お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010年4月1日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

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

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

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

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

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

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

ユーザーズ・マニュアル

RENESAS

保守／廃止

CC78Kシリーズ

Cコンパイラ

言語編

資料番号　EEU-655C（第4版）
発行年月　July 1991 P

ユーザーズ・マニュアル

NEC
電子デバイス

保守／廃止

CC78Kシリーズ

Cコンパイラ

言語編

保守／廃止

○文書による当社の承諾なしに本資料の転載複製を禁じます。
○本資料に記載された製品の使用もしくは本資料に記載の情報の使用に際して，当社は当社もしくは第三者の知的所有権その他の権利に対する保証または実施権の許諾を行うものではありません。上記使用に起因する第三者所有の権利にかかわる問題が発生した場合，当社はその責を負うものではありませんのでご了承ください。
○当社は品質，信頼性の向上に努めていますが，半導体製品はある確率で故障が発生します。当社半導体製品の故障により結果として，人身事故，火災事故，社会的な損害等を生じさせない冗長設計，延焼対策設計，誤動作防止設計等安全設計に十分ご注意願います。
○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頂く「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の上ご使用願います。
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
特別水準：輸送機器（自動車，列車，船舶等），交通用信号機器，防災／防犯装置，各種安全装置，生命維持を直接の目的としない医療機器
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。当社製品を上記の「標準水準」の用途以外でご使用をお考えのお客様は，必ず事前に当社販売窓口までご相談頂きますようお願い致します。
○この製品は耐放射線設計をしておりません。

M4 94.11

本製品は外国為替および外国貿易管理法の規定により戦略物資等（または役務）に該当しますので、日本国外に輸出する場合には、同法に基づき日本国政府の輸出許可が必要です。

○文書により当社の承諾なしに本資料の転載複製を禁じます。
○この製品を使用したことにより、第三者の工業所有権等にかかわる問題が発生した場合，当社製品の構造製法に直接かかわるもの以外につきましては、当社はその責を負いませんのでご了承ください。

保守／廃止

はじめに

「ＣＣ７８Ｋシリーズ 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－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，７８Ｋシリーズ応用製品の開発が可能となります。

「ＣＣ７８Ｋシリーズ Ｃコンパイラ ユーザーズ・マニュアル言語編」（以下本マニュアルとする）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後，本Ｃコンパイラをお使いの際には，「ＣＣ７８Ｋシリーズ Ｃコンパイラ ユーザーズ・マニュアル 操作編」をお読みください。

保守／廃止

【ターゲット・デバイス】

本Ｃコンパイラでは，次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。

シリーズ名	ターゲット・デバイス
７８Ｋ／０	μPD78001，78002 μPD78011，78012，78013，78014，78P014 μPD78022，78023，78024，78P024 μPD78042，78043，78044，78P044
７８Ｋ／Ⅱ	μPD78210注 μPD78212，78213，78214，78P214 μPD78217A，78218A，78P218A μPD78220，78224，78P224 μPD78233，78234，78237，78238，78P238 μPD78243，78244
７８Ｋ／Ⅲ	μPD78310注，78312注，78P312注 μPD78310A，78312A，78P312A μPD78320，78322，78P322，78323，78324，78P324 μPD78327，78328，78P328 μPD78330，78334，78P334 μPD78350，78P352

注　保守品です。

注意　上記の製品のうち，一部，開発中のものがあります。

【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挙げて説明します。

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

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を説明します。

第１３章　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

性能一覧，構文一覧，ライブラリ関数，ランタイム・ライブラリ一覧，saddr領域のラベル一覧などがあります。

【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併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７８Ｋシリーズ付属のユーザーズ・マニュアルを併せてご覧ください。

【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。

■本文中

…　：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

‘　’：‘　’で囲まれた文字そのもの

太文字：文字そのもの

＿　：重要箇所，使用例での下線は入力文字列

┆　：プログラム記述の省略形

（　）：（　）で囲まれた文字そのもの

／　：区切り記号

＼　：バックスラッシュ

■構文中

::＝　：左辺の意味を以降に示す

「　」：かっこ内は省略可能

｜　：行の左端にある‘｜’は‘または’の意味を示します。

保守／廃止

【関連資料】

本マニュアルに関連する資料を示します。

（1／3）

			資料名	資料番号
開発ツール	言語処理	共通	ＲＡ７８Ｋシリーズ　アセンブラ・パッケージ言語編	EEU-654
			ＲＡ７８Ｋシリーズ　アセンブラ・パッケージ操作編	EEU-662
			７８Ｋシリーズ　構造化アセンブラ・プリプロセッサ	EEU-643
			ＣＣ７８Ｋシリーズ　Ｃコンパイラ操作編	EEU-656
	ディバッガ	78K/0	ＩＥ－７８０００－Ｒ　ハードウエア編	EEU-750
			ＳＤ７８Ｋ／０　スクリーン・ディバッガ　ﾚﾌｧﾚﾝｽ編	作成中
			ＳＤ７８Ｋ／０　スクリーン・ディバッガ　入門編	作成中
		78K/Ⅱ	ＩＥ－７８２１０－Ｒ　ハードウエア取扱説明書	EEP-640
			ＩＥ－７８２１０－Ｒ　ソフトウエア取扱説明書	EEM-685
			ＩＥ－７８２１０－Ｒ　ｿﾌﾄｳｴｱ取扱説明書(MS-DOS)	EEM-677
			ＩＥ－７８２２０－Ｒ　ハードウエア取扱説明書	EEP-642
			ＩＥ－７８２２０－Ｒ　ソフトウエア取扱説明書	EEM-682
			ＩＥ－７８２２０－Ｒ　ｿﾌﾄｳｴｱ取扱説明書(MS-DOS)	EEM-678
			ＩＥ－７８２３０－Ｒ　ハードウエア編	EEM-682
			ＩＥ－７８２３０－Ｒ　ソフトウエア編	EEU-685
			ＩＥ－７８２４０－Ｒ　ハードウエア編	EEU-705
			ＩＥ－７８２４０－Ｒ　ソフトウエア編	EEU-706
			ＩＥ－７８２３０－Ｒ－Ａ　ハードウエア編	EEU-789
			ＩＥ－７８２４０－Ｒ－Ａ　ハードウエア編	EEU-796
			ＳＤ７８Ｋ／Ⅱ　スクリーン・ディバッガ　ﾚﾌｧﾚﾝｽ編	作成中
			ＳＤ７８Ｋ／Ⅱ　スクリーン・ディバッガ　入門編	作成中
		78K/Ⅲ	ＩＥ－７８３１０Ａ－Ｒ　ハードウエア編	EEU-645
			ＩＥ－７８３１０Ａ－Ｒ　ソフトウエア編	EEU-637
			ＩＥ－７８３１０Ａ－Ｒ　ソフトウエア編(MS-DOS)	EEU-646
			ＩＥ－７８３２０－Ｒ　ハードウエア編	EEU-709
			ＩＥ－７８３２０－Ｒ　ソフトウエア編	EEU-712
			ＩＥ－７８３３０－Ｒ　ハードウエア編	EEU-713
			ＩＥ－７８３３０－Ｒ　ソフトウエア編	EEU-714
			ＩＥ－７８３５０－Ｒ　ハードウエア編	EEU-754
			ＩＥ－７８３５０－Ｒ　ソフトウエア編	EEU-753

保守／廃止

			資　料　名	資料番号
デバイス	ユーザーズ・マニュアル	78K/0	μＰＤ７８０１Ｘシリーズ	IEU-780
		78K/Ⅱ	μＰＤ７８２１４シリーズ	IEM-5119
			μＰＤ７８２２４シリーズ	IEM-5019
			μＰＤ７８２１８Ａシリーズ	IEU-755
			μＰＤ７８２３４シリーズ	IEU-718
			μＰＤ７８２４４シリーズ	IEU-747
		78K/Ⅲ	μＰＤ７８３１２Ａ	IEM-5086
			μＰＤ７８３２２	IEU-619
			μＰＤ７８３２８	IEU-693
			μＰＤ７８３３４	IEU-729
			μＰＤ７８３５０	IEU-781
	レファレンス	78K/0	μＰＤ７８００Ｘシリーズ　ｲﾝｽﾄﾗｸｼｮﾝ･ｾｯﾄ	IEM-5546
			μＰＤ７８００Ｘシリーズ　インストラクション活用表	IEM-5545
			μＰＤ７８００Ｘシリーズ　モード・レジスタ活用表	IEM-5547
			μＰＤ７８０１Ｘシリーズ　ｲﾝｽﾄﾗｸｼｮﾝ･ｾｯﾄ	IEM-5521
			μＰＤ７８０１Ｘシリーズ　インストラクション活用表	IEM-5522
			μＰＤ７８０１Ｘシリーズ　モード・レジスタ活用表	IEM-5527
		78K/Ⅱ	７８Ｋ／Ⅱシリーズ　インストラクション活用表	IEM-5101
			７８Ｋ／Ⅱシリーズ　インストラクション・セット	IEM-5102
			μＰＤ７８２１４シリーズ　モード・レジスタ活用表	IEM-5100
			μＰＤ７８２１８Ａシリーズ　モード・レジスタ活用表	IEM-5532
			μＰＤ７８２２４シリーズ　モード・レジスタ活用表	IEM-999
			μＰＤ７８２３４シリーズ　モード・レジスタ活用表	IEM-5515
			μＰＤ７８２４４シリーズ　モード・レジスタ活用表	IEM-5528

			資料名	資料番号
デバイス	レファレンス	78K/Ⅲ	μＰＤ７８３１２Ａ　インストラクション・セット	IEM-5116
			μＰＤ７８３１２Ａ　インストラクション活用表	IEM-5115
			μＰＤ７８３１２Ａ　モード・レジスタ活用表	IEM-5118
			μＰＤ７８３２２　インストラクション・セット	IEM-601
			μＰＤ７８３２２　インストラクション活用表	IEM-602
			μＰＤ７８３２２　モード・レジスタ活用表	IEM-5501
			μＰＤ７８３３４　モード・レジスタ活用表	IEM-5518
			μＰＤ７８３２８　モード・レジスタ活用表	IEM-5514
			μＰＤ７８３５０　インストラクション・セット	IEM-5543
			μＰＤ７８３５０　モード・レジスタ活用表	IEM-5540

注意　関連資料の最新情報については，販売員にご連絡ください。

目次要約

保守／廃止

保守／廃止

目　次

図の目次

保守／廃止

表の目次

保守／廃止

第１章　概　　説

ＣＣ７８Ｋシリーズ　Ｃコンパイラは，７８ＫシリーズのＣ言語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。ＣＣ７８Ｋシリーズ　Ｃコンパイラにより，７８Ｋ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・ファイルが得られます。

1.1　Ｃ言語とアセンブリ言語

マイクロプロセッサに仕事をさせるには，プログラムやデータが必要です。これを人間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うことのでき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（コンピュータの理解できる言葉）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。これによりプログラマは，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。

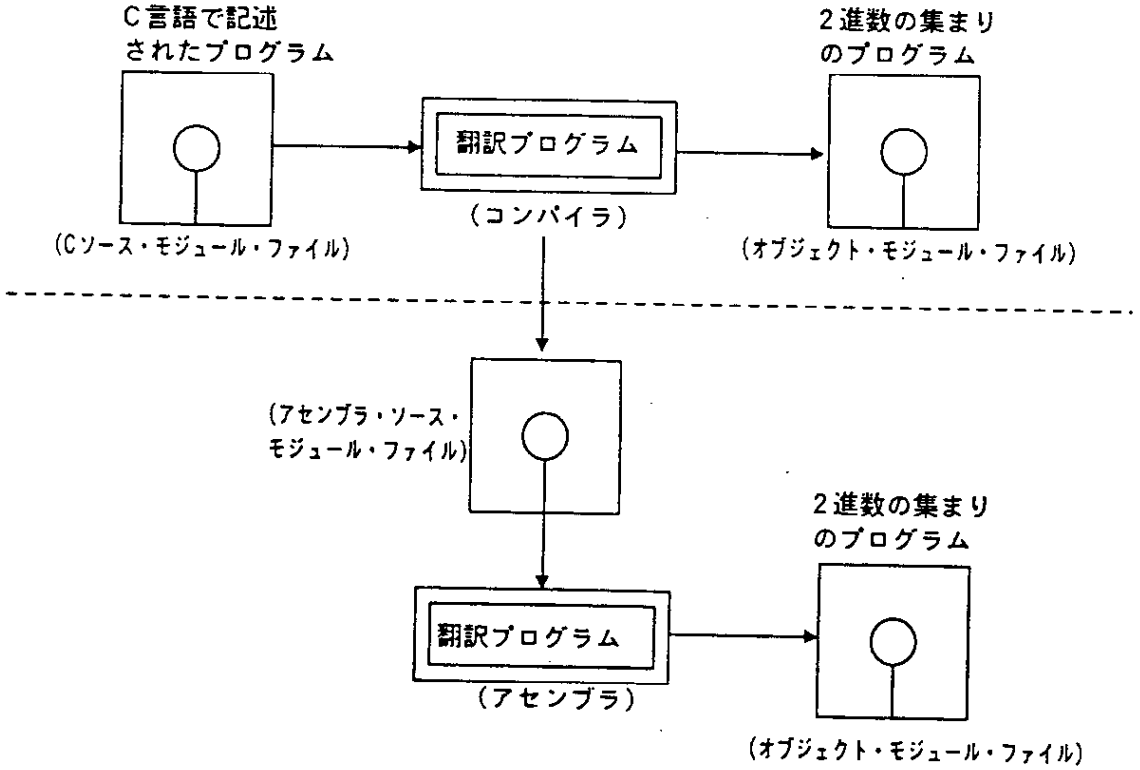
また，Ｃ言語ではプログラムを作成するための多くの部品（関数）が用意されているので，プログラマはこれらを組み合わせてプログラムを作成することができます。

Ｃ言語は，人間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。

本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，プログラマは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。本Ｃコンパイラの翻訳の流れを“図１－１　コンパイルの流れ”に示します。

図１－１　コンパイルの流れ



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ィバッガが必要になります。

本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。

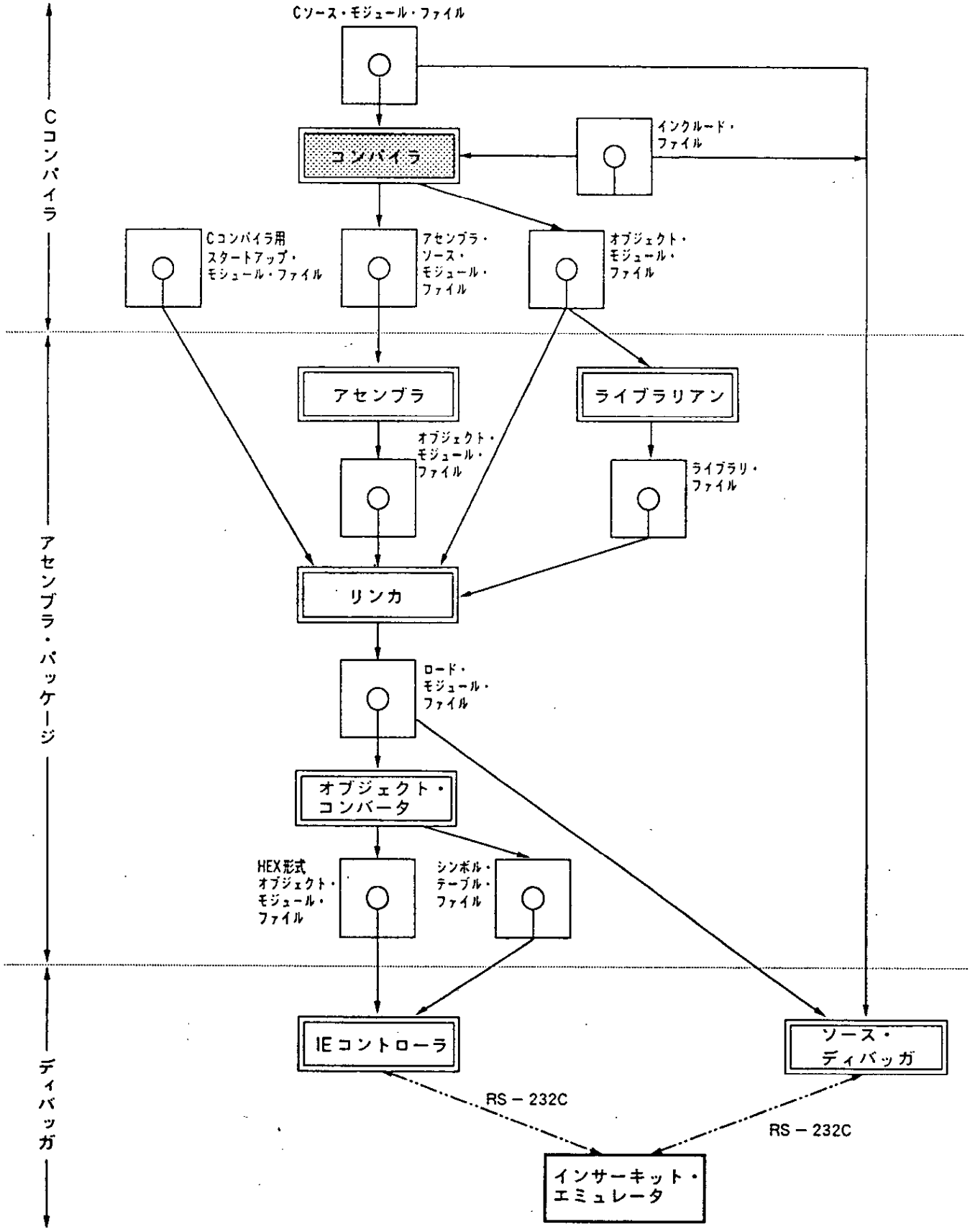
- エディタ……ソース・モジュール・ファイルの作成
- アセンブラ……アセンブラ・ソース・モジュール・ファイルのアセンブル
- リンカ……オブジェクト・モジュール・ファイルの結合
  リロケータブル・セグメントの配置アドレス決定
- オブジェクト・コンバータ……ＨＥＸファイルへの変換
- ライブラリアン……ライブラリ・ファイルの作成
- ソース・ディバッガ……Cソース・モジュール・ファイルのバグ取り

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・モジュールを作成する
③各モジュールをコンパイルする
④使用頻度の高いモジュールをライブラリ化する
⑤各モジュールをリンクする
⑥モジュールのディバグを行う
⑦オブジェクト・コンバータによりＨＥＸファイルに変換する

本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）が行え，効率のよいモジュールを作成できます。とくに，高速な処理を必要とする場合，または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　本Ｃコンパイラ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。

```
変数，定数の定義                    ―― 各データ，変数，マクロ命令の定義

main（引数）                        ―― 関数mainのヘッダ部分
 {
       命令１；
       命令２；                     ― 関数mainのボディ部分
       関数１（引数）；
       関数２（引数）；
}

関数１（引数）
 {
       命令１；                     ― 関数１
       命令２；
}

関数２（引数）
 {
       命令１；                     ― 関数２
       命令２；
}
```

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 FALSE   0                  ├─ #define xxx xxx ……前処理指令（マクロ定義）⑥
#define SIZE    200                ┘

char    mark[SIZE+1];              ┬─ char xxx……型宣言①
                                   └─ xx[xx] ……演算子②
main()
{
    int i, prime, k, count;        ─── int xxx……型宣言①

    count = 0;                     ─── xx = xx ……演算子②

    for ( i = 0 ; i <= SIZE ; i++)     ┐─ for(xx;xx;xx) xxx ; ……制御構造③
        mark[i] = TRUE;                ┘
    for ( i = 0 ; i <= SIZE ; i++) {
        if (mark[i]) {
            prime = i + i + 3;         ─── xxx = xxx + xxx + xxx ……演算子②
            printf("%6d",prime);       ─── xxx(xxx); ……外部定義⑤
            count++;
                if((count%8) == 0) putchar('¥n');   ─── if(xxx) xxx ;
            for ( k = i + prime ; k <= SIZE ; k += prime )      ……制御構造③
                    mark[k] = FALSE;
        }
    }
    printf("¥n%d primes found.",count);  ─ xxx(xxx); ……外部定義⑤
}


printf(s,i)                        ┐
char *s;                           │
int i;                             │
{                                  │
    int j;                         ├─ ……外部定義⑤
    char *ss;                      │
                                   │
    j = i;                         │
    ss = s;                        │
}                                  ┘

putchar(c)                         ┐
char c;                            │
{                                  ├─ ……外部定義⑤
    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，記憶クラスの宣言”をご覧ください。

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”をご覧ください。

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”をご覧ください。

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”をご覧ください。

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”をご覧ください。

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”をご覧ください。

1.4　プログラム開発をはじめる前に

実際にプログラム開発をはじめる前に，次のことを頭に入れておかなければなりません。

表１－１　本Ｃコンパイラの最大性能

項番	項　　目	制　限　値	ページ
1	複文，繰り返し制御文，選択制御文のネスト	４５	101
2	条件コンパイルのネスト	２５５	133
3	修飾宣言子のネスト	１２	33
4	式中のかっこのネスト	３２	61
5	マクロ名で意味を持つ文字数	３１	144
6	内部，外部シンボル名で意味を持つ文字数	７　注1	－
7	１ソース・モジュール・ファイル中のシンボル数	１０２４　注2	－
8	１ブロックでブロック・スコープを持つシンボル数	２５５　注2	16
9	１ソース・モジュール・ファイル中のマクロ数	１０２４　注3	144
１０	関数定義，関数呼び出しのパラメータ	３９	128
１１	１つのマクロ定義，マクロ呼び出しのパラメータ	３１	144
１２	１つの論理ソース行の文字数	５０９	－
１３	結合後の文字列リテラル内の文字数	５０９	29
１４	１つのオブジェクト・サイズ（データを示します）	６５５３５byte	－
１５	#includeのネスト	８	140
１６	switch文のcaseラベル数	２５７	104
１７	１コンパイル単位のソース行数	約３０００	－
１８	テンポラリ・ファイルを作成せずにコンパイルできるソース行数	約３００	－
１９	関数コールのネスティング	４０	128
２０	１ステートメントの前のラベル数	３３	103
２１	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	６５５３５byte	－
２２	１つの構造体または，共用体のメンバ数	１２７	120
２３	１つの列挙の列挙定数の数	１２７	39
２４	１つの構造体または，共用体における構造体または共用体のネスト	１５	120
２５	初期化子要素のネスト	１５	－

・ 項番５，６，１２，１３，１４，１５，２１以外は制限ではなく，保証されている最小の値です。

注1. コンパイラ・オプション（－Ｓ）により、シンボル名の長さを３０文字に拡張することができます。

2. 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できる制限値を示します。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，そのときの制限値はファイル・サイズにより変わります。

3. コンパイラの予約マクロ定義を含みます。

保守／廃止

1.5 本Cコンパイラの特徴

本Cコンパイラは，ＡＮＳＩにないＣＰＵのコードを生成する拡張機能を備えています。本Cコンパイラの拡張機能には，７８Ｋ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ものや，オブジェクト・コードを短縮し実行速度の向上を図るものがあります。拡張機能の詳細については，“言語編　第１１章　拡張機能”をご覧ください

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。

・callt領域を利用して関数を呼び出す……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……saddr領域
・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……noauto関数，norec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
・callf領域に関数本体を格納する……callf関数

①callt関数

呼び出される関数のアドレスをcallt領域に置き関数が呼び出されます。通常の呼び出しに比べ関数の切り換えが速くなります。また，オブジェクト・コードを短縮できます。

②レジスタ変数

レジスタ，またはsaddr領域に変数を取ることができ，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・コードを短縮できます。

③saddr領域

変数をsaddr領域に取ることができ，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・コードを短縮できます。

④sfr領域

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。

⑤noauto関数

前後処理（スタック・フレーム）のない関数を生成します。noauto関数の呼び出しで引数は，可能な限りレジスタ渡しになります。これにより，実行速度が向上しオブジェクト・コードが短縮ができます。

⑥norec関数

前後処理(スタック・フレーム)のない関数を生成します。norec関数の呼び出しで引数は，可能な限りレジスタ渡しになります。また，norec関数内で使用するオートマティック変数は，saddr領域に割り当てられます。これにより，実行速度の向上およびオブジェクト・コードの短縮ができます。

⑦bit型変数

bit型の変数を生成します。bit型変数によりsaddr領域へのビット・アクセスができます。

⑧ＡＳＭ文

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。

⑨漢　字

Ｃソース・ファイルのコメント文中に漢字（シフトＪＩＳコード）を記述することができます。

⑩割り込み関数

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。

⑪割り込み機能

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。

⑫callf関数

callf命令は，callf領域に関数本体を格納し，call命令に比べて短いコードで関数を呼ぶことを可能にします。これにより，オブジェクト・コードを短縮できます。

⑬１Mbyte拡張空間利用法（７８Ｋ／Ⅱのみサポート）

オブジェクトに１Mbyte拡張空間をアクセス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・ファイルを作成します。

⑭テーブル切り換え機能（７８Ｋ／Ⅲのみサポート）

Ｃコンパイラの出力するベクタ・テーブル，calltテーブルのアドレスを変更することができます。

第２章　Ｃ言語の基本構成

本章では，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，‘トークン’から構成されます。トークンには，次のものがあります。

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

```
#include "expand.h"

extern bit data1;                       ┐  extern ····· キーワード
extern bit data2;                       ┘  data1,data2 ····· 識別子

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
}

lprintf(s,i)                            ──  lprintf ····· 識別子
char *s;                                ┐  char, int ····· キーワード
int i;                                  ┘  s, i ····· 識別子
{
    int j;
    char *ss;                           ──  * ····· 演算子

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

■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。

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１０進数

０ １ ２ ３ ４ ５ ６ ７ ８ ９

２９個の図形文字　注1

！ ” ＃ ％ ＆ ’ （ ） ＊ ＋ ， － ． ／ ：

； ＜ ＝ ＞ ？ ［ ￥ ］ ＾ ＿ ｛ ｜ ｝ ～

およびスペース，水平タブ，垂直タブ，改ページなどを示す制御文字

注1. ＰＣ－ＤＯＳの場合，円記号‘￥’ではなくバックスラッシュ‘＼’です。

2. 文字定数，文字列リテラルおよびコメント中では，この他の文字を使用することができます。

■マルチバイト文字

ソース文字集合は，拡張文字集合中でマルチバイト文字を使用することができます。また，実行文字集合はシフト ＪＩＳ漢字コードのマルチバイト文字のみ使用できます。

■英字エスケープ・シーケンス

警報や改ページなどの非図形文字は，英字エスケープ・シーケンスによって表現します。英字エスケープ・シーケンスは，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エスケープ・シーケンスを次に示します。

表２－１　英字エスケープ・シーケンス一覧

英字エスケープ・シーケンス	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 キーワード

次のトークンは，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できません。

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

本Cコンパイラでは，拡張機能を実現するために次のトークンをキーワードとして追加しています。

callt……callt関数の宣言
callf……callf関数の宣言
sreg……sreg変数の宣言
noauto……noauto関数の宣言
norec……norec関数の宣言
bit……bit型変数の宣言

2.2 識別子

識別子は，次のものを示します。

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

識別子は，アンダ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。識別子として使用できる文字を次に示します。

_(ｱﾝﾀﾞｽｺｱ) 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。

2.2.1 識別子のスコープ

識別子は，宣言された場所によりその識別子が使用できる有効範囲が決まります。識別子の有効範囲のことを，識別子のスコープといいます。

識別子のスコープには，次のものがあります。

関数スコープ
ファイル・スコープ

ブロック・スコープ

関数プロトタイプ・スコープ

