドの宣言＞

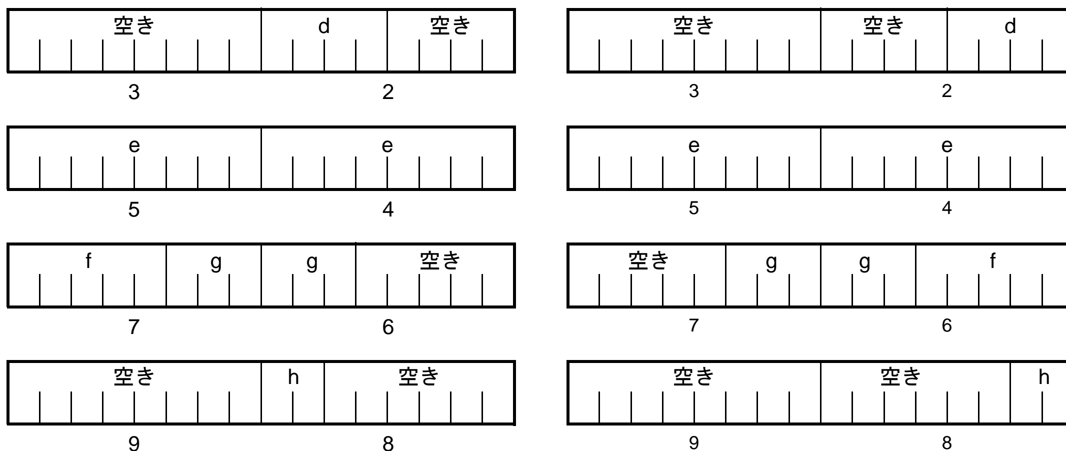
```
struct  t {
        char            a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned char   f : 5 ;
        unsigned char   g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 2 ;
} ;
```

-rb オプション指定時の MSB から割り付けたビット配置	-rb オプション無指定時の LSB から割り付けたビット配置



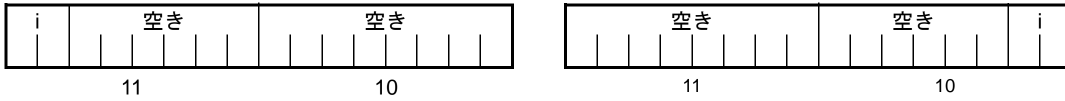


char 型のメンバ a を最初のバイト単位に割り付けます。b，c は次のバイト単位から割り付けます。十分な空きがなくなれば，次のバイト単位に割り付けます。ここでは，空きが 3 ビットで，d が 4 ビットなので，d は次のバイト単位に割り付けます。



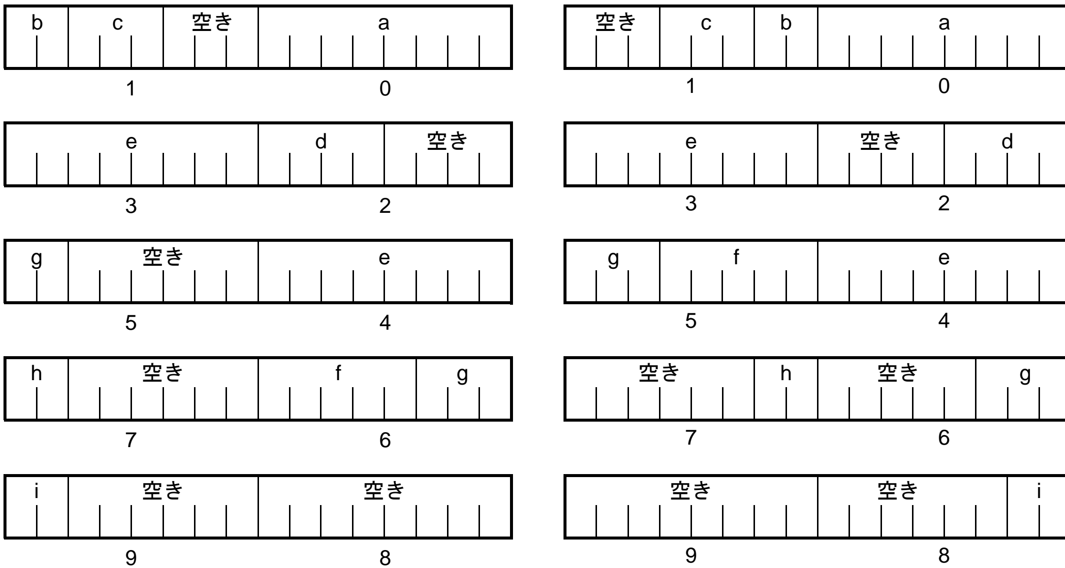
g は unsigned int 型のビット・フィールドなので，バイト境界をまたがっても割り付けます。

h は unsigned char 型のビット・フィールドなので，unsigned int 型のビット・フィールドの g と同じバイト単位ではなく，次のバイト単位に割り付けます。



i は unsigned int 型のビット・フィールドなので，次のワード単位に割り付けます。

-rc オプション指定時（構造体メンバをパッキングする）には，前記ビット・フィールドの配置は，次のとおりとなります。



備考　ビット配置図の下の数字は，構造体の先頭からのバイト・オフセット値を示します。

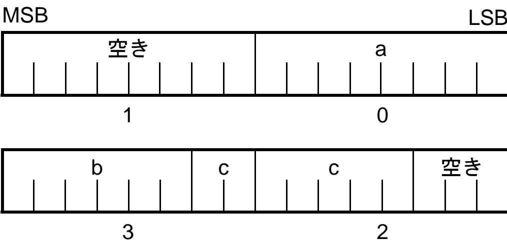
【使用例 3】

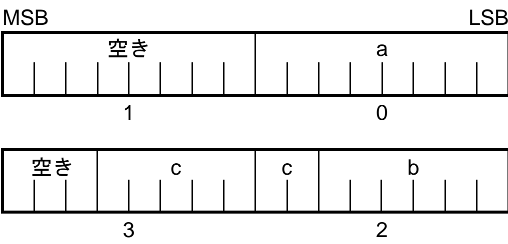
＜ビット・フィールドの宣言＞

```
struct  t {
        char            a ;
        unsigned int    b : 6 ;
        unsigned int    c : 7 ;
        unsigned int    d : 4 ;
        unsigned char   e : 3 ;
        unsigned int    f : 10 ;
        unsigned int    g : 2 ;
        unsigned int    h : 5 ;
        unsigned int    i : 6 ;
} ;
```

-rb オプション指定時の
MSB から割り付けたビット配置

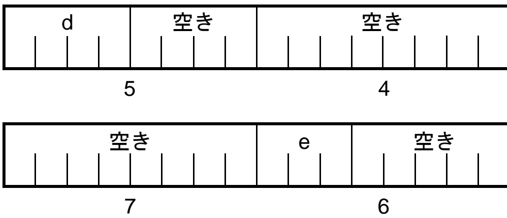
-rb オプション無指定時の
LSB から割り付けたビット配置

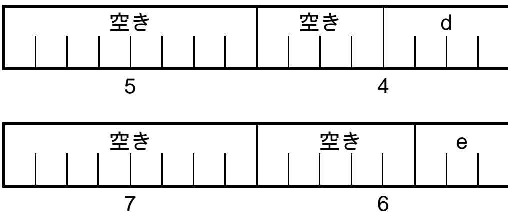




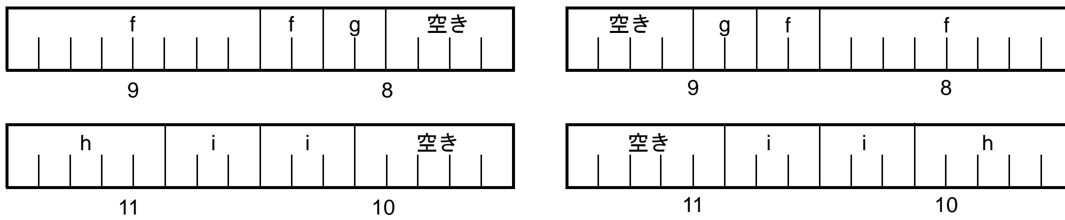
b，c は unsigned int 型のビット・フィールドなので，次のワード単位から割り付けます。

d も unsigned int 型のビット・フィールドなので，次のワード単位から割り付けます。



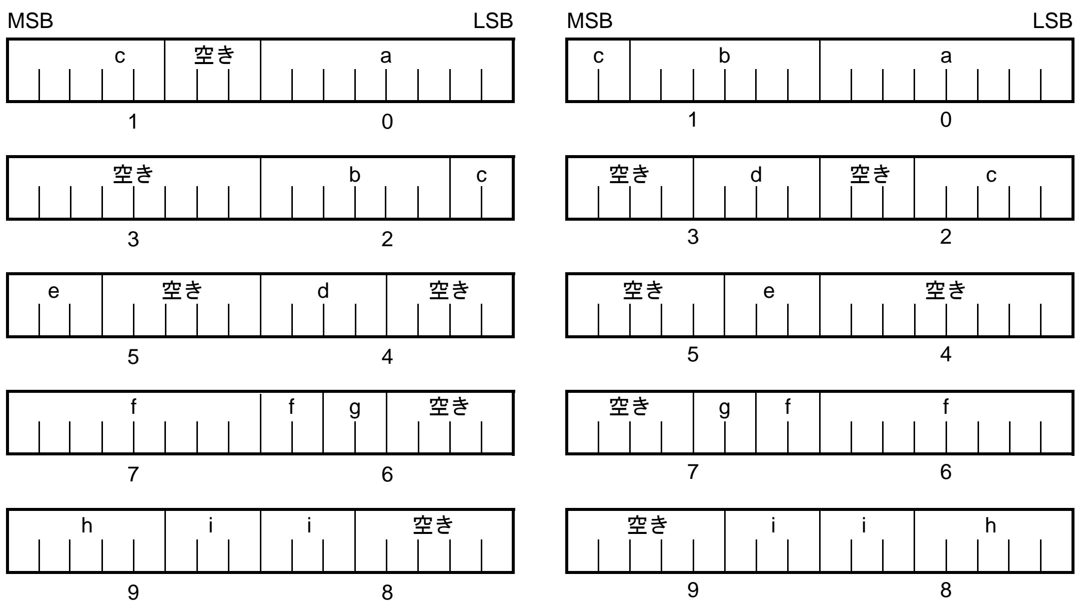


e は unsigned char 型のビット・フィールドなので，次のバイト単位に割り付けます。



f，g と h，i は，それぞれワード単位ごとに割り付けます。

-rc オプション指定時（構造体メンバをパッキングする）には，前記ビット・フィールドの配置は，次のとおりとなります。



備考　ビット配置図の下の数字は，構造体の先頭からのバイト・オフセット値を示します。

【互換性】

〈他の C コンパイラから CC78K0R〉

- 修正は必要ありません。

〈CC78K0R から他の C コンパイラ〉

- -rb オプションを指定し，ビット・フィールドが割り付けられる順序を考慮したコーディングをしている場合は，変更が必要です。

コンパイラ出力セクション名の変更（#pragma section …）

【機能】

- コンパイラ出力セクション名の変更と，開始アドレスの指定を行います。
  開始アドレスを省略した場合は，デフォルトの配置となります。コンパイラ出力セクション名とデフォルトの配置については，「付録 B セグメント名一覧」を参照してください。
  また，開始アドレスを省略し，リンク時にリンク・ディレクティブ・ファイルを使用してセクション配置を指定することができます。リンク・ディレクティブについては，「RA78K0R アセンブラ・パッケージ 操作編」のユーザーズ・マニュアルを参照してください。
- @@CALT セクション名を AT 開始アドレス指定付きで変更する場合は，callt 関数はソース・ファイル中で他の関数より前，または後ろにまとめて記述しなければなりません。
- #pragma 指令が記述された以降にデータを記述した場合，そのデータを変更セクションに配置します。
  再変更指令も可能であり，再変更指令以降にデータを記述した場合，そのデータを再変更セクションに配置します。
  変更前に定義したデータを，変更後に再定義した場合，再変更されたセクションに配置します。
  なお，（関数内）static 変数に対しても同様に有効です。

【効果】

- コンパイラ出力セクションを 1 ファイル中に何度も変更することにより，各セクションをそれぞれ独立に配置することができるようになるため，独立に配置したいデータの単位で，データを配置することができます。

【方法】

- 次の #pragma 指令により，変更するセクション名と変更後のセクション名，およびセクションの開始アドレスを指定します。
  なお，この #pragma 指令は， C ソースの先頭に記述します。
  #pragma PC (種別) を記述する場合は，それよりも後ろにこの #pragma 指令を記述します。
  次の項目は，この #pragma 指令の前に記述することができます。

  (1) コメント

  (2) 前処理指令のうち，変数の定義／参照，関数の定義／参照を生成しないもの

  ただし，BSEG のすべてのセクション，DSEG のすべてのセクション，および CSEG のうちの @@CNST，@@CNSTL セクションは，C ソース中のどこに記述してもよく，また何度でも再変更指令を行うことができます。元のセクション名に戻す場合は，変更セクション名にコンパイラ出力セクション名を記述します。
  ファイルの先頭に，次のような宣言をします。

```
#pragma section コンパイラ出力セクション名  変更セクション名   [ AT 開始アドレス ]
```

- #pragma 以降に記述するキーワードのうち，コンパイラ出力セクション名は，必ず大文字で記述してください。
  section，AT は，大文字でも小文字でも大小文字混在でも記述可能です。
- 変更セクション名の書式は，アセンブラの仕様に準拠します（セグメント名は 8 文字までです）。
- 開始アドレスには，C 言語の 16 進数および，アセンブラの 16 進数のみ記述することができます。

【C 言語の 16 進数】

```
0xn/0xn … n
0Xn/0Xn … n
(n = 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , A , B , C , D , E , F)
```

【アセンブラの 16 進数】

```
nH/n … nH
nh/n … nh
(n = 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , A , B , C , D , E , F)
```

16 進数の先頭文字は数字でなければなりません。

＜例＞

値が 255 の数値を 16 進数で表現する場合は，F の前にゼロを指定し，0FFH とする必要があります。

- CSEG のうち，@@CNST，@@CNSTL セクション以外のセクション，つまり関数を配置するセクションは，C ソースの先頭以外（C の本文記述後）にこの #pragma 指令を記述することはできません。ワーニングを出力し無視します。
- C の本文記述後にこの #pragma 指令を行った場合，オブジェクト・モジュール・ファイルは作成されず，アセンブラ・ソース・ファイルが作成されます。
- C の本文記述後にこの #pragma 指令がある場合，この #pragma 指令があり，C の本文（変数や関数の外部参照宣言を含む）のいっさいないファイルはインクルードすることはできません。エラーになります（後述の「【エラー記述例 1】」を参照）。
- C の本文記述後にこの #pragma 指令を行ったファイルでは，この記述以降，#include 文を記述することはできません。エラーになります（後述の「【エラー記述例 2】」を参照）。
- C の本文のあとに #include 文があった場合，この記述以降，この #pragma 指令を記述することはできません。エラーになります（後述の「【エラー記述例 3】」を参照）。

【使用例 1】

セクション名 @@CODEL を CC1 に変更し，開始アドレスを 2400H 番地に指定します。

＜C ソース＞

```
#pragma section @@CODEL CC1     AT      2400H

void    main ( )
{
        ; 関数本体
}
```

＜出力オブジェクト＞

```
CC1     CSEG    AT      2400H
_main :
        ; 前処理
        ; 関数本体
        ; 後処理
        ret
```

【使用例 2】

C の本文があり，そのあとにこの #pragma 指令を記述する場合の記述例を示します。

なお，// 以降に配置するセクションを示します。

```
#pragma section @@DATA          ??DATA
int             a1 ;                    // ??DATA
sreg int        b1 ;                    // @@DATS
int             c1 = 1 ;                // @@INIT と @@R_INIT
const int       d1 = 1 ;                // @@CNST
#pragma section @@DATS          ??DATS
int             a2 ;                    // ??DATA
sreg int        b2 ;                    // ??DATS
int             c2 = 1 ;                // @@INIT と @@R_INIT
const int       d2 = 1 ;                // @@CNST
#pragma section @@DATA          ??DATA2
// ??DATA が自動的に閉じられ，??DATA2 が有効となる
int             a3 ;                    // ??DATA2
sreg int        b3 ;                    // ??DATS
int             c3 = 3 ;                // @@INIT と @@R_INIT
const int       d3 = 3 ;                // @@CNST
#pragma section @@DATA          @@DATA
// ??DATA2 が閉じられ，デフォルト @@DATA に戻る
#pragma section @@INIT          ??INIT
#pragma section @@R_INIT        ??R_INIT
// @@INIT，@@R_INIT の両方の名前を変えないと ROM 化が破綻するが，それはユーザ責任
int             a4 ;                    // @@DATA
sreg int        b4 ;                    // ??DATS
int             c4 = 1 ;                // ??INIT と ??R_INIT
const int       d4 = 1 ;                // @@CNST
#pragma section @@INIT          @@INIT
#pragma section @@R_INIT        @@R_INIT
// ??INIT，??R_INIT が閉じられ，デフォルトに戻る
#pragma section @@BITS          ??BITS
__boolean e4 ;                          // ??BITS
#pragma section @@CNST          ??CNST
char            *const p = "Hello" ;    // p も "Hello" も ??CNST
```

【使用例 3】

```
#pragma section     @@DATA      ??DATA1
int          a1 ;                      // ??DATA
sreg int     b1 ;                      // @@DATS
int          c1 = 1 ;                  // @@INIT と @@R_INIT
const int    d1 = 1 ;                  // @@CNST
#pragma section     @@DATS      ??DATS
int          a2 ;                      // ??DATA
sreg int     b2 ;                      // ??DATS
int          c2 = 1 ;                  // @@INIT と @@R_INIT
const int    d2 = 1 ;                  // @@CNST
#pragma section     @@DATA      ??DATA2
// ??DATA が自動的に閉じられ，??DATA2 が有効となる
int          a3 ;                      // ??DATA2
sreg int     b3 ;                      // ??DATS
int          c3 = 3 ;                  // @@INIT と @@R_INIT
const int    d3 = 3 ;                  // @@CNST
#pragma section     @@DATA      @DATA
// ??DATA2 が閉じられ，デフォルト @@DATA に戻る
#pragma section     @@INIT      ??INIT
#pragma section     @@R_INIT    ??R_INIT
// @@INIT，@@R_INIT の両方の名前を変えないと ROM 化が破綻するが，それはユーザ責任
int          a4 ;                      // @@DATA
sreg int     b4 ;                      // ??DATS
int          c4 = 1 ;                  // ??INIT と ??R_INIT
const int    d4 = 1 ;                  // @@CNST
#pragma section     @@INIT      @@INIT
#pragma section     @@R_INIT    @@R_INIT
// ??INIT，??R_INIT が閉じられ，デフォルトに戻る
#pragma section     @@BITS      ??BITS
__boolean    e4 ;                      // ??BITS
#pragma section     @@CNST      ??CNST
char * const p = "Hello" ;             // p も "Hello" も ??CNST
--
#pragma section     @@INIT      ??INIT1
#pragma section     @@R_INIT    ??R_INIT1
#pragma section     @@DATA      ??DATA1
char         c1 ;
int          i2 ;
#pragma section     @@INIT      ??INIT2
#pragma section     @@R_INIT    ??R_INIT2
#pragma section     @@DATA      ??DATA2
char         c1 ;
int          i2 = 1 ;
#pragma section     @@DATA      ??DATA3
#pragma section     @@INIT      ??INIT3
#pragma section     @@R_INIT    ??R_INIT3
extern char c1 ;                       // ??DATA3
int          i2 ;                      // ??INIT3 と ??R_INIT3
#pragma section     @@DATA      ??DATA4
#pragma section     @@INIT      ??INIT4
#pragma section     @@R_INIT    ??R_INIT4
```

