

お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010年4月1日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したのですが、誤りが無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、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と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。

M16C/80 シリーズ

プログラム作成の手引き < C 言語編 >

はじめに

このプログラミングマニュアルは、ルネサス CMOS16 ビットマイクロコンピュータ M16C/80 シリーズ用の C コンパイラ NC308 を用いて、C 言語の基礎から ROM 化、リアルタイム OS (MR308) の使用方法までを説明しています。C 言語の入門書として、またプログラム作成時の参考資料として使用してください。

なお、M16C/80 シリーズ各品種のハードウェア、および開発サポートツールにつきましては、各ユーザーズマニュアル、および各操作説明書を使用してください。

本書の使い方

本書は M16C/80 シリーズ用 C コンパイラ NC308 のプログラミングマニュアルです。

本書を使用する上で、M16C/80 シリーズのアーキテクチャ、およびアセンブリ言語に関する知識が必要です。

本書は 3 つの章で構成されています。以下に目的に応じた参照先を示します。

はじめて C 言語を学習される方	第 1 章から
NC308 の拡張機能を知りたい方	第 2 章から
リアルタイム OS MR308 を使用される方	第 3 章から

さらに巻末には付録として、「NC30 と NC308 の機能比較」、「nc308 コマンドリファレンス」と「Q & A」を記載しています。

目次

第 1 章 C 言語入門	15
1.1 C 言語によるプログラミング	16
1.1.1 アセンブリ言語と C 言語	16
1.1.2 プログラム開発の手順	17
1.1.3 わかりやすいプログラム	19
1.2 データ型	23
1.2.1 C 言語で扱える “定数”	23
1.2.2 変数	25
1.2.3 データの特性	27
1.3 演算子	29
1.3.1 NC308 の演算子	29
1.3.2 数値計算のための演算子	30
1.3.3 データ加工のための演算子	33
1.3.4 条件を調べるための演算子	35
1.3.5 その他の演算子	36
1.3.6 演算子の優先順位	38
1.4 制御文	41
1.4.1 プログラムの構造化	41
1.4.2 条件による処理の分岐 (分岐処理)	42
1.4.3 同じ処理の繰り返し (繰り返し処理)	46
1.4.4 処理の中断	49
1.5 関数	51
1.5.1 関数とサブルーチン	51
1.5.2 関数の作成	52
1.5.3 関数間でのデータの受け渡し	54
1.6 記憶クラス	55
1.6.1 変数と関数の有効範囲	55
1.6.2 変数の記憶クラス	56
1.6.3 関数の記憶クラス	58
1.7 配列とポインタ	61
1.7.1 配列	61
1.7.2 配列の作成	62
1.7.3 ポインタ	64
1.7.4 ポインタの活用	66

1.7.5	ポインタの配列化	68
1.7.6	関数ポインタを使ったテーブルジャンプ	70
1.8	構造体と共用体	72
1.8.1	構造体と共用体	72
1.8.2	新しいデータ型の作成	73
1.9	プリプロセスコマンド	77
1.9.1	NC308 のプリプロセスコマンド	77
1.9.2	ファイルの取り込み	78
1.9.3	マクロ定義	79
1.9.4	条件コンパイル	81
第 2 章	ROM化技術	83
2.1	メモリ配置	84
2.1.1	コード/データの種類	84
2.1.2	NC308 が管理するセクション	85
2.1.3	メモリ配置の制御	87
2.1.4	構造体のメモリ配置の制御	90
2.2	スタートアッププログラム	92
2.2.1	スタートアッププログラムの役割	92
2.2.2	スタック使用量の見積もり	94
2.2.3	スタートアッププログラムの作成	97
2.3	ROM化のための拡張機能	104
2.3.1	効率よいアドレッシング	104
2.3.2	ビットの扱い	109
2.3.3	I/Oインタフェースの制御	110
2.3.4	C 言語で書けないときの対処策	112
2.3.5	アセンブラマクロ関数の使用	114
2.4	アセンブリ言語とのリンク	119
2.4.1	関数間のインタフェース	119
2.4.2	C 言語からのアセンブリ言語の呼び出し	125
2.4.3	アセンブリ言語からの C 言語の呼び出し	133
2.5	割り込み処理	134
2.5.1	割り込み処理関数の記述	134
2.5.2	高速割り込み処理関数の記述	137
2.5.3	ソフトウェア割り込み (INT 命令) 処理関数の記述	139
2.5.4	割り込み処理関数を登録する	141
2.5.5	割り込み処理関数の記述例	142
第 3 章	リアルタイム OS(MR308)の使用	144
3.1	リアルタイム OS の基礎	145

3.1.1	リアルタイム OS とタスク	145
3.1.2	リアルタイム OS の機能	148
3.1.3	割り込み管理	151
3.1.4	特殊なハンドラ	154
3.2	システムコールの利用法	155
3.2.1	MR308 のシステムコール	155
3.2.2	システムコールの記述方法	156
3.3	MR308 を用いた開発手順	159
3.3.1	開発時に必要なファイル	159
3.3.2	MR308 を用いた開発の流れ	164
3.4	NC308 を用いた MR308 の組み込み	165
3.4.1	NC308 を用いたプログラムの記述	165
3.4.2	NC308 を用いたタスクの記述	167
3.4.3	割り込みハンドラの記述	171
3.4.4	周期起動ハンドラ、アラームハンドラの記述	175
付録	177
付録 A	NC308 と NC30 の機能比較	178
付録 B	NC308 コマンドリファレンス	182
付録 C	Q & A	188

目次

第 1 章 C 言語入門	15
1.1 C 言語によるプログラミング	16
図 1.1.1 NC308 の製品一覧	17
図 1.1.2 ソースファイルから機械語ファイルができるまで	18
図 1.1.3 C 言語のソースファイルの構成	19
図 1.1.4 C 言語プログラムのプログラミングスタイルの例	20
図 1.1.5 コメントの利用例	21
1.2 データ型	23
図 1.2.1 1 と '1' の違い	23
図 1.2.2 {'a', 'b'} と "ab" の違い	24
図 1.2.3 変数の宣言	26
図 1.2.4 型修飾子 "signed / unsigned" の記述例	27
図 1.2.5 型修飾子 "const" の記述例	27
図 1.2.6 型修飾子 "volatile" の記述例	28
図 1.2.7 宣言の構文	28
1.3 演算子	29
図 1.3.1 NC308 の除算演算子	31
図 1.3.2 型の違うデータを代入する	32
図 1.3.3 算術シフトと論理シフト	34
図 1.3.4 条件演算子 utilization	36
1.4 制御文	41
図 1.4.1 if - else - if 文の記述例	42
図 1.4.2 else - if 文の記述例	43
図 1.4.3 switch - case 文の記述例	44
図 1.4.4 break を入れない switch - case 文	45
図 1.4.5 while 文の記述例	46
図 1.4.6 for 文の記述例	47
図 1.4.7 do - while 文の記述例	48
図 1.4.8 break 文の記述例	49
図 1.4.9 continue 文の記述例	49
図 1.4.10 goto 文の働き	50
1.5 関数	51
図 1.5.1 「サブルーチン」と「関数」	51
図 1.5.2 関数の記述例	53

1.6	記憶クラス	55
図 1.6.1	C 言語プログラムの階層構造と記憶クラス	55
図 1.6.2	外部変数と内部変数	56
図 1.6.3	外部変数の記憶クラス	57
図 1.6.4	内部変数の記憶クラス	57
図 1.6.5	関数の記憶クラス	58
図 1.6.6	レジスタ変数を使用するには	60
1.7	配列とポインタ	61
図 1.7.1	配列の概念	61
図 1.7.2	1 次元配列の宣言とメモリ配置	62
図 1.7.3	2 次元配列の宣言とメモリ配置	63
図 1.7.4	ポインタ変数の宣言とメモリ配置	64
図 1.7.5	ポインタ変数と変数の関係	65
図 1.7.6	ポインタ変数の演算	65
図 1.7.7	ポインタ変数と 1 次元配列	66
図 1.7.8	ポインタ変数と 2 次元配列	66
図 1.7.9	アドレス渡しの例 (配列を受け渡しする)	67
図 1.7.10	ポインタ配列の宣言と初期化	68
図 1.7.11	2 次元配列とポインタ配列の違い	69
1.8	構造体と共用体	72
図 1.8.1	基本データ型から構造体へ	72
図 1.8.2	構造体の宣言とメモリ配置	74
図 1.8.3	ポインタを使った参照例	75
図 1.8.4	共用体の宣言と参照	76
図 1.8.5	型定義 " typedef " の使用例	76
1.9	プリプロセスコマンド	77
図 1.9.1	" #include " 記述例	78
図 1.9.2	定数の定義例	79
図 1.9.3	文字列の定義例	79
図 1.9.4	マクロ関数の定義例	80
図 1.9.5	条件コンパイル記述例	82
第 2 章	ROM化技術	83
2.1	メモリ配置	84
図 2.1.1	NC308 が生成するデータ / コードと配置領域	84
図 2.1.2	初期値を持つ静的変数の扱い	84
図 2.1.3	データの種類によるセクションへの配置	85
図 2.1.4	セクション名の命名規則	86
図 2.1.5	" #pragma SECTION " の記述例	87
図 2.1.6	セクション名の追加("sect308.inc")	88
図 2.1.7	const 修飾子とメモリ配置	89

