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

RENESAS

ご注意書き

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

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

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

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

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

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

RENESAS

ユーザーズ・マニュアル

CC77016

μPD77016 ファミリ用 C コンパイラ

μPD77015
μPD77016
μPD77017
μPD77018A
μPD77019
μPD77110
μPD77111
μPD77112
μPD77113A
μPD77114
μPD77115
μPD77210
μPD77213

資料番号　U15037JJ3V0UM00（第 3 版）
発行年月　January　2003 NS　CP（K）

© NEC Electronics Corporation　2000

[メ　モ]

目　次　要　約

第 1 章　概　　説　…　17

第 2 章　インストール手順　…　19

第 3 章　C コンパイラ・ドライバ　…　25

第 4 章　チュートリアル　…　29

第 5 章　C コンパイラとオプション　…　38

第 6 章　言語仕様　…　44

第 7 章　C コンパイラ名称規則　…　68

第 8 章　スタートアップ・コードの構成　…　74

第 9 章　ブート・サポート　…　78

第 10 章　文字出力　…　80

第 11 章　挿入されたコード記述　…　81

第 12 章　シンボリック・ディバグについて　…　84

第 13 章　標準 C ライブラリの関数レファレンス　…　94

第 14 章　DSP ライブラリの関数レファレンス　…　157

Microsoft，Windows，Windows NT は，米国 Microsoft Corporation の米国およびその他の国における登録商標または商標です。
PC/AT は米国 IBM 社の商標です。

- 本資料に記載されている内容は2003年1月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品の使用に関連し発生した第三者の特許権，著作権その他の知的財産権の侵害等に関し，一切その責を負いません。当社は，本資料に基づき当社または第三者の特許権，著作権その他の知的財産権を何ら許諾するものではありません。
- 本資料に記載された回路，ソフトウエアおよびこれらに関する情報は，半導体製品の動作例，応用例を説明するものです。お客様の機器の設計において，回路，ソフトウエアおよびこれらに関する情報を使用する場合には，お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し，当社は，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。当社製品の不具合により生じた生命，身体および財産に対する損害の危険を最小限度にするために，冗長設計，延焼対策設計，誤動作防止設計等安全設計を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「特定水準」に分類しております。また，各品質水準は，以下に示す用途に製品が使われることを意図しておりますので，当社製品の品質水準をご確認ください。

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

  特別水準：輸送機器（自動車，電車，船舶等），交通用信号機器，防災・防犯装置，各種安全装置，生命維持を目的として設計されていない医療機器

  特定水準：航空機器，航空宇宙機器，海底中継機器，原子力制御システム，生命維持のための医療機器，生命維持のための装置またはシステム等

  当社製品のデータ・シート，データ・ブック等の資料で特に品質水準の表示がない場合は，標準水準製品であることを表します。意図されていない用途で当社製品の使用をお客様が希望する場合には，事前に当社販売窓口までお問い合わせください。

（注）

（1）本事項において使用されている「当社」とは，NECエレクトロニクス株式会社およびNECエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいう。

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

M8E　02.11

本版で改訂された主な箇所

箇所	内　容
p.17	**第1章　概説**，1.1　**特徴**の記述を変更。
p.18	1.2　**オーダ名称と対象OS，表1-1　表記法に**，Windows 2000を追加。
p.19	**第2章　インストール手順**の内容を変更。
p.25	**第3章　Cコンパイラ・ドライバ**を追加。
p.38	5.1　C**コンパイラを直接呼び出す方法**を追加。
p.38	5.2.3　-f *parameterfile* を追加。
p.39-41	5.4.6　-exactfp から 5.4.12　-loopdepth0 to -loopdepth4，5.4.19-noincr を追加。
p.49	6.4　**固定小数点データ型**に説明を追加。
p.59	**表6-6　ハードウエア・ループへのコンパイルを可能にする関数**にコンパイラ関数を追加。
p.60	6.7.1　**明示的ハードウエア・ループ制御**を追加。
p.62	6.8　**関数のインライン**を追加。
p.63	6.9　**割り込みサポート**を追加。
p.68	7.1.1　**関数パラメータの受け渡し**に説明を追加。
p.69	**表7-1　呼び出し規則のまとめ**にレジスタを追加。
p.71	7.4　**変数割り当てとセグメント名**を追加。
p.90	12.3.5（3）（c）C**コード（ローカル・シンボル）**のプログラムを修正。
p.160	14.4　get_eir，14.5　get_sr を追加。
p.180	14.34　set_eir，14.35　set_sr を追加。

本文欄外の★印は，本版で改訂された主な箇所を示しています。

は　じ　め　に

対 象 者　このマニュアルは，ディジタル・シグナル・プロセッサμ PD77016 ファミリおよびμ PD77116 の機能を理解し，それを用いたソフトウエア，ハードウエアなどのアプリケーション・システムを設計するユーザを対象とします。

μ PD77016 ファミリは，μ PD7701x ファミリ（μ PD77015, 77016, 77017, 77018A, 77019），μ PD77111 ファミリ（μ PD77110, 77111, 77112, 77113A, 77114, 77115），およびμ PD77210 ファミリ（μ PD77210, 77213）の総称です。特に機能面に違いがない場合は，それぞれのファミリを該当する製品に読み替えてご使用ください。機能面に違いがある場合は，製品名をあげて説明しています。

このマニュアルには，μ PD77116 に関する記述がありますが，本バージョンの CC77016 は，μ PD77116 には正式に対応しておりませんので，注意してください。

目　　的　CC77016 は，μ PD77016 ファミリの応用システムを設計，開発する際に，プログラムを効率よく作成するための開発ツールです。
このマニュアルは，CC77016 を用いて効率よくプログラムを開発していただくことを目的としています。

構　　成　このマニュアルでは，大きく分けて次の内容で構成しています。

●第 1 章　概　　説
●第 2 章　インストール手順
●第 3 章　C コンパイラ・ドライバ
●第 4 章　チュートリアル
●第 5 章　C コンパイラとオプション
●第 6 章　言語仕様
●第 7 章　C コンパイラ名称規則
●第 8 章　スタートアップ・コードの構成
●第 9 章　ブート・サポート
●第 10 章　文字出力
●第 11 章　挿入されたコード記述
●第 12 章　シンボリック・ディバグについて
●第 13 章　標準 C ライブラリの関数レファレンス
●第 14 章　DSP ライブラリの関数レファレンス

CC77016 には制限事項があります。詳細については，NESDIS 内 CC77016 製品情報の各種通知一覧をご覧ください。

読み方　このマニュアルを読むにあたっては，電気，論理回路，マイクロコンピュータに関する一般知識や，Microsoft™ Windows® 95, 98, 2000, Windows NT® 4.0 の操作に関する一般的知識が必要となります。

CC77016 の機能を理解しようとするとき
　→目次に従ってお読みください。

C ライブラリ関数を理解しようとするとき
　→**第 13 章　標準 C ライブラリの関数レファレンス**をお読みください。

DSP ライブラリ関数を理解しようとするとき
　→**第 14 章　DSP ライブラリの関数レファレンス**をお読みください。

凡　例

注	：本文中につけた注の説明
注意	：気をつけて読んでいただきたい内容
備考	：本文中の補足説明
数の表記	： 2 進数…××××または 0b×××× 10 進数…×××× 16 進数…0x××××

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

μPD77016 **ファミリに関する資料**

資料名／品名	パンフレット	データ・シート	ユーザーズ・マニュアル		アプリケーション・ノート	
			アーキテクチャ編	命令編	基本ソフトウエア編	ライブラリ編
μPD77016	ー	U10891J	U10503J	U13116J	U11958J	U12021J
μPD77015		U10902J				
μPD77017						
μPD77018A		U11849J				
μPD77019						
μPD77019-013		U13053J				
μPD77110	U12395J	U12801J	U14623J			
μPD77111						
μPD77112						
μPD77113A		U14373J				
μPD77114						
μPD77115		U14867J				
μPD77210		U15203J	U15807J			
μPD77213						

開発ツールに関する資料

資　料　名		資料番号
HSM77016　ユーザーズ・マニュアル		U11602J
WB77016　ユーザーズ・マニュアル	言語編	U10078J
	操作編	U11506J
ID77016　ユーザーズ・マニュアル		U10118J
CC77016　ユーザーズ・マニュアル		このマニュアル
RX77016　ユーザーズ・マニュアル	機能編	U14397J
	コンフィギュレーション・ツール編	U14404J
RX77016　アプリケーション・ノート	HOST API 編	U14371J

注意　**上記関連資料は，予告なしに内容を変更することがあります。設計などには，必ず最新の資料をご使用ください。**

目　　次

第 1 章　概　　説　…　17

1.1　特　　徴　…　17
1.2　オーダ名称と対象 OS　…　18
1.3　パッケージ内容　…　18
1.4　対象デバイス　…　18
1.5　表記法　…　…　18

第 2 章　インストール手順　…　19

2.1　アプリケーションの設定　…　19
2.1.1　アプリケーションの設定時の注意　…　19
2.2　ライセンス手続き　…　21
2.2.1　ライセンス手続き時の注意　…　21
2.3　ビルド・プロセス　…　22
2.3.1　SP77016(WB77016)+CC77016 によるビルド　…　22
2.3.2　SP77210+CC77016 によるビルド（コンパイラ･ドライバを用いる場合）　…　23
2.3.3　SP77210+CC77016 によるビルド（ADS 上でビルドする場合）　…　23

第 3 章　C コンパイラ・ドライバ　…　25

3.1　C コンパイラの呼び出し　…　25
3.2　グローバル・オプション　…　25
3.2.1　-?　…　25
3.2.2　-v　…　25
3.2.3　-n　…　26
3.2.4　-f *parameterfile*　…　26
3.2.5　-nologo　…　26
3.3　ファイル・オプション　…　26
3.3.1　-o *filename*　…　26
3.3.2　-s　…　26
3.3.3　-c　…　26
3.3.4　-save-temps　…　26
3.3.5　-nostdinc　…　26
3.3.6　-nostartupfile　…　26
3.3.7　-nostdlib　…　27
3.4　C コンパイラ・オプション　…　27
3.5　アセンブラ・オプション　…　27
3.5.1　-m *modelfile*　…　27
3.5.2　-WA *options [, options]*　…　27
3.5.3　-Wa *options [, options]*　…　27
3.6　リンカ・オプション　…　28
3.6.1　-m *modelfile*　…　28

3.6.2 -WL *options [, options]* … 28
3.6.3 -l *name* … 28
3.6.4 -L *path* … 28

第 4 章 チュートリアル … 29

第 5 章 C コンパイラとオプション … 38

5.1 C **コンパイラを直接呼び出す方法** … 38
5.2 **ファイル・オプション** … 38
5.2.1 -o *filename* … 38
5.2.2 -I *path* … 38
5.2.3 -f *parameterfile* … 38
5.2.4 -nologo … 38
5.3 **言語オプション** … 39
5.3.1 -nodsp … 39
5.3.2 -W … 39
5.3.3 -Wansi … 39
5.3.4 -Wstrict … 39
5.3.5 -Wall … 39
5.3.6 -D*macro* ［＝*value* ］ … 39
5.4 **コード生成オプション** … 39
5.4.1 -O0 … 39
5.4.2 -O1 … 39
5.4.3 -O2 … 39
5.4.4 -O3 … 39
5.4.5 -Osize … 39
5.4.6 -exactfp … 39
5.4.7 -inlinesize *statementLimit* … 40
5.4.8 -inline *functionName* … 40
5.4.9 -loopcallsok … 40
5.4.10 -loopcountok … 40
5.4.11 -loopcountnice … 40
5.4.12 -loopdepth0 to -loopdepth4 … 40
5.4.13 -noaar … 40
5.4.14 -nor5 … 40
5.4.15 -nor6 … 40
5.4.16 -nor7 … 41
5.4.17 -nodp2 … 41
5.4.18 -nodp6 … 41
8.4.19 -noincr … 41
5.4.20 -norlse … 41
5.5 **ディバグ・オプション** … 41
5.5.1 -g … 41
5.5.2 -gb … 41
5.5.3 -gw … 41
5.5.4 -gwg … 41
5.5.5 -gbw *functionName* … 41
5.5.6 -*i* … 42

5.5.7 *-ih* … 42
5.6 **メモリ・オプション** … 42
5.6.1 -defy … 42
5.6.2 -defser … 42
5.6.3 -defstackser … 42
5.6.4 -singlestack … 43
5.6.5 -extshared … 43
5.6.6 -extrom … 43
5.6.7 -introm … 43
5.6.8 -sepsegs … 43
5.6.9 -extimseg … 43
5.6.10 -boot … 43

第 6 章 言語仕様 … 44

6.1 **整数データ型** … 44
6.2 **アドレス演算** … 46
6.3 **宣言上のメモリ空間修飾子** … 47
6.4 **固定小数点データ型** … 48
6.5 **リング・バッファとモジュロ・アドレシング・モード** … 51
6.6 **ビット反転アドレシング・モード** … 54
6.7 **ループ処理** … 58
6.7.1 **明示的ハードウエア・ループ制御** … 60
6.8 **関数のインライン** … 62
6.9 **割り込みサポート** … 63
6.9.1 **割り込みプラグマ** … 64
6.9.2 **生成コード・セグメント** … 64
6.9.3 **レジスタの使用** … 65
6.9.4 **スタックの使用** … 66
6.9.5 **割り込み制御レジスタのアクセス** … 66

第 7 章 C コンパイラ名称規則 … 68

7.1 **呼び出し規則** … 68
7.1.1 **関数パラメータの受け渡し** … 68
7.1.2 **レジスタ・セーブ** … 69
7.2 **スタック・レイアウト** … 69
7.3 **グローバル・オブジェクト** … 70
7.4 **変数割り当てとセグメント名** … 71

第 8 章 スタートアップ・コードの構成 … 74

第 9 章 ブート・サポート … 78

第 10 章 文字出力 … 80

第 11 章 挿入されたコード記述 … 81

第 12 章　シンボリック・ディバグについて　…　84

12.1　**ディバグ命令**　…　84
12.2　**名前の矛盾**　…　85
12.3　**アセンブリ命令説明**　…　85
12.3.1　.file　…　85
12.3.2　.line　…　86
12.3.3　.func，.efunc　…　86
12.3.4　.prolog_size　…　87
12.3.5　.sym　…　87
12.3.6　.etag，.stag，.utag，.eos　…　92
12.3.7　.member　…　93

第 13 章　標準 C ライブラリの関数レファレンス　…　94

13.1　abort　…　94
13.2　abs　…　94
13.3　assert　…　95
13.4　atexit　…　96
13.5　atoi　…　96
13.6　atol　…　97
13.7　bsearch　…　98
13.8　calloc　…　99
13.9　div　…　100
13.10　fprintf　…　101
13.11　fputc　…　102
13.12　fputs　…　103
13.13　free　…　103
13.14　isalnum　…　104
13.15　isalpha　…　105
13.16　isascii　…　106
13.17　iscntrl　…　106
13.18　isdigit　…　107
13.19　isgraph　…　108
13.20　islower　…　108
13.21　isprint　…　109
13.22　ispunct　…　110
13.23　isspace　…　111
13.24　isupper　…　111
13.25　isxdigit　…　112
13.26　labs　…　113
13.27　ldiv　…　114
13.28　longjmp　…　115
13.29　malloc　…　116
13.30　memchr　…　116
13.31　memcmp　…　117
13.32　memcpy　…　119
13.33　memmove　…　120
13.34　memset　…　121
13.35　offsetof　…　122

13.36 printf … 123
13.37 putc … 126
13.38 putchar … 126
13.39 puts … 127
13.40 rand … 128
13.41 realloc … 129
13.42 setjmp … 130
13.43 sprintf … 131
13.44 srand … 131
13.45 strcmpi … 132
13.46 strcat … 133
13.47 strchr … 134
13.48 strcmp … 134
13.49 strcoll … 135
13.50 strcpy … 136
13.51 strcspn … 137
13.52 strdup … 137
13.53 strerror … 138
13.54 strlen … 139
13.55 strncmpi … 139
13.56 strncat … 140
13.57 strnchr … 141
13.58 strncmp … 142
13.59 strncpy … 142
13.60 strpbrk … 143
13.61 strrchr … 144
13.62 strspn … 145
13.63 strstr … 145
13.64 strtok … 146
13.65 strtol … 147
13.66 strtoul … 148
13.67 tolower … 149
13.68 toupper … 150
13.69 va_arg … 150
13.70 va_end … 151
13.71 va_start … 152
13.72 vfprintf … 153
13.73 vprintf … 154
13.74 vsprintf … 155

第 14 章　DSP ライブラリの関数レファレンス　…　157

14.1 circinc … 157
14.2 circlength … 158
14.3 fixed_int … 159
14.4 get_eir … 160
14.5 get_sr … 160
14.6 int_fixed … 161
14.7 laccum_sll … 161
14.8 laccum_sra … 162

14.9 lfixed_long … 163
14.10 long_lfixed … 163
14.11 rabs … 164
14.12 racos … 165
14.13 racosh … 165
14.14 rasin … 166
14.15 rasinh … 167
14.16 ratan … 167
14.17 ratan2 … 168
14.18 ratanh … 169
14.19 rclip … 169
14.20 rcos … 170
14.21 rcosh … 171
14.22 reverse … 171
14.23 rexp … 172
14.24 rexponent … 173
14.25 rlog … 174
14.26 rlogx … 174
14.27 rmatherr … 175
14.28 rpow … 176
14.29 rround … 177
14.30 rsin … 177
14.31 rsinh … 178
14.32 rsqrt … 178
14.33 rtan … 179
14.34 set_eir … 180
14.35 set_sr … 180

図の目次

図番号 **タイトル，ページ**

2-1 **設定ウインドウ** … 23

4-1 **係数とデータ・ベクタ** … 29

6-1 **ビット・フィールド例** … 46
6-2 **アドレス演算例** … 46
6-3 **メモリ空間割り当て例** … 47
6-4 **可能なメモリ空間変換** … 48
6-5 **固定小数点データのビット・パターン変換** … 50
6-6 **固定小数点データの特殊関数** … 50
6-7 **リング・バッファ操作関数** … 51
6-8 **リング・バッファ操作マクロ** … 52
6-9 **リング・バッファ使用例** … 53
6-10 **コンパイル後のリング・バッファ使用例** … 53
6-11 **ビット反転操作関数** … 55
6-12 **ビット反転操作マクロ** … 55
6-13 **データ再配列例** … 56
6-14 **コンパイル後のデータ再配列例** … 57
6-15 **典型的** C **コード（ループ）** … 58
6-16 **ループ制御プラグマの構文** … 61
6-17 **割り込みプラグマの構文** … 64
6-18 **割り込み制御レジスタ・アクセス関数** … 66
6-19 **割り込み関数定義** … 67

7-1 **セーブされるレジスタ** … 69
7-2 **スタック・レイアウト** … 70
7-3 **グローバル変数の宣言** … 70
7-4 **データ・セグメント名の構文** … 71
7-5 **セグメント・プラグマの構文** … 72
7-6 **可能な変数宣言と定義** … 73

8-1 **スタートアップ・コードの構成部** … 75

11-1 C **ソース記述例** … 81
11-2 -i **オプションでコンパイルしたアセンブラ・コード** … 81

表の目次

表番号　　**タイトル，ページ**

1-1　**表記法**　…　18

2-1　**インストールされるファイル**　…　20

5-1　C **コンパイラ定義**　…　42

6-1　**整数データ型**　…　44
6-2　**整数型変換**　…　45
6-3　**メモリ空間修飾子**　…　47
6-4　**固定小数点データ型**　…　48
6-5　**型変換**　…　49
6-6　**ハードウエア・ループへのコンパイルを可能にする関数**　…　59
6-7　**ループ制御オプション**　…　60
6-8　**モジュール範囲ループ制御オプション**　…　61
6-9　**レジスタ無効コマンド・ライン・オプション**　…　65
6-10　**レジスタ許可プラグマ・オプション**　…　65

7-1　**呼び出し規則のまとめ**　…　69

12-1　**シンボリック・ディバグのアセンブリ命令**　…　84

第1章　概　　説

CC77016（以降，C コンパイラといいます）は，16 ビット固定小数点ディジタル・シグナル・プロセッサμPD77016 ファミリ[注], μPD77116 用の C コンパイラです。本 C コンパイラは，C ソースを，μPD77016 ファミリ, μPD77116 のアセンブラ言語に変換するソフトウエア・ツールです。C 言語は，小数点データ型を除いて，ANSI 規格の X3.159 －1989 に準拠しています。追加仕様に関しては，**第 6 章　言語仕様**を参照してください。

★　また，C コンパイラは C コンパイラ単体の製品ですので，最終的なプログラム・コードを生成するには，SP77016/SP77210 のアセンブラ／リンカが必要です。SP77210 と共に使用する場合には，統合環境（ADS: Atair Developer Studio）上でコンパイルから，アセンブル，リンクまでが可能になります。

注　μPD77016 ファミリは，μPD7701x ファミリ（μPD77015，77016，77017，77018A，77019），μPD77111 ファミリ（μPD77110，77111，77112，77113A，77114, 77115），およびμPD77210 ファミリ（μPD77210，77213）の総称です。

注意　このドキュメントにμPD77116 についての記述がありますが，現在の CC77016 はμPD77116 には正式には対応していません。

1.1　特　　徴

本 C コンパイラは，次のものから構成されています。

★ • C コンパイラ・ドライバ

C コンパイラ・ドライバとは，入力ファイルによって C コンパイラやコマンド・ライン・アセンブラ／リンカを組み合わせて実行し，コンパイル→アセンブル→リンクを自動的に行なうプログラムです。
この C コンパイラ・ドライバは，リロケータブル・オブジェクト・ファイルと実行ファイルを生成するために，コマンド・ライン・アセンブラ／リンカのインストールが必要です。

• C コンパイラ

C コンパイラは，C ソースをμPD77016 ファミリのアセンブラ言語に変換するものです。

• オプティマイザ

グローバル・オプティマイザを使用すると，C コンパイラの処理速度を向上し，コンパイラが生成するコードのサイズを小さくすることができます。オプティマイザの使用方法と説明に関しては，**第 5 章　C コンパイラとオプション**を参照してください。

• ライブラリ

ライブラリは，数学関数やメモリ割り当て関数，標準出力関数，ストリング関数，固定小数点ルーチンを含んでいます。ご使用の製品で，ライブラリのサブルーチンを使用するための特別なライセンス契約は必要ありません。各ライブラリ関数のリストと詳細な説明に関しては，**第 13 章　標準 C ライブラリの関数レファレンス**，および**第 14 章　DSP ライブラリの関数レファレンス**を参照してください。

1.2　オーダ名称と対象OS

	ホスト・マシン	対象OS	オーダ名称（媒体）
★	PC-9800シリーズ	Windows 95, 98, 2000および Windows NT 4.0	μSAB17CC77016（CD-ROM）
	IBM PC/AT™		

1.3　パッケージ内容

本Cコンパイラには，次のものが含まれています。

- Cコンパイラ・コマンド・ライン実行プログラム
- Cコンパイラ・ユーザーズ・マニュアル
- 標準Cライブラリ
- μPD77016固定小数点ライブラリ
- シミュレーション時のキャラクタ出力サポート

1.4　対象デバイス

CC77016の対象デバイスは次のとおりです。

- μPD77015，77016，77017，77018A，77019
- μPD77110，77111，77112，77113A，77114，77115
- μPD77210，77213

1.5　表記法

本マニュアルで使用されている用語を表 1-1に示します。

表 1-1　表記法

	用　　語	説　　明
	モノスペース	モノスペース・テキストは，プログラム例やプログラム出力に使用されます。
	太字	太字は，文字どおりに使用すべき固有の用語を示します。これらの用語は，表示のとおり正確に入力する必要があります。ただし，大文字と小文字の区別は必要ありません。
	斜体	斜体文字を使用したテキストは，仮の表現です。ユーザは実際に入力する値に置き換える必要があります。
	カギかっこ［…］	カギかっこによって任意のフィールドやパラメータが表現されます。
	垂直バー｜	垂直バーは，バーの両側に表されたエントリの1つを入力することを要求します。
★	Windows	Microsoft Windows 95, 98, 2000, Windows NT4.0オペレーティング・システム。
	DSP	ディジタル・シグナル・プロセッサμPD7701xファミリ，μPD77111ファミリ，μPD77210ファミリ

★

第2章　インストール手順

2.1　アプリケーションの設定

C コンパイラには，Windows インストーラ・サービスに準じたインストール・ウィザードが添付されています。PC に C コンパイラをインストールするには，次の手順を実行してください。

（1）C コンパイラの配布メディアを任意のドライブに置きます。
（2）配布メディアのルート・フォルダにある setup.exe を実行します。
（3）インストール・ウィザードに従って，インストールを進めてください。

2.1.1　アプリケーションの設定時の注意

（1）アドミニストレータ権限

Windows NT システムに本 C コンパイラ・アプリケーションをインストールするには，アドミニストレータ権限が必要です。これは，システム構成要素のアップグレード，レジストリ・エントリの実行，およびシステム環境の修正に必要です。

（2）システム再起動

本 C コンパイラのインストールを開始する前に，インストール・ウィザードがシステム上の Windows インストーラ・エンジンをアップグレードする場合があります。このタスクは自動的に実行されます。システムの再起動が必要な場合は，Windows インストーラのアップグレードが完了したあとに，インストール・ウィザードによって自動的に実行されます。

（3）アプリケーションを使用可能にするとき

インストールが完了したあとは，ライセンス・ファイルを使って，本 C コンパイラを使用可能にする必要があります。ライセンス・ファイルを入手するには，NEC エレクトロニクス　DSP ツール・サポート[注] までライセンス要求ファイルを送っていただく必要があります。詳細については，2.2　ライセンス手続き を参照してください。

注　NEC エレクトロニクス　DSP ツール・サポート（dsp_tools@dsp.jp.nec.com）

（4）システム環境

コンパイラ・コマンド・ライン・ツールをシステム上の任意の場所から起動するには，PATH 環境変数をそれに応じて修正する必要があります。インストール・ウィザードでこのオプションを選択することを推奨します。

表 2-1はインストール後のディレクトリ構造と，各ファイルの簡単な説明を示しています（標準Cライブラリの組み込みファイルを除きます）。

表 2-1　インストールされるファイル

ファイル	内　容
CC77016 readme.html	
CC77016 EULA.html	エンド・ユーザ・ライセンス契約ファイル
REGFORM.txt	ライセンス/ユーザ登録テンプレート・ファイル
cc77016.exe	Cコンパイラ・ドライバの実行ファイル
cc77016r.exe	Cコンパイラの実行ファイル
char¥dspio.tmg	文字出力のタイミング・ファイル
char¥toa.exe	数字から文字への変換の実行ファイル
doc¥C-Compiler Manual.doc	ユーザーズ・マニュアル
include¥assert.h	
include¥ctype.h	
include¥dspext.h	固有関数の宣言
include¥dspio.h	文字出力関数の宣言
include¥errno.h	
include¥float.h	
include¥limits.h	
include¥locale.h	
include¥math.h	
include¥rmath.h	固定小数点数学関数の宣言
include¥setjmp.h	
include¥signal.h	
include¥stdarg.h	
include¥stddef.h	
include¥stdio.h	
include¥stdlib.h	
include¥string.h	
include¥strings.h	
include¥time.h	
lib¥c.asm	標準起動アセンブラ・コード
lib¥cx.rel	起動オブジェクト・コード（ディフォールト・メモリ_X用）
lib¥cy.rel	起動オブジェクト・コード（ディフォールト・メモリ_Y用）
lib¥libcx.lib	ライブラリ（ディフォールト・メモリ_X用）
lib¥libcy.lib	ライブラリ（ディフォールト・メモリ_Y用）

2.2　ライセンス手続き

システムに本 C コンパイラをインストールしたあとは，ライセンス・ファイルで使用可能にする必要があります。ライセンス・ファイルを入手し，インストールするには，次の手順に従ってください。

（1）本 C コンパイラのインストール中に，ライセンス要求ファイルを生成。

ライセンス要求ファイルは，インストール・ウィザードによって自動的に生成されます。ライセンス要求ファイル（CC77016.IID）は，本 C コンパイラのインストール・フォルダの Licenses サブフォルダに生成されます。

（2）ライセンス要求ファイルを，DSP ツール・サポートに送信。

（1）で生成されたライセンス要求ファイル（CC77016.IID）を DSP ツール・サポート宛てに電子メールで送信します。

（3）入手したライセンス・ファイルをシステムにインストール。

DSP ツール・サポートから入手したライセンス・ファイルを，本 C コンパイラのインストール・フォルダの Licenses サブフォルダにコピーします。本 C コンパイラ・アプリケーションを起動します。

2.2.1　ライセンス手続き時の注意

（1）ライセンス・ファイル名とフォルダ名

本 C コンパイラのインストール中に生成された Licenses フォルダの名前，ライセンス要求ファイルの名前，入手したライセンス・ファイルの名前は変更しないでください。

（2）正式ライセンスとデモ・ライセンス

本 C コンパイラの正式ライセンスは，C コンパイラのデモ・バージョンに適用することはできません。

2.3 ビルド・プロセス

アプリケーション設定とライセンス手続きが終了すると，本Cコンパイラを実行できるようになります。

Cコンパイラ・ドライバcc77016.exeを使うと，アセンブラ・ファイル（*.asm），リロケータブル・オブジェクト・ファイル（*.rel），または実行ファイル（*.lnk）を生成することができます。これらのファイルを生成するために必要なオプションについては，**第3章　Cコンパイラ・ドライバ**を参照してください。実行ファイルを生成するとき，libディレクトリからいくつかのファイルを実行ファイルにリンクする必要があります。これらのファイルは，Cコンパイラ・ドライバ・オプションに依存し，適切な起動コードと可能な標準Cライブラリになります。Cコンパイラ・ドライバには，コマンド・ライン・アセンブラ／リンカが必要です。

Cコンパイラ・ドライバの代わりとして，本Cコンパイラcc77016r.exeを直接使うことができます。このCコンパイラによるコンパイルでは，*.asmファイルのみを生成できます。このアセンブラ・ファイルはWB77016ワークベンチまたはADS（Atair Developer Studio）にロードでき，続いてアセンブラを呼び出すことで，対応するリロケータブル・ファイル（*.rel）を生成できます。正しい実行ファイルを生成するには，libディレクトリからいくつかのファイルをこのリロケータブル・ファイルにリンクする必要があります。コンパイル中のディフォールト・メモリ・スペースの選択によっては（5.6.1　-defyを参照），適切な起動コードとライブラリをリロケータブル・ファイルにリンクする必要があります（cx.rel, libcx.lib，またはcy.rel, libcy.libのどちらか一方）。WB77016ワークベンチまたはADSの代わりに，コマンド・ライン・アセンブラ／リンカを使って実行ファイルを生成することもできます。

場合によっては，標準スタートアップ・リロケータブル・ファイル（cx.rel，cy.rel）は，ハードウエアおよびユーザ・プログラムの要求を満たさないことがあります。この場合は，適応させた起動アセンブラ・コード（c.asm）から新しいリロケータブル・ファイル（c.rel）をビルドして，ほかのリロケータブル・ファイルにリンクする必要があります（**第8章　スタートアップ・コードの構成**を参照）。

ディバグのためには，シミュレータ上でプログラムを実行中にコメント行を出力する方法があると便利です。charディレクトリには，シミュレーション中の文字出力をサポートする2つのファイルが含まれています。詳細については，**第10章　文字出力**を参照してください。

2.3.1 SP77016(WB77016)+CC77016によるビルド

（1）CC77016V1.5のインストールで，Optional Tasksの設定にて”Update the PATH environment variable”をチェックしていない場合は，cc77016.exeまでのパスを設定します。

（2）CC77016rでオプションを設定した上でコンパイルし，アセンブラ･ファイル（*.asm）が出力されます。

（3）WB77016 にてアセンブル／リンク

（a）Projectを生成

（b）生成したProjectにユーザ･コードを追加

（c）生成したProjectに標準スタートアップ･コード（cx.relまたはcy.relのどちらか一方）と，標準ライブラリ（libcx.libまたはlibcy.libのどちらか一方）を追加