C の本文があり，そのあとにこの #pragma 指令を記述する場合の制限を，次のエラー記述例で説明します。

【エラー記述例 1】

```
a1.h
        #pragma section @@DATA  ??DATA1         // #pragma section のみのファイル

a2.h
        extern int      func1( void ) ;
        #pragma section @@DATA  ??DATA2         // C の本文があり，そのあとにこの
                                                // #pragma 指令があるファイル

a3.h
        #pragma section @@DATA  ??DATA3         // #pragma section のみのファイル

a4.h
        #pragma section @@DATA  ??DATA3
        extern int      func2 ( void ) ;        // C の本文を含むファイル

a.c
        #include "a1.h"
        #include "a2.h"
        #include "a3.h"         // ←エラー
                                // a2.h で C の本文があり，そのあとにこの #pragma
                                // 指令があるので，この pragma 指令のみのファイル
                                // である a3.h をインクルードできない
        #include "a4.h"
```

【エラー記述例 2】

```
b1.h
        const int i ;

b2.h
        const int       j ;
        #include "b1.h"         // C の本文があり，そのあとにこの #pragma 指令を行った
                                // ファイル（b.c）ではないので，エラーではない

b.c
        const int       k ;
        #pragma section @@DATA  ??DATA1
        #include "b2.h"         // ←エラー
                                // C の本文があり，そのあとにこの #pragma 指令を行った
                                // ファイル（b.c）においては，include 文を記述できない
```

【エラー記述例 3】

```
c1.h
        extern int       j ;
        #pragma section @@DATA  ??DATA1 // c3.h 処理前にインクルードされ，処理される
                                        // ため，エラーではない

c2.h
        extern int       k ;
        #pragma section @@DATA  ??DATA2 // ←エラー
                                        // c3.h で c の本文があり，そのあとに
                                        // #include 文があるので，それ以降この
                                        // #pragma 指令はできない

c3.h
        #include "c1.h"
        extern int       i ;
        #include "c2.h"
        #pragma section @@DATA  ??DATA3 // ←エラー
                                        // C の本文があり，そのあとに #include 文が
                                        // あるので，それ以降この #pragma 指令は
                                        // できない
c.c
        #include "c3.h"
        #pragma section @@DATA  ??DATA4 // ←エラー
                                        // c3.h で C の本文があり，そのあとに
                                        // #include 文があるので，それ以降この
                                        // #pragma 指令はできない
        int     i ;
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- セクション名変更機能をサポートしていなければ，修正は必要ありません。
- セクション名を変更をしたい場合は，上記の方法に従って変更します。

〈CC78K0R から他の C コンパイラ〉

- #pragma section …を削除，または #ifdef で切り分けます。
- セクション名を変更する場合は，各コンパイラの仕様により変更が必要です。

【制限】

- ベクタ・テーブル用セグメントを示すセクション名（たとえば @@VECT02 など）を変更することはできません。
- AT 開始アドレス指定の同名セクションは，（他ファイルも含めて）複数あるとリンク・エラーとなります。
- コンパイラ出力セクション名 @@DATS，@@BITS，@@INIS の指定アドレスは FFE20H - FFEB3H の範囲内に，@@CALT の指定アドレスは 0x80 - 0xbf の範囲内に，@@CODE，@@BASE の指定アドレスは 0x0 - 0xffff の範囲内に，その他のセクションの指定アドレスは 0x0 - 0xffeff の範囲内にしてください。

【注意】

- セクションは，アセンブラにおけるセグメントに相当します。
- コンパイラは，変更セクション名と他のシンボルとの重複チェックをしません。したがって，ユーザは出力アセンブル・リストをアセンブルするなどして，重複していないか確認してください。
- #pragma section の使用により ROM 化関連のセクション名[注]を変更した場合，スタートアップ・ルーチンの変更はユーザ責任となります。
- -zf オプション指定時には，セクション名の先頭から 2 番目の“@”を“E”に変更したセクション名となります。

注 ROM 化関連セクション名

```
@@R_INIT, @@R_INIS, @@RLINIT, @@INITL, @@INIT, @@INIS
```

ROM 化関連セクション変更にともなうスタートアップ・ルーチン，終端ルーチンの変更例について，次に示します。

【ROM 化関連セクション名変更に伴うスタートアップ・ルーチンなどの変更例】

ROM 化関連セクション名変更に伴うスタートアップ・ルーチン（cstart.asm または cstartn.asm），終端ルーチン（rom.asm）の変更例を示します。

＜C ソース＞

```
#pragma section @@R_INIT        RTT1
#pragma section @@INIT          TT1
```

上に示した #pragma section の記述により，初期値あり外部変数を格納するセクション名を変更した場合，ユーザは変更したセクションに格納する外部変数の初期化処理を，スタートアップ・ルーチンに追加する必要があります。

つまり，スタートアップ・ルーチンには，変更したセクションの先頭ラベルの宣言と，初期値のコピーを行う部分を追加し，終端ルーチンには終端ラベルの宣言を行う部分を追加します。

次に，その方法を示します。

RTT1_S，RTT1_E は，セクション RTT1 の先頭と終端のラベルの名前であり，TT1_S，TT1_E は，セクション TT1 の先頭と終端のラベルの名前です。

(1) スタートアップ・ルーチン cstartx.asm の変更点

(a) 名前を変更したセクションの終端ラベルの宣言を追加します。

```
  :
#pragma section @@R_INIT    RTT1
#pragma section @@INIT      TT1
EXTRN   RTT1_E , TT1_E          ← RTT1_E, TT1_E の EXTRN 宣言を追加する
  :
```

(b) 名前を変更した RTT1 セクションから TT1 セクションへの初期値のコピーを行う部分を追加します。

```
  :
LDATS1 :
        MOVW    AX , HL
        CMPW    AX , #LOW _?DATS
        BZ      $LDATS2
        MOV     [ HL + 0 ] , #0
        INCW    HL
        BR      $LDATS1
LDATS2 :
        MOV     ES , #HIGH RTT1_S
        MOV     HL , #LOWW RTT1_S
        MOV     DE , #LOWW TT1_S
LTT1 :                                  ┐
        MOVW    AX , HL                 │
        CMPW    AX , #LOWW TT1_E        │
        BZ      $LTT2                   │ RTT1 セクションから TT1 セクション
        MOV     A , ES : [ HL ]         │ へ初期値をコピーする部分を追加
        MOV     [ DE ] , A              │
        INCW    HL                      │
        INCW    DE                      │
        BR      $LTT1                   ┘
LTT2 :
;
        CALL    !!_main                 ; main ( ) ;
        CLRW    AX
        CALL    !!_exit                 ; exit ( 0 ) ;
        BR      $$
;
```

(c) 名前を変更したセクションの先頭のラベルを設定します。

```
  :
@@R_INIT        CSEG    UNIT64KP
_@R_INIT :
@@R_INIS        CSEG    UNIT64KP
_@R_INIS :
@@INIT          DSEG
_@INIT :
@@DATA          DSEG
_@DATA :
@@INIS          DSEG    SADDRP
_@INIS :
@@DATS          DSEG    SADDRP
_@DATS :

RTT1            CSEG    UNIT64KP        ; セクション RTT1 の先頭を示す
RTT1_S :                                ; ラベルの設定を追加
TT1             DSEG    BASEP           ; セクション TT1 の先頭を示す
TT1_S :                                 ; ラベルの設定を追加

@@CODEL         CSEG
@@CALT          CSEG    CALLT0
@@CNST          CSEG    MIRRORP
@@BITS          BSEG
;
        END
```

(2) 終端ルーチン rom.asm の変更点

(a) 名前を変更したセクションの終端を示すラベルの宣言

```
NAME            @rom
;
PUBLIC          _?R_INIT , _?R_INIS
PUBLIC          _?INIT , _?DATA , _?INIS , _?DATS

PUBLIC          RTT1_E , TT1_E          ← RTT1_E, TT1_E を追加

;
@@R_INIT        CSEG    UNIT64KP
_?R_INIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG
_?INIT :
@@DATA          DSEG
_?DATA :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS
  :
```

(b) 終端を示すラベルの設定

```
  :
RTT1    CSEG    UNIT64KP        ; セクション RTT1 の終端を示す
RTT1_E :                        ; ラベルの設定を追加

TT1     DSEG    BASEP           ; セクション TT1 の終端を示す
TT1_E :                         ; ラベルの設定を追加

;
        END
```

2 進定数（2 進定数 0bxxx）

【機能】

- 整数定数が記述可能な位置に，2 進定数を記述することができます。

【効果】

- ビット列で定数を記述したい場合，8 進数や 16 進数などに置き換えずに直接記述することができ，可読性も良くなります。

【方法】

- C ソース中で，2 進定数を記述します。
  2 進定数の記述方法は，次のとおりです。

```
0b        2 進数字
0B        2 進数字
```

備考 2 進数字：“0”か“1”のいずれか 1 つです。

- 2 進定数は先頭に 0b または 0B があり，0，または 1 の数字の並びが後ろに続きます。
- 2 進定数の値は，2 を基数として計算されます。
- 2 進定数の型は，次のリスト中でその値を表現することのできる最初のものです。

添字なし 2 進数	：	int，unsigned int，long int，unsigned long int
u，または U の添字付き	：	unsigned int，unsigned long int
l，または L の添字付き	：	long int，unsigned long int
u，または U の添字，および l，または L の添字付き	：	unsigned long int

【使用例】

＜C ソース＞

```
        unsigned        i ;
        i = 0b11100101 ;
```

コンパイラの出力オブジェクトは以下の場合と同じです。

```
        unsigned        i ;
        i = 0xE5 ;
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 修正の必要はありません。

〈CC78K0R から他の C コンパイラ〉

- 2 進定数をサポートしている場合コンパイラの場合は，そのコンパイラの仕様にあうように修正する必要があります。
- 2 進定数をサポートしていないコンパイラの場合は，8 進，10 進，16 進などの他の整定数形式に修正する必要があります。

モジュール名変更機能（#pragma name）

【機能】

- オブジェクト・モジュール・ファイルのシンボル情報テーブルに，指定されたモジュール名の先頭から 254 文字を出力します。
- アセンブル・リスト・ファイルに -g2 指定時はシンボル情報 (MOD_NAM) として，-ng 指定時は NAME 疑似命令として，指定されたモジュール名の先頭から 254 文字を出力します。
- 255 文字以上のモジュール名が指定された場合は，ワーニング・メッセージを出力します。
- 許されない文字が記述された場合は，エラーとし，アボートします。
- この #pragma 指令が 1 ソース・ファイル中に複数存在する場合は，ワーニング・メッセージを出力し，後ろに記述した方を有効とします。

【効果】

- オブジェクトのモジュール名を任意の名前に変更することができます。

【方法】

- 記述方法は，次のとおりです。

```
#pragma name      モジュール名
```

モジュール名は，OS でファイル名として許す文字から“ (”，“) ”と漢字を除いたものとします。

大文字／小文字は区別します。

【使用例】

```
#pragma name    module1
    :
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- モジュール名変更機能をサポートしていなければ，修正の必要はありません。
- モジュール名を変更したい場合は，上記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- #pragma name …を削除，または #ifdef で切り分けます。
- モジュール名を変更する場合は，各コンパイラの仕様により，変更が必要です。

ローテート関数（#pragma rot）

【機能】

- オブジェクトに式の値をローテートするコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma の指令がない場合は，ローテート用の関数は通常の関数とみなされます。

【効果】

- C ソース，または ASM 記述により，ローテートを行う処理を記述しなくてもローテート機能を実現することができます。

【方法】

- 関数呼び出しと同様の形式で，ソース中に記述します。
  ローテート用の関数名は，次の 4 つです。

```
rorb , rolb , rorw , rolw
```

[ローテート用の関数一覧]

(1) unsigned char rorb (*x* , *y*) ;

unsigned char *x* ;
unsigned char *y* ;

x を *y* 回右ローテートします。

(2) unsigned char rolb (*x* , *y*) ;

unsigned char *x* ;
unsigned char *y* ;

x を *y* 回左ローテートします。

(3) unsigned int rorw (*x* , *y*) ;

unsigned int *x* ;
unsigned char *y* ;

x を *y* 回右ローテートします。

(4) unsigned int rolw (*x* , *y*) ;

unsigned int *x* ;
unsigned char *y* ;

x を *y* 回左ローテートします。

- モジュールの #pragma rot 指令により，ローテート用の関数の使用を宣言します。

  ただし，次の項目は #pragma rot の前に記述することができます。

  (1) コメント

  (2) 他の #pragma 指令

  (3) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma 以降に記述するキーワードは，大文字でも小文字でも可能です。

【使用例】

＜C ソース＞

```
#pragma rot
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned char   c ;

void    main ( ) {
        c = rorb ( a , b ) ;
}
```

＜出力アセンブラ・ソース＞

```
        mov     x , !_b
        mov     a , !_a
L0003 :
        ror     a , 1
        dec     x
        bnz     $L0003
```

【制限】

- 関数名として，ローテート用の関数名を使用することはできません。
- ローテート用の関数は，小文字で記述します。大文字は通常の関数扱いとなります。

【互換性】

〈他の C コンパイラから CC78K0R〉

- ローテート用の関数を使用していなければ，修正は必要ありません。
- ローテート用の関数に変更したい場合は，上記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- “#pragma rot”文を削除，または #ifdef で切り分けます。
- ローテート用の関数として使用する場合は，各コンパイラの仕様により，変更が必要です（#asm，#endasm，あるいは asm ( ) ; など）。

乗算関数（#pragma mul）

【機能】

- オブジェクトに式の値を乗算するコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma の指令がない場合は，乗算用の関数は通常の関数とみなされます。

【効果】

- 乗算命令の入出力のデータ・サイズを生かしたコードが生成されるため，通常の乗算式の記述より実行スピードが速く，かつサイズが小さいコードを生成することができます。

【方法】

- 関数呼び出しと同様の形式で，ソース中に記述します。

```
mulu
```

[乗算関数一覧]

(1) unsigned int mulu (*x* , *y*) ;

unsigned char *x* ;
unsigned char *y* ;

x と *y* を符号なし乗算します。

- モジュールの #pragma mul 指令により，乗算用の関数の使用を宣言します。
  ただし，次の項目は，#pragma mul の前に記述することができます。

  (1) コメント

  (2) 他の #pragma 指令

  (3) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma 以降に記述するキーワードは，大文字でも小文字でも可能です。

【制限】

- 関数名として，乗算用の関数名を使用することはできません（#pragma mul 宣言時）。
- 乗算用の関数は，小文字で記述します。大文字は通常の関数扱いとなります。

【使用例】

<C ソース>

```
#pragma mul
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned int    i ;

void    main ( )
{
        i = mulu ( a , b ) ;
}
```

<コンパイラの出力オブジェクト>

```
mov     x , !_b
mov     a , !_a
mulu    x
movw    !_i , ax
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 乗算用の関数を使用していなければ，修正は必要ありません。
- 乗算用の関数に変更したい場合は，前記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- “#pragma mul”文を削除，または #ifdef で切り分けます。関数名として，乗算用の関数名を使用できます。
- 乗算用の関数として使用する場合は，各コンパイラの仕様により，変更が必要です（#asm，#endasm あるいは asm ( ) ; など)。

除算関数（#pragma div）

【機能】

- オブジェクトに式の値を除算するコードを生成します。
- #pragma の指令がない場合は，除算用の関数は通常の関数とみなされます。

【効果】

- CC78K0 と互換性があり，除算命令の入出力のデータ・サイズを生かしたコードが生成されるため，通常の除算式の記述より，実行スピードが速く，かつサイズが小さいコードを生成することができます。

【方法】

- 関数呼び出しと同様の形式で，ソース中に記述します。除算用の関数名は，次の 2 つです。

```
divuw, moduw
```

[除算関数一覧]

(1) unsigned int divuw (*x* , *y*) ;

unsigned int *x* ;
unsigned char *y* ;

x と *y* を符号なし除算し，商を返します。

(2) unsigned char moduw (*x* , *y*) ;

unsigned int *x* ;
unsigned char *y* ;

x と *y* を符号なし除算し，余りを返します。

- モジュールの #pragma div 指令により，除算用の関数の使用を宣言します。
  ただし，次の項目は， #pragma div の前に記述することができます。
  (1) コメント
  (2) 他の #pragma 指令
  (3) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの
- #pragma 以降に記述するキーワードは，大文字でも小文字でも可能です。

【制限】

- インライン展開をせず，ライブラリ呼び出しとなります。
- 関数名として，除算用の関数名を使用することはできません。
- 除算用の関数は，小文字で記述します。大文字は通常の関数扱いとなります。

【使用例】

＜C ソース＞

```
#pragma div
unsigned int    a = 0x1234 ;
unsigned char   b = 0x12 ;
unsigned char   c ;
unsigned int    i ;

void    main ( ) {
        i = divuw ( a , b ) ;
        c = moduw ( a , b ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
mov     c , !_b
movw    ax , !_a
callt   [ @@divuw ]
movw    !_i , ax
mov     c , !_b
movw    ax , !_a
callt   [ @@divuw ]
mov     a , c
mov     !_c , a
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- 除算用の関数を使用していなければ，修正は必要はありません。
- 除算用の関数に変更したい場合は，前記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- “#pragma div”文を削除，または #ifdef で切り分けます。関数名として，除算用の関数名を使用することができます。
- 除算用の関数として使用する場合は，各コンパイラの仕様により，変更が必要です（#asm，#endasm あるいは asm ( ) ; など）。

データ挿入関数（#pragma opc）

【機能】

- カレント・アドレスに定数データを挿入します。
- pragma の指令がない場合は，データ挿入用の関数は通常の関数とみなされます。

【効果】

- ASM 文を使わなくても，特定のデータや命令をコード領域に埋め込めます。
  ASM 文を使った場合，アセンブラを通さないとオブジェクトを得られませんが，データ挿入関数を使用した場合，アセンブラを通さなくてもオブジェクトを得られます。

【方法】

- 関数呼び出しと同様の形式でソース中に大文字で記述します。
- データ挿入用の関数名は，__OPC です。

[データ挿入関数一覧]

(1) void __OPC (unsigned char *x* , …) ;

引数に記述した定数値をカレント・アドレスに挿入します。
引数は，定数しか記述することができません。

- #pragma opc 指令によりデータ挿入用の関数の使用を宣言します。
  ただし，次の項目は，#pragma opc の前に記述することができます。

(1) コメント

(2) 他の #pragma 指令

(3) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma 以降に記述するキーワードは，大文字でも小文字でも可能です。

【制限】

- 関数名として，データ挿入用の関数名が使用できません（#pragma opc 指定時）。
- __OPC は大文字で記述します。小文字は通常の関数扱いとなります。

【使用例】

＜C ソース＞