図 2.1.8	外部変数の参照を定数に置き換える最適化	89
図 2.1.9	NC308 デフォルトの構造体の割り付けイメージ	90
図 2.1.10	構造体メンバのパックの禁止	90
図 2.1.11	構造体メンバの配置の最適化	91
2.2	スタートアッププログラム	92
図 2.2.1	サンプルスタートアッププログラムの構成	93
図 2.2.2	スタック使用量情報ファイル	94
図 2.2.3	最大スタック使用量の計算方法	95
図 2.2.4	スタックサイズ算出ユーティリティ " stk308 "	96
図 2.2.5	サンプルスタートアッププログラムの変更点	97
図 2.2.6	heap 領域の設定	98
図 2.2.7	スタックサイズの設定	98
図 2.2.9	プロセッサ動作モードの設定	99
図 2.2.8	割り込みベクタテーブルの先頭アドレスの設定	99
図 2.2.10	セクションの先頭アドレスの設定	100
図 2.2.11	可変ベクタテーブルの設定	101
図 2.2.12	固定ベクタテーブルの設定	102
図 2.2.13	シングルチップモードで使用する際の記述例	103
2.3	ROM 化のための拡張機能	104
図 2.3.1	静的変数の near / far	105
図 2.3.2	自動変数の near / far	105
図 2.3.3	ポインタに格納するアドレスサイズを指定	106
図 2.3.4	ポインタの配置領域を指定	106
図 2.3.5	ポインタの near/far における NC308 と NC30 の相違点	107
図 2.3.6	far 領域に配置された変数アドレスの near ポインタへの代入	107
図 2.3.7	" #pragma SBDATA " の展開イメージ	108
図 2.3.8	ビットフィールドの割り付け例	109
図 2.3.9	ポインタによる絶対アドレス指定	110
図 2.3.10	" #pragma ADDRESS " による絶対アドレス指定	110
図 2.3.11	asm 関数の記述例	112
図 2.3.12	asm 関数で自動変数を使用する	112
図 2.3.14	asm 関数を利用した部分的な最適化の抑止	113
図 2.3.13	" #pragma ASM " 機能使用例	113
2.4	アセンブリ言語とのリンク	119
図 2.4.1	関数の呼び出し動作	119
図 2.4.2	関数呼び出し時の ROM サイズ圧縮方法	120
図 2.4.3	スタックフレームの構成	121
図 2.4.4	スタックフレームの構築	121
図 2.4.5	引数の引き渡し例	122
図 2.4.6	戻り値の引き渡し例	123

図 2.4.7	インライン記憶クラスの記述例	124
図 2.4.8	" #pragma PARAMETER " の記述例	125
図 2.4.9	サブルーチンの呼び出し	126
図 2.4.10	#pragma PARAMETER /C の記述例	127
例 2.4.1	サブルーチンを呼び出す	128
図 2.4.11	スペシャルページサブルーチンを呼び出す	129
図 2.4.12	間接アドレッシングでサブルーチンを呼び出す	130
図 2.4.13	C 言語関数の呼び出し	133
2.5	割り込み処理	134
2.5.1	割り込み処理関数の記述	134
図 2.5.1	割り込み処理関数の展開イメージ	134
図 2.5.2	レジスタバンクを使用した割り込み処理関数の展開イメージ	135
図 2.5.3	多重割り込みを許可にする割り込み処理関数の展開イメージ	136
図 2.5.4	高速割り込み処理関数の展開イメージ	137
図 2.5.5	レジスタバンクを使用した高速割り込み処理関数の展開イメージ	138
図 2.5.6	アセンブリ言語関数を呼び出す " #pragma INTCALL " 記述例	139
図 2.5.7	C 言語関数を呼び出す " #pragma INTCALL " 記述例	140
図 2.5.8	割り込みベクタテーブル (" sect308.inc ")	141
図 2.5.9	割り込み処理関数の記述例	142
図 2.5.10	割り込みベクタテーブルへの登録例	143
第 3 章	リアルタイム OS(MR308)の使用	144
3.1	リアルタイム OS の基礎	145
図 3.1.1	プログラムの複数タスクによる構成図	145
図 3.1.2	各タスク状態 (遷移図含む)	146
図 3.1.3	" TCB " の主な構造	148
図 3.1.4	タスク実行時の OS 依存割り込みハンドラの実行	152
図 3.1.5	多重割り込みでの OS 依存割り込みハンドラの実行	153
3.2	システムコールの利用法	155
図 3.2.1	MR308 が提供するシステムコールライブラリ	155
図 3.2.2	システムコールの記述	156
図 3.2.3	パラメータを持つシステムコールの記述	156
図 3.2.4	エラーコードの利用	157
3.3	MR308 を用いた開発手順	159
図 3.3.1	MR308 用スタートアッププログラムの処理概要	159
図 3.3.2	M16C/80 シリーズ制御レジスタの初期化	160
図 3.3.3	割り込みベクタテーブルの先頭アドレスの設定	161
図 3.3.4	周辺 I / O の初期化	161
図 3.3.5	メモリ配置の変更	162
図 3.3.6	コンフィグレーションファイルの概要	163
図 3.3.7	開発の流れ	164

3.4	NC308 を用いた MR308 の組み込み	165
図 3.4.1	タスクの記述例	167
図 3.4.2	タスクの記述例	168
図 3.4.3	タスクを記述する際の注意点 - 1 - (static 型の関数について).....	168
図 3.4.4	タスクを記述する際の注意点 - 2 - (再スタートするタスクの変数の初期化).....	169
図 3.4.5	変数の参照範囲例	170
図 3.4.6	OS 依存割り込みハンドラの記述例	171
図 3.4.7	OS 依存割り込みハンドラの記述例	172
図 3.4.8	OS 依存割り込みハンドラを記述する際の注意点 (static 型の関数について).....	172
図 3.4.9	外部変数を利用したデータのやり取りの記述例	173
図 3.4.10	メールボックスを利用したデータのやり取りの記述例	174
図 3.4.11	周期起動ハンドラ、アラームハンドラの記述例	175
図 3.4.12	周期起動ハンドラの記述例	176
図 3.4.13	周期起動ハンドラの記述例 (誤りの例).....	176
付録	177
付録 A	NC308 と NC30 の機能比較	178
図 A.1	関数呼び出し時のコーリングコンベンション	178
付録 B	NC308 コマンドリファレンス	182
付録 C	Q & A	188
図 C.1	構造体の転送の記述例	188
図 C.2	生成コード変更オプション ' - fsmall_array(- fSA)" 使用例	189
図 C.3	最適化オプション " - Osp_adjust(- OSA)" 使用例	190
図 C.4	"#pragma SBDATA" の記述例	191
図 C.5	" - fJSRW" の使用例と "#pragma JSRA" の記述例	191

表目次

第 1 章 C 言語入門	15
1.1 C 言語によるプログラミング	16
表 1.1.1 C 言語とアセンブリ言語との比較	16
表 1.1.2 NC308 の予約語一覧	22
1.2 データ型	23
表 1.2.1 整数定数の表記方法	23
表 1.2.2 C 言語のエスケープシーケンス	24
表 1.2.3 NC308 の基本データ型	25
1.3 演算子	29
表 1.3.1 NC308 の演算子一覧	29
表 1.3.2 単項算術演算子一覧	30
表 1.3.3 二項算術演算子一覧	30
表 1.3.4 代入演算子一覧	32
表 1.3.5 ビット演算子一覧	33
表 1.3.6 シフト演算子一覧	33
表 1.3.7 関係演算子一覧	35
表 1.3.8 論理演算子一覧	35
表 1.3.9 条件演算子	36
表 1.3.10 sizeof 演算子	36
表 1.3.11 キャスト演算子	37
表 1.3.12 カンマ (順次) 演算子	37
表 1.3.13 演算子の優先順位	38
1.4 制御文	41
表 1.4.1 構造化プログラミングの 3 つの基本形	41
1.5 関数	5
1.6 記憶クラス	55
表 1.6.1 変数の記憶クラス	59
表 1.6.2 関数の記憶クラス	59
1.7 配列とポインタ	61
1.8 構造体と共用体	72
1.9 プリプロセスコマンド	77
表 1.9.1 NC308 の主なプリプロセスコマンド	77
表 1.9.2 条件コンパイル一覧	81

第 2 章 ROM化技術.....	83
2.1 メモリ配置	84
表 2.1.1 NC308 のセクション構成	85
表 2.1.2 セクションの属性	86
2.2 スタートアッププログラム	92
2.3 ROM 化のための拡張機能	104
表 2.3.1 near 領域と far 領域	104
表 2.3.2 near / far 属性のデフォルト	104
表 2.3.3 アセンブルマクロ関数で記述できるアセンブリ言語命令(1).....	114
表 2.3.4 アセンブルマクロ関数で記述できるアセンブリ言語命令(2).....	115
2.4 アセンブリ言語とのリンク	119
表 2.4.1 引数の引き渡し例	122
表 2.4.2 戻り値の引き渡し規則	123
表 2.4.3 シンボル変換規則	124
2.5 割り込み処理	134
第 3 章 リアルタイムOS(MR308)の使用	144
3.1 リアルタイム OS の基礎	145
表 3.1.1 タスクのスタイル	145
表 3.1.2 MR308 のオブジェクト	149
表 3.1.3 オブジェクト操作のための主なシステムコール	150
表 3.1.4 割り込みハンドラの種類	151
表 3.1.5 リアルタイム OS が用意する割り込みハンドラ	154
表 3.1.6 特殊なハンドラ	154
3.2 システムコールの利用法	155
表 3.2.1 エラーコード一覧 (注).....	157
表 3.2.2 データタイプと文字	158
3.3 MR308 を用いた開発手順	159
表 3.3.1 MR308 用メモリ配置関連ファイル	163
3.4 NC308 を用いた MR308 の組み込み	165
表 3.4.1 MR308 を使用するためのインクルードファイル	165
表 3.4.2 MR308 用拡張命令一覧.....	166
表 3.4.3 変数の参照範囲	170

付録	177
付録 A NC308 と NC30 の機能比較	178
表 A.1 NC308 の引数の引き渡し規則	178
表 A.2 NC30 の引数の引き渡し規則	178
表 A.3 アセンブラマクロ関数で記述可能なアセンブリ言語命令	179
表 A.5 追加された拡張機能	180
表 A.4 機能変更された拡張機能	180
表 A.6 NC308 ではサポートしていない拡張機能	181
付録 B NC308 コマンドリファレンス	182
表 B.1 コンパイルドライバの制御に関するオプション	182
表 B.2 出力ファイル指定オプション	183
表 B.3 バージョン情報表示オプション	183
表 B.4 デバッグ用オプション	183
表 B.5 最適化オプション	184
表 B.6 ライブラリ指定オプション	184
表 B.7 生成コード変更オプション	185
表 B.8 警告オプション	186
表 B.9 アセンブル・リンクオプション	186
表 B.10 その他のオプション	186
付録 C Q & A	188

