お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010 年 4 月 1 日
ルネサスエレクトロニクス株式会社

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

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

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

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

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

ご注意書き

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

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

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

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

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

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

ユーザーズ・マニュアル

CA850 Ver.3.20

Cコンパイラ・パッケージ

C言語編

対象デバイス
V850シリーズ

資料番号 U18513JJ1V0UM00（第1版）
発行年月 May 2007 CP(K)

〔メ　モ〕

目次要約

Windows**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。**

- 本資料に記載されている内容は2007年5月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品の使用に関連し発生した第三者の特許権，著作権その他の知的財産権の侵害等に関し，一切その責を負いません。当社は，本資料に基づき当社または第三者の特許権，著作権その他の知的財産権を何ら許諾するものではありません。
- 本資料に記載された回路，ソフトウエアおよびこれらに関する情報は，半導体製品の動作例，応用例を説明するものです。お客様の機器の設計において，回路，ソフトウエアおよびこれらに関する情報を使用する場合には，お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し，当社は，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。当社製品の不具合により生じた生命，身体および財産に対する損害の危険を最小限度にするために，冗長設計，延焼対策設計，誤動作防止設計等安全設計を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「特定水準」に分類しております。また，各品質水準は，以下に示す用途に製品が使われることを意図しておりますので，当社製品の品質水準をご確認ください。

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

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

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

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

（注）

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

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

M8E　02.11

はじめに

対象デバイス　V850シリーズCコンパイラ・パッケージは，NECエレクトロニクス製RISCマイクロプロセッサV850シリーズ用のオブジェクト・コードを生成するためのCコンパイラ・パッケージです。

対 象 者　このマニュアルは，V850シリーズコンパイラ・パッケージを使用して，アプリケーション・システムを開発するユーザを対象としています。

目　　的　このマニュアルでは，パッケージに含まれるCコンパイラ（ca850）がサポートするC言語仕様について説明しています。

構　　成　このマニュアルは，次の内容で構成されています。

- 概要
- 基本言語仕様
- コンパイル実行時の環境
- 拡張言語仕様
- プログラムの呼び出し
- スタート・アップ・ルーチン
- ライブラリ関数
- より効果的に用いるために

読み方の注意　・このマニュアルでは，V850シリーズでも“ V850E ”に特有のおもな部分については，タイトル名や“ 【V850E】 ”などで示しています。

関連資料　**このマニュアルを使用する場合は，次の資料もあわせてご覧ください。**

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。

○開発ツールに関する資料（ユーザーズ・マニュアル）

資　料　名		資料番号	
		和文	英文
CA850 Ver.3.20 **Cコンパイラ・パッケージ**	**操作編**	U18512J	U18512E
	C言語編	**このマニュアル**	U18513E
	アセンブリ言語編	U18514J	U18514E
	リンク・ディレクティブ編	U18515J	U18515E
PM+ Ver.6.30 **プロジェクト・マネージャ**		U18416J	U18416E
ID850 Ver.3.00 **統合ディバッガ**	**操作編**	U17358J	U17358E
ID850NW Ver.3.10 **統合ディバッガ**	**操作編**	U17369J	U17369E
ID850QB Ver.3.20 **統合ディバッガ**	**操作編**	U17964J	U17964E
SM+ **システム・シミュレータ**	**操作編**	U17246J	U17246E
	ユーザ・オープン・インタフェース編	U18212J	U18212E
SM850 Ver.2.50 **システム・シミュレータ**	**操作編**	U16218J	U16218E
SM850 Ver.2.00**以上　システム・シミュレータ**	**外部部品ユーザ・オープン・インタフェース仕様編**	U14873J	U14873E
RX850 Ver.3.20**以上　リアルタイム**OS	**基礎編**	U13430J	U13430E
	インストレーション編	U17419J	U17419E
	テクニカル編	U13431J	U13431E
	タスク・デバッガ編	U17420J	U17420E
RX850 Pro Ver.3.21 **リアルタイム**OS	**基礎編**	U18165J	U18165E
	内部構造編	U18164J	U18164E
	タスク・デバッガ編	U17422J	U17422E
RX850V4 Ver.4.22 **リアルタイム**OS	**機能編**	U16643J	U16643E
	内部構造編	U16644J	U16644E
	タスク・デバッガ編	U16811J	U16811E
RX-NET **ネットワーク・ライブラリ**（TCP/IP）Ver.1.30		U15083J	－－
RX-NET **ネットワーク・ライブラリ**（PPP）Ver.1.30		U15303J	－－
RX-NET **ネットワーク・ライブラリ**（DNS）Ver.1.30		U15304J	－－
RX-NET **ネットワーク・ライブラリ**（DHCP）Ver.1.30		U15382J	－－
RX-NET **ネットワーク・ライブラリ**（SMTP）		U15505J	－－
RX-NET **ネットワーク・ライブラリ**（POP）		U15539J	－－
RX-NET Ver.1.10 **ネットワーク・ライブラリ**（telnet）		U16085J	－－
RX-NET Ver.1.00 **ネットワーク・ライブラリ**（FTP）		U15946J	－－
RX-NET Ver.1.00 **ネットワーク・ライブラリ**（WebServer）		U16294J	－－
AZ850 Ver.3.30 **システム・パフォーマンス・アナライザ**		U17423J	U17423E
AZ850V4 Ver.4.10 **システム・パフォーマンス・アナライザ**		U17093J	U17093E
TW850 Ver.2.00 **性能解析チューニング・ツール**		U17241J	U17241E

〔メ　モ〕

目次

図の目次

図番号　タイトル,ページ

表の目次

表番号　タイトル,ページ

第 1 章　基本言語仕様

この章では，CA850 がサポートする基本言語仕様について説明します。

CA850 は，ANSI 規格で規定された言語仕様をサポートしていますが，その中には処理系定義として規定されている項目があります。ここでは，V850 マイクロコントローラ用マイクロプロセッサの処理系に依存した項目の言語仕様について説明します。

また，厳密な ANSI 準拠処理のオプションを指定した場合と指定しない場合の差分についても説明します。

なお，CA850 で独自に追加されている拡張言語仕様については，「第 3 章　拡張言語仕様」を参照してください。

1.1 処理系依存

この節では，ANSI 規格における処理系依存項目について説明します。

1.1.1 データ型とサイズ

- 1 バイト中のビット数は，8 ビットとします。
- オブジェクト中のバイト数，バイト順序，符号化は，次のように規定します。

表 1 － 1 データ型とサイズ

データ型	サイズ
char 型	1 バイト
short 型	2 バイト
int，long，float，double 型	4 バイト
ポインタ	unsigned int 型と同じ

ワード（4 バイト）中のバイト順序は，“ 下位から上位 ” です。また，符号付き整数は，2 の補数で表現します。最上位ビットには，符号（正，または 0 の場合 0，負の場合 1）が入ります。

1.1.2 翻訳段階

ANSI 規格では，翻訳における構文規則間の優先順位を 8 つの翻訳段階（翻訳フェーズ）に規定しています。3 段階目の “ 前処理字句と空白類文字の並びへの分割 ” で処理系定義となっている，“ 改行文字以外の空白類文字の空でない並び ” は 1 つに置き換えられずそのまま保持されます。

1.1.3 診断メッセージ

何らかの構文規則違反，および制約違反を含む翻訳単位に対して，ソース・ファイル名，行番号（特定可能な場合のみ）を含むエラー・メッセージを出力します。なお，エラー・メッセージの書式は “ 警告 ”，“ 致命的エラー ”, “ その他のエラー ” の 3 種類に区別されます。

1.1.4 フリー・スタンディング環境

(1) フリー・スタンディング環境[注]においては，プログラム開始処理時に呼び出される関数の名前，および型は特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(2) フリー・スタンディング環境におけるプログラム終了処理の効果は，特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

注 オペレーティング・システムの機能を使用せずに C 言語ソース・プログラムを実行する環境のこと。
ANSI 規格では，実行環境にはフリー・スタンディング環境とホスト環境の 2 つが規定されていますが，CA850 では，ホスト環境は現在提供されていません。

1.1.5 プログラムの実行

対話型装置の構成については，特に規定しません。

したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

1.1.6 文字集合

実行環境文字集合の要素の値は，ASCII コードです。

1.1.7 多バイト文字

多バイト文字は，文字定数ではサポートしていません。

ただし，コメントと文字列における日本語記述はサポートしています。

1.1.8 文字表示の意味

拡張表記の値は，次のように規定します。

表 1 － 2 拡張表記と意味

拡張表記	値（ASCII）	意味
\a	07	アラート（警告音）
\b	08	バックスペース
\f	0C	フォーム・フィード（改ページ）
\n	0A	ニュー・ライン（改行）
\r	0D	キャリッジ・リターン（復帰）
\t	09	水平タブ
\v	0B	垂直タブ

1.1.9 翻訳限界

次に，翻訳に際しての限界値を示します。

なお，* の付いている値は保証値であり，それ以上の値でも可能な場合もありますが，動作は保証されません。

表 1 － 3 翻訳限界

内容	限界値
複文，繰り返し制御構造，および選択制御構造の入れ子のレベル数 （ただし，“ case ” のラベル数に依存）	127
条件組み込みの入れ子のレベル数	255
1 つの宣言中の 1 つの算術型，構造体型，共用体型，または不完全型を修飾する（任意の組み合わせの）ポインタ，配列，および関数宣言子の数	16
完全な宣言子の中の，かっこで囲まれた宣言子の入れ子のレベル数	255*
完全な式の中の，かっこで囲まれた式の入れ子のレベル数	255*
マクロ名における有効先頭文字数	1023*
外部識別子における有効先頭文字数	1022
内部識別子における有効先頭文字数	1023
1 つの翻訳単位内の外部識別子，および 1 つの基本ブロック内で宣言されるブロック有効範囲をもつ識別子の数	4095*
1 つの翻訳単位内で同時に定義されるマクロ識別子の数[注]	2047
1 つの関数定義内の仮引数，および 1 つの関数呼び出し内の実引数の数	255
1 つのマクロ定義内の仮引数の数	127
1 つのマクロ呼び出し内の実引数の数	127
1 つの論理ソース行内の文字数	32768
連結後の 1 つの文字列定数，またはワイド文字列定数内の文字数	32768
インクルード・ファイルに対する入れ子のレベル数	50
1 つの “ switch ” 文に対する “ case ” ラベルの数 （ネストされている場合，それも含める）	1025
単一構造体，または単一共用体内のメンバ数	1023*
単一列挙型における列挙型定数の数	1023*
単一構造体宣言の並び内の，構造体，または共用体定義の入れ子のレベル数	63*

注 マクロ識別子の上限は，C コンパイラ・オプション（-Xm）で変更することができます。

1.1.10 数量的限界

(1) 汎整数型の限界値（limits.h ファイル）

汎整数型（char 型，符号付き／符号なし整数型，および列挙型）で表現できる値の各種限界値を limits.h ファイルに定義しています。

なお，多バイト文字はサポートしていないため，MB_LEN_MAX は該当する限界値を持ちません。そこで，MB_LEN_MAX には 1 として，定義のみ行っています。

また，CA850 の -Xchar=unsigned オプション（単なる char 型の符号を指定）が指定された場合，CHAR_MIN は 0，CHAR_MAX は UCHAR_MAX と同値となります。

次に，limits.h ファイルで定義されている各種限界値を示します。

表 1 － 4 汎整数型の各種限界値（limits.h ファイル）

名前	値	意味
CHAR_BIT	8	ビット・フィールドではない最小のオブジェクトのビット数（ = 1 バイト）
SCHAR_MIN	-128	signed char 型の最小値
SCHAR_MAX	+127	signed char 型の最大値
UCHAR_MAX	+255	unsigned char 型の最大値
CHAR_MIN	-128	char 型の最小値
CHAR_MAX	+127	char 型の最大値
SHRT_MIN	-32768	short int 型の最小値
SHRT_MAX	+32767	short int 型の最大値
USHRT_MAX	+65535	unsigned short int 型の最大値
INT_MIN	-2147483648	int 型の最小値
INT_MAX	+2147483647	int 型の最大値
UINT_MAX	+4294967295	unsigned int 型の最大値
LONG_MIN	-2147483648	long int 型の最小値
LONG_MAX	+2147483647	long int 型の最大値
ULONG_MAX	+4294967295	unsigned long int 型の最大値

(2) 浮動小数点型の各種限界値（float.h ファイル）

浮動小数点型の特性に関する各種限界値を float.h ファイルに定義しています。

次に，float.h ファイルで定義されている各種限界値を示します。

表 1 － 5 浮動小数点型の各種限界値の定義（float.h ファイル）

名前	値	意味
FLT_ROUNDS	+1	浮動小数点加算に対する丸めのモード V850 マイクロコントローラでは，1（もっとも近い方向へ丸める）とします。
FLT_RADIX	+2	指数表現の基数（b）
FLT_MANT_DIG	+24	浮動小数点仮数部における FLT_RADIX を底とする数字の数（p）
DBL_MANT_DIG	+24	
LDBL_MANT_DIG	+24	
FLT_DIG	+6	q 桁の 10 進数の浮動小数点数を基数 b の p 桁をもつ浮動小数点数に丸めることができ，再び変更なしに q 桁の 10 進数値に戻すことができるような 10 進数の桁数 注 1（q）
DBL_DIG	+6	
LDBL_DIG	+6	
FLT_MIN_EXP	-125	FLT_RADIX をその値から 1 引いた値でべき乗したとき，正規化された浮動小数点数となるような最小の負の整数 (e_{min})
DBL_MIN_EXP	-125	
LDBL_MIN_EXP	-125	
FLT_MIN_10_EXP	-37	10 をその値でべき乗したとき，正規化された浮動小数点数の範囲内になるような最小の負の整数 $\log_{10} b^{e_{min}-1}$
DBL_MIN_10_EXP	-37	
LDBL_MIN_10_EXP	-37	
FLT_MAX_EXP	+128	FLT_RADIX をその値から 1 引いた値でべき乗したとき，表現可能な有限浮動小数点数となるような最大の整数 (e_{max})
DBL_MAX_EXP	+128	
LDBL_MAX_EXP	+128	
FLT_MAX_10_EXP	+38	10 をその値でべき乗したとき表現可能な有限浮動小数点数の範囲内になるような最大の整数 $\log_{10}((1-b^{-p}) * b^{e_{max}})$
DBL_MAX_10_EXP	+38	
LDBL_MAX_10_EXP	+38	
FLT_MAX	3.40282347E + 38F	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX	3.40282347E + 38F	
LDBL_MAX	3.40282347E + 38F	
FLT_EPSILON	1.19209290E - 07F	指定された浮動小数点型で表現できる 1.0 と，1.0 より大きい最も小さい値との差異 注 2 b^{1-p}
DBL_EPSILON	1.19209290E - 07F	
LDBL_EPSILON	1.19209290E - 07F	
FLT_MIN	1.17549435E - 38F	正規化された正の浮動小数点数の最小値 $b^{e_{min}-1}$
DBL_MIN	1.17549435E - 38F	
LDBL_MIN	1.17549435E - 38F	

注 1 DBL_DIG，LDBL_DIG は，ANSI 規格では，10 以上となっていますが，V850 マイクロコントローラでは，double 型も long double 型も 32 ビットであるため 6 となります。

注 2 DBL_EPSILON と LDBL_EPSILON は，ANSI 規格では，1E - 9 以下となっていますが，V850 マイクロコントローラにおいては 1.19209290E - 07F となります。

1.1.11 識別子

外部名は，最大 1022 文字で一意に識別できるようにしてください。

なお，英字の大文字と小文字は区別されます。

1.1.12 char 型

型指定子（signed，unsigned）の付かない単なる char 型は，符号付き整数として扱います。

ただし，CA850 の -Xchar=unsigned オプションを指定することにより，符号なし整数として扱うこともできます。

ANSI 規格において要求されるソース・プログラムの文字集合に含まれないもの（エスケープ・シーケンス）は，char 型以外を char 型へ代入する場合と同様に，型変換して格納されます。

```
char c = '\777';     /* c の値は -1 となる */
```

1.1.13 浮動小数点定数

浮動小数点定数は，IEEE754 [注] に準拠しています。

注 IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。
また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

1.1.14 文字定数

(1) ソース・プログラムの文字集合と実行環境における文字集合は，基本的に両者とも ASCII コードで，同一の値をもつメンバと対応します。
ただし，ソース・プログラムにおける文字列には，日本語文字コードが使用できます（「1.1.15 文字列」を参照）。

(2) 2 つ以上の文字を含む整数文字定数の値は，最後の 1 文字が有効値となります。

(3) 基本的な実行環境文字集合で表現されない文字やエスケープ・シーケンスを含む場合，次のようになります。

(a) 8 進数エスケープ・シーケンス，および 16 進数エスケープ・シーケンスは，その 8 進数表記，および 16 進数表記で示される値となります。

\777	511

(b) 単純エスケープ・シーケンスは，次のようになります。

\'	'
\"	"
\?	?
\\	\

(c) \a，\b，\f，\n，\r，\t，\v については，「1.1.8 文字表示の意味」で示されている値と同値になります。

(4) 多バイト文字の文字定数はサポートしていません。

1.1.15 文字列

文字列中に日本語が記述できます。

デフォルトの文字コードは，シフト JIS となります。

入力ソース・ファイルの中の文字コードは，CA850 の -Xk オプション，または環境変数で選択できます。

オプション指定は環境変数よりも優先されます。ただし，n，または none を指定すると，文字コードは保証されません。

オプション指定

```
-Xk=[e | euc | n | none | s | sjis]
```

環境変数

```
LANG_V800 [e | euc | n | none | s | sjis]
```

なお，設定方法は，使用する環境の設定方法に従います（たとえば，csh の場合 setenv コマンドなど）。また，出力オブジェクト・ファイル中の文字コードは，CA850 の -Xkt オプションで変換できます。ただし，n，または none を指定すると，文字コードは変換されません。

オプション指定

```
-Xkt=[e | euc | n | none | s | sjis]
```

1.1.16 ヘッダ・ファイル名

ヘッダ・ファイル名の 2 つの形式（< >，" "）内の列を，ヘッダ・ファイル，または外部ソース・ファイル名に反映する方法は，「1.1.33 ヘッダ・ファイル取り込み」で規定します。

1.1.17 コメント

コメント中に日本語が記述できます。文字コード，「1.1.15 文字列」の場合と同じです。

1.1.18 符号付き定数と符号なし定数

汎整数型の値がよりサイズの小さい符号付き整数に変換される場合，上位ビットを切り捨てて，ビット列イメージをコピーします。

また，符号なし整数が，対応する符号付き整数に変換される場合，内部表現は変化しません。

1.1.19 浮動小数点と汎整数

汎整数型の値が浮動小数点型に型変換される際，型変換される値が，表現しうる値の範囲内にはあるが正確に表現することができない場合，その結果は，表現しうる最も近い値へ丸められます。

なお，結果がちょうど中央の値である場合には，偶数（仮数の最下位ビットが 0 のもの）に丸められます。

1.1.20 double 型と float 型

V850 マイクロコントローラ処理系では，double 型は float 型と同じ浮動小数点表現であり，32 ビット・データ（単精度）として扱われます。

1.1.21 ビット単位の演算子における符号付き型

ビット単位の演算子における符号付き型に対する特性は，シフト演算子については，「1.1.27 ビット単位のシフト演算子」の規定に準じます。

また，その他の演算子については，符号なしの値として（ビット・イメージのままで）計算するものとします。

1.1.22 構造体と共用体のメンバ

共用体のメンバの値がそれと異なるメンバに格納される場合，整列条件に従って格納されるため，その共用体のあるメンバへのアクセスは，整列条件に従って行われます（「2.1.6 構造体型」，および「2.1.7 共用体型」を参照）。

ただし，共通の先頭メンバの並びを共有している構造体だけをメンバとして含んでいる共用体の場合，内部表現は同じであるため，どの構造体の共通の先頭メンバを参照しても同じになります。

1.1.23 sizeof 演算子

“ sizeof ” 演算子の結果は，「1.1.1 データ型とサイズ」におけるオブジェクト中のバイトに関する規定に準じます。

なお，構造体と共用体については，パディング領域を含んだバイト数とします。

1.1.24 キャスト演算子

ポインタを汎整数型に変換する場合，要求される変数のサイズは，int 型と同じサイズです。変換結果は，ビット列がそのまま保存されます。

また，任意の整数はポインタに型変換できますが，int 型よりも小さい整数の場合，結果はその型に従って拡張されます。

1.1.25 乗除／剰余演算子

整数同士の除算で割り切れず，オペランドが負の値をもつ場合，“ / ”演算子の結果は，除数，または非除数のいずれか一方が負の場合は，代数的な商よりも大きい最小の整数となります。

ただし，どちらも負の場合は，代数的な商よりも小さい最大の整数となります。

また，オペランドが負の値をもつ場合，“ % ”演算子の結果の符号は第一オペランドの符号とします。

1.1.26 加減演算子

同一配列の要素を指す 2 つのポインタが減算される場合，結果の型は int 型とし，サイズは 4 バイトとします。

1.1.27 ビット単位のシフト演算子

“ E1 >> E2 ”において，E1 が符号付きの型で負の値をもつ場合，算術シフトを行います。

1.1.28 記憶域クラス指定子

記憶域クラス指定子“ register ”の宣言は，可能なかぎり高速にアクセスするために行いますが，必ずしも有効であるとはかぎりません[注]。

注 割り当てられるレジスタについては「7.5.1 register 指定子」を参照してください。

1.1.29 構造体と共用体指定子

(1) signed，unsigned の付かない単なる int 型ビット・フィールドは，符号付きとして扱い，最上位ビットは符号ビットとして扱います。ただし，CA850 の -Xbitfield オプション（単なる int 型ビット・フィールドの符号を指定）を指定することにより，符号なしとして扱うこともできます。

(2) ビット・フィールドを保持するために，十分な大きさの任意のアドレス付け可能な記憶域単位を割り付けることができますが，十分な領域がなかった場合，合わなかったビット・フィールドはフィールドの型の整列条件に合わせて次の単位に詰め込まれます。

(3) 単位内のビット・フィールドの割り付け順序は下位から上位です。

(4) 1 つの構造体，または共用体の非ビット・フィールドの各メンバは，次のように境界整列されます。

char，unsignd char 型，およびその配列	バイト境界
short，unsigned short 型，およびその配列	ハーフワード境界
その他（ポインタを含む）	ワード境界

1.1.30 列挙型指定子

列挙型の型は，signed int 型とします。

ただし，-Xenum_type=*string* オプション指定時は，次のようになります。

char	char として扱う
uchar	unsigned char として扱う
short	short として扱う
ushort	unsigned short として扱う

1.1.31 型修飾子

“ volatile ” 修飾された型をもつデータへのアクセスは，データがマッピングされているアドレス（I/O ポートなど）に依存します。

1.1.32 条件組み込み

(1) 条件組み込みで指定される文字定数に対する値と，その他の式中に現れる文字定数の値とは等しくなります。

(2) 単一文字の文字定数は，負の値を持たないようにしてください。

1.1.33 ヘッダ・ファイル取り込み

(1) “ #include < 文字列 > ” という形式の前処理指示

“ #include < 文字列 > ” という形式の前処理指示は，“ 文字列 ” が “ \ ” で始まらない場合[注]，指定されたフォルダ（-I オプション）からヘッダ・ファイルを検索し，次に ca850 が置かれた bin フォルダからの相対パスでの ..\inc850 フォルダを検索します。

なお，“ < ” と “ > ” の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “ \ ” と “ / ” の両者がフォルダの区切りとしてみなされます。

例

```
#include <header.h>
```

検索順は，次のとおりです。

- -I で指定したフォルダ
- 標準のフォルダ

(2) “ #include " 文字列 " ” という形式の前処理指示

“ #include " 文字列 " ” という形式の前処理指示は，“ 文字列 ” が “ \ ” で始まらない場合，ソース・ファイルがあるフォルダからヘッダ・ファイルを検索し，次に，指定したフォルダ（-I オプション），最後に ca850 が置かれた bin フォルダからの相対パスでの ..\inc850 フォルダを検索します。

なお，“ " ”“ " ” の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

例

```
#include "header.h"
```

検索順は，次のとおりです。

- ソース・ファイルがあるフォルダ
- -I で指定したフォルダ
- 標準のフォルダ

(3) “ #include 前処理字句列 ” という形式

“ #include 前処理字句列 ” という形式において，前処理字句列が単一で < 文字列 >，または " 文字列 " の形式に置換されるマクロである場合にのみ，単一のヘッダ・ファイル名の前処理字句として扱われます。

(4) 区切られた列とヘッダ・ファイル名との間

（最終的に）区切られた列とヘッダ・ファイル名との間においては，列中の英文字の長さを判別し，

```
コンパイラ動作環境において有効なファイル名長までが有効
```

となります。ファイルを探すフォルダについては，上記の規定に準じます。

1.1.34 #pragma 指令

CA850 では，次の #pragma 指令が指定できます。

(1) アセンブラ命令の記述

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

C 言語中に，アセンブラ命令を記述することができます。

なお，記述方法についての詳細は「3.4 アセンブラ命令の記述」を参照してください。

(2) インライン展開指定

```
#pragma inline 関数名 [, 関数名…]
```

インライン展開する関数を指定することができます。

なお，インライン展開についての詳細は「3.8 インライン展開」を参照してください。

(3) デバイス種別指定

```
#pragma cpu デバイス名
```

使用するデバイスの機種依存情報を定義したデバイス・ファイルを参照するように指定します。CA850 のデバイス指定オプション（-cpu）と同じ機能です。C 言語ソース内にデバイスを定義したい場合に用います。

(4) データ／プログラムのメモリ割り当て

```
#pragma section セクション種別 [" セクション名 "] [begin|end]

#pragma text [" セクション名 "] [ 関数名 ]
```

(a) section

変数を任意のセクションに割り当てます。

なお，配置方法についての詳細は「3.1 データのセクション割り当て」を参照してください。

(b) text

任意の名称のテキスト・セクションに関数を指定できます。

なお，配置指定についての詳細は「3.2 関数のセクション割り当て」を参照してください。

(5) 周辺 I/O レジスタ名有効化指定

```
#pragma ioreg
```

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。周辺 I/O レジスタ名をそのまま用いてプログラミングする場合はこの #pragma 指令を指定する必要があります。

(6) 割り込み／例外ハンドラ指定

```
#pragma interrupt 割り込み要求名 関数名 [ 配置方法 ]
```

割り込み／例外処理ハンドラを C 言語で記述します。

なお，記述方法については「3.7.3 割り込み／例外ハンドラの記述方法」を参照してください。

(7) 割り込み禁止関数指定

```
#pragma block_interrupt 関数名
```

関数全体を割り込み禁止にします。

(8) タスク指定

```
#pragma rtos_task [ 関数名 ]
```

リアルタイム OS 上で動作するタスクを C 言語で記述します。

なお，記述方法についての詳細は「3.9.1 タスクの記述」を参照してください。

(9) 構造体パッキング指定

```
#pragma pack([1248])
```

構造体パッキングを指定します。数値はパッキング値，すなわちメンバのアライメント値を指定します。数値には 1，2，4，8 が指定できます。数値を指定しない場合，デフォルト（8）[注]となります。

注　本バージョンではアライメント値“4”と“8”は同じになります。

1.1.35 あらかじめ定義されたマクロ名

次に，サポートしているマクロ名を示します。

なお，“ __ ” で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に “ __ ” のある形式のマクロを利用するようにしてください。

表 1 － 6 サポートしているマクロ一覧

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）。
__FILE__	仮定されたソース・ファイルの名前（文字列定数）。
__DATE__	ソース・ファイルの翻訳日付（“ Mmm dd yyyy ” の形式をもつ文字列定数。ここで，月の名前は ANSI 規格で規定されている asctime 関数で生成されるもの（英字 3 文字の並びで最初の 1 文字のみ大文字）と同じもの。dd の最初の文字は値が 10 より小さい場合空白とします）。
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような “ hh : mm : ss ” の型式をもつ文字列定数）。
__STDC__	10 進定数 1（-ansi オプション指定時に定義）[注]。
__v800 __v800__	10 進定数 1。
__v850 __v850__	10 進定数 1。
__v850e __v850e__	10 進定数 1（CA850 で，ターゲット・デバイスに V850Ex を指定した場合に定義）。
__v850e2 __v850e2__	10 進定数 1（CA850 で，ターゲット・デバイスに V850E2/xxx を指定した場合に定義）。
__CA850 __CA850__	10 進定数 1。
__CHAR_SIGNED__	10 進定数 1（-Xchar オプションで，符号つきを指定した場合，および -Xchar オプションを指定しない場合に定義）。
__CHAR_UNSIGNED__	10 進定数 1（-Xchar オプションで，符号なしを指定した場合に定義）。
__DOUBLE_IS_32BITS__	10 進定数 1。
_DOUBLE_IS_32BITS	10 進定数 1。
CPU マクロ	ターゲット CPU を示すマクロで 10 進定数 1。デバイス・ファイル中の「品種指定名」で示される文字列の先頭と末尾に “ __ ” を付けたものが定義されます。
レジスタ・モード・マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 レジスタ・モードと定義されるマクロは次のとおりです。 32 レジスタ・モード ：__reg32__ 26 レジスタ・モード ：__reg26__ 22 レジスタ・モード ：__reg22__

注 -ansi オプション指定時の処理については，「1.2 -ansi オプション」を参照してください。

1.1.36 特別なデータ型の定義

次に，stddef.h ファイルにおける NULL，size_t，ptrdiff_t の定義を示します。

表 1 － 7 NULL，size_t，ptrdiff_t の定義（stddef.h ファイル）

NULL ／ size_t ／ ptrdiff_t	定義
NULL	`((void *)0)`
size_t	`unsigned int`
ptrdiff_t	`int`

1.2 -ansi オプション

CA850 で -ansi オプションを指定した場合，ANSI 規格に厳密な処理が行われます。

次に，-ansi オプションを指定した場合と，指定しない場合の処理の違いを示します。

表 1 － 8 言語仕様に厳密な -ansi オプション指定時の処理

項目	-ansi 指定あり	-ansi 指定なし
トライグラフ系列	トライグラフ系列の置換を行います。	置換しません。
ビット・フィールド	ビット・フィールドに int 型以外の型を指定した場合，エラー[注 1] とします。	警告メッセージを出力し，許可します。
引数のスコープ	関数の引数と同名の自動変数を宣言した場合，二重定義のエラーとします。	警告メッセージを出力し，自動変数を有効とします。
ポインタの代入 1	汎整数型[注 2] 変数へのポインタ型の数値の代入は，エラーとします。	警告メッセージを出力し，キャストして代入します。
ポインタの代入 2	異なる型を指すポインタ同士の代入は，エラーとします。	警告メッセージを出力し，許可します。
型変換	左辺値でない配列のポインタへの変換は，エラーとします。	警告メッセージを出力し，許可します。
比較演算子	算術型変数とポインタの比較は，エラーとします。	警告メッセージを出力し，許可します。
条件演算子	第二式と第三式がともに汎整数型，同じ構造体，同じ共用体，または代入先と同じ型へのポインタ型の数値のいずれでもない場合，エラーとします。	警告メッセージを出力し，キャストして代入します。
# 行番号	エラーとします。	“ #line ” 行番号と同様に扱います[注 3]。
行の途中の # 文字	‘ # ’ 文字が行の途中で現れた場合，エラーとします。	警告メッセージを出力し，許可します。
_asm	警告メッセージを出力し，関数呼び出しとして扱います。 ただし，__asm は有効とします。	アセンブラ挿入[注 4] として扱います。
__STDC__	値が 1 のマクロとして定義します。	定義しません。
2 進定数	“ 0b ”，または “ 0B ” と，その後ろに続く 1 個以上の “ 0 ”，または “ 1 ” の数字の並びをエラーとします。	“ 0b ”，または “ 0B ” と，その後ろに続く 1 個以上の “ 0 ”，または “ 1 ” の数字の並びを 2 進定数とします。

注 1 “ E ” で始まる通常のエラー。以下同じです。

注 2 char 型，符号付き／符号なし整数型，および列挙型です。

注 3 ANSI 規格を参照してください。

注 4 「3.4 アセンブラ命令の記述」を参照してください。

第 2 章　コンパイル実行時の環境

この章では，CA850 におけるデータやレジスタ，デバイスの指定等，実行時の環境の取り扱い方法について説明します。

2.1　データの内部表現と領域

この節では，CA850 が扱うデータのそれぞれの型における，内部表現と値域について説明します。

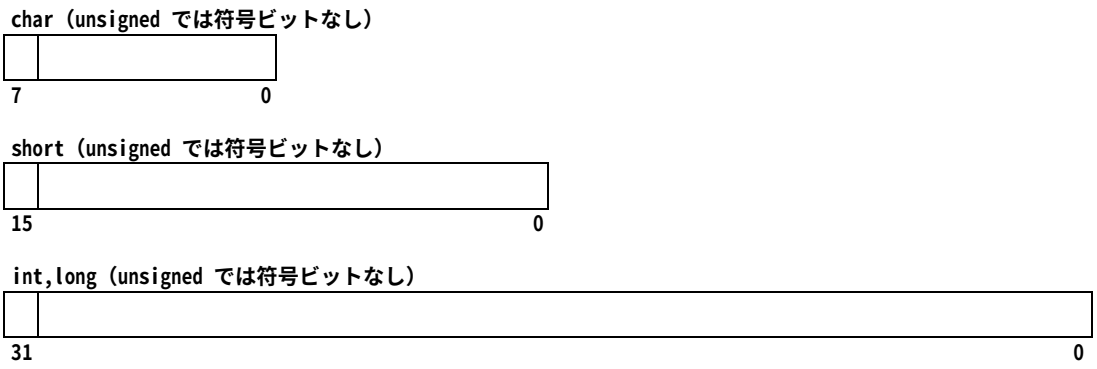
2.1.1　整数型

(1) 内部表現

領域の左端ビットは，符号付きの型（“ unsigned ”を伴わずに宣言された型）では，符号ビットとなります。符号付きの型において，値は 2 の補数表現で表されます。

ただし，-Xchar=unsigned が指定された場合，“ signed ”も“ unsigned ”も伴わずに宣言された char 型は，符号なし（unsigned）となります。

図 2 － 1 整数型の内部表現



(2) 値域

表 2 － 1 整数型の値域

型	値域
char 注	-128 ～ +127
short	-32768 ～ +32767
int	-2147483648 ～ +2147483647
long	-2147483648 ～ +2147483647
unsigned char	0 ～ 255
unsigned short	0 ～ 65535
unsigned int	0 ～ 4294967295
unsigned long	0 ～ 4294967295

注 CA850 で “ -Xchar=unsigned ” が指定された場合，0 ～ 255 の値域です。

注意 CA850 は 64 ビット長の演算はできません。

2.1.2 浮動小数点型

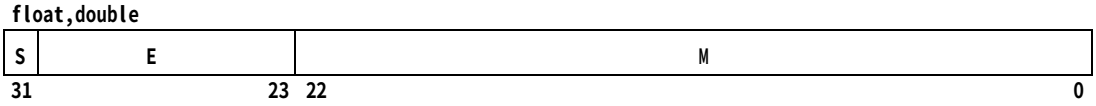
(1) 内部表現

浮動小数点型データの内部表現は，IEEE754 注に準拠しています。領域の左端のビットは，符号ビットとなります。この符号ビットの値が 0 であれば正の値に，1 であれば負の値になります。

また，double 型は float 型と同じ浮動小数点表現であり，32 ビット・データ（単精度）として扱われます。

注 IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。
また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

図 2 － 2 浮動小数点型の内部表現



S：仮数部の符号ビット

E：指数部（8 ビット）

M：仮数部（23 ビット）

(2) 値域

表 2 － 2 浮動小数点型の値域

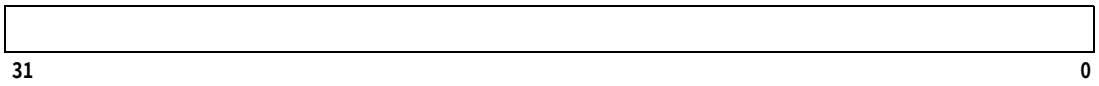
型	絶対値の値域
float，double	$1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$

2.1.3 ポインタ型

(1) 内部表現

ポインタ型の内部表現は，unsigned int 型の内部表現と同じです。

図 2 － 3 ポインタ型の内部表現

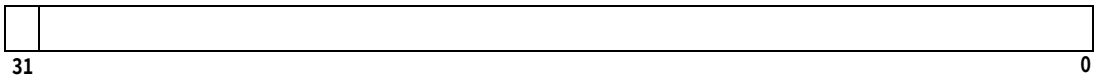


2.1.4 列挙型

(1) 内部表現

列挙型の内部表現は，singed int 型の内部表現と同じです。領域の左端のビットは，符号ビットとなります。

図 2 － 4 列挙型の内部表現



2.1.5 配列型

(1) 内部表現

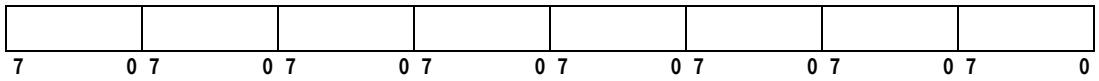
配列型の内部表現は，配列の要素を，その要素の整列条件（alignment）を満たす形で並べたものとなります。

例

```
char a[8] = {1, 2, 3, 4, 5, 6, 7, 8}
```

上記の例に示した配列に対する内部表現は，次のようになります。

図 2 － 5 配列型の内部表現



2.1.6 構造体型

(1) 内部表現

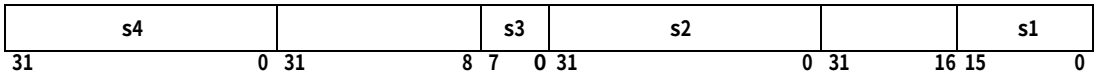
構造体型の内部表現は，構造体の要素をその要素の整列条件を満たす形で並べたものとなります。

例

```
struct {
        short   s1;
        int     s2;
        char    s3;
        long    s4;
} tag;
```

この例に示した構造体に対する内部表現は，次のようになります。

図２－６ 構造体型の内部表現



なお，構造体パッキング機能利用時の内部表現は，「3.11 構造体パッキング」を参照してください。

2.1.7 共用体型

(1) 内部表現

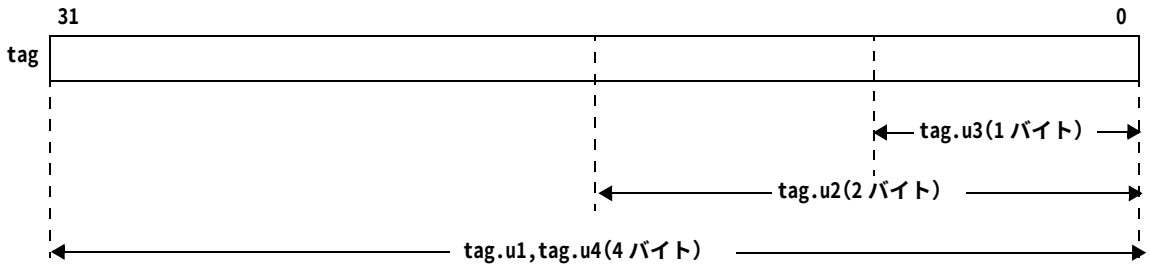
共用体はそのメンバがすべてオフセット 0 から始まり，そのメンバの任意のものを収容するのに十分なサイズを持つ構造体と考えられます。つまり，共用体型の内部表現は，同じアドレスに共用体の要素それぞれが単体で置かれているのと同様です。

例

```
union {
        int     u1;
        short   u2;
        char    u3;
        long    u4;
} tag;
```

この例に示した共用体に対する内部表現は，次のようになります。

図２－７ 共用体型の内部表現



2.1.8 ビット・フィールド

(1) 内部表現

ビット・フィールドに対しては，宣言された数のビットを含む領域が取られます。符号付きの型として宣言されたビット・フィールドに対しては，最上位ビットは符号ビットとなります。

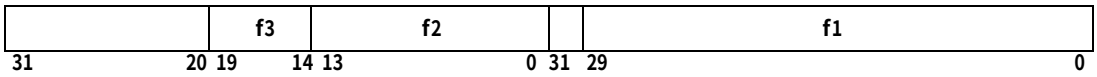
最初に宣言されたビット・フィールドは，ワード領域の最下位ビットから割り当てられます。ビット・フィールドに対し，その前のビット・フィールドに続けて領域を割り当てると，その領域がそのビット・フィールドの宣言において指定された型の整列条件を満たす境界を越えてしまう場合，そのビット・フィールドに対する領域はその整列条件を満たしている境界から割り当てられます。

例

```
struct {
       unsigned int  f1 : 30;
       int           f2 : 14;
       unsigned int  f3 :  6;
} flag;
```

この例に示したビット・フィールドに対する内部表現は，次のようになります。

図 2 － 8 ビット・フィールドの内部表現



なお，ANSI 規格ではビット・フィールドに char 型，および short 型は指定できませんが，CA850 では，ビット・フィールドに char 型，および short 型を指定することができます。

ただし，この場合，警告メッセージが出力され，指定した型の整列条件でパディングされます[注]。

なお，構造体パッキング機能利用時のビットフィールドの内部表現は，「3.11 構造体パッキング」を参照してください。

注 CA850 のオプションで，-ansi を指定した場合は，エラーとなります。

2.1.9 整列条件

(1) 基本型に対する整列条件

次に，基本型に対する整列条件を示します。

ただし，CA850 の -Xi を指定した場合，配列型はすべてワード境界となります。

表 2 － 3 基本型に対する整列条件

基本型	整列条件
（unsigned）char とその配列型	バイト境界
（unsigned）short とその配列型	ハーフワード境界
（ポインタを含む）その他の基本型	ワード境界

(2) 共用体型に対する整列条件

共用体型に対する整列条件は，最大メンバ・サイズにより，次のようになります。

表 2 － 4 共用体型に対する整列条件

最大メンバ・サイズ	整列条件
2 バイト＜サイズ	ワード境界
サイズ≦ 2 バイト	最大メンバ・サイズ境界

それぞれの場合における例を示します。

例 1

```
union tug1 {
       unsigned short i;    /* 2 バイト・メンバ */
       unsigned char  c;    /* 1 バイト・メンバ */
};                          /* 共用体は 2 バイトで整列 */
```

例 2

```
union tug2 {
       unsigned int  i;/* 4 バイト・メンバ */
       unsigned char c;     /* 1 バイト・メンバ */
};                          /* 共用体は 4 バイトで整列 */
```

(3) 構造体型に対する整列条件

構造体型に対する整列条件は，構造体のサイズ（整列部分を含めない）により，次表のようになります。
ただし，CA850 の -Xi を指定した場合，構造体型はすべてワード境界となります。

表 2 － 5 構造体型に対する整列条件

構造体サイズ	整列条件
2 バイト＜サイズ	ワード境界
サイズ≦ 2 バイト	サイズとメンバの型により，次のいずれかになります。 • int 型以上の型のメンバが存在する場合 → ワード境界 • int 型以上の型メンバがなく，1 バイト＜サイズ≦ 2 バイトの場合 → ハーフワード境界 • int 型以上の型メンバがなく，サイズ≦ 1 バイトの場合 → バイト境界

それぞれの場合における例を示します。

例 1

```
struct SS {
        int     i;              /* 4 バイト・メンバ */
        char    c;              /* 1 バイト・メンバ */
};                              /* 構造体は 4 バイトで整列 */
```

例 2

```
struct BIT_I {
        int     i1 : 5;         /* 4 バイト・メンバ（サイズは 1 バイト以下）*/
};                              /* メンバの型が int のため，構造体は 4 バイトで整列 */
```

例 3

```
struct BIT_C {
        char    c1 : 5;         /* 1 バイト・メンバ */
};                              /* 構造体は 1 バイトで整列 */
```

例 4

```
struct BIT_CC {
        char    c1 : 5;         /* 1 バイト・メンバ */
        char    c2 : 5;         /* 1 バイト・メンバ */
};                              /* サイズが 2 バイトのため，構造体は 2 バイトで整列 */
```

(4) 関数引数に対する整列条件

関数引数に対する整列条件は，ワード境界となります。

(5) 実行プログラムに対する整列条件

リロケータブルなオブジェクト・ファイルをリンクして実行可能なオブジェクト・ファイルを生成する際の整列条件は，ハーフワード境界となります。

2.2 汎用レジスタ

表 2 － 6 に，CA850 における汎用レジスタの用い方を示します。

なお，汎用レジスタには，次の機能があります。

(1) ソフトウエア・レジスタ・バンク

作業用レジスタ（r10 - r19），およびレジスタ変数用レジスタ（r20 - r29）は，CA850 の -reg オプションにより使用本数を抑制できます（「2.4 ソフトウエア・レジスタ・バンク」を参照）。

(2) マスク・レジスタ機能

32 レジスタ・モード，および 22 レジスタ・モードの場合，r20，および r21 のレジスタは，マスク値の設定のために用いることができます（「2.5 マスク・レジスタ」を参照）。

表 2 － 6 汎用レジスタの用い方

レジスタ		使用方法
r0	ゼロ・レジスタ	0 の値として演算時に使用 また，const セクション（ROM 領域に置く読み出し専用セクション）[注]などに配置されたデータの参照にも使用
r1	アセンブラ予約レジスタ	アセンブラにおける命令展開時に使用
r2（hp）	ハンドラ・スタック・ポインタ	システム予約
r3（sp）	スタック・ポインタ	スタック・フレームの先頭を指すものとして使用
r4（gp）	グローバル・ポインタ	外部変数の参照時に使用
r5（tp）	テキスト・ポインタ	テキスト・セクション（.text セクション）の先頭を指すものとして使用
r6 - r9	引数用レジスタ	引数の受け渡しに使用
r10 - r19	作業用レジスタ	演算時のワーク・レジスタとして使用（r10 は関数の戻り値の受け渡しにも使用）
r20 - r29	レジスタ変数用レジスタ	レジスタ変数用の領域として使用
r30（ep）	エレメント・ポインタ	内蔵 RAM や外部 RAM のセクション[注]に配置指定された外部変数の参照に使用
r31（lp）	リンク・ポインタ	関数の戻り先アドレスの受け渡しに使用

注 データのセクションへの配置については，「3.1 データのセクション割り当て」を参照してください。

2.3 データの参照方法

次に，CA850 におけるデータの参照方法を示します。

表 2 － 7 データの参照方法

種類	参照方法
数値定数	イミーディエト
文字列定数	グローバル・ポインタ（gp）からのオフセット r0 レジスタからのオフセット
自動変数，引数	スタック・ポインタ（sp）からのオフセット
外部変数，関数内静的変数	グローバル・ポインタ（gp）からのオフセット エレメント・ポインタ（ep）からのオフセット r0 レジスタからのオフセット
関数のアドレス	テキスト・ポインタ（tp）からのオフセットを用いて実行時に演算

2.4 ソフトウエア・レジスタ・バンク

CA850 では，ソフトウエアによるレジスタ・バンク機能を実現するため，3 つのレジスタ・モードが提供されています。レジスタ・モードを効率的に指定することにより，割り込み処理時やタスク切り替え時に，一部のレジスタの退避，復帰処理が不要となり，処理速度が高められます。レジスタ・モードの指定は，CA850 のレジスタ・モード指定オプション（-reg）によって行います。この機能は，CA850 が内部で使用するレジスタの本数を段階的に抑制し，次の効果が期待できます。

(1) 余ったレジスタをアプリケーション・プログラム（アセンブリ言語ソース・プログラム）で自由に使うことができる。

(2) 退避，復帰で生じるオーバーヘッドが減少する。

注 CA850 によるレジスタ割り付けの対象となる変数の多いアプリケーション・プログラムでは，レジスタ・モードの指定によって，それまでレジスタに割り付けられていた変数がメモリ・アクセスとなり，その分，処理速度が低下することがあります。

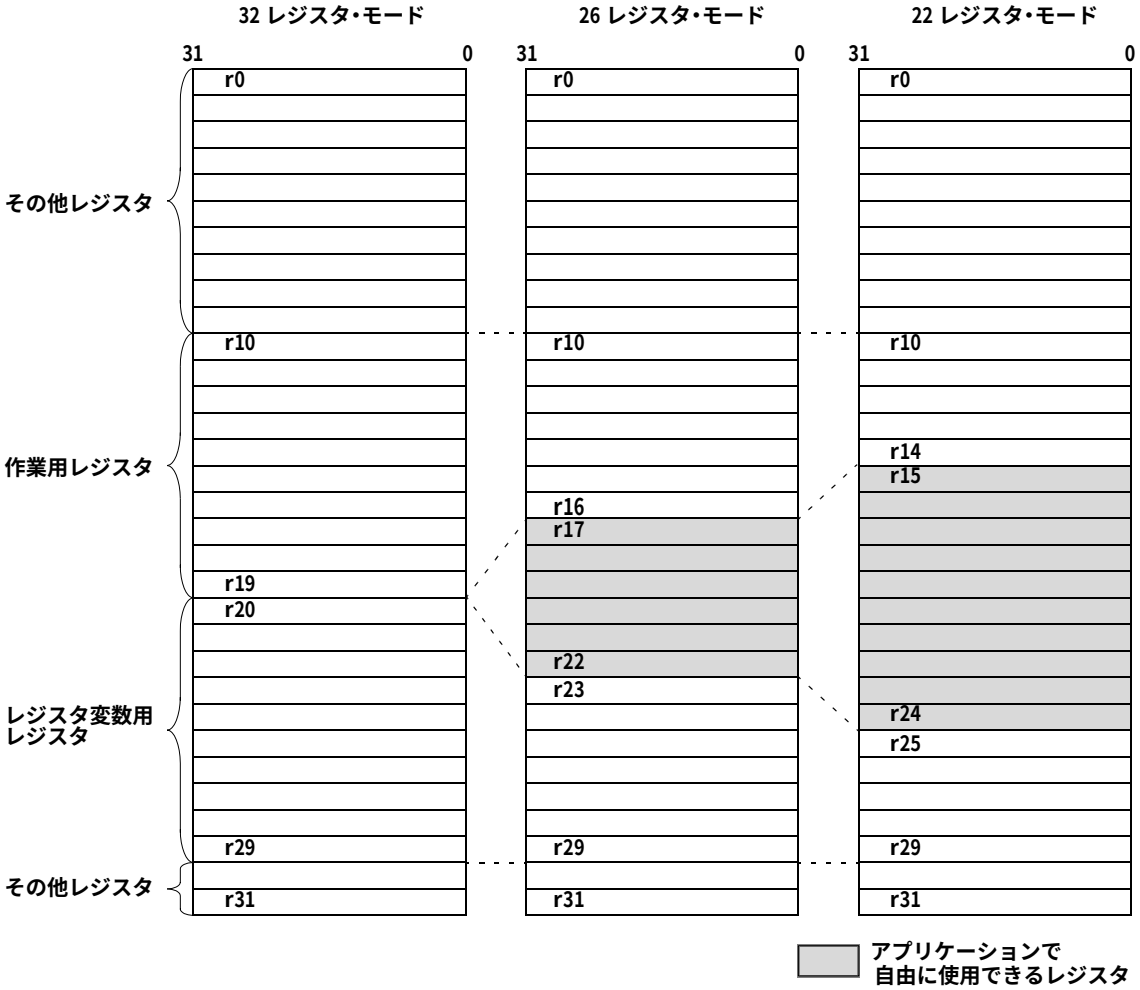
2.4.1 レジスタ・モード

次表，および次図に，CA850 のレジスタ・モードとして提供されている 3 つのモードを示します。

表 2 － 8 CA850 が提供するレジスタ・モード

レジスタ・モード	作業用レジスタ	レジスタ変数用レジスタ
32 レジスタ・モード（デフォルト）	r10-r19	r20-r29
26 レジスタ・モード	r10-r16	r23-r29
22 レジスタ・モード	r10-r14	r25-r29