```
extern bit data1;                          ┐┐ data1,data2 ····· ファイル・スコープ
extern bit data2;                          ┘┘

void main()
{
        int cot ;                          ── cot ·············· ブロック・スコープ
        data1 = 1 ;
        data2 = 0 ;

        while(data1){
                data1 = data2 ;
                j1:                        ── j1 ···················· 関数スコープ
                testb(int cot);
        }
}
void testb(int x)
int x                                      ── x ····· 関数プロトタイプ・スコープ
{
      :
      :
      :
```

(1) 関数スコープ

関数スコープは，関数内全体を指します。関数スコープを持つ識別子は，指定された関数内のどこからでも参照できます。

関数スコープを持つ識別子は，ラベル名だけです。

(2) ファイル・スコープ

ファイル・スコープは，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・スコープを持ちます。ファイル・スコープを持つ識別子は，プログラム中のどこからでも参照できます。

(3) ブロック・スコープ

ブロック・スコープは，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示します。

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・スコープを持ちます。ブロック・スコープを持つ識別子は，指定したブロック内で有効です。

(4) 関数プロトタイプ・スコープ

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・スコープを持ちます。関数プロトタイプ・スコープを持つ識別子は，指定された関数内で有効です。

2.2.2 識別子の結合

異なった，または同一のスコープ内で１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見なされます。

識別子の結合には，外部結合と内部結合および結合なしがあります。

(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。

外部結合を次にあげます。

・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・スコープを持ち，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。

内部結合を次にあげます。

・ファイル・スコープを持ち，記憶クラス指定子staticを含むオブジェクトまたは関数

(3) 結合なし

結合がないものは，固有な実体です。

結合なしの例を次にあげます。

・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

2.2.3 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。

・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。

2.2.4 オブジェクトの持続期間

各オブジェクトは，そのライフタイムを決定する‘持続期間’を持っています。持続期間には，静的(static)なものと動的(automatic)なものの２つがあります。

(1) 静的持続期間

静的持続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存在して，最後に格納された値を保持します。

静的持続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) 動的持続期間

動的持続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。

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，保証されません。

動的持続期間を持つオブジェクトを次に示します。

・結合なしのオブジェクト

・記憶クラス指定子staticで宣言されていないオブジェクト

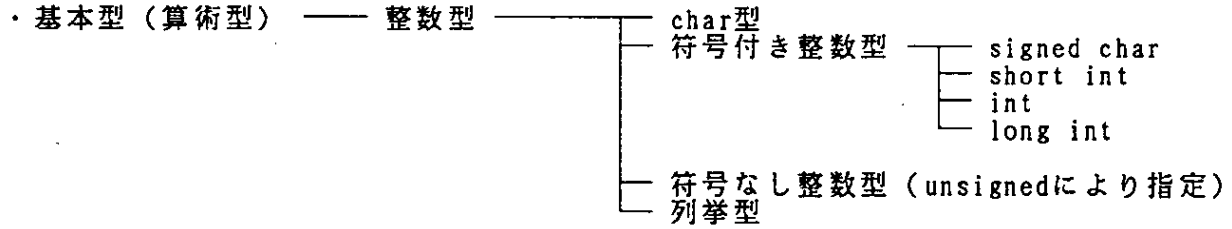
2.2.5 型

型は，オブジェクトに格納される値の性質を決定します。型は，オブジェクトを宣言する識別子の中で指定します。

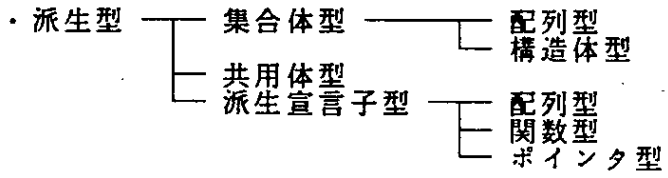
型は，宣言するものにより次の３種類に分けられます。

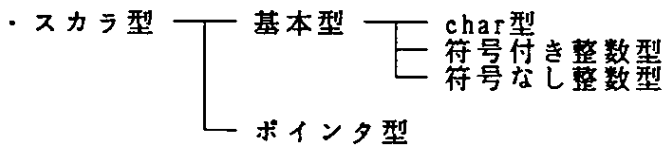
- オブジェクト型･･･オブジェクトを表す型
- 関数型･･･････････関数を表す型
- 不完全型･････････サイズに関する情報を持たないオブジェクトを表す型

型の分類を次に示します。



・不完全型 ― オブジェクトの大きさが確定しない配列や構造体，共用体とvoid型





(1) 基本型

基本型は，算術型とも呼ばれ，整数型からなります。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。

(a) 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
・符号つき整数型
・符号なし整数型
・列挙型

(i) 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、あふれが生じるとあふれた部分は無視されます。

(ii) 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共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。

(iii) 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変わるからです。

(iv) 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。

表２－２　基本型一覧

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

・signedは省略可能です。ただしchar型の場合コンパイル時の条件によりsigned charまたはunsigned charと判断されます。

・short intとintは，同じ値の範囲を持つが異なる型として扱われます。

・unsigned short intとunsigned intは，同じ値の範囲を持つが異なる型として扱われます。

(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

(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

(4) 派生型

派生型は基本型列挙型および不完全型から派生してできる型です。派生型には次の型があります。

・集合体型

・共用体型

・派生宣言子型

(a) 集合体型

集合体型には，配列型と構造体型の２種類があります。集合体型は連続して取られるメンバ・オブジェクトの集まりです。配列型は派生宣言子型にも含まれます。

(i) 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です。配列型は，１つのメンバ・オブジェクトから派生するもので，これを配列型派生といいます。したがってメンバ・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Ｔによって配列型派生を行う場合，それによってできる型をＴの配列と呼びます。

(ii) 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です。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。

(b) 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です。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。

(c) 派生宣言子型

派生宣言子型には，次の３種類の型があります。

・配列型

・関数型

・ポインタ型

(i) 配列型

配列型の派生宣言は配列型派生と呼ばれます。

(ii) 関数型

関数型は，指定する返り値を持つ関数を表します。関数型は，返り値の型とパラメータの数，およびパラメータの型によって指定します。関数は，その返り値型から派生します。返り値型がＴであれば，その関数はＴを返す関数と呼ばれます。また，返り値型から関数を作ることを関数型派生といいます。

(iii) 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。また，被参照型からポインタ型を作ることを，ポインタ型派生といいます。

(5) スカラ型

基本型と呼ばれる算術型と，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

・ポインタ型

2.2.6 適合型と合成型

(1) 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) 合成型

合成型は，2つの型の両方とも適合する型です。合成型は，適合する2つの型から作られます。合成型では次の事が成り立ちます。

- 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。

これらの規則は，派生してできた2つの型にも再帰的に適用されます。

合成型の例

ファイル・スコープを持つ2つの宣言が次のようであったとします。

```
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] ) ;
```

2.3 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３種類があります。

・整数定数
・列挙定数
・文字定数

2.3.1 整数定数

整数定数は，前もって指定された整数値です。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

整数定数の構文を次に示します。

整数定数::＝
　　１０進定数　「整数添字」
　|　８進定数　「整数添字」
　|　１６進定数　「整数添字」
１０進定数::＝
　　０以外の数字
　|　１０進定数　数字
０以外の数字::＝　次に示すいずれか１つ
　　１　２　３　４　５　６　７　８　９
８進定数::＝
　　０
　|　８進定数　８進数字
８進数字::＝　次に示すいずれか１つ
　　０　１　２　３　４　５　６　７

１６進定数::＝

０ｘ　１６進定数

| ０Ｘ　１６進定数

| １６進定数　１６進数字

１６進数字::＝　次に示すいずれか１つ

０　１　２　３　４　５　６　７　８　９

| ａ　ｂ　ｃ　ｄ　ｅ　ｆ

| Ａ　Ｂ　Ｃ　Ｄ　Ｅ　Ｆ

整数添字::＝

符号なし添字　「long添字」

| long添字　「符号なし添字」

符号なし添字::＝　以下のうちのいずれか１つ

ｕ　Ｕ

long添字::＝　以下のうちいずれか１つ

ｌ　Ｌ

整数定数の型は，次のリスト中でその値を表現できる最初のものです。

・添字なし１０進数……int,long int,unsigned long int

・添字なし８進数，１６進数……int,unsiged int,long int,
unsigned long int

・ｕまたはＵの添字付き……unsiged 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

(1) １０進定数

１０進定数は１０を基数とする整数値です。指定方法は０以外の数字を先頭にして，その後に０～９の数字を続けます。

(2) ８進定数

８進定数は８を基数とする整数値です。指定方法は０を先頭にして，その後に０～７の数字を続けます。

(3) １６進定数

１６進定数は１６を基数とする整数値です。指定方法は０ｘまたは０Ｘを先頭にして，

その後に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。

2.3.2 列挙定数

列挙定数は，列挙型変数の要素です。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。列挙定数は，列挙型変数の値を示すために使います。

列挙定数を宣言した識別子は int型になります。

列挙定数は，識別子で示します。

2.3.3 文字定数

文字列定数は’ X ’ または'ab'のようにシングルクォートで囲まれる１つまたはそれ以上の文字列です。

文字定数の構文を次に示します。

文字定数::＝
　　’ c文字の列’
c文字の列::＝
　　c文字
　　| c文字の列　c文字
c文字::＝
　　シングルクォート’，円記号￥および￥n
　　を除くマシンのソース文字セット中の任意の文字
　　| エスケープ・シーケンス
エスケープ・シーケンス::＝
　　単一エスケープ・シーケンス
　　| ８進エスケープ・シーケンス
　　| １６進エスケープ・シーケンス
単一エスケープ・シーケンス::＝ 以下のうちのいずれか１つ
　　￥’　￥”　￥?　￥￥
　　￥a　￥b　￥f　￥n　￥r　￥t　￥v
８進エスケープ・シーケンス::＝

¥　８進数字
| ¥　８進数字　８進数字
| ¥　８進数字　８進数字　８進数字

１６進エスケープ・シーケンス::＝
¥x　１６進数字
| １６進エスケープ・シーケンス　１６進数字

2.4 文字列

文字列リテラルは，”ｘｘｘ”のようにダブルコートで囲まれる０個以上の文字の並びです。

シングルクォート’は，それ自身またエスケープシーケンス＼’で表現します。また，ダブルクォート”は，エスケープシーケンス＼”で表現します。

配列要素は，char型を持ちます。

文字列リテラルの構文を次に示します。

文字列リテラル::＝
”「s文字の列」”

s文字の列::＝
s文字
| s文字の列　s文字

s文字::＝
ダブルクォート”，円記号¥および¥n
を除くマシンのソース文字セット中の任意の文字
| エスケープ・シーケンス

2.5 演算子

演算子はどのような評価を行うのかを指定するものです。演算子が作用する実体を演算数といいます。演算数は演算子によって指定された評価により，値や指示子を生成したり副作用やそれらの組合わせを生成します。

演算子の構文を次に示します。

演算子::= 以下のうちのいずれか１つ

```
[ ]   ( )   .   ->
++   --   &   *   +   -   ~   !   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できます。

‘#’および‘##’は前処理指令のマクロ定義にのみ使用します。

2.6 区切り子

区切り子は独立した文法または意味を持つシンボルです。しかし，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。しかし，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。

2.7 ヘッダ名

ヘッダ名は外部ソース・ファイル名を示します。これは‘#include’前処理指令でのみ使用されます。

ヘッダ名の構文を次に示します。

```
ヘッダ名::=
        <h文字の列>
    |   " q文字の列"
  h文字の列::=
        h文字
    |   h文字の列  h文字
    h文字::=
            改行文字および>を除くマシンのソース文字セット中の任意の文字
  q文字の列::=
        q文字
    |   q文字の列  q文字
    q文字::=
            改行文字および"を除くマシンのソース文字セット中の任意の文字
```

2.8 前処理数

前処理数は浮動小数点定数または整数定数に変換される前の数値です。前処理数の段階では型および値を持ちません。

前処理数の構文を次に示します。

前処理数::=

　　数字

　| . 数字

　| 前処理数 数字

　| 前処理数 非数字

　| 前処理数 e 符号

　| 前処理数 E 符号

　| 前処理数 .

前処理数は，数字またはピリオド‘.’を前に置いた数字から始まります。

数字に後には，次のどれかが続きます。

　　英字

　　アンダスコア

　　数字

　　ピリオド

　　e+

　　e−

　　E+

　　E−

2.9 コメント

コメントは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。本Ｃコンパイラはマルチバイト文字を識別することができ漢字を使用することができます。

第３章　型，記憶クラスの宣言

本章では，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’です。

宣言の構文を次に示します。

宣言::＝

　　宣言指定子　「初期宣言子リスト」　；

　宣言指定子::＝

　　記憶クラス指定子　「宣言指定子」

　　| 型指定子　「宣言指定子」

　　| 型修飾子　「宣言指定子」

　初期宣言子リスト::＝

　　初期宣言子

　　| 初期宣言子リスト　，　初期宣言子

　初期宣言子::＝

　　宣言子

　　| 宣言子　＝　初期化子

宣言の例を次に示します。

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

extern char    mark[SIZE+1];

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


    int count = 0;

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

```
extern・・・・・・・・記憶クラス指定子
char・・・・・・・・・・型指定子
void・・・・・・・・・・型指定子

auto・・・・・・・・・・記憶クラス指定子
int・・・・・・・・・・・型指定子
i, prime, k・・・初期宣言子リスト
count = 0・・・・・初期化
```

宣言指定子は，結合，記憶の持続期間および宣言子が示す実体の型の一部を示す指定子の列です。初期宣言子リストはコンマで区切られる宣言子の列で，各々の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。

あるオブジェクトに対する識別子が結合なしであると宣言されれば，そのオブジェクトに対する型はその宣言子の終わり，または初期化子を持てばその初期宣言子の終わりで完全になっていなければなりません。

結合がないオブジェクトに対する型は，その宣言子または初期宣言子の終わりで完全でなければなりません。

3.1 記憶クラス指定子

記憶クラス指定子は，オブジェクトの記憶クラスを指定するものです。記憶クラスは，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つの宣言中の宣言指定子中で，記憶クラス指定子は１つしか記述できません。記憶クラス指定子には，次の５つがあります。

- typedef
- extern
- static
- auto
- register

(1) typedef

typedef指定子は，構文の便宜上のみで記憶クラス指定子になっています。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”をご覧ください。

(2) extern

extern指定子は，外部変数であることを示します。

(3) 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。

(4) 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。

(5) 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。本C.コンパイラでは，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は，“第１１章　拡張機能”をご覧ください。

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
- signed
- unsigned
- 「構造体／共用体指定子」
- 「列挙指定子」
- 「typedef 名」

各型指定子のリストを次に示します。なお，これ以外の型指定はありません。
本コンパイラは，浮動小数点をサポートしていません。

- void……空の値の集合
- char……基本文字セットを格納できる大きさ
- 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(-32768～+32767)
- unsigned short，unsigned short int……符号なし整数(0～65535)
- int，signed，signed int……符号付き整数(-32768～+32767)
- unsigned，unsigned int……符号なし整数(0～65535)
- long，signed long，long int，
  signed long int……符号付き整数(-2147483648～+2147483647)
- unsigned long，unsigned long int……符号なし整数(0～4294967295)
- 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

コンマで区切られた型指定子は，同じ大きさです。

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，名付けられたメンバ・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，それぞれ異なった型を持つことができます。

構造体指定子と共用体指定子の構文を次に示します。

構造体／共用体指定子::=
　　構造体／共用体　「識別子」　{　構造体宣言リスト　}
　|　構造体／共用体　識別子
構造体／共用体::=
　　struct
　|　union
構造体宣言リスト::=
　　構造体宣言
　|　構造体宣言リスト　構造体宣言
構造体宣言::=
　　指定子／修飾子リスト　構造体宣言子リスト　;
指定子／修飾子リスト::=
　　型指定子　「指定子／修飾子リスト」
　|　型修飾子　「指定子／修飾子リスト」
構造体宣言子リスト::=
　　構造体宣言子
　|　構造体宣言子リスト　,　構造体宣言子
構造体宣言子::=
　　宣言子
　|　「宣言子」　:　定数式

構造体宣言の例

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

struct･･･構造体
tnode･･･識別子
構造体宣言リスト

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コンパイラに宣言します。個々の型はメンバ・オブジェクトと呼ばれ，それぞれに名前を付けることができます。メンバ・オブジェクトは宣言された順に，連続した領域を確保されます。

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コンパイラに宣言します。共用体のメンバ・オブジェクトは，すべて重なり合った領域を確保されます。

・メンバ・オブジェクトの宣言

メンバ・オブジェクトは，構造体宣言リストによって宣言します。メンバ・オブジェクトの型は，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・オブジェクトはビット・フィールドを持つことができます。

・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型を指定できます。修飾子のないintビット・フィールド，およびsigned intビット・フィールドの最上位ビットは，符号ビットとして判断されます。

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。０幅の無名のビット・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。

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

ビット・フィールドの指定

3.2.2 列挙指定子

列挙指定子は，順番付けされるオブジェクトを示します。列挙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。

列挙指定子の構文を次に示します。

```
列挙指定子::=
        enum 「識別子」 { 列挙子リスト }
    | enum 識別子
  列挙子リスト::=
        列挙子
      | 列挙子リスト , 列挙子
    列挙子::=
          列挙定数
          列挙定数 = 定数式
```

オブジェクトは，列挙子リスト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。この宣言で列挙の値は，‘{0，1，20，21}’になります。

```
enum hue {
        chartreuse,
        burgundy,
        claret=20,
        winedark
};
  /*...*/
enum hue col, *cp ;
  /*...*/
col = claret ;
cp = &col ;
  /*...*/
  /*...*/(*cp != burgundy) /*...*/
```

hue･･･識別子

列挙子リスト

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　｛　構造体宣言リスト　｝
または，
enum　識別子　｛　列挙子リスト　｝

タグは，リストによって宣言される構造体／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リストを省略することができます。次からの宣言では，タグが持つリストと同じ構造になります。同じスコープ中のそれ以後の宣言は中かっこで囲まれるリストを省略しなければなりません

次の型指定子

構造体／共用体　識別子

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。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えます。これは，同じスコープ中で，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’を指します。

```
typedef struct tnode TNODE;
struct tnode  {
        int count;
        TNODE *left, *right;
};
TNODE s[5], *sp;
sp = s;
sp->left = s[0];
sp->right = s[1];
s.right->count = 2;
```

次の例はお互いの構造体へのポインタを含む構造体を指定します。

```
struct s1  { struct s2 *s2p; /*...*/ } ;  /*  D 1  */
struct s2  { struct s1 *s1p; /*...*/ } ;  /*  D 2  */
```

囲んでいるスコープ中で‘s2’がタグとしてすでに宣言されていれば，宣言‘Ｄ１’は‘Ｄ２’で宣言されるタグ‘s2’ではなく前に宣言されたものを参照します。この文脈の依存性を除くために意味のない宣言‘struct s2;’をＤ１の前に挿入します。これは，内部スコープ中で新しいタグ‘s2’を宣言するものです。その次に宣言‘Ｄ２’は新しい型の指定を完全にします。

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。

非const修飾型を持つ左辺値使用を通してconst修飾型で定義されたオブジェクトを変更することはできません。また，非volatile修飾型を持つ左辺値の使用を通してvolatile修飾型で定義されたオブジェクトを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。その上，すべてのシーケンス・ポイントで最後にオブジェクトに格納される値は，認識されない因子による変更点を除き，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，“2.1 キーワード”で述べられたcallt,callf,noauto,norecを型修飾子として含めることは可能です。

適合する２つの修飾型に対しては，同じ適合型の修飾型を持たなければなりません。指定子リスト，または修飾子リスト内の型修飾子の順序は，指定した型には影響しません。なお，“2.1 キーワード”sregも型修飾子です。

次の例で‘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はできません。

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

型修飾子が集合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;              /*  ＝に対する制約である変更可能な
                           左辺値に違反している  */
pi = &ncs.mem;         /*  正しい  */
pi = &cs.mem;          /*  ＝に対する制約である型に違反している  */

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，識別子のスコープ，記憶の持続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。

宣言子の構文を次に示します。

```
宣言子::=
        「ポインタ」  直接宣言子
    ポインタ::=
            *  「型修飾子リスト」
          |  *  「型修飾子リスト」  ポインタ
        型修飾子リスト::=
                型修飾子
              |  型修飾子リスト  型修飾子
    直接宣言子::=
            識別子
          |  (  宣言子  )
          |  直接宣言子  [  「定数式」  ]
          |  直接宣言子  [  パラメータ型リスト  ]
          |  直接宣言子  [  「識別子リスト」  ]
        パラメータ型リスト::=
                パラメータリスト
              |  パラメータリスト  ,  . . .
            パラメータリスト::=
                    パラメータ宣言
                  |  パラメータリスト  ,  パラメータ宣言
                パラメータ宣言::=
                        宣言指示子  宣言子
                      |  宣言指示子  「抽象宣言子」
        識別子リスト::=
                識別子
              |  識別子リスト  ,  識別子
```

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（指す）するものです。

ポイントするオブジェクトの型を宣言することにより，ポインタ変数は演算を行う際の変位値を得ます。

ポインタ宣言

T　D1

T　：型T1（例　int）を指定する宣言指定子

D1：識別子D（例　ident）を含む宣言子

このときD1は，次の形式になります。

*「型修飾子リスト」　D

この宣言によりD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の内容は変換されてはならないが，ptr_to_constant自身は他のconst intを指すように変えられてもよいことを表しています。同様に二行目の宣言では，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します。

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

T　D1

T　：型T1（例　int）を指定する宣言指定子

D1：識別子D（例　ident）を含む宣言子

このときD1は，次の形式になります。

D［「定数式」］

この宣言によりD1は，T1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‘fa［］’と，17の要素数からなるchar型のポインタの配列‘afp［］’を宣言しています。

```
char fa[11], *afp[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が他の所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;
extren 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

T　D1

T　：型T1（例　int）を指定する宣言指定子

D1：識別子D（例　ident）を含む宣言子

このときD1は，次の形式になります。

D（パラメータ型リスト）

または

D（「識別子リスト」）

この宣言によりDは，パラメータ型リストで指定したパラメータを持ち，T1型の値を返す関数になります。関数のパラメータを表す識別子は，パラメータ型リストによって指定します。パラメータ型リストにより，パラメータ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（，･･･）で書かれたリストをパラメータに変換します。また，パラメータがない関数は，パラメータ型リストを‘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。

型名の構文を次に示します。

型名::=
　　指定子修飾子リスト　「抽象宣言子」
　抽象宣言子::=
　　ポインタ
　　| 「ポインタ」 直接抽象宣言子
　直接抽象宣言子::=
　　（ 抽象宣言子 ）
　　| 「直接抽象宣言子」 ［ 「定数式」 ］
　　| 「直接抽象宣言子」 ［ 「パラメータ型リスト」 ］

次に型名の例をあげます。

・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 *() ……パラメータ指定を持たない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) …パラメータを持たず，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型のパラメータ，およびその他の不定個のパラメータを持ち，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に対する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名の構文を次に示します。

typedef名::=

識別子

次の例でtype_identは，宣言指定子Tによって指定される型（T1とします）を持つtypedef名として定義されます。これにより識別子type_identは，T1型を持ちます。

```
typedef T type_ident;
type_ident D;
```

次の例で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。オブジェクトdistanceは，他のintオブジェクトと適合します。

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

次の宣言で，t1型とtp1の指す型，およびstruct s1型はint型と適合します。しかし，struct s2型，t2型，tp2の指す型，およびint型とは適合しません。

```
typedef struct s1 {int x ;} t1, *tp1 ;
typedef struct s2 {int x ;} t2, *tp2 ;
```

次に示す構造

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

は，signed int型を持つtypedef名t，int型を持つtypedef名plainならびに[0,15]の範囲中に値を含む名前付けられたt，（アクセスされたならば）[-16,+15]の範囲中に値を含むであろう名前なしのconst修飾されたビットフィールドおよび[-16,+15]の範囲に値を含む名前付けられたrという３つのビットフィールドメンバを持つ構造体を宣言します。最初の２つのビットフィールド宣言は，unsignedが（tを構造体メンバの名前にする）型指定子であるか，constが（typedef名として参照できるtを修飾する）型修飾子であるかで異なります。内部スコープ中で，これらの宣言の後ろに次の宣言，

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

があれば，関数fは“signed int型を持つ１つの名前なしの仮引数を持つsigned intを返す関数へのポインタ型を持つ１つの名前なしの仮引数を持つsigned intを持つ関数”型として宣言され，識別子tはlong型として宣言されます。

もう一方では，typedef名はコードをいっそう読みやすくするために使用されてもかまいません。次に示すsignal関数の３つの宣言はtypedef名を使用しない最初の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。

初期化子の構文を次に示します。

初期化子::=
　　代入式
　| { 初期化子リスト }
　| { 初期化子リスト , }
初期化子リスト::=
　　初期化子
　| 初期化子リスト , 初期化子

初期化できるのは，オブジェクト型か大きさのわからない配列型です。初期化子リストには，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。

静的持続期間を持つオブジェクトと集合体／共用体型を持つオブジェクトに対する初期化子，または初期化子リスト中のすべての式は定数式によって指定します。

ブロック・スコープを宣言する識別子で，外部結合または内部結合を持つ識別子は初期化できません。

(1) 静的持続期間を持つオブジェクト

静的持続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，静的持続期間を持つポインタ型のオブジェクトは，ヌル・ポインタ定数に初期化されます。

(2) 動的持続期間を持つオブジェクト

初期化を行わなかった動的持続期間を持つオブジェクトの初期値は，保証されません。動的持続期間を持つ構造体または共用体オブジェクトは，初期化子リストか適合型を持つ単一の式で初期化します。

(3) 文字配列

文字配列は，文字列リテラル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，上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，メンバを文字列リテラルで初期化される‘char配列’型を持つオブジェクトへのポインタpとして定義します。

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

(4) 集合体，共用体オブジェクトの初期化

・集合体

集合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リストで行います。指定する初期化リストは中かっこで囲みます。

リスト中の初期化子が集合体のメンバ数より少ない場合，残りのメンバは静的持続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。

・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。

配列の初期化は，次のように指定できます。

```
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，前にあげた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，前にあ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。本章では，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，切り捨てもしく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。型変換の一覧表を“表４－１　型変換一覧”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
(signed) char	＋	×	○	○	○	○	○	○	○
	－	×	＋	○	＋	○	＋	○	＋
unsigned char		±	×	○	○	○	○	○	○
(signed) short int	＋			×	○	×	○	○	○
	－			×	＋	×	＋	○	＋
unsigned short int				±	×	±	×	○	○
(signed) int	＋			×	○	×	○	○	○
	－			×	＋	×	＋	○	＋
unsigned int				±	×	±	×	○	○
(signed) long int	＋							×	○
	－							×	＋
unsigned long int								±	×

signedは省略可能です。ただし，char型の場合に限り，コンパイル時の条件によってsigned charまたはunsigned charと見なされます。

○　：正しく変換されます。

×　：型変換は，行なわれません。

＋　：正しい値になりません。（符号なし整数と見なされます）

±　：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。

空欄：変換時にあふれた部分は，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。

4.1 算術演算数

(1) 文字と整数（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，もしくは符号なしのもの，または列挙型を持つオブジェクトは，いずれの場合も，int型で表現できる範囲内にあれば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“整数拡張”と呼びます。その他のすべての算術型は，整数拡張によって変わることはありません。

整数拡張は，符号も含めて値を保持します。修飾子なしのcharは，通常符号付きとして扱われます。

(2) 符号付き整数と符号なし整数

符号なし整数がより大きい符号付き整数へ変換される場合，符号なし整数の値は変わりません。符号付き／符号なし整数がより小さい符号付き整数に縮小される場合，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，表現できなかった値は切り捨てられます。

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。

表４－２　符号付き整数から符号なし整数への変換

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

○：正しく変換されます。

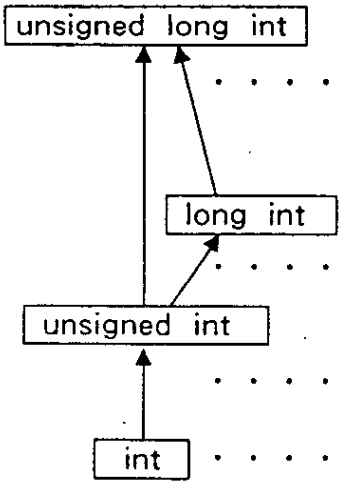
＋：正の整数に変換されます。

／：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。

図４－１　通常の算術変換



上記以外の場合で,一方がunsigned long int型である場合と,一方がlong int型でもう一方がunsigned int型のとき long int型で,unsigned int型の値を表現できない場合は,両方の演算数はunsigned long int型に変換されます。

上記以外の場合で,一方がlong int型であり,もう一方の演算数の値を全て表現できる場合は,long int型に変換されます。

上記以外の場合で,一方がunsigned int型であれば,もう一方はunsigned int型になります。

上記以外の場合は,int型に変換されます。

4.2 他の演算数

(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，－－演算子の演算数または，演算子もしくは代入演算子の左演算数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。

左辺値が修飾型を持てば，その値は左辺値の型の非修飾版を持ちます。これ以外で，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&演算子の演算数を除いて，'…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ができません。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。

修飾子qに対して，非q修飾型へのポインタは，その型のq修飾版へのポインタに変換されてもかまいません。その値はもとのポインタに格納され，変換されたポインタはもとのポインタに等しくなります。

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

す。‘ヌル・ポインタ’と呼ばれるポインタは，どんなオブジェクトまたは関数へのポインタとも等しくないことが保証されています。

‘…へのポインタ’型でキャストされる２つのヌル・ポインタは等しくなります。

第５章　演算子と式

本章では，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，算術演算や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。特にＣ言語には，ビット演算やアドレス演算を行う演算子があります。

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合せを実行する演算子および演算数の列です。

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

char    mark[SIZE+1];       ―― + ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・算術演算子

main()
{
    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) putchar('¥n');   ―― == ・・・・・・・・・関係演算子
            for ( k = i + prime ; k <= SIZE ; k += prime ) ┬― += ・・・代入演算子
                    mark[k] = FALSE;                       └― <= ・・・関係演算子
        }
    }
    lprintf("Total %d¥n",count);
loop1:
    goto loop1;
}

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

    j = i;
    ss = s;
}

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

“表５－１　演算子の評価順序”にC言語で使用される演算子とその優先順位を示します。

表５－１　演算子の評価順序

分　類	演算子	結合	優先順位
後　置	[] () . －＞ ＋＋ －－	→	高　い
単　項	＋＋ －－ ＆ ＊ ＋ － ～ ！ sizeof	←	
キャスト	（型名）	←	
乗　法	＊ ／ ％	→	
加　法	＋ －	→	
シフト	＜＜ ＞＞	→	
関　係	＜ ＞ ＜＝ ＞＝	→	
等　値	＝＝ ！＝	→	
ﾋﾞｯﾄごとのＡＮＤ	＆	→	
ﾋﾞｯﾄごとの排他的ＯＲ	＾	→	
ﾋﾞｯﾄごとのＯＲ	｜	→	
論理ＡＮＤ	＆＆	→	
論理ＯＲ	｜｜	→	
条　件	？ ：	←	
代　入	＝ ＊＝ ／＝ ％＝ ＋＝ －＝ ＜＜＝ ＞＞＝ ＆＝ ＾＝ ｜＝	←	
コンマ	,	→	低　い

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，“2.3　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，“2.4　文字列”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，その名前が示すようにオブジェクトの後ろに置かれる演算子です。
後置演算子の構文を，次に示します。

後置式::=
　　一次式
　| 後置式　[　添字式　]
　| 後置式　(　「引数式リスト」　)
　| 後置式　.　識別子
　| 後置式　－＞　識別子
　| 後置式　＋＋
　| 後置式　－－

引数式リスト::=
　　代入式
　| 引数式リスト　,　代入式

(1) 配列添字

後置演算子　　[] 添字演算子

[] 添字演算子

【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・オブジェクトを持ち，各々のメンバは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。

【構　文】

後置式　［添字式］

【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。

(2) 関数呼び出し

() 関数呼び出し

【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。

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い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

```
extern int　識別子　()　;
```

が暗黙的に行われます。

関数呼び出しでは，関数プロトタイプ宣言を行うことにより効率のよいオブジェクトが生成されます。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，整数拡張されます。これは，‘ディフォールトの整数拡張’と呼ばれます。

【構　文】

```
後置式　(　「引数式リスト」　)
```

【注　意】

呼び出す関数は，voidまたは配列を除くオブジェクトを返す関数で，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パラメータ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。

(3) 構造体と共用体のメンバ

後置演算子	. , −>

① . ドット

【機　能】

'.' は，構造体または共用体のメンバ・オブジェクトを指定します。後置式の値は指定したメンバの値になります。

【構　文】

後置式 . 識別子

② −>矢印

【機　能】

'−>' は，構造体または共用体のメンバ・オブジェクトを指定します。後置式の値は指定したメンバの値になります。

'.', '−>' 演算子の例

```
union {
      struct {
             int     alltype ;
             } n ;
      struct {
             int     type ;
             int     intnode ;
             } ni ;
      struct {
             int     type ;
             double  doublenode ;
             }nf, *nf_p ;
      } u ;
      /*...*/
      u.nf.type = 1 ;
      u.nf_p->doublenode = 3.14 ;
      /*...*/
      if(u.n.alltype == 1)
             /*...*/ sin(u.nf.doublenode) /*...*/
```

【構　文】

後置式 −> 識別子

(4) 後置インクリメントと後置デクリメント演算子

後置演算子	＋＋，－－

① 後置インクリメント演算子

【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。

【構　文】

後置式　＋＋

② 後置デクリメント演算子

【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。

【構　文】

後置式　－－

【注　意】

後置インクリメントと後置デクリメント演算子の演算数は，修飾された，またはされていないスカラ型の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と前置デクリメント演算子

＋＋　－－

・アドレスと間接演算子

＆　＊

・単項算術演算子

＋　－　～　！

・sizeof演算子

sizeof

単項演算子の構文を次に示します。

単項式::＝

後置式
| ＋＋　単項式
| －－　単項式
| 単項演算子　キャスト式
| sizeof　単項式
| sizeof　（　型名　）

単項演算子::＝　以下のうちのいずれか１つ

＆　＊　＋　－　～　！

(1) 前置インクリメントと前置デクリメント演算子

単項演算子	＋＋，－－

① 前置インクリメント演算子

【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
```

【構　文】

＋＋　単項式

② 前置デクリメント演算子

【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
```

【構　文】

－－　単項式

【注　意】

前置インクリメントと前置デクリメント演算子の演算数は，修飾された，またはされていないスカラ型の変換可能な左辺値です。

(2) アドレスと間接演算子

単項演算子	&, *

① 単項&演算子

【機　能】

指定したオブジェクトのアドレスを返します。

【構　文】

& キャスト式

② 単項*演算子

【機　能】

指定したポインタが指す値を返します。

【構　文】

* キャスト式

【注　意】

単項&演算子の演算数は，register記憶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&演算子の演算数に使用できません。

単項*演算子の演算数は，ポインタ型です。

(3) 単項算術演算子

単項演算子	+, −, ~, !