例目次

第 1 章 C 言語入門	15
1.1 C 言語によるプログラミング	16
1.2 データ型	23
1.3 演算子	29
例 1.3.1 「暗黙の型変換」の解釈間違いと対処方法	39
例 1.3.2 「優先順位」の解釈間違いと対処方法	40
例 1.3.3 演算子の誤った使用方法の検出	40
1.4 制御文	41
例 1.4.1 カウントアップ (if - else 文の記述例)	42
例 1.4.2 四則演算の切り換え - 1 - (else - if 文の記述例)	43
例 1.4.3 四則演算の切り換え - 2 - (switch - case 文の記述例)	44
例 1.4.4 総和を求める - 1 - (while 文の記述例)	46
例 1.4.5 総和を求める - 2 - (for 文の記述例)	47
例 1.4.6 総和を求める - 3 - (do - while 文の記述例)	48
1.5 関数	51
例 1.5.1 和を求める (関数の記述例)	54
1.6 記憶クラス	55
1.7 配列とポインタ	61
例 1.7.1 家族の年齢の合計を求める - 1 -	61
例 1.7.2 家族の年齢の合計を求める - 2 -	62
例 1.7.3 テーブルジャンプを使った四則演算の切り換え	71
1.8 構造体と共用体	72
1.9 プリプロセスコマンド	77
第 2 章 ROM 化技術	83
2.1 メモリ配置	84
2.2 スタートアッププログラム	92
2.3 ROM 化のための拡張機能	104
例 2.3.1 "#pragma ADDRESS" を利用した SFR 領域の定義	111
例 2.3.2 アセンブルマクロ関数 "dadd_b" を使用した 1 0 進加算	116
例 2.3.3 アセンブルマクロ関数 "smovf_b" を使用したストリング転送	117
例 2.3.4 アセンブルマクロ関数 "rmpa_w" を使用した積和演算	118

2.4	アセンブリ言語とのリンク	119
例 2.4.1	サブルーチンを呼び出す	128
例 2.4.2	テーブルジャンプでサブルーチンを呼び出す	131
例 2.4.3	少し変わったテーブルジャンプの使い方	132
2.5	割り込み処理	134
第 3 章	リアルタイムOS(MR308)の使用	144
3.1	リアルタイム OS の基礎	145
3.2	システムコールの利用法	155
3.3	MR308 を用いた開発手順	159
3.4	NC308 を用いた MR308 の組み込み	165
付録	177
付録 A	NC308 と NC30 の機能比較	178
付録 B	NC308 コマンドリファレンス	182
付録 C	Q & A	188

第 1 章

C 言語入門

- 1.1 C 言語によるプログラミング
- 1.2 データ型
- 1.3 演算子
- 1.4 制御文
- 1.5 関数
- 1.6 記憶クラス
- 1.7 配列とポインタ
- 1.8 構造体と共用体
- 1.9 プリプロセスコマンド

この章では、初めて C 言語をお使いになる方を対象に、組み込み型プログラムを作成するために必要な C 言語の基礎を紹介しています。

1.1 C 言語によるプログラミング

1.1.1 アセンブリ言語とC言語

近年の、マイクロコンピュータを利用したシステムの大規模化にともない、プログラムの生産性や保守性が注目され始めました。これとともにプログラムの開発言語として、従来のアセンブリ言語にかわり「C言語」を使用するケースが増えていきます。

以下の項では、主なC言語の特長と、C言語によるプログラムの記述方法を説明します。

C言語の特長

- (1) 処理の流れを追いやすいプログラムが記述できる
構造化プログラミングの基本である「順次処理」、「分岐処理」、「繰り返し処理」をすべて制御文で記述できます。このため、処理の流れを追いやすいプログラムが記述できます。
- (2) モジュール分割が容易にできる
C言語で記述したプログラムは「関数」と呼ばれる基本単位から構成されています。関数はパラメータの独立性が高いため、プログラムの部品化や再利用が容易にできます。また、アセンブリ言語で記述したモジュールをそのまま流用できます。
- (3) 保守性のよいプログラムを記述できる
(1)、(2)の理由から、運用後のプログラムのメンテナンスが容易にできます。また、C言語としての標準規格(ANSI規格^(注))が定められているため、ソースプログラムを少し変更するだけで他機種への移植も可能です。

C言語とアセンブリ言語の比較

ソースプログラムの記述方法に関して、アセンブリ言語との違いを表 1.1.1 にまとめます。

表 1.1.1 C言語とアセンブリ言語との比較

	C言語	アセンブリ言語
プログラムの基本単位 (記述方法)	関数 (関数名(){})	サブルーチン (サブルーチン名:)
書式	フリーフォーマット	1行1命令
大文字/小文字の区別	大文字と小文字を区別する (通常は小文字で記述する)	区別しない
データ領域の確保	「データ型」で指定する	バイト数で指定する (疑似命令を使用する)
入出力命令	入出力命令なし	入出力命令あり (ただし、H/Wに依存)