図 2 － 9 レジスタ・モードと使用可能レジスタ



コマンド・ラインにおける指定例

```
> ca850 -cpu 3201 -reg26 file.c   ← 26 レジスタ・モードでコンパイル
```

2.4.2 レジスタ・モードとライブラリ

CA850 が提供するライブラリ（「第 6 章 ライブラリ関数」を参照）は，各レジスタ・モードごとに用意されています。ライブラリを検索する標準フォルダは，デフォルトでは，“ *Install Folder*\lib850 ”，および “ *Install Folder*\lib850\r32 ”ですが，CA850 で 22，または 26 レジスタ・モードを指定した場合，“ *Install Folder*\lib850\r32 ” の代わりに，“ *Install Folder*\lib850\r22 ”，または “ *Install Folder*\lib850\r26 ” が，ライブラリに対する標準フォルダとなります。

また，CA850 から ld850 を起動するのではなく，コマンド・ラインで ld850 を直接起動してオブジェクト・ファイルをリンクする場合，ld850 の -reg オプションを指定すると，各レジスタ・モードに合ったライブラリを参照します。

2.5 マスク・レジスタ

V850 マイクロコントローラでは，バイト・データ，ハーフワード・データをメモリからレジスタにロードする場合，最上位ビットの値によりワード長へ符号拡張します。このため，unsigned char，unsigned short 型データの演算では，上位ビットのマスク・コードが生成される場合があります（「7.8 データ型」参照）。

また，演算結果をレジスタ変数へ格納する場合，符号なしバイト・データ，符号なしハーフワード・データでは，上位ビットをクリアするためにマスク・コードが生成されます。どちらの場合も，ワード・データに切り替えれば回避できますが，ワード・データにできず，マスク・コードが生成される場合，マスク・レジスタ機能を用いることにより，コード・サイズの削減ができます。

ただし，マスク・レジスタ機能を利用するか否かの判断には，利用する側で次の点を十分考慮する必要があります。

(1) マスク・コードが多く出力されるプログラムであるか。
(2) マスク・レジスタとして使用するため，レジスタ変数用レジスタが 2 本少なくなるが，その影響はないか。

次の例のように，マスク・レジスタ機能では，r20，および r21 をマスク・レジスタとして CA850 が使用します。なお，マスク・レジスタへのマスク値の設定は，プログラムで行う必要があります。

マスク・コード生成例

```
unsigned char UC;
unsigned short US;
void f(void)
{
        register unsigned char ruc;
        register unsigned short rus;
                :
        UC *= UC;
                :
        ruc = UC;
        rus = US;
                :
}
```

（通常の出力コード）

```
        ld.b    $UC, r11
        andi    0xff, r11, r11
        mulh    r11, r11
        st.b    r11, $UC
                :
        ld.b    $UC, r29
        andi    0xff, r29, r29
        ld.h    $US, r28
        andi    0xffff, r28, r28
```

（マスク・レジスタ利用時の出力コード）

```
        ld.b    $UC, r11
        and     r20, r11
        mulh    r11, r11
        st.b    r11, $UC
                :
        ld.b    $UC, r29
        and     r20, r29
        ld.h    $US, r28
        and     r21, r28
```

V850Ex を使用する場合，“ 符号なしデータの演算 ”を行う命令が追加されており，CA850 もこの命令を使用するコードを出力します。そのため V850Ex を使用する場合は，マスク・レジスタを使用する設定にしても，さほど効果があがらない場合があります。

2.5.1 マスク値の設定

マスク・レジスタとなる r20，および r21 には，プログラムでマスク値（0xff, 0xffff）を設定する必要があります。CA850 は，マスク値が設定されているものと想定して，マスク用レジスタを使用したマスク・コードを生成します。

マスク値の設定例

```
__start :
        mov     #__tp_TEXT, tp
        mov     #__gp_DATA, gp
                :
        mov     0xff, r20               --r20 にマスク値を設定
        mov     0xffff, r21             --r21 にマスク値を設定
                :
        jarl    _main, lp
```

ただし,リアルタイムOSを使用したプログラムの場合,リアルタイムOSの種類によって次のようになります。

(1) RX850 使用する場合

RX850 の初期化ルーチン内で，自動的にマスク値が設定されるため，プログラムで 設定する必要はありません。

(2) RX850 Pro を使用する場合

スタート・アップ・モジュールなどで，あらかじめマスク値を設定する必要があります。

(3) リアルタイム OS を使用しない場合

スタート・アップ・モジュールなどで，あらかじめマスク値を設定する必要があります[注]。

注 パッケージ添付のスタート・アップ・モジュール例“ crtN. s ”（32 レジスタ・モード用）では，マスク値の設定を行っています（「第 5 章 スタート・アップ・ルーチン」参照）。

2.5.2 マスク・レジスタ機能の使用方法，および注意事項

マスク・レジスタ機能を使用するための指定，および注意事項は次のとおりです。

(1) C 言語ソース・ファイルを新たにコンパイルする場合

CA850 のマスク・レジスタ機能用オプション（-Xmask_reg）を指定することにより，マスク・レジスタを使用したマスク・コードと，マスク・レジスタ機能を使用していることを示す情報（“ .option mask_reg ” 疑似命令）を含んだアセンブラ命令が出力されます。

(2) リンク時のチェック

CA850 のマスク・レジスタ機能用オプション（-Xmask_reg）を指定して，リンカまで一度に起動した場合，.c ファイルから作成されたことを示すファイル名情報（“ .file ” 疑似命令で指定された情報）を持つオブジェクト・ファイルに対して，リンク時にチェックを行います。このとき，マスク・レジスタ機能を使用しているオブジェクトと使用していないオブジェクトが混在していた場合，エラーとなります。

注意 1 アーカイブ・ファイル（a. ファイル）に含まれるオブジェクトはチェックしません。独自に作成した .a ファイルを使用する場合，マスク・レジスタを使用していないことを確認してください。

注意 2 コマンド・ラインから，ld850 を単独で起動する場合，リンク時チェック用オプション（-mc）を指定する必要があります。

(3) 作成したアセンブリ言語ソース・ファイルの場合

はじめからアセンブラ命令で記述したプログラムの場合，マスク・レジスタを破壊していないかチェックしてください。ファイル名情報が “ .c ” ではないため，リンク時にチェックされません。なお，アセンブラにマスク・レジスタの使用オプション（-m）を指定すれば，アセンブル時の警告で確認できます。

(4) 提供ライブラリの制限

アーカイブ・ファイル中のオブジェクト・ファイルはリンク時チェックされませんが，パッケージ提供のライブラリはマスク・レジスタを破壊することはありません。[注]

注 標準ライブラリ中の bsearch 関数では，アプリケーション関数を呼び出すため，マスク・レジスタを破壊する可能性があります。マスク・レジスタ機能使用時には，bsearch 関数は使用しないでください（使用しても CA850 ではエラーにはなりません）。

2.6 デバイス・ファイル

デバイス・ファイルとは，ターゲット・デバイスの各品種ごと，または各グループごとに 1 つずつ，パッケージとして用意された，機種依存情報を持つバイナリ・ファイルです。コンパイラでは，アプリケーション・システムで使用するターゲット・システムに対応したオブジェクト・コードを生成するために，デバイス・ファイルを参照します。このため，使用するデバイス・ファイルは，デバイス・ファイルの標準フォルダに置くか，デバイス・ファイルの置かれているフォルダをコンパイラのオプションで指定するようにしてください。デバイス・ファイルが見つからないと，コンパイル時にエラーとなります。

2.6.1 デバイス・ファイルの指定方法

C 言語のプログラムで参照するデバイス・ファイルの指定方法には，次の二通りがあります。

(1) コンパイラのオプション（-cpu デバイス名）によるデバイス名指定

例

```
> ca850 -cpu 3201 file.c
```

PM+ にてビルドする場合，デバイス指定したことでこのオプションと同等になります。

(2) C 言語ソース・ファイルにおける #pragma 指令（#pragma cpu デバイス名）によるデバイス名指定

例

```
#pragma cpu 3201
```

上記の例では，“ 3201 ”がデバイス名（V850ES/SA2）です。“ デバイス名 ”として指定できる文字列は，オプション指定，#pragma 指令ともに共通です。また，大文字，小文字は区別しません。

なお，デバイス名として指定できる文字列については，各デバイスの “ 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。

注意 1 #pragma 指令でデバイス名を指定する場合，すべてのソース・ファイルにデバイス指定を記述する必要があります。

注意 2 #pragma 指令によるデバイス名の指定は，ソース・ファイルの先頭に記述してください。デバイス名指定の前に記述できる処理は，C 言語の構文に関係のない前処理とコメントに限られます。デバイス名指定を C 言語の構文中に記述した場合，コンパイラは次のエラー・メッセージを出力し，処理を中止します。

```
F2625: illegal placement ' #pragma cpu '
```

誤った指定の例

```
#include <stdio.h>
int    i;
#pragma cpu 3201
       :
```

2.6.2 デバイス・ファイル指定時の注意

(1) デバイス名を指定しない場合

#pragma 指令による指定，または -cpu オプションによる指定がなく，-cn オプション，または -cnv850e オプション[注]の指定がない場合，コンパイラは次のエラー・メッセージを出力し，コンパイルを中止します。

```
F2620: unknown cpu type, cannot compile
```

注 -cn オプション，または -cnv850e オプションを指定した場合でも，リンク時にはデバイス・ファイルが必要となります。

(2) オプション指定，#pragma 指令の両方でデバイスを指定した場合

コンパイラは警告メッセージを出力し，オプション指定を優先して処理を続行します。

ただし，複数のオプション指定，または複数の #pragma 指令で，異なるデバイス名を指定した場合，コンパイラは次のエラー・メッセージを出力し，処理を中止します。

```
F2622: duplicated cpu type
```

(3) アセンブラ命令で記述したプログラムの場合

この場合も，リンク可能なオブジェクト・ファイル作成時に，アセンブラのオプション，または .option 疑似命令によるデバイス指定が必要です。

第 3 章　拡張言語仕様

この章では，CA850 で拡張されている言語仕様について説明します。

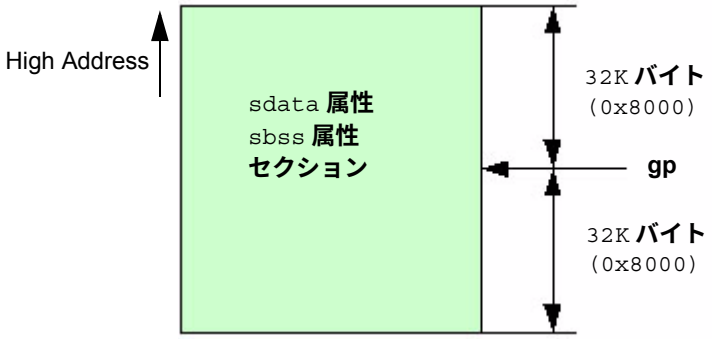
拡張仕様には，データ／テキストのセクション配置指定や，デバイス内蔵の周辺 I/O レジスタのアクセスを C 言語レベルで行う方法，C 言語にアセンブラ命令の記述を挿入する方法，関数ごとにインライン展開を指定する方法，割り込みや例外発生時のハンドラの定義，割り込み禁止指定を C 言語レベルで行う方法，ターゲット環境にリアルタイム OS を使用した場合に有効なリアルタイム OS 用機能，C 言語への命令の組み込みなどがあります。

3.1 データのセクション割り当て

CA850 は，C 言語ソース上で外部変数やデータが定義されると，それらをメモリ上に配置します。配置されるメモリ領域は，基本的に“ グローバル・ポインタ（gp）”の指すアドレスからのオフセットによって参照できる領域です。つまり，プログラム中で変数やデータにアクセスする場合，デフォルトで gp を使ってアクセスするコードを出力しようとします。

さらに CA850 は，できるだけオブジェクト効率や実行効率を上げるため，gp から 1 命令で参照可能な領域に配置するコードを出力しようとします。しかし，gp から 1 命令で参照可能な範囲は，V850 アーキテクチャ上，gp から± 32K バイト内（合計で 64K バイト内）である必要があります。CA850 では，gp から± 32K バイト内の領域に専用のセクションが設けており，“ sdata 属性／ sbss 属性セクション ”と呼びます。

図 3 － 1 sdata 属性／ sbss 属性セクション



しかし，変数の数が多いアプリケーションでは，この範囲内に収まりきらない場合がよくあります。その場合は，それ以外のセクションに割り当てなくてはなりません。CA850 では sdata 属性／ sbss 属性セクション以外にも，変数やデータを配置するためのさまざまなセクションを用意しています。それぞれに特徴があり，より高速にアクセスしたい変数を配置できるセクションなどもあるので，用途によって使い分けることができます。

次に，sdata 属性／ sbss 属性セクションを含め，CA850 で使用できるセクションを示します。

(1) sdata 属性／ sbss 属性セクション

gp から 1 命令で参照可能なセクションで，gp から± 32K バイト内に配置される必要があります。初期値ありデータは sdata 属性セクションへ，初期値なしデータは sbss 属性セクションへ配置されます。

CA850 では，まずこのセクションに配置するコードを生成しようとします。

ただし，この属性のセクションに収まりきらないような場合は，data 属性／ bss 属性セクションへ配置するコードを生成します。

なお，sdata 属性／ sbss 属性セクションへの配置データを少しでも多くする方法として，CA850 のオプション“ -G ”で，配置されるデータのサイズの上限を指定し，それ以上のサイズのデータは sdata 属性／ sbss 属性セクションに配置しないという指定ができます（オプションの詳細は“ CA850 ユーザーズ・マニュアル 操作編 ”を参照）。

なお，プログラム中で sdata 属性／ sbss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section sdata begin
int    a = 1;        /* sdata セクション配置 */
int    b;            /* sbss セクション配置 */
#pragma section sdata end
```

(2) data 属性／ bss 属性セクション

gp から 2 命令で参照可能なセクションです。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。

したがって，RAM 上であれば，どこにでも配置が可能なセクションです。

なお，C 言語プログラム中で data 属性／ bss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section data begin
int    a = 1;        /* data セクション配置 */
int    b;            /* bss セクション配置 */
#pragma section data end
```

(3) sconst 属性セクション

r0，つまり，0 番地から 1 命令で参照可能なセクションで，0 番地から +32K バイト内に配置される必要があります。基本的に “ ROM に固定してもよいデータ ” を配置するセクションです。V850 で内蔵 ROM を持つデバイスの場合，0 番地からプラス方向が内蔵 ROM である場合が多く，そこに 1 命令で参照したい，かつ ROM 固定してもよいデータを，sconst 属性セクションとして配置します。sconst 属性セクションに配置するデータは，const 修飾子をつけて宣言された変数／データが対象となります。この属性のセクションに収まりきらないような場合は，const 属性セクションへ配置することになります。

なお，sconst 属性セクションへの配置データを少しでも多くする方法として，CA850 のオプション “ -Xsconst ” で，配置されるデータのサイズの上限を指定し，それ以上のサイズは sconst 属性セクションに配置しないという指定ができます（オプションの詳細は “ CA850 ユーザーズ・マニュアル 操作編 ” を参照）。

なお，プログラム中で sconst 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します。（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section sconst begin
const int a = 1;     /* sconst セクション配置 */
#pragma section sconst end
```

(4) const 属性セクション

r0，つまり，0 番地から 2 命令で参照可能なセクションです。sconst 属性セクションに入りきらなかった“ROM 固定してもよいデータ”や，V850 の ROM レス品で，外部 ROM にデータを配置したい場合に，const 属性セクションに配置します。const 属性セクションに配置するデータは const 修飾子をつけて宣言された変数／データが対象になります。

また，const 修飾子をつけて宣言された変数，文字列定数は #pragma section 指令で .const セクションに配置する指令がない場合でも const 属性セクションに割り付けられます。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。したがって，32 ビット空間内であれば，どこにでも配置が可能なセクションです。

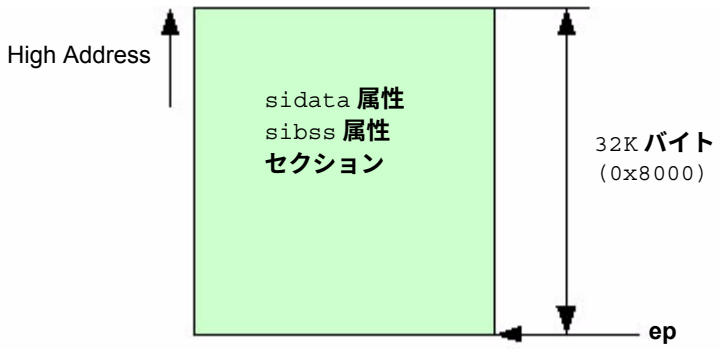
なお，プログラム中で const 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します。（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section const begin
const int a = 1;     /* const セクション配置 */
#pragma section const end
```

(5) sidata 属性／ sibss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。つまり，ep からプラス方向 32K バイト内に配置されるセクションです。

図 3 － 2 sidata 属性／ sibss 属性セクション



初期値ありデータは sidata 属性セクションへ，初期値なしデータは sibss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性／ sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。sidata 属性／ sibss 属性セクションは ep からプラス方向にアクセスするためのセクションですが，sedata 属性／ sebss 属性セクションは ep からマイナス方向にアクセスするためのセクションです。

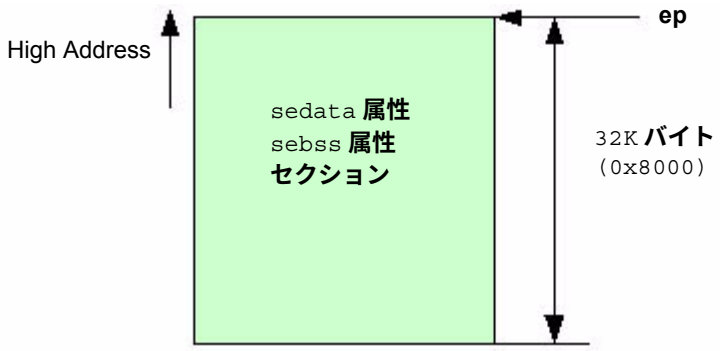
なお，プログラム中で sidata 属性／ sibss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section sidata begin
int    a = 1;        /* sidata セクション配置 */
int    b;            /* sibss セクション配置 */
#pragma section sidata end
```

(6) sedata 属性／sebss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からマイナス方向へアクセスするセクションです。つまり，ep からマイナス方向 32K バイト内に配置されるセクションです。

図 3 － 3 sedata 属性／ sebss 属性セクション



初期値ありデータは sedata 属性セクションへ，初期値なしデータは sebss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性／ sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。

sidata 属性／ sibss 属性セクションは ep からプラス方向にアクセスするためのセクションですが，sedata 属性／ sebss 属性セクションは ep からマイナス方向にアクセスするためのセクションです。

なお，プログラム中で sedata 属性／ sebss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section sedata begin
int    a = 1;        /* sedata セクション配置 */
int    b;            /* sebss セクション配置 */
#pragma section sedata end
```

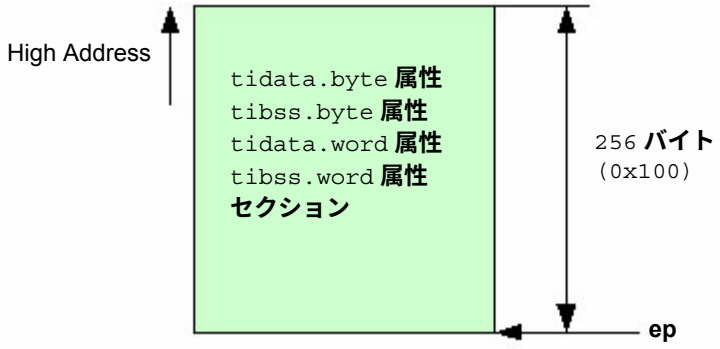
(7) tidata（tidata.byte，tidata.word）属性／tibss（tibss.byte，tibss.word）属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。sidata 属性／ sibss 属性セクションと違うところは，同じ 1 命令アクセスでも，使用するアセンブル命令が違うことです。

sidata 属性／ sibss 属性セクション，および sedata 属性／ sebss 属性セクションは，格納／参照に 4 バイト長命令の " st ／ ld 命令 " を使用しますが，tidata 属性／ tibss 属性セクションは，2 バイト長命令の " sst ／ sld 命令 " を使用してアクセスします。つまり，sidata 属性／ sibss 属性セクション，および sedata 属性／ sebss 属性セクションよりもコード効率がよくなります。

ただし，sst ／ sld 命令が適用できる範囲は小さいので，多くの変数を配置することはできません。

図 3 － 4 tidata 属性／ tibss 属性セクション



初期値ありデータは tidata（tidata.byte，tidata.word）属性セクションへ，初期値なしデータは tibss（tibss.byte，tibss.word）属性セクションへ配置されます。バイト・データを配置する場合は tidata.byte ／ tibss.byte 属性を，ワード・データを配置する場合は tidata.word ／ tibss.word 属性を指定しますが，CA850 に自動判別させたい場合は tidata ／ tibss 属性を指定します。

システムの中でも，より高速にアクセスしたいデータを配置するために使用します。

ただし，配置できる量が少ないため，厳選する必要があります。プログラム中で tidata.byte ／ tibss.byte 属性，tidata.word ／ tibss.word 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「3.1.1 #pragma section 指令」を参照）。

```
#pragma section tidata_byte begin
char   a = 1;        /* tidata.byte セクション配置 */
unsigned char b;     /* tibss.byte セクション配置 */
#pragma section tidata_byte end
```

```
#pragma section tidata_word begin
int    a = 1;        /* tidata.word セクション配置 */
short  b;            /* tibss.word セクション配置 */
#pragma section tidata_word end
```

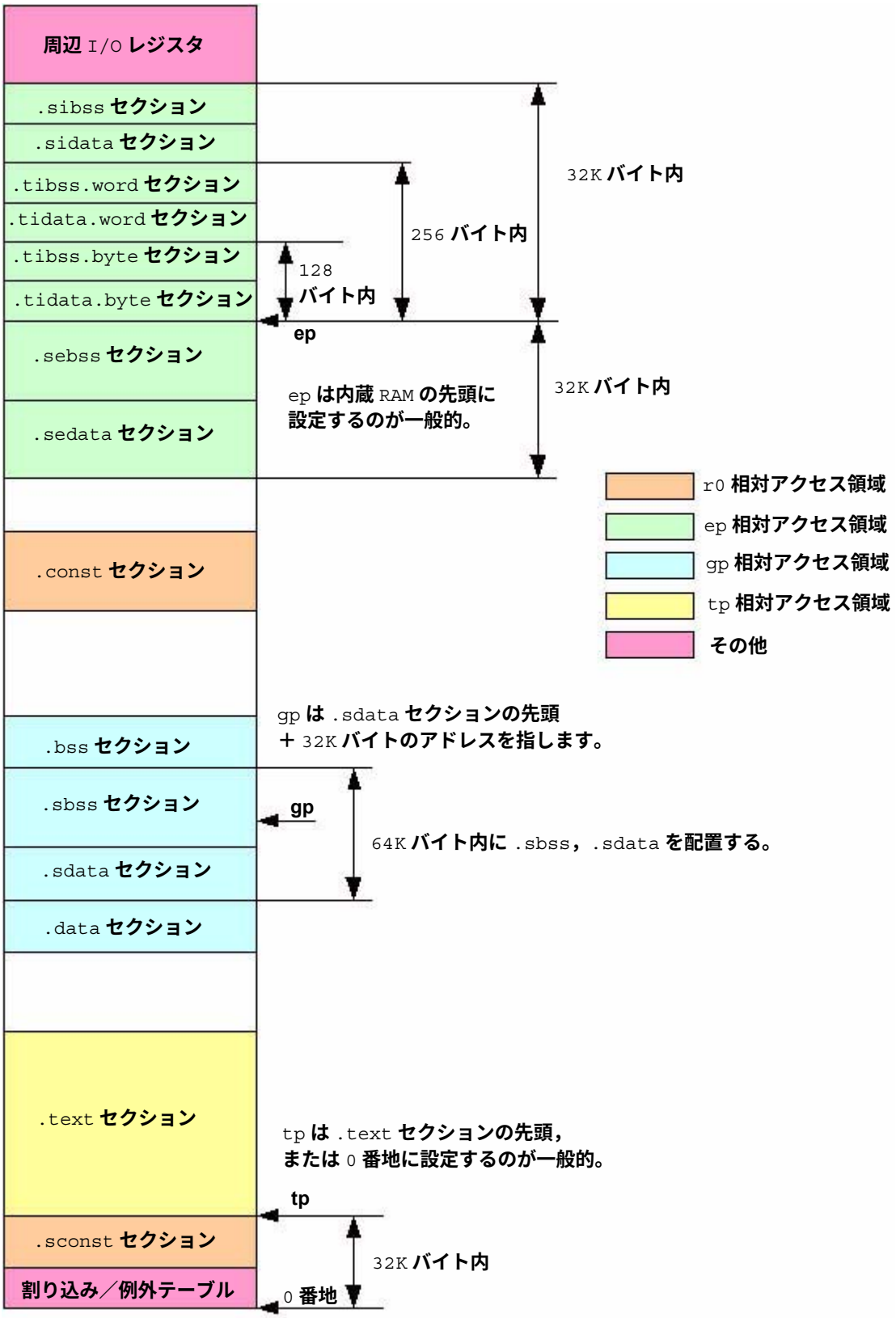
```
#pragma section tidata begin
int    a = 1;        /* tidata.word セクション配置 */
char   b;            /* tibss.byte セクション配置 */
#pragma section tidata end
```

変数／データの中で，特にシステムの中で参照頻度の高いものを選び，tidata（tidata.byte，tidata.word）属性／ tibss（tibss.byte，tibss.word）属性セクションに配置できると，実行速度の面からも効率がよくなります。CA850 には，この参照頻度を調査する“ セクション・ファイル・ジェネレータ ”があります。調査した頻度情報を“ 頻度情報ファイル ”として出力し，その情報を元にして，自動的に tidata（tidata.byte，tidata.word）属性／ tibss（tibss.byte，tibss.word）属性セクションに配置するコードを出力します。またその頻度情報ファイルをユーザが編集し，優先的に tidata（tidata.byte，tidata.word）属性／ tibss（tibss.byte，tibss.word）属性セクションに割り当てたい変数を選択することもできます。この方法を用いると，ソースに手を加えることなく，これらのセクションに配置することができます。

なお，セクション・ファイル・ジェネレータや頻度情報ファイルについての詳細は“ CA850 ユーザーズ・マニュアル 操作編 ”を参照してください。

次図に各セクションのメモリ配置イメージの例を示します。

図 3 － 5 各セクションのメモリ配置イメージ



3.1.1 #pragma section 指令

#pragma section 指令を使って，目的のセクションへデータを割り当てる方法を説明します。

(1) セクション名をデフォルトのまま使用する場合

CA850 で定義されているセクション名をそのまま使用する場合，#pragma section 指令で次のような書式で記述します。

```
#pragma section セクション種別 begin
変数宣言／定義
#pragma section セクション種別 end
```

ここで " セクション種別 " に指定できるのは，次のとおりです。

- data
- sdata
- sedata
- sidata
- tidata
- tidata.word
- tidata.byte
- sconst
- const

bss 属性のセクション名をセクション種別に指定することはできません。bss 属性については，宣言／定義された変数・データに " 初期値がある場合 " は data 属性に，" 初期値がない場合 " は bss 属性に CA850 が自動的に振り分けます。

```
#pragma section sdata begin
int    a = 1;        /* sdata セクション配置 */
int    b;            /* sbss セクション配置 */
#pragma section sdata end
```

上記の場合，" 変数 a " は初期値をもつので，data 属性の " .sdata セクション " へ，" 変数 b " は初期値を持たないので，bss 属性の " sbss セクション " へ配置します。

" #pragma section セクション種別 begin " と " #pragma section セクション種別 end " の間には，変数宣言／定義を複数記述することができます。そのセクション種別に配置したい変数を列挙します。

なお，tidata.word と tidata.byte をセクション種別に指定する場合，次のように，間の " .（ピリオド）" の代わりに " _（アンダースコア）" を指定してください。

- tidata_word
- tidata_byte

(2) セクション名に独自の名前をつける場合

次の属性を持つセクションに関しては，独自のセクション名を指定し，そこに変数やデータを配置することができます。

- data
- sdata
- sconst
- const

この場合，#pragma section 指令で次のような書式で記述します。

```
#pragma section セクション種別 “作成するセクション名” begin
変数宣言／定義
#pragma section セクション種別 “作成するセクション名” end
```

ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが作成するセクション名に“．セクション種別”が付加されたものになります。

```
作成するセクション名．セクション種別
```

これは，初期値の“ある”／“なし”で，data 属性，bss 属性に分かれるためで，同一セクション名で別属性のセクションができてしまうのを防ぐためです。リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「3.1.2 独自のデータ・セクションのリンク・ディレクティブ指定」を参照してください。ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3 － 1 ユーザが独自に指定するセクション名と生成されるセクション名

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mydata	data 属性	.data ／ .bss	mydata.data ／ mydata.bss
mysdata	sdata 属性	.sdata ／ .sbss	mysdata.sdata ／ mysdata.sbss
myconst	const 属性	.const	myconst.const
mysconst	sconst 属性	.sconst	mysconst.sconst

次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata "mysdata" begin
int    a = 1;        /* mysdata.sdata セクション配置 */
int    b;            /* mysdata.sbss セクション配置 */
#pragma section sdata "mysdata" end
```

3.1.2 独自のデータ・セクションのリンク・ディレクティブ指定

#pragma section 指令で，独自のセクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

C 言語ソースにて“変数 a”と“変数 b”を次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata "mysdata" begin
int    a = 1;        /* mysdata.sdata セクション配置 */
int    b;            /* mysdata.sbss セクション配置 */
#pragma section sdata "mysdata" end
```

このとき，リンク・ディレクティブ・ファイル内のマッピング・ディレクティブは，次のように記載すると，独自のセクションに配置されます。

```
.data = $PROGBITS ?AW .data;
.bss = $NOBITS ?AW .bss;

mysdata.data = $PROGBITS ?AW mysdata.data;
mysdata.bss = $NOBITS ?AW mysdata.bss;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mysdata.data の属性が“$PROGBITS ?AW”，mysdata.bss の属性が“$NOBITS ?AW”なので，これらと同じ属性を持つマッピング・ディレクティブにて，入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる“.data”“.bss”“mysdata.data”“mysdata.bss”のこと）を省略しないでください。

例

```
.data = $PROGBITS ?AW;
.bss = $NOBITS ?AW;
```

上記のように，同じ“$PROGBITS ?AW”属性，“$NOBITS ?AW”属性のマッピング・ディレクティブで，入力セクションが省略されていると，リンカはその属性のセクションをすべて結合して配置するため，独自に作ったセクションへ配置されない結果となります。つまり，mysdata.data に配置しようとしたデータは .data に，mysdata.bss に配置しようとしたデータは .bss に配置されることになります。

リンク・ディレクティブ・ファイルの書式などについての詳細は“CA850 ユーザーズ・マニュアル リンク・ディレクティブ編”を参照してください。

3.1.3 セクション割り当ての注意点

#pragma section 指令，const 修飾子，およびセクション・ファイルによってセクションを割り当てた場合の注意点を次に示します。

(1) #pragma section 指令は，次のような指定をするとコンパイル時にエラーになります。

- セクション割り当てがネストしている
- #pragma section の begin と end がクロスしている
- #pragma section の begin と end のどちらかしか記述していない

[誤った例 " セクションのネスト "]

```
#pragma section data begin
int    a = 1

#pragma section sdata begin
short  b;
char   c=0x10
#pragma section sdata end

int    d;
#pragma section data end
```

[誤った例 " セクションのクロス "]

```
#pragma section data begin
int    a = 1

#pragma section sdata begin
short  b;
char   c=0x10
#pragma section data end

int    d;
#pragma section sdata end
```

(2) 自動変数に対してセクション指定を行った場合，その指定は無視されます。セクション指定は外部変数に対する機能です。

(3) 独自のセクション名を指定する場合，その名前は 256 文字以内にしてください。

(4) 初期値を設定しない変数宣言は，通常 " 仮定義 " として扱われますが，セクション指定した場合は " 定義 " として扱われます。初期値を設定しない変数宣言と定義を混在させないようにしてください。

[#pragma section を使用しない変数宣言]

```
int    i;     /* 仮定義 */
int    i=10;  /* 定義 */
```

【エラーになりません】

[#pragma section を使用した変数宣言]

```
#pragma section sedata begin
int    i;     /* 定義 */
int    i=10;  /* 定義 */
#pragma section sedata end
```

【二重定義エラー】

また，通常の外部変数の“仮定義”に対してセクション指定した場合，“定義”として扱われます。外部変数を参照するファイルでは，必ず extern 宣言してください。次の場合では，file1.c 側の変数の仮定義で extern がないと，二重定義エラーになります。

[file1.c] #pragma section sedata begin extern int i; #pragma section sedata end	[file2.c] #pragma section sedata begin int i; #pragma section sedata end
【extern がないと，二重定義エラー】	

(5) セクション指定した変数を，他のファイルで参照する場合，その変数に対する extern 宣言に対しても，同じセクション種別でセクション指定する必要があります。変数定義時に指定したセクションと異なる種別のセクションを指定した場合はエラーになります。

たとえば，定義側で“#pragma section data begin ～ #pragma section data end”指定して，仮定義側（extern 宣言）で“#pragma section data begin ～ #pragma section data end”指定しなかった場合，仮定義側では sdata に配置されているものとみなされます。つまり，定義側では gp からの 2 命令でアクセスするコードが出力され，仮定義側では gp からの 1 命令でアクセスするコードが出力されることになります。この場合，つじつまが合わなくなるため，リンク時に次のエラー・メッセージが出力されます。

```
F4163: output section ".data" overflowed or illegal label reference forsymbol
"symbol" in file "file" (value: value, input section: section, offset: offset,
type:R_V850_GPHWLO_1). "symbol" is allocated in section ".data" (file: file).
```

[正しい例]

[file1.c] #pragma section sedata begin int i=1; #pragma section sedata end	[file2.c] #pragma section sedata begin extern int i; #pragma section sedata end

[誤った例 1]

[file1.c] int i=1;	[file2.c] #pragma section sedata begin extern int i; #pragma section sedata end

file1.c で定義した“変数 i”は sbss セクション，または bss セクションに配置されますが，file2.c では“変数 i”に対して sebss セクションへのアクセス・コードが出力されるため，リンカで次のエラー・メッセージが出力されます。

```
F4163: output section ".sebss" overflowed or illegal label reference forsymbol
"_i" in file "file2.o" (value: value, input section: section, offset: offset,
type: type). "_i" isallocated in section ".sbss" (file: file1.o).
```

［誤った例 2］

［file1.c］	［file2.c］
#pragma section sedata begin int i=1; #pragma section sedata end	extern int i;

file1.c で定義した“変数 i ”は sebss セクションに配置されますが，file2.c では“変数 i ”に対して sbss セクション，または bss セクションへのアクセス・コードが出力されるため，リンカで次の不整合エラーが出力されます。

```
F4156: can not find GP-symbol in segment "*DUMMY*" or illegal labelreference
for symbol "_i" in file "file2.o" (section: section, offset: offset,
type:R_V850_GPHWLO_1). "_i" is allocated in section ".sedata" (file: file1.o).
```

(6) #pragma section 指令で，sconst 属性，const 属性の変数を定義する場合，変数に必ず“ const 指定 ”をしてください。また extern 宣言で仮定義している箇所でも，同じく const 指定が必要です。

“ #pragma section sconst begin ～ #pragma section sconst end ”や“ #pragma section const begin ～ #pragma section const end ”で指定しても，変数宣言時に const 宣言がなかった場合，sconst セクション／ const セクションに配置されず（#pragma section 指令は無視され），gp 相対セクションである sdata セクションや data セクションに配置されてしまい，意図した配置にならないことになります。

［file1.c］	［file2.c］
#pragma section sconst begin const int i=1; #pragma section sconst end	#pragma section sconst begin int i; #pragma section sconst end

file1.c の方は，“変数 i ”が sconst セクションに配置するコードが出力されますが，file2.c の方は，“変数 i ”に const 指定がないため，#pragma section 指定が無視され，gp 相対の変数として扱われるため，sdata セクションや data セクションに配置されるコードになります。リンクしても sconst に“変数 i ”が配置されません。

また，次のように extern 宣言で仮定義している箇所でも const 指定が必要になります。

［file1.c］	［file2.c］
#pragma section sconst begin const int i=10; #pragma section sconst end	#pragma section sconst begin extern const int i; #pragma section sconst end

3.1.4 #pragma section 指令の例

次に，#pragma section 指令の例を数点示します。

(1) 変数 a を tidata.word セクションに，変数 b を tibss.word セクションに配置する

```
#pragma section tidata_word begin
int    a = 1;        /* tidata.word セクションに配置 */
short  b;            /* tibss.word セクションに配置 */
#pragma section tidata_word end
```

(2) 変数 c を tidata.byte セクションに，変数 d を tibss.byte セクションに配置する

```
#pragma section tidata_byte begin
char   c = 0x10;     /* tidata.byte セクションに配置 */
char   d;            /* tibss.byte セクションに配置 */
#pragma section tidata_byte end
```

tidata 属性セクションでは，ワード・データ／ハーフワード・データは tidata_word ／ tibss_word セクションに配置され，バイト・データは tidata_byte ／ tibss_byte セクションに配置されます。

ただし C 言語ソースで“ char 型の配列 ”が宣言された場合，それらは“ tidata.word セクション ”に配置されます。tidata.word セクションは 256 バイトまで使用可能ですが，配列は char 型なので sld.b ／ sst.b 命令を使ったコードが出力されます。

しかし，sld.b ／ sst.b 命令は 128 バイト以上にアクセスすることができません。

したがって，char 型の配列を宣言したとき，その配列自体が 128 バイト以上，または，ep からの相対で 128 バイトを越えた場所に配置されると，リンク時にエラーとなってしまいます。

したがって，char 型の配列を tidata 属性セクションに割り当てる場合は注意が必要です。

(3) const 指定された変数 e を sconst セクションに，ポインタ p が示す文字列定数データを sconst セクションに配置する

```
#pragma section sconst begin
const int     e = 0x10;
const char    *p = "Hello,World";
#pragma section sconst end
```

上記の記述で，ポインタ p の示す“ Hello World ”は sconst セクションへ，ポインタ変数“ p ”自体は sdata セクション，または data セクションへ配置されます。ポインタ変数とポインタの示す内容の配置場所に関しては，const 指定の方法によって変わってきます。

例 1

```
const char    *p = "Hello,World";
```

このように宣言すると，次のようになります。

ポインタ変数“ p ”	書き換え可能（“ p=0 ”はコンパイル OK）
文字列定数“ Hello World ”	書き換え不可能（“ *p=0 ”はコンパイル NG）

ポインタ変数の指す先を const 属性に配置したい場合は,上記のように記述します。ポインタ自体を ROM に固定するような場合に使用します。

```
#pragma section sconst begin
const char    *p = "Hello,World";
#pragma section sconst end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sdata ／ data セクション
文字列定数“Hello World”	sconst セクション

例 2

```
char   *const p;
```

ポインタ変数“p”	書き換え不可能（“p=0”はコンパイル NG）

ポインタ変数を const 属性に配置したい場合は，上記のように記述します。ポインタ自体を ROM に固定するような場合に使用します。

```
char *const   p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst begin
char *const   p = "Hello World";
#pragma section sconst end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

例 3

```
const char    *const p;
```

ポインタ変数“p”	書き換え不可能（“p=0”はコンパイル NG）

ポインタ変数，およびその指す先を const 属性に配置したい場合は，上記のように記述します。どちらも ROM に固定するような場合に使用します。

```
const char *const   p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst begin
const char *const   p = "Hello World";
#pragma section sconst end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

なお，const 指定で宣言した変数を sconst セクションに配置する方法として #pragma section 指令のほかに，コンパイラ・オプション“-Xsconst”指定があります。

(4) #pragma section 指令の extern 宣言する方は，共通に使用するヘッダ・ファイル内で行い，C 言語ソース内にインクルードして使用する。

```
[header.h]
#pragma section sidata begin
extern int    k;
#pragma section sidata end
```

```
[file1.c]
#include "header.h"
#pragma section sidata begin
int    k;
#pragma section sidata end
```

```
[file2.c]
#include “header.h”
void func1(void)
{
       k = 0x10;
}
```

上記のように #pragma section 指令の extern 宣言する方は，ヘッダ・ファイルでまとめておくと，間違いが少なくなり，ソースも見やすくなります。

3.2 関数のセクション割り当て

CA850 では，C 言語ソースの関数であるプログラム・コードは，デフォルトで“ .text セクション ”に配置されます。リンク・ディレクティブ・ファイルで，.text セクションの配置アドレスを決定すると，そこからプログラムを配置します。

しかし，「関数ごとに配置アドレスを変えたい」場合や，「メモリの配置上，配置アドレスを分ける必要がある」場合があります。これに対応するため，CA850 では“ #pragma text 指令 ”を用意しています。#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけ，リンク・ディレクティブ・ファイルにて配置アドレスを変えます。

3.2.1 #pragma text 指令

#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけることができます。#pragma text 指令の使い方には，次の二通りあります。

- #pragma text 指令に，作成するセクションに配置する“ 関数名 ”を指定する方法
- 関数本体（関数定義）の前に #pragma text 指令を記述し，関数名は指定しない方法

(1) #pragma text 指令に，作成するセクションに配置する“ 関数名 ”を指定する方法

```
#pragma text “作成するセクション名” 関数名
```

関数名は，C 言語記述の関数名を記述してください。たとえば "void func1(){}" という関数であれば "func1" と指定します。また，“ 作成するセクション名 ”を省略することもできます。その場合“ 関数名 ”で指定した関数を，デフォルトの“ .text セクション ”に配置します。

(2) 関数本体（関数定義）の前に #pragma text 指令を記述し，関数名は指定しない方法

```
#pragma text “作成するセクション名”
```

“ 作成するセクション名 ”を省略することもできます。その場合，その直前に指定した“ #pragma text ”作成するセクション名の指定を解除し，これ以降の関数を，デフォルトの“ .text セクション ”に配置します。

ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが指定したセクション名に“ .text ”が付加されたものになります。

```
セクション名 .text
```

リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「3.2.2 独自のテキスト・セクションのリンク・ディレクティブ指定」を参照してください。

ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3 － 2 ユーザが独自に指定するセクション名と生成されるセクション名（text）

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mytext	text 属性	.text	mytext.text

次のように指定した場合，“ 関数 func1 ” は “ mytext1.text セクション ” へ，“ 関数 func2 ” は #pragma text 指令がないので，デフォルトで “ .text セクション ” へ配置されます。

```
#pragma text "mytext1" func1
void func1(void)
{
       :
}
void func2(void)
{
       :
}
```

また，次のように指定した場合，“ 関数 func1 ” ，“ 関数 func2 ” は “ mytext2.text セクション ” へ，“ 関数 func3 ”は“ mytext3.text セクション ”へ，“ 関数 func4 ”は“ #pragma text ”で直前の“ #pragma text ”mytext3 ” が解除されているので，デフォルトで “ .text セクション ” へ配置されます。

```
#pragma text "mytext2"
void func1(void)
{
       :
}
void func2(void)
{
       :
}
#pragma text "mytext3"
void func3(void)
{
       :
}
#pragma text
void func4(void)
{
       :
}
```

3.2.2 独自のテキスト・セクションのリンク・ディレクティブ指定

#pragma text 指令で，独自のテキスト・セクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

```
#pragma text "mytext2"
void func1(void)
{
       :
}
void func2(void)
{
       :
}
#pragma text "mytext3"
void func3(void)
{
       :
}
#pragma text
void func4(void)
{
       :
}
```

C 言語ソースにて，上記のように #pragma text 指定した場合，“ 関数 func1 ”, “ 関数 func2 ” は “ mytext2.text セクション ” へ，“ 関数 func3 ” は “ mytext3.text セクション ” へ，“ 関数 func4 ” は “ #pragma text ” で直前の “ #pragma text ” mytext3 ” が解除されているので，デフォルトで “ .text セクション ” へ配置されます。

```
.text    = $PROGBITS  ?AX .text;
mytext2  = $PROGBITS  ?AX mytext2.text;
mytext3  = $PROGBITS  ?AX mytext3.text;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また，直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mytext2.text ／ mytext3.text の属性が “ $PROGBITS ?AX ” なので，これらと同じ属性を持つマッピング・ディレクティブにて，入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる “ .text ”“ mytext2.text ”“ mytext3.text ” のこと）を省略しないでください。

例

```
.text    = $PROGBITS ?AX;
```

上記のように同じ “ $PROGBITS ?AX ” 属性のマッピング・ディレクティブで，入力セクションが省略されていると，リンカはその属性のセクションをすべて結合して配置するため，独自に作ったセクションへ配置されない結果となります。

したがって,mytext2.text ／ mytext3.text に配置しようとしたプログラムは .text に配置されることになります。

リンク・ディレクティブ・ファイルの書式などについての詳細は “ CA850 ユーザーズ・マニュアル リンク・ディレクティブ編 ” を参照してください。

3.2.3 #pragma text 指令の注意点

#pragma text 指令の注意点を次に示します。

(1) #pragma text 指令は，関数の定義と同一ファイル内で，定義より前に記述してください。関数の定義よりも後ろに記述された場合は，警告メッセージを出力し，#pragma text 指令は無視します。
ただし，関数のプロトタイプ宣言の順番には関係ありません。

(2) #pragma text 指定した関数が “ direct 配置指定された割り込みハンドラ ” である場合，警告メッセージを出力し，#pragma text 指令は無視します。
なお，direct 配置指定についての詳細は「3.7 割り込み／例外処理ハンドラ」を参照してください。

(3) #pragma text 指定した関数は，#pragma inline 指定や，最適化オプションによるインライン展開ができません。インライン展開指定は無視されます。

(4) セクション名を指定する場合，その名前は 256 文字以内にしてください。

3.3 周辺 I/O レジスタへのアクセス

周辺 I/O レジスタとは，各デバイスで内蔵している周辺機能のためのレジスタです。デバイス定義の周辺 I/O レジスタ名を用いることにより，C 言語レベルで，周辺 I/O へアクセスできます。周辺 I/O レジスタ名は，C 言語ソース・プログラム中で，通常の符号なし外部変数（unsigned）のように扱うことができます。

なお，指定可能なレジスタ名，属性などについては，各デバイスの“ユーザーズ・マニュアル ハードウエア編”を参照してください。

3.3.1 アクセス方法

周辺 I/O レジスタ名を使用できるようにするには，次の #pragma 指令を記述することにより行います。

```
#pragma ioreg
```

“#pragma ioreg ”を記述した C 言語ソース内では，それ以降，周辺 I/O レジスタ名を使用することができます。

上記の指令を行わずに，あるいは，適したデバイス名を指定せずに周辺 I/O レジスタ名を使用すると，「変数が未定義である」というエラーになります。

また，指定したレジスタ固有のアクセス属性に反した使用も，エラーになります。

次の場合，例 1 は正しいですが，例 2，例 3 はエラーとなります。

なお，次の例で，P0，P1，P2，RXBO，OVF0 は V850 マイクロコントローラの周辺 I/O レジスタを示します。このように，“レジスタ名”には，デバイス・ファイルで定義されている略号を指定します。

例 1

```
#pragma ioreg
void func1(void)
{
        int     i;
        P0 = 1;         /* P0 に書き込みます */
        i = RXB0;       /* RXB0 から読み込みます */
}

void func2(void)
{
        P1 = 0;         /* P1 に書き込みます */
}
```

例 2

```
void func(void)
{
        P1 = 0;         /* 未定義エラーです */
}
```

例 3

```
#pragma ioreg
void func(void)
{
        RXB0 = 1;      /* RXB0 の属性が読み込み専用のため，エラーです */
}
```

3.3.2 ビット・アクセス

CA850 では，周辺 I/O レジスタに対して，各ビットごとにアクセスすることもできます。“ ビット番号 ” は，32 ビットのレジスタの場合，0 ～ 31 で次のように指定します。

```
レジスタ名 . ビット番号 = ...
```

(1) ビット・アクセスの際の注意

(a) ビット・アクセスで 0 と 1 以外の値を代入した場合，その値の 2 進数表現における最下位の値が設定されます（この際，メッセージは出力されません）。

指定例 1

```
#pragma ioreg
void func(void)
{
        P0.1 = 1;              /* P0 のビット 1 を 1 にします */
        P2.3 = 0;              /* P2 のビット 3 を 0 にします */
}
```

(b) 各レジスタの持つフラグのビットにアクセスする場合，それぞれのビット名を用いてアクセスすることができます。ビット名には，デバイス・ファイルで定義されている名前を指定します。

指定例 2

```
#pragma ioreg
void func(void)
{
        OVF0 = 1;              /* ビット名 OVF0 のビットを 1 にします */
}
```

3.4 アセンブラ命令の記述

CA850 では，次に示す形式において，C 言語ソース・プログラム中にアセンブラ命令が記述できます。

- asm 宣言
- #pragma 指令

挿入するアセンブラ命令でレジスタを使用する場合，必要な退避／復帰はプログラム内で行ってください。CA850 では行いません。

また，アセンブラ命令の記述は，関数の中に挿入することを推奨します。関数の外に記述した場合，次の制限があり，警告が出力されます。

- 関数部分とのコード出力順序が保証されない
- 関数の存在しないファイルでは出力されない

(1) asm 宣言

```
__asm( 文字列定数 );    または          _asm( 文字列定数 );
```

[注意事項]

(a) _asm 形式は，従来の言語仕様との互換性のために提供されている形式であり，コンパイラで -ansi オプションが指定された場合，_asm 形式に対して警告メッセージを出力し，関数呼び出しとして扱われます。-ansi オプションを指定する場合，__asm 形式を使用してください。

(b) コンパイラは，asm 宣言が指定された場合，指定された文字列定数[注]の後ろにニューライン（\n）を付けてアセンブラに渡します。

注 なお，指定された文字列定数は，一般の文字列定数と異なり，ニューライン以外の文字が後ろに続く“\”は後ろに続く文字自身を示します（ニューラインが後ろに続く“\”は誤りとなります）。

例

```
__asm("nop ");
__asm(".str \"string\\0\"");
```

(c) __asm，または _asm は宣言であり，文として扱われません。このため，次に示す例 1 のように，C 言語の文法上，宣言のみの記述が許されない構文ではエラーとなります。
そこで，例 2 のように“{}”で囲んで複合文としてください。

例 1

```
if(i == 0)
        __asm("mov r11, r10");          /* 宣言のみのためエラーとなります */
```

例 2

```
if(i == 0){
        __asm("mov r11, r10");          /* 複合文となるため記述できます */
}
```

(2) #pragma 指令

この #pragma 指令で囲まれた範囲では，そのまま，アセンブラ命令が記述できます。複数のアセンブラ命令を記述する場合に有効です。

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

次に示す例 1 の記述は，例 2 の記述と同様の意味となります。

例 1

```
extern int i;
void f(void)
{
#pragma asm
        mov     r0, r10
        st.w    r10, $_i
                :
#pragma endasm
}
```

例 2

```
extern int    i;
void f(void)
 {
        __asm("mov r0, r10 ");
        __asm("st.w r10, $_i ");
        :
}
```

“ #pragma asm ” から “ #pragma endasm ” までの記述は，そのままアセンブラに渡されます。

したがって，CA850 が内部的にアセンブラ命令を作成し，アセンブラを起動します。そのため #pragma asm 宣言後に，アセンブラ命令の疑似命令を使用することも可能です。また，アセンブラ命令で，C 言語ソース・プログラム中のローカル変数は使用できません。ローカル変数は，CA850 で “ スタック ”，または “ レジスタ ” に割り当てられるためで，インライン・アセンブラでは使用することはできません。

C 言語ソース・ファイルの #define で定義したシンボルも“ #pragma asm ”から“ #pragma endasm ”までの記述では使用できません。ファイル内で #define 定義したマクロをアセンブラ命令で展開したい場合，次の方法で回避してください。

- #pragma asm ～ #pragma endasm 指令内で .macro 命令を使用してマクロ定義する
- C 言語ソースから関数コールでアセンブラ命令を呼ぶ

マクロ定義せず，そのままアセンブラ命令で書くのも 1 つの方法です。

3.5 割り込みレベルの制御

3.5.1 __set_il 関数

CA850 では，V850 マイクロコントローラの割り込みに対して，C 言語ソース上で，次の制御を行うことができます。

- 割り込み優先順位レベルの制御
- マスカブル割り込みの受け付けの許可／禁止（割り込みのマスク）

つまり，“ 割り込み制御レジスタ ” を操作することができます。

“ 割り込み優先順位レベル ” を制御する場合は，“ __set_il 関数 ” を用いて次のように指定します。