（d）Projectの設定（アセンブラ／リンカ･オプションの設定および，Model fileの定義）

（4）プロジェクトのビルドによって，（3）（d）によって設定された出力ファイルが生成されます。

2.3.2　SP77210+CC77016 によるビルド（コンパイラ･ドライバを用いる場合）

（1）CC77016V1.5 のインストールで，Optional Tasks の設定にて “ Update the PATH environment variable ” をチェックしていない場合は，cc77016.exe までのパスを設定します。

（2）コンパイラ･ドライバ（CC77016）にて，コンパイラ･ドライバのオプションを設定し，コンパイル，アセンブル，リンクを行います（コンパイラ･ドライバでは適切な C コンパイラ／アセンブラ／リンカ･オプションを自動的に生成するため，標準インクルード･パス，標準スタートアップ･コード，標準ライブラリを明示的に指定する必要がありません）。

（3）指定された出力ファイル（*.asm, *.rel, *.lnk のいずれか）が生成されます。

2.3.3　SP77210+CC77016 によるビルド（ADS 上でビルドする場合）

（1）Project Workspace を生成します。

（2）Project Workspace に Project を追加します。

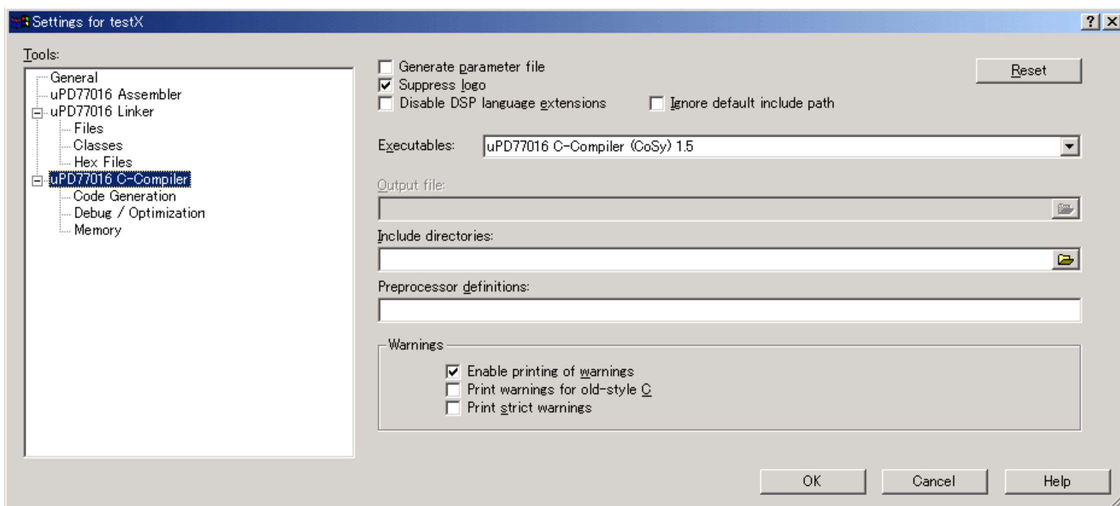
（3）生成した Project にユーザ･コードを追加します。

（4）生成した Project に標準スタートアップ･コード（cx.rel または cy.rel のどちらか一方）と，標準ライブラリ（libcx.lib または libcy.lib のどちらか一方）を追加します。

（5）Project の設定（アセンブラ／リンカ･オプションの設定および，Model file の定義）

（6）C コンパイラの設定

図 2-1　設定ウインドウ



（a）uPD77016 C-Compiler:

この項目では，次の設定が可能です。

・インクルード･ファイルまでのパス設定

・ワーニングの設定

・DSP-C を適応するかどうかなど

（b）Code Generation

この項目では，次の設定が可能です。

・高精度固定小数点乗算を有効にするかどうか

・ハードウエア・ループ制御の設定

・インライン制御の設定および，インラインする関数／しない関数の設定

（c）Debug / Optimization

ディバグ･オプション，最適化に関するコード生成オプションの設定が可能です。

（d）Memory

メモリ･オプションの設定が可能です。

★ 第3章　C コンパイラ・ドライバ

C コンパイラ・ドライバとは，入力ファイルによって C コンパイラやコマンド・ライン・アセンブラ／リンカを組み合わせて実行し，コンパイルからアセンブル，リンクまでを自動的に行うプログラムです。また，C コンパイラ・ドライバを用いると，標準インクルード・パス，スタートアップ・ファイル，標準ライブラリを明示的に指定する必要がなくなります。

C コンパイラ・ドライバは，次のことを行います。

- コマンド・ライン上で 1 つ以上の C ソース・ファイルを指定します。
- コマンド・ライン上で 1 つ以上のアセンブラ・ファイル，リロケータブル・ファイル，またはオブジェクト・ライブラリを指定します。
- 適切な C コンパイラ・コマンド・ライン・オプションを自動的に生成し，C コンパイラに送ります。
- アセンブラおよびリンカ・コマンド・ライン・オプションを自動的に生成し，コマンド・ライン・アセンブラ／リンカに送ります。

C コンパイラ・ドライバを用いて，リロケータブル・オブジェクト・ファイルと実行ファイルを生成するため，コマンド・ライン・アセンブラ／リンカのインストールが必要です。これらのコマンド・ライン・ツールがない場合は，C コンパイラ・ドライバはコマンド・ライン上で 1 つ以上の C ソース・ファイルのコンパイル（アセンブラ・ファイルの生成）にしか使用できません。また，コマンド・ライン・アセンブラ／リンカが用意されているのは SP77210 だけです。WB77016 ワークベンチでは，コマンド・ライン・アセンブラ／リンカは利用できません。

3.1　C コンパイラの呼び出し

C コンパイラ・ドライバは，次のシーケンスで呼び出せます。

```
cc77016 [options | files]*
```

ここで，files は C ソース・ファイル，アセンブラ・ファイル，リロケータブル・ファイル，またはオブジェクト・ライブラリです。options は，次に説明する一連の C コンパイラ・ドライバ・オプションです。1 つ以上の C ファイルのコンパイルとアセンブルが可能であり，C コンパイラ・ドライバを 1 回呼び出すだけで，複数の C ソース・ファイルおよびアセンブラ・ファイルをコンパイル／アセンブルできます。

3.2　グローバル・オプション

3.2.1　-?

すべての利用可能な C コンパイラ・ドライバ・オプションの簡単な説明を表示します。

3.2.2　-v

-v オプション設定後に実行されるすべてのプログラムのプログラム名とコマンド・ライン・オプションを表示します。

3.2.3　-n

-n オプション設定後に，実行されるすべてのプログラムのプログラム名とコマンド・ライン・オプションを表示しますが，実行はしません。このオプションは，あらかじめプログラム名とコマンド・ライン・オプションをチェックするために使用します。

3.2.4　-f *parameterfile*

コマンド・ライン上で入力するオプションに加えて，パラメータ・ファイルを使用することができます。

プレーン ASCII テキスト・ファイルである *parameterfile* から一連のコマンド・ライン・オプションを読み取ります。パラメータ・ファイル内のコメント行は，「#」文字で始まります。識別子，ファイル名，その他の文字列（セグメント名，クラス名，モデル・ファイル名，モデル名など）は，引用符で囲む必要があります。「#」文字が文字列に含まれる場合は，その文字列全体を引用符で囲む必要があります。

3.2.5　-nologo

著作権メッセージを表示しません。

3.3　ファイル・オプション

3.3.1　-o *filename*

出力ファイル名を *filename* とします。このファイルは，実行ファイル（*.lnk）です。ファイルの種類は，-s および-c オプションに依存します（3.3.2　-s, 3.3.3　-cを参照）。-s, -c オプションを使わない場合は，filename.lnk となります。

3.3.2　-s

C コンパイラのあとに C コンパイラ・ドライバを停止し，アセンブラ・ファイルを生成します。アセンブラ・ファイル名は入力されるファイル（source.c）から source.asm となります。

3.3.3　-c

アセンブラのあとに C コンパイラ・ドライバを停止し，リロケータブル・オブジェクト・ファイルを生成します。リロケータブル・オブジェクト・ファイル名は入力されるファイル（source.c）から source.rel となります。

3.3.4　-save-temps

C コンパイラ・ドライバは，中間ファイルを削除しません。これらのファイルは，生成されたアセンブラ・ファイル（*.asm）とリロケータブル・オブジェクト・ファイル（*.rel）です。

3.3.5　-nostdinc

C コンパイラの標準的に組み込まれるインクルード・パスを無効にします。このパスは C コンパイラのインストール中に設定されます。

3.3.6　-nostartupfile

リンカは，実行ファイルに標準スタートアップ・コードを自動的にリンクしません。スタートアップ・コードへのパスは，C コンパイラのインストール中に設定されます。

3.3.7　-nostdlib

リンカは，実行ファイルに標準 C ライブラリを自動的にリンクしません。C ライブラリへのパスは， C コンパイラのインストール中に設定されます。

3.4　C コンパイラ・オプション

第 5 章　C コンパイラとオプションで説明するすべての C コンパイラ・オプションが使えます。これらのオプションは，C コンパイラ・ドライバからコンパイラに渡され，すべての C ソース・ファイルに使われます。

3.5　アセンブラ・オプション

3.5.1　-m *modelfile*

アセンブラのモデル・ファイルを定義します。ディフォールトのモデル・ファイルはありません。アセンブラを使う場合は，モデル・ファイルを定義する必要があります。また，このオプションはリンカにも同じモデル・ファイルを設定します。

使われた modelfile は，次のように考慮されます。

- この名前または拡張子.model を持つファイルが存在する場合は，それを使用します（ディレクトリを伴うフル・パス名が可能）。
- CommonFiles ディレクトリの Atair Software サブディレクトリにこの名前または拡張子.model を持つファイルが存在する場合は，それを使用します（CommonFiles ディレクトリは，インストールされたオペレーティング・システムに依存し，「C:¥Program Files¥Common Files」のようになります）。

また，環境変数 CC77016_MODEL を使って，modelfile を設定することもできます。考慮されるファイル名は，上記と同様に解決されます。コマンド・ライン・オプションは，環境変数に優先します。

3.5.2　-WA *options [, options]*

1 つ以上のアセンブラ・オプションを定義します。これらは一般オプションであり，すべてのアセンブラ・ファイルに適応します。これらのオプションは，アセンブラに渡されます。

3.5.3　-Wa *options [, options]*

1 つ以上のアセンブラ・オプションを定義します。これらのオプションは次のアセンブラ・ファイルにのみ適用されます。アセンブラ・ファイルとしては，C ソース・ファイルから生成されたアセンブラ・ファイル，またはコマンド・ラインからのアセンブラ・ファイルが可能です。これらのオプションは，アセンブラに渡されます。

3.6　リンカ・オプション

3.6.1　-m *modelfile*

リンカのモデル・ファイルを定義します。ディフォールトのモデル・ファイルはありません。リンカを使用したい場合は，モデル・ファイルを定義する必要があります。また，このオプションはアセンブラにも同じモデル・ファイルを設定します。

使用された modelfile は，次のように考慮されます。

- この名前または拡張子.model を持つファイルが存在する場合は，それを使用します（ディレクトリを伴うフル・パス名が可能）。
- CommonFiles ディレクトリの Atair Software サブディレクトリにこの名前または拡張子.model を持つファイルが存在する場合は，それを使用します（CommonFiles ディレクトリは，インストールされたオペレーティング・システムに依存し，「C:¥Program Files¥Common Files」のようになります）。

また，環境変数 CC77016_MODEL を使用して，modelfile を設定することもできます。考慮されるファイル名は，上記と同様に解決されます。コマンド・ライン・オプションは，環境変数に優先します。

3.6.2　-WL *options [, options]*

1 つ以上のリンカ・オプションを定義します。これらのオプションは，リンカに渡されます。

3.6.3　-l *name*

追加ライブラリのショートカットを定義します。ライブラリの名前は lib*name*.lib です。コマンド・ライン上でフル・ライブラリ名を使ってライブラリを追加することもできます。

3.6.4　-L *path*

追加ライブラリ・パスを定義します。*path* は，カレント・ディレクトリでリロケータブル・オブジェクト・ファイルとオブジェクト・ライブラリを追加で検索するために使われます。オブジェクト・ファイルは，コマンド・ラインで定義するか，リンカ・オプション-l から得られます。

第4章　チュートリアル

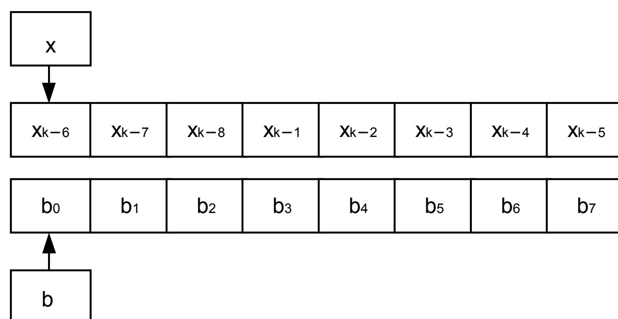
本章では，本 C コンパイラ用の C コードを効率的に記述する方法について解説します。効率的に C コードを記述した場合，コンパイラが出力するアセンブラ・コードは非常に短いものになります。

本章では，最適なアセンブラ・コードを出力するようステップごとに C コードが洗練されていく C ソース例を使用して解説します。この例では，サンプリングされたデータ値 x_k のストリームを演算する C 関数を記述する練習を行います。ここで，$k \in [-\infty, \infty]$ は，サンプリング・インスタントを意味しています。実現される FIR フィルタ・アルゴリズム（有限インパルス応答）は，次のように演算されます。

$$y_k = b_0 x_k + b_1 x_{k-1} + \cdots + b_{n-1} x_{k-n+1}$$

ここで，b_0，b_1，…，b_{n-1} は定数（フィルタ係数）を示し，n はフィルタ係数の数（フィルタ次数）を，y_k はインスタント k に対するフィルタ出力を示します。フィルタ係数と過去にサンプリングされた n 個の入力値を格納しておくことが必要となりますが，これは，2 つの C データ・ベクタによって行います。フィルタ・ルーチンには実測値（x_k）とフィルタ係数を示すポインタ（b），過去にサンプリングされた値を示すポインタ（x），およびフィルタ次数を入力します。関数からは，実際のフィルタ出力 y_k が出力されます。図 4-1に，関数の入り口直前の次数 8 のフィルタのレイアウトを示します。

図 4-1　係数とデータ・ベクタ



以前に測定されたデータ値は，周期的にデータ・ベクタに格納されることに注目してください。関数の最初の演算では，最も古い値 x_{k-8} を実測値 x_k に置換することですが，このためには，最も古い値の位置に関する情報が必要となります。したがって，最も古いサンプル値の位置に対応する C 関数入力 p が必要となります。出力値の演算中は，以前にサンプリングされた値はすべて参照され，最後の値 x_{k-7} は，データ・ベクタの位置 1 に存在しています。関数が再度呼び出されると，k が 1 だけ増加するため，位置 1 の値は，添え字 k−8 を取ります。したがって，関数が呼び出される度に，位置 p は要素を 1 つ左にシフトし，各呼び出し間で保存されていなければなりません。このために，p は関数の入出力パラメータとして使用され，ポインタ・パラメータとして関数に渡されます。

●例 1

```
int ansi_fir (int xk, int *x, int *p, int *b, int n)
{
        long    yk = 0L;
        int     i;

        x[*p] = xk;     /* rewrite x[k-n] by x[k] */
        for (i=0; i < n; ++i)
                yk += (long) x[(*p+i)%n] * (long) b[i];

        return (int) (yk >> 15);
}
```

例 1 は，FIR フィルタを ANSI 規格で実現したものです。しかしこのコードは，広いデータ型から狭いデータ型への整数変換と同様，return 文における long から int への型キャストはコンパイラの実行に依存し，シフト式 yk＞＞15 は，非明示的にデータ型 int が 16 ビットのサイズを持っていると想定しているため，ポータビリティがありません。

シフト式 yk＞＞15 の 15 という数は，次のことを示しています。ansi_fir 関数は，サンプリングされたデータの各ビット・パターンが，ビット 14 と 15 の間に 2 進小数点を持つ固定小数点数であると解釈します。この型の数は，−1 から $1-2^{-15}$ の範囲を持つため，ビット・パターン z を整数とする解釈と，固定小数点であるとする解釈の関係は，$z_{fix}=2^{-15}z_{int}$ と表現することができます。このアルゴリズムのすべての乗算は long 数の範囲で行われるため，μPD77016 のアーキテクチャではこのデータ型は 32 ビット幅となり，すべての中間結果 y_k の精度を上げています。しかし，最終結果は 16 ビットの固定小数点値に戻す必要があるため，15 ビット右シフトが行われ，この結果 y_k の最下位ビットを破壊することになります。

●例 2

```
                                        dp1 = b, r1 = yk = 0
                                        r5l = n, r4 = i = 0
                                        r2  = r5l <<24
        LOOP 51 {
__L2:
        r5 = r0 + r4;                   r5 = *p + i
        r5 = r5 SLL 24;                 sign extension of r5
        r5 = r5 SRA 24;
        r6 = r2 SRA 9;                  get n in position for division
        IF (r6 < 0) r5 = -r5;           r5 = (*p+i) % n
        REP 16;
        r5 /= r6;
        r6 = r6 SRA 15;
        r7 = ABS (r6);
        r5 = r5 SRA 16;
        IF (r5 < 0) r5 += r7;
        IF (r6 < 0) r5 = -r5;
        r5 += r3;
        dp2 = r5l;                      dp2 = x + (*p+i)%n
        NOP;
        r6 = *dp2;                      r6 = x[(*p+i)%n]
        r4 = r4 + 1     r5 = *dp1++;    r5 = b[i],      i++
        r1 = r1 + r6h * r5h;            r1 += x[(*p+i)%n] * b[i]
        };
```

モジュロ動作（*p＋i）%n は，整数除算の剰余を算出し，データ値の配列に対する“サーキュラ”インデクスを表現するものです。この機能は，C コードに限定されたものであり，通常のアセンブラ・コードとはまったく異なります。例 2 では，左側に for ループのアセンブラ・コードを，右側に“擬似 C コード”解釈を示しています。また，関連するレジスタ内容がループ命令の前にあるヘッダに示されます。このアセンブラ・コードから分かるように，ループ命令のほとんどは，モジュロ動作とその関連動作（符号拡張）に費やされており，ループ本体中のアセンブラ命令数は，μPD77016 アーキテクチャのポスト・モジュロ・インデクス加算アドレシング・モードを使用することにより大幅に削減することができます。このモードは，リング・バッファをアクセスするためのもので，本 C コンパイラの DSP 拡張子を使用することによって，実現可能となります。

●例3

```
#include "dspext.h"

int dsp_fir (int xk, __circ int **px, int *b, int n)
{
        __circ int              *x = *px;
        long                    yk = 0L;
        int                     i;

        *x = xk;              /* rewrite x[k-n] by x[k] */
        for (i=0; i < n; ++i) {
                yk += (long) (*x) * (long) b[i];
                circinc (int, x, 1);
        }
        *px = x;
        return (int) (yk >> 15);
}
```

例3に，ポスト・モジュロ・インデクス加算アドレシング・モードの使用例を示します。 __circというキーワードは，特殊なアラインメント条件を満たすメモリ配列が存在することをコンパイラに指示しています。最も古い入力データのインデクスを関数に引き渡す必要はないため，パラメータpは省略されています。これは，完全なモジュロ・アドレス計算がμPD77016アーキテクチャのハードウエアで行われるためです。ハードウエアには，モジュロ・レジスタ（DMXまたはDMY）を介して，リング・バッファのサイズに関する情報のみが渡されます。本Cコンパイラは，コンパイラ関数setmod_xおよびsetmod_yによるレジスタ割り当てをサポートしています。dsp_firはDMXレジスタを値n－1に初期化します。ポスト・モジュロ・インデクス加算アドレシング・モードを使用しても，関数呼び出し間の位置情報は保存しなければならないため，配列ポインタはポインタを介して関数に引き渡されます。

●例 4

```
                                                        dp2 = x,        r1 = n
                                                        dp3 = b,        r3 = yk = 0
        dn2 = 1;
        LOOP r1l {
__L3:
        r0 = *dp3++;                                    r0 = b[i],    i++
        r2 = *dp2%%;                                    r2 = x[j],    j = (j+1) % n
        r3 = r3 + r0h * r1h;                            r3 += x[j-1] * b[i-1]
        };
```

マクロ circinc は，コンパイラにリング・バッファ・ポインタをインクリメントするよう指示します。コンパイラは，インクリメント文の前にメモリ・アクセスを認識し，アセンブラ・ファイルの中で 2 つの結合文を，近隣のモジュロ・アドレスのインクリメントを伴うメモリ・アクセスに結合します（例 4）。6.5 **リング・バッファとモジュロ・アドレシング・モード**で，コンパイラがこれらの文を結合文と認識するために満たす必要がある条件を解説します。リング・バッファ・ポインタを示すポインタを，関数中に使用することはできません。これは，このポインタを介してリング・バッファ・ポインタがメモリ・アクセスとポインタ・インクリメント間で変化し，コンパイラが，この変化を検出できない可能性があるためです。したがって，エイリアスを使用しない（x を示すポインタがない）リング・バッファ・ポインタの動作を保証するために，ローカル変数 x が必要となります。これによって，コンパイラはメモリ・アクセスとアドレス・インクリメントを結合することができます。しかし，各 x を*px に置き換えて，生成されたアセンブラ・コードを検査すると良いでしょう。

●例5

```
#include ”dspext.h”
#include ”rmath.h”

__fixed dsp_fir (
        __fixed xk,
        __circ __X __fixed **px,
        __Y __fixed *b,
        int n)
{
        __circ __X __fixed     *x = *px;
        long __accum           yk = 0L;
        int                    i;

        *x = xk;               /* rewrite x[k-n] by x[k] */
        for (i=0; i < n; ++i) {
                yk += (long __accum) (*x) * (long __accum) b[i];
                circinc_x (__fixed, x, 1);
        }
        *px = x;
        return rround (yk);
}
```

アセンブラ・コードは，ディフォールトのメモリ空間（__X）とともに，2つの使用可能なメモリ空間（__Y）を使用することによって，さらに最適化されます。例5は，修正したCコードを示しています。

μPD77016のアーキテクチャは，固定小数点演算用に最適化されているため，このコードは，Cコンパイラのデータ型__fixedの使用を示しています。

long __accum精度（40ビット）で乗算が行われています。このため，1以上の中間結果を持つことができます。したがって，このCコードは，整数版とは完全に等価ではありません。もう1つの相違点は，インクルード・ファイルrmath.hの中で宣言された関数rroundにあります。この関数の呼び出しは，40ビット固定小数点数を16ビットに変換するアセンブラ命令ROUNDに直接変換されます。この動作は，整数版の32ビットから16ビットへシフトする命令とは異なり，特にy_kは結果の16ビット値の表現可能範囲外となります。この場合，ROUNDは飽和し，表現可能な16ビット固定小数点数の最大数または最小数（$1.0-2^{-15}$または-1.0）を返します。

●例6

```
        PUBLIC  _dsp_fir

dsp_fir_code IMSEG
_dsp_fir:
        r2 = r0;                r2   = xk
        r0l = *dp1;
        dp2 = r0l;                      dp2 = *px = x
        CLR (r0);                       r0   = yk = 0
        *dp2 = r2l;                     *x   = xk
        r2 = r1 SLL 24;                 r1＝n，r2 は n の符号を反映。
        IF (r2 <= 0) JMP __L6;
        dn2 = 1;
        LOOP r1l {
        r2 = *dp2%%     r1 = *dp5++;    r2 = x[j], j = (j+1)%n,  r1 = b[i], i++
        r0 = r0 + r1h * r2h;            yk += b[i-1] * x[j-1]
        };
__L6:
        r1l = dp2;
        *dp1 = r1l;                     *px = x
        r0 = ROUND (r0);
        r0 = r0 SRA 16;
        RET;
END
```

例6では，コンパイラのフィルタ関数のアセンブラ・コードを示しています。ループ本体は，両方のメモリ空間を使用しているため（dp5がYメモリを示し，dp2がXメモリを示す），2命令文の長さしかありません。

ループ本体を最終的に縮小するために，ソフトウエア・パイプラインと呼ばれる手法が使用されます。これは，ループ本体中の平行度を増加させるものです。μPD77016のアーキテクチャでは，1インストラクション・サイクル中で2回のメモリ・アクセスと演算動作を実行することが可能（命令が論理的に分離されている場合）ですが，これに対して，r0＝r0＋r1h*r2hという命令は，先行するメモリ・ロード命令によって利用可能となった情報を使用します。これを避けるために，メモリ・ロード命令と比較して，乗加算命令を1繰り返しステップ遅延させます。

●例 7

```
#include ”dspext.h”
#include ”rmath.h”

__fixed dsp_fir (
                __fixed xk,
                __circ __X __fixed **px,
                __Y __fixed *b,
                int n)
{
        __circ __X __fixed *x = *px, xi;
        __Y __fixed    bi;
        long __accum   yk = 0.0A;
        int            i;

        *x = xk;

        xi = **px;                                          /* prolog */
        bi = *b;
        circinc_x (__fixed, x, 1);
        b += 1;
        for (i=0; i < n-1; ++i)
        {
                yk += (long __accum) xi * (long __accum) bi;
                xi = *x;
                bi = *b;
                circinc_x (__fixed, x, 1);
                b += 1;
        }
        yk += (long __accum) xi * (long __accum) bi;        /* epilog */

        *px = x;

        return rround (yk);
}
```

例 7 に示す変更によって，ループ本体のアルゴリズム的表記は次のようになります。

$y_k ＋＝ x_i b_i$

$x_i ＝ x_{i+1}$

$b_i ＝ b_{i+1}$

これらの命令は分離され，以後の割り当ての中で，左辺の値は右辺の値として使用されません。この結果，値 x_i と b_i には，新しい変数が必要となります。また，ソフトウエア・パイプラインには，新しい変数が初期化されるループ・プロローグと，最後の割り当てである $y_k ＋＝ x_i b_i$ が演算されるエピローグとが必要になります。例 7 は変更された C コードを示し，例 8 は本 C コンパイラが生成した，対応するアセンブラ・コードを示しています。

●例8

```
        PUBLIC  _dsp_fir

dsp_fir_code IMSEG
_dsp_fir:
__L0:
        r3l = dp3;                                          後で dp3 を使用。これは格納レジスタ
        *dp0 = r3l;                                         であるため，スタックに収まる
        r2 = r0;
        r0l = *dp1;
        dp2 = r0l;                                          dp2 = &x[j] = *px
        dn2 = 1;
        *dp2%% = r2l;                                       x[j] = xk,  j = (j+1)%n
        r2l = *dp1;
        dp3 = r2l;
        r4 = r1 - 1;                                        r4 = n - 1
        CLR(r0)         r2 = *dp3       r3 = *dp5++;        yk = 0,  r2 = x[j-1],  r3 = b[0]
        r4 = r4 SLL 24;
        IF (r4 <= 0) JMP __L4;
        dn2 = 1;
        r1 = r1 - 1;
        REP r1l;                                            n-1 回の繰り返し
        r0 = r0 + r2h * r3h             r2 = *dp2%%                 r3 = *dp5++;
__L4:
        r0 = r0 + r2h * r3h;
        r1l = dp2;
        *dp1 = r1l;                                         *px = x
        r0 = ROUND(r0);
        r0 = r0 SRA 16;
        r3l = *dp0;
        dp3 = r3l;
        RET;
END
```

例8では，ループ本体は，1命令サイクルに縮小されています。このため，LOOP命令は，REP命令に置き換えられています。ここで注意していただきたいのは，ループ・プロローグとエピローグを使用したことによって，ソフトウエア・パイプラインのないバージョンよりも全体のコード・サイズが長くなっていることです。明らかに，新しいコードの実行時間の縮小は，ループの繰り返し回数によって異なります。

第5章　C コンパイラ とオプション

★ 5.1　C コンパイラを直接呼び出す方法

C コンパイラ・ドライバを使わずに C コンパイラを直接呼び出す方法は，次のシーケンスになります。

```
cc77016r [options] source.c
```

ここで，*source*.c はソース・ファイル，options は次に説明する一連の C コンパイラ・オプションです。C コンパイラは，対応するアセンブラ・コードを含む出力ファイル *source*.asm（-o オプションを指定しない場合）を生成します。「cc77016r -?」というコマンドを入力すると，利用可能なすべての C コンパイラ・オプションの説明が表示されます。

本章では，すべての C コンパイラ・オプションについて説明します。

5.2　ファイル・オプション

5.2.1　-o *filename*

出力アセンブラ・ファイルを *filename*.asm とします。本オプションを省略すると，出力アセンブラ・ファイルのディフォールト名は，source.asm となります。

5.2.2　-I *path*

インクルード・ファイルを検索するディレクトリのリストに，ディレクトリ *path* を付加します。

★ 5.2.3　-f *parameterfile*

コマンド・ライン上で入力するオプションに加えて，パラメータ・ファイルを使うことができます。プレーン ASCII テキスト・ファイルである *parameterfile* から一連のコマンド・ライン・オプションを読み取ります。パラメータ・ファイル内のコメント行は，#文字で始まります。識別子，ファイル名，その他の文字列（セグメント名，クラス名，モデル・ファイル名，モデル名など）は，引用符で囲む必要があります。#文字が文字列に含まれる場合は，その文字列全体を引用符で囲む必要があります。

5.2.4　-nologo

著作権に関するメッセージを表示しません。

5.3　言語オプション

5.3.1　-nodsp

DSP-C 言語拡張子を禁止します。このため，固定小数点データ型とメモリ修飾子が使用できなくなります。

5.3.2　-W

ワーニング・メッセージの表示を許可します。

5.3.3　-Wansi

ANSI C 規格のワーニング・メッセージを表示します。

5.3.4　-Wstrict

厳格なワーニング・メッセージを表示します。

5.3.5　-Wall

オプション-W，-Wansi，および-Wstrict を結合します。

5.3.6　-D*macro*［＝*value*　］

値 value でプロセッサ・シンボル macro を定義します。値が省略されると，シンボルには値 1 が与えられます。

5.4　コード生成オプション

5.4.1　-O0

すべての最適化を禁止します。-g オプションを指定すると，本オプションが暗黙的に設定されます。

5.4.2　-O1

基本的な最適化を許可します（コンパイルの最適化に多少時間がかかります）。

5.4.3　-O2

強制的に最適化を行ないます。C コンパイラは基本最適化のほかに，関数のインライン，定数の伝播，コピー伝播，およびその他の最適化を行ないます。

5.4.4　-O3

全最適化を許可します。結果を向上させるために，本オプションは最も重要な最適化を繰り返して行ないます。

5.4.5　-Osize

コード・サイズを増加させる，関数のインラインのような最適化を禁止します。

★　5.4.6　-exactfp

32 ビット（long_fixed）および 40 ビット（accum_fixed）のビット高精度固定小数点乗算を有効にします。ディフォールトでは，高速乗算が生成されますが，最下位のビットで誤差が生じることがあります。このオプションを使用すると，ビット高精度乗算が生成されますが，実行速度は遅くなります。

★ ### 5.4.7　-inlinesize *statementLimit*

C コンパイラがインラインする関数を選択する命令文制限を設定します。ディフォールト値は 42 です。大きな値では，インラインする関数が多くなります。インラインを無効にするには，この値を 0 に設定します。

★ ### 5.4.8　-inline *functionName*

関数のサイズにかかわらず，*functionName* で指定された関数の強制的なインラインを行います。このオプションは，関数に inline プラグマを適用することと同じです。

★ ### 5.4.9　-loopcallsok

ループ本体で呼び出された関数がハードウエア・ループを使わないことを C コンパイラに知らせます。ユーザは，この制約を保証する責任があります。したがって，C コンパイラは，ループ本体での関数呼び出しの場合でもハードウエア・ループを生成します。

★ ### 5.4.10　-loopcountok

ループ数が 0-32767 回であることを C コンパイラに知らせます。したがって，ループ開始表現が定数でない場合，またはループ終了表現の型が符号付き整数でない場合には，定数でないループ終了表現に対してハードウエア・ループが生成されます。ループが 32767 回以上繰り返されることは通常少いため，このオプションは頻繁に使われます。

★ ### 5.4.11　-loopcountnice

ループ数が 1-32767 回であることを C コンパイラに知らせます。結果は一般に-loopcountok オプションと同じです。また，保護的な条件付き分岐は生成されません。これによって，ハードウエア・ループ周辺のコードが最適化されます。C 言語では，ループ全体をスキップするためにループ条件を使用することがよくあるため，このオプションは注意して使う必要があります。したがって，対応するループが少なくとも 1 回実行されることが確かな場合のみ，このオプションを使ってください。