1.1.2 プログラム開発の手順

C 言語で記述されたソースプログラムを機械語に翻訳する作業を「コンパイル作業」といいます。この作業を行うために用意されたソフトウェアが「コンパイラ」です。

この項では、ルネサスシングルチップマイクロコンピュータ M16C/80 用 C コンパイラ「NC308」を用いてプログラム開発を行う手順を説明します。

NC308 の製品一覧

ルネサスシングルチップマイクロコンピュータ M16C/80 用 C コンパイラ NC308 に含まれる製品の一覧を図 1.1.1 に示します。

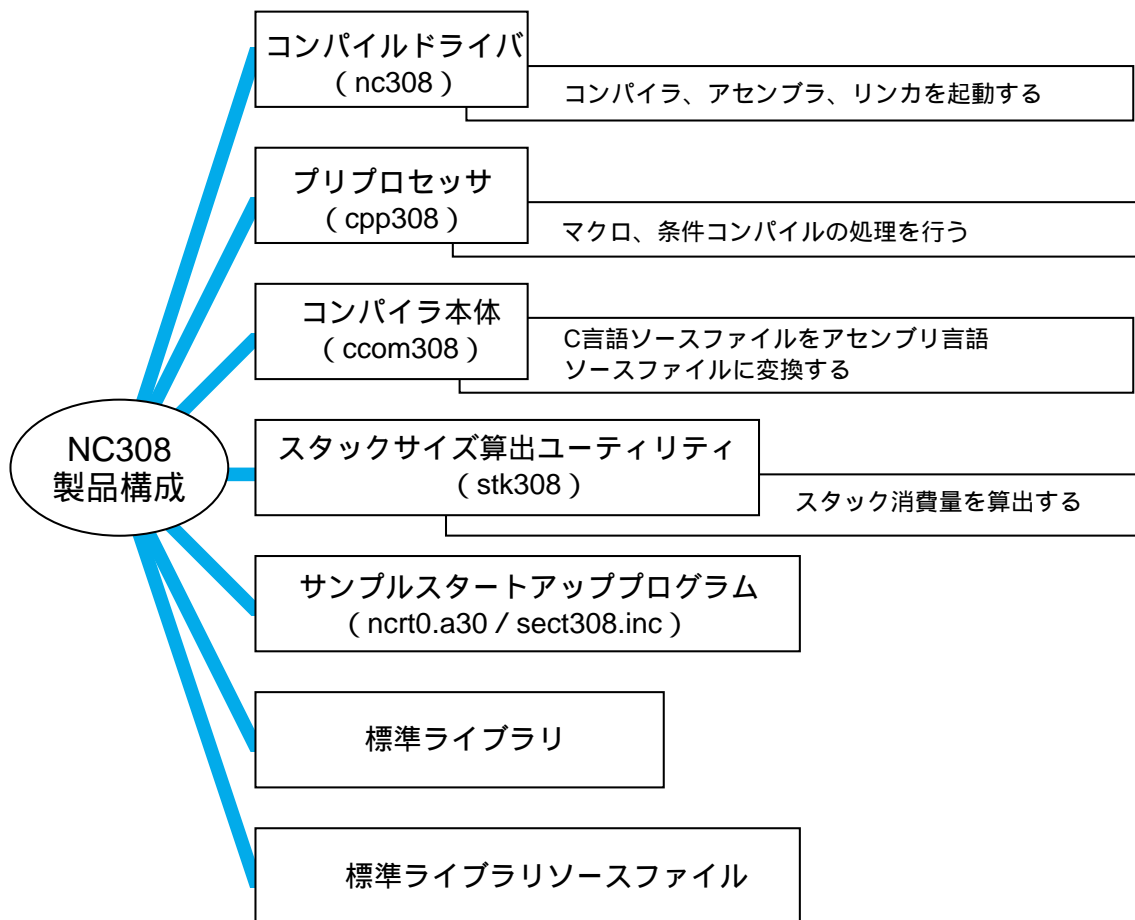


図 1.1.1 NC308 の製品一覧

ソースファイルから機械語ファイルができるまで

NC308で機械語ファイルを生成するには、C言語でプログラムを記述したソースファイルのほかに、アセンブリ言語で記述したスタートアッププログラムが必要です。

機械語ファイルができるまでのツールチェーンを図 1.1.2 に示します。

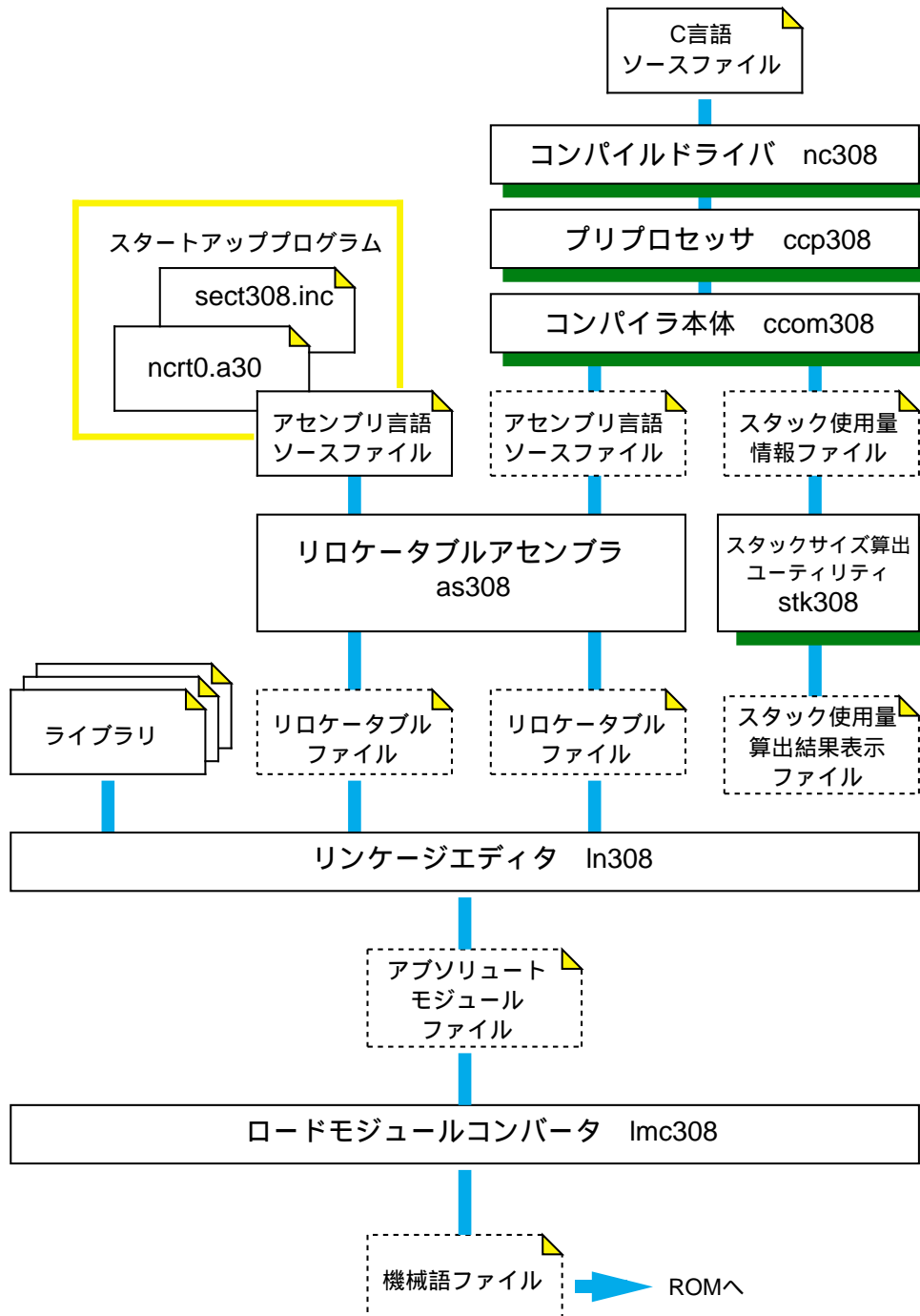


図 1.1.2 ソースファイルから機械語ファイルができるまで

1.1.3 わかりやすいプログラム

C 言語のプログラムはフリーフォーマット形式なので、ある一定の規則を守ればあとは自由にプログラムを記述できます。しかし、プログラムは読みやすく、かつ保守性の高いものにする必要があり、そのためにはいつ誰が見てもそのプログラムを理解できるように記述しなければなりません。

この項では、「わかりやすいプログラム」を記述するためのポイントを説明します。

C 言語の規則

C 言語プログラムを記述するにあたって、守らなければならない規則は次の 6 項目です。

- (1) プログラムは原則的に英小文字で記述する。
- (2) プログラムの実行文は「 ; 」で区切る。
- (3) 関数や制御文の実行単位は「 { 」 「 } 」で囲む。
- (4) 関数や変数は型宣言が必要である。
- (5) 予約語は識別子（関数名、変数名など）に使用できない。
- (6) コメントは「 /* コメント */ 」で記述する。

C 言語のソースファイルの構成

一般的な C 言語のソースファイルの構成を図 1.1.3 にまとめます。各項目については矢印で示す節を参照してください。

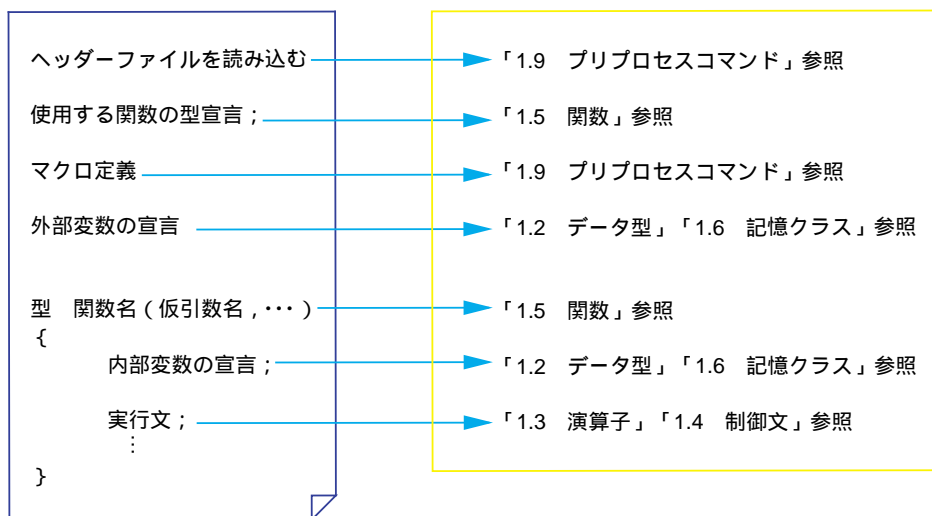


図 1.1.3 C 言語のソースファイルの構成

プログラミングスタイル

プログラムの保守性を高めるためには、開発者の間でプログラムリストのテンプレートを定めます。これを「プログラミングスタイル」として共有し、誰もが読みやすく、誰もがメンテナンスできるソースプログラムを目指します。図 1.1.4 にプログラミングスタイルの一例を示します。

- (1) 機能ごとに関数にする。
- (2) 1 つの関数内の処理を必要以上に大きくしない (50 行前後が目安)。
- (3) 1 つの行に複数の実行文は書かない。
- (4) 処理ブロックごとに字下げを行う (通常は 4 タブを使用)。
- (5) コメント文を効果的に記述してプログラムの流れを明確にする。
- (6) プログラムを複数のソースファイルから作成する場合は、共通部分を独立した別のファイルにして共有する。

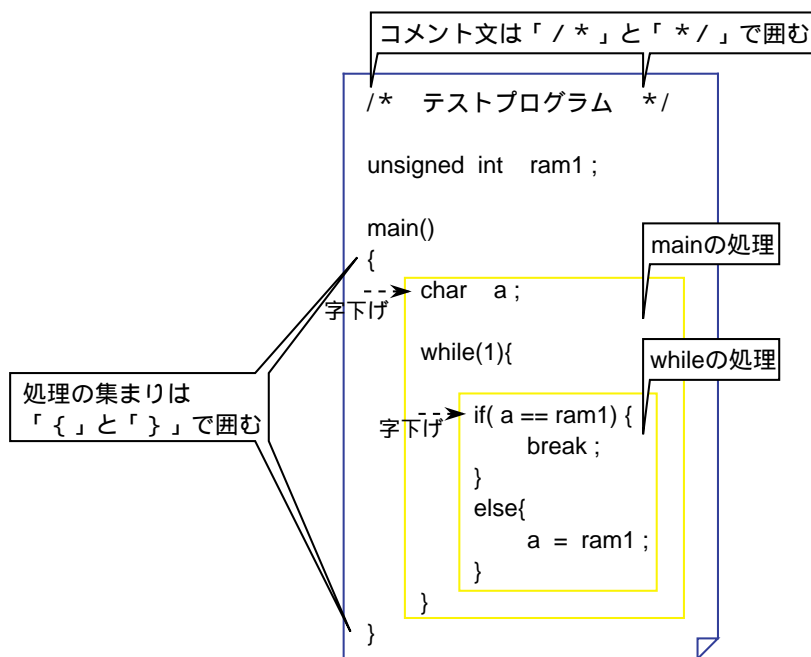


図 1.1.4 C 言語プログラムのプログラミングスタイルの例

コメント文の記述方法

コメント文の記述方法も、読みやすいプログラムを書くための重要なポイントになります。ヘッダとしてファイルや関数の機能を明示したり、プログラムの流れを明確にします。

```

/* ""FILE COMMENT"" *****
 * System Name      : テストプログラム
 * File Name       : TEST.C
 * Version         : 1.00
 * Contents        : テストプログラム
 * Customer        : -----
 * Model           : -----
 * Order           : -----
 * CPU             : M30800MC-XXXFP
 * Compiler        : NC308 (Ver.1.00)
 * OS              : 未使用
 * Programmer      : XXXX
 * Note           : このファイルに含まれるモジュールは再利用可能なように設計されている。
 * *****
 * Copyright, XXXX xxxxxxxxxxxxxxxx CORPORATION
 * *****
 * History         : XXXX.XX.XX      : Start
 * ""FILE COMMENT END"" *****

/* ""プロトタイプ宣言"" *****
void main ( void );
void key_in ( void );
void key_out ( void );

/* ""FUNC COMMENT"" *****
 * ID              : 1.
 * モジュール概要  : メイン関数
 * -----
 * Include         : "system.h"
 * -----
 * 宣言            : void main (void)
 * -----
 * 機能            : 全体の制御
 * -----
 * 引数            : void
 * -----
 * 戻り値          : void
 * -----
 * 入力            : なし
 * 出力            : なし
 * -----
 * 使用関数        : void key_in ( void )      ; 入力関数
 *                  : void key_out ( void )   ; 出力関数
 * -----
 * 注意事項        : 特になし
 * -----
 * History         : XXXX.XX.XX      : Start
 * ""FUNC COMMENT END"" *****

void main ( void )
{
    while(1){          /* 無限ループ */

        key_in();      /* 入力処理 */

        key_out();     /* 出力処理 */

    }
}

```

ファイルヘッダの例

関数ヘッダの例

図 1.1.5 コメントの利用例

コラム NC308 の予約語

表 1.1.2 に示す語は NC308 の予約語になっていますので、変数名や関数名には使用できません。

表 1.1.2 NC308 の予約語一覧

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

1.2 データ型

1.2.1 C 言語で扱える “ 定数 ”

C 言語では「整数」、「実数」、「1文字」、「文字列」の4種類の定数を扱うことができます。
この項では、それぞれの定数を使用するときの記述方法と注意点をまとめます。

整数定数

整数定数は10進数、16進数および8進数の3種類の方法で記述できます。それぞれの表記方法を表1.2.1に示します。また、定数データの場合は大文字と小文字を区別しません。

表 1.2.1 整数定数の表記方法

種類	表記方法	例
10進数	通常の数学表記 (何も付けない)	127, +127, -56
16進数	数字の前に「0x (ゼロ・エックス)」または「0X」を付ける	0x3b, 0X3B
8進数	数字の前に「0 (ゼロ)」を付ける	07, 041

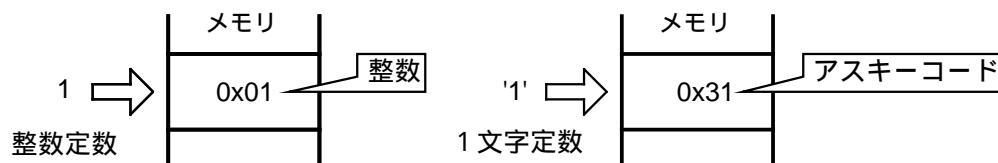
実数定数 (浮動小数点定数)

符号付きの実数を10進数で表記したものを「浮動小数点定数」といいます。表記方法としては通常的小数点表記と、「e」または「E」を用いた指数表記が記述できます。

通常的小数点表記 例: 175.5, -0.007
「e」または「E」を用いた指数表記 例: 1.755e2, -7.0E-3

1文字定数

1文字定数はシングルクォーテーション (') で囲みます。英数字のほかに、制御コードも1文字定数として



扱うことができます。図1.2.1に示すように内部的にはすべてアスキーコードとして扱われます。

図 1.2.1 1 と '1' の違い

文字列定数

英数字や制御コードの並びをダブルクォーテーション (") で囲むと文字列定数として扱うことができます。文字列定数では、データの最後にヌルコード '¥0' が自動的に付けられ、文字列の終わりであることを表します。

例: "abc", "012¥n", "Hello!"