```
__set_il( 割り込みの優先順位レベル , “ 割り込み要求名 ” );
```

指定できる“ 割り込み要求名 ”は，デバイス・ファイルで定義されている“ マスカブル割り込みの要求名 ” です。デバイス・ファイルで定義されている要求名を使用するので，この関数を使用する C 言語ソースでは，#pragma ioreg 指令を記述する必要があります。

“ 割り込みの優先順位レベル ” として指定できる値は“ 1 ～ 8 ” の整数値です。V850 マイクロコントローラの割り込み優先順位レベルは“ 0 ～ 7 までの 8 段階 ”を指定するため，“ 割り込みの優先順位レベルを 5 にしたい ” 場合は，この関数で指定する割り込みの優先順位レベルは “ 4 ” と指定します。

例

```
__set_il(2, "INTP0");
```

上記の指定の場合，割り込み “ INTP0 ” の優先順位レベルは “ 1 ” になります。

次に，“ 割り込みに対して，マスカブル割り込みの受け付けの許可／禁止 ” を制御する場合は，次のように指定します。

```
__set_il( マスカブル割り込みの許可／禁止 , “ 割り込み要求名 ” );
```

“ マスカブル割り込みの許可／禁止 ” に設定できるのは “ -1 ” か “ 0 ” です。

表 3 － 3 マスカブル割り込みの許可／禁止

設定値	動作
-1	マスカブル割り込みの受け付け禁止（割り込みをマスク）
0	マスカブル割り込みの受け付け許可（割り込みのマスクを解除）

例

```
__set_il(-1, "INTP0");
```

上記のように指定すると，割り込み“ INTP0 ”のマスカブル割り込みの受け付けを禁止します（INTP0 をマスクします）。

なお，__set_il 関数では，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作は行いません。

3.5.2 __set_il 関数と割り込み制御レジスタ

V850 マイクロコントローラの割り込み制御レジスタの構成は，次のようになっています。

7	6	5	4	3	2	1	0
xxIFn	xxMKn	0	0	0	xxPRn2	xxPRn1	xxPRn0

__set_il 関数を使用した場合は，“ 優先順位レベル ”，または “ 割り込みマスク・フラグ ” のいずれか一方の設定になります。したがって，__set_il 関数では，割り込み要求フラグの設定はできません。

割り込み要求名 “ INTP000 ” で，その割り込み制御レジスタ名が “ P00IC0 ” の場合，割り込み優先順位レベルを 6 に設定するために，次のように記述します。

```
__set_il(7, "INTP000");
```

次のコードが出力されます。

```
        ld.b    P00IC0, r1
        andi    0xf8, r1, r1
        ori     0x6, r1, r1
        st.b    r1, P00IC0
```

つまり，“ 優先順位レベルの設定 ” である “ 下位 3 ビット（xxxPR02-xxxPR00）のみ ” を変更するコードが出力されます。

割り込み要求名が “ INTP000 ” で，その割り込み制御レジスタ名が “ P00IC0 ” の場合，このマスカブル割り込みを許可するために，次のように記述します。

```
__set_il(0, "INTP000");
```

次のコードが出力されます。

```
        clr1    6, P00IC0
```

つまり，“ 割り込みマスク・フラグのみ ” を変更するコードが出力されます。

割り込み制御レジスタに，直接，値を書き込む場合には，“ 優先順位レベル ”，“ 割り込みマスク・フラグ ”，および “ 割り込み要求フラグ ” のすべてに値が設定されます。

例：割り込み制御レジスタ名が “ P00IC0 ” の場合

```
P00IC0=0x6;
```

上記のように記述すると，次のコードが出力されます。

```
        mov     0x6, r29
        st.b    r29, P00IC0
```

つまり，次のとおりとなります。

- 優先順位レベルを 6 に設定
- マスカブル割り込みを許可
- 割り込み要求フラグ・クリア設定

3.6 割り込み禁止

CA850 では，C 言語ソースにおいて，マスカブル割り込みを禁止にすることができます。

マスカブル割り込みを禁止する方法には，大きく分けて次の二通りがあります。

- 関数内で部分的に割り込みを禁止する方法
- 関数全体の割り込みを禁止する方法

3.6.1 関数内で部分的に割り込みを禁止する方法

C 言語で記述した関数内で，部分的に割り込みを禁止する場合，アセンブラ命令の“ di 命令 ”と“ ei 命令 ”を使用することができますが，CA850 では C 言語ソースで割り込み制御を行うことのできる関数を用意しています。

表 3 － 4 割り込み制御関数

割り込み制御関数	動作	CA850 の処理
__DI();	すべてのマスカブル割り込みの受け付けを禁止します。	di 命令を生成
__EI();	すべてのマスカブル割り込みの受け付けを許可します。	ei 命令を生成

例：__DI()，__EI() 関数の記述方法とその出力コード

[C 言語ソース]

```
[C 言語ソース]
void func1(void)
{
            :
        __DI();
        /* 割り込みを禁止して行いたい処理を記述 */
        __EI();
            :
}
```

[出力コード]

```
[上の C 言語ソースの出力コード]
_func1:
        -- プロローグ・コード
            :
        di
        -- 割り込みを禁止して行いたい処理
        ei
            :
        -- エピローグ・コード
        jmp [lp]
```

3.6.2 関数全体の割り込みを禁止する方法

CA850 では，関数全体のマスカブル割り込み割り込みを禁止する “ #pragma block_interrupt ” 指令を用意しています。

次のような書式で記述します。

```
#pragma block_interrupt 関数名
```

関数名は，C 言語記述の関数名を記述してください。たとえば "void func1(){}" という関数であれば "func1" と指定します。

上記で “ 関数名 ” で指定された関数に対し，マスカブル割り込みを禁止します。「3.6.1 関数内で部分的に割り込みを禁止する方法」で説明したように，関数の最初に “ __DI() ” を，最後で “ __EI() ” を記述することもできますが，この場合だと，CA850 が出力する “ プロローグ・コード ”, “ エピローグ・コード ” に対してマスカブル割り込みを禁止／許可することができず，関数全体を完全に割り込み禁止にすることができません。

#pragma block_interrupt 指令を用いると，“ プロローグ・コード ” 実行の直前にマスカブル割り込みが禁止され，“ エピローグ・コード ” 実行直後にマスカブル割り込み割り込みが許可されます。そのため関数全体を完全に割り込み禁止にすることができます。

例

#pragma block_interrupt 指令の使用方法と，出力されるコードは次のとおりです。

[C 言語ソース]

```
#pragma block_interrupt func1
void func1(void)
{
            :
      /* 割り込みを禁止して行いたい処理を記述 */
            :
}
```

[上の C 言語ソースの出力コード]

```
_func1:
      di
      -- プロローグ・コード
            :
      -- 割り込みを禁止して行いたい処理
            :
      -- エピローグ・コード
      ei
      jmp [lp]
```

3.6.3 関数全体の割り込み禁止時の注意事項

関数全体の割り込みを禁止にした場合の注意事項について，次に示します。

(1) 同じ関数に対して，割り込みハンドラ指定と #pragma block_interrupt 指定された場合，割り込みハンドラ指定の方が優先され，割り込み禁止の設定は無視されます。

(2) 割り込み禁止となっている関数内で，次の関数を呼び出した場合，その呼び出しからの復帰時に，割り込み許可状態になるため，注意が必要です。

- #pragma block_interrupt 指定された関数
- 関数の先頭で割り込み禁止をし，最後で割り込み許可している関数

(3) #pragma block_interrupt 指令は，関数の定義と同一ファイル内で，かつ，定義より前に記述してください。
関数の定義より後ろに記述された場合，コンパイル時にエラーとなります。
ただし，関数のプロトタイプ宣言順とは関係ありません。

(4) #pragma block_interrupt 指定された関数は，#pragma inline 指令を指定したり，最適化オプションでインライン展開指定ができません。インライン展開指定は無視されます。

(5) #pragma block_interrupt 指定しても，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作を行うコードは出力されません。

3.7 割り込み／例外処理ハンドラ

CA850 では，C 言語で “ 割り込み ” や “ 例外 ” が発生したときに呼ばれる “ 割り込みハンドラ ”, “ 例外ハンドラ ” を記述することができます。ここでは，その記述方法などについて説明します。

3.7.1 割り込み／例外の発生

V850 マイクロコントローラでは，割り込みや例外が発生すると，その割り込みや例外に対応したハンドラ・アドレスにジャンプします。“ 割り込み要因 ” と “ ハンドラ・アドレス ” は一対一で対応しており，ハンドラ・アドレスの集合を “ 割り込み／例外テーブル ” と呼びます。

たとえば，V850ES/SG2 の場合の割り込み／例外テーブルは次のようになっています（先頭部分のみ掲載）。

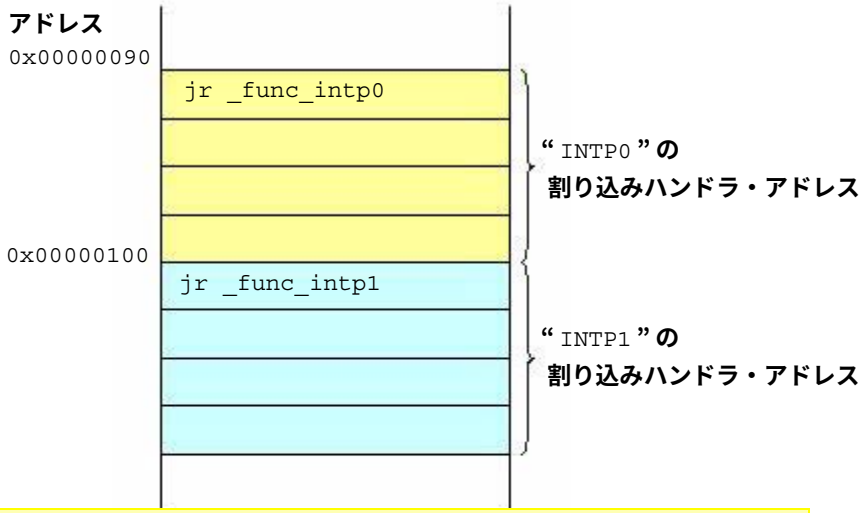
表 3 － 5 割り込み／例外テーブル（V850ES/SG2）

アドレス	割り込み名称	割り込みのトリガ
0x00000000	RESET	RESET 端子入力／内部要因からのリセット
0x00000010	NMI	NMI 端子有効エッジ入力
0x00000020	INTWDT2	WDT2 のオーバーフロー
0x00000040	TRAP0n	TRAP 命令
0x00000050	TRAP1n	TRAP 命令
0x00000060	LGOP/DBG0	不正命令コード／ DBTRAP 命令
0x00000080	INTLVI	低電圧検出
0x00000090	INTP0	外部割り込み端子入力エッジ検出（INTP0）
0x000000A0	INTP1	外部割り込み端子入力エッジ検出（INTP1）
0x000000B0	INTP2	外部割り込み端子入力エッジ検出（INTP2）
0x000000C0	INTP3	外部割り込み端子入力エッジ検出（INTP3）
⋮		

なお，ハンドラ・アドレスの並びや，搭載している割り込みは，V850 の品種ごとに異なります。詳細は，使用する各デバイスの “ ユーザーズ・マニュアル ハードウエア編 ” を参照してください。

各ハンドラ・アドレスは “ 16 バイト ” の領域を持っており，割り込みが発生すると，その 16 バイトの領域に書かれた命令を実行します。したがって，16 バイトで収まる処理であれば，ハンドラ・アドレス内だけで処理し，収まらなければ，処理の書かれた関数（割り込み／例外ハンドラ）への分岐命令を記述することになります。

図 3 － 6 ハンドラ・アドレスのイメージ



V850ES/SG2 で INTP0 割り込みが入ると，0x90 番地にジャンプし，そこにあるコードを実行します。上記の例の場合，関数 func_intp0 に分岐するコードが書かれているので，そこへ分岐します。つまり，「func_intp0 は INTP0 の割り込みハンドラ」ということになります

これらはアセンブリ言語ソース・レベルでの話になりますが，CA850 では，C 言語レベルで割り込み／例外処理を記述する場合，この点について留意することなく記述できるようになっています。具体的な記述方法は「3.7.3 割り込み／例外ハンドラの記述方法」で説明します。

3.7.2 割り込み／例外発生時に行う必要のある処理

関数実行時やタスク実行時に割り込み／例外が入ると，即座に割り込み／例外処理を行う必要があります。そして割り込み／例外処理が終わると，割り込みが入った時点の関数やタスクに戻る必要があります[注]。

したがって，割り込み／例外発生時には，そのときのレジスタ情報を保存し，割り込み／例外処理が終わった後は，そのレジスタ情報を復帰する必要があります。

注 リアルタイム OS 使用時のタスクの場合，割り込み内でのシステム・コール発行によって，割り込みが入った時点のタスクに戻らないことがあります。詳細は各リアルタイム OS のユーザーズ・マニュアルを参照してください。

通常の関数のプロローグ／エピローグ・コードでは，レジスタ変数用レジスタの退避／復帰を行います。
レジスタ変数用レジスタは次のとおりで，必要のあるものに関して退避／復帰を行います。

表 3 － 6 レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

割り込み／例外ハンドラに移る場合は，表 3 － 6 レジスタ変数用レジスタのほかに，割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表 3 － 7 割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	割り込み／例外発生時に退避／復帰するレジスタ
22 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14， r31（lp），CTPC【V850E】，CTPSW【V850E】
26 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14，r15，r16， r31（lp），CTPC【V850E】，CTPSW【V850E】
32 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14，r15，r16，r17，r18，r19， r31（lp），CTPC【V850E】，CTPSW【V850E】

また，多重割り込み／例外が発生した場合は，表 3 － 6 レジスタ変数用レジスタのほかに，多重割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表 3 － 8 多重割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	多重割り込み／例外発生時に退避／復帰するレジスタ
22 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14， r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】
26 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14，r15，r16， r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】
32 レジスタ・モード	r1，r6，r7，r8，r9， r10，r11，r12，r13，r14，r15，r16，r17，r18，r19， r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】

上記のレジスタの用途は，次のとおりです。

表 3 － 9 レジスタの用途

レジスタ	用途
r1	アセンブラ予約レジスタ
r6 - r9	引数用レジスタ（関数の引数をセットするためのレジスタ）
r10 番台	作業用レジスタ（CA850 がコード生成時に使用するレジスタ）
r31	リンク・ポインタ
CTPC【V850E】	CALLT 命令実行時のプログラム・カウンタ（PC）
CTPSW【V850E】	CALLT 命令実行時のプログラム・ステータス・ワード（PSW）
EIPC	割り込み／例外処理時のプログラム・カウンタ（PC）
EIPSW	EIPSW 割り込み／例外処理時のプログラム・ステータス・ワード（PSW）

割り込み／例外処理が終わると，退避したレジスタを復帰するコードを出力し，最後に reti 命令を出力します。この命令の発行により，割り込み／例外処理が終了したことを V850 に通知します。

CA850 では，“ レジスタの退避／復帰 ”，および “ reti 命令の出力 ” を「3.7.3 割り込み／例外ハンドラの記述方法」に従って記述すると自動的に出力します。“ レジスタの退避／復帰 ” については，「表 3 － 10 割り込みのレジスタの退避／復帰処理」に従って出力します。つまり，ユーザは，この点について留意する必要なく，割り込み／例外ハンドラ本体処理の記述に専念することができます。

表 3 － 10 割り込みのレジスタの退避／復帰処理

レジスタ名		レジスタ	説明
r1 レジスタ		r1	割り込み時には，必ず退避／復帰します。
引数用レジスタ		r6 - r9	割り込み要因が TRAP0 ／ TRAP1 の場合には，r6 は，必ず退避／復帰します。 関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば，退避／復帰します。
作業用レジスタ	22 レジスタ・モード	r10 - r14	関数コールが存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば退避／復帰します。
	26 レジスタ・モード	r10 - r16	
	32 レジスタ・モード	r10 - r19	
レジスタ変数用レジスタ	22 レジスタ・モード	r25 - r29	通常の関数と同様に，必要に応じて退避／復帰します。
	26 レジスタ・モード	r23 - r29	
	32 レジスタ・モード	r20 - r29	
リンク・ポインタ		r31 (lp)	関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，退避／復帰しません。
割り込み関連システム・レジスタ		EIPCE，EIPSW	多重割り込み __multi_interrupt 修飾子を使用した割り込み関数では，必ず退避／復帰します。 __interrupt 修飾子の場合には，退避／復帰しません。
callt 命令関連システム・レジスタ【V850E】		CTPC，CTPSW	V850E/V850ES/V850E2 コアのデバイスを指定してコンパイルしている割り込み関数では，必ず退避／復帰します。

3.7.3 割り込み／例外ハンドラの記述方法

割り込み／例外ハンドラの記述上の形態は，通常の C 言語関数と変わりませんが，C 言語で記述した関数を，CA850 に対して“割り込み／例外ハンドラ”として認識させる必要があります。CA850 では，割り込み／例外ハンドラの指定を“#pragma interrupt 指令”および“__interrupt 修飾子”，または“#pragma interrupt 指令”，および“__multi_interrupt 修飾子”で行います。

(1) 割り込み／例外ハンドラを指定する場合

`#pragma interrupt` 割り込み要求名 関数名 配置方法
`__interrupt` 関数定義，または関数宣言

(2) 多重割り込み／例外ハンドラを指定する場合

`#pragma interrupt` 割り込み要求名 関数名 配置方法
`__multi_interrupt` 関数定義，または関数宣言

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1(){}" という関数であれば "func1" と指定します。

“多重割り込みハンドラの指定”とは“多重割り込み「される」ことを許可する関数を指定”ということです。“多重割り込みする関数を指定する”ということではありません。

(a) 割り込み要求名

デバイス・ファイルに登録されている割り込み要求名を指定できます。デバイス・ファイルに登録されている割り込み要求名は，各デバイスの“ユーザーズ・マニュアル ハードウエア編”の“割り込み要求名”に書かれてある文字列になりますので，そちらを参照してください。

なお，ノンマスカブル割り込み（NMI）も，この方法で指定できますが，リセット割り込み（RESET）は指定できません。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。リセット時の処理は，スタート・アップ・ルーチンに記述するのが一般的であるため，詳細は「第 5 章 スタート・アップ・ルーチン」を参照してください。

(b) 関数名

割り込み／例外ハンドラとする関数名を指定します。ここでは“C 言語での関数名”を記述してください。void func1(void) という関数を指定する場合は，関数名に“func1”と指定します。

(c) 配置方法

ハンドラ・アドレスに関数本体を直接配置するか，それとも割り込み／例外ハンドラ関数への分岐命令だけを配置するかを指定します。直接配置するときだけに“direct”を指定します。直接配置しない場合は，“配置方法”には何も記述しないでください。direct 指定を行うことによって，指定した割り込み要因のハンドラ・アドレスからすべて配置されますが，その結果，以降のハンドラ・アドレスの領域も使用されることになる可能性があるため注意が必要です。

また，direct 指定をする場合は，関数定義より後ろに #pragma interrupt 指令を記述するとコンパイル時にエラーとなります。必ず関数定義より前で行ってください。

次に，#pragma interrupt 指定と __interrupt 修飾子，__multi_interrupt 修飾子の役割について説明します。

(1) #pragma interrupt 指令

#pragma interrupt で指定された“割り込み要求名”に対応するハンドラ・アドレスに，指定された関数への分岐命令（jr）を配置します。-Xj オプション指定時は，r1 レジスタをスタックに退避する命令と指定された関数への分岐命令（jmp）を配置します。

(2) __interrupt 修飾子

__interrupt 修飾子のついた関数に，割り込み／例外ハンドラとしてのレジスタの退避／復帰処理を加え，最後に reti 命令を加えます。-Xj オプション指定時は，r1 レジスタに関しては，退避処理はハンドラ・アドレスに出力されているので，関数では復帰処理のみ出力されます。

(3) multi_interrupt 修飾子

__multi_interrupt 修飾子のついた関数に，割り込み／例外ハンドラとしてのレジスタの退避／復帰処理を加え，EIPC，EIPSW レジスタの退避／復帰処理を加えます。また，最後に reti 命令を加えます。-Xj オプション指定時は，r1 レジスタに関しては，退避処理はハンドラ・アドレスに出力されているので，関数では復帰処理のみ出力されます。

“#pragma interrupt”と“__interrupt 修飾子，__multi_interrupt 修飾子”を同時に指定する場合は，次のコードが出力され，割り込み／例外処理として完全なハンドラが完成します。

- ハンドラ・アドレスに，指定された割り込み／例外ハンドラへの分岐命令を配置
- 割り込み／例外ハンドラとしてのレジスタの退避／復帰処理（__multi_interrupt 修飾子指定の場合はさらに EIPC，EIPSW の退避／復帰処理）を追加
- 割り込み／例外ハンドラの最後に reti 命令を追加

また，この場合，関数定義と #pragma interrupt 指令を別ファイルに記述することもできます。また，記述順序も問いません。ただし，配置方法で direct 指定した場合は，別ファイルに記述できません。

__interrupt 修飾子，__multi_interrupt 修飾子のみを指定する場合は，次のコードが出力されます。

- 割り込み／例外ハンドラとしてのレジスタの退避／復帰処理（__multi_interrupt 修飾子指定の場合はさらに EIPC，EIPSW の退避／復帰処理）を追加
- 割り込み／例外ハンドラの最後に reti 命令を追加

つまり，割り込み／例外ハンドラとしての起動ができる形の関数になりますが，#pragma interrupt 指令で出力される“ハンドラ・アドレスに割り込み／例外ハンドラへの分岐命令を配置する処理”は行われません。

例　割り込み要求名“INTP0”に対する割り込みハンドラを“void intp0_func(void)”とし，direct 指定せず，多重割り込みを許可しない場合の #pragma interrupt 指定は次のようになります。

```
#pragma interrupt INTP0 intp0_func

__interrupt
void intp0_func(void)
{
            :
      （割り込み処理本体）
            :
}
```

次に，割り込みハンドラとして指定できる“ 関数の型 ”について説明します。

(1) 関数の型

マスカブル割り込み，NMI 割り込みのハンドラに関しては，次のようになります。

```
void func(void) 型
```

引数が void 型，戻り値が void 型の関数になります。

ソフトウエア例外処理（トラップ）ハンドラについては，次のようになります。

```
void func(unsigned int) 型
```

引数には割り込み要因レジスタ（ECR）の EICC（例外コード）がセットされます。これらの型で指定しなければ，コンパイル時にエラーとなります。ソフトウエア例外処理関数については，次の項目を参照してください

(2) ソフトウエア例外処理（トラップ処理）ハンドラ

ソフトウエア例外処理（トラップ処理）を使用する場合，V850 マイクロコントローラの仕様上，エントリ・ポイントは，“ TRAP0（0x40 番地）”と“ TRAP1（0x50 番地）”の 2 箇所になります。ソフトウエア例外“ trap 0x00 ～ trap 0x0f ”が入ったときは TRAP0（0x40 番地）へ，“ trap 0x10 ～ trap0x1f ”が入ったときは，TRAP1（0x50 番地）へ分岐します。その際に，“ ソフトウエア例外コード ”として TRAP0 のときは“ 0x40 ～ 0x4f ”の値が，TRAP1 のときは“ 0x50 ～ 0x5f ”の値が割り込み要因レジスタ（ECR）にセットされます。

表 3 － 11 トラップ命令とソフトウエア例外コード

トラップ命令	ソフトウエア例外コード
trap 0x00	0x40
trap 0x01	0x41
trap 0x02	0x42
⋮	⋮
trap 0x0a	0x4a
trap 0x0b	0x4b
⋮	⋮
trap 0x10	0x50
trap 0x11	0x51
trap 0x12	0x52
⋮	⋮
trap 0x1e	0x5e
trap 0x1f	0x5f

TRAP0，TRAP1 に対するソフトウエア例外処理を記述する場合，その関数は，引数を 1 つ持ち，変数の型は“ unsigned int 型 ”になります。その引数には割り込み要因レジスタ（ECR）にセットされた“ ソフトウエア例外コード ”が入ります。TRAP0 のときは“ 0x40 ～ 0x4f ”の値が，TRAP1 のときは“ 0x50 ～ 0x5f ”の値のいずかになります。ハンドラ内では，これらの値によって場合分けした処理を記述することになります。

```
#pragma interrupt TRAP0 trap0_func

__interrupt
void trap0_func(unsigned int codenum)
{
              :
       （例外コード別に場合分けし，その処理を記述）
              :
}
```

3.7.4 割り込み／例外ハンドラの記述時の注意事項

(1) __multi_interrupt 修飾子で指定する“ 多重割り込みハンドラの指定 ”とは,“ 多重割り込み「される」ことを許可する関数を指定 ”ということです。“ 多重割り込みする関数を指定する ”ということではありません。

(2) __multi_interrupt で多重割り込みを許可するハンドラとして定義されても,割り込みハンドラ起動時には,割り込みは許可されていません。そのため，必ず割り込みハンドラの中で割り込み許可命令（__EI() など）を発行し，最後に割り込み禁止命令（__DI() など）を発行してください。最後に割り込み禁止命令を発行しなかった場合，その後のレジスタの復帰部分で割り込みを受け付けてしまう可能性があり，その場合は暴走につながりますので注意が必要です。

(3) #pragma interrupt 指令で，リセット割り込みは指定できません。

```
#pragma interrupt RESET reset_func  /* エラー */
```

上記のように記述をすると，コンパイル時にエラーになります。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。

リセット時の処理は，スタート・アップ・ルーチンに記述するのが一般的であるため，詳細は「第 5 章 スタート・アップ・ルーチン」を参照してください。

(4) #pragma interrupt 指令，__multi_interrupt 修飾子では，多重例外や多重 NMI には対応していません。

多重例外や多重 NMI を行う場合は，必要となるシステム・レジスタ（FEPC，FEPSW など）の退避／復帰を行うコードを追加してください。必要となるシステム・レジスタについては，各デバイスの“ ユーザーズ・マニュアル ハードウエア編 ”を参照してください。

(5) リンク・ディレクティブ・ファイルへの割り込みハンドラ・アドレスの追加記述は，ユーザで行う必要はありません。CA850 が内部的に出力します。

(6) 1 つの割り込み要求名に対し，異なる関数を複数指定することはできません。

(7) 同じ関数に __interrupt 修飾子，__multi_interrupt 修飾子の両方は指定できません。

(8) __interrupt 修飾子，__multi_interrupt 修飾子指定なしで関数を定義したあと，__interrupt 修飾子，__multi_interrupt 修飾子指定ありで関数宣言すると，コンパイル時にエラーになります。

(9) 割り込み／例外ハンドラとして指定された関数はインライン展開できません。#pragma inline 指定しても無視されます。

(10) 割り込み／例外ハンドラとして指定された関数は割り込み禁止となっているため，
#pragma block_interrupt 指定されていても無視されます。

(11) 割り込み／例外ハンドラとして指定された関数は，通常の関数呼び出しで呼び出すことはできません。ただし，別ファイルから呼び出された場合は，コンパイラでチェックできません。

(12) 割り込み／例外ハンドラからアセンブラ命令を呼び出し，「表 3 － 6 レジスタ変数用レジスタ」，および「表 3 － 7 割り込み／例外ハンドラ用のスタック・フレーム」に示されたレジスタを使用する場合，退避／復帰処理を記述する必要があります。
また,SP(r3),GP(r4),TP(r5),EP(r30)を書き換える場合も,退避／復帰処理を記述する必要があります。

(13) #pragma interrupt 指令，__interrupt 修飾子，__multi_interrupt 修飾子機能では，外部割り込みコントローラに対する処理終了通知（EOI コマンド）は発行していません。必要な場合はユーザで実行してください。

(14) 多重割り込みの最後は，EIPC，EIPSW の復帰コードが入るので，割り込み禁止にしてください。

(15) direct 指定をしない場合，ハンドラ・アドレスには“割り込み／例外ハンドラへの分岐命令”が配置されますが，その場合 CA850 はコード効率の面から“jr 命令”を出力しています。
ただし jr 命令で分岐できる範囲には限界があり，jr 命令から± 21 ビット内になります。もし，割り込みハンドラ本体へ jr 命令で分岐できる範囲になかった場合，リンカでエラーになります。その場合は，コンパイル・オプション“-Xj オプション”を指定することにより，jr 命令を jmp 命令に置き換える処置をしてください。

3.7.5 割り込み／例外ハンドラの記述例

割り込み／例外ハンドラの記述例を次に示します。

ただし，割り込み要求名は，デバイスによって異なりますので，各デバイスの“ ユーザーズ・マニュアル ハードウエア編 ” を参照してください。

例 1：ノンマスカブル割り込みの場合

```
#pragma interrupt NMI func1       /* ノンマスカブル割り込み */

__interrupt
void func1(void)
{
       :
}
```

例 2：トラップの場合

```
#pragma interrupt TRAP0 func2     /* トラップ 0 */

__interrupt
void func2(unsigned int num)
{
       switch(num){ /* 例外コード別に場合分け */
              :
       }
}
```

例 3：#pragma interrupt と __interrupt 修飾子を別ファイルとする場合

```
[a. c]
__interrupt                       /* __interrupt 指定 */
void func1(void)
{
       :
}

[b. c]
#pragma interrupt NMI func1       /* 定義より後ろや別ファイルに記述できます */
```

例 4：多重割り込み指定の場合

```
#pragma interrupt INTP0 func1
__multi_interrupt                 /* 多重割り込み関数指定 */
void func1(void)
{
       :
}
```

3.8 インライン展開

CA850 では，関数ごとのインライン展開ができます。ここでは，インライン展開の指定について説明します。

3.8.1 インライン展開とは

インライン展開とは，関数呼び出し部分に関数本体を展開することを言います。これにより，関数呼び出しによるオーバーヘッドが小さくなり，また最適化の可能性が高められることから，実行速度向上を図ることができます。

ただし，インライン展開を行うと，オブジェクト・サイズは増大することになります。

インライン展開したい関数は，#pragma inline で指定します。

```
#pragma inline 関数名 [, 関数名 ...]
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1(){}" という関数であれば "func1" と指定します。また，関数名は “ , ”（カンマ）で区切って複数指定することができます。

```
#pragma inline func1,func2

void func1(){ ... }
void func2(){ ... }

void func(void)
{
       func1();  /* インライン展開対象 */
       func2();  /* インライン展開対象 */
}
```

3.8.2 インライン展開の条件

#pragma inline 指定された関数をインライン展開するためには，最低限次の条件が必要となります。

ただし，“ サイズ優先最適化（-Os）”，“ 実行速度優先最適化（-Ot）”「以外」の最適化を指定していた場合，CA850 の内部処理の関係により，次の条件を満たしていてもインライン展開されない場合があります。

(1) インライン展開を “ する関数 ” と “ される関数 ” を，同一ファイル内に記述する

インライン展開を “ する関数 ” と “ される関数 ”，つまり，“ 関数呼び出し ” と “ 関数定義 ” は “ 同一ファイル内 ” に存在しなければなりません。別の C 言語ソースに書かれてある関数をインライン展開することはできません。この場合，CA850 はエラーも警告メッセージも出力せず，インライン展開指定を無視します。

(2) #pragma inline を “ 関数定義より前 ” に記述する

#pragma inline が，関数定義よりも後ろに記述されていた場合，警告を出力してインライン展開指定を無視します。ただし，関数のプロトタイプ宣言との記述順序は問いません。次に例を示します。

例

インライン展開指定 有効	インライン展開指定 無効

`#pragma inline func1,func2`
 	
`/* プロトタイプ宣言 */`
`void func1();`
`void func2();`
 	
`/* 関数定義 */`
`void func1() { ... }`
`void func2() { ... }`
	`/* プロトタイプ宣言 */`
	`void func1();`
	`void func2();`
	
	`/* 関数定義 */`
	`void func1() { ... }`
	`void func2() { ... }`
	
	`#pragma inline func1,func2`

(3) インライン展開する関数の “ 呼び出し ” と “ 定義 ” の間で，“ 引数の数 ” を同じにする

インライン展開する関数の “ 呼び出し ” と “ 定義 ” の間で “ 引数の数 ” が違う場合，警告メッセージを出力してインライン展開指定を無視します。

(4) インライン展開する関数の “ 呼び出し ” と “ 定義 ” の間で，“ 戻り値の型 ” や “ 引数の型 ” を同じにする

インライン展開する関数の “ 呼び出し ” と “ 定義 ” の間で，“ 戻り値の型 ” や “ 引数の型 ” が異なる場合，警告メッセージを出力してインライン展開指定を無視します。ただし，型変換ができる場合は，次のように変換してインライン展開します。

- 戻り値の型は “ 呼び出し側の型 ”
- 引数は “ 関数定義の型 ”

ただし，“ -ansi オプション ” を指定していたときは，型変換を行わずエラーを出力します。

(5) インライン展開する関数のサイズ，および，使用スタック・サイズは，大きすぎないようにする

インライン展開する関数のサイズ，および使用スタック・サイズが大きい場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。ここでいう “ サイズ ” とは “ 中間言語 ” でのサイズを指し，実際のオブジェクトのサイズとは異なります。CA850 では，これらのサイズの上限を変更することができます。

中間言語における“関数のサイズ”は，次のオプションになります。

```
-Wp,-Nnum
```

中間言語における“関数の使用スタック・サイズ”は，次のオプションになります。

```
-Wp,-Gnum
```

また，次のオプションで中間言語における各関数の“サイズ”,“使用スタック・サイズ”を確認することができます。

```
-Wp,-l
```

このオプションでサイズ指定の目安にすることができます。

(6) インライン展開する関数の引数は“可変個”にしない

引数が“可変個”の関数にインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。

(7) “再帰関数”はインライン展開できない

自分自身を呼び出す“再帰関数”をインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。ただし，関数呼び出しが複数ネストし，そのネストした中に自分自身を呼び出すコードが存在した場合，インライン展開する場合があります。

(8) “割り込みハンドラ”はインライン展開できない

#pragma interrupt，__interrupt，__multi_interrupt 指定で記述された関数は“割り込みハンドラ”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージも出力して，インライン展開指定を無視します。

(9) リアルタイム OS の“タスク”はインライン展開できない

#pragma rtos_task で指定された関数は，リアルタイム OS の“タスク”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージも出力して，インライン展開指定を無視します。

(10) #pragma block_interrupt 指定で関数内を割り込み禁止にすると，インライン展開できない

#pragma block_interrupt で，関数内を割り込み禁止として宣言された関数に対してインライン展開指定した場合，警告メッセージも出力して，インライン展開指定を無視します。

3.8.3 オプションによるインライン展開の制御

“ コンパイラによるインライン展開を抑止したい ” など，インライン展開を制御したい場合があります。そういった場合，オプションによって制御することができます。制御できる内容とそのオプションは，次のとおりです。

ただし，実行速度優先最適化（-Ot）を指定していた場合は「3.8.4 実行速度優先最適化とインライン展開」を参照してください。

(1) 1 回だけ参照される static 関数を，すべてインライン展開したい場合

```
-Wp,-S
```

このオプションを指定した場合は “ 最適化指定 ” や “ #pragma inline の有無 ” に関わらず，1 回だけ参照される static 変数をインライン展開します。

ただし，“ サイズ優先最適化（-Os）” 「以外」の最適化を指定していた場合，CA850 の内部処理の関係により，-Wp,-S オプションを指定してもインライン展開されない場合があります。

(2) すべての関数に対してインライン展開を抑止したい場合

```
-Wp,-no_inline
```

この場合，-Wp,-S 指定や #pragma inline があっても，インライン展開を抑止します。

3.8.4 実行速度優先最適化とインライン展開

CA850 の最適化の 1 つである“ 実行速度優先最適化（-Ot）” をオプション指定した場合，CA850 はインライン展開を最適化手段の 1 つとします。

したがって，実行速度優先最適化（-Ot）が指定されていれば，#pragma inline でインライン展開指定した関数“ 以外 ” でも「3.8.2 インライン展開の条件」をクリアしていれば，CA850 が適切な関数を選択し，インライン展開を行います。

しかし，“ コンパイラによるインライン展開を抑止したい ” など，インライン展開を制御したい場合があります。そういった場合，オプションによって制御することができます。制御できる内容とそのオプションは，次のとおりです。

(1) 実行速度優先最適化（-Ot）を指定しているが，すべての関数に対してインライン展開を抑止したい場合

```
-Wp,-no_inline
```

この場合，-Wp,-S 指定や #pragma inline があっても，インライン展開を抑止します。

(2) 実行速度優先最適化（-Ot）を指定しているが，#pragma inline で指定した関数のみをインライン展開したい場合

```
-Wp,-inline
```

この場合，インライン展開を指定した関数は「3.8.2 インライン展開の条件」をクリアしている必要があります。

3.8.5 オプション指定によるインライン展開動作の違いの例

#pragma inline 指定とオプション指定による“インライン展開動作の違い”の例は，次のようになります。

“-Os（サイズ優先最適化）指定”（-Ot 以外）

```
#pragma inline func0

void func0(){...} /* #pragma inline 指定により,インライン展開の条件が合致すれば展開 */
void func1(){...} /* 展開しない */
void func2(){...} /* 展開しない */
```

“-Ot（実行速度優先最適化）指定”

```
#pragma inline func0

void func0(){...} /* -Ot 指定により，インライン展開の条件が合致すれば展開 */
void func1(){...} /* -Ot 指定により，インライン展開の条件が合致すれば展開 */
void func2(){...} /* -Ot 指定により，インライン展開の条件が合致すれば展開 */
```

“-Ot（実行速度優先最適化）”＋“-Wp,-inline（#pragma inline 指定関数のみをインライン展開）指定”

```
#pragma inline func0

void func0(){...} /* #pragma inline 指定により，インライン展開の条件が合致すれば展開 */
void func1(){...} /* -Wp,-inline 指定により，#pragma inline 指定がないので展開しない */
void func2(){...} /* -Wp,-inline 指定により，#pragma inline 指定がないので展開しない */
```

備考 1 CA850 では，#pragma inline によりインライン展開指定された関数は，静的関数として扱いません。静的関数とするには，明示的に static 指定をする必要があります。

備考 2 デバッグの際，インライン展開をした関数に対して C 言語ソース・レベルでブレークポイントを設定することはできません。

3.9 リアルタイム OS 対応機能

CA850 は，V850 マイクロコントローラ用リアルタイム OS “ RX850，RX850Pro ” を使用したシステムを構築する場合を考慮し，プログラミング記述性向上と，コード削減の機能を備えています。

3.9.1 タスクの記述

リアルタイム OS を使用したアプリケーションは“ タスク ”を処理単位とします。リアルタイム OS は，そのタスク内で発行された “ システム・コール ” や “ 割り込み処理 ” をきっかけとして，タスクのスケジューリングを行います。タスクを切り替えるとき（コンテキストを切り替えるとき）のレジスタの退避／復帰作業は，リアルタイム OS が行うため，一般の関数としてのプロローグ処理／エピローグ処理とは異なります。

したがって，CA850 が，関数呼び出し時に生成するプロローグ処理／エピローグ処理は，タスクでは実行されないことになります。

記述された関数を “ タスク ” とする場合，関数呼び出し時のプロローグ処理／エピローグ処理を削除することにより，コード削減がはかれますが，C 言語の記述上，“ 一般の関数 ” と “ タスク ” は区別がつきません。そこでCA850では,関数を“ リアルタイムOSのタスク ”と認識させるために,次の#pragma指令を用意しています。

```
#pragma rtos_task ［関数名］
```

これにより，“ 関数名 ” で指定された関数を，リアルタイム OS のタスクとして認識させることができます。“ 関数名 ” は，C 言語記述の関数名を指定します。例として，“ void func1(int inicode){} ” という関数をタスクとする場合は，次のように記述します。

例

```
#pragma rtos_task func1
```

また，“ 関数名 ” は省略することもできます。省略した場合，そのファイル内において “ #pragma rtos_task ” 記述以降の関数をタスクと認識します。

#pragma rtos_task を指定すると，具体的には次のような効果が得られます。

(1) 通常の関数で出力される “ プロローグ／エピローグ処理 ” を行いません。具体的には次のようなコードを出力しません。

(a) レジスタ変数用レジスタの退避／復帰

(b) リンク・ポインタ（lp）の退避／復帰

(c) 戻り先からへのジャンプ

(2) システム・コール “ ext_tsk ” を，定義済みの関数として使用することができます。

特にアプリケーション内でプロトタイプ宣言しなくても，このシステム・コールを使用することができます。#pragma rtos_task の記述以降であれば，タスク指定した関数「以外」でも，同様に呼び出すことができます。このシステム・コールを呼び出したとき，コード・サイズ削減のために “ jr 命令 ” を使用したコードを出力します。システム・コール “ ext_tsk ” 本体が，jr 命令で分岐可能な範囲にない場合は，リンカ（ld850）でエラーになります。この場合は，次の処置が必要になります。

(a) リンク・ディレクティブでメモリ配置を変更する
(b) アセンブリ言語ソースで jmp 命令による分岐に切り替える
(c) far jump 指定する

また，#pragma rtos_task 指定した場合，次のような注意事項があります。

- 関数と同じように，タスクを呼び出すことはできません。
  ただし，別のファイルで呼び出された場合はチェックされません。また，関数として呼び出すことができないため，インライン展開することができません。
  したがって，#pragma rtos_task 指定された関数に，#pragma inline 指定しても，#pragma inline 指定は無視されます。
- “ #pragma rtos_task 関数名 ”を，同じファイル内の関数定義よりも後ろに記述した場合，エラーとなります。
  また，“ #pragma rtos_task 関数名 ” を記述したあと，そのファイル内に関数定義を記述しない場合は，その関数に対する #pragma 指令は無視されます。
- #pragma rtos_task 指定された関数は，通常の割り込み／例外ハンドラ（「3.7 割り込み／例外処理ハンドラ」参照）として指定することはできません。

なお，リアルタイム OS の機能については，各リアルタイム OS のユーザーズ・マニュアルを参照してください。

3.10 組み込み関数

CA850 では，アセンブラ命令の一部を “ 組み込み関数 ”として C 言語ソースに記述することができます。ただし，“ アセンブラ命令そのもの ”を記述するのではなく，CA850 で用意した関数の形式で記述します。

表 3 － 12 に，関数として記述できる命令を示します。

表 3 － 12 CA850 組み込み関数一覧

アセンブラ命令	機能	組み込み関数
di ei	割り込み制御（di ／ ei）	__DI() __EI()
nop	nop	__nop()
halt	halt	__halt()
satadd	飽和加算（satadd）	long a, b; long __satadd(a, b)
satsub	飽和減算（satsub）	long a, b; long __satsub(a, b)
bsh	ハーフワード・データのバイト・スワップ（bsh）【V850E】	long a; long __bsh(a)
bsw	ワード・データのバイト・スワップ（bsw）【V850E】	long a; long __bsw(a)
hsw	ワード・データのハーフワード・スワップ（hsw）【V850E】	long a; long __hsw(a)
sxb	バイト・データの符号拡張（sxb）【V850E】	char a; long __sxb(a)
sxh	ハーフワード・データの符号拡張（sxh）【V850E】	short a; long __sxh(a)
mul	mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令【V850E】	long a; long b; long __mul32(a, b)
mulu	mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する 命令【V850E】	unsigned long a, b; unsigned long __mul32u(a, b)
sasf	論理左シフト付きフラグ条件の設定（sasf）【V850E】	long a; unsigned int b; long __sasf(a, b)

注 1 【V850E】マークは，V850Ex コア専用であることを示します。

注 2 組み込み関数と同名の関数を定義して使用することはできません。
同名の関数を呼び出そうとしても，コンパイラが用意している組み込み関数処理を優先します。

3.10.1 割り込み制御（di ／ ei）

割り込み制御命令（di ／ ei）の記述例を示します。

例

```
void func(void)
{
              :
       __DI();       /* 割り込みを禁止して行いたい処理 */
       __EI();
              :
}
```

例の出力コード

```
_func:
       -- プロローグ・コード）

       di
       -- 割り込みを禁止して行いたい処理
       ei
              :
       -- エピローグ・コード
       jmp [lp]
```

3.10.2 nop

nop 命令の記述例を示します。

例

```
void func(void)
{
              :
       __nop();
              :
}
```

例の出力コード

```
_func:
         :
       nop
         :
```

3.10.3 halt

halt 命令の記述例を示します。

例

```
void func(void)
{
        :
       __halt();
}
```

例の出力コード

```
_func:
        :
       halt
```

3.10.4 飽和加算（satadd）

飽和加算命令（satadd）の記述例を示します。

例

```
void func(void)
{
       long   a, b, c;
              :
       c = __satadd(a, b);        /* a と b の飽和演算結果を c に格納する */
              :
}
```

例の出力コード

```
_func:
                     :
       ld.w   -4 +.A2 [sp], r10    -- 変数 a をロード
       ld.w   -8 +.A2 [sp], r11    -- 変数 b をロード
       satadd r11, r10             -- 飽和減算（ a + b ）
       st.w   r10, -12 +.A2 [sp]   -- 飽和演算結果を変数 c にストア
                     :
```

3.10.5 飽和減算（satsub）

飽和減算命令（satsub）の記述例を示します。

例

```
void func(void)
{
       long   a, b, c;
              :
       c = __satsub(a, b);        /* a と b の飽和演算結果を c に格納する (c = a - b) */
              :
}
```

例の出力コード

```
_func:
                     :
       ld.w   -4 +.A2 [sp], r10   -- 変数 a をロード
       ld.w   -8 +.A2 [sp], r11   -- 変数 b をロード
       satsub r11, r10            -- 飽和減算（ a - b ）
       st.w   r10, -12 +.A2 [sp]  -- 飽和演算結果を変数 c にストア
                     :
```

3.10.6 ハーフワード・データのバイト・スワップ（bsh）【V850E】

ハーフワード・データのバイト・スワップ命令（bsh）の記述例を示します。

例

```
void func(void)
{
       long   a, b;


              :
       b = __bsh(a);        /* a のハーフワード・データのバイト・スワップを行い，b に格納する */
              :
}
```

例の出力コード

```
_func:
               :
       ld.w   -4+.A2 [sp], r10    -- 変数 a をロード
       bsh    r10, r10            -- ハーフワード・データのバイト・スワップ
       st.w   r10, -8+.A2 [sp]    -- ハーフワード・データのバイト・スワップ結果を
                                  -- 変数 b にストア
               :
```

3.10.7 ワード・データのバイト・スワップ（bsw）【V850E】

ワード・データのバイト・スワップ命令（bsw）の記述例を示します。

例

```
void func(void)
{
       long   a, b;
            :
       b = __bsw(a);  /* aのワード・データのバイト・スワップを行い，その結果をbに格納する */
            :
}
```

例の出力コード

```
_func:
             :
       ld.w   -8+.A2 [sp], r10     -- 変数aをロード
       bsw    r10, r10             -- ワード・データのバイト・スワップ
       st.w   r10, -12+.A2 [sp]    -- 変数bにストア
             :
```

3.10.8 ワード・データのハーフワード・スワップ（hsw）【V850E】

ワード・データのハーフワード・スワップ命令（hsw）の記述例を示します。

例

```
void func(void)
{
       long   a, b;
             :
       b = __hsw(a);   /* aのワード・データのハーフワード・スワップを行い，その結果をbに格納する */
             :
}
```

例の出力コード

```
_func:
             :
       ld.w   -8+.A2 [sp], r10     -- 変数aをロード
       hsw    r10, r10             -- ワード・データのハーフワード・スワップ
       st.w   r10, -12+.A2 [sp]    -- 変数bにストア
             :
```

3.10.9 バイト・データの符号拡張（sxb）【V850E】

バイト・データの符号拡張命令（sxb）の記述例を示します。

例

```
void func(void)
{
       char   a;
       long   b;
              :
       b = __sxb(a);  /* a のバイト・データの符号拡張を行い，その結果を b に格納する */
              :
}
```

例の出力コード

```
_func:
                :
       ld.w   -8+.A2 [sp], r10     -- 変数 a をロード
       sxb    r10, r10             -- バイト・データの符号拡張
       st.w   r10, -12+.A2 [sp]    -- 変数 b にストア
                :
```

3.10.10 ハーフワード・データの符号拡張（sxh）【V850E】

ハーフワード・データの符号拡張命令（sxh）の記述例を示します。

例

```
void func(void)
{
       short  a;
       long   b;
              :
       b = __sxh(a);  /* a のハーフワード・データの符号拡張を行い，その結果を b に格納する */
              :
}
```

例の出力コード

```
_func:
              :
       ld.w   -8+.A2 [sp], r10     -- 変数 a をロード
       sxh    r10                  -- ハーフワード・データの符号拡張
       st.w   r10, -12+.A2 [sp]    -- 変数 b にストア
              :
```

3.10.11 mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令【V850E】

mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令の記述例を示します。

例

```
void func(void)
{
       long   a, b, c;
              :
       c = __mul32(a, b);          /* a × b の結果の上位 32 ビットを c に格納する */
              :
}
```

例の出力コード

```
_func:
              :
       ld.w   -4+.A2 [sp], r10     -- 変数 a をロード
       ld.w   -8+.A2 [sp], r11     -- 変数 b をロード
       mul    r11, r10, r12        --  a × b
       st.w   r12, -12+.A2 [sp]    -- 変数 c にストア
              :
```

3.10.12 mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する 命令【V850E】

mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令の記述例を示します。

例

```
void func(void)
{
       unsigned long a, b, c;
              :
       c = __mul32u(a, b);         /* a ×b の結果の上位 32 ビットを c に格納する */
              :
}
```

例の出力コード

```
_func:
              :
       ld.w   -4+.A2 [sp], r10     -- 変数 a をロード
       ld.w   -8+.A2 [sp], r11     -- 変数 b をロード
       mulu   r11, r10, r12        --  a × b
       st.w   r12, -12+.A2 [sp]    -- 変数 c にストア
              :
```

3.10.13 論理左シフト付きフラグ条件の設定（sasf）【V850E】

例 1 に第二引数に条件式を書く場合の論理左シフト付きフラグ条件の設定命令（sasf）の記述例を示します。

例 2 に第二引数に変数を書く場合の論理左シフト付きフラグ条件の設定命令（sasf）の記述例を示します。

例 1 第二引数に条件式を書く場合

```
void func(void)
{
       unsigned long a, b, c;
             :
       c = __sasf(c, a == b);  /* a == bが真ならばcを左1ビット論理シフトして1加える */
                               /* 偽ならばcを左1ビット論理シフトする */
                               /* この結果をcに格納する */
             :
}
```

例 1 の出力コード

```
_func:
             :
       ld.w     -4+.A2 [sp], r10  -- 変数aをロード
       ld.w     -8+.A2 [sp], r11  -- 変数bをロード
       cmp     r11, r10           -- 変数a, bを比較する
       ld.w     -12+.A6 [sp], r12 -- 変数cをロード
       sasf     0x2, r12           --  a  == b が真ならばcを左1ビット論理シフトして
                                   -- 1を加える。偽ならばcを左1ビット論理シフト
       st.w     r12, -12+.A2 [sp] -- 変数cにストア
             :
```

例 2 第二引数に変数を書く場合

```
void func(void)
{
       unsigned long a, b;
              :
       b = __sasf(b, a);   /* a が 0 でなければ b を左 1 ビット論理シフトして 1 加える */
                           /* a が 0 以外ならば b を左 1 ビット論理シフトする */
                           /* この結果を b に格納する */
              :
}
```