★ ### 5.4.12　-loopdepth0 to -loopd epth4

loopdepth*n* は，生成されるハードウエア・ループのネスティング深さを *n* に制限します。引数 n は，0 から 4 までの数値です。

5.4.13　-noaar

アドレス演算を減少させる最適化を禁止します。このオプションは，アドレス演算によって圧迫されるレジスタを減少させ，レジスタがあふれてしまうことを避けて，コード数を減らします。

5.4.14　-nor5

C コンパイラが r5 レジスタを使用することを禁止します。μPD77016 アーキテクチャはシャドウ・レジスタを持っていません。このオプションはアセンブラ割り込み機能用のレジスタを作ります。しかし，レジスタが圧迫されているために，C コンパイラによって生成されたアセンブラ・コードは性能が低下し，コード・サイズも増加してしまいます。

5.4.15　-nor6

C コンパイラが r6 レジスタを使用することを禁止します（詳細については，5.4.14　-nor5 を参照）。

5.4.16　-nor7

C コンパイラが r7 レジスタを使用することを禁止します（詳細については，5.4.14　-nor5 を参照）。

5.4.17　-nodp2

C コンパイラが dp2 レジスタを使用することを禁止します（詳細については，5.4.14　-nor5 を参照）。

5.4.18　-nodp6

C コンパイラが dp6 レジスタを使用することを禁止します（詳細については，5.4.14　-nor5 を参照）。

★ ### 5.4.19　-noincr

純粋なポインタ・インクリメントまたはデクリメント命令でのダミー・アクセスを無効にします。このオプションは，完全な ANSI 準拠に必要ですが，パフォーマンスは低下することがあります（6.2　**アドレス演算**を参照）。

5.4.20　-norlse

余分なロード／ストア演算を減少させる最適化を禁止します。このオプションは，アドレス演算によって圧迫されるレジスタを減少させ，レジスタがあふれてしまうことを避けて，コード数を減らします。

5.5　ディバグ・オプション

5.5.1　-g

ディバグ情報を生成します。本オプションは，すべての最適化を禁止します。このオプションが設定されると，C コンパイラは，_DEBUG をユーザが利用可能であると定義します。

5.5.2　-gb

暗黙的に-g オプションを設定し，シミュレータにロードされる可能性のある.ini ファイルを追加生成します。このファイルには，C ソース・ファイルのすべての関数用のブレークポイントが含まれています。

5.5.3　-gw

暗黙的に-g オプションを設定し，シミュレータにロードされる可能性のある.ini ファイルを追加生成します。このファイルには，C ソース・ファイルのすべての関数中のローカル変数のウォッチ情報が含まれています。

5.5.4　-gwg

暗黙的に-g オプションを設定し，シミュレータにロードされる可能性のある.ini ファイルを追加生成します。このファイルには，C ソース・ファイルのすべての関数中のグローバル変数のウォッチ情報が含まれています。

5.5.5　-gbw *functionName*

暗黙的に-g オプションを設定し，シミュレータにロードされる可能性のある.ini ファイルを追加生成します。このファイルには，C ソース・ファイルのすべての関数のブレークポイントと，関数中のローカル変数のウォッチ情報が含まれています。このオプションは，“すべての関数”を指定する-gb や-gw オプションとは違って，特定の関数用のシミュレータ情報だけを生成することができます。

5.5.6　*-i*

C ソース・コードを挿入したアセンブラ・コードを生成します（**第 11 章　挿入されたコード記述**を参照）。

5.5.7　*-ih*

暗黙的に-i を設定します。C ソース・ファイルからの情報を付加使用できます（**第 11 章　挿入されたコード記述**を参照）。

5.6　メモリ・オプション

メモリ・オプションを何も指定しないと，C コンパイラは標準μPD77016 アーキテクチャ，すなわち，パラレル・アクセスを可能にする 2 つの内部メモリ空間（X および Y）と，シリアル・アクセスのみ可能な 2 つの外部メモリ空間（X および Y）を想定します。ROM は使用できません。C コンパイラは，両方の内部メモリ空間のローカル変数とパラメータ用のデータ・スタックを使用します。X メモリはディフォールトのメモリ空間で，ユーザが変数宣言を持つメモリ空間修飾子を指定しない場合，C コンパイラは，必要メモリを X 空間に割り付けます。リンクされるすべてのアセンブラ・ファイルは，同じディフォールト空間でコンパイルされなければなりません。メモリ・オプションを使用することによって，標準アーキテクチャをほかのハードウエア構成に適応させることができます。

表 5-1に，C コンパイラのハードウエア構成結果を反映するメモリ・オプションを定義します。この定義によって，アーキテクチャに依存する C コードを公式化することができます。オプション-defstackser はオプション-defser を暗示し，スタックがシリアル・アクセスされた場合，ディフォールト・メモリ空間もシリアルにアクセスされます。-singlestack オプションが設定されていない場合は，X メモリと Y メモリにそれぞれ 1 つずつの，合計 2 つのスタックが存在することになります。このため，C コンパイラは，__STACK_X と__STACK_Y（または__STACK_XSER と__STACK_YSER）を生成します。

表 5-1　C コンパイラ定義

C コンパイラ・オプションの設定		設定なし	-defy
ディフォールト・メモリ空間	設定なし	__DEAFAULTSPACE_X	__DEAFAULTSPACE_Y
	-defser または -defstackser	__DEAFAULTSPACE_XSER	__DEAFAULTSPACE_YSER
スタック	設定なし	__STACK_X	__STACK_Y
	-defstackser	__STACK_XSER	__STACK_YSER

5.6.1　-defy

Y メモリをディフォールト・メモリ空間として宣言します。ディフォールト・メモリ空間 Y を持った正しい DSP ライブラリをプログラムにリンクしてください。

5.6.2　-defser

ディフォールト・メモリ空間のシリアル・アクセスのみを宣言します。

5.6.3　-defstackser

スタックとディフォールト・メモリ空間のシリアル・アクセスのみを宣言します。

5.6.4　-singlestack

強制的に，1 つのスタックだけを使用可能とします。スタックは，ディフォールト・メモリ空間に存在します。

5.6.5　-extshared

外部データ・メモリを共有メモリとして宣言します。つまり，Y 空間は X 空間と等価になります。外部データ・メモリには，シリアル・アクセスのみが可能です。スタートアップ・コードに適切なウエイト・サイクル数を定義してください（**第 8 章　スタートアップ・コードの構成**を参照してください）。

5.6.6　-extrom

外部 X および Y メモリ空間に，ROM が存在することを C コンパイラに知らせます。外部 ROM には，シリアル・アクセスのみが可能となります。スタートアップ・コードに適切なウエイト・サイクル数を定義してください（**第 8 章　スタートアップ・コードの構成**を参照してください）。

5.6.7　-introm

内部 ROM の存在を C コンパイラに知らせます。内部 ROM には，シリアル・アクセスのみが可能となります。

5.6.8　-sepsegs

本オプションが指定されると，C コンパイラは，各グローバル変数を別々のセグメントに置きます。

5.6.9　-extimseg

本オプションが指定されると，C コンパイラは，命令セグメントを外部領域に置きます。スタートアップ・コードに適切なウエイト・サイクル数を定義してください（**第 8 章　スタートアップ・コードの構成**を参照してください）。

5.6.10　-boot

本オプションが指定されると，C コンパイラは，プリフィクス in_x_idata_（または in_y_idata）を付加してグローバルおよびローカル・スタティック C コード変数を，ブート・データ・セグメントに置きます。リンカ（WB77016　**ユーザーズ・マニュアル**を参照）は，このプリフィクスを持つすべてのセグメントを，INID RAM 初期化セグメント・クラスに結合します。このオプションを使用する場合は，スタートアップ・コード中のメモリ転送ブート・サポートを起動する必要があります（**第 9 章　ブート・サポート**を参照してください）。

第6章　言語仕様

ANSI 規格に完全に合致した C コンパイラは存在しません。ANSI C 規格では，ハードウエアの性能を最大限に引き上げるために，意図的に詳細な仕様を決定していない部分もあります。本章では，C コンパイラのハードウエアに依存する部分を明確にし，DSP 特有の言語をどのように利用していくかを解説します。

6.1　整数データ型

すべてのサイズのデータ型 char と int は，整数データ型と呼ばれます。表 6-1に，これらのデータ型のサイズと値の範囲を示します。明示的に符号化された signed int や signed char のような符号化データ型は，int や char などといったプレーンなデータ型と同様に解釈されるため，表 6-1では省略されています。

表 6-1　整数データ型

整数データ型	サイズ	アラインメント	値の範囲
char	16 ビット	16 ビット	－32768～32767
unsigned char			0～65535
short			－32768～32767
unsigned short			0～65535
int			－32768～32767
unsigned int			0～65535
long	32 ビット		－2147483648～2147483647
unsigned long			0～4294967295

表 6-2は，C コンパイラがどのように整数データ型間の型変換を処理するかを示したものです。変換規則は，ターゲット（t）のビット・パターンがソース（s）のビット・パターンからどのように派生するかを規定しています。たとえば t_{16-31}＝s_{15} といった省略表示は，“ターゲットのビット 16 から 31 にはソースのビット 15 の値が設定される（符号拡張）”というように読みます。この表に記されていない符号あり，または符号なしの 16 ビット・データ型（char， short， unsigned char や unsigned short など）は，対応する表中の 16 ビット・データ型（int や unsigned int など）と同様の動作をとります。

表 6-2　整数型変換

データ型		変換規則
ソース（s）	ターゲット（t）	
unsigned int unsigned int	long unsigned long	$t_{0\text{-}15}$＝s, $t_{16\text{-}31}$ = 0
int int	long unsigned long	$t_{0\text{-}15}$＝s, $t_{16\text{-}31}$＝s_{15}
unsigned int unsigned long int long	int long unsigned int unsigned long	t＝s
long long unsigned long unsigned long	int unsigned int int unsigned int	t＝$s_{0\text{-}15}$

ANSI 規格では，整数定数は数字から始まると規定されています。つまり，整数定数はオプショナルな文字符号を含むものではないということです。負の符号を持つ定数のコンパイルは，定数（符号なし）を保持できるデータ型を選択することから始まります。その後値は－1 で乗算されます。この結果，符号付きデータ型の下限において予期せぬ動作が発生することがあります。

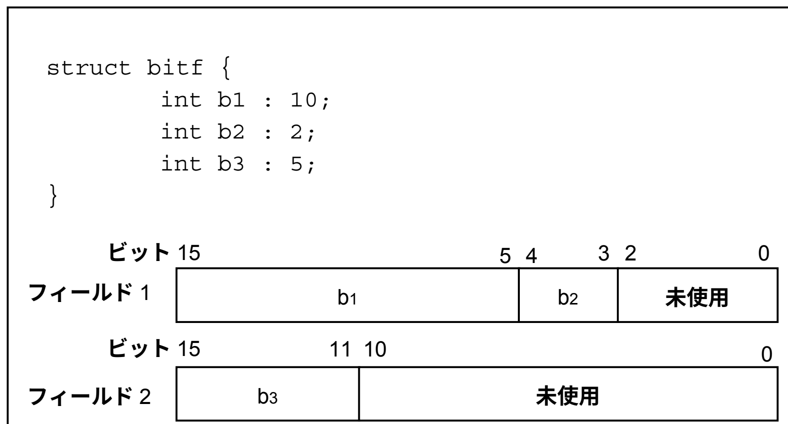
たとえば，int z＝－32768 という定義においては，定数 32768 を保持する最小のデータ型は long であり，long 値は－1 で乗算され，データ型 int に変換されます。次のステップでは，C コンパイラは，z の型と定数の型の不一致を認識し，“constant 32768 so large that it is long”というワーニング・メッセージを出力します（ワーニング・オプションの 1 つがイネーブルになっているという場合)。最後のステップでは,C コンパイラはデータ型を自動的に long から int に変換しますが，この場合値を破壊することはありません（表 6-2参照)。しかし，それぞれの符号なし定数が z の型に変換されるような，分解された定義（例：int z＝－32767－1）によるデータ変換は避けることを推奨します。すべての固定小数点データ型には，同じ引数が有効であることにも注意してください。

ANSI C では，左辺オペランドのデータ型のビット・サイズより小さい値（負の値を除く）を持つ右辺のオペランドのシフト動作（<<，>>）を定義していますが，ANSI 規格では，左辺の負のオペランドを右シフトした場合の結果を規定していません。C コンパイラでは，このような場合，算術右シフトを行い，符号ビットをビット・パターンにシフトします。たとえば，int b，a＝－8 という宣言を持つ b＝a>>1 というステートメントは，b＝－4 という値を出力することになります。

また，ANSI 規格では，左辺，右辺ともに同じ符号を持つオペランドの場合のみ，2 つの整数の除算（a/b）やモジュロ演算（a%b）を定義していますが，C コンパイラでは，片方のオペランドが負である場合，結果を切り捨てます。したがって，15/－4 という除算の結果は，－3 ではなく－4 となります。整数の除算の剰余を算出するモジュロ演算では，a＝（a/b）*b＋a%b という関係を満たすことによって整数除算を補っています。これを，上記例に当てはめると，15%－4 というモジュロ演算の結果は，3 ではなくて－1 ということになります。

ビット・フィールドは，実行形式に依存するもので，C コンパイラでは，ビット・フィールドのサイズを 1 ビットから 16 ビットまで（データ型 int および unsigned int に対応して）対応しています。後続するビット・フィールドが構造体の中で宣言された場合，それらのビット・フィールドは 1 メモリ・ワードに完全に収まるように，“高密度に圧縮された”状態となります。図 6-1に C 宣言と対応するメモリ・レイアウトを示します。

図 6-1　ビット・フィールド例



6.2　アドレス演算

ANSI 規格では，ポインタ値に対する演算をいくつか定義しています。ここでは，ポインタ値に対する加減算を解説します。μPD77016 のアーキテクチャでは，2 つの異なる方法が用意されています。図 6-2の左側に示したアセンブラ命令は，プロセッサの ALU を使用してデータ・ポインタ・レジスタ dp2 に 4 を加算することを示しています。また，右側の命令は，同じ dp2 に対して，ポスト・モディファイ・アドレシング・モードを使用して演算を行っていることを示しています。

図 6-2　アドレス演算例

```
r0l = dp2;                r0l = *dp2##4;
r0 += 4;
dp2 = r0l;
```

ポスト・モディファイ・アドレシング・モードには，1 つのアセンブラ命令しか必要でないという利点があり，これが C コンパイラに採用されている理由でもあります。しかしながら，アドレス計算のたびに，不必要なメモリ・アクセスが起きてしまうという欠点もあります。

ANSI 規格では，ポインタ変数は配列中のメモリ領域のアドレスを示すことができます。また，ポインタをインクリメントまたはデクリメントするステートメントは，ポインタが配列の上位を越える位置を示す場合，正確に演算されなければなりませんが，C コンパイラでは，この場合正確なプログラム実行は保証されません。これは，配列の上位を越えるメモリ・アクセスは，プロセッサ予約のメモリ領域にアクセスしてしまう可能性があるためです。この可能性は，非常に低いものではありますが，まったくないとは言い切れません。

このため，-noincr コンパイラ・オプションが用意されています。このオプションを使用すると，C コンパイラは，ポインタのインクリメントまたはデクリメント動作をダミー・メモリ・アクセスを必要とせず実行できるため，ANSI 規格に完全に準拠することになります。

また，-noincr オプションを使用することには，もう 1 つの利点があります。外部メモリの転送速度が遅い場合，実行時に，図 6-2に示した 3 つのアセンブラ命令によるアドレス・インクリメントよりも，ポスト・モディファイ・アドレシング・モードの方が実行速度が遅くなる場合があるということです。

C コンパイラでは，可能なかぎりメモリ・アクセス命令とポインタ・インクリメント命令を組み合わせています。これは，これらの命令が定義されたメモリ位置のみをアクセスするかぎり，命令は-noincr オプションには影響されないためです。

6.3　宣言上のメモリ空間修飾子

μPD77016のアーキテクチャでは，2つのメモリ空間（XおよびYデータ・メモリ）用に独立した2つのデータ・バスを用意しています。しかし，利用可能な外部メモリの半分だけを使用して，2つのデータ・バスを結合することにより，XおよびYメモリの同じアドレスが，同じ物理メモリ位置を参照する（共用メモリ）ようにすることもできます。また，メモリ構成によっては最大3個の命令を並列実行（Xメモリにアクセスする命令，Yメモリにアクセスする命令，および算術演算命令）できるものもあれば，できないもの（シリアル・アクセスのみ）もあります。アーキテクチャ上のバリエーションを成す5個のメモリ空間修飾子を表 6-3に示します。

表 6-3　メモリ空間修飾子

メモリ空間修飾子	メモリ割り当て	アクセス・タイプ
__X	Xメモリ	並列アクセス可能
__Y	Yメモリ	並列アクセス可能
__XSER	Xメモリ	シリアル・アクセスのみ
__YSER	Yメモリ	シリアル・アクセスのみ
__SSER	共用	シリアル・アクセスのみ

これらの修飾子を使用することによって，どのようなデータ項目（変数や定数など）でも，任意のメモリ空間に割り当てることができます。変数定義に修飾子を使用しない場合，必要となるメモリ領域はディフォールトのメモリ空間に割り当てられます（5.6　**メモリ・オプション**参照）。

図 6-3　メモリ空間割り当て例

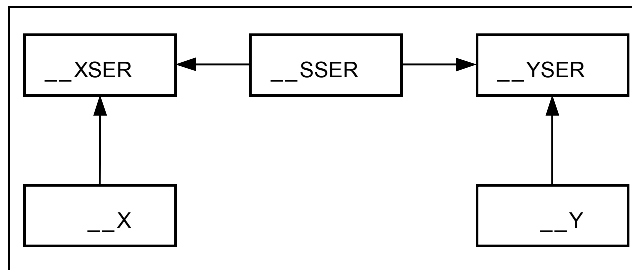
```
__X int x1;                x1，x2はXメモリに割り当てられる。
int __X x2;
__Y int y1;                y1はYメモリに割り当てられる。
__X int * __Y px;          pxはYメモリに割り当てられ，Xメモリを指す。
__Y int * __Y py;          pyはYメモリに割り当てられ，Yメモリを指す。
__XSER int *pxs;           pxsはディフォールト・メモリに割り当てられ，Xメモリを指す。
x1 = y1;                   legal
px = &x2;                  legal
y1 = *py;                  legal
*px = *py;                 legal
px = py;                   illegal
pxs = px;                  legal
px = pxs                   legal
```

図 6-3では，いくつかのメモリ空間修飾子の用法を示しています。最後の3行は，ポインタ変換を示しています。明らかに，Xメモリを示すポインタが，Yメモリにある変数を示すことはできません。このため，XメモリからYメモリへのポインタ変換を行うことはできません。並列アクセス可能な，メモリ空間に割り当てられた各ポインタは，シリアルにアクセスすることも可能です。したがって，並列メモリからシリアル・メモリへのポインタ変換は可能ですが，その逆は通常は不可能です。しかし，状況によってはこのような変換も可能となります。図 6-3では，pxs＝&x1という割り当てが可能です。

pxsは，並列アクセス可能なメモリ空間を示すことが解っているため，その後のpx＝pxsという割り当ても可能となっています。Cコンパイラでは，データ・フローを詳細に分析しないため，ワーニングを表示します（ワーニング・オプションが設定されている場合）。

図 6-4では，同様な配慮をした完全な変換を示しています。図中の矢印は可能なポインタ変換を示し，その向きは安全な変換の方向を示します。しかし，危険なポインタ変換でもまったく使用不可能という訳ではないので，無意味なポインタ変換を回避するのはユーザの責任となります。

図 6-4　可能なメモリ空間変換



6.4　固定小数点データ型

μPD77016のアーキテクチャは3種類の固定小数点データ型に対応しています。表 6-4に，標準的なANSI Cの整数データ型に加えて使用可能な，これら3種類のデータ型を示します。なお，Cコンパイラは，符号なし固定小数点データ型および浮動小数点データ型には対応していませんので，注意してください。

表 6-4　固定小数点データ型

データ型	サイズ	ビット間のバイナリ・ポイント	値の範囲	サフィックス
__fixed	16ビット	14および15	$-1.0 \sim 1.0-2^{-15}$	r
long __fixed	32ビット	30および31	$-1.0 \sim 1.0-2^{-31}$	R
long __accum	40ビット	30および31	$-256.0 \sim 256.0-2^{-31}$	A

表 6-5　型変換

データ型		変換規則
ソース（s）	ターゲット（t）	
long __fixed long __accum int long	__fixed	$t = s_{16\text{-}31}$ $t = s_{16\text{-}31}$ $t_{0\text{-}14} = 0, t_{15} = s_0$ $t_{0\text{-}14} = 0, t_{15} = s_0$
__fixed long __accum int long	long __fixed	$t_{0\text{-}15} = 0, t_{16\text{-}31} = s$ $t = s_{0\text{-}31}$ $t_{0\text{-}30} = 0, t_{31} = s_0$ $t_{0\text{-}30} = 0, t_{31} = s_0$
__fixed long __fixed int long	long __accum	$t_{0\text{-}15} = 0, t_{16\text{-}31} = s, t_{32\text{-}40} = s_{15}$ $t_{0\text{-}31} = s, t_{32\text{-}40} = s_{31}$ $t_{0\text{-}30} = 0, t_{31\text{-}40} = s_{0\text{-}9}$ $t_{0\text{-}30} = 0, t_{31\text{-}40} = s_{0\text{-}9}$
__fixed long __fixed long __accum	int	t = sign (s) • floor (|s|)
__fixed long __fixed long __accum	long	t = sign (s) • floor (|s|)

負の固定小数点定数の定義に関するガイドラインは，整数定数（6.1　整数データ型参照）と同様です。たとえば，__fixed 定数 z＝－1.0r の定義は，z＝－0.5r－0.5r とした方がわかりやすく，不必要な非明示的型変換を避けることになります。同様に，最小 long __fixed 定数 z＝－1.0R および long __accum 定数 z＝－256.0A は，それぞれ z＝－0.5R－0.5R および z＝－255.0A－1.0A となります。

★　固定小数点演算は，16 ビット，32 ビット，および 40 ビット演算を使って実行されます。整数の演算と同様に，加算や減算などの演算では，最上位のビットでオーバフローが発生することがあります。しかし，整数の演算とは異なり，乗算や除算などのその他の演算では，さらに「最下位のビットでのオーバフロー」が発生することがあります。すなわち，結果の精度は，2 進小数点以下の可能なビット数によって制限されます。μPD77016 アーキテクチャは，16 ビット入力データで実行されるアセンブラ命令のみを提供しています。これは，32 ビット演算（オペランドと結果は 32 ビット幅）は，4 つの 16 ビット演算を必要とし，64 ビットの結果を生み出すことを意味します。しかし，上位 32 ビットのみが結果のデータ型に収まり，結果の下位の 32 ビットは失われます。乗算過程を詳細に分析すると，4 つの部分的な乗算の 1 つは，最大でも 32 ビットの結果の最下位ビットにのみ影響していることが分かります。したがって，最大でも最下位ビット 1 ビットのサイズ（2^{-31}）の誤差を許容するならば，16 ビット演算のうちの 1 つをスキップすることができます。32 ビットおよび 40 ビット固定小数点乗算では，C コンパイラは，これをディフォールトで行います。しかし，ビット真数乗算の結果が必要な場合は，-exactfp コマンド・ライン・オプションを使ってください。

固定小数点データ型からほかのデータ型（ANSI C の int や long を含む）への型変換は任意に可能ですが，常に意味のあるものとはいえません。表 6-5は，固定小数点データ型が使用される場合の C コンパイラの型変換に関わる規則を示しています。省略形で示された型変換規則は表 6-5に示したものと同じですが，固定小数点データ型から整数データ型への変換が異なります。この場合，t＝sign（s）• floor（｜s｜）という規則では，引数を次の最も小さ

い整数値に切り下げる floor 関数を使用します。

sign 関数は引数の符号に応じて正または負の符号を返します。つまり，データ型 int（または long）に変換されたソース値 s＝−2.75A の結果は，t＝sign（−2.75）• floor（2.75）＝−1 • 2＝−2 となります。

注意　固定小数点型から整数型への変換では，整数データ型として解釈される固定小数点ビット・パターンは生成されません。しかし，このような型変換には，図 6-5で示すライブラリ・インクルード・ファイル dspext.h でアクセス可能なコンパイラ関数が対応しています。

データ型 long __accum（40 ビット）は，整数データ型のどれにもあてはまりません。関数 laccum_read32 は，パラメータ pos を使用して値 x の 32 ビット部分を削除しています。pos＝0 は，下位 32 ビット，つまり long __accum 値の long __fixed 部を生成します。pos＝1 はビット 1 からビット 32 よりパターンを回収するため，laccum_read32 は型変換を行うだけでなく，非明示的な右シフト動作を開始することになります。pos の値は，［0，39］の範囲を取り，pos の値が 8 以上であれば，結果として生じる unsigned long 値の余剰ビットはゼロで埋められます。

図 6-5　固定小数点データのビット・パターン変換

```
int fixed_int (__fixed x);
long lfixed_long (long __fixed x);
unsigned long laccum_read32 (long __accum x, int pos);
__fixed int_fixed (int x);
long __fixed long_lfixed (long x);
```

0x0001 のような整数リテラルは，0.000030517578125r のような固定小数点表記よりは読みやすいため，ビット・パターン変換関数は，読みやすい C コードを生成するために便利です。このような整数リテラルを使用するもう 1 つの理由は，&（ビット単位の論理積），｜（ビット単位の論理和），＾（ビット単位の排他論理和），>>（右シフト），<<（左シフト），~（補数）といった，ビット操作の C 演算子には，固定小数点データ型が対応していないためです。これらの演算子を固定小数点値に適用すると，ビット・パターンの変換後，整数値の範囲でのビット操作を行う必要があります。

図 6-6　固定小数点データの特殊関数

```
long __fixed rclip (long __accum);
__fixed rround (long __accum);
unsigned int rexponent(long __fixed);
long __accum rabs (long __accum x);

#define rrabs(type, val) ((type)rabs((long __accum)(val)))
```

図 6-6に示したコンパイラ関数宣言は，ライブラリ・インクルード・ファイル rmath.h の一部です。これらの宣言によって，μPD77016 のアーキテクチャが提供する特殊命令が使用可能となります。rclip はアセンブラ命令 CLIP に直接コンパイルされ，rround は ROUND に，rexponent は EXP にそれぞれ類似的に変換されます。ただし，rexponent は e^x を算出せず，32 ビット固定小数点数の符号ビット数を算出します。rabs は，固定小数点数の絶対値を算出し，アセンブラ命令 ABS にコンパイルされます。rabs の構造体は，すべての固定小数点データ型に対して，1 つの関数名の使用を許可しています（ANSI C の厳密なデータ型概念をクリアした場合）。しかし，型変換が長いため，ソース・コードはその可読性の大部分を失うことになります。したがって，rabs の代わりに rrabs マクロを使用することを推奨します。

6.5　リング・バッファとモジュロ・アドレシング・モード

リング・バッファはデータ値の配列です。配列の各データ値（エレメントと呼ばれる）は，その他の配列と同じように添え字やデータ・ポインタを介してアクセスすることができます。しかし，従来の C 配列と比較して，リング・バッファは次の 3 つの特長を備えています。

(1) リング・バッファは，そのサイズのため，ハードウエア特有のアラインメント制約を受けます。つまり，リング・バッファは，メモリのどこにでも配置可能という訳ではありません。したがって，関数中のローカル変数として，リング・バッファをデータ・スタック上に生成することは不可能です。これは，コンパイル時に関数スタック・フレームのアドレスが利用できないために起こります。また，リング・バッファはグローバル変数として宣言されなければなりません。リング・バッファは，キーワード__circ で示されます。

(2) incmod_x と incmod_y の 2 つのポインタ演算関数によって，リング・バッファ・ポインタを“サーキュラ型”に変更することができます。たとえば，リング・バッファ・データ・ポインタがバッファの最後のエレメント（上位アドレス）を示しており，適切な incmod 関数を使用してデータ・ポインタを 1 エレメント分インクリメントしている場合，データ・ポインタは関数呼び出しのあと最初（下位アドレス）のエレメントを示します。ただしこれはその他のアクセス形式にはあてはまりません。したがって，5 つのエレメントから成るリング・バッファ cb に対する添え字アクセス x＝cb［6］によって，2 番目のエレメントを回収しようとするのは間違いです。

(3) グローバル・リング・バッファ配列は（ほかのソース・ファイルで）external 宣言される場合があるため，C コンパイラは，バッファ・エレメントの“サーキュラ型”アクセスが可能となる前に，どうにかしてバッファのサイズ（エレメント数）を認識する必要があります。この目的のために，モジュロ・レジスタ DMX および DMY をそれぞれ初期化する関数 setmod_x と setmod_y を使用します。

図 6-7　リング・バッファ操作関数

```
void setmod_x (unsigned int dmx);
void setmod_y (unsigned int dmy);

__circ __X void * incmod_x (__circ __X void *cb, int inc);
__circ __Y void * incmod_y (__circ __Y void *cb, int inc);
```

図 6-7は，リング・バッファに関するコンパイラ関数の文法を際立たせるヘッダ・ファイル dspext.h の詳細を示しています（inc の許容範囲などその他の詳細に関しては第 14 章　DSP ライブラリの関数レファレンスを参照)。

incmod の構造体は，異なるデータ型のリング・バッファ用の 1 つの関数名を使用可能にしています（ANSI C の厳密なデータ型概念をクリアした場合)。

しかし，型変換が長いため，ソース・コードはその可読性の大部分を失うことになります。したがって，可能なっかぎり，図 6-8に示すマクロ circinc_x および circinc_y を使用することを強く推奨します。

マクロ circlength_x と circlength_y は，モジュロ・レジスタの負担を軽減します。モジュロ・レジスタを値 buffersize ∗sizeof（bufferelement）－1 に初期化しなければならないと常に意識する代わりに，データ型とエレメント数を適切な circlength マクロに挿入するだけでよいのです。

図 6-8　リング・バッファ操作マクロ

```
#define circlength_x(t,n) ¥
        (setmod_x ((n)*(unsigned int) sizeof(t) - 1))
#define circlength_y(t,n) ¥
        (setmod_y ((n)*(unsigned int) sizeof(t) - 1))

#define circinc_x(t,p,i)        ((__circ __X t *)¥
        incmod_x((__circ __X void *)(p),(i)*sizeof(t)))

#define circinc_y(t, p, i)      ((__circ __Y t *)¥
        incmod_y((__circ __Y void *)(p),(i)*sizeof(t)))

#if   __DEFAULTSPACE_X || __DEFAULTSPACE_XSER
#define circlength(t,n)         (circlength_x(t,n))
#define circinc(t,p,i)          (circinc_x(t,p,i))
#elif __DEFAULTSPACE_Y || __DEFAULTSPACE_YSER
#define circlength(t,n)         (circlength_y(t,n))
#define circinc(t,p,i)          (circinc_y(t,p,i))
#endif
```

circlength と circinc マクロは，ディフォールト空間リング・バッファの操作に利用され，適当なリング・バッファ操作関数に展開されます。すべてのリング・バッファ操作マクロおよび関数は，インクルード・ファイル dspext.h を介して利用することができます。

図 6-9　リング・バッファ使用例

```
__circ long     cb[5];

void store_data (long data)
{
        static __circ long *cps = cb;
        __circ long       *cp;

        cp = cps;
        circlength (long, 5);

        *cp = data;
        circinc (long, cp, 1);

        cps = cp;
}
```

図 6-9に，リング・バッファ cb を定期的に測定されるデータを格納するために使用する例を示します。スタティック・リング・データ・ポインタ cps は store_data 関数の呼び出し間に，データ・ポインタ cp の実際の位置を格納しています。

図 6-10　コンパイル後のリング・バッファ使用例

```
        PUBLIC  _store_data
cb_xdata XRAMSEG INTERNAL ALIGN AT 0
        PUBLIC  _cb
_cb:
        DW0,0,0,0,0,0,0,0
        DW0,0
xdata XRAMSEG INTERNAL
__L1b:
        DW_cb

store_data_code IMSEG
_store_data:
__L0:
        r1l = *__L1b:X;
        dp1 = r1l;
        dmx = 9;
        dn1 = 1;
        *dp1++ = r0h;
        *dp1%% = r0l;
        r0l = dp1;
        *__L1b:X = r0l;
        RET;
END
```