```
#pragma opc

void    main ( ) {
        __OPC ( 0xA7 ) ;
        __OPC ( 0x51 , 0x12 ) ;
        __OPC ( 0x30 , 0x34 , 0x12 ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
_main :
; line  4 : __OPC ( 0xA7 ) ;
        DB      0AFH
; line  5 : __OPC ( 0x51 , 0x12 ) ;
        DB      051H
        DB      012H
; line  6 : __OPC ( 0x30 , 0x34 , 0x12 ) ;
        DB      030H
        DB      034H
        DB      012H
; line  7 : }
        ret
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- データ挿入用の関数を使用していなければ，修正は必要ありません。
- データ挿入用の関数に変更したい場合は，上記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- “#pragma opc”文を削除，または #ifdef で切り分けます。関数名として，データ挿入用の関数名を使用できます。
- データ挿入用の関数として使用する場合は，各コンパイラの仕様により，変更が必要です（#asm，#endasm あるいは asm ( ) ; など）。

RTOS 用割り込みハンドラ（#pragma rtos_interrupt …）

【機能】

- #pragma rtos_interrupt 指令で指定された関数名を，78K0R シリーズ RTOS　RX78K0R 用割り込みハンドラと解釈します。
- 記述された関数名のアドレスを，指定された割り込み要求名に対する割り込みベクタ・テーブルに登録します。
- RTOS 用割り込みハンドラは，次の順番でコード生成を行います。

(1) 全レジスタの退避

(2) コンパイラが使用する saddr 領域の退避

(3) ES，CS レジスタの退避

(4) call !!addr20 命令によるカーネル・シンボル __kernel_int_entry の呼び出し

(5) ローカル変数領域の確保（ローカル変数があるときのみ）

(6) 関数本体

(7) ローカル変数領域の解放（ローカル変数があるときのみ）

(8) ラベル _ret_int に br !!addr20 命令で無条件分岐

エピローグは出力しません。

【効果】

- C ソース・レベルで，RTOS 用割り込みハンドラの記述が可能となります。
- 割り込み要求名を認識するため，ベクタ・テーブルのアドレスを意識する必要がありません。

【方法】

- 次の #pragma 指令により，割り込み要求名と関数名を指定します。
- なお，この #pragma 指令は，C ソースの先頭に記述します。
  #pragma PC（種別）を記述する場合は，それよりも後ろにこの #pragma 指令を記述します。

  次の項目は，この #pragma 指令の前に記述することができます。

(1) コメント

(2) プリプロセス指令のうち変数の定義／参照，関数の定義参照を生成しないもの

```
#pragma △ rtos_interrupt[ △割り込み要求名△関数名 ]
```

- #pragma 以降に記述するキーワードのうち，割り込み要求名は必ず大文字で記述してください。その他のキーワードは，大文字でも小文字でも可能です。

【制限】

- -zf 無指定時，RTOS 用割り込みハンドラはメモリ・モデルによらず，[C0H - 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- 割り込み要求名は，大文字で記述します。
- 割り込み要求名にソフトウエア割り込み，ノンマスカブル割り込みを指定することはできません。指定した場合は，エラーとします。
- 1 モジュール単位でのみ，割り込み要求名の重複チェックを行います。
- RTOS 用割り込みハンドラは， callt/__callt/norec/__leaf/__interrupt /__interrupt_brk/__flash/__flashf を指定することができません。
- __far は，-zf オプション指定時のみ指定可能です。
- 関数名として，ret_int/_kernel_int_entry を使用することはできません。

【使用例】

[スタック切り替え指定がない場合]

＜ C ソース＞

```
#pragma rtos_interrupt  INTP0   intp
int     i ;

void    intp ( ) {
        int     a [ 3 ] ;
        a [ 0 ] = 1 ;
}
```

<コンパイラの出力オブジェクト>

```
@@BASE          CSEG    BASE
_intp :
                push    ax              ; レジスタの退避
                push    bc              ;
                push    de              ;
                push    hl              ;
                movw    ax , _@RTARG0   ; コンパイラが使用する saddr 領域の退避
                push    ax              ;
                movw    ax , _@RTARG2   ;
                push    ax              ;
                movw    ax , _@RTARG4   ;
                push    ax              ;
                movw    ax , _@RTARG6   ;
                push    ax              ;
                movw    ax , _@SEGAX    ;
                push    ax              ;
                movw    ax , _@SEGDE    ;
                push    ax              ;
                mov     a , ES          ;
                mov     x , a           ;
                mov     a , CS          ;
                push    ax              ;
                call    !!__kernel_int_entry
                subw    sp , #06H
                movw    hl , sp
; line 5 :              int     a [ 3 ] ;
; line 6 :              a [ 0 ] = 1;
                onew    ax
                movw    [ hl ] , ax     ; a
; line 7 :      }
                addw    sp , #06H       ; ローカル変数の領域解放
                br      !!_ret_int
@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- RTOS 用割り込みハンドラをサポートしていなければ，修正は必要ありません。
- RTOS 用割り込みハンドラに変更したい場合は，上記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- #pragma rtos_interrupt 指定を削除すれば，通常の関数として扱われます。
- RTOS 用割り込みハンドラとして使用する場合は，各コンパイラの仕様により，変更が必要です。

RTOS 用割り込みハンドラ修飾子（__rtos_interrupt）

【機能】

- __rtos_interrupt 修飾子で宣言された関数は，RTOS 用割り込みハンドラと解釈します。RTOS 用割り込みハンドラでのレジスタ，saddr の退避／復帰については，「RTOS 用割り込みハンドラ（#pragma rtos_interrupt …）」を参照してください。

【効果】

- ベクタ・テーブルの設定と RTOS 用割り込みハンドラ関数定義を，別ファイルに記述することができます。

【方法】

- RTOS 用割り込みハンドラの修飾子に，__rtos_interrupt を付加します。

```
__rtos_interrupt          void     func ( )           { 処理 }
```

【制限】

- -zf 無指定時，RTOS 用割り込みハンドラはメモリ・モデルによらず，[C0H - 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- RTOS 用割り込みハンドラは，callt/__callt/norec/__leaf/__interrupt/__interrupt_brk/__flash/__flashf を指定することができません。
- __far は，-zf オプション指定時のみ指定可能です。
- 関数名として，ret_int/__kernel_int_entry を使用することはできません。

【注意】

- この修飾子の宣言だけでは，ベクタ・アドレスの設定を行いません。
  ベクタ・アドレスの設定は，#pragma 指令あるいはアセンブラ記述などにより，別途行う必要があります。
- #pragma rtos_interrupt …の指定と同一ファイルに RTOS 用割り込みハンドラを定義する場合は，この修飾子を記述しなくても，#pragma rtos_interrupt で指定された関数名を RTOS 用割り込みハンドラと判断します。

【互換性】

〈他の C コンパイラから CC78K0R〉

- RTOS 用割り込みハンドラをサポートしていなければ，修正は必要ありません。
- RTOS 用割り込みハンドラに変更したい場合は，上記の方法に従って変更を行います。

〈CC78K0R から他の C コンパイラ〉

- #define により可能です（「11.6 C ソースの修正」を参照してください）。
  この変更により，通常の関数として扱われます。
- RTOS 用割り込みハンドラとして使用する場合は，各コンパイラの仕様により変更が必要となります。

RTOS 用タスク関数（#pragma rtos_task）

【機能】

- #pragma rtos_task で指定された関数名を，RTOS 用のタスクと解釈します。
- 関数名指定がある場合，その実体定義が同一ファイル中にない場合はエラーとなります。
- RTOS 用タスク関数の前処理では，フレーム・ポインタ／レジスタ変数用レジスタの退避は行いません。また，後処理を出力しません。
- ext_tsk の RTOS システム・コール呼び出し関数は，call !!addr20 命令で呼びます。ext_tsk が発行されている場合は，後処理を出力しません。
- ext_tsk が RTOS 用タスク関数中に 1 つも存在しない場合で，-w2 オプションが指定されている場合は，注意をうながすワーニング・メッセージを出力します。

【効果】

- C ソース・レベルで，RTOS 用タスク関数を記述することができます。
- フレーム・ポインタ／レジスタ変数用レジスタの退避，および後処理が出力されなくなるため，コード効率が良くなります。

【方法】

- 次の #pragma 指令に，関数名を指定します。

```
#pragma △ rtos_task[ △タスク関数名 ]
```

なお，この #pragma 指令は C ソースの先頭に記述します。

ただし，次の項目は，この #pragma 指令の前に記述することができます。

(1) コメント

(2) プリプロセス指令のうち変数の定義／参照，関数の定義参照を生成しないもの

- #pragma 以降に記述するキーワードは，大文字でも小文字でも記述可能です。

【制限】

- RTOS 用タスク関数は，callt/__callt/norec/__leaf/__interrupt/__interrupt_brk/__flash/__flashf を指定することができません。
- __far は，-zf オプション指定時のみ指定可能です。
- RTOS 用タスク関数を通常の関数のように呼び出すことはできません。

【使用例】

＜C ソース＞

```
#pragma rtos_task       func
int     i ;
void    main ( )
{
        regiser int     a ;
        int             x ;
        x = 1 ;
        r = 2 ;
}
void    func ( )
{
        register int    r ;
        int             x ;

        x = 1 ;
        r = 2 ;
        ext_tsk ( ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
@@CODEL CSEG
_main :
        push    hl              ; フレーム・ポインタを退避する
        subw    sp , #04H
        movw    hl , sp
        onew    ax
        movw    [ hl ] , ax     ; x
        incw    ax
        movw    [ hl + 2 ] , ax ; r
        addw    sp , #04H       ; エピローグを出力する
        pop     hl
        ret
_func :
        subw    sp , #04H       ; フレーム・ポインタは退避されない
        movw    hl , sp
        onew    ax
        movw    [ hl ] , ax     ; x
        incw    ax
        movw    [ hl + 2 ] , ax ; r
        call    !!_ext_tsk      ; エピローグは出力されない
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- RTOS 用タスク関数をサポートしていなければ，修正は必要ありません。
- RTOS 用タスク関数に変更したい場合は，上記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- #pragma rtos_task 指定を削除すれば通常の関数として扱われます。
- RTOS 用タスク関数として使用する場合は，各コンパイラの仕様により，変更が必要です。

フラッシュ領域配置方法（-zf）

注意　この機能は，デバイスのフラッシュ・メモリ書き換え機能を有効にします。

【機能】

- フラッシュ領域に配置するオブジェクト・ファイルを生成します。
- ブート領域からは，フラッシュ領域の外部変数を参照することはできません。
- フラッシュ領域からは，ブート領域の外部変数を参照することができます。
- ブート領域のプログラムとフラッシュ領域のプログラムでは，同じ外部変数，および同じグローバル関数を定義することはできません。

【効果】

- プログラムをフラッシュ領域に配置することができるようになります。
- -zf オプションを指定せずに作成したブート領域のオブジェクトと結合して使用できるようになります。

【方法】

- コンパイル時に -zf オプションを指定します。

【制限】

- スタートアップ・ルーチン，ライブラリはフラッシュ領域用のものを使用してください。

フラッシュ領域分岐テーブル（#pragma ext_table）

注意　この機能は，デバイスのフラッシュ・メモリ書き換え機能を有効にします。

【機能】

- スタートアップ・ルーチンへの分岐テーブル，割り込み関数への分岐テーブル，およびブート領域からフラッシュ領域への関数呼び出しのための分岐テーブルの先頭アドレスを決定します。
- 分岐テーブルの先頭から 64 個分は，割り込み関数専用（スタートアップ・ルーチンを含む）とし，それぞれ 4 バイトの領域を占有します。通常関数の分岐テーブルは「分岐テーブルの先頭アドレス + 4 * 64」以降に配置し，それぞれ 4 バイトの領域を占有します。

【効果】

- スタートアップ・ルーチン，割り込み関数をフラッシュ領域に配置することができます。
- ブート領域からフラッシュ領域への呼び出しを行うことができます。

【方法】

- 次の #pragma 指令により，フラッシュ領域分岐テーブルの先頭アドレスを指定します。

```
#pragma ext_table          分岐テーブルの先頭アドレス
```

なお，#pragma 指令は，C ソースの先頭に記述してください。

- 次の項目は，#pragma 指令の前に記述しても問題ありません。

(1) コメント

(2) #pragma ext_func，-zf 指定時の #pragma vect，#pragma interrupt，または #pragma rtos_interrupt 以外の #pragma 指令

(3) プリプロセス指令のうち，変数の定義／参照，関数の定義／参照を生成しないもの

【制限】

- 分岐テーブルは，フラッシュ領域の先頭アドレスに配置します。
- #pragma ext_func，-zf 指定時の #pragma vect，#pragma interrupt，または #pragma rtos_interrupt の前に #pragma ext_table がない場合，エラーとなります。
- 分岐テーブルの先頭アドレス値は，0xc0 - 0xff00 とします。
- 指定した分岐テーブルの先頭アドレス値に従って，割り込みベクタ用ライブラリ（_@vect00 ～ _@vect7e）を再構築する必要があります。割り込みベクタ用ライブラリ中の分岐テーブルの先頭アドレス値のデフォルトは，2000H です。分岐テーブルの先頭アドレスに 2000H 以外を指定する場合は，次のようにライブラリを再構築してください。

(1) ¥Program Files¥NEC Electronics Tools¥CC78K0R¥V*x.xx*¥src¥cc78k0r¥src ディレクトリ中の vect.inc の

    ITBLTOP EQU 2000H

の H の箇所を，指定したアドレスに変更する。

(2) ¥Program Files¥NEC Electronics Tools¥CC78K0R¥V*x.xx*¥src¥cc78k0r¥bat¥repvect.bat

をコマンド・プロンプト上で起動してアセンブラなどによりライブラリを更新し，¥Program Files¥NEC Electronics Tools¥CC78K0R¥V*x.xx*¥src¥cc78k0r¥lib の更新されたライブラリを ¥Program Files¥NEC Electronics Tools¥CC78K0R¥V*x.xx*¥lib78k0r にコピーしてリンク用に使用します。

注意　上記ディレクトリは，インストール方法により異なります。

【互換性】

〈他の C コンパイラから CC78K0R〉

- #pragma ext_table を使用していなければ，修正は必要ありません。
- フラッシュ領域分岐テーブルの先頭アドレスを指定したい場合は，上記の方法に従って変更します。

〈CC78K0R から他の C コンパイラ〉

- #pragma ext_table 指令を削除，または #ifdef で切り分けます。
- フラッシュ領域分岐テーブルの先頭アドレスを指定する場合は，次のように変更が必要です。

【使用例】

[分岐テーブルを 2000H 番地以降に生成し，割り込み関数を配置する場合]

＜ C ソース＞

```
#pragma ext_table       0x2000
#pragma interrupt       INTP0   intp

void    intp ( )
{
}
```

(1) 割り込み関数をブート領域に配置する場合（-zf 指定なし）

＜出力コード＞

```
                PUBLIC  _intp
                PUBLIC  _@vect06
@@BASE          CSEG    BASE
_intp :
                reti

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

- 割り込みベクタ・テーブルに，割り込み関数の先頭アドレスを設定します。

(2) 割り込み関数をフラッシュ領域に配置する場合（-zf 指定あり）

＜出力コード＞

```
                PUBLIC  _intp
@ECODE          CSEG    BASE
_intp :
                reti

@EVECT06        CSEG    AT      0200CH
                br      !!_intp
```

- 分岐テーブルに，割り込み関数の先頭アドレスを設定します。
- 分岐テーブルの先頭アドレスが 2000H，割り込みベクタ・アドレス（2 バイト）が 0006H なので，分岐テーブルのアドレス値は，2000H + 4 *（0006H / 2）番地となります。
- 割り込みベクタ・テーブルへの 200CH 番地の設定は，割り込みベクタ用ライブラリが行います。

＜割り込みベクタ 06 用ライブラリ＞

```
                PUBLIC  _@vect06

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      200CH
```

ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）

注意　この機能は，デバイスのフラッシュ・メモリ書き換え機能を有効にします。

【機能】

- ブート領域からフラッシュ領域への関数呼び出しを，フラッシュ領域分岐テーブルを介して行います。
- フラッシュ領域からは，ブート領域中の関数を直接呼び出せます。

【効果】

- ブート領域からフラッシュ領域中の関数を呼び出すことができるようになります。

【方法】

- 次の #pragma 指令により，ブート領域から呼び出すフラッシュ領域中の関数名，および ID 値を指定します。

```
#pragma ext_func          関数名   ID 値
```

なお，この #pragma 指令は，C ソースの先頭に記述します。

次の項目は，この #pragma 指令の前に記述されても問題ありません。

(1) コメント

(2) プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの。

【制限】

- ID 値は，0 〜 255 (0xFF) とします。
- #pragma ext_func の前に，#pragma ext_table がない場合，エラーとなります。
- 同じ関数名で ID 値が異なる場合，および異なる関数名で ID 値が同じ場合は，エラーとなります。
  次の (1)，(2) は，エラーとなります。

(1) #pragma ext_func f1 3
    #pragma ext_func f1 4

(2) #pragma ext_func f1 3
    #pragma ext_func f2 3

- ブート領域からフラッシュ領域へ関数呼び出しを行い，フラッシュ領域に対応する関数定義がない場合，リンカはチェックすることができません。ユーザ責任となります。
- callt 関数は，ブート領域内のみ配置可能とし，フラッシュ領域（-zf オプション指定時）に callt 関数を定義したときは，エラーとなります。

【互換性】

〈他の C コンパイラから CC78K0R〉

- #pragma ext_func を使用していなければ，修正は必要ありません。
- ブート領域からフラッシュ領域への関数呼び出しを行いたい場合は，上記の方法に従って変更します。

〈CC78K0R から他の C コンパイラ〉

- #pragma ext_func 指令を削除，または #ifdef で切り分けます。
- ブート領域からフラッシュ領域への関数呼び出しを行う場合は，次のような変更が必要です。

【使用例】

- 分岐テーブルを 2000H 番地以降に生成し，フラッシュ領域中の関数 f1，f2 をブート領域から呼び出す場合

＜C ソース＞

```
（1） ブート領域側

#pragma ext_table       0x2000
#pragma ext_func        f1      3
#pragma ext_func        f2      4

extern  void    f1 ( void ) ;
extern  void    f2 ( void ) ;

void    func ( )
{
        f1 ( ) ;
        f2 ( ) ;
}

（2） フラッシュ領域側

#pragma ext_table       0x2000
#pragma ext_func        f1      3
#pragma ext_func        f2      4

void    f1 ( )
{
}
void    f2 ( )
{
}
```

備考 1 #pragma ext_func f1 3 は，関数 f1 への飛び先を分岐テーブルの 2000H + 4 * 64 + 4 * 3 番地に配置することを意味します。

備考 2 #pragma ext_func f2 4 は，関数 f2 への飛び先を分岐テーブルの 2000H + 4 * 64 + 4 * 4 番地に配置することを意味します。

備考 3 分岐テーブルの先頭から 4 * 64 バイトは，割り込み関数専用（スタートアップ・ルーチンを含む）です。

＜コンパイラの出力オブジェクト＞

（1） ブート領域側（-zf 指定なし）