例 2 の出力コード

```
_func:
              :
       ld.w   -4+.A2 [sp], r10     -- 変数 a をロード
       cmp    r0, r10              -- 変数 a と 0 を比較
       ld.w   -8+.A2 [sp], r11     -- 変数 b をロード
       sasf   0xa, r11             --  a が 0 でなければ b を左 1 ビット論理シフトして 1
                                   -- を加える。a が 0 ならば b を左 1 ビット論理シフト
       st.w   r11, -8+.A2 [sp]     -- 変数 b にストア
              :
```

3.11 構造体パッキング

CA850 は，構造体メンバのアライメントを C 言語レベルで指定できます。この機能は，-Xpack オプションと同等ですが，構造体パッキング指令は C 言語ソース内の任意な位置でアライメント値を指定できます。

注 構造体をパッキングすると，データ領域を小さくできますが，プログラム・サイズは増え，実行速度も低下します。

3.11.1 構造体パッキングの形式

構造体パッキング機能は次の形式で指定します。

```
#pragma pack([1248])
```

#pragma pack は，この指令が出現した時点で，構造体メンバのアライメント値に変更します。この数値をパッキング値と呼び，指定できる数値は，1，2，4，または 8 です。また，パッキング値を指定しない場合，デフォルトのアライメント 8 [注]になります。なお，この指令は出現した時点で有効となるので C 言語ソース内に複数記述することができます。

利用例

```
#pragma pack(1)      /* 構造体メンバを 1 バイトのアライメントで整列します */
struct TAG {
       char   c;
       int    i;
       short  s;
};
```

注 本バージョンではアライメント値“4”と“8”は同じになります。

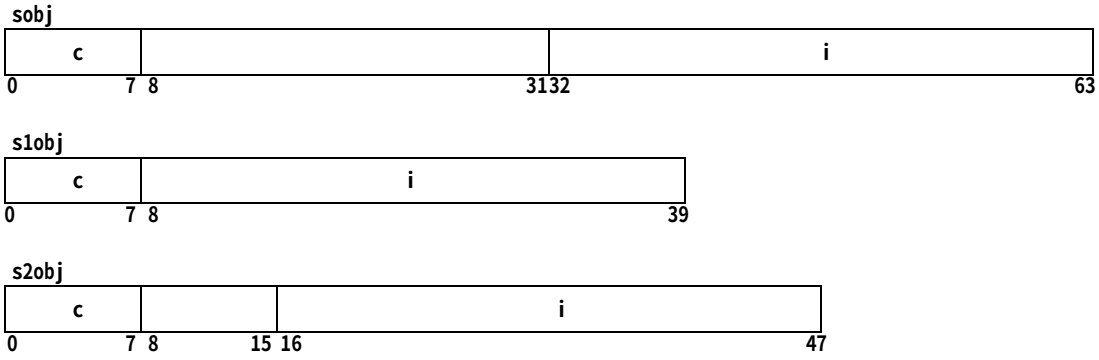
3.11.2 構造体パッキングのルール

構造体のメンバは，構造体のパッキング値とメンバの持つアライメント値の小さい方の値の整列条件を満たす形で並べられます。

たとえば，構造体のパッキング値が 2 のときメンバの形が int 型ならば 2 バイトの整列条件を満たす形で並べられます。

例

```
struct S {
       char   c;      /*1 バイトの整列条件を満たす */
       int    i;      /*4 バイトの整列条件を満たす */
};
#pragma pack(1)
struct S1 {
       char   c;      /*1 バイトの整列条件を満たす */
       int    i;      /*1 バイトの整列条件を満たす */
};
#pragma pack(2)
struct S2 {
       char   c;      /*1 バイトの整列条件を満たす */
       int    i;      /*2 バイトの整列条件を満たす */
};
struct S      sobj;          /* サイズ 8 バイト */
struct S1     s1obj;         /* サイズ 5 バイト */
struct S2     s2obj;         /* サイズ 6 バイト */
```



3.11.3 共用体

共用体をパッキングの対象として構造体パッキングと同様に扱います。

例 1

```
union U {
       struct S {
              char   c;
              int    i;
       } sobj;
};
#pragma pack(1)
union U1 {
       struct S1 {
              char   c;
              int    i;
       } s1obj;
};
#pragma pack(2)
union U2 {
       struct S2 {
              char   c;
              int    i;
       } s2obj;
};
union  U            uobj;        /* サイズ 8 バイト */
union  U1           u1obj;       /* サイズ 5 バイト */
union  U2           u2obj;       /* サイズ 6 バイト */
```

例 2

```
union U {
       int    7:i;
};
#pragma pack(1)
union U1 {
       int    7:i;
};
#pragma pack(2)
union U2 {
       int    7:i;
};
union  U      uobj;         /* サイズ 4 バイト */
union  U1     u1obj;        /* サイズ 1 バイト */
union  U2     u2obj;        /* サイズ 2 バイト */
```

3.11.4 ビット・フィールド

ビット・フィールド要素の領域は次のように割り当てます。

(1) 構造体のパッキング値がメンバの型の整列条件値と等しいあるいは大きい場合

構造体パッキング機能を利用しなかったときと同じように割り当てます。つまり，続けて割り当てるとその領域が要素の型の整列条件を満たす境界を越えてしまう場合，その整列条件を満たしている領域から割り当てます。

(2) 構造体のパッキング値が要素の型の整列条件値より小さい場合

(a) 続けて割り当てるとその領域を含むバイト数が要素の型よりも大きくなる場合

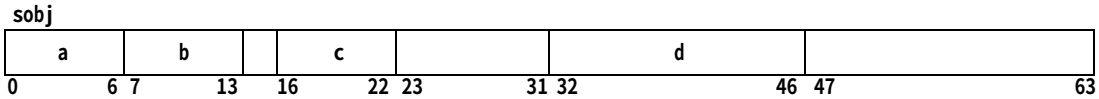
構造体のパッキング値の整列条件を満たす形で割り当てます。

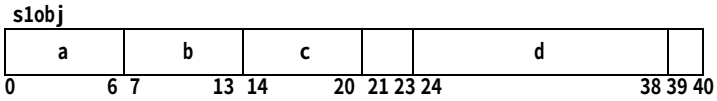
(b) それ以外の場合

続けて割り当てます。

例

```
struct S {
        short  a : 7;          /*0 ～ 6 ビット目 */
        short  b : 7;          /*7 ～ 13 ビット目 */
        short  c : 7;          /*16 ～ 22 ビット目（2 バイト境界に整列）*/
        short  d : 7;          /*32 ～ 46 ビット目（2 バイト境界に整列）*/
} sobj;
#pragma pack(1)
struct S1 {
        short  a : 7;          /*0 ～ 6 ビット目 */
        short  b : 7;          /*7 ～ 13 ビット目 */
        short  c : 7;          /*14 ～ 20 ビット目 */
        short  d : 15;         /*24 ～ 38 ビット目（バイト境界に整列）*/
} s1obj;
```





3.11.5 構造体オブジェクトの先頭の整列条件

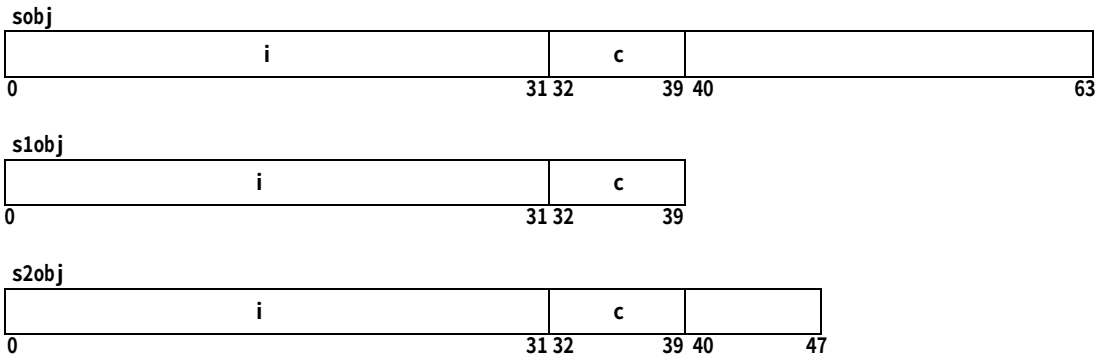
構造体オブジェクトの先頭の整列条件は，構造パッキング機能を利用しなかったときと同じです。

3.11.6 構造体オブジェクトのサイズ

構造体のサイズが構造体の整列条件の値と構造体のパッキング値の小さい方の値の倍数になるようにパッキングを行います。オブジェクトの先頭の整列条件は構造パッキング機能を利用しなかったときと同じです。

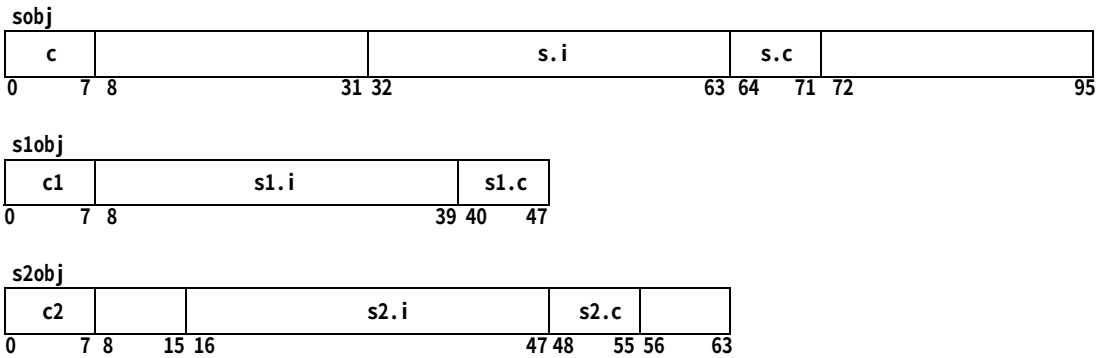
例 1

```
struct S {
       int    i;
       char   c;
};
#pragma pack(1)
struct S1 {
       int    i;
       char   c;
};
struct S      sobj;         /* サイズ 8 バイト */
struct S1     s1obj;        /* サイズ 5 バイト */
struct S2     s2obj;        /* サイズ 6 バイト */
```



例 2

```
struct S {
       int    i;
       char   c;
};
struct T {
       char   c;
       struct S s;
};
#pragma pack(1)
struct S1 {
       int    i;
       char   c;
};
struct T1 {
       char   c;
       struct S1 s1;
};
#pragma pack(2)
struct S2 {
       int    i;
       char   c;
};
struct T2 {
       char   c;
       struct S2 s2;
};
struct T      tobj;         /* サイズ 12 バイト */
struct T1     t1obj;        /* サイズ 6 バイト */
struct T2     t2obj;        /* サイズ 8 バイト */
```

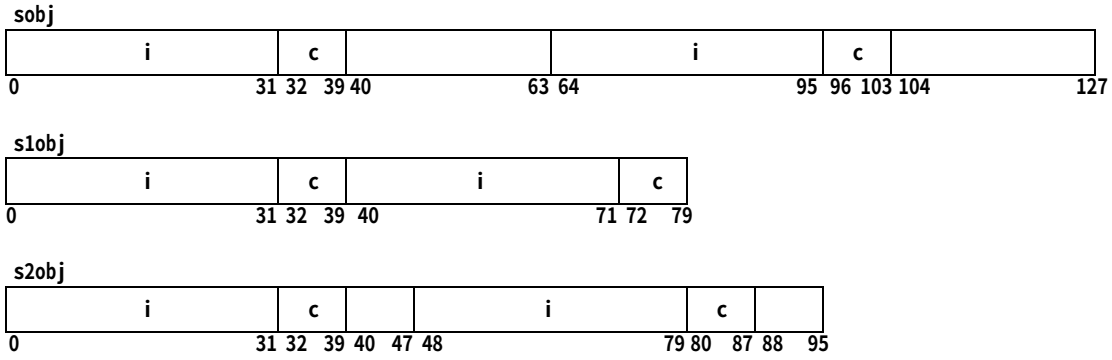


3.11.7 構造体配列のサイズ

構造体オブジェクトの配列のサイズは要素である構造体オブジェクトのサイズに要素数を乗算した値です。

例

```
struct S {
       int    i;
       char   c;
};
#pragma pack(1)
struct S1 {
       int    i;
       char   c;
};
#pragma pack(2)
struct S2 {
       int    i;
       char   c;
};
struct S      sobj [2];          /* サイズ 16 バイト */
struct S1     s1obj [2];         /* サイズ 10 バイト */
struct S2     s2obj [2];         /* サイズ 12 バイト */
```

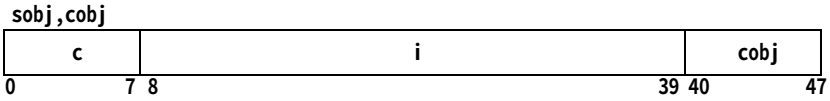


3.11.8 オブジェクト間の領域

たとえば，次のソース・プログラムでは，sobj.c，sobj.i，cobj が隙間なく続いて配置される可能性があります（sobj，cobj の配置順は保証されません）。

例

```
#pragma pack(1)
struct S {
       char   c;
       int    i;
} sobj;
char   cobj;
```



3.11.9 構造体パッキング機能の注意点

(1) -Xpack オプションと #pragma pack 指令の同時指定について

C 言語ソース中に #pragma pack 指令で構造体パッキング指定がある時に -Xpack オプションを指定した場合，最初の #pragma pack 指令が出現するまではオプション指定値がすべての構造体に適用されます。それ以降は #pragma pack 指令の値が適用されます。

ただし，#pragma pack 指令の出現後でも指定がデフォルトになった部分は，オプション指定値が適用されます。

例（-Xpack=2 を指定した場合）

```
struct S2 { ... };  /* オプションでパッキング値 2 を指定している */
                    /* オプション -Xpack=2 が有効：パッキング値 2 */

#pragma pack(1)     /* #pragma 指令でパッキング 1 を指定している */

struct S1 { ... };  /* pragma pack(1) が有効：パッキング値 1 */

#pragma pack()      /* #pragma 指令でパッキング値にデフォルトを指定している */

struct S2_2 { ... };/* オプション -Xpack=2 が有効：パッキング値 2 */
```

(2) 制限事項

V850 マイクロコントローラ，V850Ex 製品ミス・アライン・アクセス禁止の設定の CPU をご使用の場合，次の制限があります。

(a) 構造体メンバのアドレスでのアクセスが正しく行えません。

次のように構造体メンバのアドレスを取得して，そのアドレスでのアクセスは，デバイスのデータ・アライメントに従い，アドレスをマスクしてアクセスされるため，データの消失や切り捨てが生じます。

例

```
struct test {
       char   c;            /* offset 0 */
       int    i;            /* offset 1-4 */
} test;

int    *ip ,i;

void func(void)
{
       i = *ip;             /* マスクされたアドレスでアクセスされる */
}

void func2(void)
{
       ip = &(test.i);      /* 構造体メンバのアドレス取得 */
}
```

(b) ビット・フィールドへのアクセスは，そのメンバの型で読み込むためデータがない領域もアクセスします。

次のようにビット・フィールドの幅がメンバの型以下の場合，メンバの型で読み込むのでオブジェクトの外部にアクセスします。実行上，通常は問題ありませんが，I/O などがマップされていた場合に不正なアクセスとなる場合があります。

例

```
struct S {
       int    x : 21;
} sobj;              /* 3byte */

sobj.x = 1
```

3.12 2 進定数

CA850 では，整数定数を 2 進数で記述することができます。

2 進定数は，“ 0b ”, または“ 0B ”と , その後ろに続く 1 個以上の“ 0 ”, または“ 1 ”の数字の並びで構成されます。

例

```
0b00010110111110101011111010010111
```

注 -ansi オプション指定時には、2 進定数は使用できません。

第 4 章　プログラムの呼び出し

この章では，CA850 におけるプログラム呼び出し時の引数などの扱い方について説明します。

4.1　C 言語関数間の呼び出し

- 通常の関数呼び出し
  → jarl 命令
- 関数を指すポインタを用いた関数呼び出し（および関数呼び出しからの復帰）
  → jmp 命令（dispose 命令【V850E】）

C 言語関数から C 言語関数を呼び出す際，4 ワード分の引数を“ 引数用レジスタ（r6 ～ r9）”格納し，4 ワードを越えた引数は，呼び出し側の関数のスタック・フレームに格納します。その後，呼び出された関数へ移行（ジャンプ）し，呼び出されるときに格納された“ 引数用レジスタの値 ”を，呼び出された関数側で，呼び出された関数側のスタック・フレームに格納します。

スタック・フレームは，関数のプロローグ・コード，つまり，関数が呼び出されてから，関数本体のコードを実行する前に実行されるコード（図 4 － 3，図 4 － 5 で示す（4）～（7）の処理がプロローグ・コードになります）において，スタック・ポインタ（sp）を，必要サイズ分だけずらすことによって生成されます。また，関数のエピローグ・コード，つまり，関数本体のコードを実行し終わり，呼び出し側の関数に戻るまでに実行されるコード（図 4 － 3，図 4 － 5 で示す（i）～（iv）の処理がエピローグ・コードになります）において，スタック・ポインタ（sp）を戻すことにより，スタック・フレームは消滅します。

4.1.1　スタック・フレーム／関数呼び出し

スタック・フレームの形状，および関数呼び出し時のスタック・フレームの生成／消滅状態について説明します。

(1) スタック・フレームの形状

CA850 では，スタック・フレームは引数の条件によって，引数レジスタ領域を“ スタックの先頭 ”か“ スタックの中央 ”のどちらかに確保します。引数の条件は次のようになります。

(a) 引数レジスタ領域が，スタックの先頭に配置される場合
4 ワード分の引数領域を越えて，連続アクセスする必要がある場合で，次の二通りが該当します。

- 引数が可変個引数の場合
- 引数が構造体実体で，その領域が 4 ワード境界をまたぐ場合

(b) 引数レジスタ領域が，スタックの中央に配置される場合
この場合は (a) の条件以外の場合になります。

引数レジスタ領域を“スタックの先頭”に持つ場合のスタック・フレームを図 4 － 1,“スタックの中央”に持つ場合のスタック・フレームを図 4 － 2 に示します

図 4 － 1 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）

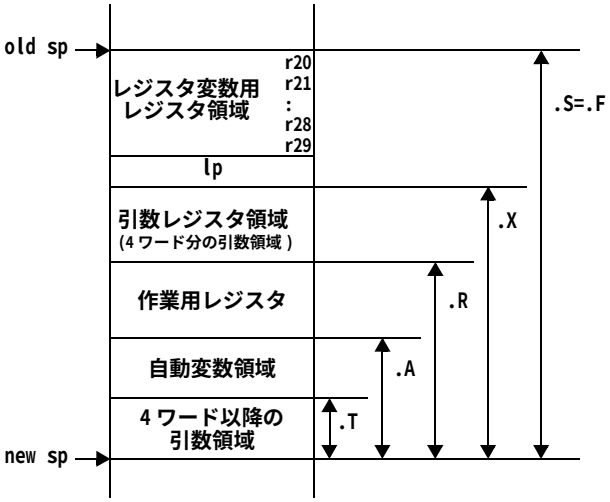
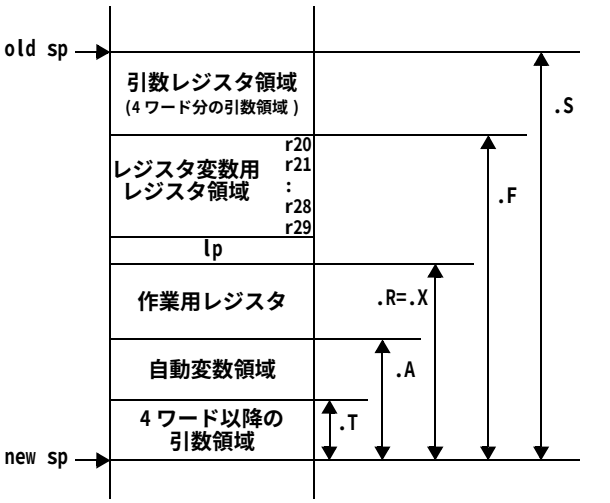


図 4 － 2 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）



図中にある“.S，.F，.X，.R，.A，.T”は，コンパイラが内部的に出力する関数用マクロです。それぞれ，使用用途が決まっており，次表のようになります。

表 4 － 1 関数用マクロの意味

マクロ名	意味
.S	スタック・サイズ
.F	スタック・サイズ - 引数レジスタ領域のサイズ（スタックの先頭にある場合）
.X	引数レジスタ領域のサイズ（スタックの中央にある場合） + .R
.R	作業用レジスタ領域のサイズ + .A + .T
.A	自動変数領域のサイズ + .T
.T	呼び出す関数の 4 ワード以降の引数領域のサイズ
.P	常に 0（コード生成用のマクロ）注

注 .P は常に 0 のため，図 4 － 1，図 4 － 2 には記述してありません。

これらのマクロを使用して，スタック領域にアクセスすることになりますが，具体的なアクセス方法（出力するアクセス・コード）は次表のようになります。

表 4 － 2 スタック領域のアクセス方法

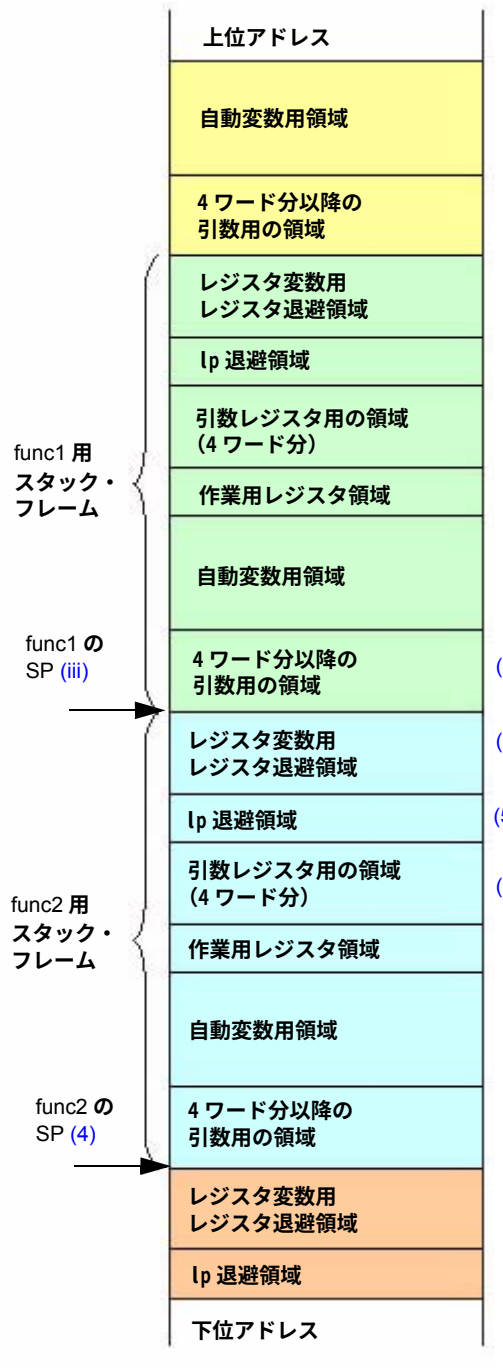
スタック領域	アクセス方法（ディスプレースメント [sp]）
レジスタ変数用レジスタ領域（lp も含む）	-offset+.Fxx[sp]
作業用レジスタ領域	-offset+.Rxx[sp]
自動変数領域	-offset+.Axx[sp]
4 ワード以降の引数領域	offset+.Pxx[sp]
引数レジスタ領域（スタックの先頭にある場合）	offset+.Fxx[sp]
引数レジスタ領域（スタックの中央にある場合）	offset+.Rxx[sp]

表中で “ offset ” は正の整数で，各領域中のオフセットを意味します。また，マクロの後に書かれている “ xx ” は正の整数で，関数のフレーム番号を示します。

(2) 関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの中央”にくる場合）

“引数レジスタ領域がスタックの中央にくる場合”の，関数呼び出し時のスタック・フレームの生成と消滅について説明します。ほとんどの関数呼び出しは，このケースになります。図 4 － 3 に，関数 func1( ) から関数 func2( ) を呼び出し，その後関数 func1( ) へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 4 － 3 スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）



関数 func2( ) を呼び出すときの関数 func1( ) 側の処理

(1) 引数を引数用レジスタに格納
呼び出す func2 の引数を r6 ～ r9 へ格納

(2) 4 ワード分以降の引数をスタックへ格納
r6 ～ r9 の 4 つに入りきらない引数を格納。引数が 5 つ以上の場合はこの処理を行う対象になります。

(3) jarl 命令へ func2( ) 関数へ分岐

関数func1( ) を呼び出されたときの関数func2( ) 側の処理

(4) SP をずらす
関数 func2 が使用するのスタックへ移動

(5) lp を退避
関数 func1( ) の戻りアドレスを格納

(6) レジスタ変数用レジスタを退避
関数 func2( ) でもレジスタ変数用レジスタを使用するとき，関数 func1 で使用していたレジスタ値を保護する必要があるために退避

(7) 引数を“引数レジスタ用領域”に格納
r6 ～ r9 の値を格納
関数 func2( ) から他の関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

[注]V850Ex の場合 (4)，(5)，(6) の処理は prepare 命令で実行できるため，CA850 は prepare 命令を出力します。

関数 func2( ) から関数 func1( ) へ戻るときの関数 func2( ) 側の処理

(i) レジスタ変数用レジスタの復帰
保護していた“関数 func1( ) のレジスタ変数用レジスタの値”をレジスタへ戻す

(ii) lp の復帰
関数 func1( ) の戻りアドレスを復帰

(iii) SP を戻す
関数 func1( ) が使用するのスタックへ移動

(iv) jmp [lp] で戻る

[注]V850Ex の場合 (i)，(ii)，(iii)，(iv)，の処理 dispose 命令で実行できるため，CA850 は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

(a) 呼び出す側 ～関数 func1( ) ～

- 呼び出す func2( ) の引数が 4 ワードを越えていた時，越えた分の引数の値

(b) 呼び出される側 ～関数 func2( ) ～

- 引数用レジスタに入れられた引数の受け渡し
  （引数用レジスタに入れるのは，呼び出す側（関数 func1( )））
- 呼び出した側（関数 func1( )）のリンク・ポインタ（lp）（＝関数 func1( ) の戻りアドレス）の退避
- “ レジスタ変数用レジスタ ” の退避
  “ レジスタ変数用レジスタ ” として割り当てられているのは，次のようになります。

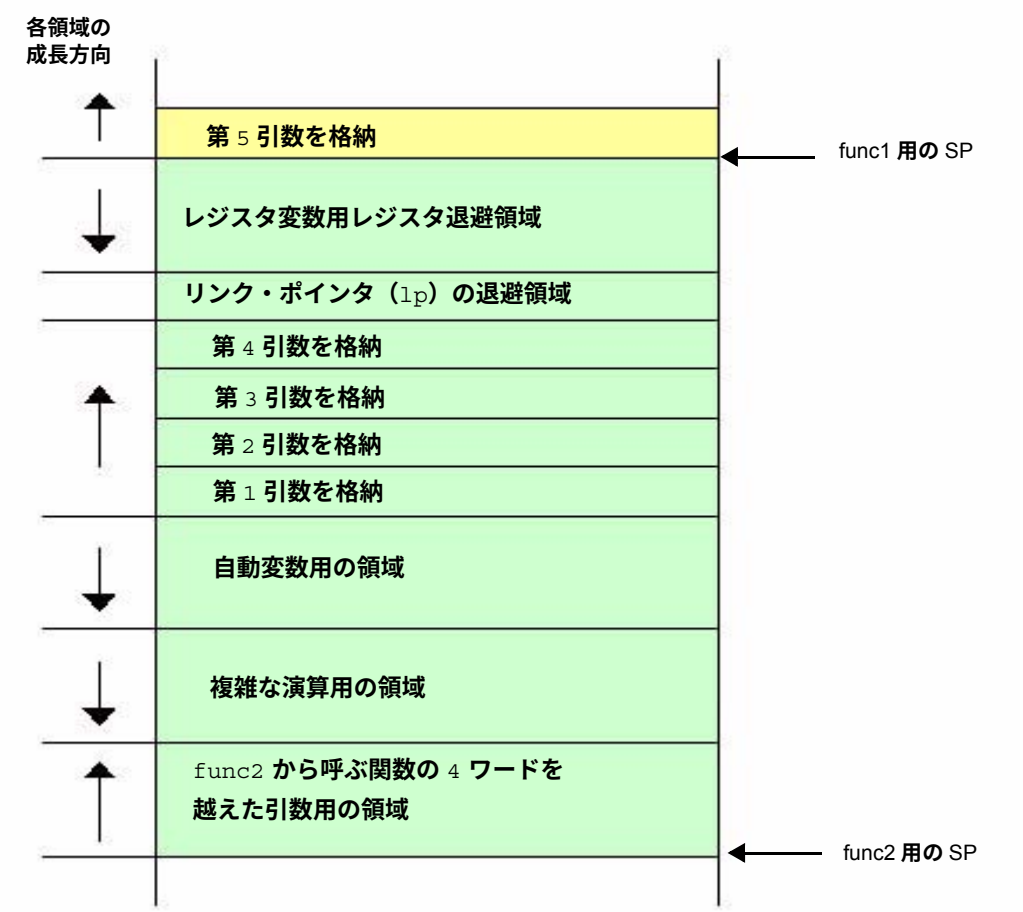
  22 レジスタ・モードのとき :“ r25，r26，r27，r28，r29 ”
  26 レジスタ・モードのとき :“ r23，r24，r25，r26，r27，r28，r29 ”
  32 レジスタ・モードのとき :“ r20，r21，r22，r23，r24，r25，r26，r27，r28，r29 ”

  このうち使用しているものを退避します。
- “ 自動変数用 ” の領域
- 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保
  この領域は，図 5-3 には示していませんが，使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます。

  また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2( ) の引数が 5 個あるとします）

図 4 － 4 スタック・フレームの各領域のスタック成長方向



次に“ C 言語関数から C 言語関数を呼び出したソース ”と“ それをコンパイルしたときのアセンブリ言語ソース ”の具体例を示します。

例

```
void func1(void)
{
       int    a, b, c, d, e;
       func2(a, b, c, d, e);
              :
}

int func2(int a, int b, int c, int d, int e)
{
       register int  i;
              :
       return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

【V850】

```
_func1:
        jbr     .L3
.L4:
        ld.w    -8+.A3 [sp], r6
        ld.w    -12+.A3 [sp], r7
        ld.w    -16+.A3 [sp], r8 --- (1)
        ld.w    -20+.A3[sp], r9
        ld.w    -24+.A3[sp], r10
        st.w    r10, [sp]          --- (2)
        jarl    _func2, lp         --- (3)
                :
        -- main に対するエピローグ
        --  (ii) から (iv) の処理
.L3:
        -- main に対するプロローグ
        --(4), (5) の処理
                :
jbr     .L4

_func2:
        jbr     .L5
.L6:
        st.w    r6, .R2[sp]
        st.w    r7, 4+.R2[sp]
        st.w    r8, 8+.R2[sp]      --- (7)
        st.w    r9, 12+.R2[sp]
        st.w    r29, -4+.A2[sp]
                :
        jbr     .L2
.L2:
        ld.w    -4+.A2[sp], r10
        ld.w    -4+.F2[sp], r29 --- (i)
        ld.w    -8+.F2[sp], lp  --- (ii)
        add     .F2, sp         --- (iii)
        jmp     [lp]            --- (iv)

.L5:
        add     -.F2, sp           --- (4)
        st.w    lp, -8+.F2[sp]     --- (5)
        st.w    r29, -4+.F2[sp]    --- (6)
        jbr     .L6
```

【V850E】

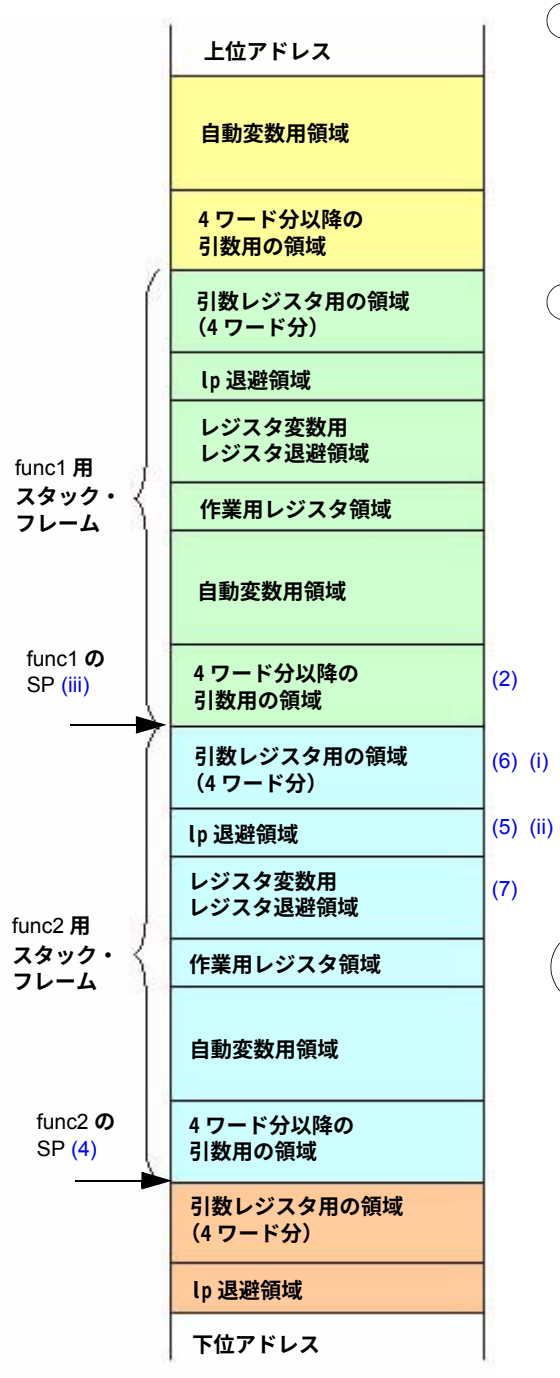
```
_func1:
        jbr     .L3
.L4:
        ld.w    -8+.A3[sp], r6
        ld.w    -12+.A3[sp], r7
        ld.w    -16+.A3[sp], r8 --- (1)
        ld.w    -20+.A3[sp], r9
        ld.w    -24+.A3[sp], r10
        st.w    r10, [sp]          --- (2)
        jarl    _func2, lp         --- (3)
                :
        --main に対するエピローグ
        --(ii) から (iv) の処理
.L3:
        --main に対するプロローグ
        --(4), (5) の処理
                :
        jbr     .L4

_func2:
        jbr     .L5
.L6:
        st.w    r6, .R2[sp]
        st.w    r7, 4+.R2[sp]
        st.w    r8, 8+.R2[sp]      --- (7)
        st.w    r9, 12+.R2[sp]
        st.w    r29, -4+.A2[sp]
                :
        jbr     .L2
.L2:
        ld.w    -4+.A2[sp], r10
        dispose .X2, 0x3, [lp]
        --(i), (ii), (iii), (iv)



.L5:
        prepare 0x3, .X2
        --(4), (5), (6)
        jbr     .L6
```

(3) 関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの先頭”にくる場合）

“引数レジスタ領域がスタックの先頭にくる場合”の関数呼び出し時のスタック・フレームの生成と消滅について説明します。

図 4 － 5 に関数 func1( ) から関数 func2( ) を呼び出し，その後関数 func1( ) へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 4 － 5 スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）



関数 func2() を呼び出すときの関数 func1() 側の処理

(1) 引数を引数用レジスタに格納
呼び出す func2 の引数を r6 ～ r9 へ格納

(2) 4 ワード分以降の引数をスタックへ格納
r6 ～ r9 の 4 つに入りきらない引数を格納。引数が 5 つ以上の場合はこの処理を行う対象になります。

(3) jarl 命令へ func2() 関数へ分岐

関数 func1() を呼び出されたときの関数 func2() 側の処理

(4) SP をずらす
関数 func2 が使用するのスタックへ移動

(5) lp を退避
関数 func1() の戻りアドレスを格納

(6) レジスタ変数用レジスタを退避
関数 func2() でもレジスタ変数用レジスタを使用するとき，関数 func1 で使用していたレジスタ値を保護する必要があるために退避

(7) 引数を“引数レジスタ用領域”に格納
r6 ～ r9 の値を格納
関数 func2() から他の関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

[注]V850Ex の場合 (4)，(5)，(6) の処理は prepare 命令で実行できるため，CA850 は prepare 命令を出力します。

関数 func2() から関数 func1() へ戻るときの関数 func2() 側の処理

(i) レジスタ変数用レジスタの復帰
保護していた“関数 func1() のレジスタ変数用レジスタの値”をレジスタへ戻す

(ii) lp の復帰
関数 func1() の戻りアドレスを復帰

(iii) SP を戻す
関数 func1() が使用するのスタックへ移動

(iv) jmp [lp] で戻る

[注]V850Ex の場合 (i)，(ii)，(iii)，(iv)，の処理 dispose 命令で実行できるため，CA850 は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

(a) 呼び出す側 ～関数 func1( ) ～

- 呼び出す func2( ) の引数が 4 ワードを越えていた時，越えた分の引数の値

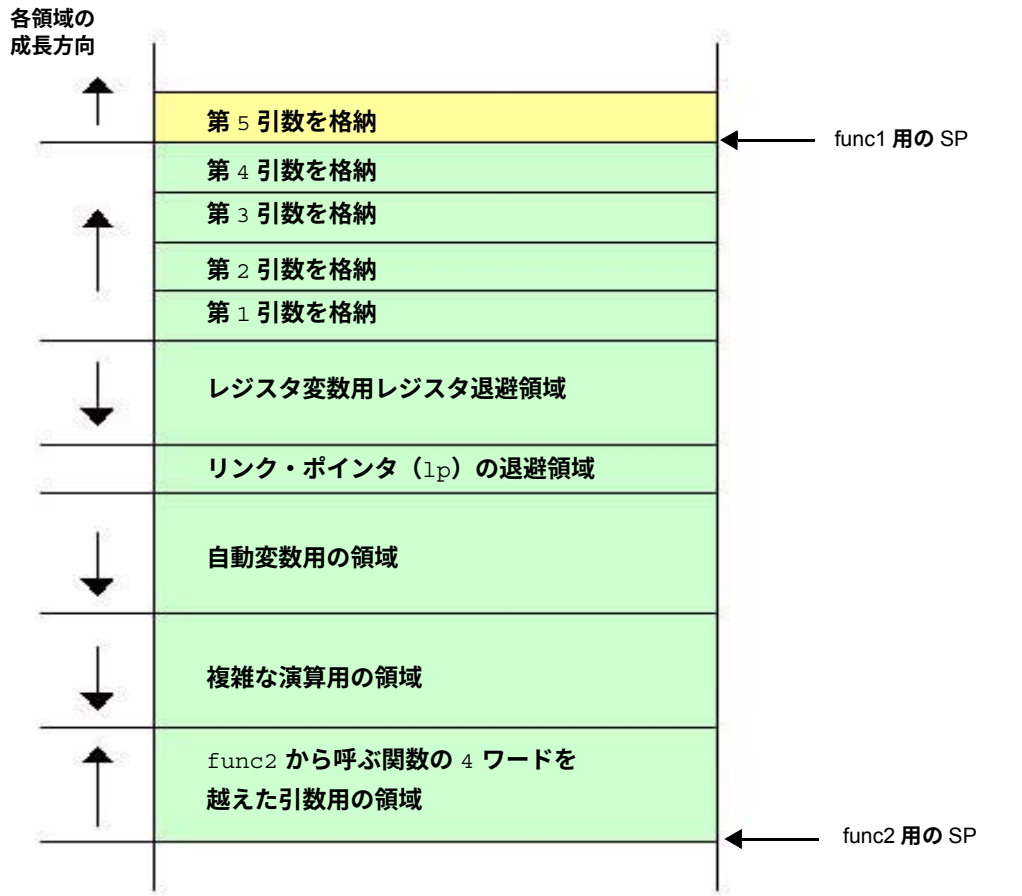
(b) 呼び出される側 ～関数 func2( ) ～

- 引数用レジスタに入れられた引数の受け渡し
  （引数用レジスタに入れるのは，呼び出す側（関数 func1( )））
- 呼び出した側（関数 func1( )）のリンク・ポインタ（lp）（＝関数 func1( ) の戻りアドレス）の退避
- “ レジスタ変数用レジスタ ” の退避
- “ 自動変数用 ” の領域
- 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保
  この領域は，図 4 － 3 には示していませんが，使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます

また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2( ) の引数が 5 個あるとします）。

図 4 － 6 スタック・フレームの各領域のスタック成長方向



次に“ C 言語関数から C 言語関数を呼び出したソース ”と“ それをコンパイルしたときのアセンブリ言語ソース ”の具体例を示します。

例

```
void func1(void)
{
       int    a, b, c, d, e;

       func2(a, b, c, d, e);
             :
}

int func2(int a, int b, int c, int d, int e)
{
       register int  i;
             :
       return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

【V850】

```
_func1:
        jbr     .L3
.L4:
        ld.w    -8+.A3[sp], r6
        ld.w    -12+.A3[sp], r7
        ld.w    -16+.A3[sp], r8   --- (1)
        ld.w    -20+.A3[sp], r9
        ld.w    -24+.A3[sp], r10
        st.w    r10, [sp]         --- (2)
        jarl    _func2, lp        --- (3)
                :
        --main に対するエピローグ
        --(ii) から (iv) の処理
.L3:
        --main に対するプロローグ
        --(4), (5) の処理
                :
        jbr     .L4

_func2:
        jbr     .L5
.L6:
        st.w    r6, .F2[sp]
        st.w    r7, 4+.F2[sp]
        st.w    r8, 8+.F2[sp]     --- (7)
        st.w    r9, 12+.F2[sp]
                :
        st.w    r29, -4+.A2[sp]
        jbr     .L2
.L2:
        ld.w    -4+.A2[sp], r10
        ld.w    -4+.F2[sp], r29 --- (i)
        ld.w    -8+.F2[sp], lp  --- (ii)
        add     .S2, sp         --- (iii)
        jmp     [lp]            --- (iv)
.L5:
        sub     -.S2, sp          --- (4)
        st.w    lp, -8+.F2[sp]    --- (5)
        st.w    r29, -4+.F2[sp]   --- (6)
        jbr     .L6
```

【V850E】

```
_func1:
        jbr     .L3
.L4:
        ld.w    -8+.A3[sp], r6
        ld.w    -12+.A3[sp], r7
        ld.w    -16+.A3[sp], r8 --- (1)
        ld.w    -20+.A3[sp], r9
        ld.w    -24+.A3[sp], r10
        st.w    r10, [sp]         --- (2)
        jarl    _func2, lp        --- (3)
                :
        --main に対するエピローグ
        --(ii) から (iv) の処理
.L3:
        --main に対するプロローグ
        --(4), (5) の処理
                :
        jbr     .L4

_func2:
        jbr     .L5
.L6:
        st.w    r6, .F2[sp]
        st.w    r7, 4+.F2[sp]
        st.w    r8, 8+.F2[sp]   --- (7)
        st.w    r9, 12+.F2[sp]
                :
        st.w    r29, -4+.A2[sp]
        jbr     .L2
.L2:
        ld.w    -4+.A2[sp], r10
        dispose .X2, 0x3
        --- (i), (ii), (iii)
        add     .S2-.F2, sp     --- (iii)
        jmp     [lp]            --- (iv)
.L5:
        add     .F2 -.S2, sp    --- (4)
        prepare 0x3, .X2
        --- (4), (5), (6)
        jbr     .L6
```

4.2 C 言語関数とアセンブラ関数間の呼び出し

C 言語関数とアセンブラ関数の間で呼び出しを行うときの注意点について説明します。

4.2.1 C 言語関数からアセンブラ関数の呼び出し

C 言語関数からアセンブラ関数を呼び出すときの注意点について説明します。

(1) 識別子について

CA850 では C 言語ソース内で外部名，たとえば，関数や外部変数が記述された場合，それらの名前をアセンブラへ出力すると，先頭に“ _（アンダースコア）”を付けた名前になります。

表 4 － 3 識別子について

C	アセンブラ
func1( )	_func1

アセンブラ命令で関数や外部変数を定義するときは，識別子の先頭に“ _ ”をつけ，C 言語関数から参照するときは“ _ ”を取った形で行ってください。

(2) スタック・フレームに関して

CA850 は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，C 言語ソースからアセンブラ関数へ分岐後は，アセンブラ関数内では，SP の指すアドレスよりも下位のアドレス領域は自由に使用することができます。逆に上位のアドレス領域の内容を変更した場合，C 言語関数で使用していた領域を破壊することにつながり，以降の動作を保証できませんので注意が必要です。上記を回避するためには，アセンブラ関数の先頭で SP を変更してからスタックを使用してください。

ただし，その際は呼び出しの前後で SP の値が保持されるようにしてください。

また，アセンブラ関数内でレジスタ変数用レジスタを使用する場合は，アセンブラ関数の呼び出し前後でレジスタ値が保持されるようにしてください（使用前にレジスタ変数用レジスタの値を退避し，使用後は復帰してください）。

レジスタ変数用レジスタは，レジスタ・モードにより異なります。

表 4 － 4 レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

(3) アセンブラ関数への引数

CA850 は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します（詳細は「4.1.1 スタック・フレーム／関数呼び出し」を参照）。アセンブラ関数内で引数値を使用するときは，それぞれに格納された値を参照してください。

なお，引数値は C 言語関数において，引数に指定された値そのもので，この値をアセンブラ関数内で変更しても，C 言語関数の動作に影響を及ぼすことはありません。

(4) アセンブラ関数からの戻り値

CA850 は「関数の戻り値は“レジスタ r10”に格納される」ことを想定したコードを生成します。アセンブラ関数からの戻り値は r10 に格納するようにしてください。

なお，構造体を返す関数の場合は“呼び出し側の関数のスタック・フレーム内”に戻り値である構造体が格納されます。

(5) C 言語関数への戻り先アドレス

CA850 は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。アセンブラ関数へ分岐するとき，lp に関数の戻り先アドレスが格納されているので，C 言語関数へ戻るときは“jmp [lp]”を実行してください。

4.2.2 アセンブラ関数から C 言語関数の呼び出し

アセンブラ関数から C 言語関数を呼び出すときの注意点について説明します。

(1) スタック・フレームについて

CA850 は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，アセンブラ関数から C 言語関数へ分岐する前に，スタック領域中の未使用領域の上位アドレスを指すように SP を設定してください。これは下位アドレスの方向にスタック・フレームが取られるためです。

(2) 作業用レジスタ

CA850 は C 言語関数呼び出しの前後において，レジスタ変数用レジスタの値は保持しますが，作業用レジスタの値は保持しません。そのため，保持しなくてはならない値を作業用レジスタに割り当てたままにしないでください。

レジスタ変数用レジスタ，作業用レジスタは，レジスタ・モードにより異なります。

表 4 － 5 レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

表 4 － 6 作業用レジスタ

レジスタ・モード	作業用レジスタ
22 レジスタ・モード	r10，r11，r12，r13，r14
26 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16
32 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16，r17，r18，r19

(3) C 言語関数への引数

CA850 は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します（詳細は「4.1.1 スタック・フレーム／関数呼び出し」を参照）。4 ワード分を越えた引数は，SP の指すアドレスから，上位方向に向かって格納してください。

(4) C 言語関数からの戻り値

CA850 は「関数の戻り値は“レジスタ r10 ”に格納される」ことを想定したコードを生成します。C 言語関数からの戻り値を使用する場合は，r10 レジスタを参照してください。

なお，構造体を返す関数の場合は，呼び出し側の戻り値用の領域に値が格納され，その領域のアドレスを引数として渡す形のコードを出力します。呼び出し側であらかじめ戻り値用の領域を確保しておく必要があります。

(5) アセンブラ関数への戻り先アドレス

CA850 は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。C 言語関数へ分岐するとき，lp に関数の戻り先アドレスを格納する必要あります。

一般的には jarl 命令によって，C 言語関数へ分岐します。

4.3 関数のプロローグ／エピローグ処理

CA850 は，関数のプロローグ／エピローグ処理の一部分を “ ランタイム・ライブラリ呼び出し ” にすることで，オブジェクト・サイズの削減を行うことができます。これを “ プロローグ／エピローグのランタイム化 ” と呼びます。つまり，関数のプロローグ／エピローグ処理は決まった処理なので，これらを “ ランタイム・ライブラリ関数 ” として CA850 でまとめて用意し，関数呼び出し時や関数戻り時にこれらの関数を呼び出します。

関数のプロローグ／エピローグ部分のアセンブラ命令の例を次に示します。

なお，例中の “（数字）” は「図 4 － 3 スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」の説明に対応しています。

例

```
int func(int a, int b, int c, int d, int e)
{
        register int  i;
              :
        return(i);
}
```

上記例の関数 f のプロローグ／エピローグ部分のアセンブラ命令

［ランタイムを使用しない場合の出力コード］

```
_func :
        jbr     .L5
.L6 :
        st.w    r6, .R2[sp]
        st.w    r7, 4+.R2[sp]
        st.w    r8, 8+.R2[sp]    ---  (7)
        st.w    r9, 12+.R2[sp]
              :
        st.w    r29, -4+.A2[sp]
        jbr     .L2
.L2 :
        ld.w    -4+.A2[sp], r10
        ld.w    -4+.F2[sp], r29  --- (i)
        ld.w    -8+.F2[sp], lp   --- (ii)
        add     .F2, sp          --- (iii)
        jmp     [lp]             --- (iv)
.L5:
        add     -.F2, sp         --- (4)
        st.w    lp, -8+.F2[sp]   --- (5)
        st.w    r29, -4+.F2[sp]  --- (6)
        jbr     .L6
```

［ランタイムを使用する場合の出力コード］

```
_func :
        jbr     .L5
.L6 :
        st.w    r6, .R2[sp]
        st.w    r7, 4+.R2[sp]
        st.w    r8, 8+.R2[sp]  ---  (7)
        st.w    r9, 12+.R2[sp]
              :
        st.w    r29, -4+.A2[sp]
        jbr     .L2
.L2 :
        ld.w    -4+ .A2[sp], r10
        add     .R2, sp        --- (iii)
        jarl    ___pop2904, lp
        -- (i), (ii), (iii), (iv)
.L5 :
        jarl    ___push2904, r10
         -- (4), (5), (6)
        add     -.R2, sp       --- (4)
        jbr     .L6
```

4.3.1 関数のプロローグ／エピローグのランタイム化の指定

プロローグ／エピローグのランタイム化は，コンパイル・オプション“ -Xpro_epi_runtime=on ”を指定することにより行います。逆にランタイム化しない場合には“ -Xpro_epi_runtime=off ”を指定します。

ただし，最適化オプション“ -Ot（実行速度優先最適化）”指定時「以外」のとき，自動的に関数のプロローグ／ エピローグのランタイム化が行われます（ 自動的にコンパイラ・オプション“ -Xpro_epi_runtime=on ”が指定されます）。

“ -Ot オプション ”「以外」を指定し，かつ，ランタイム化をしたくない場合は“ -Xpro_epi_runtime=off オプション ”を指定する必要があります。

なお，“ -Xpro_epi_runtime オプション ”は，ソース・ファイル個別に指定することも可能であるため，“ ランタイム化するファイル ”と“ ランタイム化しないファイル ”を混在することができます。

また，“ -Xpro_epi_runtime=on オプション ”で，関数のプロローグ／エピローグのランタイム化を行う場合，専用のセクション“ .pro_epi_runtime セクション ”が必要となります。

したがって，リンク・ディレクティブでは，次のように定義する必要があります。

```
.pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
```