図 6-10のアセンブラ・ファイルは，circlength の呼び出しが命令 dmx＝9 に変換されることを示しています。これは，long 型の変数は，2 つの 16 ビット・メモリ・セルを必要とするためです。このアセンブラ・ファイルを詳細に分析すると，C コンパイラは，メモリ格納文*cp＝data と，続くポインタをインクリメントする circinc（long，cp，1）とを結合していることが分かります。これらの文は結合されて，アセンブラ命令 dn1＝1；*dp1＋＋＝r0h；*dpl%%＝r01；に変換されます。C コンパイラは，同じデータ・ポインタに circinc が呼び出される前にメモリ・アクセス（ストアまたはロード）を認識し，μPD77016 のアーキテクチャが提供するポスト・モジュロ・インデクス加算アドレシング・モードを使用して，同じ命令内でデータをアクセスし，データ・ポインタをインクリメントしています。C コンパイラは，次の条件が満たされたときのみこれらの命令を結合することができます。

（1）結合リング・バッファ操作は，μPD77016 のアーキテクチャの汎用レジスタ（40 ビット）に完全に収まるデータ型，つまり整数データ型の__fixed，long __fixed，および long __accum に限定されます。

（2）結合命令は，制御フローの同じ基本ブロックになければなりません。具体的には，メモリ・アクセスと circinc マクロ呼び出しの間には if，goto，またはループ文（for や while）があってはいけません。

（3）使用するリング・バッファ・ポインタはグローバル変数であってはいけません。

（4）リング・バッファ・ポインタにエイリアスを使用してはいけません。つまり，pp＝&circpointer といったリング・バッファ・ポインタのアドレスを使用することはできません。

（5）リング・データ・ポインタは，C コンパイラによってグローバル変数と同じように扱われるため，cps といったローカル・スタティック変数であってはいけません。
命令を対で結合する場合は，いつでもメモリ・アクセスの前に対応するインデクス・レジスタの割り当てが挿入されます（図 6-10 の dn1 参照）。

6.6　ビット反転アドレシング・モード

ビット反転アドレシングによって，メモリのデータの再配列が可能になります。このようにして並べられたデータは，高速フーリエ変換（FFT）応用の入出力に使用します。C ソース・コードからアクセス可能な，プレ・ビット反転およびポスト・インデクス加算アドレシング・モードで，ハードウエアを構築する文法を解説する前に，単純な再配列アルゴリズムを説明します。

（1）入力データが存在するメモリ領域の始めを示す入力データ・ポインタを設定します。

（2）再配列されたデータを格納するメモリ領域を示すデータ・ポインタを設定します（出力バッファ）。

（3）上記（2）のデータ・ポインタのビットを反転し，結果を出力データ・ポインタに格納します。この時点で出力データ・ポインタは意味のあるデータを示していないことに注意してください。このポインタは，アドレス計算にのみ使用されるため，仮想出力データ・ポインタと呼ばれます。

（4）入力データ・ポインタを介してデータをロードし，ビット反転出力データ・ポインタを介して出力バッファに格納します。

（5）入力データ・ポインタと仮想出力データ・ポインタを，定数値分だけインクリメントします。

（6）すべての入力データが処理されるまで，（4），（5）を繰り返します。

通常は，アセンブラのプログラマは，再配列データを格納するため選択アドレスから始まるデータ・セグメントを最初に反転します。これに対して，C コンパイラでは，リンカがメモリ領域の正確な位置を選択します。領域サイズの関係上，生成セグメントに対して C コンパイラはアラインメント制限のみを与えます。この制限は，リング・バッファの場合のそれとまったく同じものであるため，C コードの__circ キーワードで必要なグローバル・データ配列を定義することが可能となります。

アセンブラのプログラマは出力メモリ領域の開始アドレスが分かっているので，上記（3）を手動操作で行い，ビット反転開始アドレスを直接出力データ・ポインタに格納することができます。これに対して C コンパイラはこのような情報を持っていないため，ビット反転動作を行う関数 bitrev_x および bitrev_y が備えられています。しかしながら，μPD77016 のアーキテクチャはビット反転動作に対応していないため（ビット反転メモリ・アクセスのみ対応），これらの関数を呼び出すには，比較的時間がかかります（50 サイクル以上）。

図 6-11　ビット反転操作関数

```
__circ __X void * bitrev_x (__circ __X void *cb);
__circ __Y void * bitrev_y (__circ __Y void *cb);
```

図 6-11は，プレ・ビット反転およびポスト・インデクス加算アドレシング・モードに関する，すべてのコンパイラ関数の文法を際立たせるヘッダ・ファイル dspext.h の詳細を示しています。bitrev 関数の構造体により，異なるデータ型を保持しているメモリ配列に対して関数名を 1 つ使用することができます（ANSI C の厳密なデータ型概念をクリアした場合）。しかし，型変換が長いためソース・コードはその可読性を大部分失うことになります。このため，可能なかぎり図 6-12に示すマクロ reverse_x および reverse_y を使用することを推奨します。

図 6-12　ビット反転操作マクロ

```
#define reverse_x(t,p)¥
        ((__circ __X t *)      bitrev_x ((__circ __X void *)(p)))
#define reverse_y(t,p)¥
        ((__circ __Y t *)      bitrev_y ((__circ __Y void *)(p)))

#if   __DEFAULTSPACE_X || __DEFAULTSPACE_XSER
#define reverse(t,p)            (reverse_x(t,p))
#elif __DEFAULTSPACE_Y || __DEFAULTSPACE_YSER
#define reverse(t,p)            (reverse_y(t,p))
#endif
```

reverse マクロは，ディフォールト空間バッファの操作に使用され，適当なビット反転関数に展開されます。すべてのビット反転操作マクロと関数は，インクルード・ファイル dspext.h を介して使用することができます。

図 6-13　データ再配列例

```
__circ __fixed        cb[8];

void reorder_data (__fixed *pin)
{
       __circ __fixed *pout;
       int            i;

       pout = reverse (__fixed, cb);

       for (i=0; i < 8; ++i) {
               *(reverse (__fixed, pout)) = *pin;
               pin++;
               pout += 0x2000;
       }
}
```

図 6-13は，C 関数によるデータ再配列を示しています。pin は__fixed 型の 8 つのデータ値のベクタを示しています。再配列されたデータは，リング出力バッファ cb［8］に格納されます。すべてのインデクス（0～7）を表現するために，8 つの値のバッファ・サイズ（BUFSIZE）に，3 ビットが必要となります。したがって，入力バッファ・インデクス 1（バイナリ 001）は出力バッファ・インデクス 4（バイナリ 100）に対応し，また，入力インデクス 3（バイナリ 011）は出力インデクス 6（バイナリ 110）に対応します。μPD77016 のアーキテクチャは，16 ビットを反転するビット反転アドレシング・モードを備えています。仮想出力データ・ポインタに必要な定数インクリメントを得るために，希望する出力バッファ・インデクス 4（0x0004）を 16 ビットの数字として反転する必要があります。

このため，図 6-13では，8192（0x2000）の値が使用されています。その他のバッファ・サイズに対しても同様な配慮がなされるため，仮想出力データ・ポインタの必要なインクリメントは，通常，2^{16}/BUFSIZE として表現されます。

図 6-14　コンパイル後のデータ再配列例

```
        PUBLIC  _reorder_data
cb_xdata XRAMSEG INTERNAL ALIGN AT 0
        PUBLIC  _cb
_cb:
        DW0,0,0,0,0,0,0,0
EXTRN __reverse_x

reorder_data_code IMSEG
_reorder_data:
__L0:
        r3l = dp1;
        dp1 = _cb;
        dp2 = r3l;
        CALL __reverse_x;
        dn1 = 8192;
        LOOP 8 {
__L1:
        r0 = *dp2++;
        *!dp1## = r0h;
        };
        RET;
END
```

図 6-14では，reverse マクロが 2 回使用されています。1 回目は，リング・バッファのビットを反転し，2 回目は，メモリ・アクセス（ストア）とともに，マクロが使用されています。図 6-14のアセンブラ・コードは，C マクロ reverse に対する最初の呼び出しに対して，__reverse_x 呼び出しに直接コンパイルされていることを示しています（約 50 サイクル）。ストア命令 ＊（reverse（__fixed，pout））＝*pin の中でのマクロは，アセンブラ・コード上では，*!dp1## ＝r0h と読まれます。ビット反転メモリ・アクセスと，そのあとのポインタ・インクリメント pout＋＝0x2000 は結合された命令であり，*!dp1##＝r0h の前にアセンブラ命令 dn1＝8192 にコンパイルされます。

C コンパイラは，次の条件が満たされたときにのみこれらの命令を結合します（6.5　**リング・バッファとモジュロ・アドレシング・モード**を参照）。

（1）ビット反転アドレシングは，16 ビット・データ型である__fixed と，long および unsigned loop を除くすべての整数データ型に限定されます。

（2）結合命令は，同じ基本ブロックの中に存在しなければいけません。

（3）リング・データ・ポインタは，グローバルであってはいけません。

（4）リング・データ・ポインタに，エイリアスを使用してはいけません。

（5）リング・データ・ポインタは，ローカル・スタティック変数であってはいけません。

しかし，ポインタ・インクリメントのないビット反転メモリ・アクセスは，プレ・ビット反転およびポスト・インデクス加算アドレシング・モード・メモリ・アクセスにコンパイルされることに注意してください。そのあと，対応するインデクス・レジスタはゼロに初期化されます。

6.7　ループ処理

μPD77016のアーキテクチャは，最大ネスティング・レベル4のハードウエア・ループ・スタックを備えています。Cコンパイラは，ユーザがいくつかの単純なプログラミング規則を守るかぎり，このハードウエアに対するアセンブラ・コードを生成します。プログラミング規則を守らなければ，Cコード・ループは，通常のアセンブラの条件付き命令およびジャンプ命令にコンパイルされます。

図6-15　典型的Cコード（ループ）

```
for (icount=istart; icount < iend; ++icount) {
... loop corps ...
}
```

図6-15の典型的なCコード（ループ）は，次に説明する考え方を明らかにしています。icountはループの誘導変数で，istartはループ開始式の例です。同様にiendもループ終了式の例としています。次の条件が満たされると，Cコード・ループはハードウエア・ループにコンパイルされます。

（1）ループには，定数インクリメントを持つ誘導変数が存在すること。

（2）ループの繰り返し回数は，32767回を越えないこと。この条件は，次のいずれかで満たされます。

- （iend-istart）/icount＜32767を持つ定数ループ開始式およびループ終了式istartおよびiend
- 0以上の定数ループ開始式およびsigned integerループ終了式
- 符号付きまたは符号なし整数型誘導変数，および2つ以上の誘導変数を持つ絶対値を含むループ開始式およびループ終了式

（3）ループには，break文のような追加出口がないこと。

（4）最大ループ・ネスティング・レベルを越えないこと。

（5）ループ開始式およびループ終了式に使用される変数や，誘導変数にエイリアスが使用されていないこと。つまり，ループが存在するC関数中のこれらの変数を示すポインタは存在しないこと。

（6）ループは，表6-6に示されるコンパイラ関数以外の関数呼び出しを含まないこと。

表 6-6　ハードウエア・ループへのコンパイルを可能にする関数

	コンパイラ関数	説　　明
	abs	int の絶対値
	labs	long の絶対値
	rabs	固定小数点値の絶対値
	rclip	long __accum から long __fixed へのクリッピング
	rround	long __accum から__fixed へのクリッピングおよび丸め（切り捨て）
	rexponent	long __fixed 値の符号ビット数
	fixed_int	__fixed から int へのビット・パターン変換
	int_fixed	int から__fixed へのビット・パターン変換
	lfixed_long	long __fixed から long へのビット・パターン変換
	long_lfixed	long から__fixed へのビット・パターン変換
	laccum_read32	long __accum 値からの 32 ビット・パターンの抽出
	setmod_x	モジュロ・レジスタ DMX の設定
	setmod_y	モジュロ・レジスタ DMY の設定
	incmod_x	X メモリを示すリング・バッファ・ポインタをインクリメント
	incmod_y	Y メモリを示すリング・バッファ・ポインタをインクリメント
	bitrev_x	X メモリを示すリング・バッファ・ポインタのビットを反転
	bitrev_y	Y メモリを示すリング・バッファ・ポインタのビットを反転
★	set_eir	eir レジスタを設定する
★	get_eir	eir レジスタ値を取得する
★	set_sr	sr レジスタを設定する
★	get_sr	sr レジスタ値を取得する

ハードウエア・ループ命令の使用は，C コード・ループに限定されたものではなく，どのような種類の C ループ文でも，これらの条件が満たされていれば，ハードウエア・ループに変換することができることに注意してください。

★ 6.7.1 明示的ハードウエア・ループ制御

ハードウエア・ループの生成は，ループ制御オプションによって明示的に制御可能です。ループ制御オプションは，3 つの異なる範囲で指定できます。

(1) ループ本体の直前でプラグマを使ったループ範囲
ループ制御オプションは，1 つのループと，そのループ内のネストされたすべてのループに適用されます。

(2) 関数定義の直前でプラグマを使った関数範囲
ループ制御オプションは，関数のすべてのループに適用されます。

(3) コマンド・ライン・オプションを使ったモジュール範囲
ループ制御オプションは，コンパイル・モジュールのすべてのループに適用されます。

表 6-7は，ループ制御オプションの意味を説明しています。これらの説明は，ループ範囲プラグマ，関数範囲プラグマ，およびコマンド・ライン・オプションに有効であることに注意してください。コマンド・ライン・オプションと関数範囲プラグマには，loop プリフィクスが必要です。ループ範囲プラグマには，loop プリフィクスを記述する必要はありません。図 6-16は，ループおよび関数範囲のループ制御プラグマの構文を定義しています。

表 6-7　ループ制御オプション

ループ制御オプション	意　味
[loop] callsok	ループ本体で呼び出された関数がハードウエア･ループを使わないことを C コンパイラに知らせます。したがって，C コンパイラは，ループ本体での関数呼び出しの場合でもハードウエア・ループを生成します。
[loop] callsnotok	callsok オプションをキャンセルします。
[loop] countok	ループ数が 0-32767 回であることを C コンパイラに知らせます。したがって，ループ開始表現も定数でない場合，またはループ終了表現の型が符号付き整数でない場合には，定数でないループ終了表現に対してハードウエア・ループが生成されます。ループが 32767 回以上繰り返されることは非常にまれであるため，このオプションは頻繁に使われます。このオプションを使うと，countnice オプションがキャンセルされます。
[loop] countnice	ループ数が 1-32767 回であることを C コンパイラに知らせます。結果は一般に-countok オプションと同じです。また，保護的な条件付き分岐は生成されません。これによって，ハードウエア・ループ周辺のコードが最適化されます。C 言語では，ループ全体をスキップするためにループ条件を使用することがよくあるため，このオプションを使用するときは注意してください。したがって，対応するループが 1 回以上実行されることを確認してから，このオプションを使ってください。
[loop] countnotok	countok または countnice オプションをキャンセルします。
[loop] depth*n*	生成されるハードウエア・ループのネスティング深さを n に制限します。引数 n は，0 から 4 までの数値です。

図 6-16　ループ制御プラグマの構文

```
loopopt =  [callsok] [callsnotok] [countok] [countnice] [countnotok]
           [depthn]
#pragma loopopt
#pragma loopopt on
#pragma loopopt off

funopt =   [loopcallsok] [loopcallsnotok] [loopcountok]
           [loopcountnice] [loopcountnotok] [loopdepthn]

#pragma funopt
#pragma funopt on
#pragma funopt off
```

1 つ以上のループまたは関数でループ制御オプションを定義するには，on/off 構文を使用できます。ループ制御オプションは，on および off プラグマの間にあるすべてのループに適用されます。on/off 構文では，プラグマ当たり 1 つのオプションのみを指定できることに注意してください。

表 6-8は，コマンド・ライン・オプションとして適用可能なすべてのモジュール範囲ループ制御オプションを示しています。

表 6-8　モジュール範囲ループ制御オプション

-loopcallsok	コンパイル・モジュールのすべてのループに callsok オプションを設定。
-loopcountok	コンパイル・モジュールのすべてのループに countok オプションを設定。
-loopcountnice	コンパイル・モジュールのすべてのループに countnice オプションを設定。
-loopdepthn	コンパイル・モジュールのすべてのループの n に depthn オプションを設定。

例

```
cc77016 -loopdepth0
```

利用可能なループ・ネストを 0 に制限することで，ハードウエア・ループの生成を無効にします。

```
#pragma loopcountok
void f() {
    for(i =a;i <b;i++) {
     /* loop 1 */
    }

   #pragma countnice
   for(i =c;i <d;i++) {
     /* loop 2 */
   }
}
```

ループ2を除く関数f()のすべてのループに，loopcountokオプションが設定されています。countniceプラグマは，ループ2のloopcountniceオプションを設定しています。Cコンパイラは両者のループにハードウエア・ループを生成し，ループ2の場合は保護を省略します。

★ 6.8　関数のインライン

ディフォールトでは， Cコンパイラは関数呼び出しのオーバヘッドを最小化するために，自動的に関数インラインを実施します。関数インラインの欠点は，コード・サイズの増大です。したがって，Cコンパイラは，ヒューリスティックにインラインを行う関数を選択します。関数は，命令文の数が特定の制限を越えず，すべての非静的ローカル変数の合計のサイズが1024ビット（64ワード）を越えない場合にインラインされます。命令文制限は，-Osizeオプションの適用に依存します。-Osizeが与えられた場合は，命令文制限は0となり，関数はインラインされません。-Osizeが与えられない場合は，命令文制限は42になります。どちらの最適化モードでも，ディフォールトの命令文制限は，-inlinesizeオプションを使って変更することができます。42より大きい値を指定すると，インラインされる関数が多くなります。

1箇所のみから呼び出される静的関数は，インライン後に削除することができるため，命令文およびローカル・サイズの制限をチェックせずにインラインされます。したがって，この場合はコードのサイズは増大しません。

-inlinesizeオプションに加えて，Cコンパイラは特定の関数の明示的なインライン制御を提供しています。-inline functionNameオプションと-unline functionNameオプションを使用すると，関数のインラインを明示的に有効または無効にできます。オプションの引数では，影響される関数の名前を指定します。どちらのオプションも，複数回適用できます。#pragma inlineと#pragma unlineの2つのインライン制御プラグマによって，関数のインラインを制御するもう1つのメカニズムが提供されています。プラグマは，関数定義の直前に置く必要があります。inlineプラグマは，それに続く関数を強制的にインラインします。これは，-inline functionNameコンパイラ・オプションと同等です。unlineプラグマは，それに続く関数を無効にし，-unline functionNameコンパイラ・オプションと同等です。

関数インライン選択メカニズムの順位は，次のとおりです。関数にインライン制御コマンド・ライン・オプションが指定されている場合は，関数はオプションに従ってインラインされたりされなかったりします。また，関数の前にインライン制御プラグマが書かれている場合は，関数はプラグマに従ってインラインされたりされなかったりします。また，関数が静的であり，1箇所のみから呼び出される場合は，関数はインラインされます。または，自動関数インライン選択が実行されます。すなわち，命令文制限とローカル・サイズが決定され，命令文制限が選択された制限を越えず，かつローカル・サイズが1024未満の場合に，関数はインラインされます。-inlinesizeオプションが指定されている場合は，選択された命令文制限がオプションの引数に設定されます。または，-Osizeオプションが指定されている場合は，命令文制限は0に設定されます。それ以外の場合は，命令文制限は42です。

インライン制御オプション＞インライン制御プラグマ＞静的関数のインライン＞自動インライン選択

例

```
cc77016 -inlinesize 0 -inline mul_emul -inline add_emul
```

関数 mul_emul と add_emul 以外のインラインを無効にします。

```
cc77016 -Osize -inlinesize 10
```

サイズの最適化を行うときに，小規模な関数のインラインを有効にします。

```
#pragma unline
void debug_print(int x){
    printf("%d¥n", x);
}
```

関数 debug_print のインラインを無効にします。

★ 6.9　割り込みサポート

C コンパイラでは，C ソース・コードでプレーン割り込み関数を作成できます。コンパイラは，その割り込みを処理するために必要なすべてのアセンブラ・コードを生成します。割り込み関数は，次の点で通常の C 関数と異なります。

- 割り込み関数を，その他の関数から呼び出さないでください。
- 割り込み関数のアドレスを取得することはできません。
- 割り込み関数は，戻り値も引数もありません。
- 関数の終わりでは，RET 命令の代わりに RETI 命令が生成されます。
- すべてのレジスタは，関数の入口でスタックに保管され，関数の出口で復元されます（特に指定されない場合は以降を参照）。
- 最大 3 つのハードウエア・スタック・レベルが占有されます。
- フレーム・ポインタ dp0 と dp4 は，関数の入口と出口で修正されます。
- 割り込み関数の最初の 4 つの命令は，特定の割り込みベクタに割り当てられたセグメントに置かれます。これらの命令には，残りの割り込み関数本体へのジャンプが含まれます。

注意　C ソースで割り込み関数を記述するには，SP77210 に含まれるリンカが必要です。

6.9.1　割り込みプラグマ

関数定義の直前に割り込みプラグマが置かれた場合は，関数は割り込み関数として指定されます。図 6-17は，割り込みプラグマの構文を定義しています。

割り込みプラグマは，2 種類のオプションの引数を受け入れます。

- vector は，割り込み関数で処理される割り込みベクタを指定します。割り込みベクタは，割り込み要因名もしくは，ベクタ・アドレス（0x210-0x23c）によって指定することができます。1 つ以上の vector 引数は，同じ割り込み関数によって複数の割り込みベクタが処理されることを意味します。
- scrregs は，割り込み関数のスクラッチ・レジスタを指定します。スクラッチ・レジスタ引数の任意の組み合わせを指定できます。

図 6-17　割り込みプラグマの構文

```
vector = [INT1][INT2][INT3][INT4][SI1][SO1][SI2][SO2][HI][HO][address]
scrregs = [scrr5][scrr6][scrr7][scrdp2][scrdp6][scr*]

#pragma interrupt [vector] [scrregs]
```

6.9.2　生成コード・セグメント

割り込み関数は，その他すべての関数と同様に，別のリロケータブル・コード・セグメントに割り当てられます。指定されたそれぞれの vector プラグマ引数に対して，対応する割り込みベクタで絶対コード・セグメントが生成されます。これには，割り込み関数の最初の 3 命令と，リロケータブル・コード・セグメントにある関数の残りの部分への JMP 命令が含まれます。割り込み関数が最大 4 命令の場合は，リロケータブル・コード・セグメントは生成されません。vector 引数が指定されない場合は，割り込みベクタ・テーブルに絶対コード・セグメントが生成されません。リロケータブル・コード・セグメントには，割り込み関数全体が含まれています。この場合は，ユーザは割り込みベクタで必要なアセンブラ・コードを提供する必要があります。割り込み関数はそのほかの場所では使用されないため，割り込み関数にはパブリック・ラベルは生成されません。割り込みベクタ引数が指定されない場合は例外です。この場合は，割り込み関数はアセンブラ・コードからアクセス可能でなければならないので，関数ラベルが生成されます。生成されたラベルは，int_functionname 形式になります。

6.9.3　レジスタの使用

ディフォールトでは，すべてのレジスタはメイン・プログラムに使われ，各割り込み関数はすべてのレジスタ値を保存する必要があります。頻繁に割り込みが行われる場合は，割り込み関数用に一部のレジスタを予約すると便利です。これによって，割り込み関数でのレジスタの保管と復元のオーバヘッドが低減できます。予約は，メイン・プログラム用のレジスタを無効にし，これらのレジスタを割り込み関数用のスクラッチ・レジスタとして指定することで行えます。表 6-9は，コンパイル・モジュールのすべての関数用にさまざまなレジスタを無効にするコマンド・ライン・オプションを示しています。これらのオプションは，割り込みプラグマで適切なスクラッチ・レジスタ・オプションによって有効にしないかぎり，割り込み関数用にもレジスタを無効にすることに注意してください。

表 6-9　レジスタ無効コマンド・ライン・オプション

-nor5	レジスタ r5 を無効にする。
-nor6	レジスタ r6 を無効にする。
-nor7	レジスタ r7 を無効にする。
-nodp2	レジスタ dp2 と dn2 を無効にする。
-nodp6	レジスタ dp6 と dn6 を無効にする。

表 6-10は，割り込みプラグマのすべての可能なスクラッチ・レジスタ引数を定義しています。

表 6-10　レジスタ許可プラグマ・オプション

scrr5	レジスタ r5 が使われ，保存されない。
scrr6	レジスタ r6 が使われ，保存されない。
scrr7	レジスタ r7 が使われ，保存されない。
scrdp2	レジスタ dp2 と dn2 が使われ，保存されない。
scrdp6	レジスタ dp6 と dn6 が使われ，保存されない。
scr*	すべての無効なレジスタが使われ，保存されない。

注意　**割り込み関数のスクラッチ・レジスタは保存されないので，対応するコンパイラ・コマンド・ライン・オプションを使用して，メイン・プログラム用に無効にする必要があります。割り込み関数用にすべての無効なレジスタをスクラッチ・レジスタとして指定するもっとも安全な方法は，scr*プラグマ・オプションを使用することです。レジスタ dmx，dmy，sr，および eir は，すべての場合で保存されません。**

6.9.4　スタックの使用

割り込み関数は，dp0 および dp4 スタック・ポインタ・レジスタで提供されるスタック・フレームを使います。このため，割り込み関数用に別のスタックが必要ありません。関数の実行中に，スタック・ポインタは関数スタック・フレーム内の任意の位置を指しています。割り込みが認められると，割り込み関数が，割り込まれた関数のスタック・フレームを上書きしないように，スタック・ポインタを修正する必要があります。割り込み関数の最初では，スタック・ポインタはプログラムの最大関数フレーム・サイズ分だけデクリメントされます。割り込み関数の出口では，スタック・ポインタはプログラムの最大関数フレーム・サイズ分だけインクリメントされます。割り込み関数がそのスタックを使わない場合は，スタック・ポインタの修正は行われません。C コンパイラは，各コンパイル・モジュールで，モジュールの最大関数フレーム・サイズを保持する__DP0_FRAMESIZE と__DP4_FRAMESIZE の 2 つのシンボルを生成します。リンカはリンクされたすべてのモジュールの最大値を計算して，__MAX_DP0_FRAMESIZE，__MAX_DP4_FRAMESIZE，__NEG_MAX_DP0_FRAMESIZE，および__NEG_MAX_DP4_FRAMESIZE の 4 つのシンボルを生成します。これらのシンボルは，プログラム全体の最大フレーム・サイズの値と，その負の値を保持します。

6.9.5　割り込み制御レジスタのアクセス

μPD77016 アーキテクチャは，割り込み制御用に，sr と eir の 2 つのレジスタを提供しています。これらのレジスタは，グローバル・レジスタとみなされます。これは，明示的に読み取りと書き込みを行うことができ，関数（割り込み関数を含む）によって保存や復元が行われないことを意味しています。図 6-18は，割り込み制御レジスタへアクセスする関数の構文を示しています。ヘッダ・ファイル dspext.h は，これらの関数を使用するために必要です。

図 6-18　割り込み制御レジスタ・アクセス関数

```
unsigned get_sr (void);
void set_sr (unsigned x);
unsigned get_eir (void);
void set_eir (unsigned x);
```

図 6-19では，割り込み INT1 と INT2 のための割り込み関数を定義しています。アドレス 0x210 と 0x214 で，2 つの絶対コード・セグメントが生成されます。これらのコード・セグメントには，割り込み関数の最初の命令と，残りの関数 inthandler()へのジャンプが含まれています。この割り込み関数は，メイン・プログラムのすべての無効なレジスタを使っており，これによって，レジスタの保存のオーバヘッドを低減しています。このプログラムがコマンド・ライン・オプション-nor6 と-nor7 を使ってコンパイルされると，割り込み関数は r6 と r7 をスクラッチ・レジスタとして使います。割り込み関数の最初の命令は，割り込みのネスティングを可能にするため，eir レジスタのビット 15 をクリアします。

図 6-19　割り込み関数定義

```
#include <dspext.h>

#pragma interrupt INT1 INT2 scr*
void inthandler (void){
   set_eir (get_eir() & 0x8000);
   /* ...*/
}
```

第7章　C コンパイラ 名称規則

C コンパイラでは，1 つのソース・ファイルの中で，C とアセンブラのコードを混在させることはできません。しかし，一方でアセンブラ関数（ほかのファイルに定義されている）を C コードから呼び出し，他方で C 関数をアセンブラ・コードから呼び出すことは可能です。したがって，ユーザは，C コンパイラのパラメータ譲渡規則，つまり呼び出し規則を守る必要があります。C コードで定義されたグローバル変数をアクセスする場合，発生したアセンブラ・コード内のデータ・オブジェクト名を知る必要があります。本章では，C コードとアセンブラ・コード間でのデータ転送に必要な事項を説明します。

7.1　呼び出し規則

7.1.1　関数パラメータの受け渡し

関数パラメータや返り値は，レジスタやスタックを介して渡されます。

- 関数が引数の可変数を持たず，かつ関数宣言もしくは関数定義がソース・コードに存在する場合，汎用レジスタ（40 ビット）に収まる最初のパラメータは，レジスタを介して渡されます。
- ポインタを除く最初の 3 つのパラメータは，r0，r1，r2 の順で渡されます。16 ビット・データは，レジスタの下位ワードを介して渡され，32 ビット・データは，レジスタの下位，上位ワードを介して渡されます。
- X メモリを示すポインタである最初のパラメータは，dp1 を介して渡されます。
- Y メモリを示すポインタである最初のパラメータは，dp5 を介して渡されます。
- その他のパラメータはすべてスタックに渡されます。
- 複合型のパラメータ（構造体や共有体）はすべてスタックに渡されます。
- ★ 関数が引数の可変数を持っている場合（printf など），すべてのパラメータはスタックに渡されます。
- パラメータがスタックに渡された場合，メモリ空間修飾子を持っていないかぎり，パラメータはディフォールト・メモリ・スタックに置かれます。この場合，パラメータはメモリ空間修飾子のメモリ空間内のスタックに渡されます。
- 返り値がポインタでなく，かつ汎用レジスタ（40 ビット）に適合した場合，返り値は r0 に渡されます。
- 返り値が X メモリを示すポインタである場合，dp1 に渡されます。
- 返り値が Y メモリを示すポインタである場合，dp5 に渡されます。
- 関数の返り値が汎用レジスタに収まらない場合は，返り値は，ディフォールト・メモリ・スタックに渡されます。dp1（または dp5）は，返り値のアドレスを保持します。

7.1.2　レジスタ・セーブ

図 7-1　セーブされるレジスタ

```
r5, r6, r7
dp3, dp7, (dp0, dp4)
dn3, dn7
```

図 7-1は C コンパイラのセーブされるレジスタを示しています。関数内でこれらのレジスタが使用されると，C コンパイラは，関数が実行される前にレジスタをセーブするコードで関数を囲み，関数の実行後，レジスタを再ロードします。ただし，スタック・ポインタ・レジスタ dp0 と dp4 は例外で，これらのレジスタは，関数の入り口で C コンパイラによってセーブされたり，関数の出口でリストアされることはありません。しかし，これらのレジスタが，ユーザが記述した C コードから呼び出されるアセンブラ関数で使用される場合は，C コードから呼び出しセーブまたはリストアする必要があります。最も単純な方法（小さなアセンブラ関数に適しています）は，セーブされたレジスタ（スタック・ポインタ・レジスタを含む）を使用しないことです。

表 7-1　呼び出し規則のまとめ

関数引数レジスタ		r0, r1, r2, dp1, dp5
関数返り値レジスタ		r0, dp1, dp5
不特定用途のレジスタ（スクラッチ・レジスタ）	汎用レジスタ	r0, r1, r2, r3, r4
	ポインタ・レジスタ	dp1, dp2, dp5, dp6
★	インデクス・レジスタ	dn0, dn1, dn2, dn4, dn5, dn6
セーブされたレジスタ	汎用レジスタ	r5, r6, r7
	ポインタ・レジスタ	dp3, dp7, (dp0, dp4)
	インデクス・レジスタ	dn3, dn7
スタック・ポインタ		dp0, dp4
★ グローバル・レジスタ		dmx, dmy, sr, eir