+, −, ~, !演算子

【機　能】

単項＋演算子は，演算数に対して正の整数拡張を行います。

単項－演算子は，演算数に対して負の整数拡張を行います。

単項 ~演算子は，演算数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! 演算子は演算数が‘0’のとき‘1’を返します。それ以外のときは‘0’を返します。

【構　文】

＋　キャスト式

－　キャスト式

~　キャスト式

!　キャスト式

【注　意】

単項＋演算の演算数は，スカラ型です。

単項－演算の演算数は，算術型です。

単項 ~演算の演算数は，整数型です。

単項! 演算の演算数は，スカラ型です。

(4) sizeof演算子

sizeof演算子

【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Ｉ／Ｏシステムとのやり取りに使用します。

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。

```
sizeof array / sizeof array[0] ;
```

【構　文】

```
sizeof　単項式
sizeof　（　型名　）
```

【注　意】

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左辺値は適用できません。

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。おもにポインタ型の変換を行う場合キャスト演算子を使用します。

キャスト演算子

【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。

【構　文】

（　型名　）　キャスト式

【注　意】

void型を指定しない限り型名はスカラ型を指定し，変換するオブジェクトもスカラ型です。

5.5 算術演算子

算術演算子は，優先順位により乗法演算子と加法演算子に分かれます。乗法演算子は，２つの演算数の積，商，余りを求め加法演算子は，２つの演算数の和，差，を求めます。２つの演算数の和，差，積，商，余りは，それぞれ‘二項＋演算子’，‘二項－演算子’，‘二項*演算子’，‘二項／演算子’，‘二項％演算子’によって示します。

・乗法演算子　　＊，　／，　％

・加法演算子　　＋，　－

二項＋演算子の各演算数は，算術型か，一方がオブジェクトへのポインタで，もう一方が整数型です。

二項－演算子の各演算子は次に示す型です。

・両方とも算術型

・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタ

・第１演算数がオブジェクトへのポインタで，第２演算数が整数型

二項*演算子と二項／演算子の各演算数は算術型です。また，二項％演算子の演算数は整数型です。

表５－２　乗法演算

a ／ b		b	
		＋	－
a	＋	＋	－
	－	－	＋

a ％ b		b	
		＋	－
a	＋	＋	＋
	－	－	－

除算は符号を取った数値によって行ない，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ないます。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つの演算数の符号のみの計算結果を示しています。

(1) 乗法演算子

乗法演算子	*, ／, %

① 二項*演算子

【機　能】

*演算子は，２つの演算数の積を求めます。

【構　文】

乗法式　*　キャスト式

② 二項／演算子

【機　能】

／演算子は，第１演算数を第２演算数で除算した商を求めます。

【構　文】

乗法式　／　キャスト式

③ 二項％演算子

【機　能】

％演算子は，第１演算数を第２演算数で除算した余りを求めます。

【構　文】

乗法式　％　キャスト式

【注　意】

乗法演算子の各演算数は，算術型です。

(2) 加法演算子

加法演算子	+, −

① 二項＋演算子

【機　能】

2つの演算数の和を求めます。

【構　文】

加法式　＋　乗法式

② 二項－演算子

【機　能】

第1演算数から第2演算数を引いた差を求めます。

【構　文】

加法式　－　乗法式

【注　意】

2つの演算数は，算術型または一方がオブジェクトへのポインタで，もう一方が整数型です。

減算を行えるのは，次に示す型の場合です。

・両方とも算術型のとき

・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタのとき

・第1演算数がオブジェクトへのポインタで，第2演算数が整数型のとき

5.6 シフト演算子

シフト演算子は，演算子の演算数を記号で示された方向へ移動します。シフトさせる演算子が符号付き型であれば算術シフトが，符号なし型であれば論理シフトが行われます。

シフト演算子の構文を次に示します。

シフト式::=

　　加法式

　| シフト式　<<　加法式

　| シフト式　>>　加法式

表５－３　シフト演算

a << b		注1 b
a	＋	0
	－	0

a >> b		注1 b
a	＋	0
	－	－1

注1.・ビット幅以上の数値またはシフト演算の結果がオーバフローした場合を表に示します。

・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。

シフト演算子　　＜＜，＞＞

① ＜＜演算子

【機　能】

左演算数を右演算数の下位１６ビットの値分左にシフトし，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。‘E1’がunsigned long型の場合結果は，‘ULONG_MAX+1’または‘UINT_MAX+1’で割った余りになります。

‘ULONG_MAX+1’および‘UINT_MAX+1’は，‘limits.h’中に定義されています。

‘limits.h’については，“10.2　(8)limits.h”を参照して下さい。

【構　文】

シフト式　＜＜　加法式

② ＞＞演算子

【機　能】

左演算子を右演算子の分右にシフトします。‘E1’が符号なし型の場合，論理シフトが行われます。論理シフトでは，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，算術シフトが行われます。算術シフトの場合，空いたビットに符号と同じものを入れます。結果の値は，‘E1’を２の‘E2’乗で割った値です。

【構　文】

シフト式　＞＞　加法式

【注　意】

シフトさせる演算数は，整数型でなければなりません。

5.7 関係演算子

関係を示す演算子には，２つの演算数の大小関係を示す‘関係演算子’と，等しいか等しくないかを示す‘等値演算子’があります。

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。

関係演算子の構文を次に示します。

関係式::=

　　シフト式
　| 関係式　＜　シフト式
　| 関係式　＞　シフト式
　| 関係式　＜＝　シフト式
　| 関係式　＞＝　シフト式

等値演算子の構文を次に示します。

等値式::=

　　関係式
　| 等値式　＝＝　関係式
　| 等値式　！＝　関係式

関係演算子で大小関係を比較できるのは，次に示す型の場合です。

・両方とも算術型のとき
・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタのとき
・両方とも適合型を持つ不完全型へのポインタのとき

等値演算子で比較できるのは，次に示す型の場合です。

・両方とも算術型のとき
・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタのとき
・一方がオブジェクト型または不完全型へのポインタで，もう一方がvoidポインタのとき
・一方がポインタでもう一方がヌル・ポインタ定数であるとき

(1) 関係演算子

関係演算子	<, >, <=, >=

① <演算子

【機　能】

第1演算数が第2演算数より小さいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　文】

関係式 < シフト式

② >演算子

【機　能】

第1演算数が第2演算数より大きいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　文】

関係式 > シフト式

③ <=演算子

【機　能】

第1演算数が第2演算数より小さいか，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。

【構　文】

関係式 <= シフト式

④　＞＝演算子

【機　能】

第１演算数が第２演算数より大きいか，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。

【構　文】

関係式　＞＝　シフト式

【注　意】

大小関係を比較できるのは，次に示す型の場合です。

・両方とも算術型のとき

・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタのとき

・両方とも適合型を持つ不完全型へのポインタのとき

(2) 等値演算子

等値演算子	＝＝，！＝

① ＝＝演算子

【機　能】

２つの演算数が等しい場合‘１’を返し，等しくない場合に‘０’を返します。

【構　文】

等値式　＝＝　関係式

② ！＝演算子

【機　能】

２つの演算数が等しくない場合‘１’を返し，等しい場合に‘０’を返します。

【構　文】

等値式　！＝　関係式

【注　意】

比較できるのは，次に示す型の場合です。

・両方とも算術型のとき

・両方とも適合型を持つオブジェクト（修飾，非修飾を含む）へのポインタのとき

・一方がオブジェクト型または不完全型へのポインタで，もう一方がvoidポインタのとき

・一方がポインタでもう一方がヌル・ポインタ定数であるとき

5.8 ビットごとの論理演算子

ビットごとの論理演算子は，オブジェクトの値をビット単位で論理演算します。ビットごとの論理演算にはＡＮＤ，排他的ＯＲ，ＯＲがあり，それぞれ‘＆’，‘＾’，‘｜’の演算子で示します。

ビットごとの論理演算子の構文を次に示します。

ＡＮＤ式::＝
　　等値式
　　｜　ＡＮＤ式　＆　等値式
排他的ＯＲ式::＝
　　ＡＮＤ式
　　｜　排他的ＯＲ式　＾　ＡＮＤ式
ＯＲ式::＝
　　排他的ＯＲ式
　　｜　ＯＲ式　｜　排他的ＯＲ式

(1) ビットごとのＡＮＤ演算子

ビットごとの論理演算子	&

二項＆演算子

【機　能】

‘&’はビットごとの論理積を返す，ビットごとのＡＮＤ演算子です。ビットごとの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。

ビットごとのＡＮＤ演算子は，‘二項＆演算子’によって指定します。

表５－４　ビットごとのＡＮＤ演算子

		左演算数の１ビットの値	
		1	0
右演算子の１ビットの値	1	1	0
	0	0	0

【構　文】

ＡＮＤ式　＆　等値式

【注　意】

二項＆演算子の各演算数は，整数型です。

(2) ビットごとの排他的ＯＲ演算子

ビットごとの論理演算子 ^

二項 ^ 演算子

【機　能】

‘^’はビットごとの排他的論理和を返す，ビットごとの排他的ＯＲ演算子です。ビットごとの排他的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。

表５－５　ビットごとの排他的ＯＲ演算子

		左演算数の１ビットの値	
		1	0
右演算数の１ビットの値	1	0	1
	0	1	0

【構　文】

排他的ＯＲ式　^　ＡＮＤ式

【注　意】

二項 ^ 演算子の各演算数は，整数型です。

(3) ビットごとのＯＲ演算子

ビットごとの論理演算子 |

二項｜演算子

【機　能】

‘｜’はビットごとの論理和を返す，ビットごとの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。

表５－６　ビットごとのＯＲ演算子

		左演算数の１ビットの値	
		1	0
右演算数の１ビットの値	1	1	1
	0	1	0

【構　文】

ＯＲ式　｜　排他的ＯＲ式

【注　意】

二項｜演算子の各演算数は，整数型です。

5.9 論理演算子

論理演算子は，２つの演算数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。両論理演算子の各演算数はスカラ型でともに int型の値‘0’または‘1’を返します。

論理演算子の構文を次に示します。

論理ＡＮＤ式::＝
　　ＯＲ式
　｜　論理ＡＮＤ式　＆＆　ＯＲ式
論理ＯＲ式::＝
　　論理ＡＮＤ式
　｜　論理ＯＲ式　｜｜　論理ＡＮＤ式

(1) 論理ＡＮＤ演算子

論理演算子	＆＆

二項＆＆演算子

【機　能】

二項＆＆演算子は，２つの演算数の論理ＡＮＤ演算を行います。論理ＡＮＤ演算は，２つの演算数が‘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。

表５－７　論理ＡＮＤ演算子

		左演算数の値	
		0	0 以外
右演算数の値	0	0	0
	0 以外	0	1

【構　文】

論理ＡＮＤ式　＆＆　ＯＲ式

【注　意】

二項＆＆演算子の各演算数は，スカラ型です。

二項＆＆演算子は，演算数を左から右に評価します。左演算数の値が‘０’であれば，右演算数の評価を行いません。

(2) 論理ＯＲ演算子

論理演算子	||

二項||演算子

【機　能】

二項||演算子は，２つの演算数の論理ＯＲを行います。論理ＯＲ演算は２つの演算数が‘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。

表５－８　論理ＯＲ演算子

		左演算数の値	
		0	0以外
右演算数の値	0	0	1
	0以外	1	1

【構　文】

論理ＯＲ式　||　論理ＡＮＤ式

【注　意】

二項||演算子の各演算数は，スカラ型です。

二項||演算子は，演算数を左から右に評価します。左演算数の値が‘０’以外であれば，右演算数の評価を行いません。

5.10 条件演算子

条件演算子は第１演算数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。

条件演算は第１演算数を評価して，値が‘０’以外であれば第２演算数を評価し，‘０’であれば第３演算数を評価します。条件演算子の結果の値は，第２または第３演算数の値になります。

条件演算子の構文を次に示します。

条件式::=

　　論理ＯＲ式

　| 論理ＯＲ式　？　式　：　条件式

保守／廃止

条件演算子	？　：

条件演算子 '？'，':'

【機　能】

条件演算子は第１演算数を評価して，値が０以外であれば第２演算数を実行し，０のときは第３演算数を評価します。条件演算子の値は，第２または第３演算数の値になります。

【構　文】

論理ＯＲ式　？　式　：　条件式

【注　意】

条件演算子の，演算数は次に示す型です。

・第１演算数は，スカラ型です。

第２第３演算数は次に示す型のどれか１つです。

・両方とも算術型
・両方とも適合型の構造体または共用体型
・両方ともvoid型
・両方とも適合型の修飾版または非修飾版を持つオブジェクトへのポインタ
・一方がポインタで，もう一方がヌル・ポインタ定数
・一方がオブジェクトまたは不完全型へのポインタで，もう一方がvoidまたはvoidの修飾版へのポインタ

5.11 代入演算子

代入演算子は右演算数そのものを左のオブジェクトに格納する単純代入と，両演算数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。

代入式の構文を次に示します。

代入式::=

　　条件式

　| 単項式　代入演算子　代入式

代入演算子::=　以下のうちのいずれか１つ

　　＝　＊＝　／＝　％＝　＋＝　－＝　＜＜＝　＞＞＝

　　＆＝　＾＝　｜＝

(1) 単純代入

単純代入	=

単純代入演算子‘＝’

【機　能】

単純代入は，右演算数を左演算数の型に変換し，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，あふれ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 ;
/*...*/
/*...*/((c = f()) == -1)/*...*/
```

【構　文】

単項式　＝　代入式

【注　意】

単純代入の演算数は次に示す型です。

- 両方とも算術型
- 左演算数は修飾された算術型で，右演算数は算術型
- 両方とも適合型の構造体または共用体型
- 右演算数は構造体または共用体型で，左演算数はその修飾版
- 両方とも適合型へのポインタ
- 一方がオブジェクトまたは不完全型へのポインタで，もう一方がvoidポインタ
- 一方がポインタで，もう一方がヌル・ポインタ定数
- 両方ともポインタで，左演算数は右演算数の修飾版へのポインタ

(2) 複合代入

代入演算子	*= ／= %= += -=
	<<= >>= &= ^= |=

複合代入

【機　能】

複合代入演算子は，左右の演算数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，左演算数の型に変換されます。

'E1 op= E2' の複合代入は，左辺値 'E1' が1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 ;
```

【構　文】

単項式　*=　代入式

単項式　／=　代入式

単項式　%=　代入式

単項式　+=　代入式

単項式　-=　代入式

単項式　<<=　代入式

単項式　>>=　代入式

単項式　&=　代入式

単項式　^=　代入式

単項式　|=　代入式

代入演算子	*= /= %= += -= <<= >>= &= ^= |=

【注　意】

‘＋＝’と‘－＝’演算子の左演算数はオブジェクトへのポインタで，右演算数は整数型を持つか，または左演算数は修飾算術型もしくは非修飾算術型です。その他の演算子の演算数は算術型です。

5.12 コンマ演算子

コンマ演算子	,

コンマ演算子

【機　能】

コンマ演算子は，左の演算数をvoid型として評価します。それから右の演算数を評価し，その値を返します。

構文によって示されるようにコンマを区切り子として使用するところ（関数の引数リストおよび初期化子リスト中）では本章で示すコンマ演算子は現れません。

次の例では，コ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。コンマ演算子により，第２引数の値は５になります。

```
f(a, (t=3, t+2), c)
```

【構　文】

式 , 代入式

5.13 定数式

定数式には，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に，次に示す演算子は使用できません。

- 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
- コンマ演算子

(1) 整数定数式

整数定数式は整数型です。整数定数式の演算数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

整数定数式中のキャスト演算子で行う型変換は，算術型から整数型への変換だけです。

(2) 算術定数式

算術定数式は算術型です。算術定数式の演算数には，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

算術定数式中のキャスト演算子で行う型変換は，sizeof式の部分を除き算術型から算術型への変換だけです。

(3) アドレス定数

アドレス定数は，静的持続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，これを利用してオブジェクトの値を参照することはできません。

・配列添え字 []
・‘．’演算子
・‘－＞’演算子
・アドレス＆単項演算子
・間接＊演算子
・ポインタ・キャスト

(4) 初期化子中の定数式

初期化子中の定数式は，次に示すうちの１つです。

・算術定数式
・ヌルポインタ定数
・アドレス定数式
・整数定数を加減算したオブジェクトに対するアドレス定数

第６章　Ｃ言語の制御構造

本章は，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，ジャンプを使います。

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文の分岐先を指定します。
- 複文（ブロック）……１つの文法単位として処理される複数の文の集まりです。
- 式文……１つの式とセミコロンからなる文です。
- 選択文……選択処理構造の制御式と実行する文を示します。
- 繰り返し文……ループ処理構造の制御式と実行する文を示します。
- ジャンプ文……制御のジャンプとそのジャンプ先を示します。

```
char    mark[SIZE+1];
main()
{
    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++) {
        if (mark[i]) {                      ── if ……選択文
            prime = i + i + 3;
            lprintf("%d ",prime);
            count++;
                if((count%8) == 0) putchar('¥n');    ── if ……選択文
            for ( k = i + prime ; k <= SIZE ; k += prime )
                    mark[k] = FALSE;
        }
    }
    lprintf("Total %d¥n",count);
loop1:                                      ── xxx: ……ラベル付き文
    goto loop1;                             ── goto ……ジャンプ文
}
```

(1) 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。

(2) 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。

(3) 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。

(4) ジャンプ処理

ジャンプ処理は，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。ジャンプにより指定したラベル名の次の文から実行されます。

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。

ラベル付き文の構文を次に示します。

ラベル付き文::=

識別子 : 文

| case 定数式 : 文

| default : 文

(1) case

ラベル付き文	caseラベル

caseラベル

【機　能】

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

【構　文】

```
case　定数式　：　文
```

【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 ;
    }
```

【説　明】

使用例では，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。同じように値が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。

(2) default

ラベル付き文	defaultラベル

defaultラベル

【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。

【構　文】

```
default :   文
```

【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 ;
     }
```

【説　明】

使用例では，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。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。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 複文(ブロック)

複文は，複数の文を１つの文法単位とします。複数の文は，中かっこ‘{}’で囲まれることにより複文となります。たとえば複文は，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。

複文の構文を次に示します。

```
複文::=
        {  「宣言リスト」  「文リスト」  }
    宣言リスト::=
          宣言
        |  宣言リスト  宣言
    文リスト::=
          文
        |  文リスト  文
```

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。

式文と空文の構文を示します。

「式文」：

「式」　；

次の例のように，式文として副作用を得るためだけ評価される関数は，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)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 ;
/*...*/
while(*s++ != '0')
        ;
```

また複文を閉じる‘}’の前にラベルを置くために使用することもできます。

```
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。

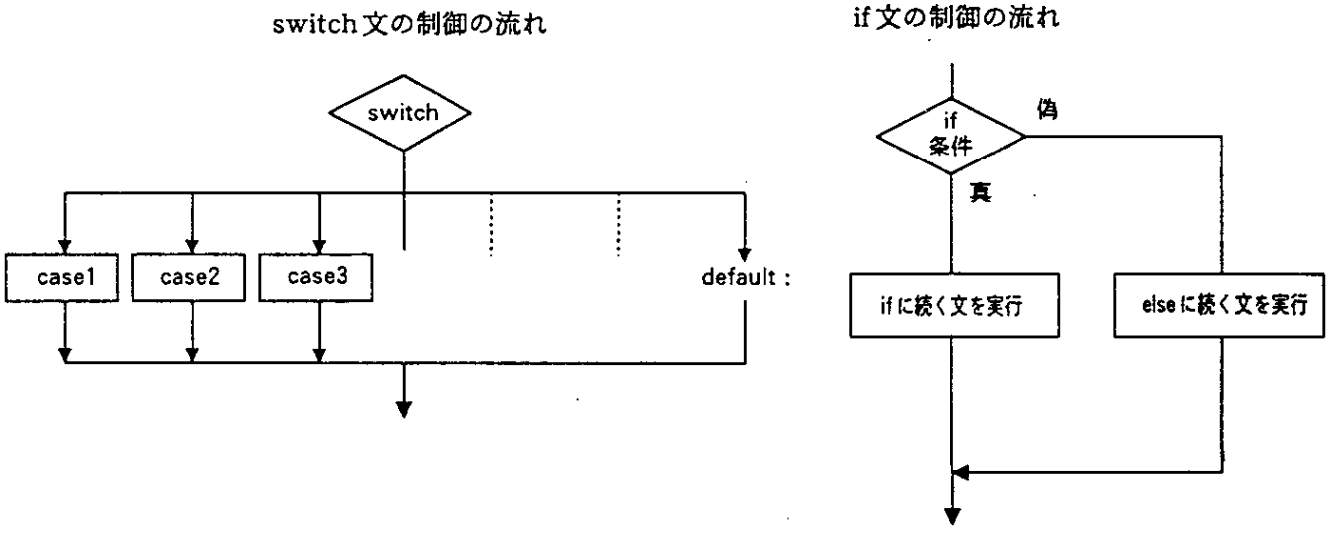
if文，switch文の構文を次に示します。

選択文::=

```
        if （ 式 ） 文
    |   if （ 式 ） 文 else 文
    |   switch （ 式 ） 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



(1) if文，if～else文

if文，if～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。

【構　文】

```
if　（　式　）　文
if　（　式　）　文　else　文
```

【使用例】

```
if( i<10 ){
        /*111*/
}
else{
        /*222*/
}
```

【説　明】

この例では，if文中の制御式によりi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文の制御式は，スカラ型です。

(2) switch文

switch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。

【構　文】

```
switch　（　式　）　文
```

【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 ;
    }
```

【説　明】

使用例では，f()の返り値が１のとき最初のcaseが選択され‘i=i+4’の式が実行されます。同じように値が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。

【注　意】

制御式は整数型で示し，各caseの式は整数定数式です。

１つのswitch文の各caseは，同じ値を設定できません。また，defaultは１つのswitch文中で１度しか使用できません。

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。

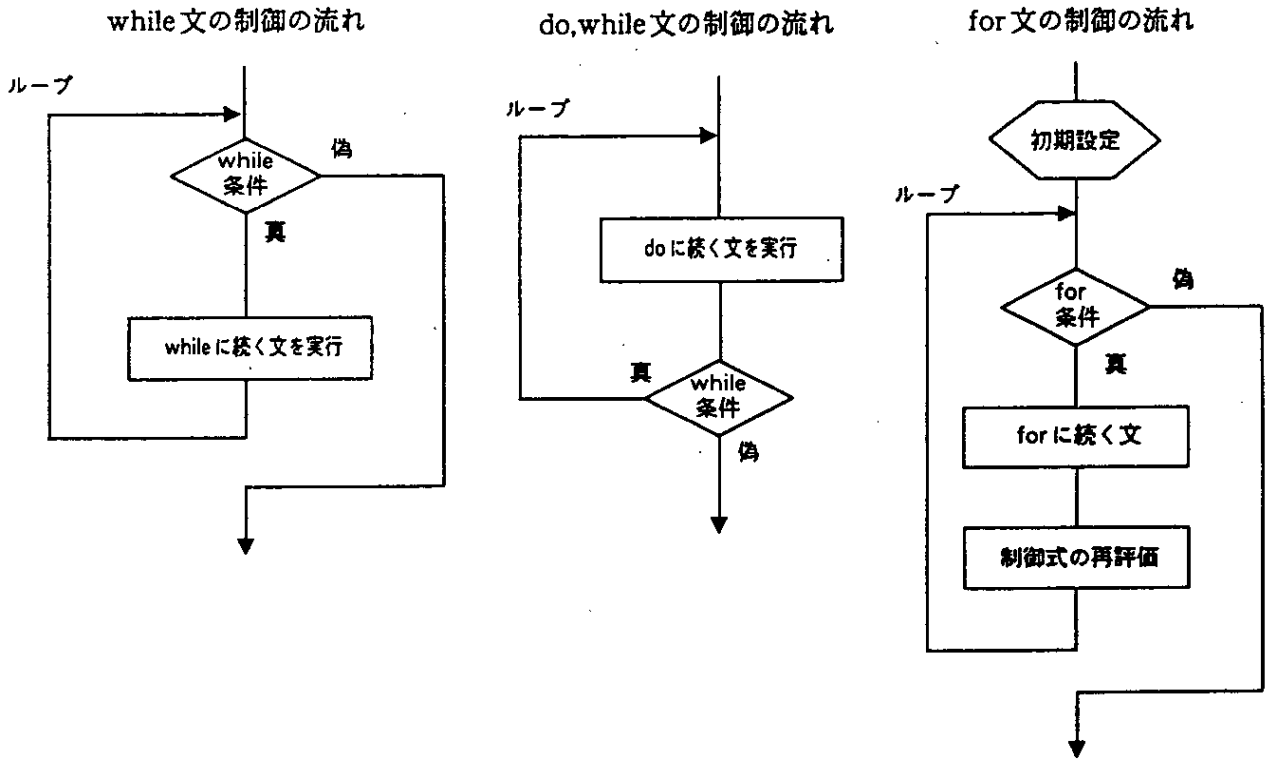
while文
do文
for文

繰り返し文の構文を次に示します。

「繰り返し文」::＝
　　while （ 式 ） 文
　| do 文 while （ 式 ） ;
　| for （ 式 ; 式 ; 式 ） 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



(1) while文

繰り返し文	while文

while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。

【構　文】

```
while　（　式　）　文
```

【使用例】

```
int i,x ;
i=1, x=0 ;
while( i<11 ){
      x += i ;
      i++ ;
}
```

【説　明】

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。

【注　意】

制御式は，スカラ型です。

(2) do文

do文

【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。

【構　文】

do　文　while　（　式　）；

【使用例】

```
int i,x ;
i=1, x=0 ;
do{
      x += i ;
      i++ ;
}
while( i<11 )
```

【説　明】

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。

【注　意】

制御式は，スカラ型です。

(3) for文

繰り返し文	for文

for文

【機　能】

for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。for文は，第１の式でカウンタとして使用する変数の初期化を行い，第２の式でカウンタの判断を行います。第３の式で変数の再評価を行います。

【構　文】

for　（「式」　；　「式」　；　「式」）　文

【使用例】

```
int i,x=0 ;
for( i=1 ; i<11 ; ++i )
     x += i ;
```

【説　明】

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】

制御式は，スカラ型です。

6.6 ジャンプ文

ジャンプ文は，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。ジャンプ文には，次の４つがあります。

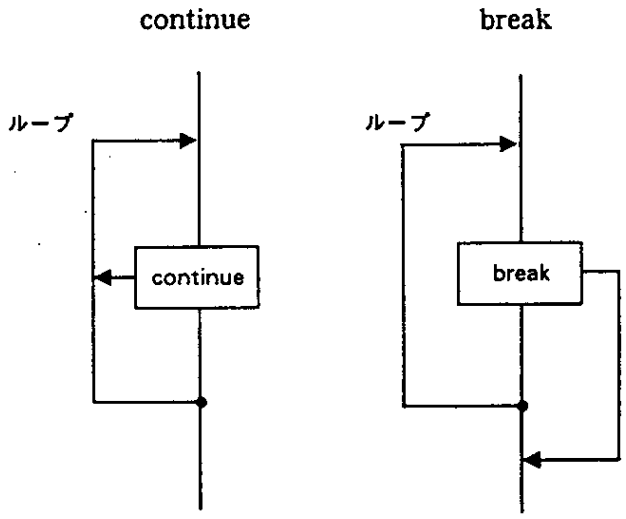
- goto文
- continue文
- break文
- return文

ジャンプ文の構文を次に示します。

```
ジャンプ文::=
        goto  識別子  ;
    |   continue  ;
    |   break  ;
    |   return  「式」  ;
```

ジャンプ文の制御の流れを次に示します。

図６－３　ジャンプ文の制御の流れ



(1) goto文

goto文

【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。

【構　文】

```
goto　識別子　;
```

【使用例】

```
do{
   /*...*/
goto point ;
   /*...*/
}while(/*...*/) ;
   /*...*/
point: ;
```

【説　明】

この例で，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文で示されるジャンプ先（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。

(2) continue文

continue文

【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ジャンプします。continue文は，これを囲む最も内側の繰り返し文に作用します。

【構　文】

```
continue  ;
```

【使用例】

```
while(/*...*/){
      /*...*/
      continue ;
      /*....*/
contin: ;
}
```

【説　明】

この例で，ループ本体中の処理がcontinue文に来ると制御は，ラベル‘contin’に無条件にジャンプします。ラベル‘contin’は，ジャンプ先を示したもので特につける必要はありません。この例は，goto文を使いcontinue文を‘goto contin ;’に代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できます。

(3) break文

break文

【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。

【構　文】

```
break   ;
```

【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 ;
        }
```

【説　明】

この例で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，またはループ本体としてのみ使用できます。

(4) return文

return文

【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。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。

【構　文】

```
return　式　;
```

【使用例】

```
void main()
{
     /*...*/
     int i ;
     y = f(i) ;
     /*...*/
}

int f(int i)
{

     int x ;
     /*...*/
     return(x) ;