このセクションには，プロローグ／エピローグ・ランタイム関数のテーブル情報が配置されます。

4.3.2 V850Ex における関数のプロローグ／エピローグのランタイム化

V850Ex を使用している場合，関数のプロローグ／エピローグ・ランタイム関数の呼び出しには，CALLT 命令を使用します。

CALLT 命令は“ 2 バイト命令 ”で，関数呼び出しにこの命令を使用することによって，コード・サイズ削減の効果が得られます。CALLT 命令は“ CTBP（Callt Base Pointer）レジスタ ”に“ CALLT 命令の対象となる，関数のテーブル ”を指すポインタを設定が必要となります。設定処理がプログラム内になかった場合，リンク時に次のエラー・メッセージが出力されます。

```
F4414: CallTBasePointer(CTBP) is not set. CTBP must be set when compileroption
"-Ot" (or "-Xpro_epi_runtime=off") is not specified.
```

CTBP レジスタへの値設定は，アセンブラ命令で行う必要があるため，スタート・アップ・ルーチンで行うのが適当です。

したがって，次の命令をスタート・アップ・ルーチンに追加してください。

```
mov    #___PROLOG_TABLE, r12 --  “ PROLOG ”の前の“ _（アンダースコア）”は 3 つ
ldsr   r12, 20
```

このとき，___PROLOG_TABLE が“ 関数のプロローグ／エピローグのランタイム関数 ”の関数テーブルの先頭シンボルとなり，CTBP に設定され，関数テーブル自体は“ .pro_epi_runtime セクション ”に配置されます。また，上記では“ 汎用レジスタ r12 ”を使用していますが，特に r12 である必要はありません。

この CA850 が提供する“ 関数のプロローグ／エピローグのランタイム化 ”で CALLT 命令を使用する以外で CALLT 命令を使用する場合，CTBP レジスタを退避／復帰する必要があります。もし，CALLT 命令を他のオブジェクト，たとえばミドルウエアやユーザ作成のライブラリで使用しており，かつ，その中に CTBP レジスタの退避／復帰コードがない，または挿入できない場合，“ 関数のプロローグ／エピローグのランタイム化 ”は使用できません。その場合は“ -Xpro_epi_runtime=off オプション ”を指定して，ランタイム化を抑止してください。

なお，CALLT 命令や CTBP レジスタについての詳細は，各デバイスの“ 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。

4.3.3 関数のプロローグ／エピローグのランタイム化の注意事項

関数のプロローグ／エピローグのランタイム化において，次のような注意事項があります。

(1) 関数のプロローグ／エピローグのランタイム化は，関数呼び出しが入るため，その分，実行速度は低下します。これを避けたい場合は“ -Xpro_epi_runtime=off オプション ”を指定してください。ファイル単位で“ -Xpro_epi_runtime=off オプション ”を指定すると効果的です。

(2) 呼び出される関数が少ないプログラムの場合，プロローグ／エピローグのランタイム化を行っても，コード・サイズの削減ができない場合があります。効果が見込めない場合は“ -Xpro_epi_runtime=off ”にしてください。

(3) 割り込み関数の場合は，プロローグ／エピローグ・ランタイム化は行いません。割り込み関数から呼び出される関数については，プロローグ／エピローグ・ランタイム化の対象になります。

4.4 far jump 機能

CA850 は，関数を呼び出す際に，次の “ jarl 命令 ” を使用したコードを出力します。

```
jarl   _func1, lp
```

V850 マイクロコントローラのアーキテクチャ上，jarl 命令の第一オペランドとして指定できるのは，符号拡張した 22 ビット値まで（22 ビット・ディスプレースメント）となっています。

したがって，分岐点から 1M バイト範囲内に分岐先がなかった場合，分岐ができず，リンカで次のエラー・メッセージが出力されます。

```
F4161:symbol " 関数名 "(output section : セクション名 ) is too far from output
section " セクション名 ".(value : disp 値，file : main.o, input section : .text,
offset: オフセット値 , type : R_V850_PC22 ).
```

これを解決するには，

・ 分岐先を，分岐点から 1M バイト範囲内に配置する

ことですぐに解決できますが，ターゲット・システムにおいては，この範囲内に分岐先を配置できないことがあります。これを解決するのが “ far jump 機能 ” です。

far jump 機能を使用すると，関数を呼び出すときに “ jmp 命令 ” を使用したコードを出力します。これにより，V850 が持つ 32 ビット空間すべてに分岐できるようになります。ただし，汎用レジスタを 1 つ使うことになります。far jump 機能を使用した関数呼び出しを “ far jump 呼び出し ” と呼びます。

4.4.1 far jump 指定の方法

far jump 呼び出しを行う場合 “ far jump 呼び出しを行う関数 ” を列挙したファイル（far jump 呼び出し関数一覧ファイル）を用意し，コンパイラ・オプション “ -Xfar_jump ” を使用します。

```
-Xfar_jump far jump 呼び出し関数一覧ファイル
```

または “ = ” をつけてもよいです。

```
-Xfar_jump=far jump 呼び出し関数一覧ファイル
```

“ far jump 呼び出し関数一覧ファイル ” の書式については次節を参照してください。

4.4.2 far jump 呼び出し関数一覧ファイル

far jump 呼び出しの対象とする関数を列挙する“ far jump 呼び出し関数一覧ファイル ”の書式について説明します。far jump 機能を適用したい関数を，1 行に 1 関数を記述していきます。記述する名前は，C 言語関数名の先頭に“ _（アンダースコア）”を付けた名前になります。

[far jump 呼び出し関数一覧ファイルのサンプル]

```
_func_led
_func_beep
_func_motor
  :
  :
_func_switch
```

“ _ 関数名 ”の変わりに次のようにを記述すると，すべての関数を far jump 呼び出しの対象にします。

```
{all_function}
```

{all_function} 指定があると,他に‘ _ 関数名 ’があっても,すべての関数が far jump 呼び出しの対象になります。

なお，far jump 機能は，ユーザ関数だけではなく，次のものにも適用できます（指定例については「4.4.3 far jump 機能の使用例」を参照）。

- 標準ライブラリ関数
- ランタイム・ライブラリ関数
- 関数のプロローグ／エピローグ・ランタイム関数
- リアルタイム OS のシステム・コール

“ far jump 呼び出し関数一覧ファイル ”の注意事項は次のとおりです。

- 使用できる文字は ASCII 文字のみです。
- コメントは挿入できません。
- 1 行に 1 関数のみです。
- 関数名の前後に，空白やタブを挿入できます。
- 1 行に 1023 文字まで記述できます。空白やタブも 1 文字と数えます。
- 関数名は C 言語関数の先頭に“ _（アンダースコア）”付で記述してください。
- フラッシュ／外付け ROM の再リンク機能と併用することはできません。

4.4.3 far jump 機能の使用例

far jump 機能の使用例について説明します。

(1) ユーザ関数（標準関数も同様）

[C 言語ソース・ファイル]

```
extern void func3(void);

void func(void)
{
       func3();
}
```

[far jump 呼び出し関数一覧ファイル]

```
_func3
```

[通常の呼び出しコード]

```
       #@CALL_ARG
       jarl   _func3, lp
```

[far jump 呼び出しコード]

```
       #@CALL_ARG
       movea  #_func3, tp, r10
       movea  .L18, tp, lp
       jmp    [r10]
.L18:
```

(2) ランタイム関数（マクロ呼び出しの場合）

[far jump 呼び出し関数一覧ファイル]

```
___mul
```

[通常の呼び出しコード]

```
.macro mul    arg1, arg2

       add    -8, sp
       st.w   r6, [sp]
       st.w   r7, 4[sp]
       mov    arg1, r6
       mov    arg2, r7
       jarl   ___mul, lp









       ld.w   4[sp], r7
       mov    r6, arg2
       ld.w   [sp], r6
       add    8, sp
.endm
```

[far jump 呼び出しコード]

```
.macro mul    arg1, arg2
       .local macro_ret
       add    -8, sp
       st.w   r6, [sp]
       st.w   r7, 4[sp]
       mov    arg1, r6
       mov    arg2, r7
       movea  macro_ret, tp, r31
       .option nowarning
       movea  #___mul, tp, r1
       jmp    [r1]
       .option warning
macro_ret:
       ld.w   4[sp], r7
       mov    r6, arg2
       ld.w   [sp], r6
       add    8, sp
.endm
```

(3) ランタイム関数の場合（ダイレクト呼び出しの場合）

[far jump 呼び出し関数一覧ファイル]

```
___mul
```

[通常の呼び出しコード]

```
       mov    r12, r6
       mov    r13, r7
       #@CALL_ARG    r6, r7
       #@CALL_USE    r6, r7
       jarl   ___mul, lp




       mov    r6, r13
```

[far jump 呼び出しコード]

```
       mov    r12, r6
       mov    r13, r7
       #@CALL_ARG    r6, r7
       #@CALL_USE    r6, r7
       movea  #___mul, tp, r14
       movea  .L13, tp, lp
       jmp    [r14]
.L13 :
       mov    r6, r13
```

ランタイムのマクロ呼び出しとダイレクト呼び出しはコンパイラが最適化の過程でレジスタ効率などを判定し自動的に切り替えます。

(4) リアルタイム OS のシステム・コールの場合

[far jump 呼び出し関数一覧ファイル]

```
_ext_tsk
```

[通常の呼び出しコード]

```
        #@B_EPILOGUE
        #@BEGIN_NO_OPT
        add     .S4, sp
        jr      _ext_tsk --C NR

        #@END_NO_OPT
        #@E_EPILOGUE
```

[far jump 呼び出しコード]

```
        #@B_EPILOGUE
        #@BEGIN_NO_OPT
        add     .S4, sp
        movea   #_ext_tsk, tp, r10
        jmp     [r10]          --C NR
        #@END_NO_OPT
        #@E_EPILOGUE
```

(5) プロローグ／エピローグ・ランタイム関数の場合

[far jump 呼び出し関数一覧ファイル]

```
___pop2900
___push2900
```

[通常の呼び出しコード]

```
        #@B_EPILOGUE
        jarl    ___pop2900, lp --1

        #@E_EPILOGUE
.L3 :
        jarl    ___push2900, r10




        #@E_PROLOGUE
```

[far jump 呼び出しコード]

```
        #@B_EPILOGUE
        movea   #___pop2900, tp, r11
        jmp     [r11] --1
        #@E_EPILOGUE
.L3 :
        movea   #___push2900, tp, r11
        movea   .L5, tp, r10
        jmp     [r11]
.L5 :
        #@E_PROLOGUE
```

表 4 － 7 に，far jump 指定可能な，プロローグ／エピローグ・ランタイム関数名を示します。プロローグ／エピローグ・ランタイム関数を指定する場合，一度コンパイル後に出力されたアセンブリ言語ソースで使用されている関数を確認し，指定します。

表 4 － 7 プロローグ／エピローグ・ランタイム関数一覧

___pop2000	___pop2001	___pop2002	___pop2003	___pop2004	___pop2040
___pop2100	___pop2101	___pop2102	___pop2103	___pop2104	___pop2140
___pop2200	___pop2201	___pop2202	___pop2203	___pop2204	___pop2240
___pop2300	___pop2301	___pop2302	___pop2303	___pop2304	___pop2340
___pop2400	___pop2401	___pop2402	___pop2403	___pop2404	___pop2440
___pop2500	___pop2501	___pop2502	___pop2503	___pop2504	___pop2540
___pop2600	___pop2601	___pop2602	___pop2603	___pop2604	___pop2640
___pop2700	___pop2701	___pop2702	___pop2703	___pop2704	___pop2740
___pop2800	___pop2801	___pop2802	___pop2803	___pop2804	___pop2840
___pop2900	___pop2901	___pop2902	___pop2903	___pop2904	___pop2940
___poplp00	___poplp01	___poplp02	___poplp03	___poplp04	___poplp40
___push2000	___push2001	___push2002	___push2003	___push2004	___push2040
___push2100	___push2101	___push2102	___push2103	___push2104	___push2140
___push2200	___push2201	___push2202	___push2203	___push2204	___push2240
___push2300	___push2301	___push2302	___push2303	___push2304	___push2340
___push2400	___push2401	___push2402	___push2403	___push2404	___push2440
___push2500	___push2501	___push2502	___push2503	___push2504	___push2540
___push2600	___push2601	___push2602	___push2603	___push2604	___push2640
___push2700	___push2701	___push2702	___push2703	___push2704	___push2740
___push2800	___push2801	___push2802	___push2803	___push2804	___push2840
___push2900	___push2901	___push2902	___push2903	___push2904	___push2940
___pushlp00	___pushlp01	___pushlp02	___pushlp03	___pushlp04	___pushlp40

関数のプロローグ／エピローグ・ランタイム・ライブラリについての詳細は「4.3 関数のプロローグ／エピローグ処理」を参照してください。

第 5 章　スタート・アップ・ルーチン

この章では，スタート・アップ・ルーチンについて説明します。

5.1　スタート・アップ・ルーチンで行うこと

スタート・アップ・ルーチンとは，V850 をリセットしたあと，メイン関数を実行する前に，実行するルーチンを言います。基本的にはシステムをリセットしたあとの初期化を行います。具体的には，次のことを行います。

- リセットが入ったときの RESET ハンドラの設定
- スタート・アップ・ルーチンのレジスタ・モード設定
- スタック領域の確保とスタック・ポインタの設定
- main 関数の引数領域の確保
- テキスト・ポインタ（tp）の設定
- グローバル・ポインタ（gp）の設定
- エレメント・ポインタ（ep）の設定
- マスク・レジスタ（r20，r21）へマスク値を設定
- main 関数実行前に行う必要のある周辺 I/O レジスタの初期化
- main 関数実行前に行う必要のあるユーザ・ターゲットの初期化
- sbss 領域のゼロクリア
- bss 領域のゼロクリア
- sebss 領域のゼロクリア
- tibss.byte 領域のゼロクリア
- tibss.word 領域のゼロクリア
- sibss 領域のゼロクリア
- 関数のプロローグ・エピローグ・ランタイム・ライブラリ用の CTBP 値の設定【V850E】
- プログラマブル周辺 I/O レジスタ値の設定【V850E】
- r6 と r7 を main 関数の引数に設定
- main 関数へ分岐する（リアルタイム OS を使用していない場合）
- リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）

もちろん，システムによっては必要のない処理もありますので，それらに関しては省略できます。

また，これ以外にもユーザで行っておきたい処理があった場合は記述しておきます。

なお，これらの処理は，基本的にアセンブラ命令で記述する必要があります。それぞれについて記述方法等を説明します。また，スタート・アップ・ルーチンは，CA850 でサンプルを用意しています。

サンプルの格納されている場所とファイル名は次のとおりです。

表 5 － 1 スタート・アップ・ルーチンのサンプル

格納場所	ファイル名	内容
Install Folder\lib850\r22	crtN.s	22 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	22 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル
Install Folder\lib850\r26	crtN.s	26 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	26 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル
Install Folder\lib850\r32	crtN.s	32 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	32 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル

また，スタート・アップ・ルーチンをプロジェクトに追加しなかった場合，CA850 はデフォルトのスタート・アップ・ルーチン（オブジェクト）を自動的にリンクします。リンクするファイルは，次のようになります。

- V850 コア用プロジェクトの場合：crtN.o
- V850Ex コア用プロジェクトの場合：crtE.o

それぞれ，サンプルのスタート・アップ・ルーチン “ crtN.s ”，および “ crtE.s ” をコンパイル（アセンブル）したファイルです。

また，これらのオブジェクトは，アセンブラ・オプション “ -cn ”，および “ -cnv850e ” を指定してアセンブルしており，V850 マイクロコントローラで共通に使用できるようにしたオブジェクトになっています。

5.1.1 リセットが入ったときの RESET ハンドラの設定

リセット（リセット割り込み）が入ったときの処理を記述します。V850 では，リセットが入ると 0x0 番地のハンドラ・アドレスに分岐します。そこで，0x0 番地にスタート・アップ・ルーチンの先頭へ分岐する命令を配置します。「3.7.4 割り込み／例外ハンドラの記述時の注意事項」で説明したように，リセット割り込みは C 言語上で #pragma interrupt 指定による記述ができないので，アセンブラ命令で記述します。記述は次のようになります。

```
        .section "RESET", text
        jr      __start

__start:
```

ハンドラ・アドレスへの配置には，.section 疑似命令を使用します。上記のように記述することによって RESET のハンドラ・アドレスに“ jr __start ”という命令が配置されます。

また，jr 命令で届かない場所，つまり，0x0 番地から± 1M バイト内に“ __start ”がなかった場合は，次のように jmp 命令を使用します。

```
        .section "RESET", text
        mov     #__start, lp
        jmp     [lp]

__start:
```

この場合，レジスタを 1 つ使用します。上記の例では lp（r31）レジスタを使用していますが，この時点で破壊してもよい汎用レジスタがあれば使用可能です。リセット時点では，関数からの戻りアドレスが入る lp（r31）レジスタを使用することはないので，lp（r31）レジスタの使用が安全です。

なお，これらの .section 疑似命令を記述は，特にスタート・アップ・ルーチン内ではなくても問題ありません。

また，例ではスタート・アップ・ルーチンのシンボルを“ __start ”としていますが，これも別の名前で問題ありません。

5.1.2 スタート・アップ・ルーチンのレジスタ・モード設定

アセンブラ命令で記述するスタート・アップ・ルーチンに，レジスタ・モードの設定する記述をします。

ただし，この設定をする必要があるのは，システム全体で 22 レジスタ・モード，26 レジスタ・モードを指定している場合です。32 レジスタ・モードを指定している場合は記述する必要がありません。

(1) 22 レジスタ・モード時

```
        .option reg_mode 5 5
```

(2) 26 レジスタ・モード時

```
        .option reg_mode 7 7
```

この設定をしていなかった場合，リンク時に次の警告メッセージが出力されます。

```
W4608: input files have different register modes. use -rc option for more
information.
```

5.1.3 スタック領域の確保とスタック・ポインタの設定

システムで使用するスタック領域を確保し，その領域の先頭にスタック・ポインタ（SP=r3）を設定します。ただし，リアルタイム OS を使用している場合，ここで指定するスタックは，リアルタイム OS の初期化ルーチンに分岐するまでに使用するスタックとなります。

したがって，ほとんど使用しない，またはまったく使用しないことが多いので，ここで多く確保してしまうと RAM 領域が無駄になります。リアルタイム OS の初期化ルーチンに分岐するまでにスタックを使用しているかどうかを確認してください。特に割り込みには注意が必要ですが，スタート・アップ・ルーチン内は割り込み禁止で実行することが通例です。

スタック領域の確保の方法は次のようになります。

```
        .set STACKSIZE, 0x200
        .bss
        .lcomm __stack, STACKSIZE, 4

        mov    #__stack+STACKSIZE, sp
```

上記は，確保するスタック・サイズは 0x200 バイトで，.bss 領域に確保する例です。スタックの内容は，初期値を持たないので “ bss 属性の領域 ” に配置します。もちろん sbss 領域に配置することも可能ですが，sbss 領域には gp 相対 1 命令でアクセス領域のために，配置できるサイズに限界があります。他の変数等を配置した方がよいこともあるので，スタック・サイズが大きくなる場合は，bss 領域に配置することを推奨しています。

確保するスタック・サイズを変更するときは，.set 命令に書かれている数値を変更します。また，CA850 は，スタック・ポインタ（sp）相対でメモリを参照する場合，sp が 4 バイト境界に位置していることを前提としたコードを出力しています。そのため，スタック・ポインタは必ず “ 4 バイト境界 ” に配置するようにしてください。必要ならば疑似命令 “ .align 4 ” を使用してください。

スタックはシステムの動作に大きく関わってきます。スタックが不足すると，確保した領域を越えて破壊するため，システムの暴走につながります。確保すべきスタック・サイズは，CA850 にパッケージされている stack850 などを用いて，関数で使用するスタック・サイズを見積もり，十分なサイズを確保してください。

5.1.4 main 関数の引数領域の確保

ANSI C 仕様では，main 関数の形式は，仮引数を持たない

```
int main(void) { ... }
```

として定義されるか，2 つの仮引数をもつ関数

```
int main(int argc, char *argv[]) { ... }
```

として定義されます。

ここで 2 つの仮引数を持つ関数の場合，argc は非負の値であり，仮引数の総計を示します。argv は引数文字列へのポインタの配列を示します。argv[argc] は NULL（空ポインタ）で，argc が 1 以上ならば argv[0] ～ argv[argc-1] は文字列へのポインタになります。

この argc と argv の領域をスタート・アップ・ルーチン内で確保します。確保の方法は次のとおりです。

```
        .data
        .size __argc, 4
        .align 4
__argc:
        .word 0
        .size __argv, 4
__argv:
        .word #.L16
.L16:
        .byte 0
        .byte 0
        .byte 0
        .byte 0
```

この領域は初期値定義しておくため，“ data 属性領域 ” に配置します。

```
int main(void) { ... }
```

の形で main 関数を定義する場合は，上記の領域は不要です。

削除することにより，上記の分 RAM 領域を削減することができます。

なお，実際に main 関数の引数（r6 と r7）へ設定する処理は main 関数の直前で行います。r6 と r7 をスタート・アップ・ルーチン内で使用しないのであれば，上記のプログラム直後に行っても問題ありません。設定する処理は「5.1.19 r6 と r7 を main 関数の引数に設定」を参照してください。

5.1.5 テキスト・ポインタ（tp）の設定

アプリケーションのテキスト領域であるプログラム・コードを参照する際に，配置される位置に依存しない参照（PIC：Position Independent Code）を実現するために用意されているポインタが“ テキスト・ポインタ（tp）”です。たとえば，プログラム実行中に，コード内のある箇所を参照する必要がある場合，CA850 は tp 相対でアクセスするコードを出力します。

したがって，tp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で tp を正しく設定する必要があります。

テキスト・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“ シンボル・ディレクティブ ”に定義されたシンボルに入っています。たとえば，テキスト・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__tp_TEXT@%TP_SYMBOL {TEXT};
```

このとき，テキスト・ポインタの値は“ TEXT セグメント ”の先頭になり，その値は“ __tp_TEXT ”に入ります。スタート・アップ・ルーチン内で tp をセットするには，次のように記述してください。

```
        .extern __tp_TEXT, 4
        mov     #__tp_TEXT, tp
```

シンボル・ディレクティブやリンク・ディレクティブについての詳細は“ CA850 ユーザーズ・マニュアル リンク・ディレクティブ編 ”を参照してください。

5.1.6 グローバル・ポインタ（gp）の設定

アプリケーション内で定義した外部変数／データはメモリ上に配置されます。そのメモリに配置されている変数／データを参照する際，配置位置に依存することのない参照（PID：Position Independent Data）を実現するために用意されているポインタが“グローバル・ポインタ（gp）”です。gp 相対でアクセスするセクションが存在する場合，CA850 は gp 相対でアクセスするコードを出力します。

したがって，gp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で gp を正しく設定する必要があります。

グローバル・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，グローバル・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__gp_DATA @ %GP_SYMBOL {DATA};
```

また，gp シンボル値は，上記のように“DATA セグメントなどの「データ用セグメント」の先頭を gp シンボル値とする方法”のほかに，“テキスト・シンボルからのオフセットを gp シンボル値とする”方法もあります。

この方法の場合，gp シンボルを「tp に，tp からのオフセット値を加える」ことによって決定できます。つまり，配置に依存しないコードの生成が可能になります。たとえば，“プログラム・コード”と“そのコードが使用するデータ”を同時 RAM 領域にコピーしてから実行させたい場合，コードの先頭（コピー先の先頭アドレス）さえ分かれば，gp の値もすぐ導き出せるというメリットがあります。この場合のシンボル・ディレクティブ記述は次のようになります。

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
__gp_DATA @ %GP_SYMBOL&__tp_TEXT {DATA};
```

グローバル・ポインタの値は，“__tp_TEXT に __gp_DATA の値を加えた値”となり，加える値（オフセット値）が“__gp_DATA”に入ります。したがって，スタート・アップ・ルーチン内で gp をセットするには，次のように記述します。

```
        .extern __tp_TEXT, 4
        .extern __gp_DATA, 4
        mov     #__tp_TEXT, tp
        mov     #__gp_DATA, gp
        add     tp, gp
```

これにより，正しいグローバル・ポインタの値が gp に設定されます。

シンボル・ディレクティブやリンク・ディレクティブについての詳細は“CA850 ユーザーズ・マニュアル リンク・ディレクティブ編”を参照してください。

5.1.7 エレメント・ポインタ（ep）の設定

アプリケーション内で定義した外部変数／データのうち，次に割り当てられているものは，エレメント・ポインタ（ep）からの相対でアクセスされます。

- sedata ／ sebss セクション
- sidata ／ sibss セクション
- tidata.byte ／ tibss.byte セクション
- tidata.word ／ tibss.word セクション

これらのセクションが存在する場合，CA850 は ep 相対でアクセスするコードを出力します。

したがって，ep が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で ep を正しく設定する必要があります。

エレメント・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“ シンボル・ディレクティブ ”に定義されたシンボルに入っています。たとえば，エレメント・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__ep_DATA@%EP_SYMBOL;
```

エレメント・ポインタの値は，デフォルトで“ SIDATA セグメントの先頭 ”になり，その値は“ __ep_DATA ”に入ります。

したがって，スタート・アップ・ルーチン内で ep をセットするには，次のように記述します。

```
        .extern __ep_DATA, 4
        mov    #__ep_DATA, ep
```

__ep_DATA の絶対アドレス参照を行い，その値を ep に設定します。

シンボル・ディレクティブやリンク・ディレクティブについての詳細は“ CA850 ユーザーズ・マニュアル リンク・ディレクティブ編 ”を参照してください。

5.1.8 マスク・レジスタ（r20，r21）へマスク値を設定

マスク・レジスタを使用する場合，スタート・アップ・ルーチン内で設定します。マスク・レジスタは“ r20 ”と“ r21 ”で，次の値を設定します。

- r20 は 8 ビットのマスク値“ 0xff ”
- r21 は 16 ビットのマスク値“ 0xffff ”

設定方法は次のようになります。

```
        .option nowarning
        mov     0xff, r20
        mov     0xffff, r21
        .option warning
```

“ .option nowarning ”と“ .option warning ”で囲まれた部分は，アセンブル時の警告メッセージの出力を抑止するための疑似命令です。アセンブラ・オプションで“ -m オプション ”（マスク・オプションの使用）を設定していると，r20 と r21 にマスク値が設定されたコードを出力します。そのため，ユーザが故意に r20 と r21 に対して値を代入しようとした場合，次の警告メッセージが出力されます。

```
W3013: mask register r20 or r21 used as destination register.
```

なお，マスク・レジスタについての詳細は「2.5 マスク・レジスタ」を参照してください。

5.1.9 main 関数実行前に行う必要のある周辺 I/O レジスタの初期化

スタート・アップ・ルーチン内で外部 RAM の初期化などを行う場合，まず周辺 I/O に対して外部メモリ設定等を行わないと，メモリ領域のアクセスができず，初期化ができません。その他，スタート・アップ・ルーチンを実行するうえで，設定しなくてはならない周辺 I/O レジスタの初期化を行います。

なお，レジスタ設定は，アセンブラ命令でそのまま記述してもよいですし，いったんスタート・アップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。C 言語で行うと，周辺 I/O への読み出しや代入を分かりやすく記述できます。たとえば C 言語関数“ void reset(void) ”を作成して，スタート・アップ・ルーチンから呼び出すときは，スタート・アップ・ルーチン内に次の命令を記述します。

```
        jarl   _reset, lp
```

アセンブラ命令の記述と C 言語記述の違いの例を示します。たとえば，P0（ポート 0）に“ 1 ”を代入する命令を，アセンブリ言語ソースと C 言語ソースで記述すると次のようになります。

アセンブリ言語ソース

```
        mov    1, r10
        st.b   r10, P0
```

上記では，例として r10 を使用しています。

C 言語ソース

```
#pragma ioreg

        P0 = 1;
```

外部メモリ設定等は，デバイスによって異なりますので，使用する各デバイスの“ ユーザーズ・マニュアル ハードウエア編 ”を参照してください。

また，クロック発生機能を使用し，V850 内蔵の各ユニットに供給される“ 内部システム・クロック ”を発生させる必要がありますが，このとき，PLL （Phase locked loop）シンセサイザによって，クロックを逓倍して使用します。したがって，使用する周波数を正しく設定しなければ，想定している動作速度に誤差を生じます。

PLL のデフォルト値は，たいてい逓倍値が小さく，動作周波数が低くなっています。スタート・アップ・ルーチンにおいても例外ではなく，「5.1.11 sbss 領域のゼロクリア」以降で説明する“ メモリ領域のクリア ”を，動作周波数が低いまま実行すると，実行完了までにたいへん時間がかかってしまいます。したがって，PLL の設定に 関しては，スタート・アップ・ルーチンの初期の方で行うことを推奨します。

```
-- V850ES/SG2 において 5MHz を 4 逓倍 （20MHz）にする設定
       mov    0x80,r10
       st.b   r10,PRCMD
       st.b   r10,PCC       -- fcpu = fxx
       nop
       nop
       nop
       nop
       nop
       set1   0, PLLCTL     -- PLLON = 1
```

その他“システム・ウエイト・コントロール・レジスタ（VSWC）”やコマンド・レジスタ（PRCMD），必要であれば“ウォッチ・ドック・タイマ（WDT）”などの設定も必要になりますので，使用する各デバイスの“ユーザーズ・マニュアル ハードウエア編”を参考に正しく設定してください。

5.1.10 main 関数実行前に行う必要のあるユーザ・ターゲットの初期化

スタート・アップ・ルーチン内でユーザ・ターゲットに初期化が必要なものがある場合は，その初期化処理を記述しておきます。

処理はアセンブリ言語ソースで記述してもよいですし，いったんスタート・アップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。

5.1.11 sbss 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ sbss 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sbss 領域をゼロクリアすることを推奨します。

なお，sbss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sbss 領域のクリアを行うときは，CA850 で予約されているシンボル“ __ssbss ”と“ __esbss ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 2 sbss 領域のシンボル

シンボル名	シンボルの意味
__ssbss	sbss 領域の先頭シンボル
__esbss	sbss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sbss 領域をゼロクリアするプログラムは次のとおりです。

```
       .extern __ssbss, 4
       .extern __esbss, 4
       mov    #__ssbss, r13
       mov    #__esbss, r12
       cmp    r12, r13
       jnl    .L11
.L12:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L12
.L11:
```

sbss 領域を 4 バイトずつゼロクリアしていきます。

5.1.12 bss 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ bss 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，bss 領域をゼロクリアすることを推奨します。

なお，bss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

bss 領域のクリアを行うときは，CA850 で予約されているシンボル“ __sbss ”と“ __ebss ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 3 bss 領域のシンボル

シンボル名	シンボルの意味
__sbss	bss 領域の先頭シンボル
__ebss	bss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，bss 領域をゼロクリアするプログラムは次のとおりです（bss 領域を 4 バイトずつゼロクリアしていきます）。

```
       .extern __sbss, 4
       .extern __ebss, 4
       mov    #__sbss, r13
       mov    #__ebss, r12
       cmp    r12, r13
       jnl    .L14
.L15:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L15
.L14:
```

5.1.13 sebss 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ sebss 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sebss 領域をゼロクリアすることを推奨します。

なお，sebss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sebss 領域のクリアを行うときは，CA850 で予約されているシンボル“ __ssebss ”と“ __esebss ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 4 sebss 領域のシンボル

シンボル名	シンボルの意味
__ssebss	sebss 領域の先頭シンボル
__esebss	sebss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sebss 領域をゼロクリアするプログラムは次のとおりです（sebss 領域を 4 バイトずつゼロクリアしていきます）。

```
       .extern __ssebss, 4
       .extern __esebss, 4
       mov    #__ssebss, r13
       mov    #__esebss, r12
       cmp    r12, r13
       jnl    .L17
.L18:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L18
.L17:
```

5.1.14 tibss.byte 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ tibss.byte 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.byte 領域をゼロクリアすることを推奨します。

なお，tibss.byte セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.byte 領域のクリアを行うときは，CA850 で予約されているシンボル“ __stibss.byte ”と“ __etibss.byte ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 5 tibss.byte 領域のシンボル

シンボル名	シンボルの意味
__stibss.byte	tibss.byte 領域の先頭シンボル
__etibss.byte	tibss.byte 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.byte 領域をゼロクリアするプログラムは次のとおりです（tibss.byte 領域を 4 バイトずつゼロクリアしていきます）。

```
       .extern __stibss.byte, 4
       .extern __etibss.byte, 4
       mov    #__stibss.byte, r13
       mov    #__etibss.byte, r12
       cmp    r12, r13
       jnl    .L20
.L21:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L21
.L20:
```

5.1.15 tibss.word 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ tibss.word 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.word 領域をゼロクリアすることを推奨します。

なお，tibss.word セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.word 領域のクリアを行うときは，CA850 で予約されているシンボル“ __stibss.word ”と“ __etibss.word ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 6 tibss.word 領域のシンボル

シンボル名	シンボルの意味
__stibss.word	tibss.word 領域の先頭シンボル
__etibss.word	tibss.word 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.word 領域をゼロクリアするプログラムは次のとおりです（tibss.word 領域を 4 バイトずつゼロクリアしていきます）。

```
        .extern __stibss.word, 4
        .extern __etibss.word, 4
        mov    #__stibss.word, r13
        mov    #__etibss.word, r12
        cmp    r12, r13
        jnl    .L23
.L24:
        st.w   r0, [r13]
        add    4, r13
        cmp    r12, r13
        jl     .L24
.L23:
```

5.1.16 sibss 領域のゼロクリア

初期値を持たない領域である“ bss 属性 ”領域の 1 つである“ sibss 領域 ”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sibss 領域をゼロクリアすることを推奨します。

なお，sibss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sibss 領域のクリアを行うときは，CA850 で予約されているシンボル“ __ssibss ”と“ __esibss ”を使用します。それぞれのシンボルの意味は次のとおりです。

表 5 － 7 sibss 領域のシンボル

シンボル名	シンボルの意味
__ssibss	sibss 領域の先頭シンボル
__esibss	sibss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sibss 領域をゼロクリアするプログラムは次のとおりです（sibss 領域を 4 バイトずつゼロクリアしていきます）。

```
       .extern __ssibss, 4
       .extern __esibss, 4
       mov    #__ssibss, r13
       mov    #__esibss, r12
       cmp    r12, r13
       jnl    .L26
.L25:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L25
.L26:
```

5.1.17 関数のプロローグ／エピローグ・ランタイム・ライブラリ用の CTBP 値の設定

V850Ex コアを使用している場合で，プロローグ／エピローグ・ランタイム・ライブラリを使用する場合にこの設定が必要になります。

V850Ex コアで関数のプロローグ／エピローグ・ランタイム・ライブラリを呼び出すとき，CALLT 命令を使用するため，その CALLT 命令に必要な CTBP の値を，関数のプロローグ・エピローグ・ランタイム・ライブラリの関数テーブルの先頭に設定しておく必要があります。

プロローグ／エピローグ・ランタイム・ライブラリを使用する設定になるのは，次の場合です。

- コンパイラ・オプション “ -Xpro_epi_runtime=on ” を設定している

コンパイラの最適化オプション“ -Ot ”「以外」を指定していると，自動的に“ -Xpro_epi_runtime=on ”になります。

関数のプロローグ／エピローグ・ランタイム・ライブラリの関数テーブルの先頭シンボルは，次のとおりです。

- ___PROLOG_TABLE

このシンボルを用いて，次のコードを記述します。

```
mov    #___PROLOG_TABLE, r12
ldsr   r12, 20
```

CTBP はシステム・レジスタ 20 番なので，ldsr 命令を使用して，値を設定します。

5.1.18 プログラマブル周辺 I/O レジスタの BPC 値の設定

プログラマブル周辺 I/O レジスタを搭載している V850 マイクロコントローラを使用していて，かつ，プログラマブル周辺 I/O レジスタを使用する場合に，BPC を設定する必要があります。

たとえば，V850E/IA1 の場合，次のようになっています。

BPC レジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PA15	0	PA13	PA12	PA11	PA10	PA9	PA8	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

表 5 － 8 BPC レジスタ

ビット位置	ビット名	意味
15	PA15	プログラマブル周辺 I/O エリアの使用許可／不許可を設定 0：プログラマブル周辺 I/O 領域の使用を不許可 1：プログラマブル周辺 I/O 領域の使用を許可
13 - 0	PA13 - PA0	プログラマブル周辺 I/O エリアのアドレスを設定

プログラマブル周辺 I/O レジスタを使用する場合，コンパイラ・オプション“ -Xbpc ”で，プログラマブル周辺 I/O レジスタの値を設定する必要があります。これによって CA850 は，プログラマブル周辺 I/O レジスタへアクセスするコードを出力します。ただし，このオプションで BPC に値がセットされるわけではありません。

BPC に値をセットするには，スタート・アップ・ルーチン等で BPC レジスタに値を書き込む処理が必要となります。

V850E/IA1 の場合は，PA15 に 1 を立て，PA13 ～ PA0 にプログラマブル周辺 I/O エリアのアドレスを設定することになります。たとえば，プログラマブル周辺 I/O エリアのアドレスを 0x1234 としたい場合は，BPC レジスタへの設定は次のようになります。

```
mov    0x9234, r13
st.h   r13, BPC
```

PA15 に 1 を立てる必要があるため，0x1234 と 0x8000 の論理和（OR）を取った値を BPC に設定します。CA850 のコンパイル・オプション“ -Xbpc ”に設定する値は 0x1234 で，BPC に設定する値は 0x9234 であるため，矛盾が生じないように注意する必要があります。

プログラマブル周辺 I/O レジスタについての詳細は，各デバイスの“ ユーザーズ・マニュアル ハードウエア編 ”を参照してください。

5.1.19 r6 と r7 を main 関数の引数に設定

main 関数を，2 つの仮引数をもつ関数

```
int main(int argc, char *argv[]) { /*・・・・*/ }
```

というように，2 つの仮引数をもつ関数として定義した場合，main 関数へ分岐する前に，引数（r6 と r7）へ値を設定する処理が必要になります。領域の確保は「5.1.4 main 関数の引数領域の確保」を参照してください。

なお，リアルタイム OS を使用したアプリケーションでは，main 関数は作成しないため，この処理は必要ありません。

r6 と r7 へ値を設定する処理は次のようになります。

```
        ld.w    $__argc, r6
        movea   $__argv, gp, r7
```

main 関数の引数の領域を .data セクションに配置したので，gp 相対のアクセス・コードを記述します。

5.1.20 main 関数へ分岐する

スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，main 関数への分岐命令を実行します。

ただし，リアルタイム OS を使用したアプリケーションの場合は，main 関数は作成しないため，この処理は必要ありません。代わりにリアルタイム OS の初期化ルーチンへ分岐する命令が必要になります。この詳細については「5.1.21 リアルタイム OS の初期化ルーチンへ分岐する」を参照してください。

main 関数への分岐には，次のコードを記述します。

```
        jarl    _main, lp
```

また，main 関数の実行がすべて終わった後，この分岐命令の次の 4 バイトに戻ってくることになります。戻って来ないことが分かっている場合は，次の命令も使用できます。

```
        jr      _main
```

```
        mov     #_main, lp
        jmp     [lp]
```

jmp 命令を使用すると，32 ビット全空間をアクセスすることができます。もし，“ jarl _main, lp ” を使用する場合は，main 関数実行後に戻ってくるので，戻ってきた後，デッドロックしないように，対策を施しておくと安全です。

5.1.21 リアルタイム OS の初期化ルーチンへ分岐する

リアルタイム OS を使用したアプリケーションで，スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，初期化ルーチンへ分岐します。リアルタイム OS を使用していないアプリケーションの場合，main 関数へ分岐することになりますので「5.1.20 main 関数へ分岐する」を参照してください。

NEC エレクトロニクス製のリアルタイム OS “ RX850 ” と “ RX850 Pro ” では，初期化ルーチンへの分岐の仕方が違います。

RX850 の場合

```
        .extern __urx_start
        jr      __urx_start
```

RX850 Pro の場合

```
        .extern _sit
        mov     #_sit, r10
        .extern __rx_start
        mov     #__rx_start, lp
        jmp     [lp]
```

それぞれ，初期化ルーチンの先頭シンボルも違うので，注意が必要です。詳細については，各リアルタイム OS のユーザーズ・マニュアルを参照してください。

5.2 スタート・アップ・ルーチンの例

スタート・アップ・ルーチンの例を，次に示します。

図5－1 スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言 1
# tp，gp，ep 用
#------------------------------------------------------------------------
       .extern __tp_TEXT, 4
       .extern __gp_DATA, 4
       .extern __ep_DATA, 4
#------------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言 2
# bss 属性セクション初期化用
# 使用していないセクションがある場合は削除。
# まだ使用するセクションが決まっていないような場合は，すべて書いておいて
# セクションの追加・削除によってスタート・アップ・ルーチンのアセンブル・エラー
# を出さないようにしておくとよい
#------------------------------------------------------------------------
       .extern __ssbss, 4
       .extern __esbss, 4
       .extern __sbss, 4
       .extern __ebss, 4
       .extern __ssebss, 4
       .extern __esebss, 4
       .extern __stibss.byte, 4
       .extern __etibss.byte, 4
       .extern __stibss.word, 4
       .extern __etibss.word, 4
       .extern __ssibss, 4
       .extern __esibss, 4
#------------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言
# V850Ex で関数のプロローグ・エピローグ・ランタイム・ライブラリを
# 使用するとき，関数テーブルの先頭アドレスを外部ラベル宣言しておく
#------------------------------------------------------------------------
       .extern ___PROLOG_TABLE
#------------------------------------------------------------------------
# main 関数の外部ラベル宣言
#------------------------------------------------------------------------
       .extern _main
```

図5－1 スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------------
# main 関数の引数の領域
# void main(void) 型の場合は必要なし
#------------------------------------------------------------------------
        .data
        .size __argc, 4
        .align 4
__argc:
        .word 0
        .size __argv, 4
__argv:
        .word #.L16
.L16:
        .byte 0
        .byte 0
        .byte 0
        .byte 0
#------------------------------------------------------------------------
# 以下はセクション生成のための“ダミーデータ”
# このダミーは，後で出てくる bss 属性のセクションをゼロクリアするためのもの
#
# その先頭シンボルと最後尾シンボルは，リンク時に該当セクションにデータが
# 存在したときに生成される。
# しかし，まだ使用するセクションが決まっていないような場合は，
# リンク・ディレクティブ・ファイルを書き換えてセクションの追加・削除
# するたびに，スタート・アップ・ルーチンのアセンブル・エラーが出てしまう。
# これを避けるために，ダミーデータをセクションに配置することで，
# セクションの先頭シンボルと最後尾シンボルをとりあえず生成しておく
# bss セクションは，スタック生成コードでデータが割り当てられており
# ここにダミーを作る必要がないため，記述していない。
#
# 使用するセクションが決まったときは，このダミー部分を削除し，また，ゼロクリア
# ルーチンも必要なものだけ残して削除すると，無駄がなくなり，コード効率が上がる
#------------------------------------------------------------------------
        .sbss
        .lcomm __sbss_dummy, 0, 0
        .sebss
        .lcomm __sebss_dummy, 0, 0
        .tibss.byte
        .lcomm __tibss_byte, 0, 0
        .tibss.word
        .lcomm __tibss_word, 0, 0
        .sibss
        .lcomm __sibss_dummy, 0, 0
```

図 5 － 1 スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------------
# スタック確保
# bss 領域に 0x200 バイト確保
#------------------------------------------------------------------------
       .set STACKSIZE, 0x200
       .bss
       .lcomm __stack, STACKSIZE, 4
#------------------------------------------------------------------------
# リセット・ハンドラ
# リセット・ハンドラに配置する命令を記述
#------------------------------------------------------------------------
       .section "RESET", text
       jr     __start
#------------------------------------------------------------------------
# スタート・アップ・ルーチン本体
#------------------------------------------------------------------------
       .text
       .align 4
       .globl __start
       .globl __exit
       .globl __startend
__start:
#------------------------------------------------------------------------
# __gp_DATA は tp からの相対値とすることをシンボル・ディレクティブで設定
# していることを想定
# そのため gp は tp に __gp_DATA の値を加算する
#------------------------------------------------------------------------
       mov    #__tp_TEXT, tp
       mov    #__gp_DATA, gp
       add    tp, gp
       mov    #__stack+STACKSIZE, sp
       mov    #__ep_DATA, ep
#------------------------------------------------------------------------
# マスク・レジスタの設定
# マスク・レジスタを使用しない場合はコード削減のため削除する
# 削除しなくても，プログラム中で書きつぶすので，動作上は問題ない
#------------------------------------------------------------------------
       .option nowarning
       mov    0xff, r20
       mov    0xffff, r21
       .option warning
.L11:
```

図5－1 スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------------
# sbss セクションのゼロクリア
# sbss セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
       .extern __ssbss, 4
       .extern __esbss, 4
       mov    #__ssbss, r13
       mov    #__esbss, r12
       cmp    r12, r13
       jnl    .L11
.L12:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L12
#------------------------------------------------------------------------
# bss セクションのゼロクリア
# bss セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
       .extern __sbss, 4
       .extern __ebss, 4
       mov    #__sbss, r13
       mov    #__ebss, r12
       cmp    r12, r13
       jnl    .L14
.L15:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L15
.L14:
#------------------------------------------------------------------------
# sebss セクションのゼロクリア
# sebss セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
       .extern __ssebss, 4
       .extern __esebss, 4
       mov    #__ssebss, r13
       mov    #__esebss, r12
       cmp    r12, r13
       jnl    .L17
.L18:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L18
.L17:
```

図 5 － 1 スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------------
# tibss.byte セクションのゼロクリア
# tibss.byte セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
       .extern __stibss.byte, 4
       .extern __etibss.byte, 4
       mov    #__stibss.byte, r13
       mov    #__etibss.byte, r12
       cmp    r12, r13
       jnl    .L20
.L21:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L21
.L20:
#------------------------------------------------------------------------
# tibss.word セクションのゼロクリア
# tibss.word セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
       .extern __stibss.word, 4
       .extern __etibss.word, 4
       mov    #__stibss.word, r13
       mov    #__etibss.word, r12
       cmp    r12, r13
       jnl    .L23
.L24:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L24
.L23:
```

図5－1 スタート・アップ・ルーチンの例