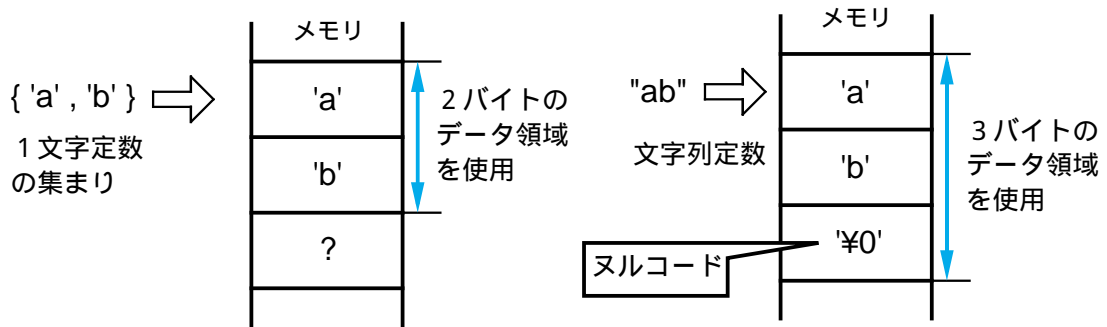


図 1.2.2 { 'a', 'b' } と "ab" の違い

コラム 制御コード一覧 (エスケープシーケンス)

C 言語のプログラムでよく使う制御コード (エスケープシーケンス) を紹介します。

表 1.2.2 C 言語のエスケープシーケンス

表記	内容	表記	内容
¥f	改ページ (FF)	¥'	シングルクォーテーション
¥n	改行復帰 (NL)	¥"	ダブルクォーテーション
¥r	復帰 (CR)	¥x 定数値	16進数
¥t	水平タブ (HT)	¥ 定数値	8進数
¥¥	¥記号	¥0	ヌルコード

1.2.2 変数

C言語のプログラムの中で変数を使用する前には、必ず変数の「データ型」を宣言しなければなりません。データ型は、その変数に割り当てるメモリサイズと、扱う数値の範囲から決めます。

この項では、NC308 で扱える変数のデータ型と宣言方法について説明します。

NC308 の基本データ型

NC308 で扱えるデータ型を表 1.2.3 に示します。() 内は省略して記述できます。

	データ型	ビット長	表現できる数値の範囲
整数	(unsigned) char	8ビット	0 ~ 255
	signed char		- 128 ~ 127
	unsigned short (int)	16ビット	0 ~ 65535
	(signed) short (int)		- 32768 ~ 32767
	unsigned int	16ビット	0 ~ 65535
	(signed) int		- 32768 ~ 32767
	unsigned long (int)	32ビット	0 ~ 4294967295
	(signed) long (int)		- 2147483648 ~ 2147483647
実数	float	32ビット	有効桁数 9桁
	double	64ビット	有効桁数 17桁
	long double	64ビット	有効桁数 17桁

表 1.2.3 NC308 の基本データ型

変数の宣言

変数の宣言は、「データ型 変数名;」という書式で行います。

例：変数 a を char 型として宣言する。

```
char a;
```

「データ型 変数名 = 初期値;」と記述すると、宣言と同時にその変数に対して初期値を設定することができます。

例：char 型の変数 a に初期値として 'A' を設定する。

```
char a = 'A';
```

また複数の変数名をカンマ(',') で区切って列記すると、同じデータ型の変数を同時に宣言できます。

```
例：int i, j;
```

```
例：int i = 1, j = 2;
```

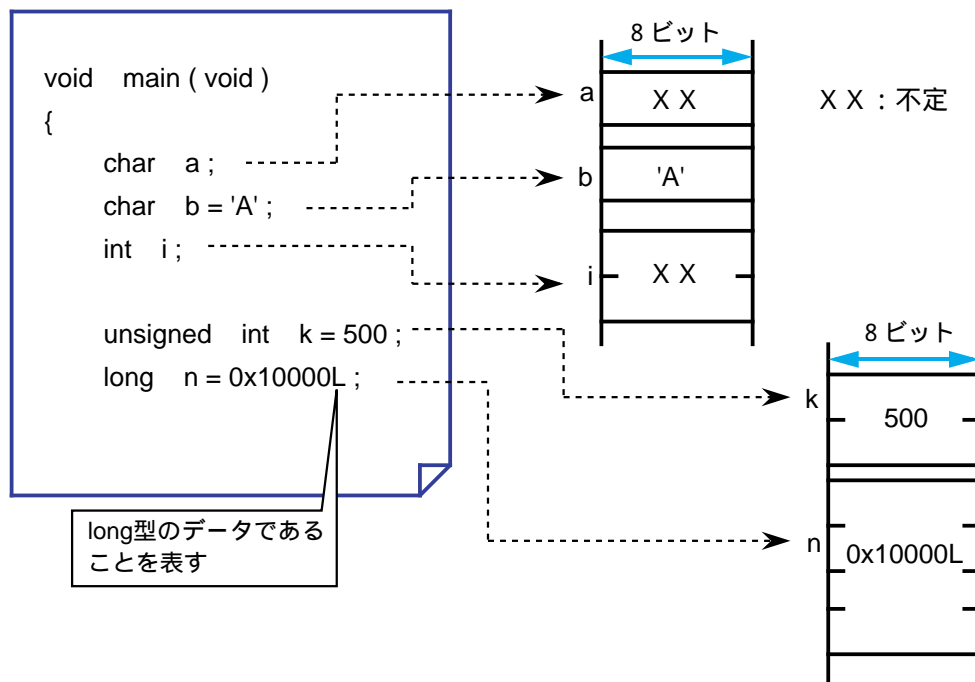


図 1.2.3 変数の宣言

1.2.3 データの特性

変数や定数を宣言するときにデータ型と合わせて、そのデータの特性を記述できます。このときに用いる指定子を「型修飾子」といいます。

この項では、NC308 で扱うデータの特性と指定方法を説明します。

符号付きか、符号なしかを明示する (signed / unsigned 修飾子)

そのデータが符号を持つ場合 " signed " を、符号を持たない場合は " unsigned " を記述します。宣言時にこれらの型修飾子を記述しない場合、NC308 では char 型データのみ符号なし、それ以外のデータはすべて符号付きデータとなります。

```

void main ( void )
{
    char a;
    signed char s_a;

    int b;
    unsigned int u_b;
    :
}

```

" unsigned char a;" と同義

" signed int b;" と同義

図 1.2.4 型修飾子 " signed / unsigned " の記述例

定数データであることを明示する (const 修飾子)

プログラムを実行しても全く値が変化しないデータについては宣言時に " const " を記述します。プログラム中にこの定数データを変化させるような記述があると、NC308 はワーニングを出力します。

```

void main ( void )
{
    char a = 10;
    const char c_a = 20;

    a = 5;
    c_a = 5;
}

```

ワーニング発生!

図 1.2.5 型修飾子 " const " の記述例

コンパイラによる最適化を禁止する (volatile 修飾子)

NC308はプログラム処理上意味のない命令については最適化を行い、不要な命令コードを生成しません。しかしデータによってはプログラムの処理とは関係なく、割り込みやポートからの入力によって変化するものもあります。このようなデータの宣言時には " volatile " を記述します。この型修飾子を記述したデータについては、NC308 は最適化を行わず、命令コードを出力します。

```

char port1;
char volatile port2;

void func(void )
{
    port1 = 0;
    port2 = 0;
    if( port1 == 0 ){
        ...
    }
    if( port2 == 0 ){
        ...
    }
}
    
```

データの宣言時に "volatile" を記述していないため、最適化によって比較が取り除かれ、コードを出力しない

データの宣言時に "volatile" を記述しているため、最適化を行わず、コードを出力する

図 1.2.6 型修飾子 "volatile" の記述例

コラム 宣言の構文

データを宣言するときに型とともに、データの特徴をいろいろな指定子や修飾子を使って記述します。図 1.2.7 に宣言の構文を示します。

図 1.2.7 宣言の構文

宣言指定子			宣言子 (データ名)
記憶クラス指定子 (後述)	型修飾子	型指定子	
static register auto extern	unsigned signed const volatile	int char float struct union	dataname

1.3 演算子

1.3.1 NC308 の演算子

NC308 ではプログラムを記述するために様々な演算子を用意しています。

以下の項では、用途別にこれらの演算子（ただし、アドレス演算子とポインタ演算子は除く^(注)）の記述方法と使用する上での注意点を説明します。

NC308 で使用できる演算子

NC308 で使用できる演算子を表 1.3.1 に示します。

表 1.3.1 NC308 の演算子一覧

単項算術演算子	++ - - -
二項算術演算子	+ - * / %
シフト演算子	<< >>
ビット演算子	& ^
関係演算子	> < >= <= == !=
論理演算子	&& !
代入演算子	= += -= *= /= %= <<= >>= &= = ^=
条件演算子	? :
sizeof演算子	sizeof()
キャスト演算子	(型)
アドレス演算子	&
ポインタ演算子	*
カンマ演算子	,

(注) アドレス演算子とポインタ演算子については、「1.7 配列とポインタ」を参照してください。

1.3.2 数値計算のための演算子

数値計算のために使用される主な演算子は、計算を行う「算術演算子」と結果を格納する「代入演算子」です。この項では、算術演算子と代入演算子を説明します。

単項算術演算子

「単項算術演算子」は、1変数に対して1つの答えを返します。

表 1.3.2 単項算術演算子一覧

演算子	記述形式	内容
++	++変数 (前置形) 変数++ (後置形)	式の値をインクリメントする
--	--変数 (前置形) 変数-- (後置形)	式の値をデクリメントする
-	- 式	式の値の符号を反転した値を返す

インクリメント演算子(++)やデクリメント演算子(--)を、代入演算子や関係演算子と組み合わせて使用するとき、前置形で記述するか後置形で記述するかにより演算結果が変わることがあります。