```
@@CODEL CSEG
_func :
        call    !0210CH
        call    !02110H
        ret
```

（2） フラッシュ領域側（-zf 指定あり）

```
@ECODEL CSEG
_f1 :
        ret
_f2 :
        ret

@EXT03  CSEG    AT      0210CH
        br      !!_f1
        br      !!_f2
```

ファーム ROM 関数（__flash）

注意　この機能は，デバイスのフラッシュ・メモリ書き換え機能を有効にします。

【機能】

- ファーム ROM 関数と C 言語の関数の間に位置するインタフェース・ライブラリを介して，フラッシュのセルフ書き込みを行うファーム ROM 関数を呼び出します。
- インタフェース・ライブラリ呼び出しのインタフェースは，第一引数がレジスタで，第二引数以降がスタック渡しになります。第一引数のレジスタは次のとおりです。

1，2 バイト・データ	：	AX
4 バイト・データ	：	AX（下位），BC（上位）

- 返り値のサイズに応じて，インタフェース・ライブラリは，次に示すレジスタに返り値を設定する必要があります。

1，2 バイト・データ	：	BC
near ポインタ	：	BC
4 バイト・データ，far ポインタ	：	BC（下位），DE（上位）

【効果】

- ファーム ROM 関数に関する操作を C ソース・レベルで記述することができます。

【方法】

- インタフェース・ライブラリのプロトタイプ宣言時に，__flash 属性を先頭に追加します。

【制限】

- 関数ポインタによる関数呼び出しは，サポートしません。
- __flash 付きの関数本体を定義したときは，エラーとなります。

【互換性】

〈他の C コンパイラからこの C コンパイラ〉

- 予約語 __flash を使用していなければ，修正は必要ありません。
- ファーム ROM 関数に変更したい場合は，上記の方法に従って変更します。

〈この C コンパイラから他の C コンパイラ〉

- #define により可能（「11.6 C ソースの修正」参照）です。
- ファーム ROM 関数あるいはそれに代わる機能のある CPU において，その領域をアクセスするにはユーザが専用のライブラリを作成する必要があります。

引数／返り値の int 拡張抑制方法（-zb）

【機能】

- 関数返り値の型定義が char/unsigned char 型の場合に，返り値の int 拡張コードを生成しません。
- 関数引数のプロトタイプが定義されていて，かつそのプロトタイプの引数定義が char/unsigned char 型の場合に，引数の int 拡張コードを生成しません。

【効果】

- int 拡張コードが生成されないため，オブジェクト・コードの短縮，実行速度の向上を図ることができます。

【方法】

- コンパイル時に -zb オプションを指定します。

【使用例】

＜C ソース＞

```
unsigned char   func1 ( unsigned char x , unsigned char y ) ;
unsigned char   c , d , e ;
void    main ( )
{
        c = func1 ( d , e ) ;
        c = func2 ( d , e ) ;
}
unsigned char   func1 ( unsigned char x , unsigned char y )
{
        return x + y ;
}
```

[-zb 指定あり]

＜コンパイラの出力オブジェクト＞

```
_main :
; line  5 :             c = func1 ( d , e ) ;
        mov     x , !_e
        push    ax
        mov     x , !_d                 ; int 拡張しない
        call    !_func1
        pop     ax
        mov     a , c
        mov     !_c , a
; line  6 :             c = func2 ( d , e ) ;
        mov     x , !_e
        clrb    a                       ; プロトタイプ宣言がないので int 拡張する
        push    ax
        mov     x , !_d
        mov     x , #00H
        xch     a , x                   ; プロトタイプ宣言がないので int 拡張する
        call    !_func2
        pop     ax
        mov     a , c
        mov     !_c , a
; line 7 :      }
        ret
; line  8 :     unsigned char   func1 ( unsigned char x , unsigned char y )
; line 9 :      {
_func1 :
        push    hl
        push    ax
        movw    ax , sp
        movw    hl , ax
        mov     a , [ hl ]
        mov     x , a
        mov     a , [ hl + 6 ]
        movw    hl , ax
; line  10 :            return x + y ;
        mov     a , l
        add     a , h
        mov     c , a                   ; int 拡張しない
; line 11 :     }
        pop     ax
        pop     hl
        ret
```

【制限】

- 関数本体の定義とその関数に対するプロトタイプ宣言がファイル間で異なる場合，不正動作となる場合があります。

【互換性】

〈他の C コンパイラから CC78K0R〉

- すべての関数本体の定義に対するプロトタイプ宣言が正しく行われていない場合は，プロトタイプ宣言を正しく行います。あるいは，-zb オプションを指定しません。

〈CC78K0R から他の C コンパイラ〉

- 修正は必要ありません。

メモリ操作関数（#pragma inline）

【機能】

- メモリ操作用標準ライブラリ関数 memcpy，memset を，関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma 指令がない場合は，標準ライブラリ関数を呼び出すコードを生成します。

【効果】

- 標準ライブラリ関数呼び出し時と比べて，実行速度の向上を図ることができます。
- 指定文字数に定数を指定した場合は，オブジェクト・コードの短縮も図ることができます。

【方法】

- 関数呼び出しと同様の形式で，ソース中に記述します。
- 次の項目は，#pragma inline の前に記述することができます。

(1)　コメント

(2)　他の #pragma 指令

(3)　プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

【使用例】

＜C ソース＞

```
#pragma inline
char    ary1 [ 100 ] , ary2 [ 100 ] ;
void    main ( )
{
        memset ( ary1 , 'A' , 50 ) ;
        memcpy ( ary1 , ary2 , 50 ) ;
}
```

＜コンパイラの出力オブジェクト＞

```
_main :
        push    hl
; line  5 :     memset ( ary1 , 'A' , 50 ) ;
        movw    de , #loww ( _ary1 )
        mov     a , #041H       ; 65
        mov     c , #032H       ; 50
L0003 :
        mov     [ de ] , a
        incw    de
        dec     c
        bnz     $L0003
; line  6 :     memcpy ( ary1 , ary2 , 50 ) ;
        movw    de , #loww ( _ary1 )
        movw    hl , #loww ( _ary2 )
        mov     c , #032H       ; 50
L0005 :
        mov     a , [ hl ]
        mov     [ de ] , a
        incw    de
        incw    hl
        dec     c
        bnz     $L0005
; line  7 : }
        pop     hl
        ret
```

【互換性】

〈他の C コンパイラから CC78K0R〉

- メモリ操作用の関数を使用していなければ，修正は必要ありません。
- メモリ操作用の関数に変更したい場合は，上記の方法に従い変更します。

〈CC78K0R から他の C コンパイラ〉

- “#pragma inline”指令を削除，または #ifdef で切り分けます。

絶対番地配置指定（__directmap）

【機能】

- __directmap 宣言された外部変数，および関数内 static 変数の初期値を，配置アドレス指定とみなして，指定アドレスに変数を配置します。
- C ソース中における __directmap 変数は，static 変数と同様に扱います。
- 初期値を配置アドレス指定とみなすため，初期値を定義することができず，初期値は不定となります。
- 指定可能なアドレス指定範囲，指定アドレスに対する領域確保用モジュールがリンクされる領域確保範囲，および変数の重複チェック範囲は，次のとおりです。

項目	範囲	
	スモール・モデル， ミディアム・モデルの場合	ラージ・モデル， コンパクト・モデルの場合
アドレス指定範囲	0xf0000 - 0xfffff	0x00000 - 0xfffff
領域確保範囲	0xffd00 - 0xffeff	0xffd00 - 0xffeff
重複チェック範囲	デバイスの内蔵 RAM の スタート・アドレス - エンド・アドレス	デバイスの内蔵 RAM の スタート・アドレス - エンド・アドレス

- アドレス指定がアドレス指定範囲外の場合は，エラーを出力します。
- __directmap 宣言された変数の配置アドレスが重複し，重複チェック範囲内であれば，ワーニング・メッセージ（W0762）を出力して，重なった変数名を表示します。
- アドレス指定範囲が saddr 領域内の場合は，__sreg 宣言を自動的に付与し，saddr 命令を生成します。
- __directmap 宣言された char/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long 型変数に対してビット参照を行う場合は，sreg/__sreg を併用する必要があります。併用しない場合は，エラーとします。
- アドレス指定範囲が near 領域内である場合は，near 領域の変数としてアクセスします。
- アドレス指定範囲が saddr 領域でも near 領域でもない場合は，far 領域の変数としてアクセスします。
- __near，__far 型修飾子を指定しない場合は，メモリ・モデルに従ったアクセスをします。
- 型修飾子の指定がある場合，その指定に従ったアクセスをします。ただし，アドレス指定範囲と型修飾子に矛盾がある場合はエラーとします。
  アドレス指定範囲，メモリ・モデル，型修飾子指定の関係をまとめると，次のとおりとなります。

アドレス指定範囲		型修飾子					
		__near __sreg	__far __sreg	__sreg	__near	__far	無指定
saddr 領域内	アクセス方法	sreg	sreg	sreg	sreg	sreg	sreg
	ポインタ長	2 バイト	4 バイト	スモール : 2 バイト ミディアム : 2 バイト コンパクト : 4 バイト ラージ : 4 バイト	2 バイト	4 バイト	スモール : 2 バイト ミディアム : 2 バイト コンパクト : 4 バイト ラージ : 4 バイト
near 領域内	アクセス方法	エラー	エラー	エラー	near	far	スモール : near ミディアム : near コンパクト : far ラージ : far
	ポインタ長				2 バイト	4 バイト	スモール : 2 バイト ミディアム : 2 バイト コンパクト : 4 バイト ラージ : 4 バイト
far 領域内	アクセス方法	エラー	エラー	エラー	エラー	far	スモール : エラー ミディアム : エラー コンパクト : far ラージ : far
	ポインタ長					4 バイト	スモール : エラー ミディアム : エラー コンパクト : 4 バイト ラージ : 4 バイト

【効果】

- 任意のアドレスに変数を配置することができ，同じアドレスに複数の変数を重ねて配置することができます。

【方法】

- 絶対番地に配置する変数を定義したいモジュール中で，__directmap 宣言を行います。

```
__directmap                 型名    変数名 = 配置アドレス指定 ;
__directmap static          型名    変数名 = 配置アドレス指定 ;
__directmap __sreg          型名    変数名 = 配置アドレス指定 ;
__directmap __sreg static   型名    変数名 = 配置アドレス指定 ;
```

- 構造体／共用体／配列に対して __directmap 宣言を行う場合は，{ } で囲んでアドレス指定を行います。

【使用例】

<C ソース>

```
__directmap     char    c = 0xffe00 ;
__directmap     __sreg  char    d = 0xffe20 ;
__directmap     __sreg  char    e = 0xffe21 ;
__directmap     struct  x {
        char    a ;
        char    b ;
} xx = { 0xffe30 } ;

void    main ( ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

<出力オブジェクト>

```
        PUBLIC  _main
_c      EQU     0FFE00H                 ; __directmap 宣言された変数は
_d      EQU     0FFE20H                 ; EQU でアドレスを定義
_e      EQU     0FFE21H                 ;
_xx     EQU     0FFE30H                 ;
        EXTRN   __mmfe00                ; 領域確保モジュール・リンク用
        EXTRN   __mmfe20                ; EXTRN 出力
        EXTRN   __mmfe21                ;
        EXTRN   __mmfe30                ;
        EXTRN   __mmfe31                ;
@@CODEL CSEG
_main :
; line 10 :
        oneb    !loww ( _c )
; line 11 :
        mov     _d , #012H              ; アドレス指定が saddr 領域内のため,
; line 12 :                             ; saddr 命令を出力
        set1    _e.5                    ; __sreg と併用しているため, ビット操作可能
; line 13 :
        mov     _xx , #05H              ; アドレス指定が saddr 領域内のため,
                                        ; saddr 命令を出力
; line 14 :
        mov     _xx + 1 , #0AH          ; アドレス指定が saddr 領域内のため,
                                        ; saddr 命令を出力
; line 15 :
        ret
        END
```

【制限】

- 関数引数，返り値，およびオートマティック変数には指定することができません。指定した場合は，エラーとなります。
- 領域確保範囲外のアドレス指定を行った場合，変数領域は確保されないので，ディレクティブ・ファイルを記述するか，領域確保用モジュールを別途作成する必要があります。
- __directmap 変数は，static 変数と同様に扱うため，extern 宣言することはできません。

【互換性】

〈他の C コンパイラから CC78K0R〉

- キーワード __directmap を使用していなければ，修正の必要はありません。
- __directmap 変数に変更したい場合は，前記の方法に従い変更を行います。

〈CC78K0R から他の C コンパイラ〉

- #define により可能です（「11.6 C ソースの修正」を参照してください）。
- 絶対番地配置指定として使用する場合は，各コンパイラの仕様により変更が必要です。

near/far 領域指定

【機能】

- __near/__far 型修飾子を指定することによって，関数，変数の配置場所を明示的に指定します。

修飾子	配置場所
__near 型修飾子	near 領域 （データ：0F0000H - 0FFFFFH，コード：000000H - 00FFFFH）
__far 型修飾子	far 領域 （000000H - 0FFFFFH）

- near 領域へのポインタは 2 バイト，far 領域へのポインタは 4 バイトとします。
- 同じ変数，関数の宣言において，__near/__far 型修飾子が混在する場合は，エラーとします。
- __near/__far 型修飾子は，文法的に型修飾子と同様に扱います。
- __near/__far 型修飾子を __callt/__interrupt/__rtos_interrupt/__interrupt_brk/__sreg/__boolean と同時に指定した場合，__near/__far 型修飾子は無視されます。
- __near/__far 型修飾子が同時に指定された場合は，エラーとなります。
- __near/__far 型修飾子をオートマティック変数，引数，レジスタ変数に指定した場合，__near/__far 型修飾子は無視されます。
- near 領域の変数は，ES レジスタを用いずにアクセスします。
  ポインタ長は，2 バイトです。
- far 領域の変数は，ES レジスタを設定してアクセスします。
  ポインタ長は，4 バイトです。
- near 領域の関数は !addr16 で，far 領域の関数は !!addr20 で関数を呼び出します。
- CS レジスタを参照せずに関数ポインタ呼び出しをする命令がないため，関数ポインタ呼び出し時は，常に CS レジスタを設定します。
- near 領域の関数への関数ポインタは，CS レジスタに 0 を設定するコードを出力します。
- far ポインタの最上位 1 バイトは，不定とします。
- near ポインタ，または int から far ポインタへのキャスト，near ポインタから long へのキャストは，以下の動作となります。
  (1) 変数ポインタは，上位に 0xf を追加します（0 は例外で，0 拡張します）。
  (2) 関数ポインタは，0 拡張します。
- far ポインタの加減算は，下位 2 バイトで行い，上位は変化しません。
- ptrdiff_t は，常に int 型とします。
- far ポインタの等値演算は，下位 3 バイトで行います。
- far ポインタの関係演算は，下位 2 バイトで行います。同一オブジェクトを指していないポインタを比較する場合は，unsigned long にキャストする必要があります。また，-za オプション指定時は，下位 3 バイトで比較を行います。

- 文字列定数は，メモリ・モデルにより，far 領域，または near 領域に配置します。

メモリ・モデル	配置場所
スモール・モデル	near 領域
ミディアム・モデル	near 領域
コンパクト・モデル	far 領域
ラージ・モデル	far 領域

- ラージ・モデル，コンパクト・モデルの場合，オート変数，引数，sreg 変数へのポインタは 4 バイトです。

【効果】

- __far 型修飾子を指定することによって，far 領域への配置，および参照を可能とします。
- __near 型修飾子を指定することによって，near 領域への配置，および参照を可能とします。
  near 領域に配置することで，短い命令で参照，および関数呼び出しが可能となります。

【方法】

- 関数，変数の宣言時に，__near/__far 型修飾子を追加します。

【使用例】

```
__near int i1;
__far int i2;
__far int * __near p1;
__far int * __near * __far p2;
__far int func1();
__far int * __near func2();
__near int ( * __far fp1 )();
__far int * __near ( * __near fp2 )();
__near int * __far ( * __near fp3 )();
__near int * __near ( * __far fp4 )();
```

- i1 は int 型で，near 領域に配置
- i2 は int 型で，far 領域に配置
- p1 は“far 領域にある int 型”を指す 4 バイト型の変数で，変数自身は near 領域に配置
- p2 は“near 領域にある，“far 領域にある int 型”を指す 4 バイト型”を指す 2 バイト型の変数で，変数自身は far 領域に配置
- func1 は「int 型」を返す関数で，関数自身は far 領域に配置
- func2 は「“far 領域にある int 型”を指す 4 バイト型」を返す関数で，関数自身は near 領域に配置
- fp1 は“「int 型」を返す near 領域にある関数”を指す 2 バイト型の変数で，変数自身は far 領域に配置
- fp2 は“「“far 領域にある int 型”を指す 4 バイト型」を返す near 領域にある関数”を 指す 2 バイト型の変数で，変数自身は near 領域に配置
- fp3 は“「“near 領域にある int 型”を指す 2 バイト型」を返す far 領域にある関数”を 指す 4 バイト型の変数で，変数自身は near 領域に配置
- fp4 は“「“near 領域にある int 型”を指す 2 バイト型」を返す near 領域にある関数”を 指す 2 バイト型の変数で，変数自身は far 領域に配置

【制限】

- __far 型修飾子を指定しても，64K バイトをまたがってデータを配置することはできません。
  関数は，64K バイトをまたがっても配置可能です。

【互換性】

〈他の C コンパイラから CC78K0R〉

- 予約語 __near/__far を使用していなければ，修正する必要はありません。

〈CC78K0R から他の C コンパイラ〉

- __near/__far 型修飾子を使用していなければ，修正する必要はありません。
- __near/__far 型修飾子を使用している場合は，#define により可能です。

【注意】

- 関係演算を下位 2 バイトで行う場合，64K バイト境界の領域の最後の 1 バイトにデータを配置することはできません。配置した場合，リンカ，またはコンパイラでエラーとなります。

  これは，配列の範囲を越えた部分を指すポインタの関係演算に対し，ANSI 準拠の動作[注]を行うためです。

  注　ANSI の関係演算子の制約

  式 P が配列オブジェクトの要素を指しており，式 Q が同じ配列オブジェクトの最後の要素を指している場合，ポインタ式 Q+1 は，P と比較してより大きいとします。

- far 領域へのポインタのサイズは 4 バイトですが，演算対象は下位 3 バイトのみであり，最上位 1 バイトは不定となります。

  ＜例＞

```
union   tag {
        __far unsigned short    *ptr;
        unsigned long           ldata;
} un;
```

  un.ptr に値を書き込んだ後に un.ldata を参照すると，最上位 1 バイトは不定となります。un.ldata の最上位 1 バイトが 0 であることを保証するためには，最初に共用体 un を 0 で初期化する必要があります。