```
#----------------------------------------------------------------------
# sibss セクションのゼロクリア
# sibss セクションを使用していないときは，コード削減のため削除する
#----------------------------------------------------------------------
       .extern __ssibss, 4
       .extern __esibss, 4
       mov    #__ssibss, r13
       mov    #__esibss, r12
       cmp    r12, r13
       jnl    .L26
.L25:
       st.w   r0, [r13]
       add    4, r13
       cmp    r12, r13
       jl     .L25
.L26:
#----------------------------------------------------------------------
# 関数のプロローグ・エピローグ・ランタイム・ライブラリの設定
# ライブラリ関数テーブルの先頭アドレスを CTBP（システムレジスタ 20 番）にセット
# V850Ex 以外は，この記述は削除
#----------------------------------------------------------------------
       mov    #___PROLOG_TABLE, r12
       ldsr   r12, 20
#----------------------------------------------------------------------
# プログラマブル周辺 I/O レジスタの設定
# プログラマブル周辺 I/O レジスタを持たない V850 の場合はこの記述は削除
# 下記は BPC レジスタの値（設定アドレス）が 0x1234 の例
# 0x1234（アドレス）と 0x8000（プログラマブル周辺 I/O 使用）の論理和を
# BPC に設定する
#----------------------------------------------------------------------
       mov    0x9234, r13
       st.h   r13, BPC
#----------------------------------------------------------------------
# main 関数の引数を r6，r7 に設定
#----------------------------------------------------------------------
       ld.w   $__argc, r6
       movea  $__argv, gp, r7
#----------------------------------------------------------------------
# main 関数へ分岐
#----------------------------------------------------------------------
       jarl   _main, lp
#----------------------------------------------------------------------
# main 関数が戻ってきた後の処理
#----------------------------------------------------------------------
__exit:
       halt
__startend:
```

第 6 章　ライブラリ関数

この章では，CA850 が提供するライブラリ関数について説明します。

6.1　提供ライブラリ

CA850 で提供しているライブラリは，次のとおりです。

表 6 － 1 提供ライブラリ

ライブラリの種類	ライブラリ名	収録されている機能
標準ライブラリ	libc.a	可変個引数関数定義 文字列／メモリの管理 文字型のマクロ／関数 標準入出力 標準ユーティリティ関数 ランタイム・ライブラリ 関数のプロローグ／エピローグ・ランタイム・ライブラリ
数学ライブラリ	libm.a	数学関数
ROM 化用ライブラリ	libr.a	ROM 化用コピー関数

標準ライブラリや数学ライブラリを，アプリケーション内で使用するときは，関連するヘッダ・ファイルをインクルードして，ライブラリ関数を使用します。

また，リンカ・オプション（-l）で，これらのライブラリを参照するようにします。

ただし，この中で“ 可変個引数関数定義 ”と“ 文字型のマクロ／文字型の関数 ”だけを使用している場合，ライブラリの参照を行う必要はありません。

PM+ を使用する場合，これらのライブラリはデフォルトで参照する設定になっています。

また，数学ライブラリは，内部で標準ライブラリを参照しているため，数学ライブラリを使用する場合は，標準ライブラリも必要となります。

ランタイム・ライブラリは，標準ライブラリの一部ですが，浮動小数点演算や整数演算（32 ビット整数乗除算や乗余算）を行うときに，CA850 が自動的に呼び出すルーチンです。また，関数のエピローグ／プロローグ・ランタイム・ライブラリも，標準ライブラリの一部ですが，関数のプロローグ／エピローグ処理で CA850 が自動的に呼び出すルーチンです。

したがって，“ ランタイム・ライブラリ ”，および“ 関数のエピローグ／プロローグ・ランタイム・ライブラリ ”の 2 つは，他のライブラリ関数とは異なり，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

また，32 レジスタフォルダ・モードを使用し，さらにマスク・レジスタ機能を使用する場合は，標準ライブラリとして“ マスク・レジスタ用フォルダ（*Install Folder*\lib850\r32msk）”に格納されているものを使用します。

リンカは，次の場合は，自動的に上記のフォルダにある標準ライブラリを参照します。

- 32 レジスタ・モードを指定
- コンパイル・オプション“ -Xmask_reg ”でマスク・レジスタ機能を使用している

ROM 化用ライブラリは，コンパイラ・オプション“ -Xr ”を指定すると，リンカが参照するライブラリです。パッキングされたデータをコピーする関数“ _rcopy ”“ _rcopy1 ”“ _rcopy2 ”“ _rcopy4 ”が格納されています。

6.1.1 標準ライブラリ

標準ライブラリに収録されている関数を示します。収録ライブラリは“ libc.a ”です。なお，“ 関数のプロローグ／エピローグ・ランタイム・ライブラリ ”は「6.1.5 関数のプロローグ／エピローグ・ランタイム・ライブラリ」で説明します。表にある各要素の意味は次のようになります。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。
ヘッダ・ファイル	この関数／マクロを使用するうえで，C 言語ソースでインクルードする必要あるヘッダ・ファイルです。#include 指令でインクルードしてください。 例外発生時の errno を使用する場合は“ errno.h ”もインクルードする必要があります。
ANSI	この関数が ANSI 規格で規定されている関数かどうかの区別です。 規定されていれば“ ○ ”，規定されていなければ“ × ”がついています。
sdata 使用	この関数／マクロが，メモリ領域“ sdata 領域 ”を使用するかどうかの区別です。 つまり，関数が初期値を持つデータを RAM に配置しているかどうかを区別します。セクション名は“ .sdata ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sdata セクション ”を生成してください。 .sdata セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。“ ○ ”の場合，初期値ありデータを持つ必要があるため，初期値をプログラム実行前に RAM にコピーする必要があります。つまり，“ _rcopy 関数 ”を用いて ROM 化処理を行う必要があります。この処理についての詳細は“ CA850 ユーザーズ・マニュアル 操作編 ”を参照してください。
sbss 使用	この関数／マクロが，メモリ領域“ sbss 領域 ”を使用するかどうかの区別です。 つまり，関数がテンポラリとして RAM を使用するかどうかを区別します。セクション名は“ .sbss ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sbss セクション ”を生成してください。 .sbss セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。.sbss セクションは，初期値を持たないデータが配置されるので，“ .sdata 使用 ”のときと違って ROM 化処理を行う必要はありません。
リエントラント性	“ リエントラント性 ”を持つかどうかの区別です。 リエントラント性がある場合“ ○ ”，ない場合“ × ”がついています。 “ リエントラント ”とは“ 再入可能 ”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

(1) 可変個引数関数定義

表 6 － 2 可変個引数関数定義

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
va_start	引数リスト走査用変数の初期化	stdarg.h	×	×	×	－
va_arg	引数リスト走査用変数の移動	stdarg.h	×	×	×	－
va_end	引数リスト走査の終了	stdarg.h	×	×	×	－

(2) 文字列／メモリの管理

表 6 － 3 文字列関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
bcmp	メモリ比較 （memcmp の char 引数版）	string.h	×	×	×	○
bcopy	メモリ・コピー （memcpy の char 引数版）	string.h	×	×	×	○
memchr	メモリ検索	string.h	○	×	×	○
memcmp	メモリ比較	string.h	○	×	×	○
memcpy	メモリ・コピー	string.h	○	×	×	○
memmove	メモリ移動	string.h	○	×	×	○
memset	メモリ・セット	string.h	○	×	×	○

表 6 － 4 メモリ管理関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
index	文字列検索（最初の位置）	string.h	×	×	×	○
rindex	文字列検索（最後の位置）	string.h	×	×	×	○
strcat	文字列連結	string.h	○	×	×	○
strchr	文字列検索（指定文字の最初の位置）	string.h	○	×	×	○
strcmp	文字列比較	string.h	○	×	×	○
strcpy	文字列コピー	string.h	○	×	×	○
strcspn	文字列検索 （指定文字を含まない最大の長さ）	string.h	○	×	×	○
strerror	エラー番号の文字列変換	string.h	○	○	×	×
strlen	文字列の長さ	string.h	○	×	×	○
strncat	文字列連結（文字数指定）	string.h	○	×	×	○
strncmp	文字列比較（文字数指定）	string.h	○	×	×	○
strncpy	文字列コピー（文字数指定）	string.h	○	×	×	○
strpbrk	文字列検索（最初の位置）	string.h	○	×	×	○
strrchr	文字列検索（最後の位置）	string.h	○	×	×	○
strspn	文字列検索 （指定文字を含む最大の長さ）	string.h	○	×	×	○
strstr	文字列検索（指定文字の最初の位置）	string.h	○	×	×	○
strtok	トークン分割	string.h	○	×	○	×

(3) 文字型のマクロ／関数

表 6 － 5 文字の変換

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
_tolower	英大文字から英小文字変換（引数が英大文字のときだけ正しく変換）	ctype.h	×	×	×	○
_toupper	英小文字から英大文字変換（引数が英小文字のときだけ正しく変換）	ctype.h	×	×	×	○
toascii	整数から ASCII 文字変換	ctype.h	×	×	×	○
tolower	英大文字から英小文字変換（引数が英大文字以外のときそのまま）	ctype.h	○	○	×	○
toupper	英小文字から英大文字変換（引数が英大文字以外のときそのまま）	ctype.h	○	○	×	○

表 6 － 6 文字の分類

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
isalnum	ASCII 英字，または数字であるかを判定	ctype.h	○	○	×	○
isalpha	ASCII 英字であるかを判定	ctype.h	○	○	×	○
isascii	ASCII コードであるかを判定	ctype.h	×	×	×	○
iscntrl	制御文字であるかを判定	ctype.h	○	○	×	○
isdigit	10 進数であるかを判定	ctype.h	○	○	×	○
isgraph	スペース以外の表示文字であるかを判定	ctype.h	○	○	×	○
islower	英小文字であるかを判定	ctype.h	○	○	×	○
isprint	表示文字であるかを判定	ctype.h	○	○	×	○
ispunct	区切り文字であるかを判定	ctype.h	○	○	×	○
isspace	スペース／タブ／復帰／改行／垂直タブ／改ページであるかを判定	ctype.h	○	○	×	○
isupper	英大文字であるかを判定	ctype.h	○	○	×	○
isxdigit	16 進数であるかを判定	ctype.h	○	○	×	○

(4) 標準入出力

表 6 － 7 標準入出力関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
perror	エラー処理	stdio.h	○	○	×	×注
fread	ストリームからの読み込み	stdio.h	○	○	×	×
fwrite	ストリームへの書き込み	stdio.h	○	○	×	×
fgetc	ストリームからの一文字読み込み (getc と同じ)	stdio.h	○	○	×	×
fgets	ストリームからの一行読み込み	stdio.h	○	○	×	×
getc	ストリームからの一文字読み込み (fgetc と同じ)	stdio.h	○	○	×	×
getchar	標準入力からの一文字読み込み	stdio.h	○	○	×	×
gets	標準入力からの文字列読み込み	stdio.h	○	○	×	×
ungetc	入力ストリームへの一文字押し戻し	stdio.h	○	○	×	×
rewind	ファイル位置指示子のリセット	stdio.h	○	○	×	×
fputc	ストリームへの文字書き込み	stdio.h	○	○	×	×
fputs	ストリームへの文字列出力	stdio.h	○	○	×	×
putc	ストリームへの文字書き込み	stdio.h	○	○	×	×
putchar	標準出力ストリームへの文字書き込み	stdio.h	○	○	×	×
puts	標準出力ストリームへの文字列出力	stdio.h	○	○	×	×
sprintf	書式付き出力	stdio.h	○	○	○	×
fprintf	フォーマット指定したテキストをストリームへ出力	stdio.h	○	○	○	×
printf	フォーマット指定したテキストを標準出力ストリームへ出力	stdio.h	○	○	○	×
vfprintf	フォーマット指定したテキストをストリームへ書き込み	stdio.h	○	○	○	×
vprintf	フォーマット指定したテキストを標準出力ストリームへ書き込み	stdio.h	○	○	○	×
vsprintf	フォーマット指定したテキストを文字列へ書き込み	stdio.h	○	○	○	×
sscanf	書式付き入力	stdio.h	○	○	○	×
fscanf	ストリームからのデータ読み込みと解釈	stdio.h	○	○	○	×
scanf	標準出力ストリームからのテキストの読み込みと解釈	stdio.h	○	○	○	×

注 stderr に関してリエントラント性は，持ちません。

(5) 標準ユーティリティ関数

表 6 － 8 標準ユーティリティ関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
abs	絶対値（int 型）を出力	stdlib.h	○	×	×	○
labs	絶対値（long 型）を出力	stdlib.h	○	×	×	○
bsearch	バイナリ検索	stdlib.h	○	×	×	○
qsort	整列	stdlib.h	○	×	×	○
div	除算（int 型）	stdlib.h	○	×	×	○
ldiv	除算（long 型）	stdlib.h	○	×	×	○
ecvtf	浮動小数点値を数字文字列へ変換 （総文字数指定）	stdlib.h	×	○	○	×
fcvtf	浮動小数点値を数字文字列へ変換 （小数点数字数指定）	stdlib.h	×	○	○	×
gcvtf	浮動小数点値を数字文字列へ変換 （書式指定）	stdlib.h	×	○	○	×
itoa	整数（int 型）を文字列に変換	stdlib.h	×	×	×	○
ltoa	整数（long 型）を文字列に変換	stdlib.h	×	×	×	○
ultoa	整数（unsigned long 型）を文字列に変換	stdlib.h	×	×	×	○
calloc	メモリ割り当て（ゼロ初期化付き）	stdlib.h	○	○	○	×注 1
free	メモリ開放	stdlib.h	○	○	○	×注 1
malloc	メモリ割り当て（ゼロ初期化なし）	stdlib.h	○	○	○	×注 1
realloc	メモリの再割り当て	stdlib.h	○	○	○	×注 1
rand	疑似乱数列生成	stdlib.h	○	○	×	×
srand	疑似欄数列の種類を設定	stdlib.h	○	○	×	×
atoff	文字列を浮動小数点数への変換	stdlib.h	○	○	×	×注 2
strtodf	文字列を浮動小数点数への変換 （最終文字列へのポインタ格納）	stdlib.h	○	○	○	×注 2
atoi	文字列を整数（int 型）へ変換	stdlib.h	○	○	×	×注 2
atol	文字列を整数（long 型）へ変換	stdlib.h	○	○	×	×注 2
strtol	文字列を整数（long 型）へ変換し，最終文字列へのポインタを格納	stdlib.h	○	○	○	×注 2
strtoul	文字列を整数（unsigned long 型）へ変換し，最終文字列へのポインタを格納	stdlib.h	○	○	○	×注 2

注 1 再帰呼び出しはできます。
注 2 例外発生時に errno が更新されたときはリエントラント性は，持ちません。

備考 例外発生時の errno を使用する場合は，errno.h をインクルードする必要があります。

(6) 非局所分岐

表 6 － 9 非局所分岐関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
setjmp	非局所分岐の分岐先をセット	setjmp.h	○	×	×	○
longjmp	非局所分岐	setjmp.h	○	×	×	○

6.1.2 数学ライブラリ

数学ライブラリに収録されている関数を示します。収録ライブラリは“ libm.a ”です。表にある各要素の意味は次のようになります。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。
ヘッダ・ファイル	この関数／マクロを使用するうえで，C 言語ソースでインクルードする必要あるヘッダ・ファイルで。#include 指令でインクルードしてください。 また，例外発生時の errno を使用する場合は“ errno.h ”を，「1.1.10 数量的限界」で示す“ 汎整数型の各種限界値 ”をマクロ名で使用する場合は“ limits.h ”を，浮動小数点型の各種限界値を使用する場合は“ float.h ”をインクルードする必要があります。
ANSI	この関数が ANSI 規格で規定されている関数かどうかの区別です。 規定されていれば“ ○ ”，規定されていなければ“ × ”がついています。
sdata 使用	この関数／マクロが，メモリ領域“ sdata 領域 ”を使用するかどうかの区別です。 つまり，関数が初期値を持つデータを RAM に配置しているかどうかを区別します。セクション名は“ .sdata ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sdata セクション ”を生成してください。 .sdata セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。“ ○ ”の場合，初期値ありデータを持つ必要があるため，初期値をプログラム実行前に RAM にコピーする必要があります。つまり，“ _rcopy 関数 ”を用いて ROM 化処理を行う必要があります。この処理についての詳細は“ CA850 ユーザーズ・マニュアル 操作編 ”を参照してください。
sbss 使用	この関数／マクロが，メモリ領域“ sbss 領域 ”を使用するかどうかの区別です。 つまり，関数がテンポラリとして RAM を使用するかどうかを区別します。セクション名は“ .sbss ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sbss セクション ”を生成してください。 .sbss セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。.sbss セクションは，初期値を持たないデータが配置されるので，“ .sdata 使用 ”のときと違って ROM 化処理を行う必要はありません。
リエントラント性	この関数が“ リエントラント性 ”を持つかどうかの区別です。 リエントラント性がある場合“ ○ ”，ない場合“ × ”，「例外発生時の matherr の呼び出しに関してのみリエントラント性がない場合」は“ △ ”がついています。 “ リエントラント ”とは“ 再入可能 ”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

(1) 数学関数

表 6 － 10 数学関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
j0f	第一種ベッセル関数（0 次）	math.h	×	○	○	△注
j1f	第一種ベッセル関数（1 次）	math.h	×	○	○	△注
jnf	第一種ベッセル関数（n 次）	math.h	×	○	○	△注
y0f	第二種ベッセル関数（0 次）	math.h	×	○	○	△注
y1f	第二種ベッセル関数（1 次）	math.h	×	○	○	△注
ynf	第二種ベッセル関数（n 次）	math.h	×	○	○	△注
erff	誤差関数（近似値）	math.h	×	○	○	△注
erfcf	誤差関数（相補確率）	math.h	×	○	○	△注
expf	指数関数	math.h	○	○	○	△注
logf	対数関数（自然対数）	math.h	○	○	○	△注
log2f	対数関数（底 =2）	math.h	○	○	○	△注
log10f	対数関数（底 =10）	math.h	○	○	○	△注
powf	べき乗関数	math.h	○	○	○	△注
cbrtf	立方根関数	math.h	×	×	×	○
sqrtf	平方根関数	math.h	○	○	○	△注
ceilf	ceiling 関数	math.h	○	×	×	○
fabsf	絶対値関数	math.h	○	×	×	○
floorf	floor 関数	math.h	○	×	×	○
fmodf	剰余関数	math.h	○	○	○	△注
frexpf	浮動小数点数を仮数部とべき乗に分割	math.h	○	○	○	△注
ldexpf	浮動小数点数をべき乗に変換	math.h	○	○	○	△注
modff	浮動小数点数を整数部と小数部に分割	math.h	○	×	×	○
gammaf	対数ガンマ関数	math.h	×	○	○	△注
hypotf	ユークリッド距離関数	math.h	×	○	○	△注
matherr	エラー処理関数	math.h	×	×	×	○
acoshf	逆双曲線余弦	math.h	×	○	○	△注
asinhf	逆双曲線正弦	math.h	×	○	○	△注
atanhf	逆双曲線正接	math.h	×	○	○	△注
coshf	双曲線余弦	math.h	○	○	○	△注

表 6 － 10 数学関数

関数 マクロ名	概要	ヘッダ・ ファイル	ANSI	sdata 使用	sbss 使用	リエント ラント性
sinhf	双曲線正弦	math.h	○	○	○	△注
tanhf	双曲線正接	math.h	○	○	○	△注
acosf	逆余弦	math.h	○	○	○	△注
asinf	逆正弦	math.h	○	○	○	△注
atanf	逆正接	math.h	○	○	○	△注
atan2f	逆正接（y/x）	math.h	○	○	○	△注
cosf	余弦	math.h	○	○	○	△注
sinf	正弦	math.h	○	○	○	△注
tanf	正接	math.h	○	○	○	△注

注 △：例外発生時の errno の更新と matherr の呼び出しに関してのみリエントラント性なし

備考 例外発生時の errno を使用する場合は“ errno.h ”を，「1.1.10 数量的限界」で示す“ 汎整数型の各種限界値 ”をマクロ名で使用する場合は“ limits.h ”を，浮動小数点型の各種限界値を使用する場合は“ float.h ”をインクルードする必要があります。

6.1.3 ランタイム・ライブラリ

ランタイム・ライブラリとして収録されている関数を示します。収録ライブラリは“ libc.a ”です。ランタイム・ライブラリは，C 言語ソース・プログラムで浮動小数点演算や整数演算（32 ビット整数乗除算や乗余算）を行うときに，CA850 が自動的に呼び出す関数です。ランタイム・ライブラリは，“ 関数のエピローグ／プロローグ・ランタイム・ライブラリ ”と同様に，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。表にある各要素の意味は次のようになります。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。
sdata 使用	この関数／マクロが，メモリ領域“ sdata 領域 ”を使用するかどうかの区別です。 つまり，関数が初期値を持つデータを RAM に配置しているかどうかを区別します。セクション名は“ .sdata ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sdata セクション ”を生成してください。 .sdata セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。“ ○ ”の場合，初期値ありデータを持つ必要があるため，初期値をプログラム実行前に RAM にコピーする必要があります。つまり，“ _rcopy 関数 ”を用いて ROM 化処理を行う必要があります。この処理についての詳細は“ CA850 ユーザーズ・マニュアル 操作編 ”を参照してください。
sbss 使用	この関数／マクロが，メモリ領域“ sbss 領域 ”を使用するかどうかの区別です。 つまり，関数がテンポラリとして RAM を使用するかどうかを区別します。セクション名は“ .sbss ”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“ .sbss セクション ”を生成してください。 .sbss セクションを使用する場合“ ○ ”，使用しない場合“ × ”がついています。.sbss セクションは，初期値を持たないデータが配置されるので，“ .sdata 使用 ”のときと違って ROM 化処理を行う必要はありません。
リエントラント性	この関数が“ リエントラント性 ”を持つかどうかの区別です。 リエントラント性がある場合“ ○ ”，ない場合“ × ”，「例外発生時の matherr の呼び出しに関してのみリエントラント性がない場合」は“ △ ”がついています。 “ リエントラント ”とは“ 再入可能 ”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

(1) ランタイム・ライブラリ

表 6 － 11 ランタイム・ライブラリ

関数 マクロ名	概要	sdata 使用	sbss 使用	リエント ラント性
___mul	符号付き 32 ビット整数の乗算	×	×	○
___mulu	符号なし 32 ビット整数の乗算	×	×	○
___div	符号付き 32 ビット整数の除算	×	×	○
___divu	符号なし 32 ビット整数の除算	×	×	○
___mod	符号付き 32 ビット整数の剰余算	×	×	○
___modu	符号なし 32 ビット整数の剰余算【V850】	×	×	○
___addf.s	単精度浮動小数点の加算	○	×	△注
___subf.s	単精度浮動小数点の減算	○	×	△注
___mulf.s	単精度浮動小数点の乗算	○	×	△注
___divf.s	単精度浮動小数点の除算	○	×	△注
___cmpf.s	単精度浮動小数点の比較およびフラグの変更	○	×	△注
___cvt.ws	整数から単精度浮動小数点への変換	×	×	○
___trnc.sw	単精度浮動小数点数から整数への変換	×	×	○

注 △：例外発生時の errno の更新と matherr の呼び出しに関してのみリエントラント性なし

備考 例外発生時の errno を使用する場合は“ errno.h ”を，「1.1.10 数量的限界」で示す“ 汎整数型の各種限界値 ”をマクロ名で使用する場合は“ limits.h ”を，浮動小数点型の各種限界値を使用する場合は“ float.h ”をインクルードする必要があります。

6.1.4 ROM 化用ライブラリ

ROM 化用ライブラリに収録されている関数を示します。収録ライブラリは“ libr.a ”です。初期値ありデータや，プログラム・コードを RAM にコピーするためのルーチンです。ROM 化の手順については“ CA850 ユーザーズ・マニュアル 操作編 ”を参照してください。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。

- ROM 化用の関数自体は sdata 領域，sbss 領域は使用しません。sdata 領域にデータを書き込む動作は行います。
- ROM 化用の関数は，通常メイン・プログラムを実行する前に一度だけ呼び出します。そのため，リエントラント性に関しては考慮しません。
- インサーキット・エミュレータ（ICE）にロード・モジュールをダウンロードした場合，data 領域や sdata 領域に置かれる初期値ありデータは，ダウンロードした時点でセットされます。
  したがって，_rcopy 関数を呼び出さなくても，デバッグが可能になります。しかし，ROM 化用ロード・モジュールを作成して実機で動作させる場合，_rcopy 関数で初期値ありデータ・コピー等を行わないと，初期値が設定されずに期待した動作をしません。「インサーキット・エミュレータで動作したが，実機でうまく動作しない」という場合，その原因のほとんどは，この _rcopy 関数で初期値を設定していないためです。また，初期化時に，RAM をゼロクリアする処理を行っているような場合は，そのルーチンよりも後に _rcopy 関数を呼び出してください。初期値もゼロクリアされてしまうためです。

(1) ROM 化用コピー関数

表 6 － 12 ROM 化用コピー関数

関数 マクロ名	概要
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

注意 _rcopy と _rcopy1 はどちらもまったく同じ動作をします。前版との互換性のために用意しています。

内蔵命令 RAM を持っている V850（V850E/ME2 など）で，プログラム・コードを内蔵命令 RAM にコピーする場合，ハードウエアの仕様上，4 バイト単位でコピーする必要があります。その場合，関数“ _rcopy4 ”を使ってコピーします。ハードウエア的な制限がなければ，どの関数を使っても問題ありませんが，2 バイト，4 バイト単位でコピーする場合は，コピーの必要がある領域を越える（パッキングされた領域のサイズが 4 の倍数でなかった場合，パッキング領域外の領域も同時にコピーされる）可能性があるため，注意が必要です。

6.1.5 関数のプロローグ／エピローグ・ランタイム・ライブラリ

関数のエピローグ／プロローグ・ランタイム・ライブラリとして収録されている関数を示します。収録ライブラリは“ libc.a ”です。関数のエピローグ／プロローグ・ランタイム・ライブラリは，関数のプロローグ／エピローグ処理で CA850 が自動的に呼び出すルーチンです。関数のエピローグ／プロローグ・ランタイム・ライブラリは，“ ランタイム・ライブラリ ”と同様に，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

V850Ex コアの場合，関数のプロローグ／エピローグ・ランタイム・ライブラリの呼び出しに CALLT 命令を使用します。これらの関数を CALLT 命令によるテーブル呼び出しにすることにより，コード効率を上げます。

関数のプロローグ／エピローグ・ランタイム・ライブラリ呼び出しを有効にするためには，次のようになります。

- 最適化オプション“ -Ot（実行速度優先最適化）”指定時「以外」のとき
- コンパイラ・オプション“ -Xpro_epi_runtime=on ”を指定したとき

関数のプロローグ処理，エピローグ処理で使用される関数は次のようになります。

表 6 － 13 プロローグ・ランタイム・ライブラリ関数一覧

機能概略	関数名
関数のプロローグ処理	___push2000，___push2001，___push2002，___push2003， ___push2004，___push2040， ___push2100，___push2101，___push2102，___push2103， ___push2104，___push2140， ___push2200，___push2201，___push2202，___push2203， ___push2204，___push2240， ___push2300，___push2301，___push2302，___push2303， ___push2304，___push2340， ___push2400，___push2401，___push2402，___push2403， ___push2404，___push2440， ___push2500，___push2501，___push2502，___push2503， ___push2504，___push2540， ___push2600，___push2601，___push2602，___push2603， ___push2604，___push2640 ___push2700，___push2701，___push2702，___push2703， ___push2704，___push2740 ___push2800，___push2801，___push2802，___push2803， ___push2804，___push2840， ___push2900，___push2901，___push2902，___push2903， ___push2904，___push2940， ___pushlp00，___pushlp01，___pushlp02，___pushlp03， ___pushlp04，___pushlp40

表 6 － 14 プロローグ・ランタイム・ライブラリ関数一覧【V850E】

機能概略	関数名
関数のプロローグ処理	___Epush250，___Epush251，___Epush252，___Epush253， ___Epush254， ___Epush260，___Epush261，___Epush262，___Epush263， ___Epush264， ___Epush270，___Epush271，___Epush272，___Epush273， ___Epush274， ___Epush280，___Epush281，___Epush282，___Epush283， ___Epush284， ___Epush290，___Epush291，___Epush292，___Epush293， ___Epush294， ___Epushlp0，___Epushlp1，___Epushlp2，___Epushlp3， ___Epushlp4

表 6 － 15 エピローグ・ランタイム・ライブラリ関数一覧

機能概略	関数名
関数のエピローグ処理	___pop2000，___pop2001，___pop2002，___pop2003， ___pop2004，___pop2040， ___pop2100，___pop2101，___pop2102，___pop2103， ___pop2104，___pop2140， ___pop2200，___pop2201，___pop2202，___pop2203， ___pop2204，___pop2240， ___pop2300，___pop2301，___pop2302，___pop2303， ___pop2304，___pop2340， ___pop2400，___pop2401，___pop2402，___pop2403， ___pop2404，___pop2440， ___pop2500，___pop2501，___pop2502，___pop2503， ___pop2504，___pop2540， ___pop2600，___pop2601，___pop2602，___pop2603， ___pop2604，___pop2640 ___pop2700，___pop2701，___pop2702，___pop2703， ___pop2704，___pop2740 ___pop2800，___pop2801，___pop2802，___pop2803， ___pop2804，___pop2840， ___pop2900，___pop2901，___pop2902，___pop2903， ___pop2904，___pop2940， ___poplp00，___poplp01，___poplp02，___poplp03， ___poplp04，___poplp40

表 6 － 16 エピローグ・ランタイム・ライブラリ関数一覧【V850E】

機能概略	関数名
関数のエピローグ処理	___Epop250，___Epop251，___Epop252，___Epop253， ___Epop254， ___Epop260，___Epop261，___Epop262，___Epop263， ___Epop264， ___Epop270，___Epop271，___Epop272，___Epop273， ___Epop274， ___Epop280，___Epop281，___Epop282，___Epop283， ___Epop284， ___Epop290，___Epop291，___Epop292，___Epop293， ___Epop294， ___Epoplp0，___Epoplp1，___Epoplp2，___Epoplp3， ___Epoplp4

6.2 ヘッダ・ファイル

CA850 でライブラリを使用するときに必要となるヘッダ・ファイルの一覧は次のとおりです。

なお，各ファイルにはマクロ定義，関数宣言が記述されています。

表 6 － 17 ヘッダ・ファイル一覧

ファイル名	概要
ctype.h	文字の変換，分類のためのヘッダ・ファイル
errno.h	エラー条件の報告のためのヘッダ・ファイル
float.h	浮動小数点表現，浮動小数点演算のためのヘッダ・ファイル
limits.h	整数の数量的限界のためのヘッダ・ファイル
math.h	数学計算のためのヘッダ・ファイル
setjmp.h	非局所分岐のためのヘッダ・ファイル
stdarg.h	可変個の引数を持つ関数をサポートするためのヘッダ・ファイル
stddef.h	共通の定義のためのヘッダ・ファイル
stdio.h	標準入出力のためのヘッダ・ファイル
stdlib.h	標準ユーティリティのためのヘッダ・ファイル
string.h	メモリ操作，文字列操作のためのヘッダ・ファイル

6.3 リンクされるオブジェクト名

ライブラリを使用した C 言語ソースをコンパイルし，リンクしてロード・モジュールを生成するとき，ライブラリに格納されているオブジェクトを必要に応じて選択し，リンクします。リンクされるオブジェクト名は，リンク結果を表示する“ リンク・マップ・ファイル ”にて確認することができます。リンクされるオブジェクト名は，ほぼライブラリ関数名と同じですが，各関数で共通に使用されるルーチンはまとめてあり，オブジェクト名がライブラリ関数名と異なったものになっています。共通に使用されるルーチンをまとめたオブジェクトには次のものがあり，必要に応じて自動的にリンクされます。

- com1f.o
- com1xf.o
- com2f.o
- com3f.o
- com4f.o
- com5f.o
- com6f.o
- com7f.o

6.4 形式の説明

CA850 がサポートするライブラリ関数について，次の形式で説明します。

ライブラリ関数の分類

〔概要〕

各関数の機能の概要を示します。

〔形式〕

各関数の指定形式を示します。

〔説明〕

各関数の機能の詳細について示します。

〔戻り値〕

各関数の戻り値について示します。

〔注意事項〕

各関数についての補足的な注意点について示します。

〔記述例〕

各関数の簡単な記述例を示します。

また，ランタイム・ライブラリは，アセンブラ命令で記述されるため，次の項目が加わる場合があります。

〔前処理〕

必要な前処理を示します。

〔引数設定用レジスタ〕

引数設定用レジスタとして使用するレジスタ名を示します。

〔フラグ〕

変更するフラグを示します。

6.5 可変個引数関数定義

ここでは，可変個引数を持つ関数を移植性を持つ形で定義するためのマクロについて説明します。これらのマクロに関連する宣言，および定義は，“ stdarg.h ” ファイルに記述されています。

表 6 － 18 可変個引数関数定義マクロ

分類	関数名	概要
STDARG	va_start	引数リスト走査用変数の初期化
	va_arg	引数リスト走査用変数の移動
	va_end	引数リスト走査の終了

STDARG

〔概要〕

可変個の引数を持つ関数の定義

va_start，va_arg，va_end

〔形式〕

```
#include <stdarg.h>

void   va_start(va_list ap, last-named-argument)
type   va_arg(va_list ap, type)
void   va_end(va_list ap)
```

〔説明〕

可変個の引数を持つ関数 func を，移植性を持つ形で定義するには，次に示した形式を用います。

```
#include<stdarg.h>

void func(arg-declarations ...)
{
       va_list       ap;
       type          argN;

       va_start(ap, last-named-argument);
       argN = va_arg(ap, type);

       va_end(ap);
}
```

arg-declarations は，引数リストで，最後に last-named-argument が宣言されているものとします。後ろに続く “ , ... ” は可変個引数リストを示します。va_list は，引数リストの走査に用いられる変数（この例の場合 ap）の型です。

va_start(va_list ap, last-named-argument)

変数 ap を，可変個引数リストの先頭（last-named-argument の次の引数）を指すように初期化します。

va_arg(va_list ap, type)

変数 ap の指している引数を返し，次の引数を指すように変数 ap を進めます。va_arg の type には，引数が関数に渡される際に変換される型を指定します。コンパイラでは，char 型，および short 型の引数に対しては int 型を指定し，unsinged char 型，および unsigned short 型の引数に対しては unsigned int 型を指定してください。引数ごとに異なる型を指定することができますが，“ どの型の引数が渡されてきているか ” は，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。
また，“ 実際に引数がいくつ渡されてきているか ” に関しても，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。

va_end(va_list ap)

リストの走査の終了を示します。va_arg … を va_start と va_end とで囲むことにより，リストの走査を繰り返すことができます。

〔記述例〕

```
#include <stdarg.h>

void abc(int first, int second, ...)
{
       va_list ap;
       int    i;
       char   c, *fmt;

       va_start(ap, second);
       i = va_arg(ap, int);
       c = va_arg(ap, int); /* char 型は int 型に変換されています */
       fmt = va_arg(ap, char *);

       va_end(ap);
}
```

6.6 文字列／メモリの管理

ここでは，文字列処理機能，およびメモリ領域管理機能について説明します。これらの関数に関連する宣言，および定義は，“ string.h ” ファイルに記述されています。

表 6 － 19 文字列／メモリ管理の関数

分類	関数名	概要
STRING	index	文字列検索（最初の位置）
	rindex	文字列検索（最後の位置）
	strcat	文字列連結
	strchr	文字列検索（指定文字の最初の位置）
	strcmp	文字列比較
	strcpy	文字列コピー
	strcspn	文字列検索（指定文字を含まない最大の長さ）
	strerror	エラー番号の文字列変換
	strlen	文字列の長さ
	strncat	文字列連結（文字数指定）
	strncmp	文字列比較（文字数指定）
	strncpy	文字列コピー（文字数指定）
	strpbrk	文字列検索（最初の位置）
	strrchr	文字列検索（最後の位置）
	strspn	文字列検索（指定文字を含む最大の長さ）
	strstr	文字列検索（指定文字列の最初の位置）
	strtok	トークン分割
MEMORY	bcmp	メモリ比較（char 引数）
	bcopy	メモリ・コピー（char 引数）
	memchr	メモリ検索
	memcmp	メモリ比較（void 引数）
	memcpy	メモリ・コピー（void 引数）
	memmove	メモリ移動
	memset	メモリ・セット

STRING

〔概要〕

文字列に対する操作

index，rindex，strcat，strchr，strcmp，strcpy，strcspn，strerror，strlen，strncat，strncmp，strncpy，strpbrk，strrchr，strspn，strstr，strtok

〔形式〕

```
#include <string.h>

char   *index(const char *s, int c)
char   *rindex(const char *s, int c)
char   *strcat(char *dst, const char *src)
char   *strchr(const char *s, int c)
int    strcmp(const char *s1, const char *s2)
char   *strcpy(char *dst, const char *src)
size_t strcspn(const char *s1, const char *s2)
char   *strerror(int errnum)
size_t strlen(const char *s)
char   *strncat(char *dst, const char *src, size_t length)
int    strncmp(const char *s1, const char *s2, size_t length)
char   *strncpy(char *dst, const char *src, size_t length)
char   *strpbrk(const char *s1, const char *s2)
char   *strrchr(const char *s, int c)
size_t strspn(const char *s1, const char *s2)
char   *strstr(const char *s1, const char *s2)
char   *strtok(char *s, const char *delimiters)
```

〔説明〕

index(const char *s, int c)

strchr と同じ機能を持つ関数です。

rindex(const char *s, int c)

strrchr と同じ機能を持つ関数です。

strcat(char *dst, const char *src)

src の指す文字列のコピーを，null 文字（\0）を含めて，dst の指す文字列の末尾に連結します。src の最初の文字は dst の終わりの null 文字（\0）を上書きします。

strchr(const char *s, int c)

char 型に変換された c と同じ文字が，s の指す文字列中に最初に現れる位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

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

s1 の指す文字列と s2 の指す文字列とを比較します。

strcpy(char *dst, const char *src)

src の指す文字列を dst の指す配列にコピーします。

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

s1 の指す文字列の中で，s2 の指す文字列（終わりの null 文字（\0）は除く）の中にない文字のみで構成されている，最大かつ最初の部分の長さを求めます。

strerror(int errnum)

処理系定義の対応関係に従って，エラー番号 errnum を文字列に変換します。errnum の値は，通常，グローバル変数 errno がコピーされたものです。指されている配列は，アプリケーション・プログラム側で変更しないでください。

strlen(const char *s)

s の指す文字列の長さを求めます。

strncat(char *dst, const char *src, size_t length)

src の指す文字列の先頭から，最大で length 文字（src の null 文字（\0）を含む）を dst の指す文字列の末尾に連結します。src の最初の文字は dst の終わりの null 文字（\0）を上書きします。この結果には，終端を示す null 文字（\0）が常に付加されます。

strncmp(const char *s1, const char *s2, size_t length)

s1 の指す配列の文字と s2 の指す配列の文字を最大で length 文字比較します。

strncpy(char *dst, const char *src, size_t length)

src の指す配列から dst の指す配列に最大で length 文字（null 文字（\0）を含む）コピーします。src の指す配列が length 文字より短い文字列の場合，全部で length 文字分書き込まれるまで，dst の指す配列内のコピーに null 文字（\0）が付加されます。

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

s2 の指す文字列中のいずれかの文字（null 文字（\0）は除く）が s1 の指す文字列中に最初に現れた位置を求めます。

strrchr(const char *s, int c)

char 型に変換された c が s の指す文字列中に最後に現れた位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

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

s1 の指す文字列の中で，s2 の指す文字列中にある文字（null 文字（\0）は除く）のみで構成されている最大かつ最初の部分の長さを求めます。

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

s1 の指す文字列の中で，s2 の指す文字列と最初に一致する部分（null 文字（\0）は除く）の位置を求めます。

strtok(char *s, const char *delimiters)

s の指す文字列を，delimiters の指す文字列中の文字で区切ることによって，トークンの列に分割します。これは最初に呼び出されると，最初の引数として s を持ち，その後は null ポインタを最初の引数とする呼び出しが続きます。delimiters の指す区切り文字列は，呼び出しごとに異なっていてもかまいません。最初の呼び出しでは，delimiters の指す区切り文字列中に含まれない最初の文字を求めて s の指す文字列中をサーチします。そのような文字が見つからなかった場合，s の指す文字列中にはトークンが存在せず strtok は null ポインタを返します。そのような文字が見つかった場合，その文字が最初のトークンの始まりとなります。その後，strtok は，そのときの区切り文字列に含まれる文字を求めてそこからサーチを行います。

そのような文字が見つからなかった場合，そのときのトークンは s の指す文字列の終わりまで拡張され，あとに続くサーチは null ポインタを返します。そのような文字が見つかった場合，その文字はトークンの終端を示す null 文字（\0）で上書きされます。strtok は，あとに続く文字を指すポインタをセーブします。null ポインタを最初の引数の値としている場合，リエントラントでないコードになります。これは，最後の区切り文字のアドレスをアプリケーション・プログラムの中で保存しておき，これを使って，s を空でない引数として渡すようにすることで回避できます。

〔戻り値〕

strcat	dst の値を返します。
index，strchr	見つかった文字を指すポインタを返します。c がこの文字列中に現れなかった場合は，null ポインタを返します。
strcmp	1 の指す文字列が s2 の指す文字列と比べて大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。
strcpy	dst の値を返します。
strcspn	見つかった部分の長さを返します。
strerror	変換した文字列へのポインタを返します。
strlen	終端を示す null 文字（\0）の前に存在する文字の数を返します。
strncat	dst の値を返します。
strncmp	s1 の指す配列が s2 の指す配列より大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。
strncpy	dst の値を返します。
strpbrk	この文字を指すポインタを返します。s2 からの文字がいずれも s1 の中に現れなかった場合は，null ポインタを返します。
rindex，strrchr	見つかった c を指すポインタを返します。c がこの文字列中に現れなかった場合，null ポインタを返します。
strspn	見つかった部分の長さを返します。
strstr	見つかった文字列を指すポインタを返します。文字列 s2 が見つからなかった場合，null ポインタを返します。s2 が長さゼロの文字列を指している場合，s1 を返します。
strtok	トークンへのポインタを返します。トークンが存在しない場合は，null ポインタを返します。

〔注意事項〕

strncat において，null 文字（\0）は常に付加されるので，コピーが length 引数によって制限される場合，dst に付加される文字の個数は n + 1 になることに注意してください。

〔記述例〕

```
#include <string.h>

void func(char *str, const char *src)
{
       strcpy(str, src);    /* src の指す文字列を str の指す配列にコピーします */
              :
}
```

MEMORY

〔概要〕

メモリ内容に対する操作

bcmp，bcopy，memchr，memcmp，memcpy，memmove，memset

〔形式〕

```
#include <string.h>

int    bcmp(const char *s1, const char *s2, size_t n)
void   bcopy(const char *in, char *out, size_t n)
void   *memchr(const void *s, int c, size_t length)
int    memcmp(const void *s1, const void *s2, size_t n)
void   *memcpy(void *out, const void *in, size_t n)
void   *memmove(void *dst, void *src, size_t length)
void   *memset(const void *s, int c, size_t length)
```

〔説明〕

bcmp(const char *s1, const char *s2, size_t n)

memcmp と同じ機能を持つ関数です。

bcopy(const char *in, char *out, size_t n)

memcpy と同じ機能を持つ関数です。

memchr(const void *s, int c, size_t length)

s の指す領域の最初の length 個の文字の中に（char 型に変換された）文字 c が最初に現れた位置を求めます。

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

s1 の指すオブジェクトの最初の n 文字を s2 の指すオブジェクトと比較します。

memcpy(void *out, const void *in, size_t n)

n バイト分を in の指すオブジェクトから out の指すオブジェクトへコピーします。

memmove(void *dst, void *src, size_t length)

length 個の文字を，src の指すメモリ領域から dst の指すメモリ領域へ移動します。コピー元とコピー先の 2 つの領域が重なり合っていても，文字を dst の指すメモリ領域に正しくコピーします。

memset(const void *s, int c, size_t length)

s の指すオブジェクトの最初の length 文字に（unsigned char 型に変換した）c の値をコピーします。

〔戻り値〕

memchr	c が見つかった場合はこの文字を指すポインタを返し，c が見つからなかった場合は null ポインタを返します。
bcmp， memcmp	s1 の指すオブジェクトが s2 の指すオブジェクトより大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，または 0 より小さい整数を返します。
bcopy， memcpy	out の値を返します。コピー元とコピー先の領域が重なっている場合，その動作は不定です
memmove	コピー先の dst の値を返します。
memset	s の値を返します。

〔記述例〕

```
#include <string.h>

int func(const void *s1, const void *s2)
{
       int    i;

       i = memcmp(s1, s2, 5);      /* s1 の指す文字列の最初の 5 文字を s2 の指す文字列の
                                   /* 最初の 5 文字と比較します */
       return(i);
}
```

6.7 文字型のマクロ／関数

ここでは，文字をいくつかのカテゴリ（英字，数字，制御文字，空白類等）に分類するマクロ，および文字の簡単なマッピングを行うマクロについて説明します。これらのマクロは，サブルーチンとしても使用できます。

これらのマクロは，“ ctype.h ” で定義されています。

表 6 － 20 文字型のマクロ

分類	関数名	概要
CONV	_tolower	英大文字から英小文字変換
	_toupper	英小文字から英大文字変換
	toascii	整数から ASCII 文字変換
	tolower	英大文字から英小文字変換
	toupper	英小文字から英大文字変換
CTYPE	isalnum	ASCII 英字，または数字であるか
	isalpha	ASCII 英字であるか
	isascii	ASCII コードであるか
	iscntrl	制御文字であるか
	isdigit	10 進数字であるか
	isgraph	（スペース以外の）表示文字であるか
	islower	英小文字であるか
	isprint	表示文字であるか
	ispunct	区切り文字であるか
	isspace	スペース，タブ，復帰，改行，垂直タブ，または改ページであるか
	isupper	英大文字であるか
	isxdigit	16 進数字であるか

CONV

〔概要〕

文字の変換

_tolower，_toupper，toascii，tolower，toupper

〔形式〕

```
#include <ctype.h>

int    _tolower(int c)
int    _toupper(int c)
int    toascii(int c)
int    tolower(int c)
int    toupper(int c)
```

〔説明〕

_tolower(int c)

引数が大文字の英字の場合に tolower **と同じ動作をするマクロです。引数のチェックは行わないため，引数が大文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“** #undef _tolower **”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。**

_toupper(int c)

引数が小文字の英字の場合に toupper **と同じ動作をするマクロです。引数のチェックは行わないため，引数が小文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“** #undef _toupper **”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。**

toascii(int c)

引数の 8 **ビット目以上のビットを** 0 **にすることで，整数を** ASCII **文字（**0 **～** 127**）に強制変換するマクロです。“** #undef toascii **”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。**

tolower(int c)

大文字の英字を対応する小文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c **が** EOF **～** 255 **の範囲の整数の場合にのみ定義されています。“** #undef tolower **”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。**

toupper(int c)

小文字の英字を対応する大文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c が EOF ～ 255 の範囲の整数の場合にのみ定義されています。“ #undef toupper ”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

〔戻り値〕

toascii	0 から 127 までの範囲の整数を返します。
_tolower，tolower	c に対して isupper が真となる場合，それに対応して islower が真となる文字を返します。そうでない場合，c を返します。 なお，_tolower では，c に不正な値が指定された場合，その動作は不定です。
_toupper，toupper	c に対して islower が真となる場合，それに対応して isupper が真となる文字を返します。そうでない場合，c を返します。 なお，_toupper では，c に不正な値が指定された場合，その動作は不定です。

〔記述例〕

```
#include <ctype.h>

int chc = 'a';
int ret = func(chc);

int func(int c)
{
       int i;

       i = toupper(c);      /* c の英小文字 'a' を英大文字 'A' に変換します */

       return(i);
}
```

CTYPE

〔概要〕

文字の分類

isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，islower，isprint，ispunct，isspace，isupper，isxdigit

〔形式〕

```
#include <ctype.h>

int    isalnum(int c)
int    isalpha(int c)
int    isascii(int c)
int    iscntrl(int c)
int    isdigit(int c)
int    isgraph(int c)
int    islower(int c)
int    isprint(int c)
int    ispunct(int c)
int    isspace(int c)
int    isupper(int c)
int    isxdigit(int c)
```

〔説明〕

isalnum(int c)

ASCII 英字，は数字であるかどうか調べるマクロです。すべての整数値に対して定義されています。“ #undef isalnum ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isalpha(int c)

ASCII 英字であるかどうか調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isalpha ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isascii(int c)

ASCII コード（0x00 ～ 0x7F）であるかどうかを調べるマクロです。すべての整数に対して定義されています。“ #undef isascii ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

iscntrl(int c)

制御文字（0x00 ～ 0x1F，または 0x7F）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef iscntrl ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isdigit(int c)

10 進数の数字であるかどうか調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isdigit ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isgraph(int c)

スペース（0x20）以外の表示文字[注]（0x20 ～ 0x7E）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isgraph ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

注 printing character のことです。

islower(int c)

小文字の英字（a ～ z）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef islower ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isprint(int c)

表示文字（0x20 ～ 0x7E）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isprint ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

ispunct(int c)

印字可能な区切り文字 (isgraph(c) && ! isalnum(c)) であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef ispunct ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isspace(int c)

スペース，タブ，復帰，改行，垂直タブ，改ページ（0x09 ～ 0x0D，または 0x20）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isspace ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isupper(int c)

大文字の英字（A ～ Z）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isupper ” を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isxdigit(int c)

16 進数の数字（0 ～ 9，a ～ f，または A ～ F）であるかどうかを調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“ #undef isxdigit ”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

〔戻り値〕

これらのマクロは，引数 c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

〔記述例〕

```
#include <ctype.h>

void func(void)
{
        int     i, j = 0;
        char    s[50];

        for(i = 50; i <= 99; i++) {   /* コード 50 ～ 99 内の表示可能文字を配列 s に格納します */
                if(isprint(i)) {
                        s[j] = i;
                        j++;
                }
        }
}
```

6.8 標準入出力

ここでは，書式文字列の指定に従い，文字列を生成，走査する関数について説明します。これらの関数に関連する定義，および宣言は，“ stdio.h ” ファイルに記述されています。

表 6 － 21 標準入出力

分類	関数名	概要
ERROR	perror	エラー処理
FILEIO	fread 注	ストリームからの読み込み
	fwrite 注	ストリームへの書き込み
GETS	fgetc 注	ストリームからの 1 文字読み込み
	fgets 注	ストリームからの 1 行読み込み
	getc 注	ストリームからの 1 文字読み込み
	getchar 注	標準入力からの 1 文字読み込み
	gets 注	標準入力からの文字列読み込み
	ungetc 注	入力ストリームへの 1 文字押し戻し
	rewind 注	ファイル位置指示子のリセット
PUTS	fputc 注	ストリームへの文字書き込み
	fputs 注	ストリームへの文字列出力
	putc 注	ストリームへの文字書き込み
	putchar 注	標準出力ストリームへの文字書き込み
	puts 注	標準出力ストリームへの文字列出力
SPRINTF	sprintf	書式化出力
	vsprintf	フォーマット指定したテキストを文字列へ書き込み
PRINTF	fprintf 注	フォーマット指定したテキストをストリームへ出力
	printf 注	フォーマット指定したテキストを標準出力ストリームへ出力
	vfprintf 注	フォーマット指定したテキストをストリームへ書き込み
	vprintf 注	フォーマット指定したテキストを標準出力ストリームへ書き込み
SSCANF	sscanf	書式化入力
SCANF	fscanf 注	ストリームからのデータ読み込みと解釈
	scanf 注	標準出力ストリームからのテキストの読み込みと解釈

注 これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

ERROR

〔概要〕

エラー処理

perror

〔形式〕

```
#include <stdio.h>

void   perror(const char *s)
```

〔説明〕

perror(const char *s)

グローバル変数 errno に対応するエラー・メッセージを stderr へ出力します。

出力されるメッセージは，次のようになります。

s が NULL でない場合	fprintf(stderr, "%s:%s\n", s, s_fix);
s が NULL の場合	fprintf(stderr, "%s\n", sfix);

s_fix は，次のようになります。

errno が EDOM の場合	"EDOM error"
errno が ERANGE の場	"ERANGE error"
errno が 0 の場合	"no error"
その他の場合	"error xxx"(xxx は abs(errno) % 1000)

〔記述例〕

```
#include <stdio.h>

void func1(double x)
{
        double d;

        errno = 0;
        d = exp(x);
        if(errno) perror("func1"); /* exp で演算例外が発生した場合 */
                                   /* perror を呼び出します */
}
```

FILEIO

〔概要〕

直接入出力

fread 注，fwrite 注

注　これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

〔形式〕

```
#include <stdio.h>

size_t fread(void *ptr, size_t, size, size_t nmemb, FILE *stream)
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

〔説明〕

fread(void *ptr, size_t, size, size_t nmemb, FILE *stream)

stream **が指す入力ストリームから，**size **の大きさの要素を** nmemb **個入力し，**ptr **へ格納します。**stream **に指定できるのは，標準入出力の** stdin **だけです。**

fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

stream **が指す出力ストリームへ，**ptr **が指す配列から，**size **の大きさの要素を** nmemb **個出力します。**stream **に指定できるのは，標準入出力の** stdout **と** stderr **だけです。**

〔戻り値〕

fread	**入力した要素数** nmemb **を返します。**
fwrite	**出力した要素数** nmemb **を返します。**

どちらも，エラー・リターンはありません。

〔記述例〕

```
#include <stdio.h>

void func(void)
{
       struct {
              int    c;
              double d;
       } buf[10];

       fread(buf, sizeof(buf[0]), sizeof(buf)/sizeof(buf [0]), stdin);
}
```

GETS

〔概要〕

文字・文字列入力

fgetc 注，fgets 注，getc 注，getchar 注，gets 注，ungetc 注，rewind 注

注　これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータではサポートされていません。

〔形式〕

```
#include <stdio.h>

int    fgetc(FILE *stream)
char   *fgets(char *s, int n, FILE *stream)
int    getc(FILE *stream)
int    getchar()
char   *gets(char *s)
int    ungetc(int c, FILE *stream)
void   rewind(FILE *stream)
```

〔説明〕

fgetc(FILE *stream)

stream が指す入力ストリームから，1 文字を入力します。stream に指定できるのは，標準入出力の stdin だけです。

fgets(char *s, int n, FILE *stream)

stream が指す入力ストリームから，最大 n - 1 文字を入力し，s へ格納します。文字の入力は，改行文字の検出によっても終了します。この場合，改行文字も s へ格納されます。最後に文字列の終結 null 文字が s へ格納されます。stream に指定できるのは，標準入出力の stdin だけです。

getc(FILE *stream)

stream が指す入力ストリームから，1 文字を入力します。getc 関数は fgetc と完全に等価です。

getchar()

標準入出力の stdin から，1 文字を入力します。

gets(char *s)

標準入出力の stdin から，改行文字を検出するまで文字を入力し，s へ格納します。入力した改行文字は捨て，最後に，文字列の終結 null 文字が s へ格納されます。

ungetc(int c, FILE *stream)

文字 c を stream が指す入力ストリームへ押し戻します。ただし,c が EOF の場合,押し戻しは行われません。押し戻された文字 c は，次の文字入力の際，最初の文字として入力されることになります。ungetc によって，押し戻すことができるのは 1 文字だけです。ungetc を続けて実行した場合，効果があるのは最後の ungetc だけです。stream に指定できるのは，標準入出力の stdin だけです。

rewind(FILE *stream)

stream が指す入力ストリームのエラー表示子をクリアし，ファイル位置表示子をファイルの先頭に位置付けます。

ただし，stream に指定できるのは，標準入出力の stdin だけです。そのため rewind は，ungetc による押し戻し文字を破棄する効果だけを持ちます。

〔戻り値〕

fgetc，getc，getchar	入力文字を返します。
fgets，gets	s を返します。
ungetc	文字 c を返します。

全関数，エラー・リターンはありません。

〔記述例〕

```
#include <stdio.h>

int func(void)
{
       int    c;

       c = fgetc(stdin);

       return(c);
}
```

PUTS

〔概要〕

文字・文字列出力

fputc 注，fputs 注，putc 注，putchar 注，puts 注

注　これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータではサポートされていません。

〔形式〕

```
#include <stdio.h>

int    fputc(int c, FILE *stream)
int    fputs(const char *s, FILE *stream)
int    putc(int c, FILE *stream)
int    putchar(int c)
int    puts(const char *s)
```

〔説明〕

fputc(int c, FILE *stream)

stream が指す出力ストリームへ，文字 c を出力します。stream に指定できるのは，標準入出力の stdout と stderr だけです。

fputs(const char *s, FILE *stream)

stream が指す出力ストリームへ，文字列 s を出力します。文字列の終端 null 文字は出力しません。stream に指定できるのは，標準入出力の stdout と stderr だけです。

putc(int c, FILE *stream)

stream が指す出力ストリームへ，文字 c を出力します。putc 関数は fputc と完全に等価です。

putchar(int c)

標準入出力の stdout へ，文字 c を出力します。

puts(const char *s)

標準入出力の stdout へ，文字列 s を出力します。文字列の終端 null 文字は出力せず，代わりに改行文字を出力します。

〔戻り値〕

fputc，putc，putchar	文字 c を返します。
fputs，puts	0 を返します。

全関数，エラー・リターンはありません。

〔記述例〕

```
#include <stdio.h>

void func(void)
{
       fputc('a', stdout);
}
```

SPRINTF

〔概要〕

書式化出力

sprintf，vsprintf

〔形式〕

```
#include <stdio.h>

int    sprintf(char *s, const char *format [, arg, ...])
int    vsprintf(char *s, const char *format, va_list arg)
```

〔説明〕

sprintf(char *s, const char *format [, arg, ...])