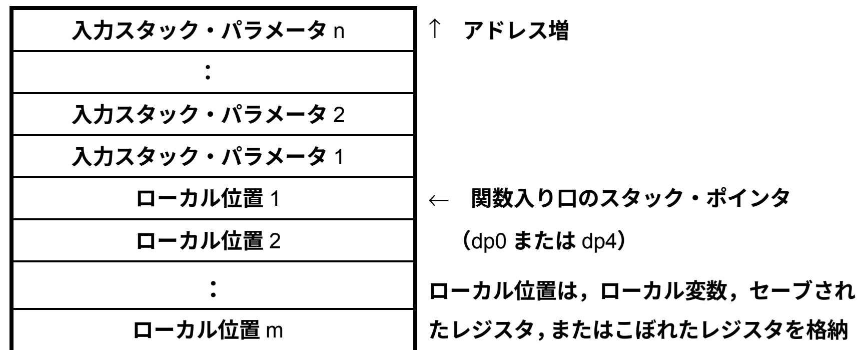
7.2　スタック・レイアウト

関数パラメータがスタックを介して渡された場合，ユーザは C コンパイラのスタック構成について詳しく知っている必要があります。

コマンド行のオプションによっては，C コンパイラは 1 つまたは 2 つのデータ・スタックを使用します。

データ・ポインタ・レジスタ dp0 は，X スタックのスタック・ポインタとして機能し，dp4 は Y スタックのスタック・ポインタとして機能します。しかし，ユーザのアセンブラ関数が 1 つのスタックしか使用しない場合でも，他方の未使用スタック・ポインタを，セーブされたレジスタとして扱う必要があります。したがって，未使用スタック・ポインタは関数中で変更することはできません（あるいは，このポインタの使用が不可避であるならば，セーブまたはリストアを正しく行う必要があります）。図 7-2に，X，Y の両メモリ空間に適応するスタック・レイアウトを示します。

図 7-2　スタック・レイアウト



メモリのアドレス単位が 16 ビットであるため，long __accum（40 ビット）や long（32 ビット）といったデータ型は，複数のメモリ位置に格納されることになります。この場合，最上位のワードは最下位アドレスに格納されるため，たとえば，アドレス 7 の long __accum は，アドレス 9 に最下位ワード（最下位の 16 ビット）が格納され，最上位の 8 ビットがアドレス 7 に格納されることになります。

7.3　グローバル・オブジェクト

アセンブラ・コード内においては，グローバル非スタティック・オブジェクト（グローバル変数や関数）の前には下線が置かれます。図 7-3は関数とグローバル変数の宣言，および対応するアセンブラ・レーベルを示しています。

図 7-3　グローバル変数の宣言

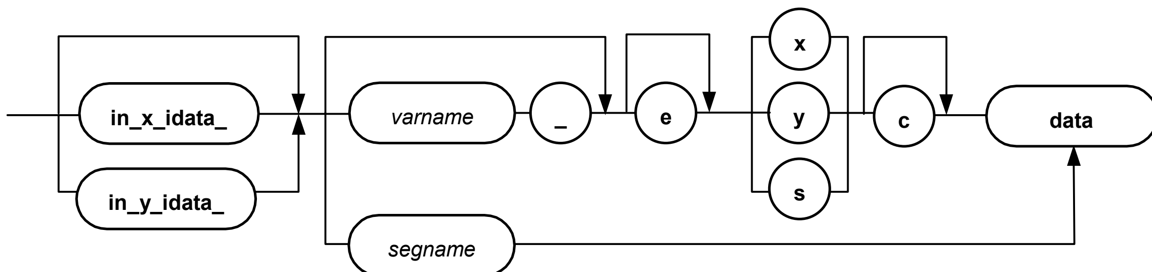
C コード	アセンブラ・レーベル
`void func(void);`	`_func:`
`int glob;`	`_glob:`

★ 7.4　変数割り当てとセグメント名

C コンパイラは，すべての変数を自動的に適切なデータ・セグメントに割り当て，すべての関数をコード・セグメントに割り当てます。しかし，ユーザは，コンパイラのディフォールトのセグメント配置動作を置き換えたり，リンカにセグメント・オプションを指定することで，コンパイル後に一部のデータまたはコード・セグメントをほかのメモリ領域にシフトしたい場合があります。

図 7-4によると，すべてのディフォールト・データ・セグメント名は data という単語で終わります。これに先行するオプション文字 c は，ROM セグメントを示しています（この文字がない場合は，RAM セグメントです）。次の文字（x, y, s）は，メモリの種類を反映しています（X，Y，または共有）。オプションの文字 e は，外部メモリに指定されたセグメントを示します。-sepsegs コンパイラ・オプションが指定される場合は，C コンパイラはすべてのグローバル・データに別のセグメントを割り当てます。この場合は，異なるセグメントは，変数名とアンダスコアを含むプリフィクスにおいて異なります。varname には先行するプリフィクスが含まれないことに注意してください。これは，C コードの変数名と完全に同一です。必要な位置合わせの条件により，すべてのサーキュラ・バッファは，独自のデータ・セグメントを持つ必要があります。-sepsegs コンパイラ・オプションがない場合でもサーキュラ・バッファ・セグメント名が C コードのバッファ名から始まるのはこのためです。いくつかのセグメント・プラグマを使って，データ・セグメント命名規則を置き換えることができます。図 7-4にユーザ定義のセグメント識別子 segname のバイパス分岐が含まれるのはこのためです。

図 7-4　データ・セグメント名の構文



-boot コンパイラ・オプションを使うと，グローバルおよびローカルな静的変数のセグメント名に，プリフィクスが追加されます。in_x_idata および in_y_idata は，そのようなすべてのプリフィクス付きセグメントを INID RAM 初期化セグメント・クラスに集めるために，リンカによって使われます（詳細については WB77016 **ユーザーズ・マニュアル**を参照）。

グローバルおよびローカルな静的変数の配置は，いくつかの簡単な規則に従います。

（1）メモリ修飾子のない変数は，ディフォールト・メモリ（-defy または-defser コンパイラ・オプションによって変更されないかぎり，内部 X メモリ）に置かれます。
（2）シリアル・アクセス・メモリ修飾子（__YSER など）を持つ変数は，外部メモリに置かれます。
（3）const 修飾子を持つ変数は，ROM に置かれます（-introm または-extrom コンパイラ・オプションにより使用可能な場合）。
（4）__circ 修飾子を持つ各変数は，別個のデータ・セグメントに置かれます。

このようなディフォールト配置とディフォールト・セグメント命名は，図 7-5のセグメント・プラグマを使うことで（部分的に）置き換えることができます。変数宣言の直前にセグメント・プラグマが記述されると，グローバ

ルまたはローカルな静的変数には，特別なセグメント配置情報を割り当てられます。

図 7-5　セグメント・プラグマの構文

```
attr = [intern] [align] [rom]
#pragma separate segment segname [attr]
#pragma separate in segname
#pragma separate attr
#pragma intern
#pragma intern on
#pragma intern off
```

最初の，もっとも一般的なタイプのプラグマを使うと，ユーザは，セグメント識別子 segname とセグメント配置属性を指定できます。ユーザ定義のセグメントに 1 つ以上の変数が置かれる場合は，2 番目のプラグマ・タイプによって，ショートカットの表記方法が使えます。3 番目のプラグマ・タイプは，ディフォールトのセグメント命名を置き換えませんが,1 つ以上のセグメント属性を定義します。#pragma intern は,#pragma separate intern のショートカットです。on/off セグメント・プラグマは，#pragma intern の有効性を後続の変数宣言に拡張します。#pragma intern on と#pragma intern off の間にあるすべてのグローバルまたはローカルな静的変数宣言は，内部メモリに置かれます。

ただし，希望したセグメント・プラグマ属性を満たすことが常に可能なわけではありません。たとえば，プラグマが rom セグメント属性を指定し，-introm または-extrom コマンド・ライン・オプションを使わずに C コンパイラが呼び出された場合は，使用可能な ROM はありません。この場合は，コンパイラは警告を発行し（警告オプションが設定されている場合），プラグマを無視します。

ディフォールト・セグメント配置の置き換えは，実行時に失敗するアセンブラ・コードを生成する可能性があることに注意してください。C コード宣言 int x=5 の前に rom セグメント属性を指定するプラグマを仮定してください。さらに，C コンパイラが-introm および-extrom コマンド・ライン・オプションで呼び出されたと仮定してください。C コンパイラは変数 x を内部 ROM に置き，この変数に対するすべての可能な割り当てを無視します。ユーザは，割り当て（初期化を除く）が行われないことを確認してください。

図 7-6は可能な変数宣言と定義を示しています。

コード・セグメント名は，簡単な構文 funcname_code に従います。funcname は，C コードでの関数名を示しています。データ・セグメント名と同様に，funcname には直前のアンダスコアは含まれません。

図 7-6　可能な変数宣言と定義

```
#pragma separate intern*
__XSER int a; /* a placed in internal segment xdata */
__XSER int b; /* b placed in external segment exdata */

/* necessary commandline option: -introm */
#pragma separate intern rom
__YSER long c = 3L; /* c placed in internal segment ycdata */

#pragma separate segment myseg
int d[10]; /* d placed in internal segment myseg */

#pragma separate in myseg
__fixed e; /* e placed in internal segment myseg*/

#pragma intern on
__XSER e, f, g; /* e, f, g placed in segment xdata */
#pragma intern off
```

第8章　スタートアップ・コードの構成

スタートアップ・コードは，すべてのユーザ・プログラムに必要な環境を構成するものです。スタートアップ・アセンブラ・コード（c.asm）は，さまざまなハードウエアおよびユーザのニーズに適応することができます。すべてのユーザ・プログラム・モジュールには，アセンブルされた c.rel ファイルをリンクする必要があります。

プロローグ部では，スタートアップ・コードはスタックとヒープを割り付け，ブート・サポートを行い（**第 9 章　ブート・サポート**を参照），文字出力を許可します（**第 10 章　文字出力**を参照）。そのあと，ユーザのメイン関数が呼び出されます。エピローグ部では，stdout が閉じられ，プログラム実行が終了します。さまざまなハードウエアおよびユーザのニーズを満たすためには，通常は，スタートアップ・アセンブラ・コード（図 8-1　**スタートアップ・コードの構成部**を参照）の構成部にあるプリプロセッサ・マクロの値を許可または禁止，あるいは変更するだけで十分です。

BOOT：	ブート時に，グローバルおよびローカル・スタティック C コード変数を初期化するアセンブラ・コードを起動します。
BOOT_FROM_Y：	Y メモリに常駐するブート ROM を定義します。
REBOOT：	グローバルおよびローカル・スタティック変数の初期化後，インストラクション・メモリの再ブートを行ないます。
HSM_SUPPORT：	HSM77016 の未定義値エラーを回避するため，レジスタやスタックをクリアします。
EXT_INSTMEM：	外部インストラクション・メモリを宣言します。
IWTR_CYCLES：	インストラクション・メモリのウエイト・サイクル数を定義します。
EXT_DATAMEM：	外部データ・メモリを宣言します。
DWTR_CYCLES：	データ・メモリのウエイト・サイクル数を定義します。
__DEFAULTSPACE_space：	ディフォールト・メモリ空間を定義します。
USE_spaceSTACK：	選択されたメモリにスタックを割り付けます。
space_STACKSIZE：	スタック・メモリ・セルの最大数を定義します。
USE_HEAP：	プログラム・ヒープを割り付けます（malloc, free 用）。
HEAPSIZE：	ヒープ・メモリ・セルの最大数を定義します。
USE_STDIO：	文字出力を許可します。

C コンパイラは，スタートアップ・コードの変更には気付かないことに注意してください。したがって，スタートアップ・コードと C コンパイラ・オプションに不一致が発生したとしても，それをユーザに知らせるツールは存在しません（**第 5 章　C コンパイラとオプション**を参照してください)。

次に，典型的な 2 つの例を示します。

- スタートアップ・コード中のメモリ空間を Y から X に変更した場合，対応する C コンパイラ・オプション-defy の使用が必須となります。このオプションを使用しないと，たとえば，ヒープがスタートアップ・コードによって Y メモリに割り付けられますが，C コンパイラで生成された各コードは X データ・メモリの割り付けられていないメモリ領域を使用してしまいます。
- スタートアップ・コードにたった 1 つのスタックを予約すると，C コンパイラを起動するたびに C コンパイラ・オプション-singlestack で，対応するスタックを定義する必要があります。このオプションを使用しない場合，スタートアップ・コードが 1 つのスタックのためのメモリを予約しているのにもかかわらず，C コンパイラは 2 つのスタックを使用するコードを生成してしまいます。

注意　スタートアップ・コード構成部のパラメータを適切に変更していない場合，構成部に続くアセンブラ・コードは書き換えられてしまう場合があります。

図 8-1　スタートアップ・コードの構成部（1/3）

```
/****************************************************************
 * The startup code  can be configured with following defines.
 ****************************************************************/

/*
 * Define BOOT to use reset boot support. BOOT invokes assembler
 * code that initializes all global and local static C-code
 * variables at boot time. Default source of the boot data (ROM)
 * is X memory. Use BOOT_FROM_Y, if the boot data source is
 * situated in Y memory.
 */

/*
#define BOOT
#define BOOT_FROM_Y
*/

/*
 * Define REBOOT in addition to BOOT to use self reboot support.
 * Before linking add the startup and the initdata instruction
 * segments to a reset boot class, and add all remaining
 * instuction segments needed by your project after data
 * initialization to a reboot class. The reboot class name has
 * to be 'RebootClass'.
 */
/*
#define REBOOT
*/
```

図 8-1　スタートアップ・コードの構成部（2/3）

```
/*
 * Define HSM_SUPPORT to clear all registers and stacks before
 * entering the main function. This prevents the HSM77016 to
 * claim about undefined values.
 */
#define HSM_SUPPORT

/*
 * If the hardware configuration includes external instruction
 * memory, define EXT_INSTMEM and select the appropriate number
 * of wait cycles.
 */
/*
#define EXT_INSTMEM
#define IWTR_CYCLES     0x00FC    /* one wait cycle */
*/

/*
 * If the hardware configuration includes external data memory,
 * define EXT_DATAMEM and select the appropriate number of wait
 * cycles.
 */
/*
#define EXT_DATAMEM
#define DWTR_CYCLES     0xFCFC    /* one wait cycle */
*/

/*
 * Define the default memory space to the same as in your
 * application. The default space _must_ be the same in all
 * modules.
 */
#define __DEFAULTSPACE_X
/*
#define __DEFAULTSPACE_XSER
#define __DEFAULTSPACE_Y
#define __DEFAULTSPACE_YSER
*/

/*
 * Select stacks and stacksizes.
 * There must be a stack in a memory space, if at least one
 * module uses the corresponding stack.
 * There must be a stack in the default memory space.
 */
#define USE_XSTACK
#define USE_YSTACK

#define X_STACKSIZE           2000
#define Y_STACKSIZE           2000
```

図 8-1　スタートアップ・コードの構成部（3/3）

```
/*
 * Select heap and heapsize
 */
#define USE_HEAP
#define HEAPSIZE             2000

/*
 * Define USE_STDIO to use stdio
 */
#define USE_STDIO

/***************************************************************
 * end of configuration section
 ***************************************************************/
```

第9章　ブート・サポート

ソース・プログラムが文法的にも論理的にも正しい場合，ディバッガ等の監視ソフトウエアなしに，ハードウエア上で実行する準備が整ったことになります。C コンパイラはセルフ・ブートに対応しています。本章では，この機能を，どのようにホスト・ブートに適応させるかを解説します。

ハードウエアがリセットされると，次の 2 つの動作が発生することが予測されます。

（1）プログラム・コードがデータ ROM からフェッチされ，インストラクション RAM にロードされる。

（2）グローバルおよびローカル・スタティック C コードが，データ ROM から初期化される。

プログラム・コードがデータ ROM の適切な場所に存在する場合は，（1）の動作が自動的に行われます。（2）の動作は，スタートアップ・コード中の専用アセンブラ・コードによって行われますが，このアセンブラ・コードを使用するためには，-boot オプションで C ソースをコンパイルする必要があります。

これによって，グローバルおよびローカル・スタティック変数セグメントに特別なセグメント名プリフィクスが付加されます（**第 7 章　C コンパイラ名称規則**を参照してください）。このプリフィクスを持つすべてのセグメントは，リンカによって，ユニークな INID 初期化データ・セグメント・クラスに結合されます。ユーザがセルフ・ブート・パラメータ（WB77016　**ユーザーズ・マニュアル**を参照）を INID クラスに割り付けた場合，リンカは，ROM 領域の初期化データ部の，指定スタート・アドレスを指すパブリック・シンボル_INI_ROM を生成します。スタートアップ・コードは，始めに，初期化データを ROM からデータ RAM にコピーします。これは，スタートアップ・コードの構成部で BOOT を定義することによって可能となります（ROM が Y メモリに存在する場合は，BOOT_FROM_Y を追加定義する必要があります）。

ホスト・ブート用に，-boot オプションでソース・コードをコンパイルすることも可能です。選択された INID クラスのパラメータはホスト・ブートとなり，リンカは RAM 初期化データを Hex ファイルとして生成します。初期化コピー・コードを許可する代わりに，Hex ファイルのデータをホスト・インタフェースを介してデータ RAM に転送し，アセンブラ・コードでスタートアップ・コードを拡張しなければなりません。

μPD77016 のアーキテクチャでは，インストラクション・メモリが多少制限されます。そのため，ハードウエアおよびアプリケーション全体のコード・サイズが大きすぎて，メモリに収まりきらない場合に使用されるリブート機能をサポートしています。基本的には，論理的に独立したいくつかのブロックにコードを分割します。分割された各ブロックは，インストラクション・メモリに完全に収まらなければなりません。1 つのブロックの実行後，いくつかのレジスタは，リブート・パラメータで満たされ，次のコード・ブロックをフェッチするためにリブート・ルーチンが呼び出されます（μPD7701×**ファミリ　ユーザーズ・マニュアル　アーキテクチャ編**，μPD77111 **ファミリ　ユーザーズ・マニュアル　アーキテクチャ編**またはμPD77210 **ファミリ　ユーザーズ・マニュアル　アーキテクチャ編**を参照してください）。

C コンパイラのスタートアップ・コードは，グローバルおよびローカル・スタティック・データの初期化に続いてリブート・コード部を備えているため，コードを 2 つのブロックに分割します。このようなコード分割手法を使用するためには，“ プレリブート ” コード・セグメント startup および initdata をリセット・ブート・セグメント・クラスに，その他すべてのコード・セグメントをリブート・セグメント・クラスに割り当てる必要があります（WB77016　**ユーザーズ・マニュアル**を参照）。

スタートアップ・コードの構成部で，REBOOT が BOOT に加えて許可されると，初期化コードは，グローバルおよびローカル・スタティック変数の初期化直後にアプリケーション・コードによって上書きされます。

第10章　文字出力

Cコンパイラは，文字出力に対応しています。本章では，この機能の使用方法を説明します。

通常，DSPアプリケーションでは，文字出力の必要はありませんが，コメント行を出力する何らかの方法があれば，少なくともプログラムをディバグする場合に役立ちます。dspioライブラリ関数は，この目的のために存在します。スタートアップ・コード（C.asm）は，すべてのユーザ・プログラムに必要な環境を定義しますが（**第8章　スタートアップ・コードの構成**を参照），その機能の1つとして，ユーザ・プログラムを実行する前に，dspioライブラリ関数open_stdoutを呼び出すことでstdoutを開き，ユーザ・プログラムの終了後に関数close_stdout( )を呼び出すことでstdoutを閉じることが挙げられます。たとえば，ユーザが列を出力するためにfprintfを呼び出した場合，fprintfは，始めにdspioライブラリ関数write_stdout（char*s）を呼び出します。このdspioライブラリ関数は，グローバル変数（_file_command）の値だけを変更するものです。プログラムがDSP上で実行されると，これら一連の動作が行われます。ファイル中のコメント行を受けるためには，プログラムをシミュレータ上で実行し，対応するタイミング・ファイルdspio.tmgをロードする必要があります。このタイミング・ファイルは，グローバル変数_file_commandの変更を検出し，オペレーティング・システムにStdout.txtというファイルを開き，データをこのファイルに書き込み，その後ファイルを閉じるように指示します。タイミング・ファイルのソフトウエアは，文法的に文字を書くことができないため，実行可能なtoa.exeによって，必要な文字（ASCII）に変換される数列を生成します。実行可能なtoaは，stdinからデータ列を読み出し，stdoutに書込みます。

Cコンパイラは，このような動作を行なうdspioライブラリを備えていますが，ユーザは，μPD77016ライブラリアンを使用して，これら3つのdspio関数（open_stdout，close_stdout， write_stdout）をほかの関数に置き換えることができます。これは，ライブラリからdspio.relモジュールを置き換えることによって実現できます。

第11章 挿入されたコード記述

コンパイラ・オプション-iと-ihは，アセンブラ・コード内にCソース記述を挿入します。

-ihは暗黙的に-iオプションを設定し，また，Cソース・ファイルに含まれるインフォメーションを付加的に使用します。

このコード記述は，Cコンパイラによって作り出されたアセンブラ・コードの分析を容易にします。

図 11-1　Cソース記述例

```
int count;
int compare (int *a, int *b, int val)
{
   count++;
   if (*a < val) return -1;
   if (*b < val) return  1;
   return 0;
}
```

図 11-2は，コンパイラ・オプション-iを使って，図 11-1をコンパイルした結果のアセンブラ・コードです。

図 11-2　-iオプションでコンパイルしたアセンブラ・コード（1/2）

```
; generated from line 3
   PUBLIC _compare
; 1: int count;
xdata XRAMSEG INTERNAL
   PUBLIC _count
_count:
  DW 0

;   2:
;   3: int compare ( int *a, int *b, int val )
compare_code IMSEG
;   4:  {
; Parameter a is in register dp1
; Parameter b is on stack at offset 1
; Parameter val is in register r01
_compare:
__L0:
  :dp0++ = r31;
 r0 = r0 SLL 24;
 r1 = r0 SRA 8;
 r01 = *dp0--;
;   5:   count++;
r21 = *_count:X;
```

図 11-2　-i オプションでコンパイルしたアセンブラ・コード（2/2）

```
   r2 = r2 + 1;
   *_count:X = r21;
;  6:  if (*a < val )  return  -1;
  r2 = *dp1;
  r2 = LT (r2, r1);
  IF ( r2 == 0)  JMP __L2;
__L1;
  r0l = 65535;
;   7:  if ( *b< val ) return 1;
;   8:  return 0;
;   9:  }
   RET;
__L2;
; generated from line 7
 dp1 =  r0l;
 NOP;
 r0 = *dp1;
 r0 = LT (r0, r1);
 IF ( r0 == 0 ) JMP __4;
__L3 :
  r0l = 1;
; generated from line 9
   RET;
__L4:
; generated from line 8
   CLR ( r0 );
; generated from line 9
   RET;
;  function size = 21

; word of code = 21
; word of xdata = 1
; word of xcdata = 0
; word of exdata = 0
; word of ydata = 0
; word of ycdata = 0
; word of eydata = 0
; word of esdata = 0
; word of excdata = 0
; word of eycdata = 0
; word of escdata = 0

END
```

挿入された C 記述は，アセンブラ・コード内ではセミコロン（；）を使用したコメント文になっています。

すべてのの C 記述には，C ソースでの行番号が添えられています。
コンパイラ・オプション-ih を使用すると，行番号の前にソース・ファイル名を出力することによって，ソース・コードの配置情報も付加されるので，インクルードされるファイルを明確にすることができます。

関係するアセンブラ命令は，常に C 記述に続いています。しかし，C 記述は 1 命令以上のアセンブラ命令に影響を与えることがあります。そのため，C 記述とアセンブラ命令の関係は異なることがありますが，その場合は，“gererate from line *number*”がアセンブラ命令の元ソースを明らかにします（コンパイラ・オプション-ih が再びソース・ファイルの名前とコメントを補います）。

しかし，アセンブラには，C ソース記述の対応がなく，残ってしまうものがあります（例　機能の始まりと終りにレジスタをセーブするコード）。また，アセンブラ・コードが対応しない C ソースも存在します（例　ハードウエア・ループを使用して削除したループ誘導変数の増分）。

あらゆる機能本体が図 11-2に示されるように，C ソース・コードの機能パラメータを，レジスタとスタック位置にリンクする注釈行から始まります。このアサインメントはコンパイラの呼び出し規則に反映します（7.1　**呼び出し規則**参照）。

しかし，C ソースの変数とレジスタの関係が，機能入力の一時的なコピーであることに注意してください。機能本体のその後のアセンブラ命令によっては，この関係は変わる可能性があります。一般に，挿入されたコード表現は，C コード変数（ローカル，グローバル，および機能パラメータ）と，ハードウエアやレジスタとの関係の詳細情報を提供しません。

挿入されたコード表現に加えて，コンパイラ・オプション-i および-ih は，アセンブラが必要とするメモリ情報を提供します。すべての機能は命令語（32 ビット）のコード数の表示に続いています。すべての機能のサイズを合計した総コード量は，アセンブラ・ファイルの最後に印字されます。

サイズ情報は，すべてのディフォールトおよび付加的発生するセグメントを伴ないます（7.4　**変数割り当てとセグメント名**参照）。印字されるデータ・セグメント・サイズはワード（16 ビット）数で表されます。

第12章　シンボリック・ディバグについて

本章では，コンパイラの-g オプションで制御し，コンパイラがアセンブラ・コードへ出力してシンボリック・ディバグを行うアセンブラ命令について説明します。この命令は，オブジェクトの型や位置情報，そしてソース・ファイルの行番号で構成されます。ディバグ情報をシミュレータに転送するには，アセンブラとリンカも同じオプションが必要になります。

表 12-1に命令とその簡単な説明をまとめてあります。詳細な説明は12.3　**アセンブリ命令説明**に示します。

表 12-1　シンボリック・ディバグのアセンブリ命令

アセンブリ命令	意　味
.file	カレント・ソース・ファイルに対するシンボルの定義
.line	C ソース・ステートメント行の指定
.func	C 関数の開始行の指定
.efunc	C 関数の終了行の指定
.prolog_size	関数のプロローグ・サイズの指定
.sym	グローバル変数，関数，型の定義（いくつかのパラメータで各シンボルに付ける属性を指定できる）
.etag	列挙体記述文の開始位置の指示
.stag	構造体記述文の開始位置の指示
.utag	共用体記述文の開始位置の指示
.member	列挙体，構造体，共用体のメンバの定義
.eos	列挙体，構造体，共用体記述文の終了位置の指定

12.1　ディバグ命令

シンボリック・ディバグ命令はすべて次の構文で表します。

```
keyword[whitespace]parameter[,][whitespace][parameter]
```

キーワードは，.sym のように必ずピリオドで始まります。パラメータ・リストの構成としては，必須のパラメータのあとをカンマで区切り，後ろに任意のパラメータが続きます。任意のパラメータを省略すると，パラメータ・リストに 0，もしくはより大きな空白ができます。リストの最後のパラメータを省略した場合，その部分は抜け落ちてしまいます。たとえば，storage_class と dim を省略した.sym 命令は，“ .sym name, mem, type, , size, tag ” となります。

12.2　名前の矛盾

シンボリック・ディバグ情報では，広域的な範囲と局所的な範囲を持つ名前を使用します。広域的名前は，グローバル変数やグローバル構造体，グローバル関数と同様，必ずアンダラインで始まります。

局所的名前は，ローカル変数やローカル関数パラメータ，ローカル・データ構造体と同じようにアンダラインでは始まりません。また，列挙体，構造体，共用体のメンバもアンダラインで始まりません。アンダラインがない名前はアセンブラ・キーワードと衝突する可能性があるため，名前の矛盾が起こる場合があります。これは，衝突する名前（大文字，小文字の区別をしない）はすべて，矛盾を避けるために名前の最後にアンダラインが付けられるためです。矛盾すると，警告を表示してユーザに通知します。シミュレータの監視ウインドウでは，必ずアンダラインが付けられた変数名を使用します。-gw または-gbw オプションで生成される.ini ファイルも，アンダラインが付けられた変数名を使用します。

矛盾するアセンブラ・キーワードは，次のとおりです。

ABS, ALIGN, AND, AT, CALL, CLIP, CLR, DMX, DMY, DS, DW, DWL, DYNAMIC, EIR,END,
EPC, EQ, EQU, ESR, EVER, EX, EXP, EXTERNAL, EXTRN, FINT, GE, GT, HALT, IF, IMSEG,
INTERNAL, JMP, LC, LE, LEA, LOOP, LPOP, LSA, LSP, LSR1, LSR2, LSR3, LT, MOD, NAME,
NE, NOP, NOT, OR, ORG, PUBLIC, RC, REP, RET, RETI, RETURNSVI, ROUND, SET, SLL, SP,
SR, SRA, SRL, STATIC, STK, STOP, THALT, XAR, XDMARAMSEG, XOR, XRAMSEG, XROMSEG,
YAR, YDMARAMSEG, YRAMSEG, YROMSEG

12.3　アセンブリ命令説明

12.3.1　.file

(1) 構文

```
.file “file name”
```

(2) 関数

.file 命令により，ディバッガは命令メモリの位置と C ソース・ファイルを相互に関連付けることができます。

filename　：C ソース・ファイルの名前で，先頭から 14 文字までが有効です。パス名の情報は含まれません。新しいファイル（すべてのヘッダ・ファイル（.h）含む）が読み込まれるたびに，この命令が出力されます。

(3) 例

ソース C ファイル text.c の場合，コンパイラは次のように出力します。

```
.file “text.c”
```

12.3.2　.line

（1）構文

```
.line number
```

（2）関数

この命令では 1 つのオペランド number を使い，これが C ソース・ファイルの行番号を示します。この命令はアセンブラ・ファイルの中で，その行番号の C コードに対するアセンブリ・コードの直前に表示されます。

12.3.3　.func，.efunc

（1）構文

```
.func begin_line_number [dynamic | static]
.efunc end_line_number
```

（2）関数

.func と.efunc は，C 関数の開始と終了を指示します。ディバッガは，行番号を使ってコードとソース・ファイル位置を相互に関連付けることができます。dynamic モードは，すべてのローカル変数とローカル関数パラメータをスタック上に配置します。

（3）例

（a）C コード:

```
long foo (void)      /* begin of function foo in line 10 */
{
     /* some C code */
}    /* end of function foo in line 35 */
```

（b）アセンブラ・コード:

```
.sym _foo, 1, _foo, 101, 101, 0
.func 10 dynamic
.prologsize 10
     /* some prolog assembler code */
     /* (in this case 10 instructions) */
_foo:
     /* some assembler code */
.efunc 35
```

12.3.4　.prolog_size

(1) 構文

```
.prolog_size size
```

(2) 関数

この命令では 1 つのパラメータ size を使い，これが関数のプロローグ・サイズを表します。ローカル変数は関数のプロローグに表示されません。

12.3.5　.sym

(1) 構文

```
.sym name, mem, value [, type, storage_class, size, tag, dims]
```

(2) 関数

.sym 命令は，C ソース・ファイルのシンボルに関する情報を与えます。シンボルは，グローバル変数，ローカル変数，データ構造体，関数などを表します。これらの命令に対してそれぞれパラメータがあり，最初の 3 つのパラメータが必須で，残りは任意です。

name　　ディバッガ・シンボル・テーブルに入力する名前で，先頭から 32 文字までが有効です。すべてのグローバル・シンボル名の先頭には，グローバル変数，グローバル構造体，グローバル関数と同様，コンパイラによってアンダライン（ _ ）が付けられます。ローカル・シンボル名には，ローカル変数，ローカル関数パラメータ，ローカル構造体と同じように，先頭にアンダラインは付きません（12.2　**名前の矛盾**を参照)。

mem　　シンボルのメモリ領域です。次の値を使用できます。

3　　Y メモリ
2　　X メモリ
1　　命令メモリ（コード）
0　　使用不可

value　　位置の値（意味はシンボルの保存クラスによって異なる）

保存クラス	値の意味
グローバルな明示的変数	再割り当て可能なアドレス
自動変数	ワードで示したフレーム・オフセット
構造体のメンバ	ワードで示した構造体の原点からのオフセット
共用体のメンバ	ワードで示したオフセット（通常は 0）
列挙体のメンバ	列挙体の値
ビット・フィールド	ビットで示した変位
関数	関数の再割り当て可能なアドレス

type　　シンボルのデータ型で，32 ビットの値で表されます。下位 5 ビットで基本型を表し，残りの 27 ビットで（3 ビット単位に分かれて）派生型を表します。
たとえば C 宣言 long__X *array [4] [10]の場合は，次の表にしたがって type = 5 (T_LONG) + 1*2^5 (DT_XPTR) + 4*2^8 (DT_ARRY) + 4*2^{11} (DT_ARRAY) = 0x2425 = 9253 と変換します。