- リンカは，以下のセグメント・タイプと再配置属性の組み合わせを持つセクションのデータ配置のチェックを行います。

  DSEG UNIT64KP
  DSEG PAGE64KP
  CSEG PAGE64KP

- #pragma section，またはリンク・ディレクティブ・ファイルでこれらの再配置属性を変更した場合，リンカでチェックは行いません。

メモリ・モデル指定

【機能】

- 関数，変数の配置場所を指定します。

メモリ・モデル	データ	関数
スモール・モデル	near 領域	near 領域
ミディアム・モデル	near 領域	far 領域
コンパクト・モデル	far 領域	near 領域
ラージ・モデル	far 領域	far 領域

- __near/__far 型修飾子が指定された場合は，__near/__far 型修飾子を優先します。

- スモール・モデル

  データ 64K バイト，コード 64K バイトの合計 128K バイトとなります。

  データ ROM は，0000H - 0FFFFH，もしくは 10000H - 1FFFFH に配置し，FxxxxH にミラーリングされます。

  コードは，00000H - 0FFFFH に配置します。

  __far 型修飾子により CS レジスタの値が変更されている可能性があるので，関数ポインタ呼び出し時は，常に CS レジスタを設定します。

- ミディアム・モデル

  変数は near 領域に，関数は far 領域に配置します。つまり，データ 64K バイト，コード 1M バイトとなります。

  データ ROM は 000000H - 00FFFFH，もしくは 010000H - 01FFFFH に配置し，FxxxxH にミラーリングされます。コードの配置に制限はありません。

- コンパクト・モデル

  変数は far 領域に，関数は near 領域に配置します。つまり，データ 1M バイト，コード 64K バイトとなります。

  データの配置に制限はありません。コードは 000000H - 00FFFFH に配置します。

- ラージ・モデル

  変数，関数ともに far 領域に配置します。つまり，データ 1M バイト，コード 1M バイトとなります。

  データ，コードの配置に制限はありません。

【方法】

- コンパイル時に，-ms/-mm/-mc/-ml オプションを指定します。

オプション	説明
-ms	スモール・モデル
-mm	ミディアム・モデル
-mc	コンパクト・モデル
-ml	ラージ・モデル

【使用例】

＜C ソース＞

```
int     i ;
int     *p ;
void    func( void ){}

void    ( *fp )( void );

void    main( void ) {
        int     r;
        r = i ;         /* データアクセス */
        func() ;        /* 関数呼び出し */
        r = *p ;        /* データポインタ */
        fp() ;          /* 関数ポインタ */
}
```

＜コンパイラの出力オブジェクト：スモール・モデルの場合＞

```
        movw    hl , !_i
        call    !_func
        movw    de , !_p
        movw    ax , [ de ]
        movw    hl , ax
        movw    ax , !_fp
        mov     CS , #00H       ; 0
        call    ax
```

＜コンパイラの出力オブジェクト：ミディアム・モデルの場合＞

```
        movw    hl , !_i
        call    !!_func
        movw    de , !_p
        movw    ax , [ de ]
        movw    hl , ax
        movw    ax , !_fp
        mov     CS , #00H       ; 0
        call    ax
```

＜コンパイラの出力オブジェクト：コンパクト・モデルの場合＞

```
        mov     ES , #highw ( _i )
        movw    hl , ES:!_i
        call    !_func
        mov     ES , #highw ( _p )
        mov     a , ES:!_p + 2
        mov     _@SEGAX , a
        movw    ax , ES:!_p
        movw    de , ax
        mov     a , _@SEGAX
        mov     _@SEGDE , a
        mov     ES , a
        movw    ax , ES:[ de ]
        movw    hl , ax
        mov     ES , #highw ( _fp )
        mov     a , ES:!_fp + 2
        mov     _@SEGAX , a
        movw    ax , ES:!_fp
        push    ax
        mov     a , _@SEGAX
        mov     CS , a
        pop     ax
        call    ax
```

＜コンパイラの出力オブジェクト：ラージ・モデルの場合＞

```
        mov     ES , #highw ( _i )
        movw    hl , ES:!_i
        call    !!_func
        mov     ES , #highw ( _p )
        mov     a , ES:!_p + 2
        mov     _p@SEGAX , a
        movw    ax , ES:!_p
        movw    de , ax
        mov     a , _@SEGAX
        mov     _@SEGDE , a
        mov     ES , a
        movw    ax , ES:[ de ]
        movw    hl , ax
        mov     ES , #highw ( _fp )
        mov     a , ES:!_fp + 2
        mov     _@SEGAX , a
        movw    ax , ES:!_fp
        push    ax
        mov     a , _@SEGAX
        mov     CS , a
        pop     ax
        call    ax
```

【制限】

- コンパクト・モデル，ラージ・モデルを指定した場合でも，データを 64K バイトをまたがって配置することはできません。
- 異なるメモリ・モデルを指定したモジュール同士は，リンクすることはできません。
- far 領域に配置された初期値あり変数，初期値なし変数の 1 ロード・モジュール・ファイルあたりのサイズは，各々（64K - 1）バイト（注：-za 指定時は 64K バイト）です。

  この制限を回避するためには，「コンパイラ出力セクション名の変更（#pragma section …）」の機能を利用して，特定のファイル中の初期値あり変数や初期値なし変数のセクション名を，別の出力セクション名に変更することで，サイズを増やすことが可能です。

  ただし，スタートアップ・ルーチン，終端ルーチンの変更を行う必要があります（「コンパイラ出力セクション名の変更（#pragma section …）」の**【ROM 化関連セクション名変更に伴うスタートアップ・ルーチンなどの変更例】**を参照してください）。

  なお，1 つの出力セクション名あたりの最大サイズは変わりません。
- -za オプションを指定しない場合は，データを 64K バイト境界の領域の最後の 1 バイトに配置することはできません（「near/far 領域指定」の**【注意】**を参照してください）。

11.6 C ソースの修正

拡張機能を使用することにより，効率の良いオブジェクトを生成することができます。しかし，拡張機能は 78K0R シリーズに即したもので，他に利用するためには修正が必要になる場合があります。

ここでは，他の C コンパイラから CC78K0R への移植と，CC78K0R から他の C コンパイラへの移植の 2 つの場合について，その方法を説明します。

＜他の C コンパイラから CC78K0R ＞

- #pragma [注]

  他の C コンパイラが #pragma をサポートしている場合は，C ソースを修正する必要があります。修正方法は，その C コンパイラの仕様によって検討します。

- 拡張仕様

  他の C コンパイラがキーワードを追加するなどの仕様の拡張を行っている場合は，修正する必要があります。修正方法はその C コンパイラの仕様によって検討します。

注 ANSI でサポートされている前処理指令の 1 つで，#pragma に続く文字列をコンパイラへの指令として認識させるものです。その指令がコンパイラによってサポートされていなければ，#pragma 指令は無視され，コンパイルが続けられて正常に終了します。

＜ CC78K0R から他の C コンパイラ＞

- CC78K0R は，拡張機能としてキーワードの追加を行っているため，他の C コンパイラへ移植するためには，キーワードを削除するか，#ifdef で切り分けなければなりません。

【例】

(1) キーワードを無効にする（callf，sreg，norec なども同様）

```
#ifndef __K0R__
#define callt            /* callt 関数を通常の関数にします。*/
#endif
```

(2) 他の型に変更する

```
#ifndef __K0R__
#define bit     char     /* bit 型変数を char 型変数にします。*/
#endif
```

11.7 関数呼び出しインタフェース

関数呼び出し時の関数間インタフェースについて次の内容を説明します。

- 返り値（すべての関数で共通）
- 通常関数呼び出しインタフェース
  (1) 引数の渡し方
  (2) 引数・自動変数の格納場所
- norec 関数呼び出しインタフェース
  (1) 引数の渡し方
  (2) 引数の格納場所
  (3) 自動変数の格納場所

11.7.1 返り値

関数の返り値は，レジスタ，またはキャリー・フラグに格納します。
返り値の格納場所を，次に示します。

表 11-3 返り値の格納場所

型	格納場所
1 バイト整数	BC
2 バイト整数	
4 バイト整数	BC（下位），DE（上位）
ポインタ（__near 属性）	BC
ポインタ（__far 属性）	BC（下位），DE（上位）
構造体，共用体 （スモール・モデル，ミディアム・モデル）	BC
構造体，共用体 （コンパクト・モデル，ラージ・モデル）	BC（下位），DE（上位）
1 ビット	CY（キャリー・フラグ）
浮動小数点数（float 型）	BC（下位），DE（上位）
浮動小数点数（double 型）	BC（下位），DE（上位）

11.7.2　通常関数呼び出しインタフェース

(1) 引数の渡し方

- 関数呼び出し時，第 2 引数以降はスタックで関数定義側に渡します。
- 第 1 引数はレジスタ，またはスタックで関数定義側に渡します。
  第 1 引数の受け渡し場所を，次に示します。

表 11-4　第 1 引数の渡し場所（関数呼び出し側）

型	格納場所
1 バイト・データ[注] 2 バイト・データ[注]	AX
3 バイト・データ[注] 4 バイト・データ[注]	AX，BC
浮動小数点数	AX，BC
その他	スタック渡し

注　1-4 バイト・データには，構造体，共用体，ポインタを含みます。

(2) 引数・自動変数の格納場所

- レジスタ宣言，または -qv 指定により，関数の先頭で引数・自動変数をレジスタに割り当てます。それ以外の引数・自動変数は，スタックに積みます。
- 関数呼び出し側からの受け渡しがスタックである引数を，レジスタに割り当てない場合は，受け渡し場所がそのまま割り当て場所になります。
- 引数・自動変数を割り当てるレジスタは HL です。ただし，スタック・フレームがある場合は，HL には割り当たりません。
  -qr 指定時は，さらに _@KREGxx にも割り当て可能となります。_@KREGxx については，「付録 A saddr 領域のラベル一覧」を参照してください。
- 参照回数の多いものから順に，レジスタに割り当てます。
  レジスタ宣言や，-qv を指定しても，参照回数が少ない引数・自動変数は，レジスタに割り当てない場合があります。
- 引数・自動変数を割り当てるレジスタは，関数定義側で退避・復帰します。

【例】

＜ C ソース 1 ＞

```
void    func0 ( register int , int ) ;
void    main ( )
{
        func0 ( 0x1234 , 0x5678 ) ;
}
void    func0 ( register int p1 , int p2 )
{
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

[-qr 指定の場合]

＜出力コード＞

```
_main :
; line  4 :     func0 ( 0x1234 , 0x5678 ) ;
        movw    ax ,  #05678H   ; 22136
        push    ax                      ; 第 2 引数以降はスタック渡し
        movw    ax , #01234H    ; 4660  ; 第 1 引数はレジスタ渡し
        call    !!_func0                ; 関数呼び出し
        pop     ax                      ; 関数呼び出し時に積んだスタックを解放
; line  5 : }
        ret
; line  6 : void func0 ( register int p1 , int p2 )
; line  7 : {
_func0 :
        push    hl
        movw    de , _@KREG14
        push    de                      ; レジスタ引数用 saddr 領域退避
        movw    de , _@KREG12
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    _@KREG14 , ax           ; 第 1 引数 p1 を saddr に割り当てる
        push    ax                      ; 自動変数 a の領域確保
        movw    hl , sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax , [ hl+12 ]  ; p2    ; 引数 p2
        movw    _@KREG12 , ax   ; r     ; 自動変数 r
; line 11 :     a = p1 ;
        movw    ax , _@KREG14   ; p1    ; 引数 p1
        movw    [ hl ] , ax     ; a     ; 自動変数 a
; line 12 : }
        pop     ax                      ; 自動変数 a の領域解放
        pop     ax
        movw    _@KREG12 , ax           ; レジスタ変数用 saddr 領域復帰
        pop     ax
        movw    _@KREG14 , ax           ; レジスタ引数用 saddr 領域復帰
        pop     hl
        ret
```

＜ C ソース 2 ＞

```
void    func1 ( int , register int ) ;
void    main ( )
{
        func1 ( 0x1234 , 0x5678 ) ;
}
void    func1 ( int p1 , register int p2 )
{
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

[-qr 指定の場合]

＜出力コード＞

```
_main :
; line  4 :     func0 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push    ax                      ; 第 2 引数以降はスタック渡し
        movw    ax , #01234H    ; 4660  ; 第 1 引数はレジスタ渡し
        call    !!_func1                ; 関数呼び出し
        pop     ax                      ; 関数呼び出し時に積んだスタックを解放
; line  5 : }
        ret
; line  6 : void func0 ( int p1 , register int p2 )
; line  7 : {
_func0 :
        push    hl
        push    ax                      ; 第 1 引数 p1 をスタックに積む
        movw    de , _@KREG14
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    de , _@KREG12
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    ax , [ sp+12 ]
        movw    _@KREG12 , ax           ; 引数 p2 を saddr に割り当てる
        push    ax                      ; 自動変数 a の領域確保
        movw    hl , sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax , _@KREG12   ; p2    ; 引数 p2
        movw    _@KREG14 , ax   ; r     ; 自動変数 r
; line 11 :     a = p1 ;
        movw    ax , [ hl+6 ]   ; p1    ; 引数 p1
        movw    [ hl ] , ax     ; a     ; 自動変数 a
; line 12 : }
        pop     ax                      ; 自動変数 a の領域解放
        pop     ax
        movw    _@KREG12 , ax           ; レジスタ引数用 saddr 領域復帰
        pop     ax
        movw    _@KREG14 , ax           ; レジスタ変数用 saddr 領域復帰
        pop     ax                      ; 第 1 引数 p1 の領域解放
        pop     hl
        ret
```

11.7.3　norec 関数呼び出しインタフェース

(1) 引数の渡し方

- 関数呼び出し時，レジスタ，または _@NRARGx で関数定義側に渡します。
- _@NRARGx での引数の受け渡しは，-qr 指定時のみ行われます。
  _@NRARGx については，「付録 A saddr 領域のラベル一覧」を参照してください。
- 引数の受け渡し場所を，次に示します。

型	格納場所
1 バイト・データ注， 2 バイト・データ注	第 1 引数から順に AX，DE，_@NRARGx
3 バイト・データ注， 4 バイト・データ注	_@NRARGx
浮動小数点数	_@NRARGx

注　1 - 4 バイト・データには，ポインタを含みます。
1 バイト・データの引数が複数ある場合も，1 つのレジスタ・ペアには 1 つの引数しか割り当てません。
第 1 引数は，1 バイト・データの場合，A レジスタに割り当てます。

(2) 引数の格納場所

- 関数の先頭で DE，_@RTARG6, 7，_@NRARGx に割り当てます。
- _@NRARGx への割り当ては，-qr 指定時のみ行われます。
  _@RTARGx，および _@NRARGx については，「付録 A saddr 領域のラベル一覧」を参照してください。
- 割り当てられない場合は，エラーとなります。
- AX，または A で渡ってきた引数は，DE を引数の受け渡しで使用していない場合は DE，または E に，DE を引数の受け渡しで使用している場合は _@RTARG6, 7 に割り当たります。
  それ以外の引数は，受け渡し場所がそのまま割り当て場所となります。

(3) 自動変数の格納場所

- 自動変数は，引数を割り当てていない DE，_@RTARG6, 7，_@NRARGx に割り当て，さらに，_@NRATxx に割り当てます。
- _@NRARGx，および _@NRATxx への割り当ては，-qr 指定時のみ行われます。
  _@NRARGx，および _@NRARGx については，「付録 A saddr 領域のラベル一覧」を参照してください。
- 割り当てられない場合は，エラーとなります。

第 12 章　アセンブラとの相互参照

この章では，アセンブリ言語で作成したプログラムとのリンク方法について説明します。

C ソース・プログラムから呼び出す関数が他言語で記述されている場合，双方のオブジェクト・モジュールをリンカで結合します。この章では，C 言語で記述されたプログラムが他言語で記述されたプログラムを呼び出す手順，および他言語で記述されたプログラムから C 言語で記述されたプログラムを呼び出す手順を説明します。

他言語とのインタフェースの方法について，CC78K0R と RA78K0R　アセンブラ・パッケージを使用し，次の順序で説明します。

- C 言語からアセンブリ言語ルーチンの呼び出し
- アセンブリ言語から C 言語ルーチンの呼び出し
- C 言語で定義した変数を参照する方法
- アセンブリ言語で定義した変数を C 言語側で参照する方法
- その他注意事項

12.1　引数・自動変数のアクセス方法

引数，および自動変数の割り当てについては，「11.7.2 通常関数呼び出しインタフェース」，および「11.7.3 norec 関数呼び出しインタフェース」を参照してください。

スタックに積まれた引数・自動変数をアクセスする際のベース・ポインタは，HL レジスタを使用します。

12.2 返り値の格納方法

「11.7.1 返り値」を参照してください。

12.3 C 言語からアセンブリ言語ルーチンの呼び出し

ここでは，デフォルトの例を示します。

C 言語からアセンブリ言語ルーチンの呼び出しを，次の順序で説明します。

- C 言語の関数呼び出し手順
- アセンブリ言語ルーチンの情報退避とリターン

12.3.1　C 言語の関数呼び出し手順

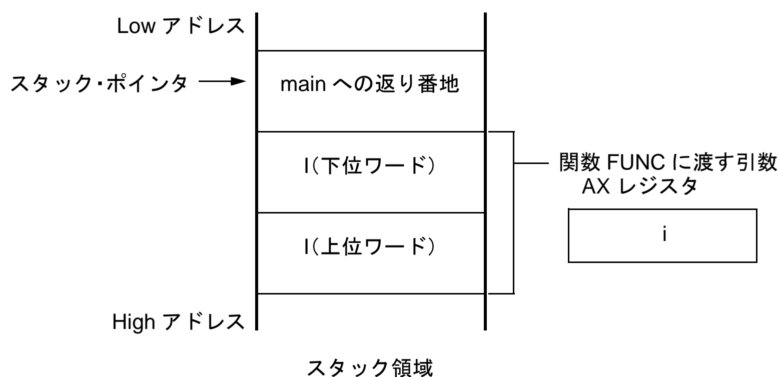
アセンブリ言語ルーチンを呼び出す C 言語のプログラム例を，次に示します。

```
extern  int     FUNC ( int , long ) ;    /* 関数プロトタイプ */

void    main ( ) {
        int     i , j ;
        long    l ;

        i = 1 ;
        l = 0x54321 ;
        j = FUNC ( i , l ) ;             /* 関数コール */
}
```

このプログラム例で，実行時に行われるプログラム間のインタフェースと制御の流れを次に示します。

1. 関数 main から関数 FUNC へ渡す第 1 引数をレジスタに入れ，第 2 引数以降をスタックに積む
2. CALL 命令により関数 FUNC に制御を渡す

上記のプログラム例により，関数 FUNC に制御を移した直後のスタックは，次のようになります。



12.3.2 アセンブリ言語ルーチンの情報退避とリターン

main 関数から呼び出される関数 FUNC では，次の手順で処理を行います。

1. ベース・ポインタ，ワーク・レジスタを退避する
2. スタック・ポインタ（SP）をベース・ポインタ（HL）へコピーする
3. 関数 FUNC 本来の処理を行う
4. 返り値をセットする
5. 退避したレジスタを復帰する
6. 関数 main へリターンする

アセンブリ言語のプログラム例を，次に示します。

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2

@@DATA  DSEG    BASEP
_DT1 :  DS      ( 2 )
_DT2 :  DS      ( 4 )