それぞれの arg 引数に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを s の指す配列に書き出します。

書式に対して引数が十分にない場合，動作は不定です。書式文字列の終わりに到達するとリターンします。書式で必要としている以上に引数がある場合，余分の引数を無視します。また，s の領域が引数の 1 つと重なっていると動作は不定になります。

引数 format は，“ 後ろに続く引数がどのような出力に変換されるか ”を指定しています。書き込まれた文字の最後には null 文字（\0）が付加されます（null 文字（\0）は返り値におけるカウントの対象とはなりません）。

format は，次に示す 2 種類のディレクティブにより構成されます。

通常文字	変換されずにそのまま出力にコピーされるものです（“ % ” 以外）。
変換指示	0 個以上の引数を取り込み，指示を与えるものです。

各変換指示は，文字 “ % ” で始まります（出力中に “ % ” を入れたい場合は，書式文字列の中では “ %% ” とします）。“ % ” の後ろは，次のようになります。

%[フラグ][フィールド長][精度][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) フラグ

任意の順に置かれた，変換指示の意味を修飾する 0 個以上のフラグです。フラグ文字とその意味を次に示します。

-	変換された結果をフィールド中に左詰めにし，右側は空白で満たされます（このフラグが指定されない場合，変換された結果は右詰めにされます）。
+	符号付きの変換の結果を常に + 符号，または - 符号で始めます（このフラグが指定されない場合，変換された結果は，負の値が変換された場合にのみ符号で始められます）。

スペース	符号付きの変換の最初の文字が符号でない場合，または符号付きの変換が文字を生じない場合，その結果の前にスペース（“ ”）を付けます。スペース・フラグと + フラグとが両方現れる場合，スペース・フラグは無視されます。
#	結果を“別の形式[注1]”に変換します。o 変換に対しては，その変換結果の最初の数字が 0 になるようにその精度を増やします。x，または X 変換に対しては，0 以外の変換結果の先頭に 0x，または 0X を付加します。e, f, g, E, G 変換に対しては，その変換結果に小数点以下の数字が存在しない場合であっても，小数点“.”を付加します[注2]。g, G 変換に対しては，変換結果から後ろに続く 0 が削除されないようにします。これら以外の変換に対しては，その動作は不定となります。
0	d, e, f, g, i, o, u, x, E, G, X 変換に対し，フィールド長を埋めるために，符号，または基底の指示に続いて 0 を付加します。 0 フラグと - フラグの両方が指定された場合，0 フラグは無視されます。d, i, o, u, x, X 変換については，精度を指定している場合，ゼロ（0）フラグを無視します。 0 はフラグとして解釈され，フィールド幅の始まりとは解釈されないことに注意してください。 これら以外の変換に対してはその動作は不定となります。

注 1 alternate format のことです。

注 2 通常，小数点は，その後ろに数字が続く場合にのみ現れます。

(2) フィールド長

オプショナルな最小フィールド長です。変換された値がこのフィールド長より小さい場合，左側にスペースが詰められます（前述の左詰めフラグが与えられた場合は右側にスペースが詰められます）。このフィールド長は“*”，または 10 進整数の形を取ります。“*”で指定した場合，int 型の引数をフィールド長として使用します。負のフィールド長は，サポートしていません。負のフィールド長を指定しようとすると，正のフィールド長の前にマイナス（-）フラグが付いたものと解釈されます。

(3) 精度[注]

これに与えられる値は，d, i, o, u, x, X 変換に対しては現れる数字の個数の最小値であり，e, f, E 変換に対しては“.”の後ろに現れる数字の個数であり，g, G 変換に対しては最大有効桁数です。精度は，“*”，または 10 進整数が後ろに続く“.”の形式を取ります。“*”を指定した場合，int 型の引数を精度として使用します。負の精度を指定した場合，精度を省略したものとみなされます。“.”のみが指定された場合，精度は 0 とされます。精度がこれら以外の変換指示とともに現れた場合，動作は不定となります。

注 precision のことです。

(4) サイズ

対応する引数のデータ型を解釈するためのデイフォールトの方法を変更する，任意選択のサイズ文字 h，l，および L です。

h を指定した場合，後ろに続く d, i, o, n, u, x, X の型指定を強制的に short，または unsigned short に適用します。

l を指定した場合，後ろに続く d, i, o, u, x，X の型指定を強制的に long，または unsigned long に適用します。l はさらに，後ろに続く n の型指定を強制的に long へのポインタに適用します。h，または l といっしょにこれと別の型指定文字を使用した場合，その動作は不定です。

L を指定した場合，後ろに続く e, E, f, g, G の型指定を強制的に long double に適用します。L といっしょにこれ以外の型指定文字を使用した場合，その動作は不定です。

(5) 型指定文字

適用される変換の型を指定する文字です。変換の型を指定する文字とその意味を次に示します。

%	文字 “ % ” を出力します。引数は変換されません。変換指示は “ %% ” となります。
c	int 型の引数を unsigned char 型に変換し，変換結果の文字を出力します。
d	int 型の引数を符号付きの 10 進数に変換します。
e, E	double 型の引数を，小数点の前に（引数が 0 でない場合 0 でない）1 つの文字を持ち，小数点以下の数字の個数は精度に等しい [-]d.dddde ± dd の形式に変換します。E 変換指示は，指数部が “ e ” ではなく “ E ” で始まる数字を生成します。
f	double 型の引数を，[-]dddd.dddd の形式の 10 進表記に変換します。
g, G	精度には仮数部の数字の個数を指定するものとし，double 型の引数を e（G 変換指示の場合 E），または f の形式に変換します。変換結果の末尾の 0 は結果の小数点部から除かれます。小数点は，後ろに数字が続く場合にのみ現れます。
i	d の変換と同じ変換をします。
n	同じオブジェクト内で出力された文字の個数を格納します。int 型へのポインタを引数とします。
p	処理系定義書式でポインタを出力します。CA850 では，ポインタを unsigned long として扱っています（lu の指定と同じです）。
o, u, x, X	unsigned int 型の引数を dddd の形式の 8 進表記（o），符号なしの 10 進表記（u），符号なしの 16 進表記（x，または X）に変換します。x 変換に対しては文字 abcdef が用いられ X 変換に対しては文字 ABCDEF が用いられます。
s	引数は文字型の配列を指すポインタでなければなりません。この配列からの文字を，終端を示す null 文字（\0）の前まで（null 文字（\0）自身は含まずに）出力します。精度が指定された場合，それ以上の個数の文字は出力されません。精度が指定されなかった，または精度がこの配列の大きさ以上の値であった場合，この配列は null 文字（\0）を含むようにしてください。

vsprintf(char *s, const char *format, va_list arg)

arg の指す引数列に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを s が指す配列に出力します。vsprintf 関数は，可変個数実引数並びを arg で置き換えた sprintf と等価です。vsprintf 関数の呼び出しの前に，va_start マクロで arg を初期化しておく必要があります。

〔戻り値〕

出力された文字（null 文字（\0）は除きます）の数を返します。エラー・リターンはありません。

〔記述例〕

```
#include <stdio.h>

void func(int val)
{
       char   s[20];
       sprintf(s,"%-10.51x\n", val); /* val の値に対し，左詰め，フィールド長 10，精度 5，*/
                                     /* サイズ long，16 進表記を指定し，改行文字を付加して */
                                     /* s の指す配列へ出力します */
}
```

PRINTF

〔概要〕

書式化出力

fprintf[注]，printf[注]，vfprintf[注]，vprintf[注]

注 これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

〔形式〕

```
#include <stdio.h>

int    fprintf(FILE *stream, const char *format[, arg, ...])
int    printf(const char *format,[, arg, ...])
int    vfprintf(FILE *stream, const char *format, va_list arg)
int    vprintf(const char *format, va_list arg)
```

〔説明〕

fprintf, printf, vfprintf, vprintf の各関数の，引数 stream（ストリーム）に stdin（標準入力），stdout（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリアドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

stdio.h におけるストリーム構造体の定義

```
typedef struct {
       int    mode;  /* with error descriptions */
       unsigend handle;
       int    unget_c;
}FILE;
typedef int fpos_t;

#pragma section sdata begin
extern FILE __struct_stdin;
extern FILE __struct_stdout;
extern FILE __struct_stderr;
#pragma section sdata end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“ なし ”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

入出力 I/O アドレス設定例

```
__struct_stdout.handle = 0xfffff000;
__struct_stderr.handle = 0x00fff000;
__struct_stdin.handle = 0xfffff002;
extern FILE __struct_stdout;
extern FILE __struct_stderr;
#pragma section sdata end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

fprintf(FILE *stream, const char *format[, arg, ...])

それぞれの arg 引数に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを stream に出力します。stream に指定できるのは，標準入出力の stdout と stderr だけです。書式 format の記述方法は sprintf 関数と同様です。sprintf と違って，最後に null 文字（\0）は出力されません。

printf(const char *format,[, arg, ...])

それぞれの arg 引数に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。書式 format の記述方法は sprintf 関数と同様です。sprintf と異なり，最後に null 文字（\0）は出力されません。

vfprintf(FILE *stream, const char *format, va_list arg)

arg の指す引数列に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを stream に出力します。stream に指定できるのは，標準入出力の stdout と stderr だけです。書式 format の記述方法は sprintf 関数と同様です。vfprintf 関数は，可変個数実引数並びを arg で置き換えた fprintf と等価です。vfprintf 関数の呼び出しの前に，va_start マクロで arg を初期化しておく必要があります。

vprintf(const char *format, va_list arg)

arg の指す引数列に format の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。書式 format の記述方法は sprintf 関数と同様です。vprintf 関数は，可変個数実引数並びを arg で置き換えた printf と等価です。vprintf 関数の呼び出しの前に，va_start マクロで arg を初期化しておく必要があります。

〔戻り値〕

出力された文字数を返します。

〔記述例〕

```
#include <stdio.h>

void func(int val)
{
       fprintf(stdout, "%-10.5x＼n", val);
}

 /* 汎用のエラー報告ルーチンにおける vfprintf の使用例 */
void error(char *function_name, char *format, ...)
{
       va_list arg;

       va_start(arg, format);

       /* エラーが発生した関数名を出力 */
       fprintf(stderr, "ERROR in %s:", function_name);

       /* 残りのメッセージを出力 */
       vfprintf(stderr, format, arg);

       va_end(arg);
}
```

SSCANF

〔概要〕

書式化入力

sscanff

〔形式〕

```
#include <stdio.h>

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

〔説明〕

sscanf(const char *s, const char *format[, arg, ...])

format の指す文字列で指定された書式に従い，その後ろに続く引数 arg を，変換された入力を格納するオブジェクトを指すポインタとして扱い，s の指す配列から変換する入力を読み込みます。

format には，認識されうる入力列，および “ 代入のためにどのように変換を行うか ” ということを指定します。format に対し十分な引数が存在しない場合，その動作は不定となります。引数が残っているのに format が使い果たされた場合，残された引数は無視されます。

format は，次に示す 3 種類のディレクティブにより構成されます。

1 個以上の空白類	スペース（ ），タブ（\t），改行（\n）です。 sscanf を実行して，文字列内に空白文字が見つかった場合，次の空白でない文字まで連続するすべての空白文字を読み込みます（格納はしません）。
通常の文字	“ % ” 以外のすべての ASCII 文字です。 sscanf を実行して，文字列内に通常の文字が見つかった場合，それを読み込みますが，格納はしません。変換指示により，sscanf は，入力フィールドから文字列を読み込み，特定の型の値に変換し，引数で指定した位置に格納します。変換指示で明示されて一致しているのでなければ，後ろに続く空白は読み込まれません。
変換指示	0 個以上の引数を取り込み，変換の指示を与えます。

各変換指示は “ % ” で始まります。“ % ” の後ろは，次のようになります。

%[代入抑制文字][フィールド長][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) 代入抑制文字

入力フィールドの解釈，および代入を抑制する “ * ” です。

(2) フィールド長

最大フィールド長を規定する 0 以外の 10 進整数です。入力フィールドを変換する前に読み込まれる最大文字数を指定します。入力フィールドがこのフィールド長より小さい場合，sscanf はフィールド内のすべての文字を読み込み，次のフィールドとその変換指示へ進みます。また，フィールド長分を読み込む前に，空白文字，または変換不能文字が見つかった場合，その文字までの文字群を読み込み，変換し，格納します。その後，sscanf は次の変換指示へ進みます。

(3) サイズ

対応する引数のデータ型を解釈するデフォルトの方法を変更する，任意選択のサイズ文字 h, l，および L です。

h を指定した場合，後ろに続く d, i, n, o, u, x の型指定を強制的に short int 型に変換し，short 型で格納します。c, e, f, n, p, s, D, I, O, U, X では，何もしません。

l を指定した場合，後ろに続く d, i, n, o, u, x の型指定を強制的に long int 型に変換し，long 型で格納します。e, f, g では，強制的に double 型に変換し，double 型で格納します。c, n, p, s, D, I, O, U, X では，何もしません。

L を指定した場合，後ろに続く d, i, o, u, x の型指定を強制的に long double 型に変換し，long double 型で格納します。他の型指定では，何もしません。

これら以外の場合，その動作は不定です。

(4) 型指定文字

適用される変換の型を指定する文字です。変換の型を指定する文字とその意味を次に示します。

%	文字 “ % ” にマッチします。変換も代入も行われません。変換指示は “ %% ” となります。
c	1 文字を走査します。対応する引数は “ char *arg ” にしてください。
d	10 進整数を対応する引数に読み込みます。対応する引数は “ int *arg ” にしてください。
e, f, g	浮動小数点数を対応する引数に読み込みます。対応する引数は “ float *arg ” にしてください。
i	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は “ int *arg ” にしてください。
n	対応する引数に読み込んだ文字の個数を格納します。対応する引数は “ int *arg ” にしてください。
o	8 進整数を対応する引数に読み込みます。対応する引数は “ int *arg ” にしてください。
p	走査したポインタを格納します。これは処理系定義です。 CA850 では，%p を %U とまったく同じように処理しています。対応する引数は “ void **arg ” にしてください。
s	与えられた配列の中に文字列を読み込みます。対応する引数は “ char arg[] ” にしてください。
u	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は “ unsigned int *arg ” にしてください。
x, X	16 進整数を対応する引数に読み込みます。対応する引数は “ int *arg ” にしてください。
D	10 進整数を対応する引数に読み込みます。対応する引数は “ long *arg ” にしてください。
E, F, G	浮動小数点数を対応する引数に読み込みます。対応する引数は “ double *arg ” にしてください。

l	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は “ long *arg ” にしてください。
O	8 進整数を対応する引数に読み込みます。対応する引数は “ long *arg ” にしてください。
U	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は “ unsigned long *arg ” にしてください。
[ ]	空でない文字列を引数 arg で始まるメモリの中へ読み込みます。この領域には，文字列と，自動的に付加される，文字列の終わりを示す null 文字（\0）とを受け入れられる大きさが必要です。対応する引数は “ char *arg ” にしてください。 [ ] で囲まれた文字パターンを，型指定文字 s の代わりに使用することができます。文字パターンは，sscanf の入力フィールドを構成する文字の検索セットを定義する文字集合です。[ ] 内の最初の文字が “ ^ ” の場合，検索セットは反転され，[ ] 内の文字以外のすべての ASCII 文字が含まれます。また，ショートカットとして使用できる範囲指定機能もあります。たとえば，%[0-9] は，すべての 10 進数字と一致します。この集合内では，“ - ” は最初，または最後の文字にはできません。“ - ” の前の文字は，その後ろの文字よりも辞書式順序で小さくなるようにしてください。 例 `%[abcd]a, b, c, d` のみを含む文字列と一致します。 `%[^abcd]a, b, c, d` 以外の任意の文字を含む文字列と一致します。 `%[A-DW-Z]A, B, C, D, W, X, Y, Z` を含む文字列と一致します。 `%[z-a] z, -, a` と一致します（範囲指定とはみなされません）。

浮動小数点数 (型指定文字 e, f, g, E, F, G) の場合，次の一般形式に対応させてください。

[+ | -]ddddd[.]ddd[E | e[+ | -]ddd]

ただし，上記の一般形式のうち [ ] で囲まれた部分は任意選択であり，ddd は 10 進，8 進，または 16 進数字を表します。

〔戻り値〕

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

〔注意事項〕

- sscanf は，通常のフィールド終了文字に到達する前に，特定フィールドの走査を停止したり完全に終了したりする可能性があります。
- sscanfは,次の状況では,その時点でのフィールドの走査,格納を停止し,次の入力フィールドに移動します。
  (a) 代入抑制文字（*）が書式指定の中で“％”の後ろに現れており，その時点の入力フィールドは走査されているが格納はされていない。
  (b) フィールド長（正の 10 進整数）指定文字を読み込んだ。
  (c) 読み込む次の文字がその変換指示では変換できない（たとえば，指示が 10 進のときに Z を読み込む場合）。
  (d) 入力フィールド内の次の文字が検索セット内に現れていない（，または反転検索セット内に現れている）。sscanf が以上の理由からその時点の入力フィールドの走査を停止すると，次の文字が未読であるとみなされ，次の入力フィールドの最初の文字，またはその入力のあとの読み込み操作の最初の文字として使用されます。
- sscanf は，次の状況では終了します。
  (a) 入力フィールド内の次の文字が変換する文字列内の対応する通常文字と一致していない。
  (b) 入力フィールド内の次の文字が EOF である。
  (c) 変換する文字列が終了した。
- 変換する文字列に変換指示の一部ではない文字の並びが含まれている場合，この同じ文字の並びは入力の中に現れないようにしてください。sscanf は一致する文字を走査しますが，格納はしません。不一致があった場合，一致していない最初の文字は読み取られていなかったかのように入力の中に残っています。

〔記述例〕

```
#include <stdio.h>

void func(void)
{
       int    i, n;
       float  x;
       const char    *s;
       char   name[10];

       s = "23 11.1e-1 NAME";

       n = sscanf(s,"%d%f%s", &i, &x, name);
                     /* i に 23，x に 1.110000，name に“NAME”を格納します */
                     /* 戻り値 n は 3 となります */
}
```

SCANF

〔概要〕

書式化入力

fscanf[注]，scanf[注]

注 これらの関数は，NEC エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

〔形式〕

```
#include <stdio.h>

int    fscanf(FILE *stream, const char *format[, arg, ...])
int    scanf(const char *format[, arg, ...])
```

〔説明〕

fscanf(FILE *stream, const char *format[, arg, ...])

format の指す文字列で指定された書式に従い，その後ろに続く引数 arg を，変換された入力を格納するオブジェクトとして扱い，stream から変換する入力を読み込みます。stream に指定できるのは，標準入出力の stdin だけです。書式 format の記述方法は sscanf 関数と同様です。

scanf(const char *format[, arg, ...])

format の指す文字列で指定された書式に従い，その後ろに続く引数 arg を，変換された入力を格納するオブジェクトとして扱い，標準入出力の stdin から変換する入力を読み込みます。書式 format の記述方法は sscanf 関数と同様です。

〔戻り値〕

sscanf 関数と同様です。sscanf の項を参照してください。

〔記述例〕

```
#include <stdio.h>

void func(void)
{
       int    i, n;
       double x;
       char   name[10];

       n = scanf("%d%lf%s", &i, &x, name);
                              /*“23 11.1e-1 NAME”の形式の stdin から入力を書式化入力します */
}
```

6.9 標準ユーティリティ関数

ここでは，いろいろなプログラムに役立つユーティリティ関数について説明します。これらの関数に関連する定義，および宣言は，“ stdlib.h ” ファイルに記述されています。

表 6 － 22 標準ユーティリティ関数

分類	関数名	概要
ABS	abs	絶対値（int 型）
	labs	絶対値（long 型）
BSEARCH	bsearch	バイナリ検索
	qsort	整列
DIV	div	除算（int 型）
	ldiv	除算（long 型）
ECVTF	ecvtf	浮動小数点数値を数字文字列へ変換（総文字数指定）
	fcvtf	浮動小数点数値を数字文字列へ変換（小数点数字数指定）
	gcvtf	浮動小数点数値を数字文字列へ変換（書式指定）
ITOA	itoa	整数（int 型）を文字列に変換
	ltoa	整数（long 型）を文字列に変換
	ultoa	整数（unsigned long 型）を文字列に変換
MALLOC	calloc	動的メモリの割り当て
	free	動的メモリの開放
	malloc	動的メモリの割り当て
	realloc	動的メモリの再割り当て
RAND	rand	疑似乱数列生成
	srand	疑似乱数列の種の設定
STRTODF	atoff	文字列を浮動小数点数へ変換
	strtodf	文字列を浮動小数点数へ変換（最終文字列へのポインタ格納）
STRTOL	atoi	文字列を整数へ変換（int 型）
	atol	文字列を整数へ変換（long 型）
	strtol	文字列を整数へ変換（long 型，最終文字列へのポインタ格納）
	strtoul	文字列を整数へ変換（unsigned long 型，最終文字列へのポインタ格納）

ABS

〔概要〕

整数絶対値

abs，labs

〔形式〕

```
#include <stdlib.h>

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

〔説明〕

abs(int j)

j の絶対値（j の大きさ），| j | を求めます。つまり，j が負の数の場合，結果は j の反転であり，負でない場合，j となります。

labs(long j)

abs と同じですが，int 型の値の代わりに long 型を使用し，戻り値も long 型です。

〔戻り値〕

j の絶対値 (j の大きさ)，| j | を返します。

〔記述例〕

```
#include <stdlib.h>

void func(int l)
{
        int     val;

        val = -15;
        l = abs(val); /* val の値の絶対値，15 を l に返します */
}
```

BSEARCH

〔概要〕

バイナリ検索

bsearch，qsort

〔形式〕

```
#include <stdlib.h>

void bsearch(const void *key, const void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void*))
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void*, const
void*))
```

〔説明〕

bsearch(const void *key, const void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void*))

バイナリ検索法により，base から始まる配列の中で，key と一致する要素を検索します。nmemb は，配列の要素数です。size は，各要素のサイズです。配列は，compar（最後の引数）が指す比較関数に関し昇順で整列するようにしてください。compar が指す比較関数は，2 つの引数を持つように定義してください。結果は，1 番目の引数が 2 番目の引数よりも小さい場合は負，2 つの引数が一致する場合はゼロ，1 番目の引数が 2 番目の引数よりも大きい場合は正の整数を返すようにしてください。

qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void*, const void*))

base の指す配列を compar が指す比較関数に関し昇順に整列します。nmemb は配列の要素数，size は各要素のサイズです。compar が指す比較関数は bsearch と同様です。

〔戻り値〕

key と一致する配列の要素へのポインタを返します。一致する要素が複数ある場合，結果はその中で最初に見つかった要素を指します。key と一致する要素が見つからなかった場合，null ポインタを返します。

〔記述例〕

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

int    compar(char **x, char **y);

void func(void)
{
       static char    *base[ ] = {"a", "b", "c", "d", "e", "f"};
       char   *key = "c";          /* 検索キーは“c”*/
       char   **ret;

       ret =(char **)bsearch((char *)&key,(char *)base, 6,
       sizeof(char *), compar);   /* ret に“c”へのポインタが格納されます */
}

int compar(char **x, char **y)
{
       return(strcmp(*x, *y));     /* 引数を比較して正，ゼロ，または負の整数を返します */
}
```

DIV

〔概要〕

除算

div，ldiv

〔形式〕

```
#include <stdlib.h>

div_t  div(int n, int d)
ldiv_t ldiv(long n, long d)
```

〔説明〕

div(int n, int d)

分子 n を分母 d で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 div_t のメンバとして格納します。

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

quot は商で，rem は剰余です。d がゼロでない場合，“ r = div(n, d); ” であれば，n は “ r.rem+d*r.quot ” に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が n と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

ldiv(long n, long d)

int 型の値ではなく long 型の値を除算する場合に使用します。結果は，次に示す構造体 ldiv_t のメンバとして格納します。

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

〔戻り値〕

除算の結果を格納した構造体を返します。

〔記述例〕

```
#include <stdlib.h>

void func(void)
{
       div_t  r;

       r = div(110, 3);     /* r.quot には 36，r.rem には 2 が格納されます */
}
```

ECVTF

〔概要〕

浮動小数点数から文字列への変換

ecvtf，fcvtf，gcvtf

〔形式〕

```
#include <stdlib.h>

char   *ecvtf(float val, int chars, int *decpt, int *sgn)
char   *fcvtf(float val, int decimals, int *decpt, int *sgn)
char   *gcvtf(float val, int prec, char *buf)
```

〔説明〕

ecvtf(float val, int chars, int *decpt, int *sgn)

float 型数値 val を数字で表した（null 文字（\0）で終端する）文字列を生成します。2 番目の引数 chars には，書き込む総文字数を指定します（数字のみを書き込むので変換された文字列の中の有効数字の数でもあります）。常に，val の整数部の桁がすべて含まれます。

fcvtf(float val, int decimals, int *decpt, int *sgn)

2 番目の引数の解釈以外は ecvt と同じです。2 番目の引数 decimals には，小数点後に書き込む文字の数を指定します。ecvtf と fcvtf は出力文字列の中に数字だけを書き込むので，小数点の位置を *decpt に，数値の符号を *sgn に記録しておきます。数をフォーマットしたあと，*decpt には小数点の左側の桁数が入ります。*sgn には，数値が正の場合は 0 が，負の場合は 1 が入ります。

gcvtf(float val, int prec, char *buf)

数値を文字列に書式変換し，バッファ buf に格納します。gcvtf は，sprintf の書式“ %.prec ”（負数だけに符号が付く）と同じ規則を使用し，有効桁数（prec で指定）に応じて，指数形式か，または通常の小数形式を選択します。

〔戻り値〕

ecvtf, fcvtf	val の文字列表現を含む新しい文字列を指すポインタを返します。
gcvtf	val の書式付き文字列表現へのポインタ（引数 buf と同じです）を返します。

〔記述例〕

```
#include <stdlib.h>

void func(void)
{
       float  val;
       int    dec, sgn;

       val = 111.11;
       ecvtf(val, 12, &dec, &sgn); /* val の値 111.11 を 12 文字の文字列へ変換します */
                                   /* dec には小数点の左側の桁数 3，*/
                                   /* sgn には符号（正のため 0）が記録されます  */
}
```

ITOA

〔概要〕

整数から文字列への変換

itoa，ltoa，ultoa

〔形式〕

```
#include <stdlib.h>

char   *itoa(int value, char *string, int radix)
char   *ltoa(long int value, char *string, int radix)
char   *ultoa(unsigned long int value, char *string, int radix)
```

〔説明〕

itoa(int value, char *string, int radix)

int 型数値 value を radix 進数の文字列に変換して，string の示す配列に格納します。文字列の終わりには終端を示す null 文字（\0）が常に付加されます。radix に指定できるのは，2 から 36 までの数値です。radix が 10 の場合，value は符号付き数値として扱われ，value ＜ 0 の場合文字列の先頭に ’ - ’ 文字が付きます。その他の場合，value は符号なし数値として扱われます。radix ＞ 10 の場合，10 から 35 に英小文字の a から z が当てられます。

ltoa(long int value, char *string, int radix)

long int 型数値 value を radix 進数の文字列に変換して，string の示す配列に格納します。value の型を除き，itoa と同じです。

ultoa(unsigned long int value, char *string, int radix)

unsigned long int 型数値 value を radix 進数の文字列に変換して，string の示す配列に格納します。value の型を除き，itoa と同じです。

〔戻り値〕

itoa, ltoa, ultoa	string を返します。

〔記述例〕

```
#include <stdlib.h>

void func(void)
{
       char   buf[128];

       itoa(12345, buf, 16);      /*12345 を 16 進数文字列に変換します */
}
```

MALLOC

〔概要〕

メモリ割り付けと管理

calloc，free，malloc，realloc

〔形式〕

```
#include <stdlib.h>

void   *calloc(size_t nmemb, size_t size)
void   free(void *ptr)
void   *malloc(size_t size)
void   *realloc(void *ptr, size_t size)
```

〔説明〕

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，コンパイラは自動でこの領域を確保しないため，calloc, malloc, realloc の各関数を利用する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

ヒープ・メモリ設定例

```
#define SIZEOF_HEAP 0x1000
int    __sysheap[SIZEOF_HEAP>>2];
size_t __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1 変数“ __sysheap ”（アンダースコア‘ _ ’は 2 つ）のシンボル“ ___sysheap ”（アンダースコア‘ _ ’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，ワード整数値にしてください。

備考 2 変数“ __sizeof_sysheap ”（最初のアンダースコア‘ _ ’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“ ___sizeof_sysheap ”（最初のアンダースコア‘ _ ’は 3 つ）に設定してください。

calloc(size_t nmemb, size_t size)

大きさが size の，要素数 nmemb 個の配列領域を割り付けます。割り付けられた領域は 0 で初期化されます。

free(void *ptr)

ptr が指す領域を開放し，その後の割り付けに使用できるようにします。ptr には，calloc，malloc，および realloc で獲得した領域を指定しなければなりません。

malloc(size_t size)

大きさ size の領域を割り付けます。領域は初期化されません。

realloc(void *ptr, size_t size)

ptr が指す領域の大きさを，size の大きさに変更します。以前の大きさと，size の小さい方までの領域の内容は変わりません。領域を拡張する場合の，以前の大きさ以降の領域内容は初期化されません。ptr が null ポインタのときは，malloc(size) と同じ動作をします。それ以外の場合，ptr には，calloc，malloc，および realloc で獲得した領域を指定しなければなりません。

〔戻り値〕

calloc, malloc, realloc	領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

〔記述例〕

```
#include <stdlib.h>

typedef struct {
       double d[3];
       int    i[2];
} s_data;

int func(void)
{
       sdata *buf;
       int    i;

       /* s_data40 個のための領域を割り付けます */
       if((buf = calloc(40, sizeof(s_data)))== NULL) return(1);

       for(i = 0; i < 40; i++)
       {

       }

       /* 領域を開放します */
       free(buf);

       return(0);
}
```

〔注意事項〕

calloc, free, malloc, realloc **関数で獲得・開放するメモリ領域は，ヒープ領域と呼ばれますが，このヒープ領域を確保する領域／大きさをあらかじめ設定しておく必要があります。**C **言語ソース・プログラムで設定する場合は“ヒープ・メモリ設定例”のように記述しますが，アセンブラ命令で記述する場合には，スタート・アップ・モジュールなどに次のようなプログラムを追加してください。**C **言語ソース・プログラムとアセンブラ命令で同時に指定するとエラーになりますので，どちらか一方を記述してください。**

```
#-------------------------------------------------
#       system heap
#-------------------------------------------------
       .set   HEAPSIZE, 0x1000
       .globl __sysheap
       .bss
       .lcomm __sysheap, HEAPSIZE, 4
       .data
       .globl __sizeof_sysheap
__sizeof_sysheap;
       .word HEAPSIZE
# この例では，ヒープ領域が .bss 領域に 0x1000 バイト確保されます。
```

RAND

〔概要〕

疑似乱数列生成

rand，srand

〔形式〕

```
#include <stdlib.h>

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

〔説明〕

rand()

0 以上 RAND_MAX 以下の乱数を返します。

srand(unsigned int seed)

後続する rand の呼び出しで使用する新しい疑似乱数列の種として，seed を与えます。srand を同じ seed の値で呼んだ場合，rand により得られる乱数は，同じ値が同じ順番で現れることになります。srand を実行せずに rand を実行した場合，最初に srand(1) を実行した場合と同じ結果になります。

〔戻り値〕

rand	乱数を返します。

〔記述例〕

```
#include <stdlib.h>

void func(void)
{
       if(rand()& 0xf)<4) func1(); /*25% の確率で func1 を実行します */
}
```

STRTODF

〔概要〕

文字列から浮動小数点数への変換

atoff，strtodf

〔形式〕

```
#include <stdlib.h>

float  atoff(const char *str)
float  strtodf(const char *str, char **p)
```

〔説明〕

atoff(const char *str)

str の指す文字列の最初の部分を float 型の表現に変換します。atoff は，“ strtodf(str, NULL) ” と同じです。

strtodf(const char *str, char **p)

str の指す文字列の最初の部分を long 型の表現に変換します。変換される部分文字列は，次の形式の，空白でない通常の文字から始まる，str の最長先頭部分文字列です。

```
[+|-]digits[.][digits][(e|E)[+|-]digits]
```

str が空か，または空白文字だけから成り立っている場合，および最初の通常文字が “ + ，“ - ”，“ . ”，または数字以外の場合，部分文字列には文字が含まれていません。部分文字列が空の場合，変換は行われず，str の値が ptr の指す領域に格納されます。空でない場合，部分文字列は変換され，最終文字列（少なくとも str の終端を示す null 文字（\0）を含む）へのポインタが ptr の指す領域に格納されます。なお，この関数は，リエントラントではありません。

〔戻り値〕

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

〔記述例〕

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

void func(float ret)
{
       char   *p, *str, s[30];

       str = "+5.32a4e";
       ret = strtodf(str, &p);         /* ret には 5.320000 が返され，p の領域には*/
                                       /*‘a’へのポインタが格納されます */
       sprintf(s, "%lf\t%c", ret, *p);/* s の指す配列には“5.320000 a”が格納されます */
}
```

STRTOL

〔概要〕

文字列から整数への変換

atoi，atol，strtol，strtoul

〔形式〕

```
#include <stdlib.h>

int    atoi(const char *str)
long   atol(const char *str)
long   strtol(const char *str, char **ptr, int base)
unsigned long strtoul(const char *str, char **ptr, int base)
```

〔説明〕

atoi(const char *str)

str の指す文字列の最初の部分を int 型の表現に変換します。atoi は，“ (int) strtol(str,NULL, 10) ”と同じです。

atol(const char *str)

str の指す文字列の最初の部分を long int 型の表現に変換します。atol は，“ strtol(str, NULL, 10) ”と同じです。

strtol(const char *str, char **ptr, int base)

str の指す文字列の最初の部分を long 型の表現に変換します。まず，strtol は，入力文字を次の 3 つの部分，“ 最初の空白類 ”, “ base の値により定められる基数において表現され，整数にする対象となる列 ”, “ (null 文字（\0）を含む）最後の 1 個以上の認識されない文字列 ” に分割します。その後，strtol は対象となる列を整数へ変換し，その結果を返します。

(1) 引数 base は，0，または 2 ～ 36 を指定します。

(a) base が 0 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号，16 進数であることを示す “ 0x ” を前に持つ整数の形式となります。

(b) base の値が 2 ～ 36 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号を前に持ち，base によって基数が指定された整数を表す文字列，または数字列となります。“ a ” (，または “ A ”) から “ z ” (，または “ Z ”) までの英字は 10 から 35 までの値を示すものとみなされます。与えられた値が base よりも小さい英字しか使用できません。

(c) base の値が 16 の場合

“ 0x ” が文字と数字の列の前（符号が存在する場合は符号の後ろ）に置かれます（省略可能）。

(2) 対象となる列は，空白類以外の最初の文字で始まり，予期される形式を持つ入力文字列の先頭部分の最長の部分列として定義されます。

(a) 入力の文字列が空である場合やすべて空白類で構成されている場合，または空白類でない最初の文字が符号でも許容されうる文字でも数字でもない場合，対象となる列は空となります。

(b) 対象となる列が予期される形式を持ち，かつ，base の値が 0 の場合，入力文字列から基数を判断します。0x が先行する文字列は，16 進数数値とみなされ，先行 0 が付いていて x が付いていない文字列は 8 進数としてみなされます。他の文字列はすべて 10 進数としてみなされます。

(c) base が 2 から 36 までの間の値の場合，上述のように，これを変換用基数として使用します。

(d) 対象となる列が - 符号で始まる場合，変換結果の値の符号は反転されます。

(3) 最初の文字列を指すポインタ

(a) ptr が null ポインタでない場合，ptr の指すオブジェクトの中に格納されます。

(b) 対象となる列が空である場合，または予期された形式を持たない場合，変換は行われません。str の値は，ptr が null ポインタでない場合，ptr の指すオブジェクトに格納されます。

なお，この関数は，リエントラントではありません。

strtoul(const char *str, char **ptr, int base)

返却値の型が unsigned long 型になること以外，strtol と同じです。

〔戻り値〕

atoi, atol	部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。
strtol	部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。 オーバーフローが生じる（変換された値の大きさが大きすぎる）場合，LONG_MAX，または LONG_MIN を返し，マクロ ERANGE をグローバル変数 errno にセットします。
strtoul	部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。 オーバーフローが生じる場合，ULONG_MAX を返し，マクロ ERANGE をグローバル変数 errno にセットします。

〔記述例〕

```
#include <stdlib.h>

void func(long ret)
{
       char   *p;

       ret = strtol("10", &p, 0);        /* ret には 10 が返されます */
       ret = strtol("0x10", &p, 0);      /* ret には 16 が返されます */

       ret = strtol("10x", &p, 2);       /* ret には 2 が返され，p の領域には ‘x’ への */
                                         /* ポインタが格納されます */
       ret = strtol("2ax3", &p, 16);     /* ret には 42 が返され，p の領域には ‘x’ への */
                                         /* ポインタが格納されます */
             :
}
```

6.10 非局所分岐

ここでは，非局所分岐機能について説明します。これらの関数に関する宣言，および定義は，“ setjmp.h ”ファイルに記述されています。

表 6 － 23 非局所分岐関数／マクロ一覧

分類	関数名	概要
SETJMP	setjmp	非局所分岐の分岐先をセット
	longjmp	非局所分岐

SETJMP

〔概要〕

非局所分岐

setjmp, longjmp

〔形式〕

```
# include <setjmp.h>

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

〔説明〕

setjmp(jmp_buf env)

非局所分岐のための戻り先を env にセットします。env にはその他，setjmp が実行された時点の環境が保存されます。

longjmp(jmp_buf env, int val)

setjmp で保存された env を使い，setjmp 直後へ非局所分岐します。val は，setjmp の返却値となります。

〔戻り値〕

setjmp	setjmp からの戻りの場合 0 を返します。longjmp によって非局所分岐の場合，longjmp の第二引数 val を返します。ただし，val が 0 の場合，1 を返します。

〔記述例〕

```
#include <setjmp.h>

#define ERR_XXX1    1
#define ERR_XXX2    2

jmp_buf jmp_env;

void func(void)
{
       for(;;) {
              switch(setjmp(jmp_env))
              {
              case ERR_XXX1:
                     /* error XXX1 の終結処理 */
                     break;
              case ERR_XXX2:
                     /* error XXX2 の終結処理 */
                     break;
              case 0: /* 非局所分岐ではない */
              default:
                     break;
              }
       }
}

void func1(void)
{
       longjmp(jmp_env, ERR_XXX1); /* error XXX1 の発生により非局所分岐する */

       longjmp(jmp_env, ERR_XXX2); /* error XXX2 の発生により非局所分岐する */
}
```

6.11 数学関数

ここでは，数学関数について説明します。これらの関数に関連する定義，および宣言は，“ math.h ”ファイルに記述されています。なお，数学ライブラリ libm. a は，内部で標準ライブラリ libc. a を参照しています。

したがって，ld850 を単独で起動して libm. a を参照する場合，libc. a も同時に参照しなければなりません。また，ld850 を単体で起動してアーカイブ・ファイルを複数参照する場合，未定義シンボルは，参照を指定した順序に従って検索されるため，指定順序は，libc. a の参照“ -lc ”を後ろにしてください。

ただし，コンパイラから ld850 を起動する場合は，libc. a ファイルを自動的に参照します。

表 6 － 24 数学関数

分類	関数名	概要
BESSEL	j0f	第一種ベッセル関数（0 次）
	j1f	第一種ベッセル関数（1 次）
	jnf	第一種ベッセル関数（n 次）
	y0f	第二種ベッセル関数（0 次）
	y1f	第二種ベッセル関数（1 次）
	ynf	第二種ベッセル関数（n 次）
ERFF	erff	誤差関数（近似値）
	erfcf	誤差関数（相補確率）
EXPF	expf	指数関数
	logf	対数関数（自然対数）
	log2f	対数関数（2 を底）
	log10f	対数関数（10 を底）
	powf	べき乗関数
	cbrtf	立方根関数
	sqrtf	平方根関数
FLOORF	ceilf	ceiling 関数
	fabsf	絶対値関数
	loorf	floor 関数
	fmodf	剰余関数
FREXPF	frexpf	浮動小数点数を仮数部とべき乗に分割
	ldexpf	浮動小数点数をべき乗に変換
	modff	浮動小数点数を整数部と小数部に分類
GAMMAF	gammaf	対数ガンマ関数
HYPOTF	hypotf	ユークリッド距離関数
MATHERR	matherr	エラー処理関数

表 6 － 24 数学関数

分類	関数名	概要
SINHF	acoshf	逆双曲線余弦
	asinhf	逆双曲線正弦
	atanhf	逆双曲線正接
	coshf	双曲線余弦
	sinhf	双曲線正弦
	tanhf	双曲線正接
TRIG	acosf	逆余弦
	asinf	逆正弦
	atanf	逆正接
	atan2f	逆正接（y/x）
	cosf	余弦
	sinf	正弦
	tanf	正接

BESSEL

〔概要〕

ベッセル関数

j0f，j1f，jnf，y0f，y1f，ynf

〔形式〕

```
#include <math.h>

float  jnf(int n, float x)
float  j0f(float x)
float  j1f(float x)
float  ynf(int n, float x)
float  y0f(float x)
float  y1f(float x)
```

〔説明〕

ベッセル関数とは，次に示す微分方程式の解となる関数をいいます。

$$x^2\frac{d^2y}{dx^2}+\frac{dy}{dx}+(x^2-p^2)y=0$$

jnf(int n, float x)

n 次の第一種ベッセル関数値を求めます。

j0f(float x)

0 次の第一種ベッセル関数値を求めます。

j1f(float x)

1 次の第一種ベッセル関数値を求めます。

ynf(int n, float x)

n 次の第二種ベッセル関数値を求めます。

y0f(float x)

0 次の第二種ベッセル関数値を求めます。

y1f(float x)

1 次の第二種ベッセル関数値を求めます。

〔戻り値〕

jnf	n 次の第一種ベッセル関数値を返します。
j0f	0 次の第一種ベッセル関数値を返します。
j1f	1 次の第一種ベッセル関数値を返します。
ynf	n 次の第二種ベッセル関数値を返します。
y0f	0 次の第二種ベッセル関数値を返します。
y1f	1 次の第二種ベッセル関数値を返します。

〔記述例〕

```
#include <math.h>

float func(void)
{
       float  ret, x;

       ret = j1f(x);        /* xの値に対して，1次の第一種ベッセル関数値を求め，retに返します */
             :
       return(ret);
}
```

ERFF

〔概要〕

誤差関数

erff，erfcf

〔形式〕

```
#include <math.h>

float  erff(float x)
float  erfcf(float x)
```

〔説明〕

erff(float x)

観測値が平均の x 標準偏差の範囲になる確率を推定する“誤差関数”の近似値（0 と 1 の間の数値）を求めます。誤差関数の定義式は，次のとおりです。

$$\frac{2}{\sqrt{\pi}} \times \int_0^x e^{-t^2} dt$$

erfcf(float x)

“1.0-erff(x)”をして相補確率を求めます。これは，値の大きな x について erff(x) が呼び出された場合，その結果を 1.0 から引かれると精度が損なわれるために用意されています。

〔戻り値〕

erff	“誤差関数”の近似値（0 と 1 の間の数値）を返します。
erfcf	相補確率を返します。

〔記述例〕

```
#include <math.h>

float func(void)
{
        float  ret, x;

        ret = erff(x);/* x の値に対して，誤差関数の近似値を求め，ret に返します */
              :
        return(ret);
}
```

EXPF

〔概要〕

指数／対数／べき／立方根／平方根関数

expf，logf，log2f，log10f，powf，cbrtf，sqrtf

〔形式〕

```
#include <math.h>

float  expf(float x)
float  logf(float x)
float  log2f(float x)
float  log10f(float x)
float  powf(float x, float y)
float  cbrtf(float x)
float  sqrtf(float x)
```

〔説明〕

expf(float x)

e の x 乗を求めます（e は自然対数の底で，約 2.71828 です）。

logf(float x)

x の自然対数，つまり，底を e としてその対数を求めます。

log2f(float x)

2 を底とする x の対数を求めます。これは “ log(x)/log(2) ” によって実現されています。

log10f(float x)

10 を底とする x の対数を求めます。これは，“ log(x)/log(10) ” によって実現されています。

powf(float x, float y)

x の y 乗を求めます。

cbrtf(float x)

x の立方根を求めます。

sqrtf(float x)

x の正の平方根を求めます。

〔戻り値〕

expf	e の x 乗を返します。 expf は，アンダフローが生じた場合（x が結果を表せない大きさの負の数の場合），非正規化数を返し，グローバル変数 errno にマクロ ERANGE をセットします。オーバーフローが生じた場合（x が大きすぎる数の場合），HUGE_VAL（表現可能な最大の double 型数値）を返し，グローバル変数 errno にマクロ ERANGE をセットします。
logf	x の自然対数を返します。 logf は，x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞（0xff800000）を返し，グローバル変数 errno にマクロ ERANGE をセットします。
log2f	2 を底とする x の対数を返します。 log2f は，x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。
log10f	10 を底とする x の対数を返します。 log10f は，x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。
powf	x の y 乗を返します。 powf は，x ＜ 0 かつ y が奇整数の場合にのみ負の解を返します。x ＜ 0 で y が非整数の場合，または x = y = 0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x = 0 かつ y ＜ 0 の場合，またはオーバーフローが発生した場合，± HUGE_VAL を返し，errno にマクロ ERANGE をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，errno に ERANGE をセットします。解が非正規化数の場合，errno に ERANGE をセットします。
cbrtf	x の立方根を返します。
sqrtf	x の正の平方根を返します。 sqrtf は，x が実数で負の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。

matherr 関数を使用して，これらの関数のエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

float func(void)
{
       float  ret, x, y;

       ret = powf(x, y)     /* x を y 乗した値を ret に返します */
             :
       return(ret);
}
```

FLOORF

〔概要〕

ceiling ／絶対値／ floor ／剰余関数

ceilf，fabsf，floorf，fmodf

〔形式〕

```
#include <math.h>

float  ceilf(float x)
float  fabsf(float x)
float  floorf(float x)
float  fmodf(float x, float y)
```

〔説明〕

ceilf(float x)

x 以上の最小の整数値を求めます。

fabsf(float x)

x のビット表現を直接操作して，x の絶対値（大きさ）を求めます。

floorf(float x)

x 以下の最大の整数値を求めます。

fmodf(float x, float y)

x を y で割った剰余である浮動小数点数値を求めます。つまり，y が 0 でない場合に，その結果の符号が x と同じ符号で大きさが y よりも小さい最大整数 i に対し，値“x - i × y”を求めます。

〔戻り値〕

ceilf	x 以上の最小の整数値を返します。
fabsf	x の絶対値（大きさ）を返します。
floorf	x 以下の最大の整数値を返します。
fmodf	x を y で割った剰余である浮動小数点数値を返します。 fmodf(x, 0) は x を返します。

matherr 関数を使用して，これらの関数のエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

void func(void)
{
       float  ret, x, y;

       ret = fmodf(x, y);  /* x を y で割った剰余を ret に返します */
             :
}
```

FREXPF

〔概要〕

浮動小数点数の各部分の操作

frexpf，ldexpf，modff

〔形式〕

```
# include <math.h>

float  frexpf(float val, int *exp)
float  ldexpf(float val, int exp)
float  modff(float val, float *ipart)
```

〔説明〕

0 以外の数はすべて，m × 2^p で表すことができます。

frexpf(float val, int *exp)

float 型の val を仮数部 m と 2 の p 乗で表します。結果の仮数部 m は，val が 0 でないかぎり，0.5 ≤ |x| < 1.0 となります。p は *exp に格納されます。m，および p は，val = m × 2^p となるように計算されます。

ldexpf(float val, int exp)

val × 2^{exp} を求めます。

modff(float val, float *ipart)

float 型の val を整数部と小数部とに分割し，整数部を *ipart に格納します。丸めは行いません。整数部と小数部の和は，正確に val と一致するように保証されています。たとえば，realpart = modff(val, &intpart) であるとき，“ realpart+intpart ” は val と一致します。

〔戻り値〕

frexpf	仮数部 m を返します。 frexpf は，val が 0 の場合，*exp に 0 をセットし，0 を返します。
ldexpf	val × 2^{exp} で求めた値を返します。 ldexpf でアンダフロー，またはオーバーフローが生じた場合，グローバル変数 errno にマクロ ERANGE がセットされます。アンダフローの場合，ldexpf は非正規化数を返します。オーバーフローの場合，HUGE_VAL と同じ符号∞（+ ∞ =0x7f800000，- ∞ =0xff800000）を返します。 matherr 関数を使用して，このエラー処理を変更することができます。
modff	小数部を返します。結果の符号は val の符号と同じです。

〔記述例〕

```
#include <math.h>

void func(void)
{
       float  ret, x;
       int    exp;

       x = 5.28;

       ret = frexpf(x, &exp);   /* 結果の仮数部 0.66 が ret に返り exp には 3 倍が格納されます */
             :
}
```

GAMMAF

〔概要〕

対数ガンマ関数

gammaf

〔形式〕

```
# include <math.h>

float  gammaf(float x)
```

〔説明〕

gammaf(float x)

ln(Γ(x))，つまり，x のガンマ関数の自然対数を求めます。ガンマ関数 (expf(gammaf(x))) は階乗の一般化であり，Γ(N) ≡ N × Γ(N - 1) という関係式を持っています。したがって，ガンマ関数自体の結果は非常に早く大きくなります。そのため，gammaf は，表現可能な結果の有効範囲を拡大するために，単なる“ Γ(x) ”ではなく“ ln(Γ(x)) ”として定義されています。

〔戻り値〕

x のガンマ関数の自然対数を返します。

x が 0 のとき，またはオーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にマクロ ERANGE をセットします。matherr 関数を使用して，このエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

float func(float x)
{
       float  ret;

       ret = gammaf(x);     /* x のガンマ関数の自然対数を ret に返します */
             :
       return(ret);
}
```

HYPOTF

〔概要〕

ユークリッド距離関数

hypotf

〔形式〕

```
# include <math.h>

float  hypotf(float x, float y)
```

〔説明〕

hypotf(float x, float y)

原点（0, 0）とカーテシアン座標（x, y）で表される点との間のユークリッド距離 $\sqrt{x^2+y^2}$ を求めます。

〔戻り値〕

原点（0, 0）とカーテシアン座標（x, y）で表される点との間のユークリッド距離 $\sqrt{x^2+y^2}$ を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

matherr 関数を使用して，このエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

void func(float x)
{
       float  ret, y;

       ret = hypotf(x, y); /* （0, 0）座標と（x, y）座標の間のユークリッド距離を ret に返します */
}
```

MATHERR

〔概要〕

エラー処理関数

matherr

〔形式〕

```
#include <math.h>

int    matherr(struct exception *e)
```

〔説明〕

matherr(struct exception *e)

数学ライブラリ関数内でエラーが発生した場合に呼ばれる関数です。

したがって，matherr という名前の関数をユーザ・サブルーチンで用意することにより，エラー処理をカスタマイズできます。カスタマイズする matherr は，エラーの解決に失敗した場合に 0 を返し，エラーを解決した場合に 0 以外の値を返すようにする必要があります。matherr が 0 以外の値を返した場合，グローバル変数 errno の値は変更されません。

エラー処理のカスタマイズは，構造体 exception へのポインタ *e で渡された情報を利用して行うことができます。構造体 exception は “ math.h ” 中で次のように定義されています。