}
```

【説　明】

この例で関数‘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，リターン値として変数‘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，リターン値を示す式をreturn文に使用できません。

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，共用体は重なりあ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ることができ，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とられていきます。

メンバ・オブジェクトの型は，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を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

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

・ビット・フィールド

```
unsigned int a:2;
```

```
unsigned int b:3;
unsigned int c:1;
```

(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‘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;
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二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];

no[0].name = 'NAME' ;
```

・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],*data_ptr ;

data_ptr->name = 'NAME' ;
```

7.2 共用体

共用体は，同じ領域に割り付けられるメンバ・オブジェクトの集まりです。

(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ることができ，以降このタグ名によって同一構造の共用体変数を宣言することができます。

・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とられていきます。

メンバ・オブジェクトの型は，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，共用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を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値である場合に，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・フィールドは，必要最小限のビット数の指定で，複数のメンバを１個の整数に格納することができます。

・共用体宣言リスト

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

・ビット・フィールド

```
unsigned int a:2;
unsigned int b:3;
unsigned int c:1;
```

(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ることができます。

・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;
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二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];

no[0].name = 'NAME' ;
```

・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];
}no[5],*data_ptr ;

data_ptr->name = 'NAME' ;
```

第８章　　外部定義

前処理後のプログラム・テキストは，コンパイル単位になります。コンパイル単位のプログラム・テキストは，外部宣言の列です。これらは，関数の外で現れるので，‘外部’といわれます。

‘外部定義’は，関数または外部オブジェクトを定義します。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。

外部定義の構文を次に示します。

コンパイル単位::＝

　　外部宣言

　｜　コンパイル単位　外部宣言

外部宣言::＝

　　関数定義

　｜　宣言

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

char    mark[SIZE+1];          ―― 外部オブジェクト宣言

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;                                  │
            lprintf("%d ",prime);                               ├― 関数定義
            count++;                                            │
                if((count%8) == 0) putchar('¥n');               │
            for ( k = i + prime ; k <= SIZE ; k += prime )      │
                    mark[k] = FALSE;                            │
        }                                                       │
    }                                                           │
    lprintf("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ができません。

関数定義の構文を次に示します。

関数定義::=

「宣言指定子」　宣言子　「宣言リスト」　複文

次の例でexternは記憶クラス指定子，intは型指定子です。これらは，ディフォールトの値なので省略することができます。‘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。

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

この関数定義は，関数宣言中でパラメータ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。パラメータに対して識別子リストの形を用いることにより記述することもできます。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);
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 では次のように定義します。

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

あるいは

```
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・スコープまたは初期化子を持つものです。また，ファイル・スコープを持つオブジェクトに対する識別子の宣言で，記憶クラス指定子がなく初期化子を持たないか，または記憶クラス指定子が，staticである場合は仮りの定義です。これは初期化子０のファイル・スコープを持つ宣言になります。

外部オブジェクト定義の例を次に示します。

・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，‘#’前処理トークンから改行文字までの前処理トークンの列です。

前処理トークン列の間で使用できる空白文字は，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。

前処理指令の構文を次に示します。

```
前処理ファイル::=
        「グループ」
    グループ::=
            グループ部分
        |   グループ　グループ部分
      グループ部分::=
              「前処理トークン列」　改行
          |   if節
          |   制御行
        if節::=
                ifグループ　「elifグールプの集まり」　「elseグループ」
                                                        endif行
            ifグループ::=
                    #　if　　定数式　改行　「グループ」
                |   #　ifdef　　識別子　改行　「グループ」
                |   #　ifndef　　識別子　改行　「グループ」
            elifグループの集まり::=
                    elifグループ
                |   elifグループの集まり　elifグループ
              elifグループ::=
                      #　elif　　定数式　改行　「グループ」
```

elseグループ::=

　　#　else　　改行　「グループ」

endif行::=

　　#　endif　　改行

制御行::=

　　#　include　前処理トークン列　改行

　|　#　define　識別子　置き換えリスト　改行

　|　#　define　識別子　左かっこ　「識別子リスト」　)

　　置き換えリスト　改行

　|　#　undef　　識別子　改行

　|　#　line　　前処理トークン列　改行

　|　#　error　　「前処理トークン列」　改行

　|　#　pragma　「前処理トークン列」　改行

　|　#　　改行

左かっこ::=

　　直前に空白がない（

置き換えリスト::=

　　「前処理トークン列」

前処理トークン列::=

　　前処理トークン

　|　前処理トークン列　前処理トークン

改行::=

　　改行文字

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，列挙定数を使用できません。

条件付きコンパイルの指令には‘#if’，‘#elif’，‘#ifdef’，‘#ifndef’，‘#else’，‘#endif’があります。

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。

(1) #if指令

条件付きコンパイル	#if

【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。

【構　文】

```
#if　定数式　改行　「グループ」
```

【使用例】

```
#if FAG == 0
      ⋮
#endif
```

【説　明】

使用例では，'FAG == 0'によって後に続く文をコンパイルするかどうか判断しています。'FAG'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。

(2) #elif指令

条件付きコンパイル	#elif

【機　能】

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

【構　成】

```
#elif　定数式　改行　「グループ」
```

【使用例】

```
#if FAG == 0
      ┆
#elif FAG != 0
      ┆
#endif
```

【説　明】

使用例では，‘FAG’の値によって後に続く文をコンパイルするかどうか判断しています。‘FAG’の値が0の場合，#if指令と#endif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。

(3) ＃ifdef指令

条件付きコンパイル	＃ifdef

【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。

【構　文】

```
#ifdef　識別子　改行　「グループ」
```

【使用例】

```
#define ON
#ifdef ON
    ┆
#endif
```

【説　明】

使用例では，＃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。

(4) #ifndef指令

条件付きコンパイル　　#ifndef

【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に続くプログラムをコンパイルしません。

【構　文】

```
#ifndef　識別子　改行　「グループ」
```

【使用例】

```
#define ON
#ifndef ON
     ┆
#endif
```

【説　明】

使用例では，#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。

(5) ＃else指令

条件付きコンパイル	＃slse

【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。

【構　文】

```
#else      改行   「グループ」
```

【使用例】

```
#define ON
#ifdef ON
     ┇
#else
     ┇
#endif
```

【説　明】

使用例では，＃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。

(6) #endif指令

条件付きコンパイル	#endif

【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。

【構　文】

```
#endif　改行
```

【使用例】

```
#define ON
#ifdef ON
    ┆
#endif
```

【説　明】

使用例で‘#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３つの方法があります。

・＃include　＜h文字の列＞
・＃include　”q文字の列”
・＃include　前処理トークンの列

(1) #include< >指令

ソース・ファイルの取り込み	#include< >

【機　能】

指定されたヘッダを環境変数で指定されているディレクトリ中から検索し，#include指令をヘッダの内容すべてに置き換えます。

【構　文】

#include　<　h文字の列　>　改行

【使用例】

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

【説　明】

環境変数により指定されたディレクトリ中から‘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。

(2) #include" " 指令

ソース・ファイルの取り込み	#include" "

【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，初めにカレント・ディレクトリ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環境変数で指定されたディレクトリを検索します。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。

【構　文】

#include　"　q文字の列　"　改行

【使用例】

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

【説　明】

カレント・ディレクトリまたは環境変数により指定されたディレクトリ中から‘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。

(3) #include　前処理トークン列　指令

ソース・ファイルの取り込み	#include　前処理トークン列

【機　能】

前処理トークン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。

【構　文】

#include　前処理トークン列　改行

【説　明】

‘#include　前処理トークン列　改行’によるソース・ファイルの取り込みでは，指定された前処理トークン列がマクロ置換により＜h文字の列＞または”q文字の列”に置き換わらなければなりません。＜h文字の列＞に置き換わった場合，ソース・ファイルは環境変数により指定されたディレクトリ中から検索し，”q文字の列”の場合カレント・ディレクトリから検索し，なければ環境変数により指定されたディレクトリ中から検索します。

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‘置き換えリスト’に置き換えます。特に指定されないかぎりマクロ名に置き換えられていきます。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　識別子　置き換えリスト　改行

・関数形式

#define　識別子　左かっこ　「識別子リスト」　）　置き換えリスト

(1) 引数置換

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後に行われます。置き換えリストのパラメータに＃または＃＃前処理トークンを前に付けずに，＃＃前処理トークンが後ろに続かなければ，リスト中に含まれるマクロがすべて展開された後に対応する引数に置き換えられます。

(2) ＃演算子

＃前処理トークンは，対応する引数をchar文字列処理トークンに置き換えます。置き換えリスト中のパラメータ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

＃＃前処理トークンは，前後にあるトークンを結合します。結合は，次のマクロ展開が行われる前に実行され，＃＃前処理トークンは削除されます。この結果，生成されるトークン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

```
#define debug(s,t)printf("x"#s"=%d,x"#t"=%s",x##s,x##t);
debug(1,2);
```

これは次のように展開されます。

```
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);
```

(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トークン，およびソース・ファイルの残りの前処理トークンの中にマクロ名がある場合，マクロ置換を行います。置き換えられたマクロは，ソース・ファイルの残りの前処理トークン列を含まない置き換えリストの走査中に見つけられても，置き換えられません。

(5) マクロ定義のスコープ

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。

(1) ＃define指令

マクロ置換	＃define

【機　能】

＃define指令は，指定した識別子を置き換えリストに置き換えます。この指令以降の同じ識別子は置き換えリストに置き換えられます。

【構　文】

#define　識別子　置き換えリスト　改行

【使用例】

```
#define PAI 3.1415
```

【説　明】

使用例では，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。

(2) #define () 指令

マクロ置換	#define ()

【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き換えリストに置き換えます。この指令以降の同じ識別子は置き換えリスト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ができます。

【構　文】

#define　識別子　左かっこ　「識別子リスト」　)　置き換えリスト　改行

【使用例】

```
#define F(n) (n*n)
int i;
i=F(2)
```

【説　明】

使用例の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指令の置き換えリストは()で囲んでおきます。

(3) #undef指令

マクロ置換	#undef

【機　能】

対応するマクロ置換指令を終わらせます。

【構　文】

```
#undef　識別子　改行
```

【使用例】

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

【説　明】

使用例で‘#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，次のように指定します。

#line　数字列　改行

・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

・上記の指定の他に次のように指定することができます。この場合には，指定した前処理トークン列は，すべての置き換えの後に前記の２つの例のうちどちらかになるようにします。

#line　前処理トークン列　改行

9.5 Error指令

Error指令は，指定した前処理トークンを含むメッセージを生成します。
次のように指定します。

#error 「前処理トークン列」 改行

9.6 Pragma指令

＃pragma指令は，指定した文字列をコンパイラへ指令します。本Ｃコンパイラでは，７８Ｋシリーズ用のコードを生成するために次に示す＃pragma指令が用意されています。
pragma指令の詳細は“１１章　拡張機能”をご覧ください。

9.7 Null指令

Null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 ＡＳＭ指令

ＡＳＭ指令は，次に続く文をアセンブラ・ソースとして，本Ｃコンパイラが出力するアセンブラ・ソース・モジュール・ファイル中に埋め込みます。

ＡＳＭ指令には，アセンブラ・ソースの始まりを示す‘#asm’と終了を示す‘#endasm’があります。

【構　文】

```
#asm
  ┆
#endasm
```

【使用例】

```
#asm
        callf !init
#endasm
```

【説　明】

この例で，‘#asm’と‘#endasm’の間にある‘callf !init’は，コンパイルされずコンパイラが出力するアセンブラ・ソース・モジュール・ファイル中に書き出されます。

保守／廃止

9.9 コンパイラ定義のマクロ名

本コンパイラには，次のマクロ名があらかじめ定義されています。これらのマクロ名は，１回のコンパイルの間不変です。（__ＬＩＮＥ__，__ＦＩＬＥ__を除く）前処理指令＃defineまたは＃undefの適用を受けません。

__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（‘Mmm 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__	１０進定数　1

上記の他，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。

保守／廃止

＜７８Ｋ／０の場合＞

・デバイスのシリーズ名を示すマクロ名

‘＿＿Ｋ０＿＿’

・デバイス名を示すマクロ名

表９－１　デバイス種別のマクロ名（７８Ｋ／０）

デバイス名	種　別	マクロ名
μＰＤ７８００１	００１	＿＿００１＿
μＰＤ７８００２	００２	＿＿００２＿
μＰＤ７８０１１	０１１	＿＿０１１＿
μＰＤ７８０１２	０１２	＿＿０１２＿
μＰＤ７８０１３	０１３	＿＿０１３＿
μＰＤ７８０１４，７８Ｐ０１４	０１４	＿＿０１４＿
μＰＤ７８０２２	０２２	＿＿０２２＿
μＰＤ７８０２３	０２３	＿＿０２３＿
μＰＤ７８０２４，７８Ｐ０２４	０２４	＿＿０２４＿
μＰＤ７８０４２	０４２	＿＿０４２＿
μＰＤ７８０４３	０４３	＿＿０４３＿
μＰＤ７８０４４，７８Ｐ０４４	０４４	＿＿０４４＿

保守／廃止

＜７８Ｋ／Ⅱの場合＞

・デバイスのシリーズ名を示すマクロ名

‘＿＿Ｋ２＿＿’

・デバイス名を示すマクロ名

表９－２　デバイス種別のマクロ名（７８Ｋ／Ⅱ）

デバイス名	種　別	マクロ名
μＰＤ７８２１０	２１０	＿＿２１０＿
μＰＤ７８２１２	２１２	＿＿２１２＿
μＰＤ７８２１３	２１３	＿＿２１３＿
μＰＤ７８２１４，７８Ｐ２１４	２１４	＿＿２１４＿
μＰＤ７８２１７Ａ	２１７Ａ	＿＿２１７Ａ
μＰＤ７８２１８Ａ，７８Ｐ２１８Ａ	２１８Ａ	＿＿２１８Ａ
μＰＤ７８２２０	２２０	＿＿２２０＿
μＰＤ７８２２４，７８Ｐ２２４	２２４	＿＿２２４＿
μＰＤ７８２３３	２３３	＿＿２３３＿
μＰＤ７８２３４	２３４	＿＿２３４＿
μＰＤ７８２３７	２３７	＿＿２３７＿
μＰＤ７８２３８，７８Ｐ２３８	２３８	＿＿２３８＿
μＰＤ７８２４３	２４３	＿＿２４３＿
μＰＤ７８２４４	２４４	＿＿２４４＿

保守／廃止

＜７８Ｋ／Ⅲの場合＞

・デバイスのシリーズ名を示すマクロ名

‘＿＿Ｋ３＿＿’

・デバイス名を示すマクロ名

表９－３　デバイス種別のマクロ名.（７８Ｋ／Ⅲ）

デバイス名	種　別	マクロ名
μＰＤ７８３１０	３１０	＿＿３１０＿
μＰＤ７８３１２，７８Ｐ３１２	３１２	＿＿３１２＿
μＰＤ７８３１０Ａ	３１０Ａ	＿＿３１０Ａ
μＰＤ７８３１２Ａ，７８Ｐ３１２Ａ	３１２Ａ	＿＿３１２Ａ
μＰＤ７８３２０	３２０	＿＿３２０＿
μＰＤ７８３２２，７８Ｐ３２２	３２２	＿＿３２２＿
μＰＤ７８３２３	３２３	＿＿３２３＿
μＰＤ７８３２４，７８Ｐ３２４	３２４	＿＿３２４＿
μＰＤ７８３２７	３２７	＿＿３２７＿
μＰＤ７８３２８，７８Ｐ３２８	３２８	＿＿３２８＿
μＰＤ７８３３０	３３０	＿＿３３０＿
μＰＤ７８３３４，７８Ｐ３３４	３３４	＿＿３３４＿
μＰＤ７８３５０，７８Ｐ３５２	３５０	＿＿３５０＿

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，Ｃ言語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。

本Ｃコンパイラには，次のライブラリ関数があります。

・入出力関数
・文字・文字列関数
・メモリ関数
・プログラム制御関数
・数学関数
・特殊関数

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命令になります。そして，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を参照するだけです。

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。

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ＲＰ１からＲＰ４に向かって下位から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ＲＰ１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

ＲＰ６，ＲＰ７および返り値を格納しないＲＰ３，ＲＰ４を使用するライブラリは，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。

μＰＤ７８３１０／３１２の引数と返り値の受け渡し手順の例を次に示します。

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，aの順にスタックに積まれます。

②call命令によりfuncを呼び出す（呼び出す側）

a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）

ＲＰ４を使用する場合ＲＰ４がスタックに積まれます。

④関数内で使用するsaddr領域を保存する（呼び出される側）

ＦＥ９０Ｈ，ＦＥ９２Ｈ，ＦＥ９４Ｈ番地の領域を使用する場合ＦＥ９４Ｈ，ＦＥ９２Ｈ，ＦＥ９０Ｈ番地の順に格納されたデータがスタックに退避されます。

⑤関数内で使用するワーク・エリアを確保する（呼び出される側）

ワーク・エリアのバイト数（２バイト単位）分がスタック・ポインタから引かれます。図１０－１では，４バイト分ワーク・エリアを使った場合。

⑥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ＲＰ１に，上位１６ビットがＲＰ２に格納されます。

⑦ワーク・エリアを解放する（呼び出される側）

ワーク・エリアのバイト数（２バイト単位）がスタック・ポインタに加えられます。

⑧退避したsaddr領域を回復する（呼び出される側）

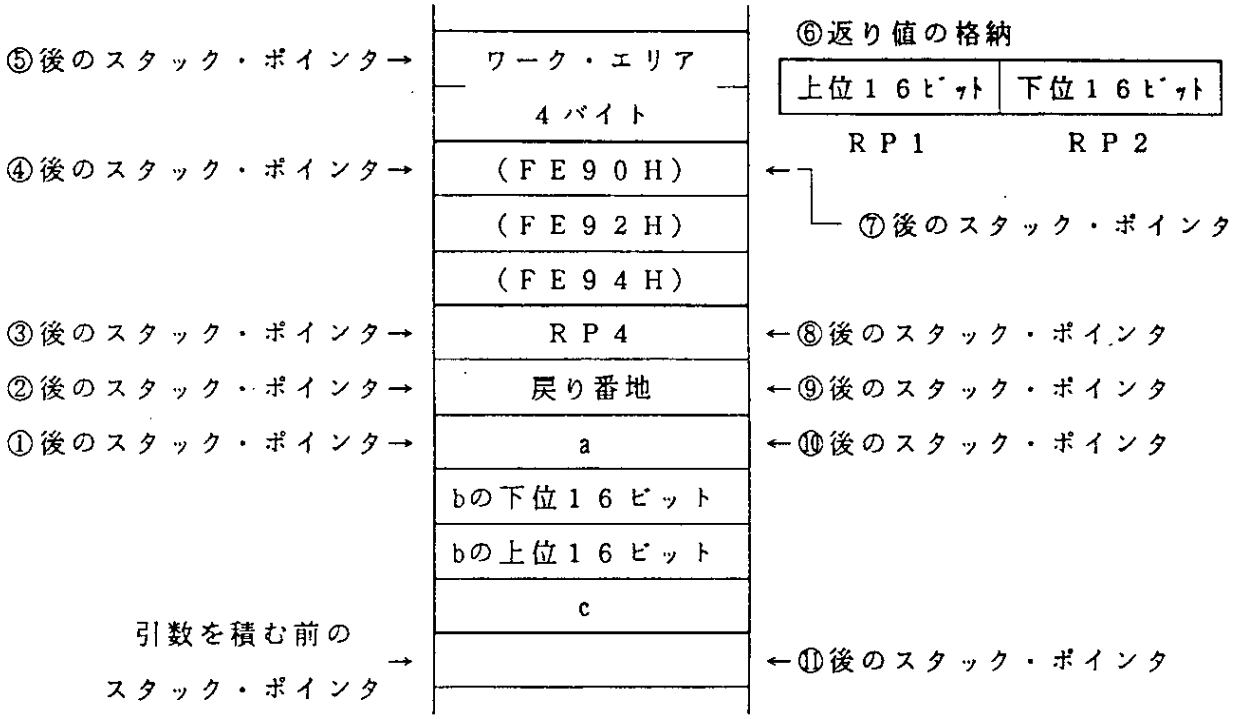
⑨退避したレジスタを回復する（呼び出される側）

⑩ret命令で呼び出した関数に制御を戻す（呼び出される側）

⑪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。図１０－１の場合８が加えられます。

図１０－１　関数呼び出し時のスタック領域



10.2 ヘッダ・ファイル

本Cコンパイラには，９個のヘッダ・ファイルがあり，それぞれライブラリ関数，型名，マクロ名を定義または，宣言します。

本Cコンパイラのヘッダ・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	limits.h
stddef.h	math.h	float.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。

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

(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‘setjmp’，‘longjmp’が定義されています。

setjmp.hでは，次のオブジェクトが宣言されています。

・大きさ１３のint型配列の型‘jmp_buf’の宣言

```
typedef int jmp_buf[15] ;
```

(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_start　　va_arg　　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，‘sprintf’，‘sscanf’が定義されています。

(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。

strtol	strtoul	atoi	atol	itoa	ltoa
ultoa	malloc	calloc	realloc	free	brk
sbrk	abort	atexit	exit	abs	labs
rand	srand	div	ldiv	qsort	bsearch

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 ;
```

・long int型のメンバ‘quot’，‘rem’を持つ構造体型‘ldiv_t’の宣言

```
typedef struct {
              long int quot ;
              long int rem ;
              } ldiv_t ;
```

・マクロ名‘RAND_MAX’の定義

```
#define RAND_MAX 32767
```

(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。

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

(7) error.h

error.hでは，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 ;
```

(8) 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
```

(9) 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 ;
```

保守／廃止

10.3 標準ライブラリ関数

本Ｃコンパイラの標準ライブラリ関数を機能別に分けて説明します。

・項番（1-1～1-2）・・・・・入出力関数

・項番（2-1～2-17）・・・・文字・文字列関数

・項番（3-1～3-9）・・・・・メモリ関数

・項番（4-1～4-3）・・・・・プログラム制御関数

・項番（5-1～5-9）・・・・・数学関数

・項番（6-1～6-4）・・・・・特殊関数

表１０－１　ライブラリ関数一覧（１／３）

項番	分類	関数名	ヘッダ・ファイル名
１－１	入出力関数	sprintf	stdio.h
１－２		sscanf	
２－１	文字・文字列関数	isalpha isupper islower isdigit isalnum isxdigit isspace ispunct isprint isgraph iscntrl isascii	ctype.h
２－２		toupper tolower	
２－３		_toupper _tolower	
２－４		toascii	

表１０－１　ライブラリ関数一覧（２／３）

項番	分類	関数名	ヘッダ・ファイル名
２－５	文字・文字列関数	strlen	string.h
２－６		strcpy strncpy	
２－７		strcat strncat	
２－８		strcmp strncmp	
２－９		strchr strrchr	
２－１０		strpbrk	
２－１１		strspn strcspn	
２－１２		strstr	
２－１３		strtok	
２－１４		strtol strtoul	stdlib.h
２－１５		atoi atol	
２－１６		itoa ltoa ultoa	
３－１	メモリ関数	malloc	stdlib.h
３－２		calloc	
３－３		realloc	
３－４		free	
３－５		brk sbrk	

表１０－１　ライブラリ関数一覧（３／３）

項番	分類	関数名	ヘッダ・ファイル名
３－６	メモリ関数	memcpy memmove	string.h
３－７		memcmp	
３－８		memchr	
３－９		memset	
４－１	プログラム制御関数	setjmp longjmp	setjmp.h
４－２		abort	stdlib.h
４－３		atexit exit	
５－１	数学関数	abs labs	stdlib.h
５－２		rand srand	stdlib.h
５－３		div ldiv	stdlib.h
６－１	特殊関数	qsort	stdlib.h
６－２		bsearch	
６－３		strerror	string.h
６－４		va_start va_arg va_end	stdarg.h

【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】

・formatで指定された出力変換仕様に従い，formatの後に続く（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）。

　・０個以上のフラグ（後述）は変換仕様の意味を修飾します。

　・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，左にパットを入れます（左寄せのフラグ（－）が指定されていれば右にパットが入ります）。パットは，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。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と見なします。この精度指定から生ずるパットの量は，フィールド幅指定のパットに優先します。

・オプションのh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－フラグの後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。

・－･･･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。

その他の変換では，#フラグを無視します。

◎変換指定は次のとおりです。

・d,i,o,u,x,X･･･int引数を符号付き１０進（dまたはi），無符号８進（o），無符号１０進（u），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。

・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かれません。

・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。精度指定は無視します。

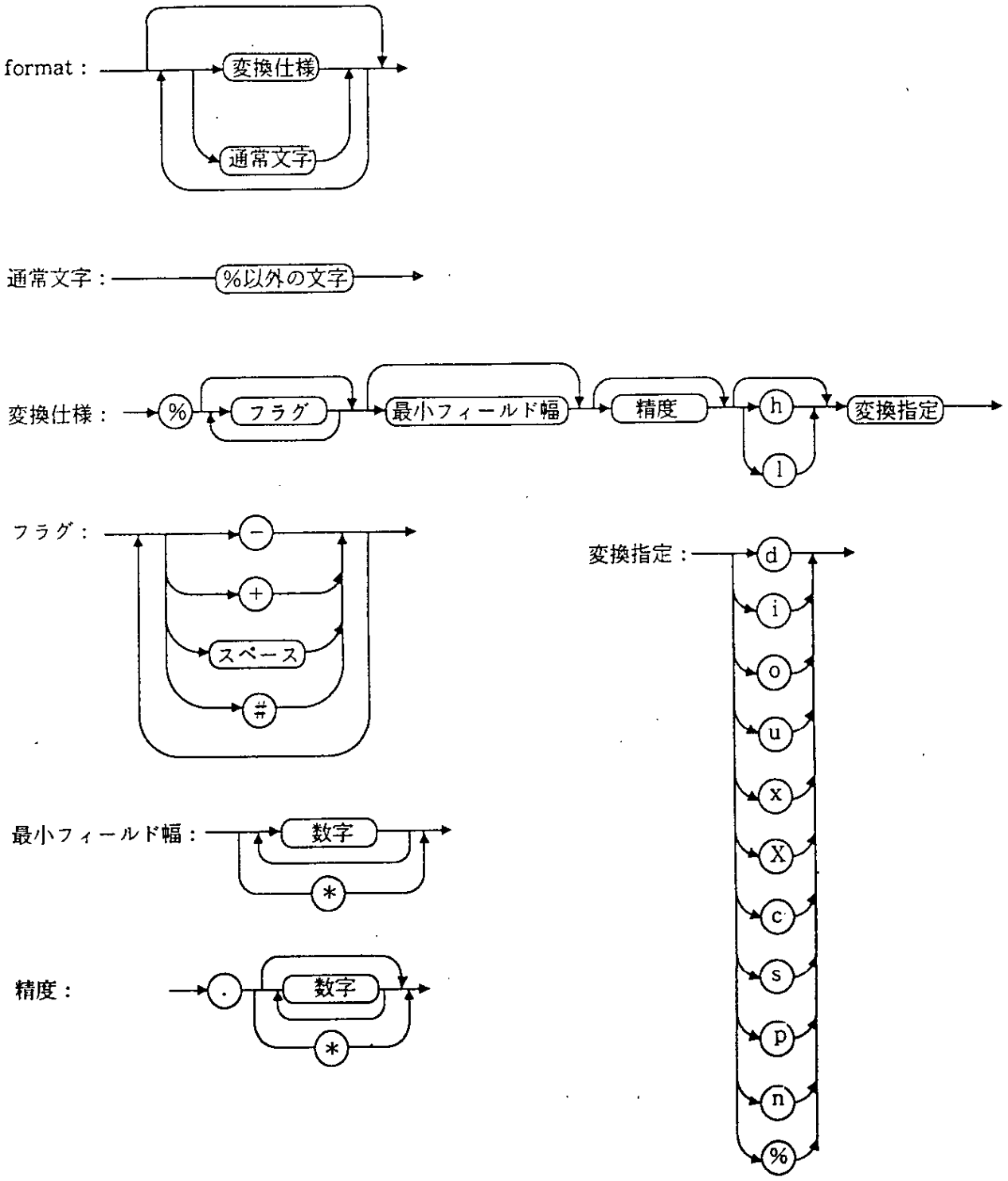
・n･･･引数は整数へのポインタで，そこに対してこれまでに文字列sに書き出した文字数を入れます。変換は行いません。

・%･･･%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）

・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の構文図を図１０－２に示します。

図１０－２　出力formatの構文図



【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しません。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。

・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）

・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）

・オプションのh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。

・対応する変換の種類を示す文字。変換指示子（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。

・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。

・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見なします。入力項目の長さが０のとき，指令の実行は失敗します。

・％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型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限り，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ストアされます。

変換指示子を次に示します。

・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進整数と見なします。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。