@@CODE  CSEG
_FUNC :
        PUSH    HL                      ; save base pointer     (1)
        PUSH    AX
        MOVW    HL , SP                 ; copy stack pointer    (2)
        MOVW    AX , [ HL ]             ; arg1
        MOVW    !_DT1 , AX              ; move 1st argument ( i )
        MOVW    AX , [ HL + 10 ]        ; arg2
        MOVW    !_DT2 + 2 , AX
        MOVW    AX , [ HL + 8 ]         ; arg2
        MOVW    !_DT2 , AX              ; move 2nd argument ( l )
        MOVW    BC , #0AH               ; set return value      (4)
        POP     AX
        POP     HL                      ; restore base pointer  (5)
        RET                             ; (6)
        END
```

(1) ベース・ポインタ，ワーク・レジスタの退避

C ソースで記述した関数名の先頭に，“_”を付加したラベルを記述します。C ソース中で記述した関数名と同じ名前になります。

ラベルを記述したあと，HL レジスタ（ベース・ポインタ）を退避します。

C コンパイラが生成するプログラムでは，レジスタ変数用レジスタを退避せずに他の関数を呼び出します。このため，呼ばれる関数でこれらのレジスタの値を変更する場合は，事前に値の退避を行わなければなりません。ただし，呼び出し側でレジスタ変数を使っていない場合，ワーク・レジスタを退避する必要はありません。

(2) スタック・ポインタ（SP）のベース・ポインタ（HL）へのコピー

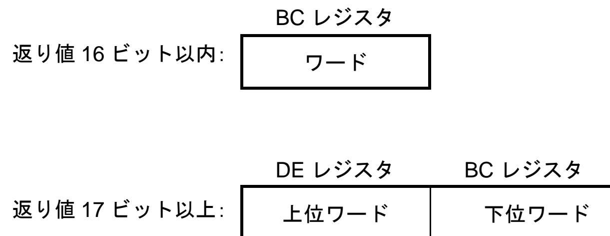
関数内の“PUSH，POP”によりスタック・ポインタ（SP）は変わります。このため，スタック・ポインタを“HL”レジスタにコピーして，引数のベース・ポインタとして使用します。

(3) 関数 FUNC 本来の処理を行う

(1)，(2) の処理を行ったあと，呼び出される関数の本来の処理を行います。

(4) 返り値のセット

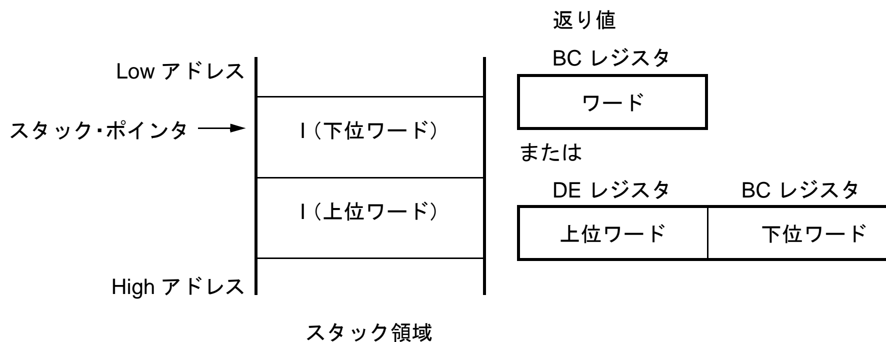
返り値がある場合，返り値を“BC”，“DE”レジスタへセットします。返り値がない場合，セットする必要はありません。



(5) レジスタの復帰

退避したベース・ポインタとワーク・レジスタを復帰します。

(6) 関数 main へのリターン



12.4 アセンブリ言語から C 言語ルーチンの呼び出し

12.4.1　アセンブリ言語の関数呼び出し

C 言語により記述された関数をアセンブリ言語ルーチンから呼び出す手順は，次のようになります。

1. C のワーク・レジスタ（AX，BC，DE）を退避する
2. 引数をスタックに積む
3. C 言語関数をコールする
4. 引数のバイト数分スタック・ポインタ（SP）の値を修正する
5. C 言語関数の返り値（BC，または DE，BC）を参照する

アセンブリ言語のプログラム例を，次に示します。

```
$PROCESSOR ( F1166A0 )

        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2

@@CODE  CSEG
_FUNC2 :
        movw    ax , #20H       ; set 2nd argument ( j )
        push    ax              ;
        movw    ax , #21H       ; set 1st argument ( i )
        call    !_CSUB          ; call "CSUB ( i ,  j )"
        pop     ax              ;
        ret
        END
```

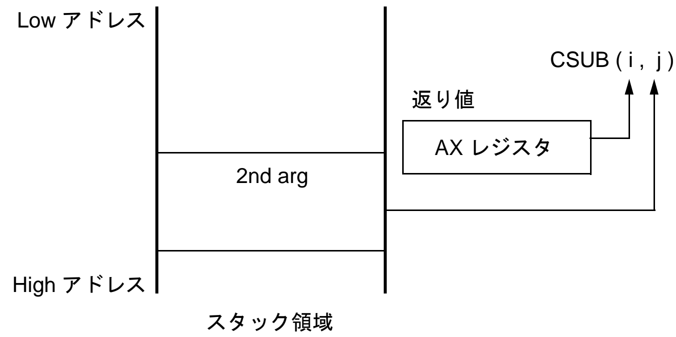
(1) ワーク・レジスタ（AX，BC，DE）の退避

C 言語では，AX，BC，DE の 3 つのレジスタ・ペアを作業用として使用し，戻り時に値の復帰を行いません。このため，レジスタ内の値が必要な場合は，呼び出し側で退避します。
レジスタの退避／復帰は，引数受け渡しコードの前後で行ってください。
なお，HL レジスタについては，C 言語側で使用している場合，常に C 言語側で退避されます。

(2) 引数の積み込み

引数があれば引数をスタックに積みます。

引数の受け渡しは，次のようになります。



(3) C 言語関数のコール

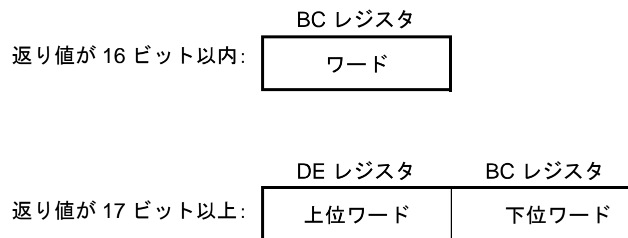
C 言語関数の呼び出しは，CALL 命令で行います。C 言語関数が callt 関数の場合，“callt”命令，callf 関数の場合，“callf”命令で呼び出します。

(4) スタック・ポインタ（SP）の復帰

引数を積んだバイト数分，スタックを復帰します。

(5) 返り値（BC，DE）の参照

C 言語からの返り値は，次のように返されます。



12.5　他言語で定義された変数の参照

12.5.1　C 言語で定義した変数を参照する方法

C 言語プログラム中で定義した外部変数をアセンブリ言語ルーチン中で参照する場合，extrn 宣言します。

アセンブリ言語ルーチン中では，定義した変数の先頭に“_”（アンダースコア）を付けます。

＜C 言語のプログラム例＞

```
extern  void    subf ( ) ;

char    c = 0 ;
int     i = 0 ;

void    main ( ) {
        subf ( ) ;
}
```

RA78K0R アセンブラでは，次のように行います。

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf :
        MOV     !_c , #04H
        MOVW    AX , #07H
        MOVW    !_i , AX
        RET
        END
```

12.5.2　アセンブリ言語で定義した変数を C 言語側で参照する方法

アセンブリ言語で定義した変数を C 言語側で参照するには，次のように行います。

＜C 言語のプログラム例＞

```
extern  char    c ;
extern  int     i ;

void    subf ( ) {
        c = ' A ' ;
        i = 4 ;
}
```

RA78K0R アセンブラでは，次のように行います。

```
        NAME    ASMSUB

        PUBLIC  _i
        PUBLIC  _c

ABC     DSEG    BASEP
_i :    DW      0
_c :    DB      0

        END
```

12.6 その他注意事項

(1) “_”（アンダースコア）

CC78K0R は，出力するオブジェクト・モジュールの外部定義，および参照名に“_”（アンダースコア，ASCII コード“5FH”）を付けます。

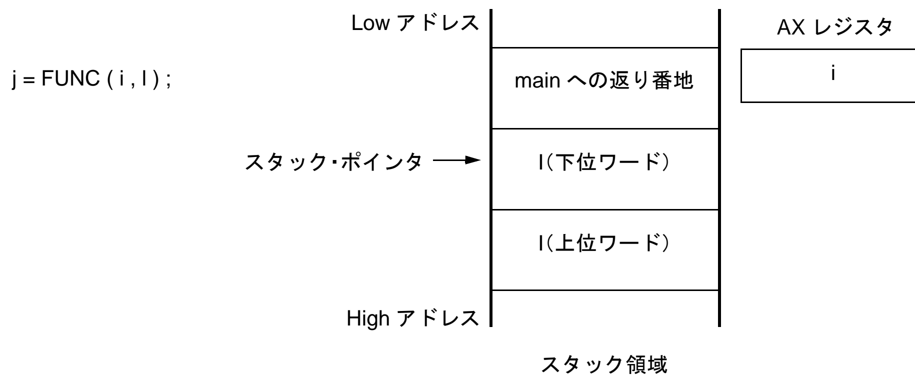
次の C プログラム例で，“j = FUNC (i , l) ;”は，“_FUNC という外部名を参照する”と訳されます。

```
extern  int     FUNC ( int , long ) ;    /* 関数プロトタイプ */

void    main ( ) {
        int     i ,  j ;
        long    l ;

        i = 1 ;
        l = 0x54321;
        j = FUNC ( i , l ) ;             /* 関数コール */
}
```

RA78K0R では，ルーチン名を“_FUNC”と記述します。

(2) スタックに積む引数の配置

スタックに積まれる引数は，後位置引数から前位置引数へと High アドレスから Low アドレス方向に積まれます。



第 13 章 効率の良いコンパイラの活用法

この章では，CC78K0R を有効に利用する方法を紹介します。

13.1 効率の良いコーディング

78K0R シリーズ応用製品の開発を行う場合，CC78K0R ではデバイスの saddr 領域，callt 領域を利用することにより，効率の良いオブジェクトを生成することができます。

```
-  外部変数を使用する
   └── if（saddr 領域が使用可能）
           └── sreg/__sreg 変数を使用する /
               コンパイラ・オプション（-rd）を使用する

-  1 ビットのデータを使用する
   └── if（saddr 領域が使用可能）
           └── bit/boolean/__boolean 型変数を使用する

-  関数の定義
   ├── if（何回も呼ばれる関数）
   │       ├── if（callt 領域が使用可能）
   │       │       └── __callt/callt 関数とする（コード・サイズ削減に有効）
   │       └── if（再帰的に使用しない）
   │               └── __leaf/norec 関数とする
   └── if（オートマティック変数を使用する && saddr 領域が使用可能）
```

(1) 外部変数の使用

外部変数を定義するときに saddr 領域が利用可能であれば，定義する外部変数を sreg/__sreg 変数にします。sreg/__sreg 変数は，メモリに対する命令と比べ命令コードが短く，オブジェクト・コードを縮小することができ，実行速度も向上します（sreg 変数にする代わりに，オプション -rd によっても同様のことを行うことができます）。

```
sreg/__sreg 変数の定義 ： extern sreg int    変数名 ；
                         extern __sreg int 変数名 ；
```

備考　「11.5　saddr 領域利用（sreg/__sreg）」を参照してください。

(2) 1 ビット・データの使用

1 ビットのデータしか使用しないオブジェクトは，bit 型変数 (または boolean/__boolean 型変数) にします。bit/boolean/__boolean 型変数に対する操作には，ビット操作命令が生成されます。また，sreg 変数と同様，saddr 領域を使用しますので，コードを縮小することができ，実行速度も向上します。

```
bit/boolean 型変数の宣言 ： bit               変数名 ；
                           boolean           変数名 ；
                           __boolean         変数名 ；
```

備考　「11.5　bit 型変数，boolean 型変数（bit/boolean/__boolean）」を参照してください。

(3) 関数定義の工夫

何回も呼ばれる関数で，callt 領域を利用できる場合は callt 関数にします。

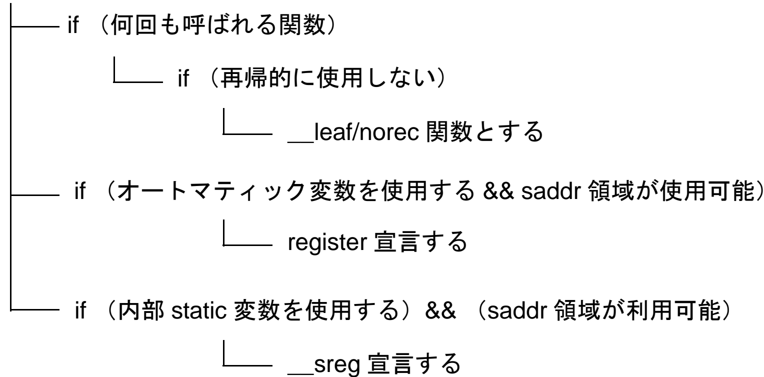
callt 関数は，デバイスの callt 領域を利用して呼び出されるので，通常の呼び出しよりも速く，かつ短いコードで呼び出すことができます。

```
callt 関数の定義 ： callt   int     tsub ( ) {
                      ：
                    }
```

備考　「11.5　callt 関数（callt/__callt）」，「11.5　norec 関数（norec）」を参照してください。

saddr 領域の使用に加え，最適化オプションを使用してコンパイルすることにより，C ソースの修正を行わずに，良いオブジェクトを生成することができます。なお，各 -q サブオプションの効果については，「CC78K0R C コンパイラ 操作編」のユーザーズ・マニュアルを参照してください。

(4) 拡張機能の使用



- 再帰的に使用しない関数

何回も呼ばれる関数の中で再帰的に使用しないものは，__leaf/norec 関数にします。

norec 関数は，関数の前後処理（スタック・フレーム）のない関数になります。このため，通常の関数に比べ，オブジェクト・コードを短縮することができ，実行速度も向上します。

備考　norec 関数の定義（norec int rout ( ) …）については，「11.5　norec 関数（norec）」，「11.7.3 norec 関数呼び出しインタフェース」を参照してください。

- オートマティック変数を使用する関数

オートマティック変数を使用する関数で，saddr 領域が使用可能であれば register 宣言します。register 宣言は，宣言されたオブジェクトをレジスタに割り当てます。

レジスタを用いたプログラムは，メモリを使ったプログラムと比べ高速に動作し，またオブジェクト・コードも短縮されます。

備考　register 変数の定義（register int i ; …）については，「11.5　レジスタ変数（register）」を参照してください。

- 内部 static 変数を使用する関数

内部 static 変数を使用する関数で，saddr 領域が使用可能であれば，__sreg 宣言，または，-rs オプションを指定します。sreg 変数と同様，オブジェクト・コードを縮小することができ，実行速度も向上します。

備考　「11.5　saddr 領域利用（sreg/__sreg）」を参照してください。

その他，次のような方法で，コード効率，または実行スピードを向上することができます。

- SFR 名（または SFR ビット名称）の使用
  #pragma sfr
- 1 ビットのメンバのみからなるビット・フィールドには，__sreg 宣言を使用（メンバには unsigned char 型も使用可能）

```
__sreg  struct  bf {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
} bf_1 ;
```

- 割り込み処理にはレジスタ・バンク切り替えを使用
  #pragma interrupt INTP0 inter RB1
- 乗算，除算組み込み関数の使用

  #pragma mul

  #pragma div
- 高速化したいモジュールのみ，アセンブリ言語で記述

付録 A saddr 領域のラベル一覧

CC78K0R では，次に示すラベル名によって saddr 領域を参照しています。したがって，C ソース・プログラム，またはアセンブラ・ソース・プログラム中で，次のラベルと同じ名前を使用することはできません。

(1) レジスタ変数

ラベル名	アドレス
_@KREG00	0FFEB4H
_@KREG01	0FFEB5H
_@KREG02	0FFEB6H
_@KREG03	0FFEB7H
_@KREG04	0FFEB8H
_@KREG05	0FFEB9H
_@KREG06	0FFEBAH
_@KREG07	0FFEBBH
_@KREG08	0FFEBCH
_@KREG09	0FFEBDH
_@KREG10	0FFEBEH
_@KREG11	0FFEBFH
_@KREG12	0FFEC0H [注]
_@KREG13	0FFEC1H [注]
_@KREG14	0FFEC2H [注]
_@KREG15	0FFEC3H [注]

注　関数の引数が register 宣言，または -qv オプションが指定され，かつ -qr オプションが指定されている場合に，引数を saddr 領域に割り当てます。

(2) norec 関数の引数

ラベル名	アドレス
_@NRARG0	0FFEC4H
_@NRARG1	0FFEC6H
_@NRARG2	0FFEC8H
_@NRARG3	0FFECAH

(3) norec 関数のオートマティック変数

ラベル名	アドレス
_@NRAT00	0FFECCH
_@NRAT01	0FFECDH
_@NRAT02	0FFECEH
_@NRAT03	0FFECFH
_@NRAT04	0FFED0H
_@NRAT05	0FFED1H
_@NRAT06	0FFED2H
_@NRAT07	0FFED3H

(4) セグメント情報格納用

ラベル名	アドレス
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

(5) ランタイム・ライブラリの引数

ラベル名	アドレス
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

付録 B　セグメント名一覧

コンパイラが出力する全セグメントと配置について，説明をします。
なお，表に使用されているオプション，再配置属性は，次の (1)，(2) のとおりです。

(1) CSEG の再配置属性

- CALLT0

  指定セグメントを 80H - BFH 番地で先頭が 2 の倍数になるように配置します。

- AT　絶対式

  指定セグメントを絶対番地に配置します（00000H - FFEFFH 内）。

- UNITP

  指定セグメントを任意の位置へ先頭が 2 の倍数になるように配置します（C0H - EFFFEH 内）。

(2) DSEG の再配置属性

- SADDRP

  指定セグメントを saddr 領域内の FFE20H - FFEFFH に先頭が 2 の倍数になるように配置します。

- UNITP

  指定セグメントを任意の位置へ先頭が 2 の倍数になるように配置します（デフォルトは RAM 領域内）。

B.1　セグメント名一覧

B.1.1　プログラム領域，データ領域

セクション名	セグメント・タイプ	再配置属性	説明
@@CODE	CSEG	BASE	コード部用セグメント（near 領域配置）
@@CODEL	CSEG		コード部用セグメント（far 領域配置）
@@LCODE	CSEG	BASE	ライブラリ・コード用セグメント（near 領域配置）
@@LCODEL	CSEG		ライブラリ・コード用セグメント（far 領域配置）
@@CNST	CSEG	MIRRORP	const 変数用セグメント（near 領域配置）
@@CNSTL	CSEG	PAGE64KP	const 変数用セグメント（far 領域配置）
@@R_INIT	CSEG	UNIT64KP	near 初期化データ用セグメント（初期値あり）
@@RLINIT	CSEG	UNIT64KP	far 初期化データ用セグメント（初期値あり）
@@R_INIS	CSEG	UNIT64KP	初期化データ用セグメント（初期値あり sreg 変数）
@@CALT	CSEG	CALLT0	callt 関数のテーブル用セグメント
@@VECT*nn*	CSEG	AT 00*mm*H	ベクタ・テーブル用セグメント[注]
@@BASE	CSEG	BASE	callt 関数・割り込み関数用セグメント
@@LBASE	CSEG	BASE	ライブラリ・callt 関数用セグメント
@@INIT	DSEG	BASEP	データ領域用セグメント（初期値あり，near 領域配置）
@@INITL	DSEG	UNIT64KP	データ領域用セグメント（初期値あり，far 領域配置）
@@DATA	DSEG	BASEP	データ領域用セグメント（初期値なし，near 領域配置）
@@DATAL	DSEG	UNIT64KP	データ領域用セグメント（初期値なし，far 領域配置）
@@INIS	DSEG	SADDRP	データ領域用セグメント（初期値あり sreg 変数）
@@DATS	DSEG	SADDRP	データ領域用セグメント（初期値なし sreg 変数）
@@BITS	BSEG		boolean 型変数，bit 型変数用セグメント