<例>

前置形；インクリメントまたはデクリメントしてから代入します。

$b = ++a;$ $a = a + 1; b = a;$

後置形；代入してからインクリメントまたはデクリメントします。

$b = a++;$ $b = a; a = a + 1;$

二項算術演算子

通常の四則演算のほかに、整数÷整数の「剰余(あまり)」を求める演算もできます。

表 1.3.3 二項算術演算子一覧

演算子	記述形式	内容
+	式1 + 式2	式1の値と式2の値を加算した結果を返す
-	式1 - 式2	式1の値から式2の値を減算した結果を返す
*	式1 * 式2	式1の値と式2の値を乗算した結果を返す
/	式1 / 式2	式1の値を式2の値で除算した結果を返す
%	式1 % 式2	式1の値を式2の値で割った剰余を返す

コラム NC308 の除算演算子

NC308では除算の演算結果がオーバーフローした場合の演算結果を保証しています。そのため、以下の組み合わせで演算を行った場合は、デフォルトでランタイムライブラリ(__i4divUまたは__i4div)を呼び出します。

```
unsigned int = unsigned long / unsigned int
int = long / int
```

これを強制的に M16C/80 シリーズの div 命令、または divu 命令で展開させる場合は、生成コード変更オプション「- fuse_DIV(- fUD)」を指定します。ただし、演算結果がオーバーフローした場合は、演算結果が不定となります。

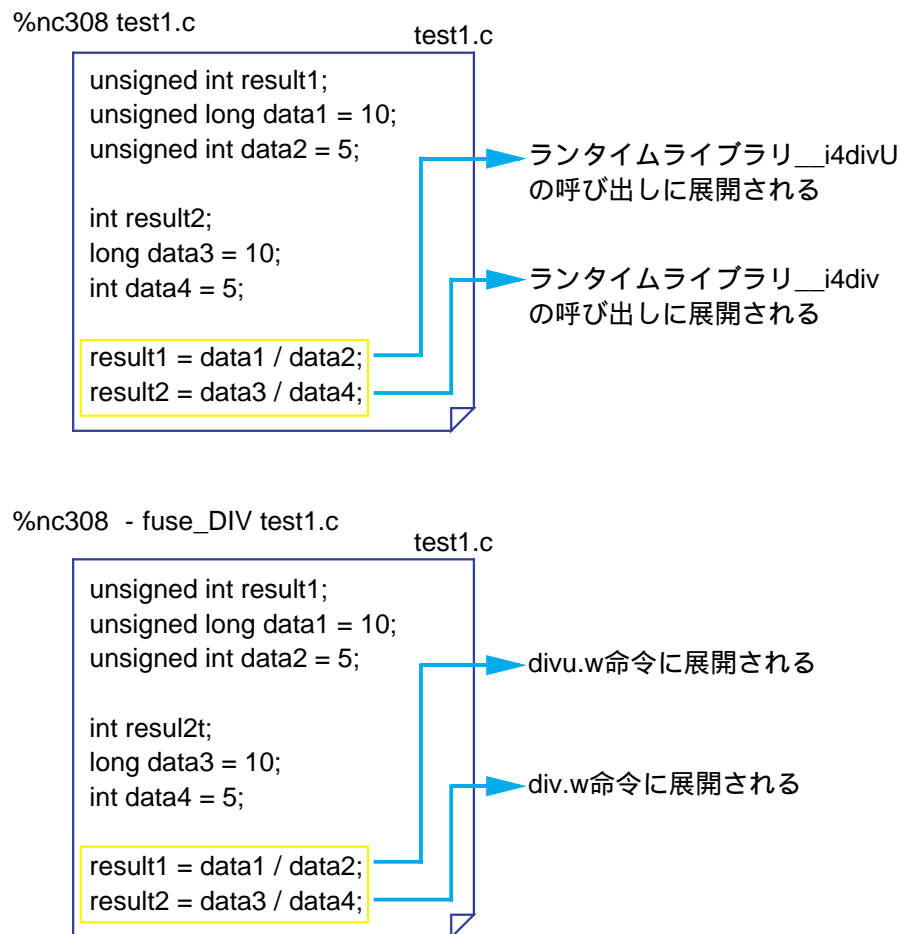


図 1.3.1 NC308 の除算演算子

代入演算子

「式1 = 式2」で、式2の値を式1へ代入します。また、代入演算子 '=' は前述の算術演算子や後述のビット演算子、シフト演算子とも組み合わせて記述できます（「複合代入演算子」）。このとき、必ず代入演算子 '=' を右側に記述します。

表 1.3.4 代入演算子一覧

演算子	記述形式	内容
=	式1 = 式2	式2の値を式1へ代入する
+=	式1 += 式2	式1の値と式2の値を加算し、式1へ代入する
-=	式1 -= 式2	式1の値から式2の値を減算し、式1へ代入する
*=	式1 *= 式2	式1の値と式2の値を乗算し、式1へ代入する
/=	式1 /= 式2	式1の値を式2の値で除算し、式1へ代入する
%=	式1 %= 式2	式1の値を式2の値で割った剰余を式1へ代入する
<<=	式1 <<= 式2	式1の値を式2の値だけ左シフトし、式1へ代入する
>>=	式1 >>= 式2	式1の値を式2の値だけ右シフトし、式1へ代入する
&=	式1 &= 式2	式1の値と式2の値のビット論理積を式1へ代入する
=	式1 = 式2	式1の値と式2の値のビット論理和を式1へ代入する
^=	式1 ^= 式2	式1の値と式2の値のビット排他的論理和を式1へ代入する

コラム 暗黙の型変換

NC308 では、型の違うデータ間で演算を行うとき、以下の規則に従って「暗黙の型変換」を行います。

- ・ビット長が長いデータの型に合わせたのち、演算します。
- ・代入するときは、左辺の型に合わせます。

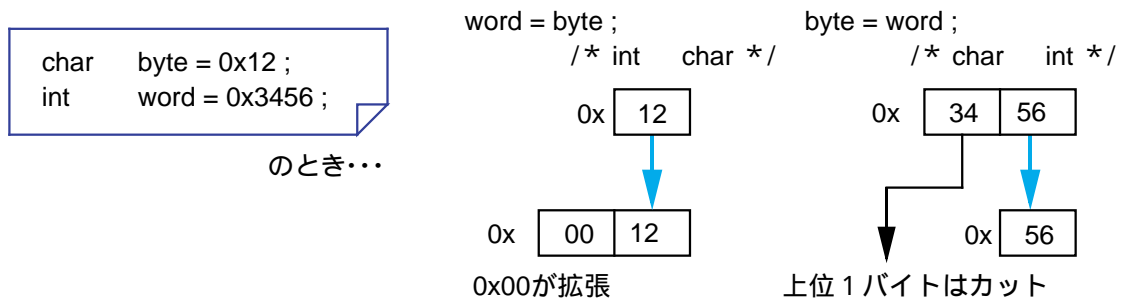


図 1.3.2 型の違うデータを代入する

1.3.3 データ加工のための演算子

データ加工によく使用される演算子は「ビット演算子」と「シフト演算子」です。
この項では、ビット演算子とシフト演算子を説明します。

ビット演算子

ビット演算子を使用すると、データのマスク処理やアクティブ変換を行うことができます。

表 1.3.5 ビット演算子一覧

演算子	記述形式	内容
&	式 1 & 式 2	式 1 の値と式 2 の値のビットごとの論理積を返す
	式 1 式 2	式 1 の値と式 2 の値のビットごとの論理和を返す
^	式 1 ^ 式 2	式 1 の値と式 2 の値のビットごとの排他的論理和を返す
	式	式の値のビット反転を返す

シフト演算子

シフト動作だけでなく、簡単な乗除算にも利用できます（「コラム シフト演算子を使った乗除算」参照）。

表 1.3.6 シフト演算子一覧

演算子	記述形式	内容
<<	式 1 << 式 2	式 1 の値を式 2 の値だけ左シフトした値を返す
>>	式 1 >> 式 2	式 1 の値を式 2 の値だけ右シフトした値を返す

算術シフトと論理シフトの比較

右シフトを実行するとき、対象データが符号付きか符号なしかでシフト動作が異なります。

- ・符号なしのとき 「論理シフト」: 最上位ビットに'0'を挿入します。
- ・符号付きのとき 「算術シフト」: 符号を保持するようにシフト動作をします。つまり、正の数のときは'0'を、負の数のときは'1'を最上位ビットから挿入します。

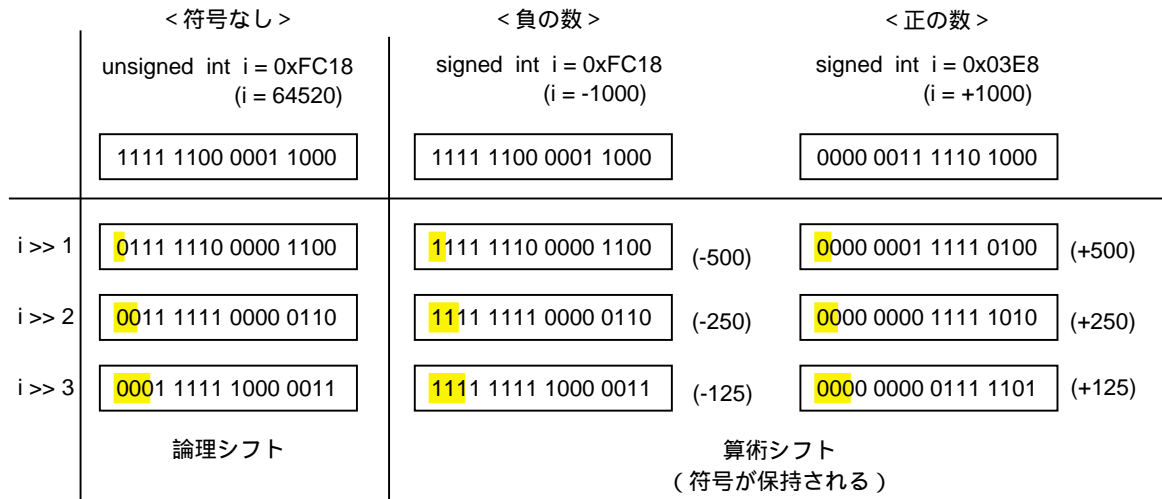


図 1.3.3 算術シフトと論理シフト

コラム シフト演算子を使った乗除算

シフト演算子を使って簡単な乗除算ができます。通常の乗除算演算子を使用するよりも、演算速度が速くなります。NC308 ではこの点を考慮し、" * 2"、" * 4"、" * 8" などに対しては乗算命令ではなくシフト命令を生成します。