・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。

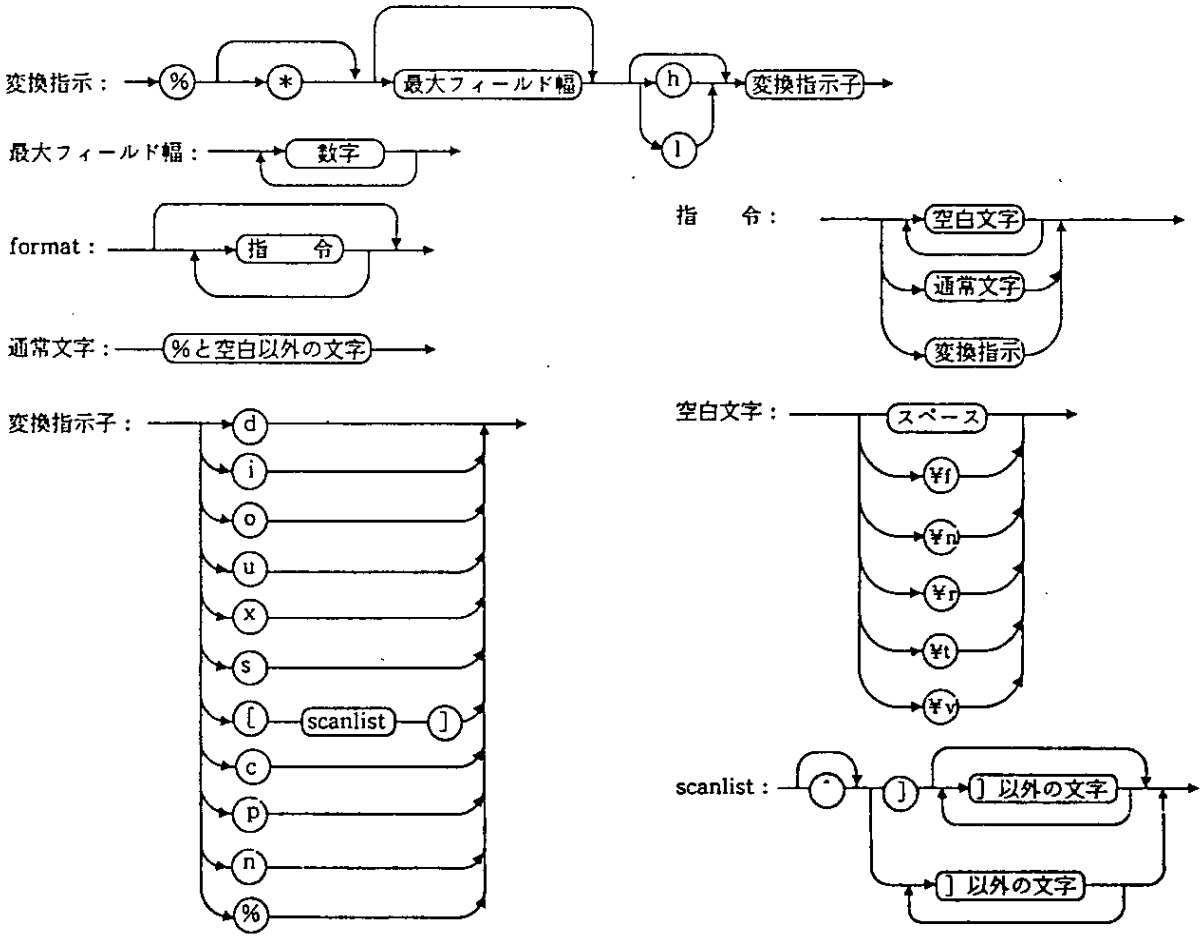
変換指示が不正な場合，指令は失敗します。

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。

図１０－３　入力format構文図



【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文字であるかを判定します。

【機　能】

・文字種の変換を行います。

・touppert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 to～(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
・touppert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。

【機　能】

・_toupperは，cから‘a’を引き‘A’を加えます。　　a：英小文字
・_tolowerは，cから‘A’を引き‘a’を加えます。　　A：英大文字

【ヘッダ・ファイル】

・ctype.h

【関数プロトタイプ】

・int _to～(int c);

関数名	引　数	返り値
_toupper	c･･･変換される文字	cから‘a’を引き‘A’を加えた値
_tolower		cから‘A’を引き‘a’を加えた値

a：英小文字
A：英大文字

【説　明】

_toupper
・_toupperは，touppper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。

【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

【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。

【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。

【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 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，正常動作は保証しません。（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，正常動作は保証しません。（先頭から順にコピーするため）

・s2が指す文字列がn文字未満の場合には，終端のヌル文字までをコピーします。n文字以上の場合には，先頭からn文字分をコピーし終端のヌル文字はコピーしません。

2-7 strcat
strncat 文字・文字列関数

【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 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。

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。

【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。

【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 ･･･ヌル・ポインタ

【説　明】

strchr

・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，ヌル・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，ヌル・ポインタを返します。

【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･･･ヌル・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，ヌル・ポインタを返します。

2-11 strspn
strcspn 文字・文字列関数

【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	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	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。

【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･･･ヌル・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，ヌル・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。

【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，ヌル・ポインタ s2･･･トークンの区切り文字を示す文字列へのポインタ	トークンがある場合 ･･･トークンの第1文字へのポインタ トークンがない場合 ･･･ヌル・ポインタ

【説 明】

・トークンとは，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がヌル・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，保存ポインタがヌル・ポインタの場合は何もせずにヌル・ポインタを返します。

・s1がヌル・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ヌル・ポインタにして，ヌル・ポインタを返します。見つかれば，その文字をトークンの第1文字とします。

・トークンの第1文字が見つかった場合，文字列s2に含まれる文字をトークンの第1文字以降から探します。見つからなければ，保存ポインタをヌル・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。

・トークンの第1文字へのポインタを返します。

【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 正のオーバフローの場合 ･･･LONG_MAX (2147483647) 負のオーバフローの場合 ･･･LONG_MIN (-2147483648) 変換が行われない場合･･･0
strtoul		正常の場合･･･変換した値 オーバフローの場合 ･･･ULONG_MAX (4294967295U) 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

(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（０x～または０Ｘ～（１６進数），０～（８進数），０以外の数字～（１０進数））と解釈（符号が前にあってもよい）します。

・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がヌル・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。

・オーバフローの場合，正はＬＯＮＧ＿ＭＡＸ（2147483647），負はＬＯＮＧ＿ＭＩＮ（-2147483648）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がヌル・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

(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がヌル・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。

・オーバフローの場合，ＵＬＯＮＧ＿ＭＡＸ（4294967295U）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がヌル・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。

【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 正のオーバフローの場合 ･･･INT_MAX (32767) 負のオーバフローの場合 ･･･INT_MIN (-32768) 不正文字列の場合･･･0
atol		正常の場合･･･変換された値 正のオーバフローの場合 ･･･LONG_MAX (2147483647) 負のオーバフローの場合 ･･･LONG_MIN (-2147483648) 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。オーバフローが起こった場合は，正のときはＩＮＴ＿ＭＡＸ（32767）負のときはＩＮＴ＿ＭＩＮ（-32768）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。オーバフローが起こった場合は，正のときはＬＯＮＧ＿ＭＡＸ（2147483647）負のときはＬＯＮＧ＿ＭＩＮ（-2147483648）を返します。

【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。

・ultioa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･･･ヌル・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，ヌル・ポインタを返します。

【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 ･･･ヌル・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，ヌル・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。ブレーク値は，brkで設定します。brkについては“3-5　brk”をご覧ください。

【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 ･･･ヌル・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，ヌル・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。ブレーク値は，brkで設定します。brkについては“3-5　brk”をご覧ください。

【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がヌル・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･･･ヌル・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大きさ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開放します。

・ptrがヌル・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，何もせずにヌル・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。

【機　能】

・割り付けられているブロックを開放します。

【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･･･開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）を開放します（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。（開放は，ptrを新たなブレーク値とすることで行います。）

3-5 brk
sbrk　　メモリ関数

【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のＥＮＯＭＥＭ（３）をセットします。

sbrk

・sbrkは，ブレーク値をincrバイト増減（incrの符号による）する。

・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。

【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，正常動作は保証しません（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。

【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。

【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 ･･･ヌル・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，ヌル・ポインタを返します。

【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。

4-1 setjmp
longjmp　　プログラム制御関数

【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回復します。

【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は戻りません。

【説　明】

setjmp
・setjmpは，直接呼び出された場合，ＲＰ３，ＲＰ４，ＲＰ７，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（ＲＰ３，ＲＰ４，ＲＰ７およびレジスタ変数として使用するsaddr,sp）を回復し，対応するsetjmpがval（ただしvalが０の場合は１）を返したかのごとくプログラムの実行が続きます。

【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。

【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個まで登録できます。登録できた場合は，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。

・ループし戻りません。

・ユーザはexitの処理を作成します。

5-1 abs
labs　　数学関数

【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	-32767≦j≦32767の場合 ･･･jの絶対値 jが-32768の場合･･･-32768 (0x8000)
labs		-2147483647≦j≦2147483647の場合･･･jの絶対値 jが-2147483648の場合 ･･･-2147483648(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，-32768を返します。

labs
・labsは，jの値(long型)の絶対値を求めます。

・jが-2147483648の場合は，-2147483648を返します。

【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関数をコールした後にrand関数をコールするのと同じです。

5-3 div
ldiv
数学関数

【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，符号は数学と同じ（numerとdenomが同符号の場合は正，異符号の場合は負）です。

・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－32768，denomが－１の場合，商は－32768，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，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－2147483648，denomが－１の場合は，商は－2147483648，剰余は０です。

【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。

【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･･･配列の２つの要素を比較し，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 ･･･ヌル・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

【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･･･ヌル・ポインタ

【説 明】

・errnumの値に対応して，次の文字列へのポインタを返します。

0 ･･･“Error 0”

1（EDOM） ･･･“Argument too large”

2（ERANGE）･･･“Result too large”

3（ENOMEN）･･･“Not enough memory”

その他はヌル・ポインタを返します。

6-4 va_start
va_arg
va_end 特殊関数

【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_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);

・type va_arg(va_list ap,type);

・void va_end(va_list ap);

関数名	引　数	返り値
va_start	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引数 type･･･可変引数の該当箇所をポイントするための型	正常の場合･･･可変引数の該当箇所の値 apがヌル・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

6-4 va_start
va_arg
va_end 特殊関数

【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，正常動作は保証しません。

va_arg
・va_argで引数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（それ以上の正常動作は保証しません）。
・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がヌル・ポインタの場合は，type型の０を返します。

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にヌル・ポインタをセットします。

保守／廃止

第１１章　拡張機能

本章では，ＡＮＳＩ（American National Standards Institute）に規定されていない，本Ｃコンパイラ特有の拡張機能について説明します。

本Ｃコンパイラの拡張機能は，ターゲット・デバイスである７８Ｋ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。

本Ｃコンパイラの拡張機能を使ったＣソース・プログラムは，マイクロプロセッサに依存した機能を利用しますが，他のマイクロプロセッサ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容易な修正により他のマイクロプロセッサへ移植できます。

11.1 マクロ名

本Cコンパイラは，ターゲット・デバイスによって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します。

マクロ名の詳細については，“9.9　コンパイラ定義のマクロ名”を参照してください。

11.2 キーワード

拡張機能により本Cコンパイラでは，次のトークンをキーワードとして使用します。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。

表１１－１　追加キーワード一覧

予　約　語	用　　途
callt	callt関数
callf	callf関数
sreg	sreg変数
noauto	noauto関数
norec	norec関数
bit	bit型変数

(1) 関　数

callt, callf, noauto, norec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

修飾属性子の指定は，次のものに限ります。（noautoと，norecは，同時に指定できません）。また，他の関数内で宣言することはできません。

- callt
- callf
- noauto
- norec
- callt　noauto
- callt　norec
- noauto　callt
- norec　callt
- callf　noauto
- callf　norec
- noauto　callf
- norec　callf

(2) 変　数

- sregの指定は，Ｃ言語のregisterと同じ規定です。
  （sregの詳細は，“11.4　(3) saddr領域利用法”をご覧ください）
- bit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。

保守／廃止

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。

＜７８Ｋ／０の場合＞

(1) メモリ・モデル

メモリ空間は最大６４Ｋバイトです。このため，ソース・ファイルもコード部，データ部をあわせて６４Ｋバイト以内にします。

(2) レジスタ・バンク

スタート・アップ時にレジスタ・バンクを‘ＲＢ０’に設定します。これ以外には，レジスタ・バンクの切り換え命令は出力しません。。

(3) メモリ空間

本Ｃコンパイラは，次のようにメモリ空間を利用します。

表１１－２　メモリ空間の利用（７８Ｋ／０）

アドレス		用　途		サイズ (byte)
００	４０～７ＦＨ	ＣＡＬＬＴテーブル		６４
０８	00～0FFFH	ＣＡＬＬＦ関数		８００
ＦＥ	２０～８ＢＨ	sreg変数，bit型変数		１０８
ＦＥ	８Ｃ～８ＦＨ	ディバグ用予約領域　注1		4
ＦＥ	９０～９ＦＨ	ランタイムライブラリの引数		１６
ＦＥ	Ａ０～ＡＦＨ	レジスタ変数		１６
ＦＥ	Ｂ０～ＢＦＨ	norec関数の引数		１６
ＦＥ	Ｃ０～ＤＦＨ	norec関数のオートマチック変数		３２
ＦＥ	Ｅ０～ＥＦＨ	ＲＢ３～ＲＢ１	（コンパイラは使用しません）	２４
	Ｆ０～ＦＦＨ	ＲＢ０	ワークレジスタ	8
ＦＦ	００～ＦＦＨ	sfr名		２５６

保守／廃止

注1. ディバグ用予約領域

コンパイル時にディバグ・オプションを指定した場合，この領域をディバグ用に使用するので，ユーザ・プログラムでは他の用途には使用できません。

保守／廃止

＜７８Ｋ／Ⅱの場合＞

(1) メモリ・モデル

メモリ空間は最大１Ｍバイトですが，ソース・ファイルはコード部，データ部をあわせて６４Ｋバイト以内にします。ただし，拡張外部メモリへのアクセスもライブラリにより可能です。

(2) レジスタ・バンク

スタート・アップ時にレジスタ・バンクを‘ＲＢ０’に設定します。これ以外には，レジスタ・バンクの切り換え命令は出力しません。。

(3) メモリ空間

本Ｃコンパイラは，次のようにメモリ空間を利用します。

表１１－３　メモリ空間の利用（７８Ｋ／Ⅱ）

アドレス		用　途		サイズ (byte)
００	４０～７ＦＨ	ＣＡＬＬＴテーブル		６４
０８	00～0FFFH	ＣＡＬＬＦ関数		８００
ＦＥ	２０～７ＢＨ	sreg変数，bit型変数		９２
ＦＥ	７Ｃ～７ＦＨ	ディバグ用予約領域　注1		４
ＦＥ	８０～８ＦＨ	ランタイムライブラリの引数		１６
ＦＥ	９０～９ＦＨ	レジスタ変数		１６
ＦＥ	Ａ０～ＡＢＨ	norec関数の引数		１２
ＦＥ	ＡＣ～Ｃ１Ｈ	norec関数のオートマチック変数		２２
ＦＥ	Ｃ２～ＤＦＨ	マクロ・サービス　注2 （コンパイラは使用しません）		３０
ＦＥ	Ｅ０～ＥＦＨ	ＲＢ３～ＲＢ１	（コンパイラは使用しません）	２４
	Ｆ０～ＦＦＨ	ＲＢ０	ワークレジスタ	８
ＦＦ	００～ＦＦＨ	sfr名		２５６

保守／廃止

注1. ディバグ用予約領域

コンパイル時にディバグ・オプションを指定した場合，この領域をディバグ用に使用するので，ユーザ・プログラムでは他の用途には使用できません。

2. マクロ・サービスの領域はチップにより異なりますが，最大の領域をマクロ・サービスのための領域として確保し，コンパイラでは使用しません。

保守／廃止

＜７８Ｋ／Ⅲの場合＞

(1) メモリ・モデル

メモリ空間は最大６４Ｋバイトです。このため，ソース・ファイルもコード部，データ部をあわせて６４Ｋバイト以内にします。

(2) レジスタ・バンク

スタート・アップ時にレジスタ・バンクを‘ＲＢ７’に設定します。これ以外には，レジスタ・バンクの切り換え命令は出力しません。。

(3) メモリ空間

本Ｃコンパイラは，次のようにメモリ空間を利用します。

表１１－４　メモリ空間の利用（７８Ｋ／Ⅲ）

アドレス		用途		サイズ (byte)
００	４０～７ＦＨ	ＣＡＬＬＴテーブル		６４
０８	00～0FFFH	ＣＡＬＬＦ関数		８００
ＦＥ	２０～７ＢＨ	sreg変数，bit型変数		９２
ＦＥ	２Ｃ～７ＢＨ	sreg変数，bit型変数　注1		８０
ＦＥ	７Ｃ～７ＦＨ	ディバグ用予約領域　注2		４
ＦＥ	８０～８ＦＨ	ＲＢ７	ワーク・レジスタ	１６
	９０～９ＦＨ	ＲＢ６	レジスタ変数	１６
	Ａ０～ＡＦＨ	ＲＢ５	norec関数の引数	１６
	Ｂ０～ＣＦＨ	ＲＢ４～ＲＢ３	norec関数のオートマティック変数	３２
	Ｄ０～ＤＦＨ	ＲＢ２	（Ｃコンパイラは使用しません）	１６
	Ｅ０～ＦＦＨ	ＲＢ１～ＲＢ０	マクロ・サービス　注1 （Ｃコンパイラは使用しません）	３２
ＦＦ	００～ＦＦＨ	sfr名		２５６

保守／廃止

注1. μＰＤ７８３１０／３１２／Ｐ３１２，μＰＤ７８３１０Ａ／３１２Ａ／Ｐ３１２Ａ以外は，マクロ・サービス・コントロール・ワードがＦＥ０６Ｈ～ＦＥ２ＢＨに配置されるためsreg変数，bit型変数に使用できる領域が異なります。

2. ディバグ用予約領域

コンパイル時にディバグ・オプションを指定した場合，この領域をディバグ用に使用するので，ユーザ・プログラムでは他の用途には使用できません。

11.4 拡張機能の使用方法

個々の拡張機能について，次の順に説明します。

機　能：拡張機能により実現される機能を説明します。
方　法：拡張機能の利用法を説明します。
制　限：拡張機能を利用する場合の制限を説明します。
使用例：拡張機能の使用例を示します。
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・プログラムを本Ｃコンパイラ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。

(1) callt関数

callt関数	callt

【機　能】

・calltテーブルの領域を利用して，関数の呼び出しを行います。呼び出される関数のアドレス（callt関数と呼びます）をcalltテーブルに格納し，関数が呼び出されます。これは，通常のcall命令よりも短いコードとなり，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属性を追加します。

```
callt  extern  型名  関数名
```

【制　限】

・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。

・calltテーブルの領域：４０Ｈ～７ＦＨ

・１つのロード・モジュール内で宣言できるcallt関数の数：最大３２

・リンクするモジュールで宣言できるcallt関数の数のトータル：最大３２

【使用例】

（Ｃソース）

```
============ ca1.c ============  ============ ca2.c ============
callt extern int tsub();

void main()                       callt int tsub()
{                                 {
        int ret_val;                      int val;

        ret_val = tsub();                  return val;
}                                 }
```

（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           ;
_tsub:                          ;関数定義
            :
        関数本体
            :

@@CALT    CSEG    CALLT0        ;セグメントへの割り付け

?tsub:    DW      _tsub
```

【説　明】

・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コンパイラから本Ｃコンパイラ＞

・キーワード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。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#defineによって行います。詳しくは“11.5　Ｃソースの修正”をご覧ください。

(2) レジスタ変数

レジスタ変数	register

【機　能】

・宣言した変数をレジスタまたはsaddr領域に割り当てます。レジスタ，saddr領域に対する命令は，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，実行速度の向上が図れます。

＜７８Ｋ／０の場合＞

・レジスタ変数を宣言された順にsaddr領域（FEA0H～FEAF）に割り当てます。ただし，コンパイル時にレジスタ変数の最適化オプションを指定したときのみ割り当てられます。

＜７８Ｋ／Ⅱの場合＞

・レジスタ変数を宣言された順にsaddr領域（FE90H～FE9F）に割り当てます。ただし，コンパイル時のオプション指定でレジスタ変数の最適化オプションを指定したときのみ割り当てられます。

＜７８Ｋ／Ⅲの場合＞

・レジスタ変数を宣言された順にレジスタ（RP3,VP），saddr領域（RB6:FE90H～FE9F）に割り当てます。ただし，saddr領域にはコンパイル時のオプション指定でレジスタ変数の最適化オプションを指定したときのみ割り当てられます。

【方　法】

・記憶クラス指定子registerで，registerクラスであることを宣言します。

register　型名　変数名

【制　限】

・レジスタ変数宣言は，char，int，short，およびポインタに対して使用できます。charは，他の型に対して１／２の領域を使用します。char同士は，バイト境界を持ちそれ以外はワード境界を持ちます。

・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，内容に依存する）。

＜７８Ｋ／０，Ⅱの場合＞

・１つの関数内におけるint，short型またはポインタのレジスタ変数：最大８

・９番目の変数からは通常のメモリに割り当てます。

＜７８Ｋ／Ⅲの場合＞

・１つの関数内におけるint，short型またはポインタのレジスタ変数：最大１０

・１１番目の変数からは通常のメモリに割り当てます。

【使用例】

＜７８Ｋ／Ⅲの場合＞

（Ｃソース）

```
========== rei.c ==============
void main()
{
        register int i,j;
        i = 0;          j = 1;
        i += j;
}
```

（出力オブジェクト）

レジスタ変数がレジスタに割り当てられた例

```
_main:
      push    rp3,vp,up       ; 関数の先頭でレジスタの内容を退避します。
      movw    rp3,#0H         ; 関数中では以下のようなコードを出力します。
      movw    vp,#01H         ;
      addw    rp3,vp          ;
      pop     rp3,vp,up       ; 関数の終わりでレジスタの内容を復帰します。
```

レジスタ変数がsaddr領域に割り当てられた例

```
      EXTRN   _@KREG00        ; 使用するsaddr領域の宣言を行います。
      EXTRN   _@KREG02        ;

      movw    ax,_@KREG00     ; 関数の先頭でsaddrの内容を退避します。
      push    ax              ;
      movw    ax,_@KREG02     ;
      push    ax              ;

      movw    _@KREG00,#0H    ; 関数中では以下のようなコードを出力します。
      movw    _@KREG02,#01H   ;
      addw    _@KREG00,@KREG02;

      pop     ax              ; 関数の終わりでsaddrの内容を復帰します。
      movw    _@KREG02,ax     ;
      pop     ax              ;
      movw    _@KREG00,ax     ;
```

【説　明】

・レジスタ変数を使用するには，変数の記憶クラスをregisterクラスにするだけです。レジスタ変数は宣言された順にレジスタ（RP3,VP），saddr領域(RB:FE90H～FE9FH）に割り当てられます。

【互換性】

＜他のＣコンパイラから本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。

・レジスタ変数にしたい場合は，register宣言を追加します。

＜本Ｃコンパイラから他の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，またどの様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。

(3) saddr領域利用

saddr領域利用	sreg

【機　能】

・sreg宣言されたオブジェクトをsreg変数と呼びます。saddr領域に割り当てられたsreg変数に対する命令は，メモリに対する命令よりも短く，オブジェクト・コードが短縮し実行速度が向上します。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。

・sreg変数は，自動的にビット型変数になります。

＜７８Ｋ／０の場合＞

・sreg宣言した変数は，saddr領域（FE20H～FE8BH）にリロケータブルに割り付けます。

・アセンブラ・ソース中で宣言したsreg変数のうちで，Ｃソース・プログラムから参照できる領域は，saddr領域（FE20H～FF1FH）です。ただし，レジスタ・バンク（RB0）は，Ｃコンパイラがワーク・レジスタとして使用するので注意する必要があります。

＜７８Ｋ／Ⅱの場合＞

・sreg宣言した変数は，saddr領域（FE20H～FE7BH）にリロケータブルに割り付けます。

・アセンブラ・ソース中で宣言したsreg変数のうちで，Ｃソース・プログラムから参照できる領域は，saddr領域（FE20H～FF1FH）です。ただし，レジスタ・バンク（RB0）は，Ｃコンパイラがワーク・レジスタとして使用するので注意する必要があります。

＜７８Ｋ／Ⅲの場合＞

・sreg宣言した変数は，saddr領域にリロケータブルに割り付けます。

・アセンブラ・ソース中で宣言したsreg変数のうちで，Ｃソース・プログラムから参照できる領域は，saddr領域（FE20H～FF1FH）です。ただし，次のレジスタ・バンク（RB2～RB7）は，本Ｃコンパイラによって使用されるので注意する必要があります。

レジスタ・バンク（ＲＢ３～ＲＢ６）：拡張仕様で使用

レジスタ・バンク（ＲＢ７）：ワーク・レジスタとして使用

【方　法】

・変数を定義するモジュール中でsreg宣言を行います。関数内では記述できません。

sreg　型名　変数名

・変数を参照するモジュール中で次の宣言を行います。関数内でも記述可能です。

extern　sreg　型名　変数名

【制　限】

・変数の型には，char，int，short，ポインタ型を使用します。

・char型は，他の型の半分の領域をとります。char型同士の時はバイト境界で，それ以外の時はワード境界です。

＜７８Ｋ／０の場合＞

・int，short，ポインタの場合，１ロード・モジュールあたり最大５４変数まで使用可能です。ただし，bit型変数を使用すると使用可能数が減ります。

＜７８Ｋ／Ⅱの場合＞

・int，short，ポインタの場合，１ロード・モジュールあたり最大４６変数まで使用可能です。ただし，bit型変数を使用すると使用可能数が減ります。

＜７８Ｋ／Ⅲの場合＞

・int，short，ポインタの場合，１ロード・モジュールあたり使用可能な変数は，次のとおりです。

μPD78310/312/P312，μPD78310A/312A/P312Aの場合　　：４６変数まで

μPD78310/312/P312，μPD78310A/312A/P312A以外の場合：４０変数まで

ただし，bit型変数を使用すると使用可能数が減ります。

【使用例】

＜７８Ｋ／Ⅲの場合＞

（Ｃソース）

```
========== sa1.c ===============

extern sreg int hsmm0;
extern sreg int hsmm1;
extern sreg int *hsptr;

void main()
{
        hsmm0 -= hsmm1;
        hsmm1 += *hsptr;
        ++hsmem0;
        ++(*hsptr);
}
```

（アセンブラ・ソース）

（ユーザが作成する場合です。ただし，extern宣言を付けない場合は本Ｃコンパイラが出力します。この場合ＯＲＧ疑似命令は出力しません）

```
        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: DB      (2)             ;
_hsmm1: DB      (2)             ;
_hsptr: DB      (2)             ;
```

（出力オブジェクト）

関数中では次のようなコードを出力します。

```
        subw    _hsmm0,_hsmm1
```

【説　明】

sreg変数を使用するには，変数にsreg属性を加えるだけです。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・キーワード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。

＜本Ｃコンパイラから他のＣコンパイラ＞

・＃defineによって行います。詳しくは“11.5　Ｃソースの修正”をご覧ください。これによりsreg変数は，通常の変数として扱われます。

(4) sfr領域利用

sfr領域利用	sfr

【機　能】

・sfr領域は，７８Ｋ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・レベルで記述できます。

・sfr変数は，次の機能を持ちます。

　・sfr変数は，初期値なしの外部変数です。

　・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。定数データ以外の代入チェックは，コンパイル時のオプションによって行います。詳しくは，“ＣＣ７８Ｋシリーズ　Ｃコンパイラ　ユーザーズ・マニュアル　操作編”をご覧ください。

　・８ビットのsfr変数は，自動的にビット型変数になります。

　・使用できるsfr名は，アドレスＦＦ００Ｈ～ＦＦＦＦＨに割り付けてあるものだけです。

【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。

```
#pragma sfr
```

・#pragma sfrは，Ｃソースの先頭に記述します。ただし，次のものは#pragma sfrの前に記述することができます。

　・コメント

　・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。ただし，コンパイル時にシンボル名ケース指定オプションが指定された場合，大文字・小文字の区別はせず大文字と見なされます。この場合には，小文字で記述してもsfr名として扱われます。コンパイル時のオプションについては，“ＣＣ７８Ｋシリーズ　Ｃコンパイラ　ユーザーズ・マニュアル　操作編”をご覧ください。

【使用例】

（Ｃソース）

```
========== sfl.c ============
#ifdef __K3__
        #pragma sfr
#endif

void main()
{
        P0 -= ISPR;
        /* ISPR = 10;     ==> error */
#ifdef __310_
        TXB = 10;
        /* P0  = TXB;     ==> error */
#endif
}
```

（出力オブジェクト）

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。

```
mov     a,P0
sub     a,ISPR
mov     P0,a
```

【説　明】

・この例では，‘#pragma sfr’によりsfr変数を使用することを示しています。プログラム中では，sfr名を使ってＰ０（ポート０）や，ＩＳＰＲ（インサービス・プライオリティ・レジスタ），ＴＸＢ（シリアル・コミュニケーション送信バッファ）などの特殊機能レジスタを利用します。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・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。

＜本Ｃコンパイラから他のＣコンパイラ＞

・‘#pragma sfr’文を削除するかまたは，‘#ifdef’により切り分けます。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。

＜７８Ｋ／０の場合＞

```
#ifdef __K0__
        #pragm sfr
#endif
```

＜７８Ｋ／Ⅱの場合＞

```
#ifdef __K2__
        #pragm sfr
#endif
```

＜７８Ｋ／Ⅲの場合＞

```
#ifdef __K3__
        #pragm sfr
#endif
```

・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。

(5) noauto関数

noauto関数	noauto

【機　能】

・オートマティック変数を使用しない関数は，noauto宣言によりnoauto関数にできます。noauto関数は前後処理（スタック・フレームの形成）のコードがない関数で，関数から，前後処理を省くことにより，オブジェクト・コードの短縮と，実行速度の向上が図れます。

・引数はレジスタ渡しとします。

＜７８Ｋ／０，Ⅱの場合＞

・norec関数扱いとします。

＜７８Ｋ／Ⅲの場合＞

・第１引数をレジスタrp3に，第２引数をレジスタvpに格納します。

・ポインタは優先的にvpに格納します。

【方　法】

・関数宣言の時にnoauto属性を宣言します。

```
noauto 型名　関数名
```

【制　限】

・noauto関数中では，オートマティック変数は使用できません。レジスタ変数も同様です。

・noauto関数に渡す引数には，型と数に制限があります。

＜７８Ｋ／０，Ⅱの場合＞

・norec関数に準じます。

＜７８Ｋ／Ⅲの場合＞

・noauto関数で使用できる引数の型を次に示します。

- ポインタ
- char／signed char／unsigned char
- int／signed int／unsigned int
- short／signed short／unsigned short

・使用できる引数の数は，どの型を使用しても最大２変数です。

・これらの制限は，コンパイル時にチェックされます。

【使用例】

＜７８Ｋ／Ⅲの場合＞

（Ｃソース）

```
========== su1.c ===============

noauto int rout(int a);

int i,j
void main()
{
        int k,l;

        i = 1 + rout(k) + ++k ;
}

noauto  int rout(int a)
{
        return  (i + (a<<2));
}
```

（出力オブジェクト）

関数の先頭の前後処理（スタック・フレームの形成）のコードは出力しません。