注　割り込みの種類により，*nn*，*mm* の値が変わります。

B.1.2　フラッシュ・メモリ領域

セクション名	セグメント・タイプ	再配置属性	説明
@ECODE	CSEG	BASE	コード部用セグメント（near 領域配置）
@ECODEL	CSEG		コード部用セグメント（far 領域配置）
@LECODE	CSEG	BASE	ライブラリ・コード用セグメント（near 領域配置）
@LECODEL	CSEG		ライブラリ・コード用セグメント（far 領域配置）
@ECNST	CSEG	MIRRORP	const 変数用セグメント（near 領域配置）
@ECNSTL	CSEG	PAGE64KP	const 変数用セグメント（far 領域配置）
@ER_INIT	CSEG	UNIT64KP	near 初期化データ用セグメント（初期値あり）
@ERLINIT	CSEG	UNIT64KP	far 初期化データ用セグメント（初期値あり）
@ER_INIS	CSEG	UNIT64KP	初期化データ用セグメント（初期値あり sreg 変数）
@EVECT*nn*	CSEG	AT *mmmm*H	ベクタ・テーブル用セグメント注 1
@EXT*xx*	CSEG	AT *yyyy*H	フラッシュ領域分岐テーブル用セグメント注 2
@EINIT	DSEG	BASEP	データ領域用セグメント（初期値あり，near 領域配置）
@EINITL	DSEG	UNIT64KP	データ領域用セグメント（初期値あり，far 領域配置）
@EDATA	DSEG	BASEP	データ領域用セグメント（初期値なし，near 領域配置）
@EDATAL	DSEG	UNIT64KP	データ領域用セグメント（初期値なし，far 領域配置）
@EINIS	DSEG	SADDRP	データ領域用セグメント（初期値あり sreg 変数）
@EDATS	DSEG	SADDRP	データ領域用セグメント（初期値なし sreg 変数）
@EBITS	BSEG		boolean 型変数，bit 型変数用セグメント

注 1　割り込みの種類により，*nn*，*mmmm* の値が変わります。

注 2　フラッシュ領域関数の ID により，*xx*，*yyyy* の値が変わります。

B.2　セグメントの配置

表 B-1　セグメントの配置

セグメント・タイプ	配置先（デフォルト）
CSEG	ROM
BSEG	RAM の saddr 領域
DSEG	RAM

B.3　C ソース例

```
#pragma INTERRUPT       INTP0   inter   rb1     /* 割り込みベクタ */

void    main ( void );                          /* 関数プロトタイプ宣言 */
const   int     i_cnst = 1 ;                    /* const 変数 */
callt   void    f_clt ( void ) ;                /* callt 関数プロトタイプ宣言 */
boolean b_bit ;                                 /* boolean 型変数 */
long    l_init = 2 ;                            /* 初期値あり外部変数 */
int     i_data ;                                /* 初期値なし外部変数 */
__sreg  int     sr_inis = 3 ;                   /* 初期値あり sreg 変数 */
__sreg  int     sr_dats ;                       /* 初期値なし sreg 変数 */

void    main ( )                                /* 関数定義 */
{
        int     i ;
        i = 100 ;
}

void    inter ( )                               /* 割り込み関数定義 */
{
}

callt   void    f_clt ( )                       /* callt 関数定義 */
{
}
```

B.4 出力アセンブラ・モジュール例

アセンブラ・ソース中の疑似命令，命令セットは，各デバイスにより異なります。

詳細については，「RA78K0R アセンブラ・パッケージ」のユーザーズ・マニュアルを参照してください。

[スモール・モデルの場合]

```
; 78K0R Series C Compiler Vx.xx Assembler Source
;                            Date : xx xxx xxxx Time : xx : xx : xx

; Command    : -cxxx sample.c -ms -sa -ng
; In-file    : sample.c
; Asm-file   : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH , 0330H , 00H , 020H , 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@CNST      CSEG  MIRRORP               ; const 変数用セグメント
_i_cnst :   DW    01H               ; 1

@@R_INIT    CSEG  UNIT64KP              ; 初期化データ用セグメント（初期値あり外部変数）
            DW    00002H , 00000H ; 2

@@INIT      DSEG  BASEP                 ; 仮データ用セグメント（初期値あり）
_l_init :   DS    ( 4 )

@@DATA      DSEG  BASEP                 ; 仮データ用セグメント（初期値なし）
_i_data :   DS    ( 2 )

@@R_INIS    CSEG  UNIT64KP              ; 初期化データ用セグメント（初期値あり sreg 変数）
            DW    03H               ; 3

@@INIS      DSEG  SADDRP                ; 仮データ領域用セグメント（初期値あり sreg 変数）
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP                ; 仮データ領域用セグメント（初期値なし sreg 変数）
_sr_dats :  DS  ( 2 )

@@CALT      CSEG  CALLT0                ; callt 関数のテーブル用セグメント
?f_clt :    DW    _f_clt
```

```
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* 割り込み関数 */
; line  2 :
; line  3 : void    main ( void ) ;             /* 関数プロトタイプ宣言 */
; line  4 : const   int     i_cnst = 1 ;        /* const 変数 */
; line  5 : callt   void    f_clt ( void ) ;    /* callt 関数プロトタイプ宣言 */
; line  6 : boolean b_bit ;                     /* boolean 型変数 */
; line  7 : long    l_init = 2 ;                /* 初期値あり外部変数 */
; line  8 : int     i_data ;                    /* 初期値なし外部変数 */
; line  9 : sreg    int     sr_inis = 3 ;       /* 初期値あり sreg 変数 */
; line 10 : sreg    int     sr_dats ;           /* 初期値なし sreg 変数 */
; line 11 :
; line 12 : void    main ( )                    /* 関数定義 */
; line 13 : {

@@CODE   CSEG  BASE                             ; コード部用セグメント
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H        ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )                   /* 割り込み関数定義 */
; line 19 : {

@@BASE   CSEG  BASE                          ; callt 関数・割り込み関数用セグメント
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )              /* callt 関数定義 */
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                    ; ベクタ・テーブル用セグメント
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

[ミディアム・モデルの場合]

```
; 78K0R Series C Compiler Vx.xx Assembler Source
;                            Date : xx xxx xxxx Time : xx : xx : xx

; Command    : -cxxx sample.c -mm -sa -ng
; In-file    : sample.c
; Asm-file   : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH , 0330H , 00H , 020H , 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                      ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@CNST      CSEG  MIRRORP             ; const 変数用セグメント
_i_cnst :   DW    01H             ; 1

@@R_INIT    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり外部変数）
            DW    00002H , 00000H ; 2

@@INIT      DSEG  BASEP               ; 仮データ用セグメント（初期値あり）
_l_init :   DS    ( 4 )

@@DATA      DSEG  BASEP               ; 仮データ用セグメント（初期値なし）
_i_data :   DS    ( 2 )

@@R_INIS    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり sreg 変数）
            DW    03H             ; 3

@@INIS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値あり sreg 変数）
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値なし sreg 変数）
_sr_dats :  DS  ( 2 )

@@CALT      CSEG  CALLT0              ; callt 関数のテーブル用セグメント
?f_clt :    DW    _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* 割り込み関数 */
; line  2 :
; line  3 : void    main ( void ) ;             /* 関数プロトタイプ宣言 */
; line  4 : const   int     i_cnst = 1 ;        /* const 変数 */
; line  5 : callt   void    f_clt ( void ) ;    /* callt 関数プロトタイプ宣言 */
; line  6 : boolean b_bit ;                     /* boolean 型変数 */
```

```
; line  7 : long    l_init = 2 ;                /* 初期値あり外部変数 */
; line  8 : int     i_data ;                    /* 初期値なし外部変数 */
; line  9 : __sreg  int     sr_inis = 3 ;       /* 初期値ありsreg変数 */
; line 10 : __sreg  int     sr_dats ;           /* 初期値なしsreg変数 */
; line 11 :
; line 12 : void    main ( )                    /* 関数定義 */
; line 13 : {

@@CODEL  CSEG                                   ; コード部用セグメント
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H         ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )                   /* 割り込み関数定義 */
; line 19 : {

@@BASE   CSEG  BASE                           ; callt関数・割り込み関数用セグメント
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )              /* callt関数定義*/
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                    ; ベクタ・テーブル用セグメント
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

[コンパクト・モデルの場合]

```
; 78K0R Series C Compiler Vx.xx Assembler Source
;                              Date : xx xxx xxxx Time : xx : xx : xx

; Command    : -cxxx sample.c -mc -sa -ng
; In-file    : sample.c
; Asm-file   : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH , 0330H , 00H , 020H , 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                      ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@R_INIS    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり sreg 変数）
            DW    03H             ; 3

@@INIS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値あり sreg 変数）
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値なし sreg 変数）
_sr_dats :  DS  ( 2 )

@@CNSTL     CSEG  PAGE64KP            ; const 変数用セグメント
_i_cnst :   DW    01H             ; 1

@@RLINIT    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり）
            DW    00002H , 00000H ; 2

@@INITL     DSEG  UNIT64KP            ; 仮データ用セグメント（初期値あり）
_l_init :   DS    ( 4 )

@@DATAL     DSEG  UNIT64KP            ; 仮データ用セグメント（初期値なし）
_i_data :   DS    ( 2 )

@@CALT      CSEG  CALLT0              ; callt 関数のテーブル用セグメント
?f_clt :    DW    _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* 割り込み関数 */
; line  2 :
; line  3 : void    main ( void ) ;             /* 関数プロトタイプ宣言 */
; line  4 : const   int     i_cnst = 1 ;        /* const 変数 */
; line  5 : callt   void    f_clt ( void ) ;    /* callt 関数プロトタイプ宣言 */
; line  6 : boolean b_bit ;                     /* boolean 型変数 */
```

```
; line  7 : long    l_init = 2 ;                /* 初期値あり外部変数 */
; line  8 : int     i_data ;                    /* 初期値なし外部変数 */
; line  9 : __sreg  int     sr_inis = 3 ;       /* 初期値あり sreg 変数 */
; line 10 : __sreg  int     sr_dats ;           /* 初期値なし sreg 変数 */
; line 11 :
; line 12 : void    main ( )                    /* 関数定義 */
; line 13 : {

@@CODE   CSEG  BASE                             ; コード部用セグメント
_main :
         push  hl
; line 14 :        int     i ;
; line 15 :        i = 100 ;
         movw  hl , #064H          ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )                   /* 割り込み関数定義 */
; line 19 : {

@@BASE   CSEG  BASE                          ; callt 関数・割り込み関数用セグメント
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )              /* callt 関数定義 */
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                    ; ベクタ・テーブル用セグメント
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

[ラージ・モデルの場合]

```
; 78K0R Series C Compiler Vx.xx Assembler Source
;                             Date : xx xxx xxxx Time : xx : xx : xx

; Command   : -cxxx sample.c -ml -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH , 0330H , 00H , 020H , 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                      ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@R_INIS    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり sreg 変数）
            DW    03H             ; 3

@@INIS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値あり sreg 変数）
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP              ; 仮データ領域用セグメント（初期値なし sreg 変数）
_sr_dats :  DS  ( 2 )

@@CNSTL     CSEG  PAGE64KP            ; const 変数用セグメント
_i_cnst :   DW    01H             ; 1

@@RLINIT    CSEG  UNIT64KP            ; 初期化データ用セグメント（初期値あり）
            DW    00002H , 00000H ; 2

@@INITL     DSEG  UNIT64KP            ; 仮データ用セグメント（初期値あり）
_l_init :   DS    ( 4 )

@@DATAL     DSEG  UNIT64KP            ; 仮データ用セグメント（初期値なし）
_i_data :   DS    ( 2 )

@@CALT      CSEG  CALLT0              ; callt 関数のテーブル用セグメント
?f_clt :    DW    _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* 割り込み関数 */
; line  2 :
; line  3 : void    main ( void ) ;              /* 関数プロトタイプ宣言 */
; line  4 : const   int     i_cnst = 1 ;         /* const 変数 */
; line  5 : callt   void    f_clt ( void ) ;     /* callt 関数プロトタイプ宣言 */
; line  6 : __boolean       b_bit ;              /* boolean 型変数 */
```

```
; line  7 : long    l_init = 2 ;                    /* 初期値あり外部変数 */
; line  8 : int     i_data ;                        /* 初期値なし外部変数 */
; line  9 : __sreg  int     sr_inis = 3 ;           /* 初期値ありsreg変数 */
; line 10 : __sreg  int     sr_dats ;               /* 初期値なしsreg変数 */
; line 11 :
; line 12 : void    main ( )                        /* 関数定義 */
; line 13 : {

@@CODEL  CSEG                                       ; コード部用セグメント
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H         ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )                       /*割り込み関数定義 */
; line 19 : {

@@BASE   CSEG  BASE                               ; callt関数・割り込み関数用セグメント
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )                  /* callt関数定義 */
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                        ; ベクタ・テーブル用セグメント
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

付録 C ランタイム・ライブラリ一覧

ランタイム・ライブラリの一覧を，次に示します。

これらの演算の命令は，@@ などを関数名の頭につけた形式で呼び出されます。ただし，cstart，cstarte，cprep，cdisp は，先頭に _@ を付加した形式で呼び出されます。

なお，表 C-1 にない演算については，ライブラリのサポートはありません。コンパイラがインライン展開を行います。

long の加減算，and/or/xor，シフトは，インライン展開される場合があります。

表 C-1 ランタイム・ライブラリ

分類	関数名	機能
インクリメント	lsinc	signed long をインクリメントする
	luinc	unsigned long をインクリメントする
	finc	float をインクリメントする
デクリメント	lsdec	signed long をデクリメントする
	ludec	unsigned long をデクリメントする
	fdec	float をデクリメントする
符号反転	lsrev	signed long を符号反転する
	lurev	unsigned long を符号反転する
	frev	float を符号反転する
1 の補数	lscom	signed long の 1 の補数を求める
	lucom	unsigned long の 1 の補数を求める
論理否定	lsnot	signed long の否定を求める
	lunot	unsigned long の否定を求める
	fnot	float の否定を求める
乗算	csmul	signed char 同士の乗算
	cumul	unsigned char 同士の乗算
	ismul	signed int 同士の乗算
	iumul	unsigned int 同士の乗算
	lsmul	signed long 同士の乗算
	lumul	unsigned long 同士の乗算
	fmul	float 同士の乗算

分類	関数名	機能
除算	csdiv	signed char 同士の除算
	cudiv	unsigned char 同士の除算
	isdiv	signed int 同士の除算
	iudiv	unsigned int 同士の除算
	lsdiv	signed long 同士の除算
	ludiv	unsigned long 同士の除算
	fdiv	float 同士の除算
剰余算	csrem	signed char 同士の剰余算
	curem	unsigned char 同士の剰余算
	isrem	signed int 同士の剰余算
	iurem	unsigned int 同士の剰余算
	lsrem	signed long 同士の剰余算
	lurem	unigned long 同士の剰余算
加算	lsadd	signed long 同士の加算
	luadd	unsigned long 同士の加算
	fadd	float 同士の加算
減算	lssub	signed long 同士の減算
	lusub	unsigned long 同士の減算
	fsub	float 同士の減算
左シフト	lslsh	signed long の左シフト
	lulsh	unsigned long の左シフト
右シフト	lsrsh	signed long の右シフト
	lursh	unsigned long の右シフト
比較	cscmp	signed char 同士の比較
	iscmp	signed int 同士の比較
	lscmp	signed long 同士の比較
	lucmp	unsigned long 同士の比較
	fcmp	float 同士の比較
ビット AND	lsband	signed long 同士の AND
	luband	unsigned long 同士の AND
ビット OR	lsbor	signed long 同士の OR
	lubor	unsigned long 同士の OR

<table>
<tr><th>分類</th><th>関数名</th><th>機能</th></tr>
<tr><td rowspan="2">ビット XOR</td><td>lsbxor</td><td>signed long 同士の XOR</td></tr>
<tr><td>lubxor</td><td>unsigned long 同士の XOR</td></tr>
<tr><td rowspan="2">浮動小数点数からの変換</td><td>ftols</td><td>float から signed long に変換する</td></tr>
<tr><td>ftolu</td><td>float から unsigned long に変換する</td></tr>
<tr><td rowspan="2">浮動小数点への変換</td><td>lstof</td><td>signed long から float に変換する</td></tr>
<tr><td>lutof</td><td>unsigned long から float に変換する</td></tr>
<tr><td>bit からの変換</td><td>btol</td><td>bit を long に変換する</td></tr>
<tr><td>スタートアップ・ルーチン</td><td>cstart</td><td>スタートアップ・モジュール
- atexit 関数で関数を登録する領域（4 * 32 バイト）を確保し，先頭のラベル名を _@FNCTBL とする。
- ブレーク領域（32 バイト）を確保し，先頭のラベル名を _@MEMTOP とし，領域の次のアドレスのラベル名を _@MEMBTM とする。
- リセット・ベクタ・テーブルのセグメントを次のように定義し，スタートアップ・モジュールの先頭アドレスを指定する。
@@VECT00 CSEG AT 0000H
DW _@cstart
- ミラー領域を設定する。
- レジスタ・バンクを RB0 に設定する。
- エラー番号を入れる変数 _errno に 0 を設定する。
- atexit 関数で登録した関数の数を入れる変数 _@FNCENT に 0 を設定する。
- ブレーク値の初期値として，_@MEMTOP のアドレスを変数 _@BRKADR に設定する。
- rand 関数の疑似乱数の発生元となる変数 _@SEED に初期値 1 を設定する。
- 初期化データのコピー処理，および初期値なし外部データの 0 クリアを行う。
- main 関数（ユーザ・プログラム）を呼び出す。
- exit 関数をパラメータ 0 で呼び出す</td></tr>
<tr><td rowspan="7">関数前後処理</td><td>cprep</td><td>関数の前処理</td></tr>
<tr><td>cdisp</td><td>関数の後処理</td></tr>
<tr><td>cprep2</td><td>関数の前処理（レジスタ変数用 saddr 領域を含む）</td></tr>
<tr><td>cdisp2</td><td>関数の後処理（レジスタ変数用 saddr 領域を含む）</td></tr>
<tr><td>cprep3</td><td>関数の前処理（レジスタ変数用 saddr 領域を含む）</td></tr>
<tr><td>cdisp3</td><td>関数の後処理（レジスタ変数用 saddr 領域を含む）</td></tr>
<tr><td>hdwinit</td><td>CPU リセット直後に周辺装置（sfr）の初期化処理を行う</td></tr>
</table>