- ・乗算：シフト演算と足し算を組み合わせて行います。

```

a * 2    a<<1
a * 3    (a<<1)+a
a * 4    a<<2
a * 7    (a<<2)+(a<<1)+a
a * 8    a<<3
a * 20   (a<<4)+(a<<2)
    
```

- ・除算：下位ビットから押し出されたデータを検出することにより、剰余を知ることもできます。

```

a/4     a>>2
a/8     a>>3
a/16    a>>4
    
```

1.3.4 条件を調べるための演算子

制御文の中で条件を調べるために使用するのが「関係演算子」と「論理演算子」です。どちらの演算子も、条件が成立しているときは'1'、条件が成立していないときには'0'を返します。

この項では、関係演算子と論理演算子を説明します。

関係演算子

2つの式の間の大小関係を調べます。そして結果が真ならば'1'を、偽ならば'0'を返します。

表 1.3.7 関係演算子一覧

演算子	記述形式	内容
<	式 1 < 式 2	式 1 の値が式 2 の値より小さければ真、それ以外は偽
<=	式 1 <= 式 2	式 1 の値が式 2 の値より小さいか等しければ真、それ以外は偽
>	式 1 > 式 2	式 1 の値が式 2 の値より大きければ真、それ以外は偽
>=	式 1 >= 式 2	式 1 の値が式 2 の値より大きい等しければ真、それ以外は偽
==	式 1 == 式 2	式 1 の値が式 2 の値と等しければ真、それ以外は偽
!=	式 1 != 式 2	式 1 の値が式 2 の値と等しくなければ真、それ以外は偽

論理演算子

関係演算子とともに用いる演算子で、複数の条件式の組み合わせ条件を調べます。

表 1.3.8 論理演算子一覧

演算子	記述形式	内容
&&	式 1 && 式 2	式 1 と式 2 の両方が真ならば真、それ以外は偽
	式 1 式 2	式 1 と式 2 の両方が偽ならば偽、それ以外は真
!	!式	式が真ならば偽、偽ならば真

1.3.5 その他の演算子

この項では、C 言語ならではのちょっと変わった性質を持つ 4 種類の演算子を説明します。

条件演算子

条件式が真ならば式 1 を、偽ならば式 2 を実行する演算子です。条件式、式 1、式 2 ともに処理の記述が短いときに使用すると、条件分岐のコーディングを簡単にできます。条件演算子を表 1.3.9 に、利用例を図 1.3.4 に示します。

表 1.3.9 条件演算子

演算子	記述形式	内容
? :	条件式 ? 式 1 : 式 2	条件式が真ならば式 1 を、偽ならば式 2 を実行する

・大きい方の値を選択する

`c = a > b ? a : b;`

=

```
if (a > b){
    c = a;
}
else{
    c = b;
}
```

・絶対値を求める

`c = a > 0 ? a : -a;`

=

```
if(a > 0){
    c = a;
}
else{
    c = -a;
}
```

図 1.3.4 条件演算子の利用例

sizeof 演算子

あるデータ型または式が使用しているメモリのバイト数を知りたいときに使用します。

表 1.3.10 sizeof 演算子

演算子	記述形式	内容
sizeof()	sizeof 式 sizeof(データ型)	式またはデータ型のメモリ使用量をバイト単位で返す

キャスト演算子

異なる型どうして演算を行うと、演算で使用するデータは暗黙のうちに式中のいちばん大きなデータ型へと変換されます。しかし、これが思わぬ不具合の原因となる可能性があるため、「キャスト演算子」を利用して型変換を明示します。

表 1.3.11 キャスト演算子

演算子	記述形式	内容
()	(新データ型)変数	変数のデータ型を新データ型に変換する

カンマ（順次）演算子

式 1 から式 2 へと、左から順に実行していきます。短い記述の処理を羅列するときに使用します。

表 1.3.12 カンマ（順次）演算子

演算子	記述形式	内容
,	式 1 , 式 2	式 1、式 2 と左から順に実行する

1.3.6 演算子の優先順位

数学の演算子と同じように C 言語で使用する演算子にも「優先順位」と「結合規則」があります。
この項では、演算子の優先順位と結合規則について説明します。

優先順位と結合規則

1 つの式の中に複数の演算子が含まれているとき、まず「優先順位」の高いものから演算していきま。 「結合規則」は同じ優先順位の演算子が複数存在するとき、左右どちらから計算するのかを示しています。

表 1.3.13 演算子の優先順位

優先順位	演算子の種類	演算子	結合規則
高	式	() [] . (_注 - >	
↑	単項算術演算子 etc	! ++ - - - * (_注 & (_注 sizeof () (型)	
	乗除算演算子	* (_注 /) % ₂ ₃	
	加減算演算子	+ ₄ -	
	シフト演算子	<< >>	
	関係演算子 (比較)	< <= > >=	
	関係演算子 (等価)	== !=	
	ビット演算子 (AND)	&	
	ビット演算子 (EOR)	^	
	ビット演算子 (OR)		
↓	論理演算子 (AND)	&&	
	論理演算子 (OR)		
	条件演算子	?:	
	代入演算子	= += -= *= /= %= <<= >>= &= ^= =	
低	カンマ演算子	,	

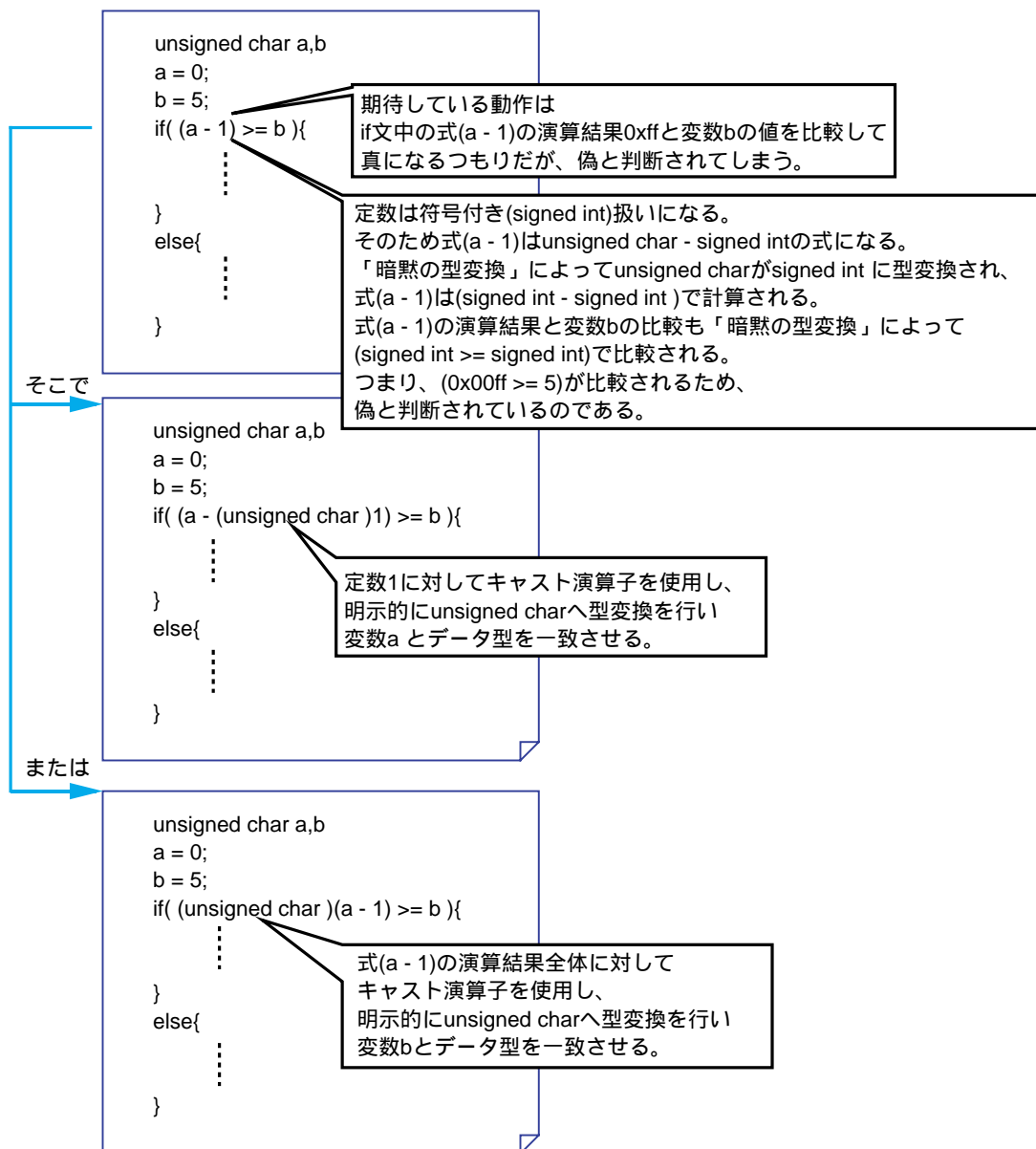
(注 1) '!' は構造体と共用体のメンバを指定するメンバ演算子です。
 (注 2) '*' はポインタ変数を表すポインタ演算子です。
 (注 3) '&' は変数のアドレスを表すアドレス演算子です。
 (注 4) '*' は乗算を表す算術演算子です。

1.3.7 間違いやすい演算子の使用例

演算子の「暗黙の型変換」や「優先順位」の解釈を間違えると、プログラムが期待通りに動作しません。この項では間違いやすい演算子の使用例とその対処方法について説明します。