```
_rout:                  ;関数名のラベル
        :
        関数本体
        :
        ret
```

ここで変数aは，レジスタrp3に格納されています。

【説　明】

・この例では，ヘッダ部分でnoauto属性を追加しています。関数宣言と関数呼び出しでnoautoを宣言してスタック・フレームの生成を行わないようにしています。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・キーワードnoautoを使用していなければ修正する必要はありません。

・noauto関数に変更する場合前記の方法に従って修正します。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#defineによって行います。詳しくは“11.5　Ｃソースの修正”をご覧ください。

(6) norec関数

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。このため，オブジェクト・コードが短縮でき，プログラムの実行速度が向上します。
・引数は，saddr領域に格納し，norec関数を呼び出します。可能であればレジスタ渡しとします。
・オートマチック変数は，saddr領域に割り当てます。レジスタ変数も同様です。

＜７８Ｋ／０の場合＞

・引数の数によらずレジスタは使用せず，saddr領域（FEB0H～FEBFH）に格納します。第１引数から順にＦＥＢ０Ｈから昇順に格納します。
・オートマティック変数を，saddr領域（FEC0H～FEDFH）に割り当てます。宣言された順にＦＥＣ０Ｈから順に割り当てます。

＜７８Ｋ／Ⅱの場合＞

・引数の数によらずレジスタは使用せず，saddr領域（FEA0H～FEABH）に格納します。第１引数から順にＦＥＡ０から昇順に格納します。
・オートマティック変数を，saddr領域（FEA0H～FEC1H）に割り当てます。宣言された順にＦＥＡ０Ｈから順に割り当てます。

＜７８Ｋ／Ⅲの場合＞

・引数が２つ以内の場合，第１引数をrp3レジスタ，第２引数をvpレジスタに格納します。
・ポインタは優先的にvpに格納します。
・引数が３つ以上の場合，レジスタは使用せず，saddr領域（RB5:FEA0H～FEAFH）に格納します。第１引数から順にＦＥＡ０Ｈから昇順に格納します。
・オートマティック変数を，saddr領域（RB4～RB3:FEB0H～FECFH）に割り当てます。宣言された順にＦＥＢ０Ｈから順に割り当てます。

【方　法】

・関数の宣言時に，norec属性を宣言します。

norec　型名　関数名

【制　限】

・norec関数中から他の関数は，呼び出せません。

・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。

・norec関数中で使用するオートマティック変数に対して，スタティック宣言はできません。

＜７８Ｋ／０，Ⅲの場合＞

・norec関数で使用できる引数の型を次に示します。

・ポインタ

・char／signed char／unsigned char

・int／signed int／unsigned int

・short／signed short／unsigned short

・norec関数に渡す引数の数は，どの型を使用しても最大８変数です。

・norec関数内で使用できるオートマチック変数は，上記の引数の型と同じ型を使用し，総計３２byteまでです。

・char／signed char／unsigned char同士は連続してsaddr領域に割り当てますが，これ以外の組合せでは２byteアラインで割り当てます。

＜７８Ｋ／Ⅱの場合＞

・norec関数で使用できる引数の型を次に示します。

・ポインタ

・char／signed char／unsigned char

・int／signed int／unsigned int

・short／signed short／unsigned short

・norec関数に渡す引数の数は，どの型を使用しても最大６変数です。

・norec関数内で使用できるオートマチック変数は，上記の引数の型と同じ型を使用し，総計２２byteまでです。

【使用例】

（Cソース）

```
========== su1.c ===============
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));
}
```

（出力オブジェクト）

```
        EXTRN   _@NRARG0        ;使用するsaddr領域の参照を行います。
        EXTRN   _@NRARG1        ;（引数用）
        EXTRN   _@NRARG2        ;

        :
movw    _@NRARG0,ax             ;引数をsaddr領域に格納します。
        :
movw    _@NRARG1,ax             ;
        :
movw    _@NRARG2,ax             ;
call    !_rout                  ;norec関数を呼び出します。

        EXTRN   _@NRAT00        ;使用するsaddr領域の参照を行います。
        EXTRN   _@NRAT02        ;（オートマティック変数用）

_rout:
        movw    ax,_@NRARG0     ;saddr領域から引数を受け取ります。
        shlw    ax,2
        addw    ax,_@NRAT00     ;saddr領域のオートマティック変数を使
        movw    bc,ax           ;用します。
        ret
```

【説　明】

rout関数の定義にもnorec関数であることを示すためにnorec属性を付けます。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・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。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#defineによって行います。詳しくは“11.5　Ｃソースの修正”をご覧ください。

(7) bit型変数

bit型変数	bit

【機　能】

・bit型変数は，1ビットのデータを定義します。

・bit型変数は初期値なし（不定）の外部変数と同様に扱えます。

・このビット変数に対してコンパイラは，ビット操作命令を出力できます。これにより，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。

・出力するビット操作命令

　・MOV1,AND1,OR1,XOR1,SET1,CLR1,NOT1,BT,BF命令を出力します。

【方　法】

・bit型変数を使用するモジュール中でbit型宣言を行います。

　　bit　変数名

・bit型変数を参照するモジュール中でbit型宣言を行います。

　　extern bit　変数名

・char型のsreg変数と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0～7）

【制　限】

・bit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できません。

・bit型変数では，次の演算を行えます。ただし，定数との演算は０と１のみです。

分　類	演算子	分　類	演算子
代入	＝	ＮＯＴ	！
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	｜｜	等号	＝＝
不等号	！＝	キャスト	（型名）

・上記以外の演算，または他の型と演算する場合，キャスト演算子を用いて明示的に型変換を行わなければなりません。

＜７８Ｋ／０の場合＞

・bit型変数のみで１つのロード・モジュールあたり最大８６４変数まで使用可能です。ただし，sreg変数を使用すると，使用できる数は減ります。

＜７８Ｋ／Ⅱの場合＞

・bit型変数のみで１つのロード・モジュールあたり最大７３６変数まで使用可能です。ただし，sreg変数を使用すると，使用できる数は減ります。

＜７８Ｋ／Ⅲの場合＞

・bit型変数のみで１つのロード・モジュールあたり使用可能な変数次のとおりです。

μPD78310/312/P312，μPD78310A/312A/P312Aの場合　　：７３６変数まで

μPD78310/312/P312，μPD78310A/312A/P312A以外の場合：６４０変数まで

ただし，sreg変数を使用すると，使用できる数は減ります。

【使用例】

（Ｃソース）

```
=====   bi1.c   =====
#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();
        }
}
```

（出力オブジェクト）

```
PUBLIC  _data1          ;宣言
PUBLIC  _data2

@@BITS BSEG             ;セグメントに割り付けます。
_data1 DBIT
_data2 DBIT
```

関数中では次のようなコードを出力します。

```
set1    _data1          (初期化)
clr1    _data2          (初期化)
bf      _data1,$L0001   (判断)
mov1    CY,_data2       (代入)
mov1    _data1,CY       (代入)
bf      _data1,$L0005   (ＡＮＤ式)
bf      _data2,$L0005   (ＡＮＤ式)
```

【説　明】

data1, data2の宣言でビット型を指定し，ビット型変数であることを示しています。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・キーワードbitを使用していなければ修正する必要はありません。

・bit関数に変更する場合前記の方法に従って修正します。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#defineによって行います。詳しくは“11.5　Ｃソースの修正”をご覧ください。

(8) ＡＳＭ文

ＡＳＭ文	＃asm, ＃endasm

【機　能】

・本Ｃコンパイラ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。これによりＣソースで次のことができます。

　・Ｃソースの引数，グローバル変数をアセンブラ・ソースで操作する。

　・Ｃソースには記述できない機能を実現する。

　・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が得られる。

・#asmの行と＃endasmの行は出力しません。

【方　法】

・#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
```

【制　限】

・‘#asm～#endasm’は，Ｃソースの関数中にしか記述できません。したがって，アセンブラ・ソースはセグメント名@@ＣＯＤＥのＣＳＥＧに出力されます。

・＃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。

【使用例】

（Ｃソース）

```
=====   as1.c   =====

void main()
{
        #asm
                callf !init
        #endasm
}
```

（アセンブラ・ソース）

ユーザの記述したアセンブラ・ソースをアセンブラ・ソース・ファイルへ出力します。

```
@@CODE  CSEG
main:
        callf   !init
        ret
        END
```

【説　明】

・#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。

【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ます。

・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ます。

(9) 漢　　字

漢　　字	／*漢字*／

【機　能】

・Ｃソースのコメント文を漢字入りで記述できます。これによって，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。漢字はシフトＪＩＳコードです。

【方　法】

・Ｃソースのコメント文中に漢字（２byteコード）を記述します。

【制　限】

・ＯＳで漢字をサポートしていない場合は，漢字を使用できません。

・漢字が記述できるのは，コメント文中のみです。

【使用例】

（Ｃソース）

```
=====   ka.c   =====
        void main()     /*  ｍａｉｎ関数  */
        {
                /*  コメント文  */
        }
```

（出力オブジェクト）

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         void main()     /*  main関数  */
        ;line 2         {
        ;line 3                         /*  コメント文  */
```

【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。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・コメント文を書ける以外の場所（‘/*…*/’の外）に漢字がある場合修正しなければなりません。

＜本Ｃコンパイラから他のＣコンパイラ＞

・コメント中に漢字を書ける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けない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。

(10) 割り込み関数

割り込み関数	#pragma vect

【機　能】

・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。

【方　法】

・#pragma vect指令により割り込み要求名と関数名を指定します。ただし，関数の定義，参照が行われる前に記述します。

```
#pragma vect  要求名  関数名
```

【制　限】

・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。

【使用例】

（Ｃソース）

```
#pragma vect NMI  inter

void inter()
{
  /*  ＮＭＩ端子入力に対する割り込み処理  */
}
```

（コンパイラの出力するオブジェクト）

```
@@VECT02  CSEG    AT      02H : NMI
          DW      @inter

@@CODE    CSEG
@inter:
    コンパイラが使用するsaddr空間の退避コード
    全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）
    全レジスタの復帰コード
    コンパイラが使用するsaddr空間の復帰コード
    reti
```

【説　明】

・inter関数が割り込み関数であることを示すために#pragma vectの後に関数名を指定します。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・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。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#pragma vect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。

(11) 割り込み機能

割り込み機能	#pragma DI, #pragma EI

【機　能】

・割り込み禁止の関数を作成します。

・通常#asmを記述するとオブジェクト・ファイルは出力されませんが，この機能を使用するとオブジェクト・ファイルを得ることができます。

・#pragma指令がない場合，ＤＩ（），ＥＩ（）は通常の関数とみなされます。

【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。

- コメント
- 他の#pragma指令
- 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

【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。ただし，コンパイル時にシンボル名ケース指定オプションが指定された場合，大文字，小文字の区別はされず，すべて大文字とみなします。このため，小文字で記述しても割り込み機能として扱われます。詳細は，ＣＣ７８Ｋシリーズ　ユーザーズ・マニュアル　操作編をお読みください。

【使用例】

（Ｃソース）

```
#pragma DI
#pragma EI
void main()
{
        DI();
        関数本体
        EI();
}
```

（出力オブジェクト）

```
_main:
        di
        前処理
        関数本体
        後処理
        ei
        ret
```

【説　明】

・main関数は，割り込み禁止です。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・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。

＜本Ｃコンパイラから他の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。

・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。

例

＜７８Ｋ／Ⅲの場合＞

```
#ifdef  __K3__
        #pragma DI
        #pragma EI
#endif
```

(12) callf関数

callf関数	callf

【機　能】

・callf命令は，callf領域に関数本体を格納し，call命令に比べて短いコードで関数を呼ぶことを可能にします。これにより，オブジェクト・コードを短縮できます。

・callf関数をプロトタイプ宣言なしに参照した場合には，通常のcall命令によって関数を呼びます。

・呼ばれる関数は，通常の関数と何ら変わりません。

・callf宣言された関数（callf関数）の呼び出しには，通常の関数と同じく関数名の先頭に_を付加した名前を使用します。

【方　法】

・関数の宣言時に，callf属性を先頭に追加します。

```
callf extern　型名　関数名
```

【制　限】

・callf宣言された関数は，callf領域に配置します。callf領域のどこに配置するかは，リンク時に決定されます。したがって，アセンブラ・ソース・モジュール中でcallf関数を呼び出す場合は，シンボルを使ったリロケータブルなアセンブラ・ソースで記述する必要があります。

・callf関数の占めるサイズに関するチェックはリンク時に行います。

・callf領域は，８００Ｈ～ＦＦＦＨです。

・callf属性の許される関数は，特に制限はありません。

・callf属性の関数のトータル・サイズは，リンクするモジュールで最大２０４８byteです。

【使用例】

（Cソース）

```
========== cf1.c =============
callf extern int fsub();

void main()
{
        int ret_val;

        ret_val = fsub();
}
```

```
========== cf1.c ============

callf int fsub()
{
        int val;

        return val;
}
```

（コンパイラの出力オブジェクト）

```
cf1のモジュール
        EXTRN   _fsub   ;宣言
        callf   !_fsub  ;呼び出し

cf2のモジュール（callf領域に配置されます。）
        PUBLIC  _tsub   ;宣言

@@CALF  CSEG    FIXED
_tsub:                  ;関数定数
          :
        関数本体
          :
```

【説　明】

・呼ばれる関数‘fsub’は，callf領域に配置するためにcallf属性を加えて宣言します。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・キーワードcallfを使用していなければ修正する必要はありません。

・callf関数に変更したい場合は，前記の方法に従って修正します。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#defineにより使用可能となります。これにより，callf関数は通常の関数として扱われます。

保守／廃止

(13) 1 Mbyte拡張空間利用法

1 Mbyte拡張空間利用法	#pragma extend

【機　能】

・オブジェクトに1 Mbyte拡張空間をアクセス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・ファイルを作成します。

・#asmにより記述すると通常オブジェクト・ファイルは出力されませんが，この機能を使用することによりオブジェクト・ファイルを得ることができます。

・#pragma指令がない場合は，1 Mbyte拡張用の関数は通常の関数とみなされます。

・７８Ｋ／０，Ⅲは，本機能をサポートしません。

【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。1 Mbyte拡張空間用の関数名には，次のものがあります。

```
extgetb
extgetw
extputb
extputw
extcpy
extcpyn
extcpy0
```

・#pragma extend指令をＣソースの先頭に記述します。

```
#pragma extend
```

次の項目は，#pragma extendの前に記述することが可能です。

・コメント

・他の#pragma指令

・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】

・関数名として１Mbyte拡張空間用の関数は使用できません。大文字は通常の関数として扱われます。ただし，コンパイル時にシンボル名ケース指定オプションが指定された場合，大文字，小文字の区別はされず，すべて大文字とみなします。このため，大文字で記述しても１Mbyte拡張空間用の関数として扱われます。詳細は，ＣＣ７８Ｋシリーズ　ユーザーズ・マニュアル　操作編をお読みください。

【使用例】

＜７８Ｋ／Ⅱの場合＞

（Ｃソース）

```
#pragma extend

char c;
void main()
{
        c = (char)extgetb(0x12345)
}
```

（コンパイラの出力オブジェクト）

```
_main:
        and     P6,#0F0H        ;P6レジスタの下位4ビットを0クリア
        or      P6,#01H         ;P6レジスタの下位4ビットを0x01を設定
        movw    de,#02345H      ;deレジスタにアドレスの下位16ビットを設定
        mov     a,&[de]
        mov     !_c,a
        ret
```

【説　明】

・Ｃソースの先頭に#pragma extend指令が記述されているので，関数 extgetbは１Mbyte拡張空間用の関数として扱われます。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・1 Mbyte拡張空間用の関数を使用していなければ修正する必要はありません。

・1 Mbyte拡張空間用の関数に変更する場合，前記の方法に従って修正します。

＜本Ｃコンパイラから他のＣコンパイラ＞

・“#pragma extend”文を削除するか，あるいは#ifdefで切り分けます。関数名として1 Mbyte拡張空間用の関数名を使用することができます。

・1 Mbyte拡張空間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。

```
#ifdef __K2__
        #pragma extend
#endif
```

【1 Mbyte拡張空間用の関数一覧】

(1) unsigned char extgetb(extadr);

unsigned long extadr;

２０ビットのアドレスextadrの内容１バイトを返します。

(2) unsigned int extgetw(extadr);

unsigned iong extadr;

２０ビットのアドレスextadrの内容２バイトを返します。

(3) void extputb(extadr, c);

unsigned iong extadr;

unsigned char c;

２０ビットのアドレスextadrが指す位置に，１バイトのデータｃをかき込みます。

(4) void extputw(extadr, i);

unsigned iong extadr;

unsigned int i;

２０ビットのアドレスextadrが指す位置に，２バイトのデータiを書き込みます。

(5) void extcpy(d_extadr, s_extadr, n);

unsigned long d_extadr;

unsigned long s_extadr;

unsigned char n;

ｎバイト分のデータを２０ビットのアドレスs_extadrから２０ビットのアドレスd_extadrにコピーします。

(6) void extcpyn(d_extadr, src, n);

unsigned long d_extadr;

char *src;

unsigned char n;

nバイト分のデータをバンク０のsrcから２０ビットのアドレスd_extadrにコピーします。

(7) void extcpy0(dst, s_extadr, n);

char *dst;

unsigned long s_extadr;

unsigned char n;

nバイト分のデータを２０ビットのアドレスs_extadrからバンク０のdstにコピーします。

(14) テーブル切り換え機能

テーブル切り換え機能	#pragma table

【機　能】

・Ｃコンパイラの出力するベクタ・テーブル，calltテーブルのアドレスを変更することができます。

・７８Ｋ／０，Ⅱは，本機能をサポートしません。

【方　法】

・#pragma vectを使用しているすべてのＣソースの先頭に#pragma table 0または#pragma table 1を記述します。

```
#pragma table 0
#pragma table 1
```

・次の項目は，#pragma tableの前に記述することができます。

- ・コメント
- ・他の#pragma指令
- ・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

【制　限】

・#pragma tableを２回以上指定した場合は後者優先となります。

・#pragma table 0と指定した場合と#pragma table 1と指定した場合では，ベクタ・テーブル，calltテーブルのアドレスが異なります。

＜#pragma table 0（ディフォールト）指定時＞

ベクタ・テーブル：００Ｈ～３ＦＨ

calltテーブル　：４０Ｈ～７ＦＨ

＜#pragma table 1 指定時＞

ベクタ・テーブル：８０００Ｈ～８０３ＦＨ

calltテーブル　：８０４０Ｈ～８０７ＦＨ

・μＰＤ７８３１０／３１２，μＰＤ７８３１０Ａ／３１２Ａでは，ベクタ・テーブル，calltテーブルを変更する機能がないためサポートしません。

・アドレスを変更した場合（#pragma table 1 指定時）は，スタート・アップ・モジュール中でＣＰＵコントロール・ワード（ＣＣＷ）のビット１（ＴＰＦ）を１にする必要があります。

【使用例】

（Ｃソース）

```
#pragma table 1
#pragma vect NMI  inter

void inter()
{
  /*  ＮＭＩ端子入力に対する割り込み処理  */
}
```

（コンパイラの出力するオブジェクト）

```
@@VECT02  CSEG    AT      8002H : NMI
          DW      @inter

@@CODE    CSEG
@inter:
    コンパイラが使用するsaddr空間の退避コード
    全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）
    全レジスタの復帰コード
    コンパイラが使用するsaddr空間の復帰コード
    reti
```

【説　明】

・#pragma table 1 と指定しているのでベクタ・テーブル，calltテーブルのアドレスが変更されます。

【互換性】

＜他のＣコンパイラから本Ｃコンパイラ＞

・#pragma tableを使用していなければ修正する必要はありません。

・テーブルのアドレスを変更する場合，前記の方法に従って修正します。

＜本Ｃコンパイラから他のＣコンパイラ＞

・#pragma table指定を削除します。

保守／廃止

11.5 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７８Ｋ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コンパイラから本Ｃコンパイラへの移植と，本Ｃコンパイラから他の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コンパイラから本Ｃコンパイラ＞

・#pragma

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。

＜本Ｃコンパイラから他のＣコンパイラ＞

本Ｃコンパイラは，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。

【使用例】

＜７８Ｋ／Ⅲの場合＞

①キーワードを無効にする

```
#ifndef __K3__
        #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 __K3__
        #define bit char       /* bit型変数をchar型変数にします。*/
#endif
```

７８Ｋ／０，Ⅱの場合も同様です。

保守／廃止

第１２章　アセンブラとの相互参照

本章では，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。本章では，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，本ＣコンパイラとＲＡ７８Ｋシリーズ　アセンブラ・パッケージを使い次の順序で説明します。

(1) Ｃ言語からアセンブリ言語ルーチンの呼び出し
(2) アセンブリ言語からＣ言語関数の呼び出し
(3) Ｃ言語で定義した変数を参照する方法
(4) 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
(5) その他注意事項

なお，プログラムは，７８Ｋ／Ⅲを例として説明します。

12.1 C言語からアセンブリ言語ルーチンの呼び出し

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

(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 */
}
```

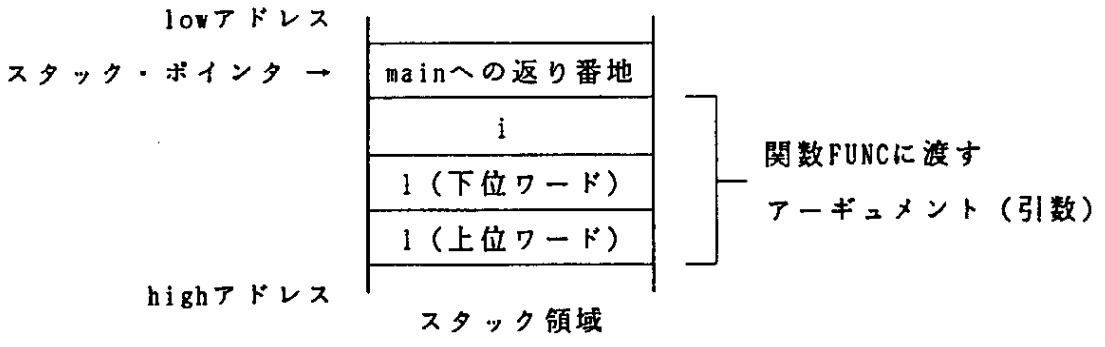
このプログラム例で，実行時に行われるプログラム間のインタフェースと制御の流れを次に示します。

①関数mainから関数FUNCへ渡す引数（アーギュメント）をスタックに積む

②CALL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。

図１２－１　コール後のスタック領域



(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，次の処理を行います。

①ベース・ポインタを退避する

②ワーク・レジスタ（rp3，vp）を退避する

③スタック・ポインタ（sp）をベース・ポインタ（up）へコピーする

④関数FUNC本来の処理を行う

⑤返り値をセットする

⑥退避したレジスタ（up，rp3，vp）を復帰する

⑦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(310)
$NODEBUG

        NAME    FUNC

        PUBLIC  _FUNC

@@CODE  CSEG
_FUNC:
        push    up                      ; save base pointer ·····①
        movw    ax,sp                   ; copy stack pointer ····③
        movw    up,ax
;
        mov     a,[up+4]        ; arg1
        xch     a,x                     ; move 1st argument(i)
        mov     a,[up+5]        ; arg1
        mov     !_DT1+1,a
        xch     a,x
        mov     !_DT1,a
;
        mov     a,[up+8]        ; arg2
        xch     a,x                     ; move 2nd argument(low of 1)
        mov     a,[up+9]        ; arg2                          ④
        movw    de,ax
        mov     a,[up+6]        ; arg2
        xch     a,x                     ; move 2nd argument(high of 1)
        mov     a,[up+7]        ; arg2
        mov     !_DT2+1,a
        xch     a,x
        mov     !_DT2,a
        xchw    ax,de
        mov     !_DT2+3,a
        xch     a,x
        mov     !_DT2+2,a
;
        movw    bc,#0AH ;10             ; set return value ·····⑤
;
        pop     up                      ; restore base pointer ··⑥
        ret ·····⑦

@@DATA  DSEG
_DT1:   DB      (2)
_DT2:   DB      (4)
        END
```

①ベース・ポインタの退避

関数名（入り口となるレーベル）を大文字で記述します。ただし，アセンブル時に大文字，小文字を区別するオプションを指定した場合は大文字で記述する必要はありません。Ｃソース中で記述した関数名と同じ名前になります。

②ワーク・レジスタ（rp3，vp）の退避

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。

呼び出し側でレジスタ変数を使っていない場合‘rp3，vp’レジスタを退避する必要はありません。

③スタック・ポインタ（sp）のベース・ポインタ（up）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‘up’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。

④関数FUNC本来の処理を行う

‘①～③’の処理を行った後呼び出される関数の本来の処理を行います。

⑤返り値のセット

返り値がある場合，返り値を‘bc’，‘rp2’レジスタへセットします。返り値が無い場合，セットする必要はありません。

	bcレジスタ
返り値１６ビット以内：	ワード

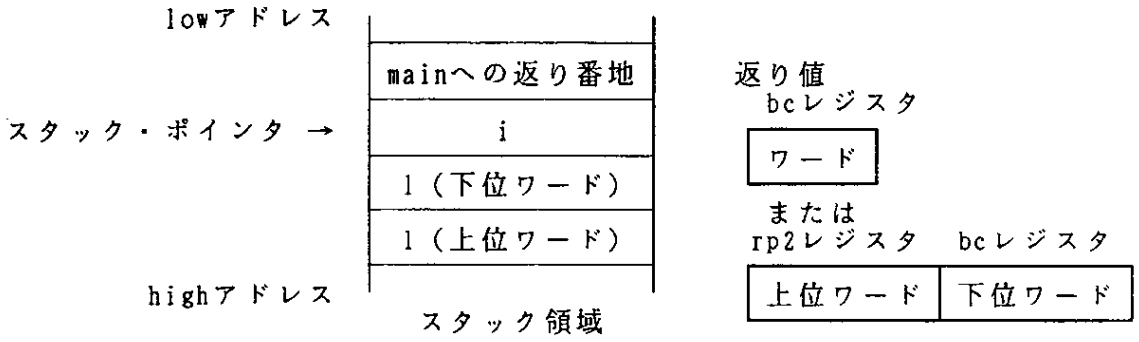
	rp2レジスタ	bcレジスタ
返り値１７ビット以上：	上位ワード	下位ワード

⑥レジスタ（up，rp3，vp）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。

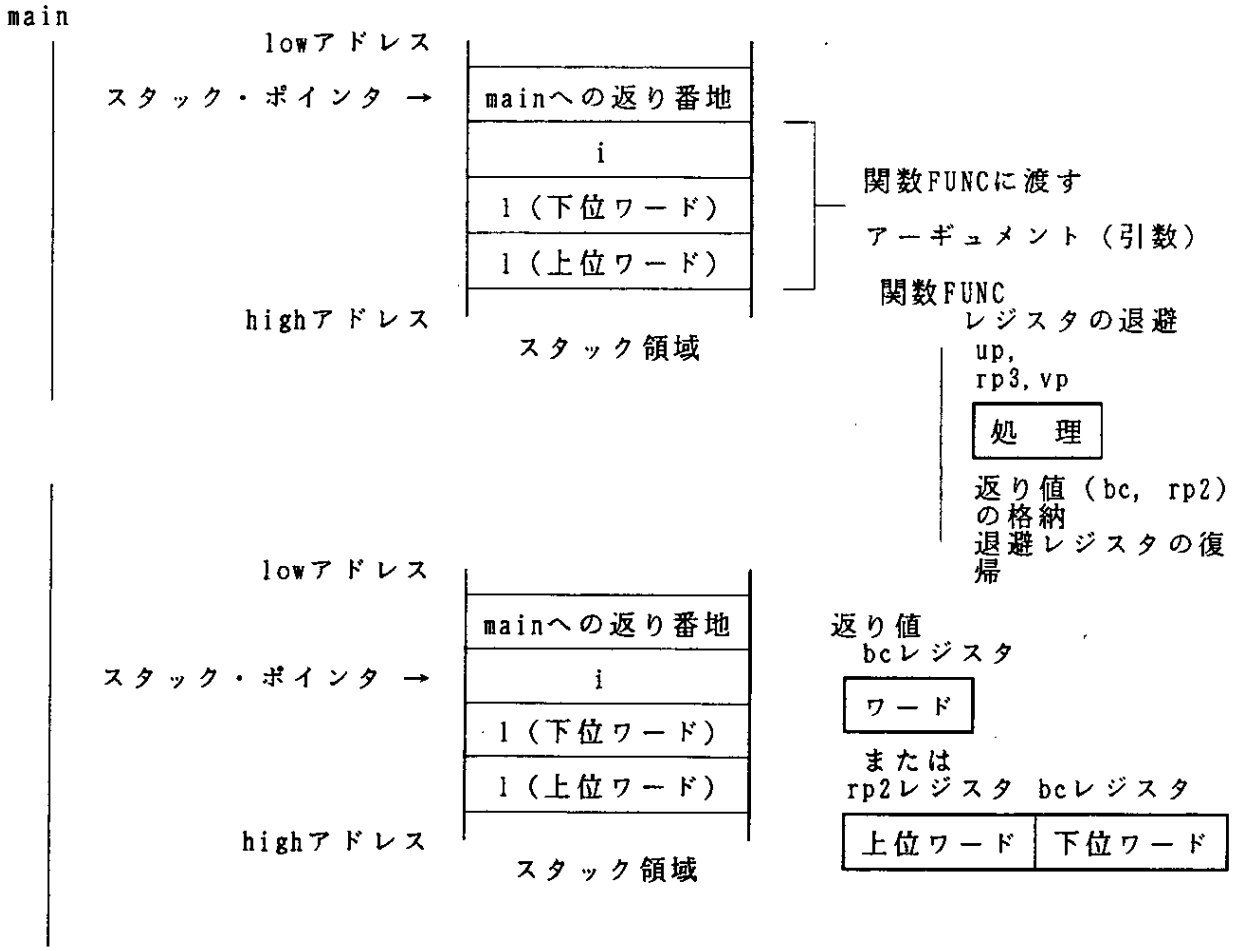
⑦関数mainへのリターン

図１２－２　リターン後のスタック領域



Ｃ言語からアセンブリ言語ルーチンを呼び出す手順とアセンブリ言語ルーチンの処理を図にすると次のようになります。

図１２－３　Ｃ言語からアセンブリ言語の呼び出し



保守／廃止

12.2 アセンブリ言語からC言語ルーチンの呼び出し

(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引数をスタックに積む

②Cのワーク・レジスタ（ax, bc, rp2, de, hl）を退避する

③C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

⑤C言語関数の返り値（bcまたはrp2, bc）を参照する

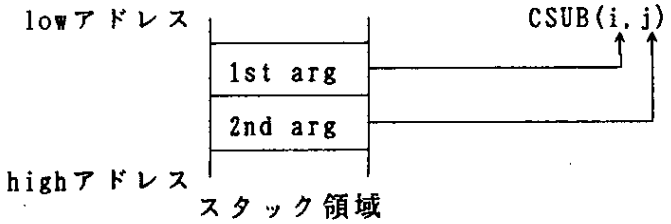
アセンブリ言語のプログラム例を次に示します。