データ型のコード・レイアウト					
型	派生	...	派生	派生	基本
サイズ［ビット］	3		3	3	5

基本型		
ニモニック	値	タイプ
T_NULL	0	型割り当てなし
T_VOID	1	Void
T_INT	4	Int
T_LONG	5	Long
T_STRUCT	8	構造体
T_UNION	9	共用体
T_ENUM	10	列挙体
T_MOE	11	列挙体のメンバ
T_UINT	14	Unsigned int
T_ULONG	15	Unsigned long
T_FIXED	17	__fixed
T_LFIXED	18	long __fixed
T_ACCUM	19	long __accum

派生型		
ニモニック	値	タイプ
DT_NON	0	派生型なし
DT_XPTR	1	X メモリに対するポインタ
DT_YPTR	2	Y メモリに対するポインタ
DT_FCN	3	関数
DT_ARY	4	配列
DT_CONST	5	型修飾子 const
DT_VOL	6	型修飾子 volatile

storage_class

保存クラス		
ニモニック	値	タイプ
C_NULL	0	保存クラスなし
C_AUTO	1	自動変数
C_EXT	2	グローバル明示的変数
C_MOS	8	構造体のメンバ
C_MOU	11	共用体のメンバ
C_MOE	16	列挙体のメンバ
C_FIELD	18	ビット・フィールド
C_FCN	101	関数

size　　オブジェクトのサイズ。ほとんどのオブジェクトの場合，このパラメータの値はワードで表します。ビット・フィールド・オブジェクトの場合はビットで表します。関数の場合，コンパイラは 0 を入力し，真のサイズの挿入はアセンブラに任されます（ワード単位）。

tag　　ユーザー定義の構造体または共用体の名前は，事前に.stag または.utag を使って定義されています。入力した文字の基本型フィールドが T_STRUCT または T_UNION の場合に，この名前が意味を持ちます。

dims　　配列型オブジェクトに対してのみ使用可能です。このリストの式はそれぞれ対応する配列の次元の長さを表します。カンマ（,）で区切って最高 4 つの式を使用できます。C 言語ではより高い次元の配列が可能ですが，ディバグ情報は余分な配列を切り捨てます。

（3）例

（a）C コード（グローバル・シンボル）

```
struct xglobalStr {
int mem_1, mem_2;
} __X xglobalstr;

typedef long __X XglobalArray[5][10];
Xglobalarray xglobalarray;

int __Y yglobalval;

long * __Y yglobalptr;

int foo (void){ }
```

（b）アセンブラ・コード

```
.sym _xglobalstr, 2, _xglobalstr, 8, 2, 2, _xglobalStr
```

name　　xglobalstr に対するグローバル名は_xglobalstr で，これは位置の値が再割り当て可能な式であることも示しています。

mem　　2 は X メモリを示します。

type　　8 は構造体型を示します。

storage class　　2 はグローバル変数を表します。

size　　構造体 xglobalStr は 2 つの整数で構成されているため，サイズは 2 ワードです。

tag　　ユーザ定義の構造体 xglobalStr は，広域的名前_xglobalStr を持ちます。C コード tag がない場合，コンパイラは一意の名前を生成します。

```
.sym _xglobalarray, 2, _xglobalarray, 1157, 2, 100, , 5, 10
```

type　　157（バイナリ 100 100 00101）は，long 型配列の 1 配列を示します。

size　　long 型の値の 50 は 100 ワードです。

tag　　不要なため，省略されています。

dims　　二次元配列のため，5 と 10 の 2 つがあります。

```
.sym _yglobalval, 3, _yglobalval, 4, 2, 1
```

mem　　3 は Y メモリを示します。

```
.sym _yglobalptr, 3, _yglobalptr, 37, 2, 1
```

type　　37（バイナリ 001 00101）は，X メモリの long 型に対するポインタを示します。

```
.sym _foo, 1, _foo, 100, 101, 0
```

type　　100（バイナリ 011 00100）は，int 型を返す関数を示します。

size　　コード・ワードでの真のサイズの挿入はアセンブラに任されます。

★

（c）C コード（ローカル・シンボル）

```
int foo (int parm1, int parm2)
{
  struct localStr {
    int mem_1, mem_2;
  } __X xlocalstr;

  typedef long __X XlocalArray[5][10];
  XlocalArray xlocalarray;

  int __Y ylocalval;

long * __Y  ylocalptr ;

}
```

(d) アセンブラ・コード

```
.sym _foo, 1, _foo, 100, 101, 0
```

type　100（バイナリ 011 00100）は，int 型を返す関数を示します。

size　コード・ワードでの真のサイズの挿入はアセンブラに任されます。

```
.sym par1, 2, -4, 4, 1, 1
```

name　par1 に対する関数のパラメータ名も par1 です。

mem　2 は X メモリを示します。

value　X スタック・フレームにおけるローカル変数の位置は－4 です。

type　4 は整数型を示します。

storage class　1 は自動変数を示します。

size　整数型は 1 ワードです。

```
.sym par2, 2, -3, 5, 1, 2
```

value　X スタック・フレームにおけるローカル変数の位置は－3 です。

type　5 は整数型を示します。

size　long 型は 2 ワードです。

```
.sym xlocalstr, 2, -108, 8, 1, 2, localStr
```

name　xlocalstr に対する関数のローカル名も xlocalstr です。

value　X スタック・フレームにおけるローカル変数の位置は－108 です。

type　8 は構造体型を示します。

size　構造体 localStr は 2 つの整数で構成されているため，サイズは 2 ワードです。

tag　ユーザ定義の構造体 localStr です。C コード tag がない場合，コンパイラは一意の名前を生成します。

```
.sym xlocalarray, 2, -104, 1157, 1, 100, , 5, 10
```

value　X スタック・フレームにおけるローカル変数の位置は－104 です。

type　1157（バイナリ 100 100 00101）は，long 型配列の 1 配列を示します。

size　long 型の値の 50 は 100 ワードです。

tag　不要なため，省略されています。

dims　二次元配列のため，5 と 10 の 2 つがあります。

```
.sym ylocalval, 3, -1, 4, 1, 1
```

mem　3 は Y メモリを示します。

value　Y スタック・フレームにおけるローカル変数の位置は－1 です。

```
.sym ylocalptr, 3, 0, 37, 1, 1
```

value　Y スタック・フレームにおけるローカル変数の位置は 0 です。

type　37（バイナリ 001 00101）は X メモリの long 型に対するポインタを示します。

12.3.6　.etag，.stag，.utag，.eos

（1）構文

```
.etag name [, size]
.stag name [, size]
.utag name [, size]
.eos
```

（2）関数

.etag，.stag，.utag 命令は，それぞれ列挙体，構造体，共用体の宣言を開始します。.eos 命令は宣言を終了します。

name　　.member および.sym のタグ・フィールドで使用する構造体の名前です。名前が指定されていない場合，コンパイラは構文“ _TAG_n ”（n は一意の数値）を使ってタグ名を生成します。

size　　構造体または共用体をワードで表したサイズです。size は列挙体のサイズです。

（3）例 1

（a）C コード

```
struct s_struct {
     int     i;
     int     b1:12;
     int     b2:4;
     int     j;
}    s;
```

（b）アセンブラ・コード

```
.stag _s_struct 3
.member i, 0, 4, 8, 1
.member b0, 16, 4, 18, 12
.member b1, 28, 4, 18, 4
.member j, 2, 4, 8, 1
.eos
.sym s, 2, sample, 8, 2, 3, _s_struct
```

この例には，ビット・フィールドのメンバとビット・フィールドでないメンバが混在する構造体がありますが，このような宣言はほとんどありません（説明用の例）。構造体全体のサイズは，すべてのメンバがビット・フィールドの場合でも必ずワードで示します。b1 と b2 の保存クラスは C_FIELD のため，構造体全体の原点からのオフセットをビットで示し，サイズもビットで示します。

（4）例2

（a）C **コード**

```
union {  /* note that the user did not specify a name */
     int  val_i;
     long  val_l;
}    u;
```

（b）アセンブラ・コード

```
.stag _TAG_0x86dde18 2
.member val_i, 0, 4, 11, 1
.member val_l, 0, 5, 11, 2
.eos

.sym s, 2, _u, 8, 2, 3, _TAG_0x86dde18
```

（5）例3

（a）C **コード**

```
enum regs {
result=32, regs_1, regs_2, regs_3
}    parm_register;
```

（b）アセンブラ・コード

```
.etag _regs 1
.member result, 32
.member reg_1, 33
.member reg_2, 34
.member reg_3, 35
.eos

.sym _parm_register, 2, _parm_register, 4, 2, 1
```

12.3.7　.member

（1）構文

.member *name, value* [, *type, storage_class, size, tag, dims*]

（2）関数

.member 命令は，列挙体，構造体，共用体のメンバを記述します。列挙体，構造体，共用体の定義で現れる場合のみ使用できます。mem が不要なことを除いて，パラメータは.sym 命令のパラメータと同じですので，詳しい説明については12.3.5　.symを参照してください。列挙体，構造体，共用体のメンバ名の先頭にはアンダラインは付きません（12.2　**名前の矛盾**を参照）。

（3）例

.etag, .stag, .utag の命令に従って例を示しています。

第13章　標準 C ライブラリの関数レファレンス

13.1　abort

[説　明]

Abort program（中止）。

abort 関数は，無限ループを入力して実行中のプログラムの流れを中止します。abort は制御を呼び出しプロセスへ返しません。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　文]

```
void abort (void);
```

[　例　]

```
#include <stdlib.h>

void main (void)
{
  int error = 1;

  if (error)
    abort();
}
```

13.2　abs

[説　明]

Absolute value（絶対値）。

abs は引数の絶対値を返します。その他のデータ型の絶対値については，13.26　labs または 14.11　rabs を参照してください。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　文]

```
int  abs (int i);
```

[パラメータ]

i　　絶対表示を計算すべき値

[戻り値]

abs は整数の引数 i の絶対値を返します。エラーの戻り値はありません。

[　例　]

```
#include <stdlib.h>

void main (void)
{
  int i, j;
  i = -5;

  j = abs (i);
}
```

13.3　assert

[説　　明]

Debug assertion（ディバグのアサート）。

assert マクロは stderr ストリーム上に診断メッセージを出力し，与えられた式が偽（0）の場合は，プログラムを終了させます。診断メッセージは次のようなフォーマットで出力されます。

```
Assertion failed: expression, file filename, line linenumber
```

filename はソース・ファイル名で，linenumber はアサートできなかった行の番号です。filename と linenumber は，それぞれプリプロセッサのマクロ“__FILE__”と“__LINE__”の値です。式が真（0 以外）の場合，特別な動作はありません。assert マクロは値を返しません。通常この値はプログラムの論理エラーを特定するためにプログラムを展開するときに使用します。プログラムが意図したとおりに動作している場合，与えられる式は必ず真です。

ディバグ作業が完了したら，ソース・コードを修正することなくアサート診断を終了できます。これは，コンパイラの呼び出しコマンド行のオプション-D NDEBUG（5.3.6 -D*macro*［ = value ］を参照）を追加するか，assert.h の前に#define NDEBUG 命令を挿入すると実行できます。NDEBUG が定義されると，式は数値が求められず，ソース・コードは生成されません。

[ライブラリ・インクルード・ファイル]

```
assert.h
```

[構　　文]

```
void  assert (int expression);
```

[パラメータ]

expression	数値を求める式

[　例　]

```
#include <assert.h>
void main (void)
{
  char* pa = NULL;
  assert (pa != NULL);
}
```

13.4　atexit

[説　　明]

Exit procedure（手順の終了）。

終了手順に対応していないため，atexit は必ず 0 以外のエラー値を返します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
void  atexit (void (*func)(void));
```

13.5　atoi

[説　　明]

Ascii to integer（整数に変換）。

atoi は，所定の引数が指す文字列を整数値に変換します。atoi は小数点や指数を認識しません。文字列引数は，次のようなフォーマットです。

```
[whitespace][sign]digits
```

whitespace は無視される空白文字またはタブ文字で構成され，sign はプラス（＋）またはマイナス（－）が入ります。digits は 1 以上の 10 進数です。10 進数でない数字が来たときに変換を終了します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
int  atoi (const char *ptr);
```

[パラメータ]

ptr　　整数値に変換する文字列に対するポインタ

[戻り値]

atoi は ptr が指す文字列を変換した値を返します。入力した値を整数型に変換できない場合，戻り値は 0 です。オーバフローの場合，戻り値は不定です。

[　例　]

```
#include <.h>

void main (void)
{
  int x;

  x = atoi ("-382");
}
```

13.6　atol

[説　　明]

Ascii to long（長整数値に変換）。

atol は，指定した引数が指す文字列を長整数値に変換します。文字列引数は，次のようなフォーマットです。

```
[whitespace][sign]digits
```

whitespace は無視される空白文字またはタブ文字で構成され，sign はプラス（＋）またはマイナス（－）が入ります。digits は 1 以上の 10 進数です。10 進数でない数字が来たときに変換を終了します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
long  atol (const char *ptr);
```

[パラメータ]

ptr　長整数値に変換する文字列に対するポインタ

[戻り値]

atol は ptr が指す文字列を変換した値を返します。入力した値を長整数に変換できない場合，戻り値は 0 です。オーバフローの場合，戻り値は不定です。

[　例　]

```
#include <.h>

void main()
{
  long x;

  x = atol("-382");
}
```

13.7 bsearch

[説　明]

Binary serch（バイナリ・サーチ）

bsearch は，base が指す要素番号 num をソートした配列から，key が指すオブジェクトと一致する要素のバイナリ・サーチを行います。配列の各要素のサイズは width 文字となります。compar が指す比較関数は，配列要素を指す 2 つの引数で呼び出します。第 1 引数 pkey は，key が指す同じオブジェクトを指します。第 2 引数 phase は，配列要素を指します。key が配列要素より小さい場合，比較関数は 0 より小さい値を，配列要素と等しい場合は 0 を，配列要素よりも大きい場合は 0 より大きい値を返します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　文]

```
void  *bsearch(const void* key,
               const void* base,
               size_t num,
               size_t width,
               int (*compar) (const void* pkey,
                              const void* pbase));
```

[パラメータ]

key	配列のその他すべてのオブジェクトと比較するオブジェクトに対するポインタ
base	検索する配列に対するポインタ
num	配列のオブジェクト番号
width	各オブジェクトのサイズ
compar	比較ごとに呼び出される関数に対するポインタ

[戻り値]

bsearch は一致するオブジェクトに対するポインタを返します。また，一致する要素が見つからない場合は NULL を返します。配列に key と同じ値が複数ある場合は，戻り値は必ずしも配列を直線的に検索したときに最初に一致した値とはかぎりません。

［　例　］

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

static const char *keywords[] =
{
  "auto",
  "break",
  "test"
}

#define NUM_KW sizeof(keywords) / sizeof(char *)

int kw_compare (vonst void *p1, const void *p2)
{
  const char *p1c = (const char *) p1;
  const char **p2c = (const char **) p2;
  return (strcmp (p1c, *p2c));
}

int keyword_lookup (const char *name)
{
  const char **key;
  key = (char const **) bsearch (name, keywords, NUM_KW,
                              sizeof(char *), kw_compare);
  if (key == NULL) return (-1);
  return key - keywords;
}

void main()
{
  int a, b, c;
  a = keyword_lookup ("test");          /* a =  2 */
  b = keyword_lookup ("anything");      /* b = -1 */
  c = keyword_lookup ("auto");          /* c =  0 */
}
```

13.8　calloc

［説　　明］

Allocate memory block（メモリ・ブロックの割り当て）

calloc は，オブジェクトの数がｎ，それぞれの長さが size 文字である配列に対する領域を割り当てます。各要素は 0 に初期化されます。

［ライブラリ・インクルード・ファイル］

```
stdlib.h
```

[構　　文]

```
void  *calloc (size_t n, size_t size);
```

[パラメータ]

n　　　割り当てる要素の数

size　　文字で表した各要素のサイズ

[戻り値]

calloc は，割り当てられたメモリ・ブロックの先頭にポインタを返します。使用可能な領域が不足している場合や，引数 size の値が 0 の場合は，戻り値は NULL です。戻り値が指す保存領域は，保存するオブジェクトの型に合わせて調整できます。

[　例　]

```
#include <stdlib.h>

void main (void)
{
  char *buffer;

  buffer = (char *) calloc (80, sizeof(char));
}
```

13.9　div

[説　　明]

Division（除算）

div は分子 numer を分母 denom で割った商と余りを計算します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

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

typedef struct
{
  int quot;        /* Quotient  */
  int rem;         /* Remainder */
} div_t;
```

[パラメータ]

numer　　　実行する除算の分子

denom　　　実行する除算の分母

[戻り値]

div は，フィールド quot と rem を持つ div_t 型の構造体を返します。denom が 0 の場合，戻り値は不定です。

[　例　]

```
#include <stdlib.h>

void main (void)
{
  int   seconds = 423;
  div_t min_sec;

  min_sec = div (seconds, 60);

  /* min_sec.quot are the minutes */
  /* min_sec.rem are the seconds  */
}
```

13.10　fprintf

[説　　明]

Print to file（ファイル書式印字）。

fprintf は，引数 format の制御下で，fp が指すストリームに出力を書き込みます。フォーマット文字列については 13.36　printf を参照してください。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int  fprintf (FILE *fp, const char *format, ...);
```

[パラメータ]

fp	出力ストリームに対するポインタ（stdout または stderr のみ）
format	フォーマット文字列（13.36　printf を参照）
...	フォーマット文字列で指定した引数

[戻り値]

fprintf は，ストリームに書き込まれた文字数を返します。出力エラーが発生した場合は，負の値を返します。

[備　　考]

本 C コンパイラの fprintf 関数は，シミュレータの stdout または stderr へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

char *text = { "hello!" };
int  number = 12;

void main()
{
  fprintf (stdout, "%s, %d¥n", text, number);
}
```

13.11　fputc

[説　　明]

Put char to file（ファイル文字出力）

fputc は，引数で指定した文字を fp が指す出力ストリームに書き込みます。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int  fputc(int c, FILE *fp);
```

[パラメータ]

c　　書き込む文字

fp　　出力するファイルに対するポインタ（stdout または stderr）

[戻り値]

fputc は書き込まれた文字を返します。また，エラーが発生した場合は EOF を返します。

[備　　考]

本 C コンパイラの fputc 関数は，シミュレータの stdout または stderr へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

 void main (void)
{
  fputc (65, stdout);
}
```

13.12　fputs

[説　明]

Put string to file（ファイル文字列出力）

fputs は，buf が指す文字列を fp が指す出力ストリームに書き込みます。終端の NULL 文字は書き込まれません。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　文]

```
int  fputs (const char *buf, FILE *fp);
```

[パラメータ]

buf　書き込む文字列に対するポインタ

fp　出力するファイルに対するポインタ

[戻り値]

fputs は，書き込みエラーや符号化エラーが発生した場合は EOF を返します。それ以外の場合は，ストリームへ書き込んだ文字列を示す負でない値を返します。

[備　考]

本 C コンパイラの fputs 関数は，シミュレータの stdout または stderr へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
# include <stdio.h>

char *stringbuffer = { "hello!" };

void main (void)
{
  fputs (stringbuffer, stdout);
}
```

13.13　free

[説　明]

Free memory block（メモリ・ブロックの解放）

free は，ptr が指すメモリ・ブロックを解放します。prt は，calloc，malloc，realloc の適切なバージョンへの呼び出しで確保したメモリ・ブロックを指すポインタです。このポインタが NULL の場合，free 関数は何も行いません。関数を呼び出した後，解放されたメモリ・ブロックを割り当て用に使用できます。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
void  free (void *ptr);
```

[パラメータ]

ptr　　解放するメモリ・ブロックに対するポインタ

[　例　]

```
#include <stdlib.h>

void main (void)
{
  char *buffer;

  buffer = malloc (80);

  /* use the allocated memory block here */

  free (buffer);
}
```

13.14　isalnum

[説　　明]

Is alphanumeric character（英数字判定）

isalnum は，引数が英数字（a～z，A～Z，0～9）かどうかを調べます。英数字は，isalpha または isdigit 関数が真となる文字です。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  isalnum (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

引数が英文字（A～Z，a～z）と数字（0～9）のどちらでもない場合，isalnum は 0 を返します。それ以外の場合は 0 以外の値を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = 'A';
  int  a;

  if (isalnum (c))
    a++;
}
```

13.15　isalpha

[説　　明]

Is alphabethical character（英文字判定）

isalpha は，引数が英文字（a～z，A～Z）かどうかを調べます。英文字は，isupper または islower 関数が真となる文字です。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int isalpha (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

引数が英文字（A～Z，a～z）でない場合，isalpha は 0 を返します。それ以外の場合は 0 以外の値を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = 'A';
  int  a;

  if (isalpha (c))
    a++;
}
```

13.16　isascii

[説　　明]

Is ascii character（ASCII 文字判定）

isascii は，引数が 0〜127 の範囲にある文字かどうかを調べます。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  isascii (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が 0〜127 の範囲にある場合，isascii は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = 'A';
  int  a;

  if (isascii (c))
    a++;
}
```

13.17　iscntrl

[説　　明]

Is control character（制御文字判定）

iscntrl は，引数が制御文字かどうかを調べます。制御文字は，0〜31 の範囲にある文字です。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  iscntrl (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が 0〜31 の範囲にある場合，iscntrl は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = 'A';
  int  a;

  if (isctrl (c))
    a++;
}
```

13.18　isdigit

[説　　明]

Is digit character（数字判定）

isdigit は，引数が 0〜9 の 10 進数字かどうかを調べます。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  isdigit (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が 0〜9 の 10 進数字の場合，isdigit は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (isdigit (c))
    a++;
}
```

13.19　isgraph

[説　　明]

Is graphical character（図形文字判定）

isgraph は，引数が空白文字（「 」）以外の印刷可能な文字かどうかを調べます。対象となる文字セットに空白文字も含まれること以外は isprint 関数と同じです。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  isgraph (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が印刷可能な文字（空白を除く）の場合，isgraph は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (isgraph (c))
    a++;
}
```

13.20　islower

[説　　明]

Is lowercase character（英小文字判定）

islower は，引数が a～z の英小文字かどうかを調べます。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  islower (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が英小文字（a～z）の場合，islower は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (islower (c))
    a++;
}
```

13.21　isprint

[説　　明]

Is printable character（印刷可能な文字かどうかを調べます）

isprint は，引数が空白文字（「 」）を含む印刷可能な文字かどうかを調べます。対象となる文字セットに空白文字も含まれること以外は isgraph 関数と同じです。

[ライブラリ・インクルード・ファイル]

ctype.h

[構　　文]

```
int  isprint (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が印刷可能な文字（空白を含む）の場合，isprint は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (isprint (c))
    a++;
}
```

13.22　ispunct

[説　　明]

Is punctuation character（区切り文字判定）

ispunct は，引数がカンマ（,）やピリオド（.）などの区切り文字かどうかを調べます。区切り文字は，空白文字，または isalnum が偽となる文字以外の印刷可能な文字です。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  ispunct (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が区切り文字の場合，ispunct は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (ispunct (c))
    a++;
}
```

13.23　isspace

[説　明]

Is space（空白類文字かどうかを調べます）

isspace は，引数が次の空白類文字かどうかを調べます。

「 」	空白
「¥f」	書式送り
「¥n」	改行
「¥r」	行の先頭への復帰
「¥t」	水平タブ
「¥v」	垂直タブ

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　文]

```
int  isspace (inc c);
```

[パラメータ]

c　調べる文字

[戻り値]

引数が上のいずれかの空白類文字の場合，isspace は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '¥n';
  int  a;

  if (isspace (c))
    a++;
}
```

13.24　isupper

[説　明]

Is uppercase（英大文字判定）

isupper は，引数が A～Z の英大文字かどうかを調べます。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

```
int  isupper (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が英大文字（A〜Z）の場合，isupper は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = '7';
  int  a;

  if (isupper (c))
    a++;
}
```

13.25　isxdigit

[説　　明]

Is hex digit（16 進数字判定）

isxdigit は，引数が 0〜9，a〜f，A〜F の 16 進数字かどうかを調べます。

[ライブラリ・インクルード・ファイル]

ctype.h

[構　　文]

```
int  isxdigit (inc c);
```

[パラメータ]

c　　調べる文字

[戻り値]

文字 c が 16 進数字の場合，isxdigit は 0 以外の値を返します。それ以外の場合は 0 を返します。

[　例　]

```
#include <ctype.h>

void main (void)
{
  char c = 'D';
  int  a;

  if (isxdigit (c))
    a++;
}
```

13.26　labs

[説　　明]

Absolute long value（長整数絶対値の計算）

labs は所定の引数の絶対値を返します。その他のデータ型の絶対値については，13.2　absと 14.11　rabs を参照してください。

[ライブラリ・インクルード・ファイル]

stdlib.h

[構　　文]

```
long  labs (long i);
```

[パラメータ]

i　　絶対表示を計算する値

[戻り値]

labs は長整数の引数 i の絶対値を返します。エラーの戻り値はありません。

[　例　]

```
#include <stdlib.h>

void main (void)
{
  long i, j;
  i = -5;

  j = labs (i);
}
```

13.27　ldiv

[説　明]

Long division（長整数の除算）

ldiv は分子 numer を分母 denom で割った商と余りを計算します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　文]

```
ldiv_t  div (long int numer, long int denom);

typedef struct
{
  long int quot;        /* Quotient  */
  long int rem;         /* Remainder */
} ldiv_t;
```

[パラメータ]

numer	実行する除算の分子
denom	実行する除算の分母

[戻り値]

ldiv は，フィールド quot と rem を持つ ldiv_t 型の構造体を返します。denom が 0 の場合，戻り値は不定です。

[　例　]

```
#include <stdlib.h>

void main (void)
{
  long int seconds = 423;
  ldiv_t   min_sec;

  min_sec = ldiv (seconds, 60);

  /* min_sec.quot are the minutes */
  /* min_sec.rem are the seconds  */
}
```

13.28　longjmp

[説　　明]

Restore state information（状態情報の復元）

longjmp は，以前の setjmp を呼び出しで状態が保存されたバッファ env からプログラム状態の情報を復元します。関数は longjmp の呼び出しに戻るのではなく，以前の setjmp 呼び出しのあとにジャンプします。ここで longjmp の入力引数 val を返します。setjmp と longjmp は，通常の関数呼び出しと復帰を省略するため，例外処理によく使用されます。呼び出し setjmp の自動変数値は，volatile と定義されていないかぎり longjmp のあと不定となります。

[ライブラリ・インクルード・ファイル]

```
setjmp.h
```

[構　　文]

```
void longjmp (jmp_buf env, int val);
```

[パラメータ]

env　プログラム状態の情報を保存するバッファ。本 C コンパイラは，実際の命令ポインタと，setjmp を呼び出した関数が保存されたレジスタを保存します。

val　対応する setjmp の呼び出しによって返される値。

[例]

```
#include <setjmp.h>

jmp_buf  env;

void main (void)
{
  ...
  if (setjmp (env) == 0)              /* save status */
  {
    /* sequence of function calls with the following*/
    /* statement in the innermost function body:    */
    if (rmatherr() != ME_NOERROR)     /* if error happens */
      longjmp (env, 1);               /* restore status */
  }
  else                                /* and use else path*/
  {
    /* some error case code */
  }
}
```

13.29 malloc

[説　　明]

Allocate memory block（メモリ・ブロックの割り当て）

malloc は，size 文字のオブジェクトに対する領域を割り当てます。引数 size の値が 0 の場合，割り当ては行われません。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

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

[パラメータ]

size　割り当てるメモリ・ブロックの，文字で表したサイズ

[戻り値]

malloc は，割り当てたメモリ・ブロックの先頭にポインタを返します。使用可能な領域が不足している場合や，要求した size の値が 0 の場合は，戻り値は NULL です。

[例]

```
#include <stdlib.h>

void main (void)
{
  char *buffer;

  buffer = (char *) malloc (sizeof(char));
}
```

13.30 memchr

[説　　明]

Locate character（文字の検索）

memchr は，buf が指すオブジェクトの最初の length 文字の中で最初に現われた（unsigned char 型に変換した）文字 ch を探します。使用可能な各メモリ領域（X，Y，XSER，YSER）ごとに実行関数が用意されています。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
void  *memchr (const void *buf, int ch, size_t length);
void __X  *memchr_X (const void __X *buf, int ch, size_t length);
void __Y  *memchr_Y (const void __Y *buf, int ch, size_t length);
```

[パラメータ]

buf　　検索するバッファに対するポインタ
ch　　バッファ内で検索する文字
length　最大の検索文字長

[戻り値]

memchr は見つかった文字に対するポインタを返します。また，バッファで文字が出現しなかった場合は，NULL を返します。

[備　　考]

memchr はディフォールトのメモリ領域を使用します（詳細については，5.6　**メモリ・オプション**を参照)。2 つの関数 memchr_X と memchr_Y は，X および Y のデータ・メモリのアクセスに向けて最適化されています。

[　例　]

```
#include <string.h>

char buffer[11] = "abcedfghij";
char *where;

void main (void)
{
  where = memchr (buffer, 'f', 10);
}
```

13.31　memcmp

[説　　明]

Compare memory（メモリ比較）

memcmp は，s1 が指すオブジェクトと s2 が指すオブジェクトの最初の length 文字を比較します。あらゆる種類のメモリ領域間を比較する関数があります。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
int  memcmp (const void *s1, const void *s2, size_t length);
int  memcmp_XtoXser (const void __XSER *s1,
                     const void __XSER *s2,
                     size_t length);
int  memcmp_XtoYser (const void __XSER *s1,
                     const void __YSER *s2,
                     size_t length);
int  memcmp_YtoXser (const void __YSER *s1,
                     const void __XSER *s2,
                     size_t length);
int  memcmp_YtoYser (const void __YSER *s1,
                     const void __YSER *s2,
                     size_t length);
int  memcmp_XtoY (const void __X *s1,
                  const void __Y *s2,
                  size_t length);
int  memcmp_YtoX (const void __Y *s1,
                 const void __X *s2,
                 size_t length);
```

[パラメータ]

s1，s2	比較するオブジェクトに対するポインタ
length	文字で表したオブジェクトの長さ

[戻り値]

memcmp は，s1 の指すオブジェクトが s2 の指すオブジェクトよりも小さい場合は 0 より小さい整数， s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。

[備　　考]

memcmp は，ディフォールトのメモリ領域を使用します（詳細については，5.6　**メモリ・オプション**参照）。上記にて示した関数は，X および Y のデータ・メモリで使用するために最適化されています。memcmp_XtoY と memcmp_YtoX は，パフォーマンスの向上のために並行データ・アクセスを使用します。

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char buffer2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = memcmp (buffer1, buffer2, 10);
}
```

13.32　memcpy

[説　明]

Copy memory（メモリの複写）

memcpy は，length 文字を src が指すバッファから dst が指すバッファへコピーします。重複するオブジェクトをコピーすると，正常に動作しない場合があります。重複するオブジェクトのコピーについては，13.33　memmoveを参照してください。あらゆる種類のメモリ領域間のコピーを行う関数があります。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　文]