分類	関数名	機能
補助	mulu	mulu 命令互換
	mulue	mulu 命令互換
	divuw	divuw 命令互換
	divuwe	divuw 命令互換
	cmpa1	定型命令パターン置換用
	cmpax1	
	ctoi	
	incde	
	decde	
	inchl	
	dechl	
	dellab	
	dell03	
	della4	
	delsab	
	dels03	
	hlllab	
	hlll03	
	hllla4	
	hllsab	
	hlls03	
	apinch	
	apdech	
	incwhl	
	decwhl	
	swap4	
	tableh	
	uctoi	

付録 D ライブラリ消費スタック一覧

標準ライブラリのスタック消費量一覧を，次に示します。

表 D-1 標準ライブラリのスタック消費量一覧

分類	関数名	スモール・モデル， ミディアムモデル兼用	コンパクト・モデル， ラージ・モデル兼用
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longjmp	2	2
stdarg.h	va_arg	0	0
	va_start	0	0
	va_starttop	0	0
	va_end	0	0

分類	関数名	スモール・モデル， ミディアムモデル兼用	コンパクト・モデル， ラージ・モデル兼用
stdio.h	sprintf	52（102）注 1	52（112）注 1
	sscanf	290（330）注 1	290（338）注 1
	printf	60（100）注 1	64（104）注 1
	scanf	302（328）注 1	306（336）注 1
	vprintf	60（100）注 1	66（110）注 1
	vsprintf	52（100）注 1	52（110）注 1
	getchar	0	0
	gets	8	14
	putchar	0	0
	puts	6	10
	__putc	4	4

分類	関数名	スモール・モデル， ミディアムモデル兼用	コンパクト・モデル， ラージ・モデル兼用
stdlib.h	atoi	4	4
	atol	10	10
	strtol	20	20
	strtoul	20	20
	calloc	12	12
	free	8	8
	malloc	6	6
	realloc	12	12
	abort	0	0
	atexit	0	0
	exit	6 + n 注2	6 + n 注2
	abs	0	0
	div	6	6
	labs	0	0
	ldiv	16	16
	brk	0	0
	sbrk	2	2
	atof	18	18
	strtod	18（30）注6	20（34）注6
	itoa	10	10
	ltoa	16	16
	ultoa	16	16
	rand	18（14）注3	18（14）注3
	srand	0	0
	bsearch	40 + n 注4	44 + n 注4
	qsort	16 + n 注5	18 + n 注5
	strbrk	0	0
	strsbrk	2	2
	stritoa	10	10
	strltoa	16	16
	strultoa	16	16

分類	関数名	スモール・モデル, ミディアムモデル兼用	コンパクト・モデル, ラージ・モデル兼用
string.h	memcpy	4	8
	memmove	4	6
	strcpy	2	6
	strncpy	4	10
	strcat	2	6
	strncat	4	8
	memcmp	2	4
	strcmp	2	2
	strncmp	2	2
	memchr	2	4
	strchr	4	2
	strcspn	4	4
	strpbrk	4	6
	strrchr	4	6
	strspn	4	6
	strstr	4	8
	strtok	4	4
	memset	4	6
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	6

分類	関数名	スモール・モデル， ミディアムモデル兼用	コンパクト・モデル， ラージ・モデル兼用
math.h	acos	34	34
	asin	34	34
	atan	34	34
	atan2	38	38
	cos	32	32
	sin	32	32
	tan	38	38
	cosh	38	38
	sinh	38	38
	tanh	44	44
	exp	34	34
	frexp	8（20）注 6	12（24）注 6
	ldexp	6（20）注 6	6（22）注 6
	log	34	34
	log10	34	34
	modf	8（20）注 6	12（24）注 6
	pow	38	38
	sqrt	26	26
	ceil	6（18）注 6	6（20）注 6
	fabs	4	4
	floor	6（18）注 6	6（20）注 6
	fmod	10（22）注 6	10（24）注 6
	matherr	0	0
	acosf	34	34
	asinf	34	34
	atanf	34	34
	atan2f	38	38
	cosf	32	32
	sinf	32	32
	tanf	38	38
	coshf	38	38
	sinhf	38	38

分類	関数名	スモール・モデル，ミディアムモデル兼用	コンパクト・モデル，ラージ・モデル兼用
math.h	tanhf	44	44
	expf	34	34
	frexpf	8（20）注 6	12（24）注 6
	ldexpf	6（20）注 6	6（22）注 6
	logf	34	34
	log10f	34	34
	modff	8（20）注 6	12（24）注 6
	powf	38	38
	sqrtf	26	26
	ceilf	6（18）注 6	6（20）注 6
	fabsf	4	4
	floorf	6（18）注 6	6（20）注 6
	fmodf	10（22）注 6	10（24）注 6
assert.h	__assertfail	72（112）注 7	82（122）注 7

注 1　（ ）内は浮動小数点対応版使用時の値

注 2　n は atexit 関数で登録された外部関数中の最大スタック消費量

注 3　（ ）内は乗算器を使用した場合

注 4　n は bsearch から呼び出される外部関数のスタック消費量

注 5　n は（X + qsort から呼び出される外部関数のスタック消費量）×（1 + 再帰呼び出しの発生回数）

スモール・モデル，ミディアム・モデル兼用ライブラリ使用時　：X = 38

コンパクト・モデル，ラージ・モデル兼用ライブラリ使用時　：X = 40

注 6　（ ）内は演算例外発生時

注 7　（ ）内は浮動小数点数対応版 printf 使用時

ランタイム・ライブラリのスタック消費量一覧を，次に示します。

表 D-2 ランタイム・ライブラリのスタック消費量一覧

分類	関数名	スタック消費量
インクリメント	lsinc	0
	luinc	0
	finc	20
デクリメント	lsdec	0
	ludec	0
	fdec	20
符号反転	lsrev	2
	lurev	2
	frev	4
1 の補数	lscom	0
	lucom	0
論理否定	lsnot	0
	lunot	0
	fnot	4
乗算	csmul	0
	cumul	0
	ismul	4（2）注 2
	iumul	4（2）注 2
	lsmul	8（4）注 2
	lumul	8（4）注 2
	fmul	12 （28，30）注 1
除算	csdiv	8
	cudiv	2
	isdiv	10
	iudiv	4
	lsdiv	12
	ludiv	6
	fdiv	12 （28，30）注 1

分類	関数名	スタック消費量
剰余算	csrem	8
	curem	2
	isrem	12
	iurem	6
	lsrem	12
	lurem	6
加算	lsadd	0
	luadd	0
	fadd	12 （28，30）注 1
減算	lssub	2
	lusub	2
	fsub	12 （28，30）注 1
左シフト	lslsh	4
	lulsh	4
右シフト	lsrsh	4
	lursh	4
比較	cscmp	0
	iscmp	0
	lscmp	2
	lucmp	2
	fcmp	8 （26，28）注 1
ビット AND	lsband	0
	luband	0
ビット OR	lsbor	0
	lubor	0
ビット XOR	lsbxor	0
	lubxor	0
浮動小数点数からの変換	ftols	10
	ftolu	10
浮動小数点数への変換	lstof	10
	lutof	10
bit からの変換	btol	0
スタートアップ・ルーチン	cstart	4

分類	関数名	スタック消費量
関数前後処理	cprep	2 + n 注 3
	cdisp	0
	cprep2	ベース・ポインタ + 第 1 引数 + レジスタ変数 + 自動変数のサイズ
	cdisp2	0
	cprep3	ベース・ポインタ + 第 1 引数 + レジスタ変数 + 自動変数のサイズ
	cdisp3	0
	hdwinit	0

分類	関数名	スタック消費量
補助	mulu	0
	mulue	0
	divuw	6
	divuwe	6
	cmpa1	0
	cmpax1	0
	ctoi	0
	incde	0
	decde	0
	inchl	0
	dechl	0
	dellab	0
	dell03	0
	della4	0
	delsab	0
	dels03	0
	hlllab	0
	hlll03	0
	hllla4	0
	hllsab	0
	hlls03	0
	apinch	0
	apdech	0
	incwhl	0
	decwhl	0
	swap4	0
	tableh	0
	uctoi	0

注1　（ ）内は演算例外発生時（コンパイラ付属の matherr 関数を使用した場合）

（A，B）

A：スモール・モデル，ミディアム・モデル兼用ライブラリ使用時

B：コンパクト・モデル，ラージ・モデル兼用ライブラリ使用時

注2　（ ）内は乗算器を使用した場合

注3　n は確保するオートマティック変数のサイズ

付録 E　ライブラリ最大割り込み禁止時間一覧

乗算器を使用したライブラリの中では，割り込み時に演算内容が途中で壊されないように，割り込み禁止になる時間があります。

乗算器を使用したライブラリの中での，ライブラリの最大割り込み禁止時間一覧を，次に示します。

乗算器を使用しないライブラリでは，割り込み禁止になる区間はありません。

表 E-1　ライブラリの最大割り込み禁止時間（クロック数）

分類	関数名	最大割り込み禁止時間	備考
乗算	@@ismul	12	signed int 同士の乗算
	@@iumul	12	unsigned int 同士の乗算
	@@lsmul	24	signed long 同士の乗算
	@@lumul	24	unsigned long 同士の乗算
stdlib.h	rand	24	@@lumul を使用
	qsort	12	@@iumul を使用

総合索引

Symbols

演算子 … 170
#asm ～ #endasm … 346
#define 指令 … 172
#include … 52, 166
#pragma BRK … 363
#pragma DI … 360
#pragma div … 390
#pragma EI … 360
#pragma ext_func … 405
#pragma ext_table … 402
#pragma HALT … 363
#pragma inline … 412
#pragma interrupt … 352
#pragma mul … 388
#pragma name … 385
#pragma NOP … 363
#pragma opc … 392
#pragma rot … 386
#pragma rtos_interrupt … 394
#pragma rtos_task … 399
#pragma section … 372
#pragma sfr … 337
#pragma STOP … 363
#pragma vect … 352
#pragma 指令 … 324
演算子 … 170
?? … 31

A

abort … 236
abs … 238
acos … 269
acosf … 292
ANSI … 319
asin … 270
asinf … 293
__asm … 346
ASM 文 … 26, 346
assert … 191
__assertfail … 315
atan … 271
atan2 … 272
atan2f … 295
atanf … 294
atexit … 192, 237
atof … 192, 241
atoi … 228
atol … 228
auto … 55

B

bit 型変数 … 26, 342
__boolean … 342
boolean/__boolean 型変数 … 26
boolean 型変数 … 342
break 文 … 146
BRK … 363
brk … 192, 240
bsearch … 245

C

calloc … 232
callt/__callt … 328
callt/__callt 関数 … 26
callt 関数 … 328
ceil … 287
ceilf … 310
char 型 … 39
const … 62
continue 文 … 145
cos … 273
cosf … 296
cosh … 276
coshf … 299
CPU 制御命令 … 27, 363
ctype.h … 185
C 言語 … 15

D

__DATE__ … 179
DI … 360
__directmap … 414
div … 192, 239
do 文 … 141

E

EI … 360
errno.h … 188
error.h … 187
EUC … 349
exit … 192, 237
exp … 279
expf … 302
extern … 55

F

fabs … 288
fabsf … 311
__FILE__ … 179
__flash … 408
float.h … 190
floor … 289
floorf … 312
fmod … 290
fmodf … 313

for 文 … 142
free … 233
frexp … 280
frexpf … 303

G

getchar … 222
gets … 223
goto 文 … 144

H

HALT … 363

I

if 〜 else 文 … 137
if 文 … 137
__interrupt … 358
__interrupt_brk … 358
isalnum … 199
isalpha … 199
isascii … 199
iscntrl … 199
isdigit … 199
isgraph … 199
islower … 199
isprint … 199
ispunct … 199
isspace … 199
isupper … 199
isxdigit … 199
itoa … 243

L

labs … 238
LANG78K … 349
ldexp … 281
ldexpf … 304
ldiv … 192, 239
limits.h … 188
__LINE__ … 179
log … 282
log10 … 283
log10f … 306
logf … 305
longjmp … 192, 204
ltoa … 243

M

malloc … 234
math.h … 189
matherr … 291
memchr … 256
memcmp … 254
memcpy … 251
memmove … 251
memset … 262
modf … 284
modff … 307

N

near/far 領域指定 … 28, 418
NONE … 349
NOP … 363
norec/__leaf 関数 … 26
norec 関数 … 339

O

__OPC … 392

P

pow … 285
powf … 308
printf … 192, 218
__putc … 226
putchar … 224
puts … 225

Q

-ql オプション … 329
qsort … 246

R

rand … 192, 244
realloc … 235
register … 55, 331
return 文 … 147
rolb … 386
rolw … 386
ROM 化関連セクション名 … 379
rorb … 386
rorw … 386
RTOS … 319
__rtos_interrupt 修飾子 … 397
RTOS 用タスク関数 … 27, 399
RTOS 用割り込みハンドラ … 27, 394
RTOS 用割り込みハンドラ修飾子 … 27, 397

S

saddr 領域利用 … 26, 333
sbrk … 192, 240
scanf … 192, 219
setjmp … 192, 204
setjmp.h … 185
sfr 変数 … 337
sfr 領域 … 26, 337
sin … 274
sinf … 297
sinh … 277
sinhf … 300
SJIS … 349
sprintf … 192, 209
sqrt … 286

sqrtf … 309
srand … 192, 244
sreg 宣言 … 333
sscanf … 192, 214
static … 55
stdarg.h … 186
__STDC__ … 179
stddef.h … 189
stdio.h … 186
stdlib.h … 186
STOP … 363
strbrk … 247
strcat … 253
strchr … 257
strcmp … 255
strcoll … 265
strcpy … 252
strcspn … 258
strerror … 263
string.h … 187
stritoa … 249
strlen … 264
strltoa … 249
strncat … 253
strncmp … 255
strncpy … 252
strpbrk … 259
strrchr … 257
strsbrk … 248
strspn … 258
strstr … 260
strtod … 192, 241
strtok … 192, 261
strtol … 230
strtoul … 230
struct … 148
strultoa … 249
strxfrm … 266
switch 文 … 138

T

tan … 275
tanf … 298
tanh … 278
tanhf … 301
__TIME__ … 179
toascii … 201
tolow … 202
_tolower … 202
tolower … 200
toup … 202
_toupper … 202
toupper … 200
typedef … 55

U

ultoa … 243
union … 151

V

va_arg … 206
va_end … 206
va_start … 206
va_starttop … 206
void … 76
void ポインタ … 76
volatile … 62
vprintf … 192, 220
vsprintf … 192, 221

W

while 文 … 140

Z

-zb … 409
-zf … 401

【あ行】

アセンブリ言語 … 15
エスケープ・シーケンス … 31
オブジェクト型 … 38

【か行】

外部オブジェクト定義 … 157
外部結合 … 36
外部定義 … 154
型指定子 … 56
型名 … 65
関係演算子 … 105
漢字 … 26, 349
関数 … 20
関数型 … 44
関数宣言子 … 64
関数定義 … 155
関数プロトタイプ有効範囲 … 36
関数有効範囲 … 35
カンマ演算子 … 124
キーワード … 32
記憶域クラス指定子 … 55
機械語 … 15
キャスト演算子 … 94
共用体 … 151
共用体型 … 43
区切り子 … 51
繰り返し文 … 128
合成型 … 45
構造体 … 148
構造体型 … 43
構造体指定子 … 58
構造体のポインタ … 150
構造体変数 … 148
後置演算子 … 80
コメント … 53
コンパイラ出力セクション名の変更 … 372
コンパイラ出力セクション名の変更機能 … 27

【さ行】

算術演算子 … 96
式文 … 128
識別子 … 36
シフト演算子 … 102
集成体型 … 43
16 進定数 … 47
10 進定数 … 47
条件演算子 … 119
乗算関数 … 27, 388
除算関数 … 27, 390
スカラ型 … 44
スタートアップ・ルーチン … 316, 379
スタック切り替え指定 … 354
整数型 … 39
整数定数 … 46
絶対番地配置指定 … 28, 414
選択文 … 128

【た行】

代入演算子 … 121
タグ … 61
タスク … 399
単項演算子 … 87
単純代入 … 122
定数 … 46
定数式 … 126
データ挿入関数 … 27, 392
適合型 … 44
デバイス種別 … 179
トライグラフ・シーケンス … 31

【な行】

内部結合 … 36
2 進定数 … 27, 383

【は行】

配列 … 149
配列型 … 43
配列宣言子 … 64
8 進定数 … 47
汎整数拡張 … 74
引数／返り値の int 拡張抑制方法 … 28, 409
ビット単位の AND 演算子 … 113
ビット単位の OR 演算子 … 115
ビット単位の排他 OR 演算子 … 114
ビット・フィールド … 365
ビット・フィールド宣言 … 27, 365
ファーム ROM 関数 … 27, 408
ファイル有効範囲 … 35
ブート領域からフラッシュ領域への関数呼び出し機能 … 27, 405
不完全型 … 43
複合代入 … 123
複合文 … 128
符号付き整数型 … 39
符号なし整数型 … 39
浮動小数点型 … 40
浮動小数点定数 … 46
フラッシュ領域配置方法 … 27, 401
フラッシュ領域分岐テーブル … 27, 402
ブロック有効範囲 … 35
分岐文 … 128
ヘッダ・ファイル … 185
ヘッダ名 … 52
ポインタ … 149
ポインタ宣言子 … 63

【ま行】

前処理指令 … 158
マクロ置換 … 170
マクロ名 … 179
マルチバイト文字 … 30
無結合 … 36
メモリ空間 … 323
メモリ操作関数 … 28, 412
メモリ・モデル指定 … 28, 422
文字型 … 43
文字定数 … 48
モジュール名変更 … 27, 385
文字列リテラル … 49

【ら行】

ラベル付き文 … 128
リエントラント … 192
レジスタ・バンク … 322
レジスタ・バンク指定 … 352
レジスタ変数 … 331, 26
列挙型 … 39
列挙型指定子 … 60
列挙定数 … 48
ローテート関数 … 27, 386
論理 AND 演算子 … 117
論理 OR 演算子 … 118

【わ行】

割り込み関数 … 26, 352
割り込み関数修飾子 … 26, 358
割り込み機能 … 26, 360

〔メ　モ〕

【発　行】

NECエレクトロニクス株式会社

〒211-8668　神奈川県川崎市中原区下沼部1753

電話（代表）：044(435)5111

お問い合わせ先

【ホームページ】

NECエレクトロニクスの情報がインターネットでご覧になれます。

URL（アドレス）　http://www.necel.co.jp/

【営業関係，技術関係お問い合わせ先】

半導体ホットライン　　電　話　：044-435-9494

（電話：午前 9:00～12:00，午後 1:00～5:00）　　E-mail　：info@necel.com

【資料請求先】

NECエレクトロニクスのホームページよりダウンロードいただくか，NECエレクトロニクスの販売特約店へお申し付けください。

C04.2T