```
#if !defined(__cplusplus)
#define __exception exception
#endif
struct exception{
       int    type;
       char   *name;
       double arg1, arg2, retval;
};
```

各メンバの意味は，次のとおりです。

type	発生した数学関数エラーのタイプです。 マクロ・エンコーディング・エラーのタイプも “ math.h ” の中で定義されています。
name	エラーが発生した数学ライブラリ関数の名前を保持し，空文字で終わっている文字列を指すポインタです。
arg1, arg2	エラーの原因となった引数です。
retval	呼び出し関数が返すエラー・リターン値です。

発生する可能性のある数学ライブラリ関数エラーのタイプは，次のとおりです。

DOMAIN	引数が関数の定義域の範囲にない 例：logf(-1)
OVERFLOW	オーバーフロー 例：expf(1000)
UNDERFLOW	アンダフロー，非正規化数の解 解＜ 1.1755e -38 かつ非 0 の数で，精度が通常の数値より落ちた状態
Z_DIVISION	ゼロ除算

なお，演算例外発生時の matherr の呼び出しと，標準関数でのグローバル変数 errno の更新は，リエントラント性を持ちません。

〔戻り値〕

e->retval の値を変更することにより，カスタマイズした matherr からの呼び出し関数の結果を変更できます。これは，呼び出し側の関数にも伝播します。matherr は，エラーを解決した場合，0 以外の値を返し，エラーを解決できなかった場合，0 を返します。matherr が 0 を返す場合，呼び出し側でグローバル変数 errno に適切な値をセットします。

〔記述例〕

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

void func(void)
{
       float  ret;
       ret = logf(-0.1);    /* ret には 3 が返されます */
}

int matherr(struct exception *e)
{
       char   s[30];

       switch(e->type) {
              case DOMAIN:
                     sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
                     e->retval = 3; /* エラー・リターン値を 3 に変更します */
                     break;
              default:
                     sprintf(s, "%s other error %e\n", e->name, e->arg1);
       }
       return(1);
}
```

SINHF

〔概要〕

双曲線関数

acoshf，asinhf，atanhf，coshf，sinhf，tanhf

〔形式〕

```
#include <math.h>

float  acoshf(float x)
float  asinhf(float x)
float  atanhf(float x)
float  coshf(float x)
float  sinhf(float x)
float  tanhf(float x)
```

〔説明〕

acoshf(float x)

x（x は 1 以上の数値）の逆双曲線余弦を求めます。定義式は次のとおりです。

$$\ln(x + \sqrt{x^2 - 1})$$

asinhf(float x)

x の逆双曲線正弦を求めます。定義式は次のとおりです。

$$\text{sign}(x) \times \ln(|x| + \sqrt{1 + x^2})$$

atanhf(float x)

x の逆双曲線正接を求めます。

coshf(float x)

x の双曲線余弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$\frac{(e^{x} + e^{-x})}{2}$$

sinhf(float x)

x の双曲線正弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$\frac{(e^{x} - e^{-x})}{2}$$

tanhf(float x)

x の双曲線正接を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

```
sinh(x) / cosh(x)
```

〔戻り値〕

acoshf	x（x は 1 以上の数値）の逆双曲線余弦を返します。 acoshf は，x が 1 より小さい場合，非数を返します。また，グローバル変数 errno にはマクロ EDOM をセットします。
asinhf	x の逆双曲線正弦を返します。
atanhf	x の逆双曲線正接を返します。 atanhf において，x の絶対値が 1 以上の場合，非数を返し，グローバル変数 errno にはマクロ EDOM をセットします。
coshf	x の双曲線余弦を返します。 coshf は，オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。
sinhf	x の双曲線正弦を返します。 sinhf は，オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。
tanhf	x の双曲線正接を返します。

matherr 関数を使用して，これらの関数のエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

float func(float x)
{
       float  ret;

       ret = acoshf(x);     /* x の逆双曲線余弦の値を ret に返します */
             :
       return(ret);
}
```

TRIG

〔概要〕

三角関数

acosf，asinf，atanf，atan2f，cosf，sinf，tanf

〔形式〕

```
#include <math.h>

float  acosf(float x)
float  asinf(float x)
float  atanf(float x)
float  atan2f(float y, float x)
float  cosf(float x)
float  sinf(float x)
float  tanf(float x)
```

〔説明〕

acosf(float x)

x の逆余弦（アークコサイン）を求めます。x は，-1 ≦ x ≦ 1 で指定します。

asinf(float x)

x の逆正弦（アークサイン）を求めます。x は，-1 ≦ x ≦ 1 で指定します。

atanf(float x)

x の逆正接（アークタンジェント）を求めます。

atan2f(float y, float x)

y/x の逆正接（アークタンジェント）を求めます。atan2f は，π/2，または -π/2 付近（x が 0 に近い場合）の角度の場合も正しい結果を求めます。

cosf(float x)

x の余弦を求めます。角度はラジアン単位で指定します。

sinf(float x)

x の正弦を求めます。角度はラジアン単位で指定します。

tanf(float x)

x の正接を求めます。角度はラジアン単位で指定します。

〔戻り値〕

acosf	x の逆余弦（アークコサイン）を返します。返す値はラジアン単位で，0 から π までの範囲です。 x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットます。
asinf	x の逆正弦（アークサイン）を返します。返す値はラジアン単位で，-π/2 から π/2 までの範囲です。 x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットします。
atanf	x の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π/2 から π/2 の範囲です。
atan2f	y/x の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π から π までの範囲です。 atan2f は，x と y がともに 0.0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，グローバル変数 errno にマクロ ERANGE をセットします。解が非正規化数の場合，グローバル変数 errno に ERANGE をセットします。
cosf	x の余弦を返します。
sinf	x の正弦を返します。
tanf	x の正接を返します。

matherr 関数を使用して，これらの関数のエラー処理を変更できます。

〔記述例〕

```
#include <math.h>

float func(float x)
{
       float  ret;

       ret = atanf(x);      /* x の逆正接の値を ret に返します */
             :
       return(ret);
}
```

6.12 ランタイム・ライブラリ

ここでは，ランタイム・ライブラリについて説明します。V850 マイクロコントローラ のアーキテクチャでは，32 ビット・データの乗算，除算や浮動小数点演算などの命令を持ちません。そこで CA850 では，ANSI 規格の言語仕様を満たすため，32 ビット・データの整数の乗算，除算，剰余算，およびすべての浮動小数点演算を，libc.a ファイルに含まれるランタイム・ライブラリを呼び出して行います。

また，V850 マイクロコントローラ用の新規のアセンブリ言語ソースを作成する際に，ランタイム・ライブラリを呼び出すこともできます。ただし，V850E では 32 ビット・データの乗算命令があるため，CA850 は 32 ビット・データの乗算，除算，剰余算についてランタイム・ライブラリを使用しません。浮動小数点演算のランタイム・ライブラリは使用します。

ランタイム・ライブラリは，CA850 がコンパイル時に自動的に使用するルーチンです（「8. 2 提供ライブラリ」を参照）。標準ライブラリとともに，libc.a ファイルに含まれています。ヘッダ・ファイルのインクルードは必要ありません。

なお，ランタイム・ライブラリをアプリケーション・プログラムで使用する場合，実行可能なオブジェクト・ファイル作成時に，ld850 で libc.a を参照する必要があります。

図 6 － 1 ランタイム・ライブラリ使用イメージ その 2

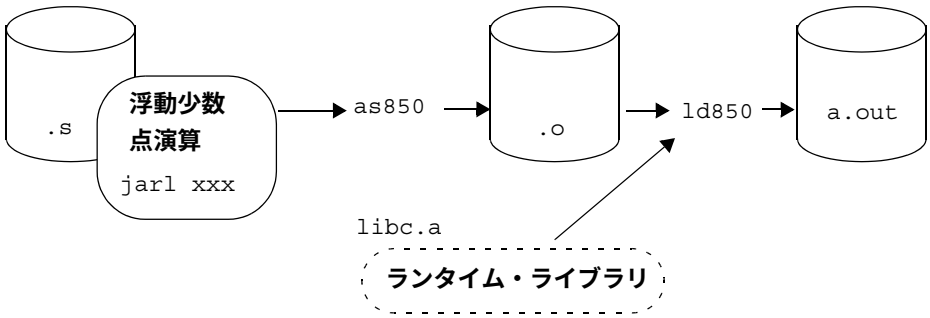


表 6 － 25 ランタイム・ライブラリ

分類	関数名	概要
ADDF.S	___addf.s	単精度浮動小数点の加算
CMPF.S	___cmpf.s	単精度浮動小数点の比較，およびフラグの変更
CVT.WS	___cvt.ws	整数から単精度浮動小数点への変換
DIV	___div	符号付き 32 ビット整数の除算
	___divu	符号なし 32 ビット整数の除算
DIVF.S	___divf.s	単精度浮動小数点の除算
MOD	___mod	符号付き 32 ビット整数の剰余算
	___modu	符号なし 32 ビット整数の剰余算
MUL	___mul	符号付き 32 ビット整数の乗算
	___mulu	符号なし 32 ビット整数の乗算
MULF.S	___mulf.s	単精度浮動小数点の乗算
SUBF.S	___subf.s	単精度浮動小数点の減算
TRNC.SW	___trnc.sw	単精度浮動小数点数から整数への変換

〔注意事項〕

(1) ランタイム・ライブラリは，本来，コード生成部（cgen）が使用するものであり，単体で使用することを前提としていません。したがって，アセンブリ言語ソースで使用する場合，ランタイム・ライブラリを呼び出すための前処理が必要です。

(2) ランタイム・ライブラリは，C 言語ソース・プログラムでは使用できません。

(3) CA850 のデフォルト処理では，16 ビット・データ以下の整数に対しては，ランタイム・ライブラリのうち，乗算の ___mul ／ ___mulu 関数，および除算の ___div ／ ___divu 関数は利用されず，それぞれ，mulh，divh 命令が利用されます。コンパイラで -Xe オプションを指定した場合，16 ビット・データ以下の整数に対してもランタイム・ライブラリが利用されます。
この場合，ランタイム・ライブラリを利用すると，ANSI 規格に厳密に従った乗除算処理を行いますが，mulh，divh 命令で行うよりも実行速度が遅くなります。

ADDF.S

〔概要〕

単精度浮動小数点の加算

___addf.s

〔形式〕

```
jarl   ___addf.s, lp
```

〔説明〕

___addf.s

単精度浮動小数点の加算を行います。この関数は，単精度浮動小数点数における，C 言語の算術演算子“ + ”の実体として，ca により利用されます。整数の加算には利用されません。

〔前処理〕

アセンブラ命令で本関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

加算結果を r6 に設定します。

〔記述例〕

“ reg2 の値 +reg1 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6,[sp]              -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6             -- 引数として値を代入します
mov    reg2, r7
jarl   ___addf.s, lp        -- 関数を呼び出します
mov    r6, reg2             -- 加算結果を reg2 に格納します
ld.w   4[sp], r7            -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

CMPF.S

〔概要〕

単精度浮動小数点の比較，およびフラグの変更

___cmpf.s

〔形式〕

```
jarl   ___cmpf.s, lp
```

〔説明〕

___cmpf.s

単精度浮動小数点の比較を行い，その結果に従って，フラグ S，および Z を変更します。渡された PSW にフラグの変更を反映させ，PSW を変更します。

〔前処理〕

アセンブラ命令でこの関数を利用する場合，汎用レジスタ r6-r8 を退避し，r6 と r7 に引数として値を代入する必要があります。また，PSW の値を r8 に渡す必要があります。

この関数は，“ r7 - r6 ” の結果によりフラグを変更します。

〔引数設定用レジスタ〕

r6, r7, r8

〔戻り値〕

比較結果により PSW を変更し，その値を r6 に設定します。

〔フラグ〕

CY	比較結果が負の場合 1，そうでない場合 0
OV	0
S	比較結果が負の場合 1，そうでない場合 0
Z	比較結果が 0 の場合 1，そうでない場合 0

〔記述例〕

“ reg2 の値と reg1 の値の比較 ”（reg2 は r6 ～ r8 以外）の場合

```
        add    -12, sp
        st.w   r6, [sp]             -- r6-r8 を退避します
        st.w   r7, 4[sp]
        st.w   r8, 8[sp]
        mov    reg1, r6             -- 引数として値を代入します
        mov    reg2, r7
        stsr   5, r8                -- r8 に PSW の値を渡します
        jarl   ___cmpf.s, lp        -- 関数を呼び出します
        mov    r6, reg2             -- 変更した PSW の値を reg2 に格納します
        ld.w   8[sp], r8            -- r6-r8 を復帰します
        ld.w   4[sp], r7
        ld.w   [sp], r6
        add    12, sp
```

CVT.WS

〔概要〕

整数から単精度浮動小数点数への変換

___cvt.ws

〔形式〕

```
jarl   ___cvt.ws, lp
```

〔説明〕

___cvt.ws

整数から単精度浮動小数点数への変換を行います。

〔前処理〕

アセンブラ命令でこの関数を利用する場合，汎用レジスタ r6 を退避し，r6 に引数として変換する値を代入する必要があります。

〔引数設定用レジスタ〕

r6

〔戻り値〕

変換後の値を r6 に設定します。

〔記述例〕

“ reg1 の値を変換し reg2 へ格納 ”（reg2 は r6 以外）の場合

```
add    -4, sp
st.w   r6, [sp]            -- r6 を退避します
mov    reg1, r6            -- 引数として値を代入します
jarl   ___cvt.ws, lp       -- 関数を呼び出します
mov    r6, reg2            -- 変換後の値を reg2 に格納します
ld.w   [sp], r6            -- r6 を復帰します
add    4, sp
```

DIV

〔概要〕

32 ビット整数の除算

___div, ___divu

〔形式〕

```
jarl   ___div, lp
jarl   ___divu, lp
```

〔説明〕

___div

符号付き 32 ビット整数の除算を行います。

___divu

符号なし 32 ビット整数の除算を行います。これらの関数は，C 言語の算術演算子“ / ”の実体として，CA850 により利用されます。

〔前処理〕

アセンブラ命令でこれらの関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。これらの関数は，“ r7 / r6 ” とみなして除算を行います。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

除算結果の下位 32 ビットを r6 に設定します。剰余は無視します。

〔記述例〕

“ reg2 の値 /reg1 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6, [sp]      -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6      -- 引数として値を代入します
mov    reg2, r7
jarl   ___div, lp    -- 関数を呼び出します
mov    r6, reg2      -- 除算結果を reg2 に格納します
ld.w   4[sp], r7     -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

DIVF.S

〔概要〕

単精度浮動小数点の除算

___divf.s

〔形式〕

```
jarl   ___divf.s, lp
```

〔説明〕

___divf.s

単精度浮動小数点の除算を行います。この関数は，単精度浮動小数点数における，C 言語の算術演算子“ / ”の実体として，CA850 により利用されます。整数の除算には利用されません。

〔前処理〕

アセンブラ命令でこの関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。この関数は，“ r7 / r6 ” とみなして除算を行います。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

除算結果を r6 に設定します。

〔記述例〕

“ reg2 の値 /reg1 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6, [sp]            -- r6, r7 を退避します
st.w   r7, 4 [sp]
mov    reg1, r6            -- 引数として値を代入します
mov    reg2, r7
jarl   ___divf.s, lp       -- 関数を呼び出します
mov    r6, reg2            -- 除算結果を reg2 に格納します
ld.w   4[sp], r7           -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

MOD

〔概要〕

32 ビット整数の剰余算

___mod, ___modu

〔形式〕

```
jarl   ___mod, lp
jarl   ___modu, lp
```

〔説明〕

___mod

符号付き 32 ビット整数の剰余算を行います。

___modu

符号なし 32 ビット整数の剰余算を行います。これらの関数は，C 言語の算術演算子 “ % ” の実体として，CA850 により利用されます。

〔前処理〕

アセンブラ命令でこれらの関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。これらの関数は，“ r7 % r6 ” とみなして剰余算を行います。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

剰余算結果を r6 に設定します。

〔記述例〕

“ reg2 の値 % reg1 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6, [sp]      -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6      -- 引数として値を代入します
mov    reg2, r7
jarl   ___mod, lp    -- 関数を呼び出します
mov    r6, reg2      -- 剰余算結果を reg2 に格納します
ld.w   4[sp], r7     -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

MUL

〔概要〕

32 ビット整数の乗算

___mul, ___mulu

〔形式〕

```
jarl   ___mul, lp
jarl   ___mulu, lp
```

〔説明〕

___mul

符号付き 32 ビット整数の乗算を行います。

___mulu

符号なし 32 ビット整数の乗算を行います。これらの関数は，C 言語の算術演算子“ * ”の実体として，CA850 により利用されます。

〔前処理〕

アセンブラ命令でこれらの関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

乗算結果の下位 32 ビットを r6 に設定します。上位 32 ビットは無効となります。

〔記述例〕

“ reg1 の値× reg2 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6, [sp]      -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6      -- 引数として値を代入します
mov    reg2, r7
jarl   ___mul, lp    -- 関数を呼び出します
mov    r6, reg2      -- 乗算結果を reg2 に格納します
ld.w   4[sp], r7     -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

MULF.S

〔概要〕

単精度浮動小数点の乗算

___mulf.s

〔形式〕

```
jarl   ___mulf.s, lp
```

〔説明〕

___mulf.s

単精度浮動小数点の乗算を行います。この関数は，単精度浮動小数点数における，C 言語の算術演算子“ * ”の実体として，CA850 により利用されます。整数の乗算には利用されません。

〔前処理〕

アセンブラ命令でこの関数を利用する場合，汎用レジスタ r6, r7 を退避し，r6 と r7 に引数として値を代入する必要があります。

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

乗算結果を r6 に設定します。

〔記述例〕

“ reg2 の値× reg1 の値 ”（reg2 は r6, r7 以外）の場合

```
add    -8, sp
st.w   r6, [sp]            -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6            -- 引数として値を代入します
mov    reg2, r7
jarl   ___mulf.s, lp       -- 関数を呼び出します
mov    r6, reg2            -- 乗算結果を reg2 に格納します
ld.w   4[sp], r7           -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

SUBF.S

〔概要〕

単精度浮動小数点の減算

___subf.s

〔形式〕

```
jarl   ___subf.s, lp
```

〔説明〕

___subf.s

単精度浮動小数点の減算を行います。この関数は，単精度浮動小数点数における，C 言語の算術演算子“ _ ”の実体として，CA850 により利用されます。整数の減算には利用されません。

〔前処理〕

アセンブラ命令でこの関数を利用する場合，汎用レジスタ r6, r7 **を退避し，**r6 **と** r7 **に引数として値を代入する必要があります。この関数は，“** r7 - r6 **” とみなして減算を行います。**

〔引数設定用レジスタ〕

r6, r7

〔戻り値〕

減算結果を r6 **に設定します。**

〔記述例〕

“ reg2 **の値** - reg1 **の値 ”（**reg2 **は** r6, r7 **以外）の場合**

```
add    -8, sp
st.w   r6, [sp]            -- r6, r7 を退避します
st.w   r7, 4[sp]
mov    reg1, r6            -- 引数として値を代入します
mov    reg2, r7
jarl   ___subf.s, lp       -- 関数を呼び出します
mov    r6, reg2            -- 減算結果を reg2 に格納します
ld.w   4[sp], r7           -- r6, r7 を復帰します
ld.w   [sp], r6
add    8, sp
```

TRNC.SW

〔概要〕

単精度浮動小数点数から整数への変換

___trnc.sw

〔形式〕

```
jarl   ___trnc.sw, lp
```

〔説明〕

___trnc.sw

単精度浮動小数点数から整数への変換を行います。なお，変換時に 0 方向への丸めを行います。

〔前処理〕

アセンブラ命令で本関数を利用する場合，汎用レジスタ r6 を退避し，r6 に引数として変換する値を代入する必要があります。

〔引数設定用レジスタ〕

r6

〔戻り値〕

変換後の値を r6 に設定します。

〔記述例〕

“ reg1 の値を変換し reg2 へ格納 ”（reg2 は r6 以外）の場合

```
add    -4, sp
st.w   r6, [sp]            -- r6 を退避します
mov    reg1, r6            -- 引数として値を代入します
jarl   ___trnc.sw, lp      -- 関数を呼び出します
mov    r6, reg2            -- 変換後の値を reg2 に格納します
ld.w   [sp], r6            -- r6 を復帰します
add    4, sp
```

第 7 章　より効果的に用いるために

この章では，CA850 をより効果的に用いるためのプログラミング技法，および拡張機能の利用方法について説明します。

7.1　volatile 修飾子

volatile 宣言した変数に対しては，最適化およびレジスタ割り付けは行われません。volatile 宣言は必要最低限にしてください。

また，volatile 宣言した変数の値を使用する際，ある区間でその変数の値が外部から変更されないことが自明な場合，volatile 宣言されていない変数にその値を代入してその変数を参照することにより，その変数が最適化され，実行速度が向上する可能性があります。

7.2 戻り値がない関数の宣言

戻り値がない関数を void **型で宣言しない場合，無駄なリターン処理コードが生成されます。戻り値がない関数は必ず** void **型として宣言するようにしてください。**

7.3 ポインタと最適化

CA850 は，-Og ／ -O ／ -Os ／ -Ot オプションを指定して最適化を行う場合，ポインタの解析を実行しています。最適化レベルがそれより低い場合，ポインタの解析を実行しません。そのため，ポインタを用いた間接メモリ・アクセスが存在すると，それによってすべての変数がアクセスされたものと仮定して処理が行われ，最適化やレジスタ割り付けが効率良く実行されません。

また，サイズ優先最適化，または実行速度優先最適化の場合でも，グローバルなポインタやポインタ引数を使用した間接メモリ・アクセスが存在すると，同様のことが起こる場合があります。グローバルなポインタを使用する箇所はできるだけ局所化してください。

たとえば，例 1 のような場合には最適化が行われませんが，ローカルな変数を用いて例 2 のようにすると，最適化が効率良く行われます。

例 1

```
int*   sp;
int    s1, s2, s3;

void func(void)
{
       int    a = 0;
       int    b = 1;

       *sp = s1 / s2 * 100;

       if(s1 == 0) {
              s3 = *sp + a :
       }
       else {
              s3 = *sp - b;
       }
}
```

例 2

```
int*   sp;
int    s1, s2, s3;

void func(void)
{
       int    a = 0;
       int    b = 1;
       register intt mp = s1 / s2 * 100;

       if(s1 == 0) {
              s3 = tmp + a;        /* a は 0 で置換される */
       }
       else{
              s3 = tmp - b;        /* b は 1 で置換される */
       }
       *sp = tmp;
}
```

7.4 アセンブラ命令の記述と最適化

アセンブラ命令の記述（「3.4 アセンブラ命令の記述」を参照）が含まれる場合，そこですべての変数の値が使用，および変更されると仮定して処理が行われるため，アセンブラ命令の記述を越えた最適化は実行されません。したがって，処理の効率の低下を避けるために，アセンブラ命令の記述を含む関数はできるだけ少なくするようにしてください。また，実行速度優先最適化を指定し，ラベルを定義しているアセンブラ命令の記述を含む関数を使用した場合，関数の定義とインライン展開された部分とで，同じラベルが定義されることになります。この場合，ラベルの多重定義エラーになるため注意が必要です。

7.5 レジスタ

7.5.1 register 指定子

デバッグ優先最適化（-Od）オプション指定の場合，register 指定子を伴って宣言された変数には，register 指定子のない変数よりも優先的にレジスタ変数用のレジスタが割り当てられます。

ただし，register 宣言された変数でも，参照回数が少ないものはレジスタに割り付けられないため，コードが悪化することはありません。

したがって，register 指定子で変数を定義して，デバッガで変数参照してみても，期待した値が見えないことがあります。register 指定子で宣言した変数の数や CA850 の最適化の影響になります。レジスタ変数用レジスタの数が，register 指定子で宣言した変数の数より少ないと，変数はレジスタに割り当てられなくなります。

レジスタ・モード別のレジスタ変数用レジスタの数は，次のとおりです。

- 22 レジスタ・モード：5（r25 ～ r29）
- 26 レジスタ・モード：7（r23 ～ r29）
- 32 レジスタ・モード：10（r20 ～ r29）

たとえば，22 レジスタ・モード時に 6 個以上 register 指定子で変数を宣言しても，すべてはレジスタに入りません。

デバッグ優先最適化（-Od）オプションでは，register 指定子で宣言した変数を優先的にレジスタ変数用レジスタに割り当てます。

ただし，register 指定された変数でも参照回数が少ないものはレジスタに割り付けられません。そして，デバッグ優先最適化（-Od）オプション以外を指定した場合は，register 指定子に関係なく，相対的に参照頻度の多い変数にレジスタを割り付けられます。

このように割り付けられた場所が変わってしまうため，シンボル情報などにも影響が及び，デバッガから参照できなくなることがあります。参照できない変数を使った演算結果が問題なければ，正常に動作していると思われます。

7.5.2 静的変数，および外部変数

デバッグ優先最適化（-Od）オプション／デフォルト最適化（-Ob）オプション指定の場合，レジスタ割り付けの際に，静的変数や外部変数はレジスタに割り付けられません。

したがって，これらの変数が関数内で頻繁に使用されていて，かつその関数内での関数呼び出しや asm 宣言などによって値が変更されることがない場合，関数の先頭でその静的変数，または外部変数の値を register 宣言された自動変数に代入し，関数の終わりで静的変数に戻すようにすれば，レジスタが活用される機会が増え，速度の向上が望めます。

- デフォルト最適化（-Ob）オプション
- 標準最適化（-Og）オプション
- 高度な最適化（-O）オプション
- より高度な最適化（オブジェクトサイズ優先）（-Os）オプション
- より高度な最適化（実行速度優先）（-Ot）オプション

指定の場合，静的変数や外部変数であっても相対的に参照頻度の多い変数であれば，レジスタに割り付きます。

7.5.3 K&R 形式の関数の引数

K&R 形式の関数定義で，int 型よりもサイズが小さい形の引数が存在する場合，オブジェクト・サイズ優先最適化，または実行速度優先最適化であっても，仮引数が register 指定子を伴って宣言されないと，レジスタに割り付きません。レジスタに割り付けたい場合，register 指定子を伴った宣言をしてください。

また，K&R 形式で関数定義を書く場合，引数の型に char，signed char，unsigned char，short，signed short，unsigned short 型はできるだけ使用しないでください。

7.5.4 レジスタに割り当たるローカル変数は，いくつまでが適当か？

ローカル変数（auto 変数）は，10 個以下，なるべく 6 ～ 7 個くらいにしてください。ローカル変数はレジスタに割り当てられます[注]。CA850 では，1 つの関数内で，作業用レジスタとして 10 本，レジスタ変数用レジスタとして 10 本，合計 20 本のレジスタを変数用として使用可能です（32 レジスタ・モードの場合）。1 つの関数内の処理が，時間のかかる処理であれば，ローカル変数を多く使用することを推奨します。時間のかからない処理であれば，できるだけ作業用レジスタの 10 本のみを使用するようにしてください。

レジスタ変数用レジスタは，退避／復帰のオーバーヘッドがかかります。レジスタ変数を使用するかしないかは，CA850 が自動的に判断します。したがって，ローカル変数は，6 ～ 7 個にし，残りの 3 ～ 4 本のレジスタは，CA850 の作業用として使用できるようにすると効率がよくなります。

注 非 volatile 変数で，かつアドレスが取られないものが対象となります。したがって，アドレスを使用するローカル変数の場合は，スタック領域に確保されます。

7.5.5 関数の引数は，いくつまでが適当か？

引数用のレジスタは，r6 ～ r9 の 4 個です。引数が 5 個以上の場合は，5 個目以降はスタックを使用します。

したがって，引数はなるべく 4 個以内にしてください。どうしても 5 個以上になる場合，構造体のポインタで引数を渡すようにすると効率がよくなります。

7.5.6 その他

値が設定される前に変数が参照される可能性のあるパスが存在する場合（例 3），メモリからレジスタへの無駄な転送コードが生成される可能性があります。変数は値を設定してから使用してください（例 4）。

例 3

```
int    s;

void f(int x)
{
       int    y;
       int    i;

       for(i = x; i < 10; i++) {
              if(i == 3) {
                     y = 10;
              }
       }
       s = y * y * x;
}
```

例 4

```
int    s;

void f(int x)
{
       int    y = 0;
       int    i;

       for(i = x; i < 10; i++) {
              if(i == 3) {
                     y = 10;
              }
       }
       s = y * y * x;
}
```

7.6 スタック・サイズ

CA850 は，1 つの変数をスタック上の 1 つの領域に割り付けます。複数の変数を同一の領域に割り付けることはしません。そのため，変数を使い回すことにより[注]，スタック・サイズを小さくできます。

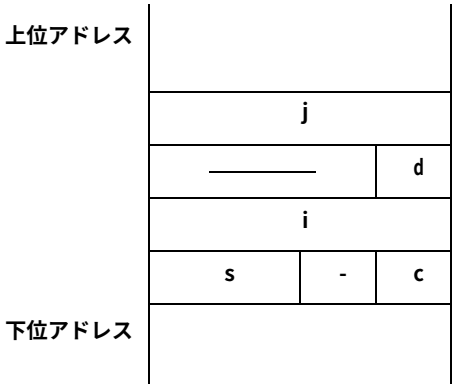
注 このようにした場合，読みづらいプログラムになるため注意してください。

7.7 データの整列

データの定義は，データ長の長いものからまとめて宣言してください。V850 マイクロコントローラでは，int 型などのワード・データはワード境界，short 型などのハーフワード・データはハーフワード境界に整列している必要があります。

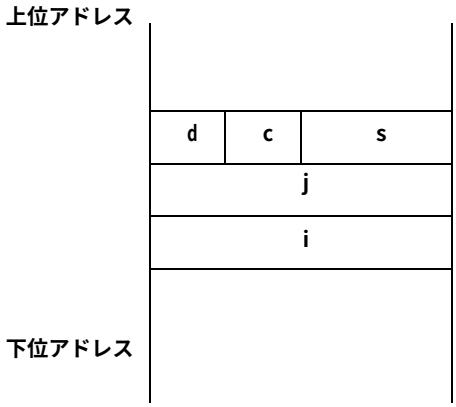
このため，次のようなソースに対しては整列のためのパディング領域が発生します。

```
char   c = 'a ';
short  s = 0;
int    i = 1;
char   d = 'b';
int    j = 2;
```



このようなパディング領域の発生を防ぐため，データ長の長いものからまとめて宣言してください。

```
int    i = 1;
int    j = 2;
short  s = 0;
char   c = 'a';
char   d = 'b';
```



7.8 データ型

V850 マイクロコントローラでは，バイト・データ，ハーフワード・データをメモリからレジスタにロードする場合，最上位ビットの値によりワード長へ符号拡張します。このため，unsigned char，unsigned short 型データの演算では，上位ビットのマスク・コードが生成される場合があります[注]。できるだけワード・データを使用するようにしてください。

また，バイト・データ，ハーフワード・データを使用する場合は，符号付きの型を使用してください。

注 データがすでにレジスタ上にある場合の演算では生成されません。

注意 CA850 で，ターゲット・デバイスに V850Ex を使用する場合，V850Ex のアーキテクチャでは，符号なしロード命令，および型変換命令が存在するため，マスク・コードは生成されません。

レジスタ変数の場合，符号付きバイト・データ，符号付きハーフワード・データの演算ではオペランドを汎整数拡張[注]するので，符号拡張のためのシフト命令が生成されます。また，演算結果をレジスタ変数へストアする際，符号付きバイト・データ，符号付きハーフワード・データでは符号拡張のためのシフト命令が生成され，符号なしバイト・データ，符号なしハーフワード・データでは上位ビットをクリアするためのマスク・コードが生成されます。

このようなコード生成を避けるため，レジスタ変数の場合は，できるだけワード・データ（int，long，unsigned int，unsigned long 型データ）を使用してください。

注 汎整数拡張とは，元の型でのすべての値を int 型で表現できる場合その値を int 型に変換し，そうでない場合 unsigned int 型に変換することをいいます。

マスク・レジスタ機能の利用

ワード・データを用いることができず，マスク・コードが生成されてしまうプログラムの場合，マスク・レジスタ機能（「2.5 マスク・レジスタ」参照）を利用することにより，コード・サイズを削減できます。

次に，バイト・データ，ハーフワード・データ，ワード・データにおける命令生成の例を示します。

例（C 言語記述）

```
int    i, j, k;
unsigned short s, t, u;
unsigned char c, d, e;

void f(void)
{

       register int  ri, rj, rk;
       register short rs, rt, ru;
       register unsigned char ruc, rud, rue;

       c = d + e;
       s = t + u;
       i = j + k;

       rs = rt + ru;
       ruc = rud + rue;
       ri = rj + rk;
}
```

（出力コード）

```
# バイト・データ：
        ld.b    $_d, r10
        andi    0xff, r10, r10      -- マスク・コード
        ld.b    $_e, r11
        andi    0xff, r11, r11      -- マスク・コード
        add     r11, r10
        st.b    r10, $_c

# ハーフワード・データ：
        ld.h    $_t, r12
        andi    0xffff, r12, r12    -- マスク・コード
        ld.h    $_u, r13
        andi    0xffff, r13, r13    -- マスク・コード
        add     r13, r12
        st.h    r12, $_s

# ワード・データ：
        ld.w    $_j, r14
        ld.w    $_k, r15
        add     r15, r14
        st.w    r14, $_i

# 符号付きハーフワード・データ（レジスタ変数）：
        mov     r25,r16
        shl     16, r16             -- シフト命令（汎整数拡張）
        sar     16, r16             --     〃
        mov     r24,r17
        shl     16, r17             -- シフト命令（汎整数拡張）
        sar     16, r17             --     〃
        add     r17,r16
        shl     16, r16             -- シフト命令（演算結果の符号拡張）
        sar     16, r16             --     〃

# 符号なしバイト・データ（レジスタ変数）：
        mov     r22,r18
        add     r21,r18
        addi    0xff, r18, r18      -- マスク・コード
        mov     r18, r23

# ワード・データ（レジスタ変数）：
        mov     r28, r19
        add     r27, r19
        mov     r19, r29
```

付録 A　CC78K*x* の拡張機能

ここでは，CA850 での CC78K*x* の拡張機能について説明します。

A.1　#pragma 指令

CA850 では，CC78K*x* の互換の次の #pragma 指令が指定できます。

なお，【78K 互換】マークは，次の意味を示します。

【78K 互換】	-cc78K オプションを指定しないと有効になりません。
	#pragma 以降のキーワードについては，大文字，小文字を区別しません。

(1) デバイス種別指定

【78K 互換】

```
#pragma pc( デバイス名 )
```

使用するデバイスの機種依存情報を定義したデバイス・ファイルを参照するように指定します。CA850 の“#pragma cpu デバイス名”，およびデバイス指定オプション（-cpu）と同じ機能です。

(2) 周辺 I/O レジスタ名有効化指定

【78K 互換】

```
#pragma sfr
```

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。CA850 の“#pragma ioreg”と同じ機能です。

(3) 割り込み禁止指定

【78K 互換】

```
#pragma di
```

関数 DI() を組み込み関数 __DI() として扱います。

(4) 割り込み許可指定

【78K 互換】

```
#pragma ei
```

関数 EI() を組み込み関数 __EI() として扱います。

(5) CPU 停止関数指定

【78K 互換】

```
#pragma halt
```

関数 HALT() を組み込み関数 __halt() として扱います。

(6) ノーオペレーション関数指定

【78K 互換】

```
#pragma nop
```

関数 NOP() を組み込み関数 __nop() として扱います。

(7) CC78K*x* の #pragma 指令

次の指定については，78K 互換ではありません。CA850 の #pragma 指令として扱います。

(a) 割り込み／例外ハンドラ指定

【78K 互換】

```
#pragma interrupt 割り込み要求名 関数名 [ スタック切り替え ] ...
#pragma vect 割り込み要求名 関数名 [ スタック切り替え ] ...
```

CC78K*x* の “ #pragma interrupt ”／“ #pragma vect ”を CA850 の “ #pragma interrupt 割り込み要求名 関数名 [配置方法] ” として扱います。“ [スタック切り替え] ” 以降の記述がある場合，CA850 で認識できない場合には，次のメッセージを出力します。

```
W2150: unexected character(s) following directive 'directive'
```

(b) セクション指定

【78K 互換】

```
#pragma section ...
```

CA850 の “ #pragma section セクション種別 [" セクション名 "] [begin|end] ” として扱います。CA850 で認識できない場合には，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma directive', ignored
```

(c) メモリ操作関連指定

【78K 互換】

```
#pragma inline
```

CC78K*x* では，memcpy，memset，memchr，および memcmp をインライン展開しますが，CA850 では，指定関数をインライン展開するため，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma inline', ignored
```

(d) モジュール名指定

【78K 互換】

```
#pragma name モジュール名
```

CA850 では，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma name', ignored
```

(e) データ挿入関数指定

【78K 互換】

```
#pragma opc
```

対応する組み込み関数

```
__OPC();
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

```
W2162: unrecognized pragma directive '#pragma opc', ignored
E2752: cannot call opc function
```

(f) バイトアドレス挿入／生成関数指定

【78K 互換】

```
#pragma addraccess
```

対応する組み込み関数

```
FP_SEG();     FP_OFF();     MK_FP();
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

```
    W2162: unrecognized pragma directive '#pragma addraccess', ignored
E2752: cannot call addraccess function
```

(g) レジスタ直接参照関数指定

【78K 互換】

```
#pragma realregister
```

対応する組み込み関数

```
__absa();    __ashra();    __clr1cy();   __coma();    __deca();    __geta();
__getax();   __getcy();    __inca();     __nega();    __not1cy();  __rola();
__rolca();   __rora();     __rorca();    __set1cy();  __seta();    __setax();
__setcy();   __shla();     __shra();
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

```
W2162: unrecognized pragma directive '#pragma realregister', ignored
E2752: cannot call realregister function
```

(h) ファームウエア内蔵セルフ書き込みサブルーチン直接呼び出し関数指定

【78K 互換】

```
#pragma hromcall
```

対応する組み込み関数

```
__FlashAreaBlankCheck();   __FlashAreaErase();        __FlashAreaIVerify();
__FlashAreaPreWrite();     __FlashAreaWriteBack();
__FlashBlockBlankCheck();
__FlashBlockErase();       __FlashBlockIVerify();     __FlashBlockPreWrite();
__FlashBlockWriteBack();   __FlashByteRead();         __FlashByteWrite();
__FlashEnv();              __FlashGetInfo();          __FlashSetEnv();
__FlashWordWrite();        __hromcall();              __hromcalla();
__setsp();
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

```
W2162: unrecognized pragma directive '#pragma hromcall', ignored
E2752: cannot call hromcall function
```

A.2 アセンブラ制御命令

【78K 互換】

```
#asm
        アセンブラ命令
#endasm
```

CA850 の “ #pragma asm ” ～ “ #pragma endasm ” として扱います。

それぞれに対して次のメッセージを出力します。

```
W2166: recognized pragma directive '#pragma asm'
W2166: recognized pragma directive '#pragma endasm'
```

A.3 割り込み／例外ハンドラの指定方法

割り込み／例外ハンドラの指定は，C 言語ソース・プログラムにおいて，次の #pragma 指令と修飾子で行います（「3.7 割り込み／例外処理ハンドラ」参照）。

【78K 互換】

```
#pragma interrupt 割り込み要求名 関数名（C 言語記述）[ 配置方法 ]

__interrupt_brk 関数定義，または関数宣言
```

__interrupt_brk 関数修飾子は，CA850 の __interrupt 関数指定として扱います。

A.4 サポートしていない拡張機能

CA850 は，サポートしていない CC78K*x* の拡張仕様に対してメッセージを出力します。

【78K 互換】

```
__banked1    __banked2    __banked3    __banked4    __banked5
__banked6    __banked7    __banked8    __banked9    __banked10
__banked11   __banked12   __banked13   __banked14   __banked15
callf        __callf      callt        __callt      noauto
norec        __pascal     sreg         __sreg       __sreg1
__temp
```

CA850 では，次のメッセージを出力します。

```
W2761: unrecognized specifier 'specifier', ignored
```

付録 B　注意事項

ここでは，CA850 を用いる際に注意すべき点について説明します。

(1) フォルダ／パスの区切り

“ \ ” と “ / ” の両方が区切りとみなされます。

(2) オプションの指定順序

CA850 は，コマンド・ラインにおけるコマンド起動時に指定したオプションの指定順序について，次の制限があります。

-W オプションで特定のモジュールに渡される引数と，ドライバによって認識されたオプションの引数とで，モジュール起動時に実際に渡される順序は保証されません[注]。

注　CA850 から ld850 を起動する場合，ld850 へは -W オプションで指定しなくても，デフォルトで -lm -lc が渡されます。また，CA850 から ld850 を起動する場合，デフォルトでスタート・アップ・モジュール crtN.o ／ crtE.o が，ld850 へ渡されます。

例

```
> ca850 -cpu 3201 file.o -Wl,-D,dfile.dir
```

ld850 起動時には次のように渡されます。

```
ld850 Install Folder\lib850\r32\crtN.o -o a.out file.o -lm -lc -D dfile.dir
```

ただし，ld850 は *Install Folder*\bin に置かれているものとします。

注意　ld850 を直接起動する場合，ライブラリの指定は，数学ライブラリが標準ライブラリを参照するため，“ -lc ” は “ -lm ” の後ろに指定してください（「6.11　数学関数」を参照）。

(3) 関数宣言／定義における K&R 形式との混在

関数の宣言と定義において，K&R 形式と ANSI 規格形式が混在している場合，K&R 形式における引数拡張処理の結果，CA850 によるコンパイル時にエラーとなる場合があります。

たとえば，次の例では，関数宣言を ANSI 規格で行っていますが，関数定義は K&R 形式で行っているため，引数の型に不整合が生じ，CA850 は " 関数の再宣言 " エラーを出力します。

エラーとなる例

```
void func(int a, int b, float c);
       /* ANSI 規格形式で宣言します */
       /* 第三引数は float 型として宣言します */
       :

void func(a, b, c)
int    a, b;
float  c;
{
       /* K&R 形式で定義します */
       /* 第三引数は，K&R のデフォルトの拡張のため，double 型となります */
       :
}
```

この例の場合，関数宣言で " void func(); " として K&R 形式に統一する，または関数定義で " void func(int a, int b, float c) " として ANSI 規格形式に統一することにより，正常にコンパイルができます。

ただし，CA850 では，ANSI 規格の形式で統一することを推奨しています。

(4) ポジション・インディペンデントではないコード出力

CA850 は，基本的に，位置に依存しない（ポジション・インディペンデント）コードを出力します。ただし，「自動変数以外のポインタ型の変数に対する，数値以外の初期値による初期化指示」に対しては，次の例に示すようなコードを出力します。

例

（C 言語の記述）	（出力コード）

`char   *ptr = "test\n";`	`.size  LL20, 6`
	`LL20 :`
	`.str   "test\n\0"`
	`.align 4`
	`.globl _ptr, 4`
	`_ptr :`
	`.word  #LL20 --` ラベルの絶対アドレス参照

CA850 は，-Xd オプションを指定した場合，自動変数以外のポインタ変数に対する，数値以外の初期値による初期化の指示が現れると，次に示す警告メッセージを出力し，コンパイルを続行します。

```
W2231: Initialization of non-auto pointer using non-number initializer is not
position independent.
```

(5) 型の構成における派生型修飾の回数

CA850 は，型の構成において 17 回以上の派生型修飾[注]が行われた場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2260: compiler limit : complicated type modifiers [16]
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

注　宣言子の中に含まれる *（ポインタ），[]（配列），および関数宣言子のことを意味します。

(6) 識別子の長さと有効文字数

CA850 は，外部識別子で 1023 文字，内部識別子で 1024 文字以上の長さの識別子が記述された場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2117: compiler limit:too long identifier 'symbol' [1022 / 1023]
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

識別子名における有効な文字は，外部識別子の場合，先頭から 1022 文字まで，内部識別子の場合，先頭から 1023 文字までです。

(7) ブロックのネスティングの回数

CA850 は，“{ ”と“ }”の組（ブロック）が 128 回以上ネストした形で用いられた場合，次のメッセージを出力します。

```
F2020: compiler limit : scope level too deep [127]
```

(8) switch 文中の case ラベルの個数

CA850 は，1 つの switch 文中に 1026 個以上の case ラベルが記述された場合，次のエラー・メッセージを出力し，コンパイルを中止します。

```
F2410: compiler limit : too many case labels [1025]
```

ただし，switch 文のネストの数によっては，case ラベルの数が 1025 個に満たなくても，上記のメッセージを出力し，コンパイルを中止することがあります。

(9) 定数式の演算による浮動小数点演算例外

CA850 は，定数式の演算において浮動小数点演算例外が発生した場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2519: exception has occurred at compile time.
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

なお，exception には例外の種類によって inexact，underflow，overflow，division-by-0，または others のいずれかが出力されます。

(10) 巨大な／大量のファイルのマージ

CA850 では，最適化レベルに応じて中間言語ファイルのマージを行います。このとき，マージを行うプリオプティマイザ（popt850）はコンパイル処理の高速化のために，メモリ上で処理します。そのため，巨大な中間言語ファイル，または大量の中間言語ファイルをマージする場合，メモリ不足から次のエラー・メッセージを出力し，異常終了することがあります。

```
F7009: out of memory
```

この場合，プリオプティマイザがメモリ消費を抑える処理を行うオプション（-Wp, -D）を指定して，コマンド・ラインで再コンパイルしてください。

(11) 巨大なファイルの最適化

CA850 では，オブジェクト・サイズ優先最適化，または実行速度優先最適化の場合，広域最適化部（opt850）内部で，データ・フローを関数単位で解析し，広域的な最適化を行います。この最適化はメモリを多く必要とするため，巨大な関数を含むソース・ファイルを最適化する場合，メモリ不足から次のエラー・メッセージを出力し，異常終了することがあります。

```
F5104: out of memory
```

特に，実行速度優先最適化の場合，関数のインライン展開が行われた結果として，関数の大きさが巨大になっている可能性があります。この場合，最適化レベルを低くして，再コンパイルしてください。

(12) オプション指定によるライブラリ・ファイル検索

CA850 では，オプション（-L, -l）によるライブラリ・ファイルの検索[注]の結果，指定されたライブラリ・ファイルが存在しなくてもメッセージを表示しません。ただし，ライブラリ・ファイル名をコマンド・ライン，およびコマンド・ファイルで直接指定した場合はフォルダ，メッセージを表示します。

注 -L オプションを指定しない場合，標準のフォルダ（*Install Folder*\lib850，およびその下の各レジスタ・モード・フォルダ）で検索します。

例

```
> ca850 -cpu 3201 a.c usr.a
```

```
F4002: can not open input file"usr.a".
```

(13) volatile 修飾子

volatile 修飾子をつけて変数宣言すると，その変数は最適化の対象からはずされ，レジスタに割り付ける最適化などを行わなくなります。volatile 指定された変数に対する操作を行うときは，必ずメモリから値を読み込み，操作後にメモリへ値を書き込むコードになります。また volatile 指定された変数のアクセス幅も変更されません。

volatile 指定されていない変数は，最適化によってレジスタに割り付けられ，その変数をメモリからロードするコードが削除されることがあります。また volatile 指定されていない変数に同じ値を代入する場合，冗長な命令と解釈されて最適化により命令が削除されることもあります。特に周辺 I/O レジスタへアクセスする変数や，割り込み処理で値が変更される変数，また外部から値が変更される変数に対しては，volatile 指定する必要があります。ただし，CA850 では，# pragma ioreg 指令を使って周辺 I/O レジスタにアクセスする場合，内部的に volatile 指定されているコードが出力されるので，改めて volatile 修飾子をつけて宣言する必要ありません。

volatile 指定すべきところで指定されていなかった場合，次の現象が起こることがあります。

- 正しい計算結果が得られない
- for ループ内で変数を使っていた場合，ループから抜け出せない

ただし，volatile 指定した変数を使用する際，ある区間でその変数の値が外部から変更されないことが自明な場合，volatile 指定されていない変数に，その値を代入してその変数を参照することにより，その変数が最適化され，実行速度が向上する可能性があります。

【volatile 指定しなかった場合のソースと出力コードの例】

“変数 a”,“変数 b”，および “変数 c” を volatile 指定しなかった場合，これらの変数がレジスタに割り付けられ，最適化されます。たとえば，この間に割り込みが入り，割り込み内で変数値を変更しても，値が反映されないことになります。

```
int a;
int b;
int c;

void func(void)
{
       if (a <= 0) {
              b++;
       } else {
              c++;
       }
       b++;
       c++;
}
```

```
_func:
       #@B_PROLOGUE
       #@E_PROLOGUE
       ld.w $_a, r12
       cmp r0, r12
       jgt .L2
       ld.w $_b, r11
       ld.w $_c, r10
       add 1, r11
       jbr .L3
.L2:
       ld.w $_c, r10
       ld.w $_b, r11
       add 1, r10
.L3:
       addi 1, r11, r13
       st.w r13, $_b
       addi 1, r10, r14
       st.w r14, $_c
       #@B_EPILOGUE
       jmp [lp]
       #@E_EPILOGUE
```

【volatile 指定した場合のソースと出力コードの例】

“ 変数 a ”,“ 変数 b ”, および “ 変数 c ”を volatile **指定した場合，これらの変数値を必ずメモリから読み込み，操作後にメモリへ書き込むコードが出力されます。たとえば，この間に割り込みが入り，割り込み内で変数値が変更されても，その変更が反映された結果を取得することができます（このような例の場合，割り込みのタイミングによっては，変数の操作区間内を割り込み禁止にするなどの処置が必要となります）。**

volatile **指定をすると，メモリの読み込み／書き込み処理が入るため，**volatile **指定しなかった場合よりもコード・サイズは大きくなります。**

```
volatile int a;
volatile int b;
volatile int c;

void func(void)
{
       if (a <= 0) {
              b++;
       } else {
              c++;
       }
       b++;
       c++;
}
```

```
_func:
       #@B_PROLOGUE
       #@E_PROLOGUE
       .option volatile
       ld.w $_a, r10
       .option novolatile
       cmp r0, r10
       jgt .L2
       .option volatile
       ld.w $_b, r11
       .option novolatile
       add 1, r11
       .option volatile
       st.w r11, $_b
       .option novolatile
       jbr .L3
.L2:
       .option volatile
       ld.w $_c, r12
       .option novolatile
       add 1, r12
       .option volatile
       st.w r12, $_c
       .option novolatile
.L3:
       .option volatile
       ld.w $_b, r13
       .option novolatile
       add 1, r13
       .option volatile
       st.w r13, $_b
       .option novolatile
       .option volatile
       ld.w $_c, r14
       .option novolatile
       add 1, r14
       .option volatile
       st.w r14, $_c
       .option novolatile
       #@B_EPILOGUE
       jmp [lp]
       #@E_EPILOGUE
```

(14) 関数宣言での余計な括弧

関数宣言で，括弧“( )”が余計に記述されている場合，ANSI-C では次のように既定されていますが，CA850 では，エラーとなります。

例

```
typedef int Int;
void f1((Int));
```

【ANSI-C での規定】

仮引数宣言内では，括弧で囲まれた型定義名は，単一の仮引数をもつ関数を指定する抽象宣言子と解釈する。宣言子の識別子を囲む冗長な括弧とは解釈しない。

したがって，上記の例では，ANSI-C では次のように解釈されます。

```
void f(int (*)(int));
```

括弧を余計に記述してしまった場合には，次のように不必要な括弧を削除してください。

例

```
typedef int Int;
void f1(Int);
```

付録 C　索引

【発　行】
NECエレクトロニクス株式会社
〒211-8668　神奈川県川崎市中原区下沼部1753
電話（代表）：044(435)5111

お問い合わせ先

【ホームページ】

NECエレクトロニクスの情報がインターネットでご覧になれます。
URL(アドレス)　http://www.necel.co.jp/

【営業関係，技術関係お問い合わせ先】

半導体ホットライン
（電話：午前 9:00～12:00，午後 1:00～5:00）
電　話　：044−435−9494
E-mail　：info@necel.com

【資料請求先】

NECエレクトロニクスのホームページよりダウンロードいただくか，NECエレクトロニクスの販売特約店へお申し付けください。