```
void  *memcpy (const void *dst, const void *src, size_t length);
void __XSER  *memcpy_XtoXser (const void __XSER *dst,
                              const void __XSER *src,
                              size_t length);
void __YSER  *memcpy_XtoYser (const void __YSER *dst,
                              const void __XSER *src,
                              size_t length);
void __XSER  *memcpy_YtoXser (const void __XSER *dst,
                              const void __YSER *src,
                              size_t length);
void __YSER  *memcpy_YtoYser (const void __YSER *dst,
                              const void __YSER *src,
                              size_t length);
void __Y     *memcpy_XtoY (const void __Y *dst,
                          const void __X *src,
                          size_t length);
void __X     *memcpy_YtoX (const void __X *dst,
                          const void __Y *src,
                          size_t length);
```

[パラメータ]

dst	コピー先のバッファに対するポインタ
src	コピー元のバッファに対するポインタ
length	コピーする文字数

[戻り値]

memcpy は dst の元のポインタ値を返します。

[備　考]

memcpy は，ディフォールトのメモリ領域を使用します（詳細については 5.6　メモリ・オプションを参照）。上記に示した関数は，X および Y のデータ・メモリで使用するために最適化されています。memcpy_XtoY と memcpy_YtoX は，パフォーマンスの向上のために並行データ・アクセスを使用します。

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char buffer2[11];
char *buffer;

void main (void)
{
  buffer = memcpy (buffer1, buffer2, 11);
}
```

13.33　memmove

[説　　明]

Copy overlapping memory（重複メモリ複写）

memmove は，length 文字を src が指すバッファから dst が指すバッファへコピーします。重複するオブジェクトを正しくコピーします。あるメモリ領域から別の領域へデータをコピーするための関数があります。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
void *memmove (const void *dst, const void *src, size_t length);
void __XSER  *memmove_XtoXser (const void __XSER *dst,
                               const void __XSER *src,
                               size_t length);
void __YSER  *memmove_XtoYser (const void __YSER *dst,
                               const void __XSER *src,
                               size_t length);
void __XSER  *memmove_YtoXser (const void __XSER *dst,
                               const void __YSER *src,
                               size_t length);
void __YSER  *memmove_YtoYser (const void __YSER *dst,
                               const void __YSER *src,
                               size_t length);
void __Y    *memmove_XtoY (const void __Y *dst,
                          const void __X *src,
                          size_t length);
void __X    *memmove_YtoX (const void __X *dst,
                          const void __Y *src,
                          size_t length);
```

[パラメータ]

dst	コピー先のバッファに対するポインタ
src	コピー元のバッファに対するポインタ
length	コピーする文字数

[戻り値]

memmove は dst の元のポインタ値を返します。

[備　　考]

memmove は，ディフォールトのメモリ領域を使用します（詳細については，5.6　**メモリ・オプション**を参照)。上記に示した関数は，X および Y のデータ・メモリで使用するために最適化されています。memmove_XtoY と memmove_YtoX は，パフォーマンスの向上のために並行データ・アクセスを使用します。

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char buffer2[11];
char *buffer;

void main (void)
{
  buffer = memmove (buffer1, buffer2, 10);
}
```

13.34　memset

[説　　明]

Fill memory（メモリの充填）

memset は，dst が指すオブジェクトの最初の length 文字を値 c で充填します。使用可能な各メモリ領域に対する関数があります。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
void  *memset (const void *dst, int c, size_t length);
void __X  *memset_X (const void __X *dst, int c, size_t length);
void __Y  *memset_Y (const void __Y *dst, int c, size_t length);
```

[パラメータ]

dst	充填するバッファに対するポインタ
c	バッファを充填する文字
length	バッファを充填する文字数

[戻り値]

memset は入力ポインタ dst を返します。

[備　　考]

memset **は，ディフォールトのメモリ領域を使用します（詳細については，**5.6　**メモリ・オプション**を参照）。2 **つの関数** memset_X **と** memset_Y **は，**X **および** Y **のデータ・メモリにアクセスするために最適化されています。**

[　例　]

```
#include <string.h>

char buffer[10];
char *where;

void main (void)
{
  where = memset (buffer, 'x', 10);
}
```

13.35　offsetof

[説　　明]

Get offset of（offset **値を求める）**

offsetof **マクロは，構造体または共用体の合成物における要素名の** offset **値を返します。これにより，** offset **値を簡単に求めることができます。**

[ライブラリ・インクルード・ファイル]

```
stddef.h
```

[構　　文]

size_t　offsetof (*composite*, *name*);

[パラメータ]

composite	**オブジェクトが属する構造体または共用体**
name	**オブジェクト名**

[戻り値]

offsetof **マクロは，オブジェクト** name **の** offset **値を返します。ビット・フィールドは不定です。**

[　例　]

```
#include <stddef.h>

struct def
{
  char *first;
  char second[10];
  char third;
}

void main (void)
{
  int a = offsetof (struct def, second);
}
```

13.36　printf

[説　　明]

Print（印字書式）

printf は，引数 format の制御下で，stdout が指すストリームに出力を書き込みます。

[ライブラリ・インクルード・ファイル]

stdio.h

[構　　文]

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

[パラメータ]

format	フォーマット制御文字列（後の説明を参照）
...	フォーマット文字列で指定した可変引数

[戻り値]

printf は書き込まれた文字数を返します。出力エラーが発生した場合は，負の値を返します。

[備　　考]

本 C コンパイラの printf 関数は，シミュレータの stdout へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

char *text = { "hello!" };
int  number = 12;

void main (void)
{
  printf ("%s, %d¥n", text, number);
}
```

[フォーマット制御文字列]

任意と必須のフィールドで構成されるフォーマット指定は，次のようなフォーマットです。

%[*flags*][*width*][*.precision*][{h|l}]*type*

フォーマット指定の各フィールドには，特定のフォーマット・オプションを表す 1 つの文字または数字が入ります。最も単純なフォーマットの指定は，「%s」のように%記号と 1 つの型を表す文字だけを含むものです。%の後にフォーマット・フィールドとしての意味を持たない文字が続くと，その文字は stdout へコピーされます。%%の場合は「%」が出力されます。

本 C コンパイラは浮動小数点値に対応していないため，浮動小数点のフォーマット制御文字は認識されず，通常の文字として出力されます。ただし，固定小数点値の出力はリテラル A の型によってサポートされています。

各文字はそれぞれ次に示すフォーマットの項目を設定します。

type　　指定した引数を文字，文字列，数値のどれに変換するかを判断する必須文字

文　字	型	出力フォーマット
c, C	char	文字
d, i	int	符号付き 10 進整数値
o	unsigned int	符号なし 8 進整数値
u	unsigned int	符号なし 10 進整数値
x	unsigned int	“abcdef”を使用する符号なし 16 進整数値
X	unsigned int	“ABCDEF”を使用する符号なし 16 進整数値
A	long __accum	符号付き固定小数点値
s, S	char*	文字列

flags　任意の文字または文字列で，出力の行末揃えと，符号，空白，小数点，8 進数または 16 進数のプリフィクスの表示を制御します。1 つのフォーマット指定に複数のフラグを指定できます。

フラグ	意　　味	ディフォールト
－	所定のフィールド幅の範囲内で結果を左寄せにします。	右寄せ
＋	出力値の型が符号付きの場合，先頭に符号（＋または－）を付けます。	負の値の場合のみ符号を付ける
0	s，S の型の場合のみ使用します（それ以外の場合は無視されます）。最小のフィールド幅になるまで 0 を追加します。0 とマイナス（－）が存在する場合，0 は無視されます。	0 を詰めない
空白	出力値が符号付き正の値の場合，先頭を空白にします。空白とプラス（＋）の両方が存在する場合，空白は無視されます。	空白にしない
#	o，x，X の型の場合のみ使用します（それ以外の場合は無視されます）。#を指定すると，0 以外の出力値の先頭にそれぞれ 0，0x，0X を表示します。	空白にしない

width　最小の文字を指定する任意の数値です。フォーマット指定の 2 番目の任意フィールドで幅を指定します。幅の引数は，出力する文字の最小数を制御する負でない 10 進数の整数です。出力値の文字が指定した幅よりも小さい場合，左寄せフラグを指定したかどうかによって，値の右または左に空白を詰めます。幅の先頭に 0（ゼロ）がある場合は，最小幅まで 0 を詰めます（左寄せの場合は無効）。幅の指定では，値が切り詰められることはありません。出力値の文字が指定した幅よりも大きい場合や，幅が指定されていない場合は，文字がすべて出力されます。幅をアスタリスク（*）で指定すると，引数リストの引数 int が値となります。引数リストでは，引数 width は必ずフォーマット設定する値の前に書きます。値が存在しない場合やフィールド幅が小さい場合，フィールド幅は切り詰められません。
また，変換結果がフィールド幅よりも大きい場合，フィールドを延長して変換結果を保存します。

precision　出力フィールドの全体または一部を出力する最大の文字数，または整数値を出力する最小の桁数を指定する任意の数値です。

型	意　　味	ディフォールト
c, C	精度は無効です。	文字が出力される
d, i, u, o, x, X	精度は出力する最小の桁数を指定します。引数の桁数が精度未満の場合，出力値の左に 0 を詰めます。 桁数が精度を越える場合，出力値を切り詰めません。	ディフォールトの精度は 1
A	精度は出力する小数位置の最大桁数を指定します。	ディフォールトの精度は 6
s,S	精度は出力する最大の文字を指定します。精度を越える文字は出力しません。	NULL 文字が現われるまで文字を印刷する

h|l　引数サイズを指定する型に対する任意のプリフィクスです。

指定する型	プリフィクス	指定子
long int	l	d, i, o, x, X
long unsigned int	l	u
short int	h	d, i, o, x, X
short unsigned int	h	u

13.37　putc

[説　明]

Put character（文字出力）

putc は，引数で指定した文字を fp が指す出力ストリームに書き込みます。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　文]

```
int putc (int c, FILE *fp);
```

[パラメータ]

c　書き込む文字

fp　出力ストリームに対するポインタ（stdout または stderr）

[戻り値]

putc は書き込まれた文字を返します。また，エラーが発生した場合は EOF を返します。

[備　考]

本 C コンパイラの putc 関数は，シミュレータの stdout または stderr へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

 void main (void)
{
  putc ('A', stdout);
}
```

13.38　putchar

[説　明]

Put character（文字出力）

putchar は，引数で指定した文字を出力ストリーム stdout に書き込みます。この関数は，stdout ストリームが指定された putc 関数と同じです。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　文]

```
int putchar (int c);
```

[パラメータ]

c　　書き込む文字

[戻り値]

putchar は書き込まれた文字を返します。また，書き込みエラーが発生した場合は EOF を返します。

[備　　考]

本 C コンパイラの putchar 関数は，シミュレータの stdout へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

 void main (void)
{
  putchar (65);
}
```

13.39　puts

[説　　明]

Put string（文字列出力）

puts は，buf が指す文字列を出力ストリーム stdout に書き込みます。終端の NULL 文字は，新しい改行文字（「¥n」）として書き込みます。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int puts (const char *buf);
```

[パラメータ]

buf　書き込む文字列に対するポインタ

[戻り値]

puts は，書き込みエラーや符号化エラーが発生した場合は EOF を返します。正常な場合は，負でない値を返します。

[備　　考]

本 C コンパイラの puts 関数は，シミュレータの stdout へのみ出力できます。正確な動作にはタイミング・ファイル dspio.tmg が必要です。ファイル出力の詳細については，**第 10 章　文字出力**を参照してください。

[　例　]

```
# include <stdio.h>

char *stringbuffer = { "hello!" };

void main (void)
{
  puts (stringbuffer);
}
```

13.40　rand

[説　　明]

Random value（乱数値生成）

rand は，0〜RAND_MAX（32767）の範囲内で擬似乱数の整数シーケンスを計算します。まえもって srand 関数を呼び出すと，シーケンスをさまざまな値で開始できます。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
int rand (void);
```

[戻り値]

rand は擬似乱数整数値を返します。エラーの戻り値はありません。

[　例　]

```
#include <.h>

int i;

void main (void)
{
  i = rand();
}
```

13.41　realloc

[説　　明]

Rellocate memory（メモリの再割り当て）

realloc はメモリの再割り当てを行います。ポインタ old_blk が NULL の場合は，malloc 関数と同じ動作を行います。size が 0 の場合は，指定したメモリ・ブロックを解放します。

それ以外の場合は，次のいずれかの方法で size 文字のオブジェクトの領域を割り当て直します。

- size が old_blk のサイズよりも小さく不足している場合は，割り当て済みのメモリ・ブロック old_blk を縮小します。
- old_blk の直後に十分な大きさがある未割り当てのメモリ・ブロックがある場合は，割り当て済みのメモリ・ブロック old_blk のサイズを拡大します。
- 新たにブロックを割り当て，old_blk の内容を新しいブロックにコピーします。

新しいブロックを割り当てることができるため，ユーザは前に割り当てたメモリ・ブロックに対するポインタを削除する必要があります。これらのポインタは解放されたメモリを指している場合があります。そのため，新しいブロックを割り当てたときにエラーが発生する可能性があります。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
void *realloc (void *old_blk, size_t size);
```

[パラメータ]

old_blk	前に割り当てたメモリ・ブロックに対するポインタ
size	新しいメモリ・ブロックのサイズ

[戻り値]

関数は再割り当てを行ったメモリ・ブロックに対するポインタを返します。また，再割り当てが正常に行われなかった場合や引数 size が 0 の場合は，NULL を返します。

[　例　]

```
#include <stdlib.h>

char *buffer;
char *newbuffer;

void main (void)
{
  buffer = malloc (10);
  newbuffer = realloc (buffer, 15);
  /* do not use buffer pointer after realloc function call */
  free (newbuffer);
}
```

13.42　setjmp

[説　明]

Save state information（状態情報の保存）

setjmp は，バッファ env にプログラム状態の情報を保存します。longjmp 呼び出しで保存した情報をのちに復元し，setjmp 呼び出しの戻り値を変更したあとすぐにプログラムを続行します。setjmp と longjmp は，通常の関数呼び出しを省略するため，例外処理によく使用されます。setjmp は，if や switch などの簡単な条件文やループの開始位置で使用できます。

[ライブラリ・インクルード・ファイル]

```
setjmp.h
```

[構　文]

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

[パラメータ]

env　プログラム状態の情報を保存するバッファ。本 C コンパイラは，実際の命令ポインタと， setjmp を呼び出した関数を保存したレジスタを保存します。

[戻り値]

setjmp は，longjmp の呼び出しの直後を除き 0 を返します。その後，戻り値は longjmp によって決まります。

[　例　]

```
#include <setjmp.h>

jmp_buf  env;

void main (void)
{
  ...
  if (setjmp (env) == 0)             /* save status */
  {
    /* sequence of function calls with the following */
    /* statement in the innermost function body:    */
    if (rmatherr() != ME_NOERROR)   /* if error happens */
      longjmp (env, 1);             /* restore status */
  }
  else                              /* and use else path*/
  {
    /* some error case code */
  }
}
```

13.43　sprintf

[説　　明]

Print to string（文字列出力）

sprintf は，引数 format の制御下で，buf が指すバッファに出力を書き込みます。フォーマット文字列については 13.36　printfを参照してください。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int sprintf (char *buf, const char *format, ...);
```

[パラメータ]

buf	出力するバッファに対するポインタ
format	フォーマット文字列（13.36　printfを参照）
...	フォーマット文字列で指定した引数

[戻り値]

sprintf は，書き込まれた文字数を返します（終端を表す NULL 文字は数えません）。また，出力エラーが発生した場合は，負の値を返します。

[　例　]

```
#include <stdio.h>

char buffer[20];
int  number = 12;

void main (void)
{
  sprintf (buffer, "Number: %d¥n", number);
}
```

13.44　srand

[説　　明]

Random seed（乱数種）

srand は，擬似乱数ジェネレータの初期値を設定します。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

```
void srand (int seed);
```

[パラメータ]

seed　　擬似乱数ジェネレータの乱数種

[備　考]

ジェネレータを再初期化するには，seed = 1 を使用します。その他の seed 値は，ジェネレータを乱数の開始点に設定します。rand は生成された擬似乱数値を取り出します。srand を呼び出す前に rand を呼び出すと，seed = 1 の srand を呼び出すのと同じシーケンスを生成します。

[　例　]

```
#include <.h>

int i;

void main (void)
{
  srand (23041);
  i = rand();
}
```

13.45　strcmpi

[説　明]

Compare strings caseinsensitive（文字列比較（大文字小文字の区別なし））

strcmpi は，s1 が指す文字列と s2 が指す文字列を大文字小文字の区別をせずに比較します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　文]

```
int strcmpi (const char *s1, const char *s2);
```

[パラメータ]

s1，s2　　大文字小文字の区別をせずに比較する文字列に対するポインタ

[戻り値]

strcmpi は，s1 の指す文字列が s2 の指す文字列よりも小さい場合は 0 より小さい整数，s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。

[　例　]

```
#include <string.h>

char str1[11] = "abcedfghij";
char str2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = strcmpi (str1, str2);
}
```

13.46　strcat

[説　　明]

Append strings（文字列追加）

strcat は，src が指す文字列のコピーを dst が指す文字列の最後に追加します。文字列 src の最初の文字は，文字列 dst の末尾の NULL 文字を上書きします。この関数で新しいメモリは割り当てられないため，追加先のバッファには十分なスペースが必要です。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strcat (char *dst, const char *src);
```

[パラメータ]

dst　　追加先の文字列に対するポインタ

src　　追加元の文字列に対するポインタ

[戻り値]

strcat は dst の元の値を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <.h>

char buffer[80];

void main (void)
{
  strcpy (buffer, "Hello ");
  strcat (buffer, "world!");
}
```

13.47　strchr

[説　　明]

Locate character in string（文字列中の文字の検索）

strchr は，s が指す文字列の中で最初に現われる文字 c を探します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

s　　検索する文字列に対するポインタ

c　　バッファで検索する文字

[戻り値]

strchr は最初に見つかった文字に対するポインタを返します。また，文字が文字列に出現しなかった場合は，NULL を返します。

[　例　]

```
#include <string.h>

char buffer[11] = "abcedfghij";
char *where;

void main (void)
{
  where = strchr (buffer, 'f');
}
```

13.48　strcmp

[説　　明]

Compare strings（文字列比較）

strcmp は，s1 が指す文字列と s2 が指す文字列を比較します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

s1，s2　　　比較する文字列に対するポインタ

[戻り値]

strcmp は，s1 の指す文字列が s2 の指す文字列よりも小さい場合は 0 より小さい整数，s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。

[　例　]

```
#include <string.h>

char str1[11] = "abcedfghij";
char str2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = strcmp (str1, str2);
}
```

13.49　strcoll

[説　　明]

Compare strings locate-specific（地域特有の文字列を比較）

strcoll は s1 が指す文字列と s2 が指す文字列を地域設定を使って比較します。DSP ライブラリにはこれらの設定がないため，この関数は strcmp 関数と同じように動作します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

s1，s2　　　　比較する文字列に対するポインタ

[戻り値]

strcoll は，s1 の指す文字列が s2 の指す文字列よりも小さい場合は 0 より小さい整数，s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。

[　例　]

```
#include <string.h>

char str1[11] = "abcedfghij";
char str2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = strcoll (str1, str2);
}
```

13.50　strcpy

[説　　明]

Copy strings（文字列複写）

strcpy は，文字列を src が指すバッファから dst が指すバッファへコピーします。重複する文字列をコピーすると，正常に動作しない場合があります。重複するオブジェクトのコピーについては，13.33　memmoveを参照してください。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strcpy (const void *dst, const void *src);
```

[パラメータ]

dst　コピー先のバッファに対するポインタ

src　コピー元のバッファに対するポインタ

[戻り値]

strcpy は dst の元の値を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char buffer2[11];
char *buffer;

void main (void)
{
  buffer = strcpy (buffer2, buffer1);
}
```

13.51　strcspn

[説　明]

Length of initial segment（先頭セグメント長）

strcspn（文字列カウンタ設定位置）は，str が指す文字列の中の charset が指す文字列内で最初に現われる文字の 1 つを検索します。str の末尾で終端を表す NULL 文字は，str の一部とみなしません。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　文]

```
size_t strcspn (const char *str, const char *charset);
```

[パラメータ]

str	文字列に対するポインタ
charset	比較文字セットを含む文字列に対するポインタ

[戻り値]

strcspn は，str の先頭にあり charset にも含まれる文字のインデクス，つまり charset にない文字で全体が構成されている str の先頭セグメント長を返します。先頭文字に対するポインタを取得する方法については，13.60　strpbrkを参照してください。

[　例　]

```
#include <string.h>

char str1[11] = "abcbaccbac";
char str2[11] = "xxxabcbcac";
int  a, b;

void main (void)
{
  a = strcspn (str1, "abc");      /* a = 0 */
  b = strcspn (str2, "abc");      /* b = 3 */
}
```

13.52　strdup

[説　明]

Dupulicate string（文字列複製）

strdup は，str が指す文字列の複製コピーを作成します。メモリは malloc 関数で割り当て，free 関数で解放できます。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strdup (const void *src);
```

[パラメータ]

src　**複製元のバッファに対するポインタ**

[戻り値]

strdup **は，文字列** src **の複製に対するポインタを返します。複製した文字列を保存するメモリが不足している場合は，**NULL **を返します。**

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char *buffer;

void main (void)
{
  buffer = strdup (buffer1);
}
```

13.53　strerror

[説　　明]

Get error string **（エラー文字列）**

strerror **は，判読可能なエラー・メッセージにエラー番号を割り当てます。**DSP **ライブラリではエラー処理が実行されないため，**strerror **は著作権メッセージを返します。**

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

errnum　　**使用されない引数**

[戻り値]

著作権メッセージに対するポインタ

[　例　]

```
#include <string.h>

void main (void)
{
  printf ("%s", strerror (0));
}
```

13.54　strlen

[説　　明]

String length（文字列長）

strlen は，s が指す文字列（終端を表す NULL 文字を除く）の長さを計算します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

s　　長さを計算する文字列に対するポインタ

[戻り値]

文字で表した文字列の長さ。エラーを示す戻り値はありません。

[　例　]

```
#include <.h>

int i;

void main (void)
{
  i = strlen ("Hello");
}
```

13.55　strncmpi

[説　　明]

Compare strings caseinsensitive（文字列数の比較（大文字小文字の区別なし））

strncmpi は，s1 が指す n 文字までの文字列と s2 が指す文字列を，大文字小文字の区別をせずに比較します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

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

[パラメータ]

s1，s2	大文字小文字の区別をせずに比較する文字列に対するポインタ
n	比較する最大の文字数

[戻り値]

strncmpi は，s1 の指す文字列が s2 の指す文字列よりも小さい場合は 0 より小さい整数，s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。

[例]

```
#include <string.h>

char str1[11] = "abcedfghij";
char str2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = strncmpi (str1, str2, 5);
}
```

13.56　strncat

[説　　明]

Append strings（文字列数の追加）

strncat は，src が指す文字列の n 文字までをコピーして dst が指す文字列の最後に追加します。src の先頭文字は dst の末尾の NULL 文字を上書きします。この関数で新しいメモリは割り当てられないため，追加先の文字列には十分なスペースが必要です。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strncat (char *dst, const char *src, size_t n);
```

[パラメータ]

dst	追加先の文字列に対するポインタ
src	追加元の文字列に対するポインタ
n	追加する最大の文字数

[戻り値]

strncat は dst の元の値を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <.h>

char buffer[80];

void main (void)
{
  strcpy (buffer, "Hello ");
  strncat (buffer, "world!", 3);
}
```

13.57　strnchr

[説　　明]

Locate character in string（文字列数中の文字の検索）

stchr は，s が指す文字列の最初の n 文字で，最初に発生する c を探します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strnchr (const char *s, int c);
```

[パラメータ]

s　　検索する文字列に対するポインタ

c　　バッファで探す文字

[戻り値]

strnchr は最初に見つかった文字に対するポインタを返します。また，文字が文字列に出現しなかった場合は，NULL を返します。

[　例　]

```
#include <string.h>

char buffer[11] = "abcedfghij";
char *where;

void main (void)
{
  where = strchr (buffer, 'f');
}
```

13.58　strncmp

[説　明]

Compare strings（文字列数比較）

strncmp は，s1 が指す文字列と s2 が指す文字列の n 文字までを比較します。大文字小文字の区別をして比較します。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　文]

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

[パラメータ]

s1，s2	比較する文字列に対するポインタ
n	比較する最大の文字

[戻り値]

strncmp は，s1 の指す文字列が s2 の指す文字列よりも小さい場合は 0 より小さい整数，s1 と s2 が等しい場合は 0，s1 が s2 よりも大きい場合は 0 より大きい整数を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <string.h>

char str1[11] = "abcedfghij";
char str2[11] = "abcedxghij";
int  check;

void main (void)
{
  check = strncmp (str1, str2, 5);
}
```

13.59　strncpy

[説　明]

Copy strings（文字列数複写）

strncpy は，n 文字までの文字列を src が指すバッファから dst が指すバッファへコピーします。重複する文字列をコピーすると，正常に動作しない場合があります。重複するオブジェクトのコピーについては，13.33　memmoveを参照してください。コピー元の文字列を適合させるにはコピー先のバッファに十分なメモリがなくてはなりません。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strncpy (const void *dst, const void *src, size_t n);
```

[パラメータ]

dst	コピー先のバッファに対するポインタ
src	コピー元のバッファに対するポインタ
n	コピーする最大の文字数

[戻り値]

strncpy は dst の元の値を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <string.h>

char buffer1[11] = "abcedfghij";
char buffer2[11];
char *buffer;

void main (void)
{
  buffer = strncpy (buffer1, buffer2, 5);
}
```

13.60　strpbrk

[説　　明]

Find first occurrence of character（最初に現われる文字の検索）

strpbrk（文字列ポインタ区切り）は，str が指す文字列の charset が指す文字列で，最初に現われる文字の 1 つを検索します。str の末尾で終端を表す NULL 文字は，str の一部とみなしません。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strpbrk (const char *str, const char *charset);
```

[パラメータ]

str	文字列に対するポインタ
charset	比較文字セットを含む文字列に対するポインタ

[戻り値]

strpbrk は，str で最初に現われる charset に含まれる文字に対するポインタを返します。また，両方の文字列に共通の文字がない場合は，NULL を返します。最初に現われるインデクスを取得する方法については，13.51　strcspnを参照してください。

[　例　]

```
#include <string.h>

char str1[11] = "dfebaccbac";
char *s;

void main (void)
{
  s = strpbrk (str1, "abc");
}
```

13.61　strrchr

[説　　明]

Find last occurrence of character（最後に現われる文字の検索）

strrchr は，s が指す文字列で最後に現われる文字 c を探します。終端を表す NULL 文字は，文字列の一部とみなしません。

[ライブラリ・インクルード・ファイル]

string.h

[構　　文]

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

[パラメータ]

s　　検索する文字列に対するポインタ

c　　文字列で検索する文字

[戻り値]

strrchr は，見つかった文字に対するポインタを返します。また，文字列に文字が出現しなかった場合は，NULL を返します。

[　例　]

```
#include <string.h>

char buffer[11] = "abcedfghij";
char *where;

void main (void)
{
  where = strrchr (buffer, 'f');
}
```

13.62　strspn

[説　　明]

Length of the initial segment（先頭セグメント長）

strspn は，charset が指す文字列に含まれる文字で構成された str が指す文字列の先頭セグメント長を文字で計算します。str の末尾で終端を表す NULL 文字は，str の一部とみなしません。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
size_t strspn (const char *str, const char *charset);
```

[パラメータ]

str	文字列に対するポインタ
charset	比較文字セットを含む文字列に対するポインタ

[戻り値]

strspn は，charset の文字を含む先頭セグメントの長さを文字で返します。str の先頭文字が charset の文字でない場合は，NULL を返します。エラーを示す戻り値はありません。

[　例　]

```
#include <string.h>

char str1[11] = "abcbaccbxc";
char str2[11] = "xbxabcbcac";
int  a, b;

void main (void)
{
  a = strspn (str1, "abc");          /* a = 9 */
  b = strspn (str2, "abc");          /* b = 0 */
}
```

13.63　strstr

[説　　明]

Find substring in string（文字列中の文字列の検索）

strstr は，str が指す文字列で最初に現われる substr が指す文字列を検索します。str の末尾で終端を表す NULL 文字は，str の一部とみなしません。

[ライブラリ・インクルード・ファイル]

```
string.h
```

[構　　文]

```
char *strstr (const char *str, const char *substr);
```

[パラメータ]

str	文字列に対するポインタ
substr	str の中で検索する副次文字列に対するポインタ

[戻り値]

str の中で最初に現われる substr に対するポインタを返します。また，文字列が一致しない場合は NULL を返します。

[　例　]

```
#include <string.h>

char str1[11] = "abcbaccbac";
char *pos;

void main (void)
{
  pos = strstr (str1, "bac");
}
```

13.64　strtok

[説　　明]

Find next token（次のトークンを探す）

strtok は，s1 の中で次のトークンを探します。s2 にある文字セットは，現在の呼び出しで s1 から探すトークンに使用できる区切り文字を指定します。

[ライブラリ・インクルード・ファイル]

string.h

[構　　文]

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

[パラメータ]

s1	文字列に対するポインタ
s2	トークンの区切り文字の文字列に対するポインタ

[戻り値]

strtok は次のトークンに対するポインタを返します。また，トークンが見つからない場合は NULL を返します。

[備　　考]

strtok を最初に呼び出すと，関数は最初の区切り文字をスキップして s1 の最初のトークンが指すポインタを返し，トークンの最後に NULL 文字を付けます。複数のトークンがある場合，strtok への一連の呼び出しで s1 の残りの位置から抜けることができます。strtok を呼び出すたびに，その呼び出しによって返されたトークンの後に NULL 文字を挿入して s1 を修正します。s1 から次のトークンを読み込むには，s1 に対する NULL 値で strtok を呼び出す必要があります。これにより，strtok は s1 の残りの位置で次のトークンを探します。s2 は呼び出しごとに任意の値をとることができるため，区切り文字の設定を変更できます。文字列をトークンへ構文解析するために，strtok は静的変数を使用します。

strtok に対して複数または同時の呼び出しがあると，データが破損したり，不正確な結果になる可能性が高くなります。したがって，異なる文字列に対して strtok を同時に呼び出さないようにしてください。また，ループ内から strtok を呼び出すと，同じ関数を使用する別のルーチンが呼び出されることに注意してください。

[　例　]

```
#include <string.h>


char string[] = "A¥tstring of ,,tokens ";
char seps[] = " ,¥t¥n";
char *token;

void main (void)
{
  token = strtok ( string, seps); /* get first token: */
  while (token != NULL)         /* more tokens ? */
  {
    printf ( " %s¥n", token);
    token = strtok ( NULL, seps); /* Get next token: */
  }
}
```

13.65　strtol

[説　　明]

String to long（文字列を long 型に変換）

strtol は，ptr が指す文字列を long 型のオブジェクトに変換します。認識できない文字が現われた時点で変換を終了します。この認識できない文字に対するポインタは，endptr が NULL でない場合は endptr で返します。

base が ［2, 36］の範囲にある場合，endptr までの文字は基数 base で表現する数字として変換します。

base が 0 の場合は，ptr の先頭文字で変換に使用する基数を判断します。

先頭文字が 0x または 0X の場合，数字は 16 進数として扱われます。

同様に先頭文字が O の場合は 8 進数，それ以外の場合は 10 進数として数字を扱います。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

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

[パラメータ]

ptr　　変換する文字列に対するポインタ
endptr　認識できない文字が最初に現われた位置を保存するアドレス
base　　変換に使用する基数

[戻り値]

strtol は，ptr が指す文字列を long 型の数字として表現した値を返します。また，変換が正常に実行されなかった場合（オーバフローによって戻り値が不定の場合を含む）は，0 を返します。

[　例　]

```
#include <string.h>

char str1[10] = "13123";
long result;

void main (void)
{
  result = strtol (str1, NULL, 10);
}
```

13.66　strtoul

[説　　明]

String to unsigned long（文字列を unsigned long 型に変換）

strtoul は，ptr が指す文字列を unsigned long 型のオブジェクトに変換します。認識できない文字が最初に現われた位置で変換を終了します。この認識できない文字に対するポインタは，endptr が NULL でない場合は endptr で返されます。

base が［2, 36］の範囲にある場合，endptr までの文字は基数 base で表現する数字として変換します。

base が 0 の場合は，ptr の先頭文字で変換に使用する基数を判断します。

先頭文字が 0x または 0X の場合は，数字は 16 進数として扱われます。

同様に先頭文字が O の場合は 8 進数，それ以外の場合は 10 進数として数字を扱います。

[ライブラリ・インクルード・ファイル]

```
stdlib.h
```

[構　　文]