```
$PROCESSOR(310)
$NODEBUG

        NAME    FUNC2

        PUBLIC  _FUNC2

@@CODE  CSEG
_FUNC2:
        movw    ax,#20H         ; set 2nd argument(j)
        push    ax              ;
        movw    ax,#21H         ; set 1st argument(i)
        push    ax              ;
        call    !_CSUB          ; call "CSUB(I,J)"
        movw    ax,sp           ; modify for argument push
        addw    ax,#04H         ;
        movw    sp,ax           ;
        ret
        END
```

①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，次の図１２－４のようになります。

図１２－４　スタックへの引数の積み込み



②ワーク・レジスタ（ax, bc, rp2, de, hl）の退避

Ｃ言語では，‘ax, bc, rp2, de, hlの５つのレジスタをワーク用として使用し，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。

③Ｃ言語関数のコール

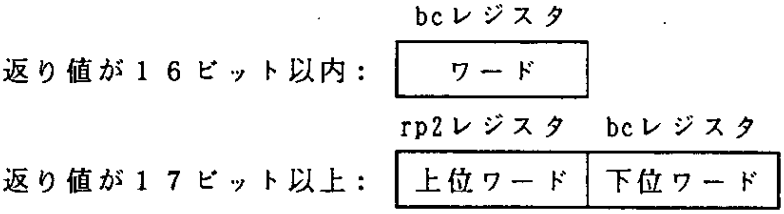
Ｃ言語関数の呼び出しは，ＣＡＬＬ命令で行います。Ｃ言語関数がcallt関数の場合，‘callt’命令で呼び出します。

④スタック・ポインタ（sp）の修正

引数を積んだバイト数分スタック・ポインタを修正します。例では，４バイト分の引数を渡すのでスタック・ポインタに４を加えます。

⑤返り値（bc, rp2）の参照

Ｃ言語からの返り値は次のように返されます。

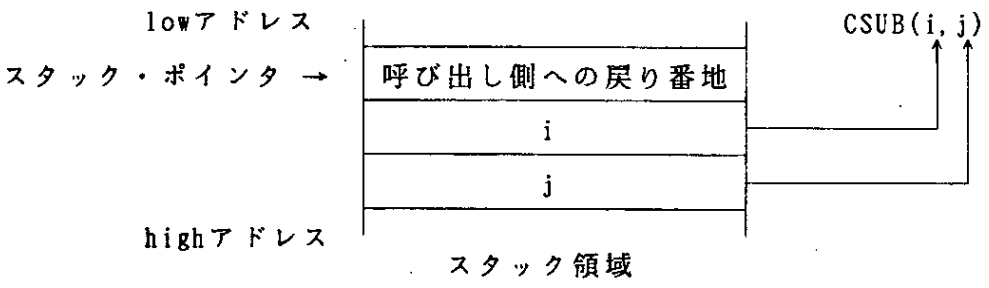


(2) Ｃ言語関数の引数参照方法

次に示すＣ言語プログラムに引数‘i, j’を正しく渡すにはスタックに“図１２－５Ｃ言語への引数の受け渡し”のように積みます。

```
void CSUB(i,j)
int     i,j ;
{
        i += j ;
}
```

図１２－５　Ｃ言語への引数の受け渡し



12.3 他言語で定義された変数の参照

(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‘_’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() ;
}
```

ＲＡ７８Ｋ／Ⅲアセンブラでは次のように行います。

（この例では，アセンブル時に大文字，小文字を区別するオプションを指定する必要があります）

```
$PROCESSOR(310)
$NODEBUG

        NAME    subf

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf:
        MOV     _c,#04H
        MOV     _i,#07H
        RET
        END
```

(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 ;
}
```

保守／廃止

ＲＡ７８Ｋ／Ⅲアセンブラでは次のように行います。

（この例では，アセンブル時に大文字，小文字を区別するオプションを指定する必要があります）

```
            NAME    ASMSUB

            PUBLIC  _c
            PUBLIC  _i

ABC         DSEG
_c          db      0
_i          dw      0

            END
```

12.4　その他注意事項

(1) 外部名の先頭には‘_’が付けられる

本Ｃコンパイラは，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,1) ;’は，‘‘_FUNC’という外部名を参照する’と訳されます。

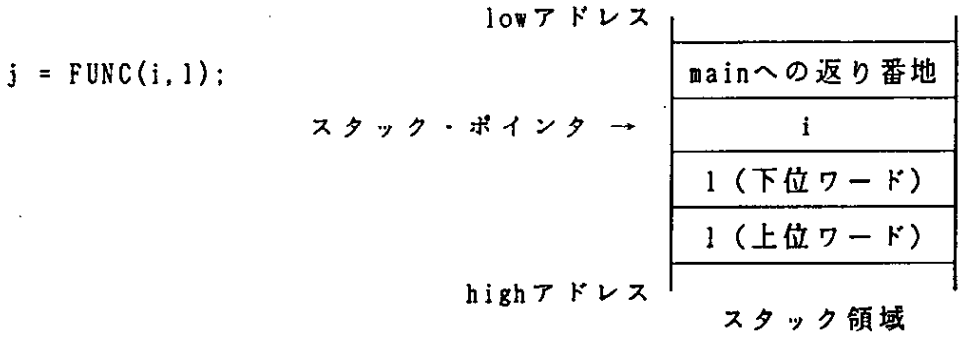
```
extern int FUNC(int, long);    /* 関数プロトタイプ */

void main()
{
        int         i,j;
        long        l;

        i = 1;
        l = 0x54321;
        j = FUNC(i,l);          /* 関数コール */
}
```

ＲＡ７８Ｋ／Ⅲ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。

図１２－６　引数のスタック配置



保守／廃止

(3) ランタイム・ライブラリー覧

以下の演算の命令は，@@などを関数名の頭に付けた形式で呼び出されます。

表１２－１　ランタイム・ライブラリー覧（１／４）

分　類	関数名	機　　能	ｴﾗｰ・ﾁｪｯｸ 注1	ｻﾎﾟｰﾄ 注2		
				0	Ⅱ	Ⅲ
ｲﾝｸﾘﾒﾝﾄ	csinc	signed charをｲﾝｸﾘﾒﾝﾄする	2	○	○	○
	cuinc	unsigned charをｲﾝｸﾘﾒﾝﾄする	2	○	○	○
	isinc	signed intをｲﾝｸﾘﾒﾝﾄする	2	○	○	○
	iuinc	unsigned intをｲﾝｸﾘﾒﾝﾄする	2	○	○	○
	lsinc	signed longをｲﾝｸﾘﾒﾝﾄする	2	◎	◎	◎
	luinc	unsigned longをｲﾝｸﾘﾒﾝﾄする	2	◎	◎	◎
ﾃﾞｸﾘﾒﾝﾄ	csdec	signed charをﾃﾞｸﾘﾒﾝﾄする	2	○	○	○
	cudec	unsigned charをﾃﾞｸﾘﾒﾝﾄする	2	○	○	○
	isdec	signed intをﾃﾞｸﾘﾒﾝﾄする	2	○	○	○
	iudec	unsigned intをﾃﾞｸﾘﾒﾝﾄする	2	○	○	○
	lsdec	signed longをﾃﾞｸﾘﾒﾝﾄする	2	◎	◎	◎
	ludec	unsigned longをﾃﾞｸﾘﾒﾝﾄする	2	◎	◎	◎
符号反転	csrev	signed charを符号反転する	2	○	○	○
	curev	unsigned charを符号反転する	2	○	○	○
	isrev	signed intを符号反転する	2	○	○	○
	iurev	unsigned intを符号反転する	2	○	○	○
	lsrev	signed longを符号反転する	2	◎	◎	◎
	lurev	unsigned longを符号反転する	2	◎	◎	◎
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	×	◎	◎	◎
乗　　算	csmul	signed charどうしの乗算	1	◎	○	◎
	cumul	unsigned charどうしの乗算	1	◎	○	○
	ismul	signed intどうしの乗算	1	◎	◎	◎
	iumul	unsigned intどうしの乗算	1	◎	◎	○
	lsmul	signed longどうしの乗算	1	◎	◎	◎
	lumul	unsigned longどうしの乗算	1	◎	◎	◎

保守／廃止

表１２－１　ランタイム・ライブラリ一覧（２／４）

分　類	関数名	機　　能	エラー・チェック 注1	サポート 注2		
				0	II	III
除　　算	csdiv	signed charどうしの除算	0,1	◎	◎	◎
	cudiv	unsigned charどうしの除算	0	◎	○	○
	isdiv	signed intどうしの除算	0,1	◎	◎	◎
	iudiv	unsigned intどうしの除算	0	◎	◎	○
	lsdiv	signed longどうしの除算	0,1	◎	◎	◎
	ludiv	unsigned longどうしの除算	0	◎	◎	◎
剰 余 算	csrem	signed charどうしの剰余算	0	◎	◎	◎
	curem	unsigned charどうしの剰余算	0	◎	○	○
	isrem	signed intどうしの剰余算	0	◎	◎	◎
	iurem	unsigned intどうしの剰余算	0	◎	◎	○
	lsrem	signed longどうしの剰余算	0	◎	◎	◎
	lurem	unsigned longどうしの剰余算	0	◎	◎	◎
加　　算	csadd	signed charどうしの加算	2	○	○	○
	cuadd	unsigned charどうしの加算	2	○	○	○
	isadd	signed intどうしの加算	2	○	○	○
	iuadd	unsigned intどうしの加算	2	○	○	○
	lsadd	signed longどうしの加算	2	◎	◎	◎
	luadd	unsigned longどうしの加算	2	◎	◎	◎
減　　算	cssub	signed charどうしの減算	2	○	○	○
	cusub	unsigned charどうしの減算	2	○	○	○
	issub	signed intどうしの減算	2	○	○	○
	iusub	unsigned intどうしの減算	2	○	○	○
	lssub	signed longどうしの減算	2	◎	◎	◎
	lusub	unsigned longどうしの減算	2	◎	◎	◎

保守／廃止

表１２－１　ランタイム・ライブラリー覧（３／４）

分　類	関数名	機　　能	ｴﾗｰ・ﾁｪｯｸ 注1	ｻﾎﾟｰﾄ 注2		
				0	Ⅱ	Ⅲ
左シフト	cslsh	signed charの左シフト	2	○	○	○
	culsh	unsigned charの左シフト	2	○	○	○
	islsh	signed intの左シフト	2	◎	○	○
	iulsh	unsigned intの左シフト	2	◎	○	○
	lslsh	signed longの左シフト	2	◎	◎	◎
	lulsh	unsigned longの左シフト	2	◎	◎	◎
右シフト	csrsh	signed charの右シフト	2	◎	◎	◎
	cursh	unsigned charの右シフト	2	○	○	○
	isrsh	signed intの右シフト	2	◎	◎	◎
	iursh	unsigned intの右シフト	2	◎	○	○
	lsrsh	signed longの右シフト	2	◎	◎	◎
	lursh	unsigned longの右シフト	2	◎	◎	◎
比　　較	cscmp	signed charどうしの比較	×	◎	◎	－
	iscmp	signed intどうしの比較	×	◎	◎	－
	lscmp	signed longどうしの比較	×	◎	◎	◎
	lucmp	unsigned longどうしの比較	×	◎	◎	◎
ビット	lsband	signed longどうしのﾋﾞｯﾄＡＮＤ	×	◎	◎	◎
ＡＮＤ	luband	unsigned longどうしのﾋﾞｯﾄＡＮＤ	×	◎	◎	◎
ビット	lsbor	signed longどうしのﾋﾞｯﾄＯＲ	×	◎	◎	◎
ＯＲ	lubor	unsigned longどうしのﾋﾞｯﾄＯＲ	×	◎	◎	◎
ビット	lsbxor	signed longどうしのﾋﾞｯﾄＸＯＲ	×	◎	◎	◎
ＸＯＲ	lubxor	unsigned longどうしのﾋﾞｯﾄＸＯＲ	×	◎	◎	◎
論理	lsand	signed longどうしの論理ＡＮＤ	×	◎	◎	◎
ＡＮＤ	luand	unsigned longどうしの論理ＡＮＤ	×	◎	◎	◎
論理ＯＲ	lsor	signed longどうしの論理ＯＲ	×	◎	◎	◎
	luor	unsigned longどうしの論理ＯＲ	×	◎	◎	◎

保守／廃止

表１２－１　ランタイム・ライブラリ一覧（４／４）

分　類	関数名	機　　能	ｴﾗｰ・ﾁｪｯｸ 注1	ｻﾎﾟｰﾄ 注2		
				0	II	III
bitから	btoc	bitをcharに変換する	×	◎	◎	◎
の変換	btoi	bitをintに変換する	×	◎	◎	◎
	btol	bitをlongに変換する	×	◎	◎	◎
bitへの	cstob	signed charをbitに変換する	1	○	○	◎
変換	cutob	unsigned charをbitに変換する	1	○	○	◎
	istob	signed intをbitに変換する	1	○	○	◎
	iutob	unsigned intをbitに変換する	1	○	○	◎
	lstob	signed longをbitに変換する	1	○	○	◎
	lutob	unsigned longをbitに変換する	1	○	○	◎
ｽﾀｰﾄｱｯﾌﾟ・ﾙｰﾁﾝ	cstart	システムの実行に必要な準備を行う	×	◎	◎	◎
関数前後	cprep	関数の前処理	×	◎	◎	◎
処理	cdisp	関数の後処理	×	◎	◎	◎
エラー・	chkstk	ｽﾀｯｸ・ｵｰﾊﾞﾌﾛｰのチェック	×	◎	◎	◎
チェック	asto8	sfrアクセス・チェック(8ﾋﾞｯﾄ)	×	◎	◎	◎
	asto16	sfrアクセス・チェック(16ﾋﾞｯﾄ)	×	◎	◎	◎
ｴﾗｰ処理	errstk	ｽﾀｯｸ・ｵｰﾊﾞﾌﾛｰのエラー処理ルーチン	×	◎	◎	◎
ﾙｰﾁﾝ	errdiv	０除算のエラー処理ルーチン	×	◎	◎	◎
	errptr	ﾎﾟｲﾝﾀ・ｱｸｾｽのエラー処理ルーチン	×	◎	◎	◎
	errovf	オーバフローのエラー処理ルーチン	×	◎	◎	◎
	errsfr	sfrアクセスのエラー処理ルーチン	×	◎	◎	◎
	errini	ＲＯＭ化領域のエラー処理ルーチン	×	◎	◎	◎

注1. コンパイル時に－Lオプションの指定を行った場合，各関数がサポートするエラー・チェックの内容を次の記号および数字で示します。

エラー・チェックなし	・・・・・・・・	×
0除算	・・・・・・・・	0
オーバフロー1	・・・・・・・・	1
オーバフロー2	・・・・・・・・	2
ＲＯＭ化領域	・・・・・・・・	3

2. 各デバイス・シリーズのライブラリ・サポート方法について次に示します。

◎ ・・・・ エラー・チェック指定オプションの有無にかかわらず，ライブラリが呼び出されます。

○ ・・・・ エラー・チェック指定オプションが指定されたときのみ，ライブラリが呼び出されます。

－ ・・・・ ライブラリでサポートされません。コンパイラがインライン展開を行います。

3. ポインタ・アクセス・チェックを行う標準ライブラリを次に示します。

setjmp, sscanf, strtol, strtoul, itoa, ltoa, ultoa, strtok, sprintf, qsort, memcpy, memmove, memset, strcpy, strncpy, strcat, strncat

4. オーバフロー・チェックを行う標準ライブラリを次に示します。

abs(2), labs(2), div(1), ldiv(1)

5. 0除算チェックを行う標準ライブラリを次に示します。

div, ldiv

なお，以上の表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。

保守／廃止

第１３章　効率の良い活用法

本章では，本Ｃコンパイラを有効に利用する方法を紹介します。

13.1 コンパイル時のコマンド入力

本Ｃコンパイラでは，ソース・ファイルのコンパイル時にターゲット・デバイスの種別を指定します。このデバイス種別の指定は，Ｃソース中に記述することができます。

＜７８Ｋ／Ⅲの場合＞

・デバイスのシリーズ名を示すマクロ名

‘＿＿Ｋ３＿＿’

・デバイス名を示すマクロ名

表１３－１　デバイス種別のマクロ名

デバイス名	種　別	マクロ名
μＰＤ７８３１０	３１０	＿＿３１０＿
μＰＤ７８３１２，μＰＤ７８Ｐ３１２	３１２	＿＿３１２＿
μＰＤ７８３１０Ａ	３１０Ａ	＿＿３１０Ａ
μＰＤ７８３１２Ａ，μＰＤ７８Ｐ３１２Ａ	３１２Ａ	＿＿３１２Ａ
μＰＤ７８３２０	３２０	＿＿３２０＿
μＰＤ７８３２２，μＰＤ７８Ｐ３２２	３２２	＿＿３２２＿
μＰＤ７８３２３	３２３	＿＿３２３＿
μＰＤ７８３２４	３２４	＿＿３２４＿
μＰＤ７８３２７	３２７	＿＿３２７＿
μＰＤ７８３２８，μＰＤ７８Ｐ３２８	３２８	＿＿３２８＿
μＰＤ７８３３０	３３０	＿＿３３０＿
μＰＤ７８３３４，μＰＤ７８Ｐ３３４	３３４	＿＿３３４＿
μＰＤ７８３５０	３５０	＿＿３５０＿

コンパイル時のデバイス種別の指定方法は次のようにします。

コンパイル時のコマンド行に次の指定を加える

'－c種別'

（指定方法の詳しい説明は，“ＣＣ７８Ｋシリーズ　Ｃコンパイラ　ユーザーズ・マニュアル　操作編”をご覧ください）

次のように，Ｃソース中で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。

Ｃソース・プログラムの先頭に次の文を指定します。

'#pragma PC(種別)'

ただし，次のものは'#pragma PC(種別)'の前に記述することが可能です。

・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

13.2 効率の良いコーディング

７８Ｋシリーズ応用製品の開発を行う場合，本Ｃコンパイラではデバイスのsaddr領域，callt領域あるいはcallf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変数を使用する

・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関数とする
  │ ├ if（callf領域が使用可能）
  │ │ └ callf関数とする
  │ └ if（再帰的に使用しない）
  │ 　└ norec関数とする
  ├ if（オートマティック変数を使用しない）
  │ └ noauto関数とする
  └ if（オートマティック変数を使用する　＆＆　saddr領域が利用可能）
  　└ register宣言する

(1) 外部変数の定義

外部変数を定義する時にsaddr領域が利用可能であれば，定義する外部変数をsreg変数にします。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・コードを縮小でき，実行速度も向上します。

sreg変数の定義：　extern sreg int 変数名;“11.4 (3)saddr領域の利用”

(2) 1ビットのデータ

1ビットのデータしか使用しないオブジェクトは，bit変数にします。bit変数に対する操作には，ビット操作命令が生成されます。

bit変数の宣言：　bit 変数名;“11.4 (7)bit型変数”

(3) 関数定義

・何回も呼ばれる関数

何回も呼ばれる関数は，オブジェクト・コードを短縮するか，高速に呼び出せる構造にする必要があります。したがって，何回も呼ばれる関数で，callt領域を利用できる関数はcallt関数にし，callf領域を利用できる関数はcallf関数にします。callt/callf関数は，デバイスのcallt/callf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呼び出せます。

何回も呼ばれる関数の中で再帰的に使用しないものは，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・コードが短縮でき，実行速度も向上します。

callt関数の定義：　callt int tsub() “11.4　(1)callt関数”
callf関数の定義：　callt int tsub() “11.4　(12)callf関数”
norec関数の定義：　norec int rout() “11.4　(6)norec関数”

・オートマティック変数を使用しない関数

オートマティック変数を使用しない関数は，noauto関数にします。noauto関数は，スタック・フレームのない関数です。また，引数も可能な限りレジスタ渡しとなります。オブジェクト・コードの短縮ができ，実行速度が向上します。

noautoを宣言しない関数でも，オートマティック変数を使用せずに引数をレジスタ宣言することにより，noauto関数と同じ機能が得られます。

noauto関数の定義： noauto int sub1(int i)“11.4 (5)noauto関数”

register変数の定義：int sub1(register int i)“11.4 (2)レジスタ変数”

・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マイクロプロセッサの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。

register変数の定義：register int i“11.4 (2)レジスタ変数”

保守／廃止

付録A　性能一覧

項番	項　　目	制　限　値	ページ
1	複文，繰り返し制御文，選択制御文のネスト	45	101
2	条件コンパイルのネスト	255	133
3	修飾宣言子のネスト	12	33
4	式中のかっこのネスト	32	61
5	マクロ名で意味を持つ文字数	31	144
6	内部，外部シンボル名で意味を持つ文字数	7　注1	－
7	1ソース・モジュール・ファイル中のシンボル数	1024　注2	－
8	1ブロックでブロック・スコープを持つシンボル数	255　注2	16
9	1ソース・モジュール・ファイル中のマクロ数	1024　注3	144
10	関数定義，関数呼び出しのパラメータ	39	128
11	1つのマクロ定義，マクロ呼び出しのパラメータ	31	144
12	1つの論理ソース行の文字数	509	－
13	結合後の文字列リテラル内の文字数	509	29
14	1つのオブジェクト・サイズ（データを示します）	65535byte	－
15	#includeのネスト	8	140
16	switch文のcaseラベル数	257	104
17	1コンパイル単位のソース行数	約3000	－
18	テンポラリ・ファイルを作成せずにコンパイルできるソース行数	約300	－
19	関数コールのネスティング	40	128
20	1ステートメントの前のラベル数	33	103
21	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	65535byte	－
22	1つの構造体または，共用体のメンバ数	127	120
23	1つの列挙の列挙定数の数	127	39
24	1つの構造体または，共用体における構造体または共用体のネスト	15	120
25	初期化子要素のネスト	15	－

・ 項番5，6，12，13，14，15，21以外は制限ではなく，保証されている最小の値です。

注1. コンパイラ・オプション（－Ｓ）により、シンボル名の長さを３０文字に拡張することができます。

2. 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できる制限値を示します。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，そのときの制限値はファイル・サイズにより変わります。

3. コンパイラの予約マクロ定義を含みます。

付録B　構文一覧

構文要約中の凡例を次に示します。

::=　：左辺の意味を以降に示す
「　」：かっこ内は省略可能
|　：行の左端にある‘|’は‘または’を示す

言語仕様の構文要約を次に示します。

B.1　構成要素

トークン::=
　　キーワード
　|　識別子
　|　定数
　|　文字列リテラル
　|　演算子
　|　区切り子

前処理トークン::=
　　ヘッダ名
　|　識別子
　|　前処理数
　|　文字列定数
　|　文字列リテラル
　|　演算子
　|　区切り子
　|　上記以外の空白でない文字

B.1.1 キーワード

キーワード::= 次に示すいずれか１つ

auto	bit	break	callf	callt	case
char	const	continue	default	do	else
enum	extern	for	goto	if	int
long	noauto	norec	register	return	short
signed	sizeof	sreg	static	struct	switch
typedef	union	unsigned	void	volatile	while

B.1.2 識別子

識別子::=

　　非数字
　| 識別子　非数字
　| 識別子　数字

非数字::= 次に示すいずれか１つ

　　_　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

数字::= 次に示すいずれか１つ

　　0 1 2 3 4 5 6 7 8 9

B.1.3 定数

定数::=

浮動小数点定数
| 整数定数
| 列挙定数
| 文字定数

整数定数::=

１０進定数　「整数添字」
| ８進定数　「整数添字」
| １６進定数　「整数添字」

１０進定数::=

０以外の数字
| １０進定数　数字

０以外の数字::=　次に示すいずれか１つ

1　2　3　4　5　6　7　8　9

８進定数::=

0
| ８進定数　８進数字

８進数字::=　次に示すいずれか１つ

0　1　2　3　4　5　6　7

１６進定数::=

０x　１６進定数
| ０X　１６進定数
| １６進定数　１６進数字

１６進数字::＝　次に示すいずれか１つ

0　1　2　3　4　5　6　7　8　9

a　b　c　d　e　f

A　B　C　D　E　F

整数添字::＝

符号なし添字　「long添字」

| long添字　「符号なし添字」

符号なし添字::＝　以下のうちのいずれか１つ

u　U

long添字::＝　以下のうちいずれか１つ

l　L

列挙定数::＝

識別子

文字定数::＝

’c文字の列’

c文字の列::＝

c文字

| c文字の列　c文字

c文字::＝

シングルクォート’，円記号¥および¥n

を除くマシンのソース文字セット中の任意の文字

| エスケープ・シーケンス

エスケープ・シーケンス::＝

単一エスケープ・シーケンス

| ８進エスケープ・シーケンス

| １６進エスケープ・シーケンス

単一エスケープ・シーケンス::＝　以下のうちのいずれか１つ

¥'　¥"　¥?　¥¥

¥a　¥b　¥f　¥n　¥r　¥t　¥v

８進エスケープ・シーケンス::＝

¥　８進数字

| ¥　８進数字　８進数字

| ¥　８進数字　８進数字　８進数字

１６進エスケープ・シーケンス::＝

¥x　１６進数字

| １６進エスケープ・シーケンス　１６進数字

B.1.4 文字列リテラル

文字列リテラル::＝

"「s文字の列」"

s文字の列::＝

s文字

| s文字の列　s文字

s文字::＝

ダブルクォート"，円記号¥および¥n

を除くマシンのソース文字セット中の任意の文字

| エスケープ・シーケンス

B.1.5 演算子

演算子::= 以下のうちのいずれか１つ

[ ] ( ) . ->

++ -- & * + - ~ ! sizeof

/ % << >> < > <= >= == !=

^ | && ||

? ::=

= *= /= %= += -= <<= >>=

&= ^= |=

, # ##

B.1.6 区切り子

区切り子::= 以下のうちのいずれか１つ

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

B.1.7 ヘッダ名

ヘッダ名::=

<h文字の列>

| "q文字の列"

h文字の列::=

h文字

| h文字の列 h文字

h文字::=

改行文字および>を除くマシンのソース文字セット中の任意の文字

q文字の列::=

q文字

| q文字の列 q文字

q文字::=

改行文字および"を除くマシンのソース文字セット中の任意の文字

B.1.8 前処理数

前処理数::=

　　数字

| . 数字

| 前処理数 数字

| 前処理数 非数字

| 前処理数 e 符号

| 前処理数 E 符号

| 前処理数 .

B.2 式

B.2.1 一次式

一次式::=

　　識別子

| 定数

| 文字列リテラル

| （ 式 ）

B.2.2 後置式

後置式::=

　　一次式

| 後置式 ［ 添字式 ］

| 後置式 （ 「引数式リスト」 ）

| 後置式 . 識別子

| 後置式 －＞ 識別子

| 後置式 ＋＋

| 後置式 －－

引数式リスト::=

代入式

| 引数式リスト , 代入式

B.2.3 単項式

単項式::=

後置式

| ++ 単項式

| -- 単項式

| 単項演算子 キャスト式

| sizeof 単項式

| sizeof (型名)

単項演算子::= 以下のうちのいずれか1つ

& * + - ~ !

B.2.4 キャスト式

キャスト式::=

単項式

| (型名) キャスト式

B.2.5 乗法式

乗法式::=

キャスト式

| 乗法式 * キャスト式

| 乗法式 ／ キャスト式

| 乗法式 % キャスト式

B.2.6 加法式

加法式::=

乗法式

| 加法式 ＋ 乗法式

| 加法式 － 乗法式

B.2.7 シフト式

シフト式::=

加法式

| シフト式 ＜＜ 加法式

| シフト式 ＞＞ 加法式

B.2.8 関係式

関係式::=

シフト式

| 関係式 ＜ シフト式

| 関係式 ＞ シフト式

| 関係式 ＜＝ シフト式

| 関係式 ＞＝ シフト式

B.2.9 等値式

等値式::=

関係式

| 等値式 ＝＝ 関係式

| 等値式 ！＝ 関係式

B.2.10 （ビットごとの）ＡＮＤ式

ＡＮＤ式::＝

　　等値式

　| ＡＮＤ式 ＆ 等値式

B.2.11 （ビットごとの）排他的ＯＲ式

排他的ＯＲ式::＝

　　ＡＮＤ式

　| 排他的ＯＲ式 ^ ＡＮＤ式

B.2.12 （ビットごとの）ＯＲ式

ＯＲ式::＝

　　排他的ＯＲ式

　| ＯＲ式 | 排他的ＯＲ式

B.2.13 論理ＡＮＤ式

論理ＡＮＤ式::＝

　　ＯＲ式

　| 論理ＡＮＤ式 ＆＆ ＯＲ式

B.2.14 論理ＯＲ式

論理ＯＲ式::＝

　　論理ＡＮＤ式

　| 論理ＯＲ式 || 論理ＡＮＤ式

B.2.15 条件式

条件式::＝

　　論理ＯＲ式

　| 論理ＯＲ式 ？ 式 ： 条件式

B.2.16 代入式

代入式::＝

　　条件式

　| 単項式 代入演算子 代入式

代入演算子::＝ 以下のうちのいずれか１つ

　　＝ *＝ ／＝ ％＝ ＋＝ －＝ ＜＜＝ ＞＞＝

　　＆＝ ^＝ |＝

B.2.17 式（コンマ演算子）

式::＝

　　代入式

　| 式 ， 代入式

B.3 定数式

定数式::＝

　　条件式

B.4 宣言

宣言::=

　　宣言指定子　「初期宣言子リスト」　;

宣言指定子::=

　　記憶クラス指定子　「宣言指定子」

　| 型指定子　「宣言指定子」

　| 型修飾子　「宣言指定子」

初期宣言子リスト::=

　　初期宣言子

　| 初期宣言子リスト　,　初期宣言子

初期宣言子::=

　　宣言子

　| 宣言子　=　初期化子

B.4.1 記憶クラス指定子

記憶クラス指定子::=

　　typedef

　| extern

　| static

　| auto

　| register

B.4.2 型指定子

型指定子::=
　　void
　| char
　| short
　| int
　| long
　| signed
　| unsigned
　| 構造体／共用体指定子
　| 列挙指定子
　| typedef名

構造体／共用体指定子::=
　　構造体／共用体　「識別子」　{　構造体宣言リスト　}
　| 構造体／共用体　識別子

構造体／共用体::=
　　struct
　| union

構造体宣言リスト::=
　　構造体宣言
　| 構造体宣言リスト　構造体宣言

構造体宣言::=
　　指定子／修飾子リスト　構造体宣言子リスト　;

指定子／修飾子リスト::=
　　型指定子　「指定子／修飾子リスト」
　| 型修飾子　「指定子／修飾子リスト」

構造体宣言子リスト::=
　　構造体宣言子
　| 構造体宣言子リスト , 構造体宣言子

構造体宣言子::=
　　宣言子
　| 「宣言子」 : 定数式

列挙指定子::=
　　enum 「識別子」 { 列挙子リスト }
　| enum 識別子

列挙子リスト::=
　　列挙子
　| 列挙子リスト , 列挙子

列挙子::=
　　列挙定数
　　列挙定数 = 定数式

B.4.3 型修飾子

型修飾子::=
　　const
　| volatile

B.4.4 宣言子

宣言子::=
　　「ポインタ」 直接宣言子

ポインタ::=

　　＊　「型修飾子リスト」

　|　＊　「型修飾子リスト」　ポインタ

型修飾子リスト::=

　　型修飾子

　|　型修飾子リスト　型修飾子

直接宣言子::=

　　識別子

　|　（　宣言子　）

　|　直接宣言子　［　「定数式」　］

　|　直接宣言子　［　パラメータ型リスト　］

　|　直接宣言子　［　「識別子リスト」　］

パラメータ型リスト::=

　　パラメータリスト

　|　パラメータリスト　，　．．．

パラメータリスト::=

　　パラメータ宣言

　|　パラメータリスト　，　パラメータ宣言

パラメータ宣言::=

　　宣言指示子　宣言子

　|　宣言指示子　「抽象宣言子」

識別子リスト::=

　　識別子

　|　識別子リスト　，　識別子

B.4.5 型名

型名::=
　　指定子修飾子リスト　「抽象宣言子」

抽象宣言子::=
　　ポインタ
　　| 「ポインタ」 直接抽象宣言子

直接抽象宣言子::=
　　(抽象宣言子)
　　| 「直接抽象宣言子」 [「定数式」]
　　| 「直接抽象宣言子」 [「パラメータ型リスト」]

B.4.6 typedef名

typedef名::=
　　識別子

B.4.7 初期化子

初期化子::=
　　代入式
　　| { 初期化子リスト }
　　| { 初期化子リスト , }

初期化子リスト::=
　　初期化子
　　初期化子リスト , 初期化子

B.5 文

文::=

ラベル付き文
| 複文
| 式文
| 選択文
| 繰り返し文
| ジャンプ文

B.5.1 ラベル付き文

ラベル付き文::=

識別子 : 文
| case 定数式 : 文
| default : 文

B.5.2 複文(ブロック)

複文::=

{ 「宣言リスト」 「文リスト」 }

宣言リスト::=

宣言
| 宣言リスト 宣言

文リスト::=

文
| 文リスト 文

B.5.3 式文

式文::=

「式」 ;

B.5.4 選択文

選択文::=