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

[パラメータ]

ptr　　変換する文字列に対するポインタ
endptr　認識できない文字が最初に現われた位置を保存するアドレス
base　　変換に使用する基数

[戻り値]

strtol は，ptr が指す文字列を unsigned long 型の数字として表現した値を返します。また，変換が正常に実行されなかった場合（オーバフローによって戻り値が不定の場合を含む）は，0 を返します。

[　例　]

```
#include <string.h>

char str1[10] = "13123";
unsigned long result;

void main (void)
{
  result = strtoul (str1, NULL, 10);
}
```

13.67　tolower

[説　　明]

Make lowercase（英小文字変換）

tolower は，文字 c が英大文字で表現されている場合は小文字に変換し，それ以外の場合は変更せずに c を返します。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　　文]

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

[パラメータ]

c　　文字

[戻り値]

小文字で表現した c。エラーを示す戻り値はありません。

[　例　]

```
#include <ctype.h>

char x;

void main (void)
{
  x = tolower ('R');
}
```

13.68　toupper

[説　明]

Make uppercase（英大文字変換）

toupper は，文字 c が英小文字で表現されている場合は大文字に変換し，それ以外の場合は変更せずに c を返します。

[ライブラリ・インクルード・ファイル]

```
ctype.h
```

[構　文]

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

[パラメータ]

c　　文字

[戻り値]

大文字で表現した c。エラーを示す戻り値はありません。

[　例　]

```
#include <ctype.h>

char x;

void main (void)
{
  x = toupper ('r');
}
```

13.69　va_arg

[説　明]

Variable argument-list（可変個引数リスト）

va_arg，va_end，va_start マクロを使用すると，引数に変数値をとる関数の引数に簡単にアクセスできます。va_arg は，必ず対応するマクロ va_end および va_start と合わせて使用します。

[ライブラリ・インクルード・ファイル]

```
stdarg.h
```

[構　文]

```
type va_arg (va_list param, type);
```

[パラメータ]

param	（va_start マクロで初期化した）可変引数リスト
type	引数の型

[戻り値]

va_arg は，param で指定した位置からデータ型 type の値を取り出し，次の引数を指すように param をインクリメントします（ここでのインクリメントは，type のサイズから次の引数の開始位置を与える動作です）。va_arg は，リストから引数を取り出す関数内で何度でも使用できます。

[　例　]

```
#include <stdarg.h>

void check (int num, ...)
{
  char *p1;
  char *p2;

  auto va_list ap;
  va_start (ap, row);

  p1 = va_arg (ap, char *);
  p2 = va_arg (ap, char *);

  /* use p1 and p2 here */

  va_end (ap);
}

void main (void)
{
  check (1, "ad", "tr");
}
```

13.70　va_end

[説　　明]

Variable argument-list end（可変個引数リストの終了）

va_arg，va_end，va_start マクロを使用すると，引数に変数値をとる関数の引数に簡単にアクセスできます。va_end は，必ず対応するマクロ va_start および va_arg と合わせて使用します。引数をすべて取り出すと，va_end はポインタを NULL にリセットします。

[ライブラリ・インクルード・ファイル]

```
stdarg.h
```

[構　　文]

```
void va_end (va_list param);
```

[パラメータ]

param　　（va_start マクロで初期化した）可変個引数リスト

[　例　]

```
#include <stdarg.h>

void check (int num, ...)
{
  char *p1;
  char *p2;

  auto va_list ap;
  va_start (ap, row);

  p1 = va_arg (ap, char *);
  p2 = va_arg (ap, char *);

  /* use p1 and p2 here */

  va_end (ap);
}

void main (void)
{
  check (1, "ad", "tr");
}
```

13.71　va_start

[説　　明]

Variable argument-list start args（可変個引数リストの開始）

va_arg，va_end，va_start マクロを使用すると，引数に変数値をとる関数の引数に簡単にアクセスできます。va_start は param を，関数へ渡された引数リストの第 1 任意引数に設定します。このとき param は va_list データ型である必要があります。引数 previous は，引数リストの第 1 任意引数の必須パラメータ名です。previous を保存クラス register で宣言すると，マクロの動作は不定です。va_start は必ず va_arg を最初に使用する前に使用します。

[ライブラリ・インクルード・ファイル]

```
stdarg.h
```

[構　　文]

void va_start (va_list *param*, *previous*);

[パラメータ]

param	（va_start マクロで初期化した）可変個引数リスト
previous	「...」という表記の直前の関数定義における引数

[　例　]

```
#include <stdarg.h>

void check (int num, ...)
{
  char *p1;
  char *p2;

  auto va_list ap;
  va_start (ap, row);

  p1 = va_arg (ap, char *);
  p2 = va_arg (ap, char *);

  va_end (ap);
}

void main (void)
{
  check (1, "ad", "tr");
}
```

13.72　vfprintf

[説　　明]

Print to file with variable argument-list（可変個引数リスト・ファイル書式印字）

vfprintf は，出力ストリームに書き込む文字列と値をフォーマット設定して，出力します。本 C コンパイラは，stdout および stderr のみをサポートします。フォーマット文字列の規則については13.36　printfを参照してください。vfprintf は，可変個引数リストを arg に置き換えると，fprintf と同等です。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int vfprintf (FILE *fp, const char *format, va_list arg);
```

[パラメータ]

fp	出力ストリームに対するポインタ
format	フォーマット文字列（13.36　printfを参照）
arg	初期化された可変個引数リスト

[戻り値]

vfprintf は，ストリームに書き込まれた文字数を返します。出力エラーが発生した場合は，負の値を返します。

[備　　考]

本 C コンパイラの vfprintf **関数は，シミュレータの** stdout **または** stderr **へのみ出力できます。正確な動作にはタイミング・ファイル** dspio.tmg **が必要です。ファイル出力の詳細については，第** 10 **章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

void errmsg (char *format, ...);
{
  va_list arglist;
  va_start (arglist,format);
  vfprintf (stderr, format, arglist);
  va_end (arglist);
}

void main (void)
{
  errmsg ("%s, %d", "ERR", 1);
}
```

13.73　vprintf

[説　　明]

Print to stdout with variable argument-list**（可変個引数リストによる標準出力への印刷）**

vprintf **は，引数** format **の制御下で，**stdout **が指すストリームに出力を書き込みます。フォーマット文字列については**13.36　printf **を参照してください。**vprintf **は，可変個引数リストを** arg **に置き換えると，**printf **と同等です。**

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

```
int vprintf (const char *format, va_list arg);
```

[パラメータ]

format	**フォーマット制御文字列（**13.36　printf**を参照）**
arg	**初期化された可変引数リスト**

[戻り値]

vprintf **はストリームに書き込まれた文字数を返します。出力エラーが発生した場合は，負の値を返します。**

[備　　考]

本 C コンパイラの vprintf **関数は，シミュレータの** stdout **または** stderr **へのみ出力できます。正確な動作にはタイミング・ファイル** dspio.tmg **が必要です。ファイル出力の詳細については，第** 10 **章　文字出力**を参照してください。

[　例　]

```
#include <stdio.h>

 void stdmsg (char *format, ...);
{
  va_list arglist;
  va_start (arglist,format);
  vprintf (format, arglist);
  va_end (arglist);
}

void main (void)
{
  stdmsg ("%s, %d", "WARNING", 1);
}
```

13.74　vsprintf

[説　　明]

Print to string with variale argument-list（可変個引数リストによる文字列の印刷）

vsprintf は，引数 format の制御下で，buf が指すバッファに出力を書き込みます。フォーマット文字列については 13.36　printf を参照してください。vsprintf は，可変個引数リストを arg に置き換えると，sprintf と同等です。

[ライブラリ・インクルード・ファイル]

```
stdio.h
```

[構　　文]

int vsprintf(char **fp*, const char **format*, va_list *arg*);

[パラメータ]

fp	出力するバッファに対するポインタ
format	フォーマット文字列（13.36　printfを参照）
arg	初期化された可変個引数リスト

[戻り値]

vsprintf は，書き込まれた文字数を返します。出力エラーが発生した場合は，負の値を返します。

[例]

```
#include <stdio.h>

void msgstring (char *string, char *format, ...);
{
  va_list arglist;
  va_start (arglist,format);
  vsprintf (string, format, arglist);
  va_end (arglist);
}

void main (void)
{
  char buffer[80];

  errmsg (buffer, "%s, %d", "ERR", 1);
}
```

第14章　DSP ライブラリの関数レファレンス

14.1　circinc

[説　明]

Increment circular pointer（循環ポインタのインクリメント）

circinc マクロはポインタ引数をとり，値 inc をインクリメントして，増加したポインタを返します。

同じ基礎ブロックの同じポインタに対するメモリ・アクセス文のあとに circinc マクロがある場合は，これらの文を連結してμPD77016 アーキテクチャのモジュロ・インデクス追加後のアドレス指定モードにコンパイルします。循環バッファのアクセスと呼び出し circinc を連結した文の前には，対応する呼び出し setmod が必要となることに注意してください。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　文]

```
void  circinc   (type, __circ     type *pd, int inc);
void  circinc_x (type, __circ __X type *px, int inc);
void  circinc_y (type, __circ __Y type *py, int inc);
```

[パラメータ]

type	第 2 引数が指す循環バッファ要素のデータ型。循環バッファの操作は，μPD77016 アーキテクチャの汎用レジスタ（40 ビット）に完全に収まるデータ型，つまり，すべての整数データ型，__fixed 型，long __fixed 型，long __accum 型に応じて制限されます。
pd	ディフォールトのメモリに対する循環ポインタ
px	X メモリに対する循環ポインタ
py	Y メモリに対する循環ポインタ
inc	inc*sizeof (type) が［−32768, 32767］の範囲に収まり，かつ対応するモジュロ・レジスタ（DMX, DMY）のモジュロ値よりも小さくなるようなインクリメント値。

[戻り値]

px + inc*sizeof (type) などの増加したポインタ

[　例　]

```
#include <dspext.h>

int get_value (__circ __Y int *cp);

__circ __Y int buf[3] = {1,2,3};

int get_value (__circ __Y int *cp)
{
  int data;

  data = *cp;
  circinc_y (int, cp, -1);
  return data;
}

void main (void)
{
  setmod_y (2*sizeof(int));
  data =  get_value (buf);      /* data = 1 */
  data += get_value (buf);      /* data = 1+3 = 4 */
}
```

14.2　circlength

[説　　明]

Set modulo register（モジュロ・レジスタの設定）

circlength マクロは，μPD77016 アーキテクチャのモジュロ・レジスタ DMX，DMY を初期化します。

この初期化は，必ず同じメモリ領域内の循環バッファに対するポスト・モジュロ・インデクス追加アクセスの前に行う必要があります。circlength マクロは，ディフォールトのメモリ領域の循環バッファで使用するように指定されています。circlength マクロは，コンパイラに渡す前の処理で適切な setmod 関数へ展開されます。同じメモリ領域内の，互いにサイズの等しい循環バッファ間でのモジュロ・アクセスについては，対応するモジュロ・レジスタの初期化は一度行うだけでかまいません。

[ライブラリ・インクルード・ファイル]

dspext.h

[構　　文]

```
void  circlength   (type, unsigned int n);
void  circlength_x (type, unsigned int nx);
void  circlength_y (type, unsigned int ny);
```

[パラメータ]

type　後でアクセスする循環バッファ要素のデータ型。循環バッファの操作は，データ型がμPD77016 アーキテクチャの汎用レジスタに完全に収まらなければならない，という点において制約を受けます。

n，nx，ny　後でアクセスする循環バッファのベクトル長（バッファの要素数）。マクロは，μPD77016 アーキテクチャにしたがって n*sizeof (type) －1 に対するモジュロ値を計算します。正しい演算のためには，モジュロ値は必ず［1, 32767］の範囲内の値にします。

[　例　]

```
#include <dspext.h>

__circ __Y __fixed cb[5];

void main (void)
{
  circlength_y (__fixed, 5);  /* DMY = 4 */
}
```

14.3　fixed_int

[説　　明]

Interpret fixed data as int data（データを__fixed 型から int 型に変換）

fixed_int は引数を int として返します。この関数は，引数のビット・パターンを変更しません。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
int  fixed_int (__fixed x);
```

[パラメータ]

x　　16 ビット長の固定小数点値

[戻り値]

fixed_int は，整数値に変換した引数のビット・パターンを返します。

[　例　]

```
#include <dspext.h>

void main (void)
{
  __fixed x = 0.00390625r;  /* x = 2^-8    */
  int result;

  result = fixed_int (x);   /* result = 2^7 */
  result >>= 7;             /* result = 1   */
}
```

★ 14.4　get_eir

[説　明]

Read interrupt enable flag stack register（割り込み許可フラグ・スタック・レジスタの値を読み出し）

get_eir は，割り込み許可フラグ・スタック・レジスタの値を unsigned int 型で返します。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　文]

```
unsigned int get_eir(void) ;
```

[戻り値]

get_eir は，割り込み許可フラグ・スタック・レジスタの値を返します。

[　例　]

```
#include <dspext.h>
void main (void)
{
  unsigned int x = get_eir();
}
```

★ 14.5　get_sr

[説　明]

Read interrupt status register（割り込みステータス・レジスタの値を読み出し）

get_sr は，割り込みステータス・レジスタの値を unsigned int 型で返します。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構文]

```
unsigned int get_sr (void);
```

[戻り値]

get_sr は，割り込み許可フラグ・スタック・レジスタの値を返します。

[　例　]

```
#include <dspext.h>
void main (void)
{
  unsigned int x = get_sr();
}
```

14.6　int_fixed

[説　　明]

Interpret int data as fixed data（データを int 型から__fixed 型に変換）

int_fixed は引数のビット・パターンを変更しません。データ型を int 型から__fixed 型に変換するだけです。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
__fixed  int_fixed (int x);
```

[パラメータ]

x　　整数値

[戻り値]

int_fixed は，固定小数点値に変換した引数のビット・パターンを返します。

[　例　]

```
#include <dspext.h>

void main (void)
{
  int x = 128;            /* x = 2^7      */
  int result;

  result = int_fixed (x);  /* result = 2^-8 */
}
```

14.7　laccum_sll

[説　　明]

Logical left shift of 40 bit data（40 ビット論理左シフト）

laccum_sll は，引数 shift を使って 40 ビット値 x の論理左シフトを行います。論理左シフトは，シフト量だけ LSB から 0 を挿入されることを意味します。シフト量は 0-39 まで設定できます。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
long __accum  laccum_sll (long __accum x, unsigned int shift);
```

[パラメータ]

x　　long __accum 型の値

shift　x のシフト量［0, 39］の範囲内の値

[戻り値]

long __accum として解釈されたシフト・ビット・パターン

[　例　]

```
#include <dspext.h>
__long accum x = 255.0A;
void main (void)
{
   long __accum result;
  result = laccum_sll (x, 1);   /* result = -2.0A */
}
```

14.8　laccum_sra

[説　　明]

Arithmetical right shift of 40 bit data（40 ビット算術右シフト）

laccum_sra は，引数 shift を使って 40 ビット値 x の算術右シフトを行います。算術右シフトは，シフト量だけ MSB から符号ビットを挿入（符号拡張）されることを意味します。シフト量は 0-39 まで設定できます。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
long __accum  laccum_sra (long __accum x, unsigned int shift);
```

[パラメータ]

x　　long __accum 型の値

shift　x のシフト量［0, 39］の範囲内の値

[戻り値]

long __accum として解釈されたシフト・ビット・パターン

[　例　]

```
#include <dspext.h>
__long accum x = -1.0A;
void main (void)
{
  long __ accum result;
  result = laccum_sra (x, 1);   /* result = -0.5A */
}
```

14.9　lfixed_long

[説　　明]

interpret long fixed data as long dta（データを long __fixed 型から long 型に変換）

lfixed_long は引数のビット・パターンを変更しません。データ型を long __fixed 型から long 型に変換するだけです。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
long  lfixed_long (long __fixed x);
```

[パラメータ]

x　　32 ビット長の固定小数点値

[戻り値]

lfixed_long は，long 型に変換した引数のビット・パターンを返します。

[　例　]

```
#include <dspext.h>

void main (void)
{
  long __fixed x = 0.00390625r;   /* x = 2^-8    */
  long result;

  result = lfixed_long (x);      /* result = 2^7 */
  result >>= 7;                  /* result = 1   */
}
```

14.10　long_lfixed

[説　　明]

Interpret long data as long fixed data（データを long 型から long __fixed 型に変換）

long_lfixed は引数のビット・パターンを変更しません。データ型を long 型から long __fixed 型に変換するだけです。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
long __fixed  long_lfixed (long x);
```

[パラメータ]

x　　32 ビット長の整数値

[戻り値]

long_lfixed は，固定小数点値に変換した引数のビット・パターンを返します。

[　例　]

```
#include <dspext.h>

void main (void)
{
  long x = 128;               /* x = 2^7       */
  long __fixed result;

  result = long_lfixed (x); /* result = 2^-8 */
}
```

14.11　rabs

[説　　明]

Absolute value（絶対値の計算）

rabs は所定の引数 long __accum の絶対値を返します。rabs は適切な文字の型を用いることで，すべての固定小数点データ型に使用できます。ただし，ソース・コードの判読が困難なため，代わりに rrabs マクロを使用することをお勧めします。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rabs (long __accum x);
type  rrabs (type, type val)
```

[パラメータ]

x　　絶対値を計算する long __accum 値

val　絶対値を計算する固定小数点値

type　固定小数点データ型（__fixed，long __fixed，long __accum）

[戻り値]

rabs は x の絶対値を返し，rrabs は val の絶対値を返します。

[　例　]

```
#include <rmath.h>

void main()
{
  __long accum result1;
  __fixed     result2;

  result1 = rabs(-0.707A);          /* result1 = 0.707A */
  result2 = rrabs(__fixed, -0.5r)  /* result2 = 0.5r   */
}
```

14.12　racos

[説　　明]

Arc cosine（逆余弦）

racos は所定の引数の逆余弦を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  racos (long __fixed x);
```

[パラメータ]

x　　逆余弦を計算する値

[戻り値]

racos は，ラジアンで示す値が［0, π］の範囲にある x の逆余弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = racos (0.707R);    /* result = 0.7855 rad */
}
```

14.13　racosh

[説　　明]

Area hyperbolic cosine（領域双曲線余弦）

racosh は所定の引数の領域双曲線余弦を返します。

[ライブラリ・インクルード・ファイル]

rmath.h

[構　　文]

```
long __accum  racosh (long __accum x);
```

[パラメータ]

x　　領域双曲線余弦を計算する［1.0A, 15.967A］の範囲内の値。このパラメータが上限を越える場合は，演算エラー変数を ME_OVERFLOW に設定します。パラメータが 1.0A よりも小さい場合は，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

racosh は x の領域双曲線余弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = racosh (1.5A);     /* result = 0.9624A */
}
```

14.14　rasin

[説　　明]

Arc sine（逆正弦）

rasin は所定の引数の逆正弦を返します。

[ライブラリ・インクルード・ファイル]

rmath.h

[構　　文]

```
long __accum  rasin (long __fixed x);
```

[パラメータ]

x　　逆正弦を計算する値

[戻り値]

rasin は，ラジアンで示す値が［0, π］の範囲にある x の逆正弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rasin (0.707R);       /* result = 0.7852 rad */
}
```

14.15　rasinh

[説　　明]

Area hyperbolic sine（領域双曲線正弦）

rasinh は所定の引数の領域双曲線正弦を返します。

[ライブラリ・インクルード・ファイル]

rmath.h

[構　　文]

```
long __accum  rasinh (long __accum x);
```

[パラメータ]

x　　領域双曲線正弦を計算する［－15.967A, 15.967A］の範囲内の値。このパラメータが制限を越える場合は，演算エラー変数を ME_OVERFLOW に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

rasinh は x の領域双曲線正弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rasinh (-1.5A);      /* result = -1.1948A */
}
```

14.16　ratan

[説　　明]

Arc tangent（逆正接）

ratan は所定の引数の逆正接を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  ratan (long __accum x);
```

[パラメータ]

x　　逆正接を計算する値

[戻り値]

ratan は，ラジアンで示す値が［−π/2, π/2］の範囲にある x の逆正接を返します。x が 0 の場合は，ratan は−π/2 を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = ratan (0.707A);      /* result = 0.6154 rad */
}
```

14.17　ratan2

[説　　明]

Arc tangent(full quadrant)（逆正接（全象限））

ratan2 は正確な象限情報を持つ逆正接を返します。x が 0 の場合，ratan2 は π/2*sign (y) を返します。両方のパラメータが 0 の場合，関数は 0 を返し，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherr　を参照してください）。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  ratan2 (long __accum y, long __accum x);
```

[パラメータ]

x, y　引数を複合値 x ＋ yj の座標に変換し，正の実軸から測定した角度（単位円の弧の長さ）を計算します。

[戻り値]

ratan2 は，ラジアンで示す値が［−π, π］の範囲にある比率 y/x の逆正接を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = ratan2 (-3.0A, 2.0A);      /* result = -0.9828 rad */
}
```

14.18　ratanh

[説　　明]

Area hyperbolic tangent（領域双曲線正接）

ratanh は所定の引数の領域双曲線正接を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  ratanh (long __fixed x);
```

[パラメータ]

x　　領域双曲線正接を計算する値

[戻り値]

ratanh は x の領域双曲線正接を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = ratanh (-0.5A);     /* result = -0.5493A */
}
```

14.19　rclip

[説　　明]

Clip long __accum data to long __fixed data（データ型を long __accum から long __fixed に切り落とす）

rclip は 40 ビット長の引数を 32 ビット長に変換します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __fixed  rclip (long __accum x);
```

[パラメータ]

x　　40 ビット長の固定小数点値

[戻り値]

x が［－1.0A, 1.0A］の範囲にある場合，rclip は x の小数部分を返します。それ以外の場合は，x の記号に応じて，それぞれの制限値－1.0R または (1.0－2^{-31})R を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  long __accum x = -2.5A;
  long __fixed result;

  result = rclip (x);     /* result = -1.0R */
}
```

14.20　rcos

[説　　明]

Cosinus（余弦）

rcos は所定の引数の余弦を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rcos (long __accum x);
```

[パラメータ]

x　　余弦を計算するラジアンで示した値

[戻り値]

rcos は［－1.0A, 1.0A］の範囲にある x の余弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rcos (-1.5A);     /* result = 0.0707A */
}
```

14.21　rcosh

[説　　明]

Hyperbolic cosine（双曲線余弦）

rcosh は所定の引数の双曲線余弦を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rcosh (long __accum x);
```

[パラメータ]

x　　双曲線余弦を計算する値

[戻り値]

rcosh は x の双曲線余弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rcosh (-1.5A);    /* result = 2.3524A */
}
```

14.22　reverse

[説　　明]

Reverse bits of argument（引数のビット反転）

reverse マクロはポインタ引数をとり，ビットの逆順序でポインタを返します。メモリ・アクセス文でビット反転マクロを使用し，同じ基礎ブロックで引数ポインタをインクリメントすると，この連結した文はμPD77016 アーキテクチャのビット反転前およびインデクス追加後のアドレス指定モードにコンパイルします。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　　文]

```
__circ      type  *reverse   (type, __circ      type *pd);
__circ __X type  *reverse_x (type, __circ __X type *px);
__circ __Y type  *reverse_y (type, __circ __Y type *py);
```

[パラメータ]

type　第 2 引数が指す循環バッファ要素のデータ型。ビット反転アドレス指定は，16 ビット・データ型，つまり，long 型と unsigned long 型を除くすべての整数データ型と__fixed 型の制約を受けます。

pd　ディフォールトのメモリに対する循環ポインタ

px　X メモリに対する循環ポインタ

py　Y メモリに対する循環ポインタ

[戻り値]

ビット反転循環ポインタ引数を返します。

[　例　]

```
#include <dspext.h>

__circ __X int a[2] = {1,2};

void main (void)
{
  __circ __X int *pbr;
  int result;

  pbr = reverse_x (int, &a);          /* reverse address of a   */

  result  = *(reverse_x (int, pbr));  /* fetch first element    */
  pbr    += 0x8000;                   /* of a and increment prb */
  result += *(reverse_x (int, prb));  /* result = 1+2 = 3      */
  prb    -= 0x8000;                   /* reset prb             */
}
```

14.23　rexp

[説　　明]

Exponent（指数関数）

rexp は e^x を計算します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rexp (long __accum x);
```

[パラメータ]

x　　［－256.0A, 5.0A］の範囲内の引数値。引数が上限値よりも大きい場合は，演算エラー変数を ME_OVERFLOW に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

rexp は，e = 2.71828...の場合に e^x を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rexp (-0.5A);    /* result = 0.6065A */
}
```

14.24　rexponent

[説　　明]

Calculate number of sign bits（符号ビット）

rexponent は，32 ビット引数のバイナリ・ポイントの後の符号ビット数を与えます。この命令は，固定小数点値の底数 2 に対する対数を計算する高速手順の一部として使用できます。

[ライブラリ・インクルード・ファイル]

rmath.h

[構　　文]

```
unsigned int  rexponent (long __fixed x);
```

[パラメータ]

x　　32 ビット長の固定小数点値

[戻り値]

rexponent は引数のバイナリ・ポイントの後の符号ビット数を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  long __fixed x = 0.125R;
  unsigned int result;

  result = rexponent (x);     /* result = 2ul */
}
```

14.25　rlog

[説　明]

Natural Logarithm（自然対数）

rlog は所定の引数の自然対数を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　文]

```
long __accum  rlog (long __accum x);
```

[パラメータ]

x　自然対数を計算する［0.0A, 256.0A］の範囲内の値。引数がこの制限を越える場合は，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

rlog は x の自然対数を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rlog (5.0A);      /* result = 1.6094A */
}
```

14.26　rlogx

[説　明]

Genaral Logarithm（一般対数）

rlogx は指定された底に対する所定の引数の対数を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　文]

```
long __accum  rlogx (long __accum val, long __accum base);
```

[パラメータ]

val　対数を計算する 0 より大きい値。この引数が 0 以下の場合は，演算エラー変数を ME_ILLEGAL に設定します。

base　底を指定する［2.0A, 256.0A］の範囲内の値。このパラメータが 2 より小さい場合は，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

rlogx　底数 base に対する val の対数を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rlogx (8.0A, 2.0A);    /* result = 3.0A */
}
```

14.27　rmatherr

[説　　明]

Math library error checking（演算ライブラリ・エラーの確認）

rmatherr は演算エラー変数のステータスを返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
int  rmatherr (void);
```

[戻り値]

rmatherr は内部演算エラー変数の実際のステータスを返し，値を ME_NOERROR にリセットします。

演算エラー変数は ME_NOERROR に初期化され，幾つかの演算処理実行間のエラーを示す値に設定できます。戻り値は次のとおりです。

ME_NOEROR　エラーは発生していない

ME_OVERFLOW　オーバフローが発生

ME_INFINITE　最後の演算結果が無限大

ME_ILLEGAL　最後の演算が不正なオペランドで呼び出された

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;
  int        error;

  result = rsqrt (-4.0A); /* try to calculate the square */
                          /* root from a negative number */
  error = rmatherr();     /* error = ME_ILLEGAL        */
}
```

14.28　rpow

[説　　明]

Power（累乗）

rpow は y^x を計算します。

[ライブラリ・インクルード・ファイル]

rmath.h

[構　　文]

```
long __accum  rpow (long __accum y, long __accum x);
```

[パラメータ]

y　　［0.0A, 256.0A］の範囲内の値。このパラメータが 0 より小さい場合は，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

x　　Y を累乗する［−256.0A, 5.0A］の範囲内の値。この引数がこの制限を越える場合は，演算エラー変数を ME_OVERFLOW に設定します。

[戻り値]

rpow は y^x を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

result = rpow (2.0A, 4.0A);    /* result = 2^4 = 16.0A */
}
```

14.29　rround

[説　　明]

Clip and round long __accum data to__fixed data（long __accum データを切り離し__fixed データに丸める）

rround は 40 ビット引数の整数部分を切り離し，下位ワードを 16 ビットに丸めます。丸め処理は，引数の小数部分 2^{-16} を加算し，この加算結果の下位ワードを切り離すことによって行います。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __fixed  rround (long __accum x);
```

[パラメータ]

x　　40 ビット長の固定小数点値

[戻り値]

x が［−1.0A, 1.0A］の範囲にある場合，rround は 16 ビット長に丸めた x の小数部分を返します。それ以外の場合は，x の符号に応じて，それぞれの飽和値−1.0r または $(1.0-2^{-15})$r を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  long __accum x = 0.0000152587890625A   /* x = 2^-16A */
  __fixed     result;

  result = rround (x);                /* result = 2^-15r */
}
```

14.30　rsin

[説　　明]

Sinus（正弦）

rsin は所定の引数の正弦を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rsin (long __accum x);
```

[パラメータ]

x　　正弦を計算する値

[戻り値]

risin は［－1.0A, 1.0A］の範囲にある x の正弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rsin (-1.5A);     /* result = -0.9975A */
}
```

14.31　rsinh

[説　　明]

Hyperbolic sine（双曲線正弦）

rsinh は所定の引数の双曲線正弦を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rsinh (long __accum x);
```

[パラメータ]

x　　双曲線正弦を計算する値

[戻り値]

rasinh は x の双曲線正弦を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rsinh (-1.5A);     /* result = -2.1293A */
}
```

14.32　rsqrt

[説　　明]

Square root（平方根）

rsqrt は所定の引数の平方根を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rsqrt (long __accum x);
```

[パラメータ]

x　　平方根を計算する［0.0A, 256.0A］の範囲内の値。このパラメータが 0 よりも小さい場合は，演算エラー変数を ME_ILLEGAL に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[戻り値]

rsqrt は x の平方根を返します。

[　例　]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rsqrt (1.5A);       /* result = 1.2247A */
}
```

14.33　rtan

[説　　明]

Tangent（正接）

rtan は所定の引数の正接を返します。

[ライブラリ・インクルード・ファイル]

```
rmath.h
```

[構　　文]

```
long __accum  rtan (long __accum x);
```

[パラメータ]

x　　正接を計算する値

[戻り値]

rtan は x の正接を返します。計算結果の絶対値が long __accum の最大値より大きい場合，x を$\pm(256-2^{-31})$A に設定し，演算エラー変数を ME_OVERFLOW に設定します（エラーの確認方法については，14.27　rmatherrを参照してください）。

[例]

```
#include <rmath.h>

void main (void)
{
  __long accum result;

  result = rtan (0.5A);       /* result = 0.5463A */
}
```

★ 14.34　set_eir

[説　明]

Write interrupt enable flag stack register（割り込み許可フラグ・スタック・レジスタ設定）

set_eir は，割り込み許可フラグ・スタック・レジスタの値を設定します。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　文]

```
void set_eir (unsigned int x);
```

[パラメータ]

x　　割り込み許可フラグ・パターン

[例]

```
#include <dspext.h>
void main (void)
{
  unsigned int x = 0xafffu;
  set_eir (x);
}
```

★ 14.35　set_sr

[説　明]

Write interrupt status register（割り込みステータス・レジスタ設定）

set_sr は，割り込みステータス・レジスタの値を設定します。

[ライブラリ・インクルード・ファイル]

```
dspext.h
```

[構　文]

```
void set_sr (unsigned int x);
```

[パラメータ]

x　　**割り込みステータス・ビット・パターン**

[　例　]

```
#include <dspext.h>
void main (void)
{
  unsigned int x = 0xcfefu
  set_sr (x);
}
```

【発　行】
NECエレクトロニクス株式会社
〒211-8668　神奈川県川崎市中原区下沼部1753
電話（代表）：044(435)5111

【ホームページ】

NECエレクトロニクスの情報がインターネットでご覧になれます。

URL（アドレス）　http://www.necel.co.jp/

【営業関係お問い合わせ先】

下記のページに最新版のお問い合わせ先が記載されています。

URL（アドレス）　http://www.necel.com/ja/contact/contact_j.html

【技術的なお問い合わせ先】

半導体テクニカルホットライン
（電話：午前 9:00～12:00，午後 1:00～5:00）

電　話　：044-435-9494
FAX　：044-435-9608
E-mail　：info@lsi.nec.co.jp

【資料請求先】

NECエレクトロニクス特約店または上記ホームページ記載の営業関係お問い合わせ先へお申し付けください。