```
if ( 式 ) 文
| if ( 式 ) 文 else 文
| switch ( 式 ) 文
```

B.5.5 繰り返し文

「繰り返し文」::=

```
while ( 式 ) 文
| do 文 while ( 式 ) ;
| for ( 「式」 ; 「式」 ; 「式」 ) 文
```

B.5.6 ジャンプ文

ジャンプ文::=

```
goto 識別子 ;
| continue ;
| break ;
| return 「式」 ;
```

B.6 外部定義

コンパイル単位::=

```
外部宣言
| コンパイル単位 外部参照
```

外部宣言::=

```
関数定義
| 宣言
```

B.6.1 関数定義

関数定義::=

「宣言指定子」　宣言子　「宣言リスト」　複文

B.7 前処理指令

前処理ファイル::=

「グループ」

グループ::=

グループ部分

| グループ　グループ部分

グループ部分::=

「前処理トークン列」　改行

| if節

| 制御行

if節::=

ifグループ　「elifグループの集まり」　「elseグループ」

endif行

ifグループ::=

if　定数式　改行　「グループ」

| # ifdef　識別子　改行　「グループ」

| # ifndef　識別子　改行　「グループ」

elifグループの集まり::=

elifグループ

| elifグループの集まり　elifグループ

elifグループ::=

elif　定数式　改行　「グループ」

elseグループ::=

　　#　elif　　改行　「グループ」

endif行::=

　　#　endif　　改行

制御行::=

　　#　include　前処理トークン列　改行

　|　#　define　識別子　置き換えリスト　改行

　|　#　define　識別子　左かっこ　「識別子リスト」　）
　　置き換えリスト　改行

　|　#　undef　　識別子　改行

　|　#　line　　前処理トークン列　改行

　|　#　error　　「前処理トークン列」　改行

　|　#　pragma　「前処理トークン列」　改行

　|　#　　改行

左かっこ::=

　　直前に空白がない（

置き換えリスト::=

　　「前処理トークン列」

前処理トークン列::=

　　前処理トークン

　|　前処理トークン列　前処理トークン

改行::=

　　改行文字

保守／廃止

付録C　ライブラリ関数

C.1　ライブラリ関数機能別一覧

C.1.1　入出力関数

関　数　名	機　　能	ページ
sprintf	フォーマットに従ってデータを文字列に書く	166
sscanf	入力文字列からフォーマットに従ってデータを読む	170

C.1.2　文字・文字列関数

（1／2）

関　数　名	機　　能	ページ
isalpha	英文字（A～Z，a～z）の判定	174
isupper	英文字（A～Z）の判定	
islower	英文字（a～z）の判定	
isdigit	数字（0～9）の判定	
isalnum	英数字（0～9，A～Z，a～z）の判定	
isxdigit	16進数字（0～9，A～F，a～f）の判定	
isspace	空白，タブ，復帰，改行，垂直タブ，改ページの判定	
ispunct	空白文字と英数字以外の表示可能文字の判定	
isprint	表示可能文字の判定	
isgraph	空白文字以外の表示可能文字の判定	
iscntrl	コントロール文字の判定	
isascii	ＡＳＣＩＩ文字の判定	
toupper	英小文字を英大文字に変換する	175
tolower	英大文字を英小文字に変換する	
_toupper	cから‘a’を引き‘A’を加える	176
_tolower	cから‘A’を引き‘a’を加える	
toascii	ＡＳＣＩＩコードへの変換を行う	177
strlen	文字列の長さを求める	178

関　数　名	機　　能	ページ
strcpy	文字列をコピーする	179
strncpy	指定文字列以外をコピーする	
strcat	文字列に文字列を連結する	180
strncat	文字列に指定文字数以内の文字列を連結する	
strcmp	２つの文字列を比較する	181
strncmp	２つの文字列の指定文字数以内を比較する	
strchr	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す	183
strrchr	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す	
strpbrk	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る	184
strspn	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る	185
strcspn	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る	
strstr	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る	186
strtok	文字列を区切り文字以外からなる文字列に分解する	187
strtol	文字列をlongに変換する	188
strtoul	文字列をunsigned longに変換する	
atoi	１０進整列文字列をintに変換する	191
atol	１０進整列文字列をlongに変換する	
itoa	intを文字列に変換する	193
ltoa	longを文字列に変換する	
ultoa	unsigned longを文字列に変換する	

保守／廃止

C.1.3 メモリ関数

関数名	機能	ページ
malloc	ブロックを割り付ける	194
calloc	配列の領域を割り付けて0で初期化する	195
realloc	ブロックの再割り付けを行う	196
free	割り付けられているブロックを解放する	197
brk	ブレーク値をセットする	198
sbrk	ブレーク値を増減する	
memcpy	バッファを指定文字数分コピーする	199
memmove	バッファを指定文字数分コピーする（バッファが重なっても正常に動作する）	
memcmp	２つのバッファの指定文字数分を比較する	200
memchr	指定文字数分のバッファから指定文字を探す	201
memset	バッファの指定文字数分を指定文字で初期化する	202

C.1.4 プログラム制御関数

関数名	機能	ページ
setjmp	呼び出し時の環境をセーブする	203
longjmp	setjmpでセーブされた環境を回復する	
about	プログラムを異常終了させる	204
atexit	正常終了時に呼び出される関数を登録する	205
exit	プログラムを終了させる	

C.1.5 数学関数

関数名	機能	ページ
abs	int型の値の絶対値を求める	206
labs	long型の値の絶対値を求める	
rand	疑似乱数を発生させる	207
srand	疑似乱数の発生状態の初期化を行う	
div	int型の除算を行い商と剰余を求める	208
ldiv	long型の除算を行い商と剰余を求める	

C.1.6 特殊関数

関数名	機能	ページ
qsort	クイック・ソートを行う	210
bsearch	バイナリ・サーチを行う	211
strerror	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す	212
va_start	可変個の引数の処理のための設定を行う　（マクロ）	213
va_arg	可変個の引数の処理を行う　（マクロ）	
va_end	可変個の引数の処理の終了を知らせる　（マクロ）	

保守／廃止

C.2 ライブラリ関数一覧（アルファベット順）

保守／廃止

付録D　saddr領域のラベル一覧

７８Ｋシリーズでは，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。

＜７８Ｋ／０の場合＞

(1) ディバグ用予約領域

ラベル名	アドレス
_@D_FUNC	0FE8CH
_@D_LINE	0FE8EH

(2) レジスタ変数

ラベル名	アドレス
_@KREG00	0FEA0H
_@KREG01	0FEA1H
_@KREG02	0FEA2H
_@KREG03	0FEA3H
_@KREG04	0FEA4H
_@KREG05	0FEA5H
_@KREG06	0FEA6H
_@KREG07	0FEA7H
_@KREG08	0FEA8H
_@KREG09	0FEA9H
_@KREG10	0FEAAH
_@KREG11	0FEABH
_@KREG12	0FEACH注１
_@KREG13	0FEADH注１
_@KREG14	0FEAEH注１
_@KREG15	0FEAFH注１

注１ 関数の引数がregister宣言されていて，かつ－ｑｒオプションが指定された場合に引数をsaddr領域に割り当てます。

(3) norec関数の引数

ラベル名	アドレス
_@NRARG0	0FEB0H
_@NRARG1	0FEB2H
_@NRARG2	0FEB4H
_@NRARG3	0FEB6H
_@NRARG4	0FEB8H
_@NRARG5	0FEBAH
_@NRARG6	0FEBCH
_@NRARG7	0FEBEH

(4) norec関数のオートマティック変数（１／２）

ラベル名	アドレス
_@NRAT00	0FEC0H
_@NRAT01	0FEC1H
_@NRAT02	0FEC2H
_@NRAT03	0FEC3H
_@NRAT04	0FEC4H
_@NRAT05	0FEC5H
_@NRAT06	0FEC6H
_@NRAT07	0FEC7H
_@NRAT08	0FEC8H
_@NRAT09	0FEC9H
_@NRAT10	0FECAH
_@NRAT11	0FECBH
_@NRAT12	0FECCH

(4) norec関数のオートマティック変数
（２／２）

ラベル名	アドレス
_@NRAT13	0FECDH
_@NRAT14	0FECEH
_@NRAT15	0FECFH
_@NRAT16	0FED0H
_@NRAT17	0FED1H
_@NRAT18	0FED2H
_@NRAT19	0FED3H
_@NRAT20	0FED4H
_@NRAT21	0FED5H
_@NRAT22	0FED6H
_@NRAT23	0FED7H
_@NRAT24	0FED8H
_@NRAT25	0FED9H
_@NRAT26	0FEDAH
_@NRAT27	0FEDBH
_@NRAT28	0FEDCH
_@NRAT29	0FEDDH
_@NRAT30	0FEDEH
_@NRAT31	0FEDFH

(5) ランタイム・ライブラリの引数

ラベル名	アドレス
_@RTARG0	0FE98H
_@RTARG1	0FE99H
_@RTARG2	0FE9AH
_@RTARG3	0FE9BH
_@RTARG4	0FE9CH
_@RTARG5	0FE9DH
_@RTARG6	0FE9EH
_@RTARG7	0FE9FH

保守／廃止

＜７８Ｋ／Ⅱの場合＞

(1) ディバグ用予約領域

ラベル名	アドレス
_@D_FUNC	0FE7CH
_@D_LINE	0FE7EH

(2) レジスタ変数

ラベル名	アドレス
_@KREG00	0FE90H
_@KREG01	0FE91H
_@KREG02	0FE92H
_@KREG03	0FE93H
_@KREG04	0FE94H
_@KREG05	0FE95H
_@KREG06	0FE96H
_@KREG07	0FE97H
_@KREG08	0FE98H
_@KREG09	0FE99H
_@KREG10	0FE9AH
_@KREG11	0FE9BH
_@KREG12	0FE9CH注１
_@KREG13	0FE9DH注１
_@KREG14	0FE9EH注１
_@KREG15	0FE9FH注１

(3) norec関数の引数

ラベル名	アドレス
_@NRARG0	0FEA0H
_@NRARG1	0FEA2H
_@NRARG2	0FEA4H
_@NRARG3	0FEA6H
_@NRARG4	0FEA8H
_@NRARG5	0FEAAH

(4) norec関数のオートマティック変数

ラベル名	アドレス
_@NRAT00	0FEACH
_@NRAT01	0FEADH
_@NRAT02	0FEAEH
_@NRAT03	0FEAFH
_@NRAT04	0FEB0H
_@NRAT05	0FEB1H
_@NRAT06	0FEB2H
_@NRAT07	0FEB3H
_@NRAT08	0FEB4H
_@NRAT09	0FEB5H
_@NRAT10	0FEB6H
_@NRAT11	0FEB7H
_@NRAT12	0FEB8H
_@NRAT13	0FEB9H
_@NRAT14	0FEBAH
_@NRAT15	0FEBBH
_@NRAT16	0FEBCH
_@NRAT17	0FEBDH
_@NRAT18	0FEBEH
_@NRAT19	0FEBFH
_@NRAT20	0FEC0H
_@NRAT21	0FEC1H

注１ 関数の引数がregister宣言されていて，
かつ－ｑｒオプションが指定された場合
に引数をsaddr領域に割り当てます。

(5) ランタイム・ライブラリの引数

ラベル名	アドレス
_@RTARG0	0FE88H
_@RTARG1	0FE89H
_@RTARG2	0FE8AH
_@RTARG3	0FE8BH
_@RTARG4	0FE8CH
_@RTARG5	0FE8DH
_@RTARG6	0FE8EH
_@RTARG7	0FE8FH

保守／廃止

＜７８Ｋ／Ⅲの場合＞

(1) ディバグ用予約領域

ラベル名	アドレス
_@D_FUNC	0FE7CH
_@D_LINE	0FE7EH

(2) レジスタ変数

ラベル名	アドレス
_@KREG00	0FE90H
_@KREG01	0FE91H
_@KREG02	0FE92H
_@KREG03	0FE93H
_@KREG04	0FE94H
_@KREG05	0FE95H
_@KREG06	0FE96H
_@KREG07	0FE97H
_@KREG08	0FE98H
_@KREG09	0FE99H
_@KREG10	0FE9AH
_@KREG11	0FE9BH
_@KREG12	0FE9CH
_@KREG13	0FE9DH
_@KREG14	0FE9EH
_@KREG15	0FE9FH

(3) norec関数の引数

ラベル名	アドレス
_@NRARG0	0FEA0H
_@NRARG1	0FEA2H
_@NRARG2	0FEA4H
_@NRARG3	0FEA6H
_@NRARG4	0FEA8H
_@NRARG5	0FEAAH
_@NRARG6	0FEACH
_@NRARG7	0FEAEH

(4) norec関数のオートマティック変数 (１／２)

ラベル名	アドレス
_@NRAT00	0FEB0H
_@NRAT01	0FEB1H
_@NRAT02	0FEB2H
_@NRAT03	0FEB3H
_@NRAT04	0FEB4H
_@NRAT05	0FEB5H
_@NRAT06	0FEB6H
_@NRAT07	0FEB7H
_@NRAT08	0FEB8H
_@NRAT09	0FEB9H
_@NRAT10	0FEBAH
_@NRAT11	0FEBBH
_@NRAT12	0FEBCH
_@NRAT13	0FEBDH
_@NRAT14	0FEBEH
_@NRAT15	0FEBFH
_@NRAT16	0FEC0H
_@NRAT17	0FEC1H
_@NRAT18	0FEC2H

(4) norec関数のオートマティック変数
（2／2）

ラベル名	アドレス
_@NRAT19	0FEC3H
_@NRAT20	0FEC4H
_@NRAT21	0FEC5H
_@NRAT22	0FEC6H
_@NRAT23	0FEC7H
_@NRAT24	0FEC8H
_@NRAT25	0FEC9H
_@NRAT26	0FECAH
_@NRAT27	0FECBH
_@NRAT28	0FECCH
_@NRAT29	0FECDH
_@NRAT30	0FECEH
_@NRAT31	0FECFH

保守／廃止

付録E　割り込み要求名一覧

＜７８Ｋ／０の場合＞

アドレス	001/002	011/012/013/014/P014	022/023/024/P024	042/043/044/P044
0000H	RST	RST	RST	RST
0004H	INTWDT	INTWDT	INTWDT	INTWDT
0006H	INTP0	INTP0	INTP0	INTP0
0008H	INTP1	INTP1	INTP1	INTP1
000AH	INTP2	INTP2	INTP2	INTP2
000CH	INTP3	INTP3	INTP3/INTUD	INTP3/INTUD
000EH	INTCSI0	INTCSI0	INTCSI0	INTCSI0
0010H	INTCSI1	-	INTCSI1	INTCSI1
0012H	INTTM3	INTTM3	INTTM3	INTTM3
0014H	INTTM0	-	INTTM0	INTTM0
0016H	INTTM1	INTTM1	INTTM1	INTTM1
0018H	INTTM2	INTTM2	INTTM2	INTTM2
001AH	INTAD	-	-	INTAD
001CH	-	- 注	INTKS	INTKS
003EH	BRK_ I	BRK_ I	BRK_ I	BRK_ I

注　μＰＤ７８０１ｘの製品としては内蔵していません。

エミュレーションの関係上μＰＤ７８０１４にのみハードウエアを内蔵します。

保守／廃止

＜７８Ｋ／Ⅱの場合＞

アドレス	210 212/213/214 217A/218A 233/234 237/238	220/224	243/244
00000H	RST	RST	RST
00002H	NMI	NMI	NMI
00006H	INTP0	INTP0	INTP0
00008H	INTP1	INTP1	INTP1
0000AH	INTP2	INTP2	INTP2
0000CH	INTP3	INTP3	INTP3
0000EH	INTP4/INTC30	INTP4	INTP4/INTC30
00010H	INTP5/INTAD	INTP5	INTP5/INTAD
00012H	INTC20	INTP6	INTC20
00014H	INTC00	INTC00	INTC00
00016H	INTC01	INTC01	INTC01
00018H	INTC10	INTC10	INTC10
0001AH	INTC11	INTC11	INTC11
0001CH	INTC21	INTC21	INTC21
00020H	INTSER	INTSER	INTSER
00022H	INTSR	INTSR	INTSR
00024H	INTST	INTST	INTST
00026H	INTCSI	INTCSI	INTCSI
00028H	－	－	INTEER
0002AH	－	－	INTEPW
0003EH	BRK_ I	BRK_ I	BRK_ I

保守／廃止

＜７８Ｋ／Ⅲの場合＞

アドレス		310/312	320/322	327/328	330/334	350/352
TPF=0	TPF=1	310A/312A	323/324			
0000H	0000H	RST	RST	RST	RST	RST
0002H	8002H	NMI	NMI	NMI	NMI	NMI
0004H	8004H	INTE0	INTWDT	INTWDT	INTWDT	INTWDT
0006H	8006H	INTE1	INTOV	INTOV0	INTOV	INTOV
0008H	8008H	INTE2	INTP0	INTP0	INTP0	INTP0
000AH	800AH	INTWDT	INTP1	INTP1	INTP1	INTP1
000CH	800CH	INTTB	INTP2	INTP2	INTP2	INTCM10
000EH	800EH	INTTM0	INTP3	INTOV1	INTP3/INTCC00R	INTCM20
0010H	8010H	INTTM1	INTP4/INTCCX0	INTCM00	INTP4/INTCC01R	INTP2
0012H	8012H	INTTM2	INTP5/INTCC01	INTCM01	INTP5	INTP3
0014H	8014H	-	INTP6	INTCM02	INTP6	-
0016H	8016H	-	INTCM00	INTCM03	INTCMX0	-
0018H	8018H	-	INTCM01	INTCM04	INTCM11	-
001AH	801AH	INTCR00	INTCM02	INTCM05	INTCM12	-
001CH	801CH	INTCR01	INTCM03	INTCM06	INTCM20	-
001EH	801EH	INTCR10	INTCM10	INTCC10	INTCM21	-
0020H	8020H	INTCR11	INTCM11	INTCM20	INTCM30	-
0022H	8022H	INTSER	INTSER	INTSER	INTSER	-
0024H	8024H	INTSR	INTSR	INTSR	INTSR	-
0026H	8026H	INTST	INTST	INTST	INTST	-
0028H	8028H	INTAD	INTCSI	INTCSI	INTCSI	-
002AH	802AH	-	INTAD	INTAD	INTAD	-
003CH	003CH	-	TRAP0	TRAP0	TRAP0	TRAP0
003EH	003EH	BRK_ I	BRK_ I	BRK_ I	BRK_ I	BRK_ I

付録F　セグメント名

コンパイラが出力する全セグメントの説明をします。

(1) Cソース・モジュール例（SAMPLE.C）

```
#pragma VECT NMI inter          /* 割り込み関数 */
void main(void);                /* 関数プロトタイプ */
const int i_cnst = 1;           /* const変数 */
callf int f_clf(void);          /* callf関数プロトタイプ */
callf int f_clt(void);          /* callt関数プロトタイプ */
bit b_bit;                      /* bit型変数 */
long l_intt = 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;
}

callt int f_clt()               /* callt関数定義 */
{
}

callf int f_clf()               /* callf関数定義 */
{
}

void inter()                    /* 割り込み関数定義 */
{
}
```

(2) アセンブラ・ソース・モジュール例

アセンブラ・ソース中の疑似命令，命令セットは各チップにより異なります。詳細は「ＣＣ７８Ｋシリーズ　Ｃコンパイラ　ユーザーズ・マニュアル　操作編」をご覧ください。

①ＲＯＭ化指定時＜７８Ｋ／Ⅲの場合＞

```
; 78K/III C Compiler V1.00 Assembler Source Date:XX XXX XXXX Time:XX:XX:XX

; Command   : -c310 -r -sasample1.asm sample.c
; In-file   : SAMPLE.C
; Asm-file  : SAMPLE1.ASM
; Para-file :

$PROCESSOR(310)
$NODEBUG

        NAME    SAMPLE
        PUBLIC  _main
        PUBLIC  _i_cnst
        PUBLIC  ?f_clt
        PUBLIC  _f_clt
        PUBLIC  _f_clf
        PUBLIC  _b_bit
        PUBLIC  _l_init
        PUBLIC  _i_data
        PUBLIC  _sr_inis
        PUBLIC  _sr_dats

BITS    BSEG                    bit型変数用セグメント
_b_bit  DBIT

CODE    CSEG                    コード部用セグメント
_main:
        push    hl
        movw    ax,sp
        subw    ax,#02H
        movw    hl,ax
        movw    sp,ax
        movw    ax,#064H        ;100
        mov     [hl+1],a        ;i
        xch     a,x
        mov     [hl],a          ;i
        movw    ax,hl
        addw    ax,#02H
        movw    sp,ax
        pop     hl
        ret
```

```
@inter:                                 割り込み関数
        saddr空間の退避
        push    ax,bc,rp3,rp4,vp,up,de,hl       ;全レジスタの退避
        関数本体
        push    ax,bc,rp3,rp4,vp,up,de,hl       ;全レジスタの退避
        saddr空間の退避
        reti

_f_clt:                                 callt関数
        ret

CALF    CSEG    FIXED                   callf関数用セグメント
_f_clf:
        ret

CALT    CSEG    CALLT0                  callt関数用セグメント
?f_clt: DW      _f_clt

CNST    CSEG                            const変数用セグメント
_i_cnst:        DW      01H     ;1

R_INIT  CSEG                            初期化データ用セグメント（初期値あり）
        DW      00002H,00000H   ;2

R_DATA  CSEG                            初期化データ用セグメント（初期値なし）
        DB      (2)

INIT    CSEG                            仮データ領域用セグメント（初期値あり）
_l_init:
        DS      (4)

DATA    CSEG                            仮データ領域用セグメント（初期値なし）
_i_data:
        DS      (2)

R_INIS  CSEG                            初期化データ用セグメント（初期値ありsreg変数）
        DW      03H     ;3

R_DATS  CSEG                            初期化データ用セグメント（初期値なしsreg変数）
        DB      (2)

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)
        END
```

②ＲＯＭ化指定時＜７８Ｋ／Ⅲの場合＞

```
; 78K/III C Compiler V1.00 Assembler Source Date:XX XXX XXXX Time:XX:XX:XX

; Command   : -c310 -nr -sasample2.asm sample.c
; In-file   : SAMPLE.C
; Asm-file  : SAMPLE2.ASM
; Para-file :

$PROCESSOR(310)
$NODEBUG

        NAME    SAMPLE
        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  _f_clf

BITS    BSEG                    bit型変数用セグメント
_b_bit  DBIT

CODE    CSEG                    コード部用セグメント
_main:
        push    hl
        movw    ax,sp
        subw    ax,#02H
        movw    hl,ax
        movw    sp,ax
        movw    ax,#064H        ;100
        mov     [hl+1],a        ;i
        xch     a,x
        mov     [hl],a          ;i
        movw    ax,hl
        addw    ax,#02H
        movw    sp,ax
        pop     hl
        ret
```

```
@inter:                                 割り込み関数
        saddr空間の退避
        push    ax,bc,rp3,rp4,vp,up,de,hl       ;全レジスタの退避
        関数本体
        push    ax,bc,rp3,rp4,vp,up,de,hl       ;全レジスタの退避
        saddr空間の退避
        reti

_f_clt:                                 callt関数
        ret

CALF    CSEG    FIXED                   callf関数用セグメント
_f_clf:
        ret

CALT    CSEG    CALLT0                  callt関数用セグメント
?f_clt: DW      _f_clt

CNST    CSEG                            const変数用セグメント
_i_cnst:        DW      01H     ;1

INIT    DSEG                            初期化データ用セグメント（初期値あり）
_i_init:        DW      00002H,00000H   ;2

DATA    DSEG                            初期化データ用セグメント（初期値なし）
_i_data:
        DB      (2)

INIS    DSEG    SADDRP                  初期化データ用セグメント（初期値ありsreg変数）
_sr_inis:        DW     03H     ;3

DATS    DSEG    SADDRP                  初期化データ用セグメント（初期値なしsreg変数）
_sr_dats:
        DB      (2)
        END
```

保守／廃止

索　引

保守／廃止

保守／廃止

アンケート記入のお願い

お手数ですが、このドキュメントに対するご意見をお寄せください。今後のドキュメント作成の参考にさせていただきます。

［ドキュメント名］　ＣＣ７８Ｋシリーズ　Ｃコンパイラ　言語編　ユーザーズ・マニュアル

（　EEU-655C(第4版)，July 1991 P　）

［お名前など］　（さしつかえのない範囲で）

御社名（学校名、その他）　（　　）
ご住所　（　　）
お電話番号　（　　）
お仕事の内容　（　　）
お名前　（　　）

１．ご評価（各欄に∨をご記入ください）

項　　目	大変良い	良い	普通	悪い	大変悪い
全体の構成					
説明内容					
用語解説					
調べやすさ					
デザイン、字の大きさなど					
そ　の　他（　　）					
（　　）					

２．わかりやすい所（第　章、第　章、第　章、第　章、その他　　）

理由［　　］

３．わかりにくい所（第　章、第　章、第　章、第　章、その他　　）

理由［　　］

４．ご意見、ご要望

５．このドキュメントをお届けしたのは

ＮＥＣ販売員、特約店販売員、ＮＥＣ半応技本部員、その他（　　）

ご協力ありがとうございました。
下記あてにＦＡＸで送信いただくか、最寄りの販売員にコピーをお渡しください。

ＮＥＣ半導体応用技術本部インフォメーションセンター
ＦＡＸ：（０４４）５４８－７９００（直通ＦＡＸでの２４時間受付）

キリトリ

保守／廃止

保守／廃止

NEC 日本電気株式会社

本社	〒108-01 東京都港区芝五丁目7番1号(日本電気本社ビル)	
半導体第一、第二販売事業部	〒108-01 東京都港区芝五丁目7番1号(日本電気本社ビル)	東京(03)3454−1111
関西支社半導体販売部	〒540 大阪市中央区城見一丁目4番24号(日本電気関西ビル)	大阪(06)945−3178 大阪(06)945−3200
中部支社半導体販売部	〒460 名古屋市中区栄四丁目14番5号(松下中日ビル)	名古屋(052)242−2755

北海道支社	札幌(011)231−0161	立川支社	立川(0425)26−0911
東北支社	仙台(022)261−5511	千葉支社	千葉(0472)27−5441
岩手支店	盛岡(0196)51−4344	静岡支社	静岡(054)255−2211
山形支店	山形(0236)23−5511	沼津支店	沼津(0559)63−4455
郡山支店	郡山(0249)23−5511	浜松支店	浜松(0534)52−2711
いわき支店	いわき(0246)21−5511	北陸支社	金沢(0762)23−1621
長岡支店	長岡(0258)36−2155	福井支店	福井(0776)22−1866
水戸支店	水戸(0292)26−1717	富山支店	富山(0764)31−8461
神奈川支社	横浜(045)324−5511	京都支社	京都(075)221−8511
群馬支店	高崎(0273)26−1255	神戸支社	神戸(078)332−3311
太田支店	太田(0276)46−4011	中国支社	広島(082)242−5504
宇都宮支店	宇都宮(0286)21−2281	鳥取支店	鳥取(0857)27−5311
小山支店	小山(0285)24−5011	岡山支店	岡山(0862)25−4455
長野支店	長野(0262)35−1444	四国支社	高松(0878)36−1200
松本支店	松本(0263)35−1666	新居浜支店	新居浜(0897)32−5001
上諏訪支店	諏訪(0266)53−5350	松山支店	松山(0899)45−4111
甲府支店	甲府(0552)24−4141	九州支社	福岡(092)271−7700
埼玉支社	大宮(048)641−1411	北九州支店	北九州(093)541−2887

(技術お問い合せ先)

半導体応用技術本部 第一応用システム技術部	〒108-01 東京都港区芝五丁目7番1号(日本電気本社ビル)	東京(03)3798−6105
半導体応用技術本部 第二応用システム技術部	〒540 大阪市中央区城見一丁目4番24号(日本電気関西ビル)	大阪(06)945−3383
半導体応用技術本部 専用マイクロコンピュータ技術部	〒210 川崎市川崎区駅前本町15番5号(十五番館)	川崎(044)246−3922

インフォメーションセンター
FAX(044)548-7900
(24時間受付)