例 1.3.1 「暗黙の型変換」の解釈間違いと対処方法

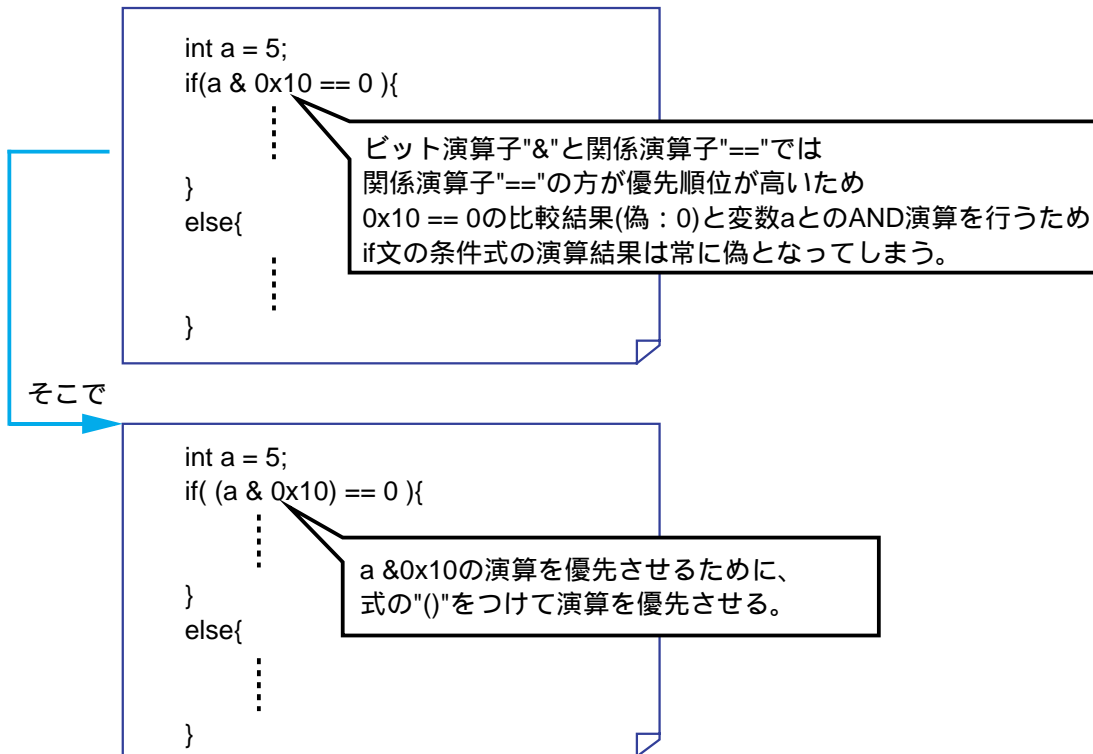
NC308 では型の違うデータ間で演算を行うとき、「暗黙の型変換」によってビット長が長いデータ型に型を合わせた後に演算が行われます。期待通りに動作させるためには、キャスト演算子を使用した明示的な型変換で対応します。



例 1.3.1 「暗黙の型変換」の解釈間違いと対処方法

例 1.3.2 「優先順位」の解釈間違いと対処方法

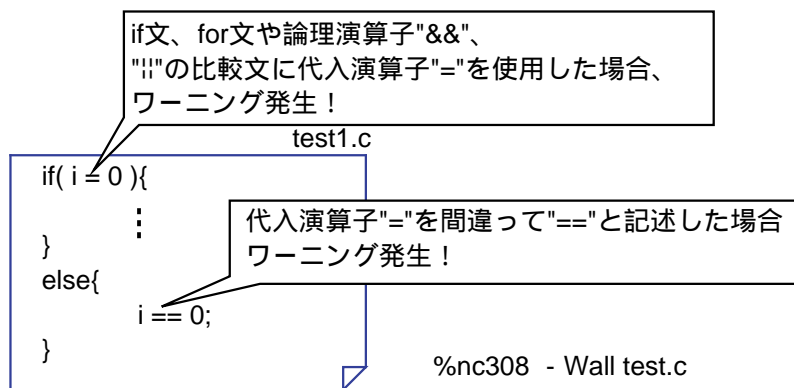
1つの式の中に複数の演算子が含まれる場合は、演算子の「優先順位」と「結合規則」を正しく解釈する必要があります。また、期待通りに動作させるためには、式の"()"を使用して対応します。



例 1.3.2 「優先順位」の解釈間違いと対処方法

コラム 演算子の誤った使用方法の検出 (警告オプション「-Wall」)

NC308の警告オプション「-Wall」を使用して演算子の誤った使用方法を検出することができます^(注)。そのほかに警告オプション「-Wall」は「-Wnon_prototype(-WNP)」や「-Wunknown_pragma(-WUP)」と同等の警告を表示します。



例 1.3.3 演算子の誤った使用方法の検出

(注) コンパイラの判断で誤った記述と推測できる範囲で検出します。

1.4 制御文

1.4.1 プログラムの構造化

C 言語では構造化プログラミングの基本処理「順次処理」、「分岐処理」、「繰り返し処理」をすべて制御文を使用して記述できます。したがって、C 言語で記述されたプログラムはすべて構造化されており、これが処理の流れが読みやすい理由です。

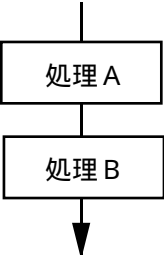
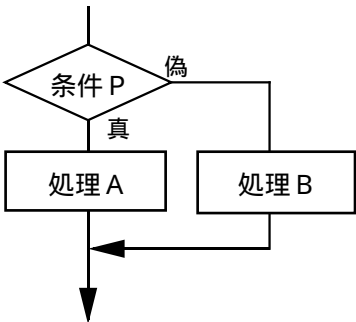
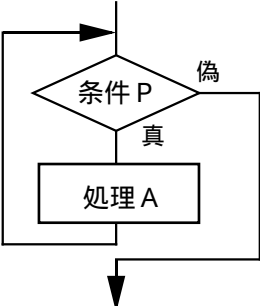
以下の項では、これらの制御文の記述方法と使用例を説明します。

プログラムの構造化

プログラムをわかりやすくするための最大のポイントは「いかにプログラムの流れを読みやすくするか」ということです。そこでプログラムの流れを好き勝手にするのではなく、「順次処理」、「分岐処理」、「繰り返し処理」の3つに限定しようという動きが盛んになりました。これが「構造化プログラミング」と呼ばれる手法です。

表 1.4.1 に、構造化プログラミングの3つの基本形を示します。

表 1.4.1 構造化プログラミングの3つの基本形

<p>順次処理</p>		<p>上から下へ、トップダウンで実行する。</p>
<p>分岐処理</p>		<p>条件 P の真偽によって処理 A と処理 B に振り分ける。</p>
<p>繰り返し処理</p>		<p>条件 P が成立している間、処理 A を繰り返す。</p>

1.4.2 条件による処理の分岐 (分岐処理)

分岐処理を記述するための制御文は、「if - else 文」、「else - if 文」、「switch - case 文」です。
この項では、これらの制御文の記述方法と使用例を説明します。

if - else 文

ある条件が真の場合はすぐ次のブロックを、偽であれば「else」のブロックを実行します。
"else" のブロックは省略できます。

・ else文を省略した場合

図 1.4.1 if - elseelse - if 文の記述例

例 1.4.1 カウントアップ (if - else 文の記述例)

秒のカウンタ "second" と分のカウンタ "minute" をカウントアップします。このモジュールを 1 秒間隔で呼び出すと、時計の役割をはたします。

```

void count_up(void);
unsigned int second = 0;
unsigned int minute = 0;

void count_up(void)
{
    if(second >= 59){
        second = 0;
        minute ++;
    }
    else{
        second ++;
    }
}
    
```

"count_up"関数を宣言 (「1.5 関数」を参照)
 "second" (秒のカウンタ) と "minute" (分のカウンタ) 用の変数を宣言
 "count_up"関数の定義
 59秒以上であれば「秒」をリセットし「分」をカウントアップする
 59秒未満であれば「秒」をカウントアップする

例 1.4.1 カウントアップ (if - else 文の記述例)

else - if 文

複数の条件により 3 個以上の処理に分けたいときに使用します。各条件が真のときに実行したい処理をすぐ次のブロックに記述します。すべての条件に当てはまらなかったときの処理を最後の "else" のブロックに記述します。

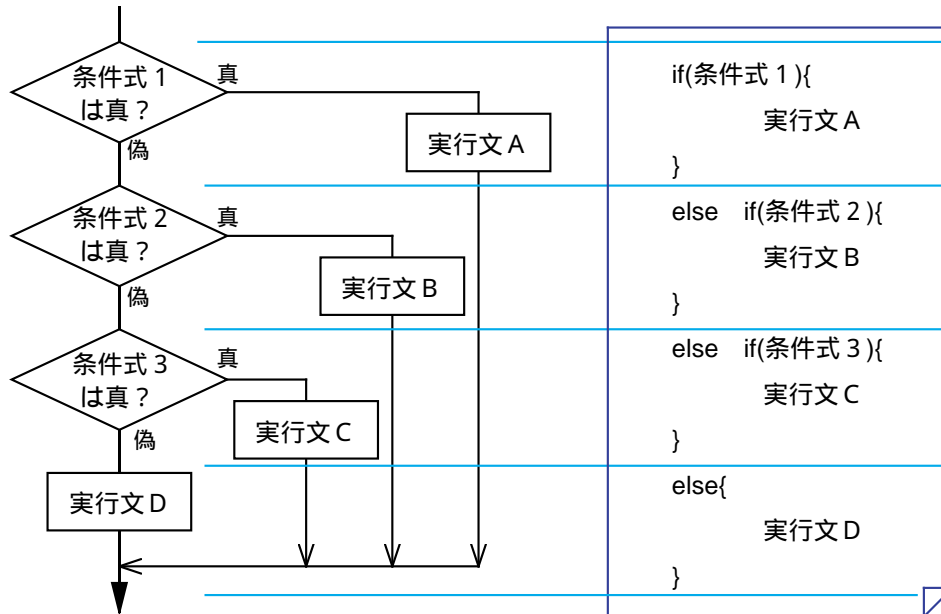


図 1.4.2 else - if 文の記述例

例 1.4.2 四則演算の切り換え - 1 - (else - if 文の記述例)

入力データ "sw" の内容により、実行する演算を切り換えます。

```

void select(void);
int a = 29, b = 40;
long int ans;
char sw;

void select(void)
{
    if(sw == 0){
        ans = a + b;
    }
    else if(sw == 1){
        ans = a - b;
    }
    else if(sw == 2){
        ans = a * b;
    }
    else if(sw == 3){
        ans = a / b;
    }
    else{
        error();
    }
}
    
```

← "select"関数を宣言 (「1.5 関数」を参照)
 ← 使用する変数を宣言
 ← "select"関数の定義
 ← "sw"の内容が 0 であれば 足し算を実行する
 ← "sw"の内容が 1 であれば 引き算を実行する
 ← "sw"の内容が 2 であれば かけ算を実行する
 ← "sw"の内容が 3 であれば わり算を実行する
 ← "sw"の内容が 4 以上であれば エラー処理を行う

例 1.4.2 四則演算の切り換え - 1 - (else - if 文の記述例)

switch - case 文

ある式の結果により複数の処理に分岐します。式の結果は定数として判定しますので、関係演算子などはいりません。

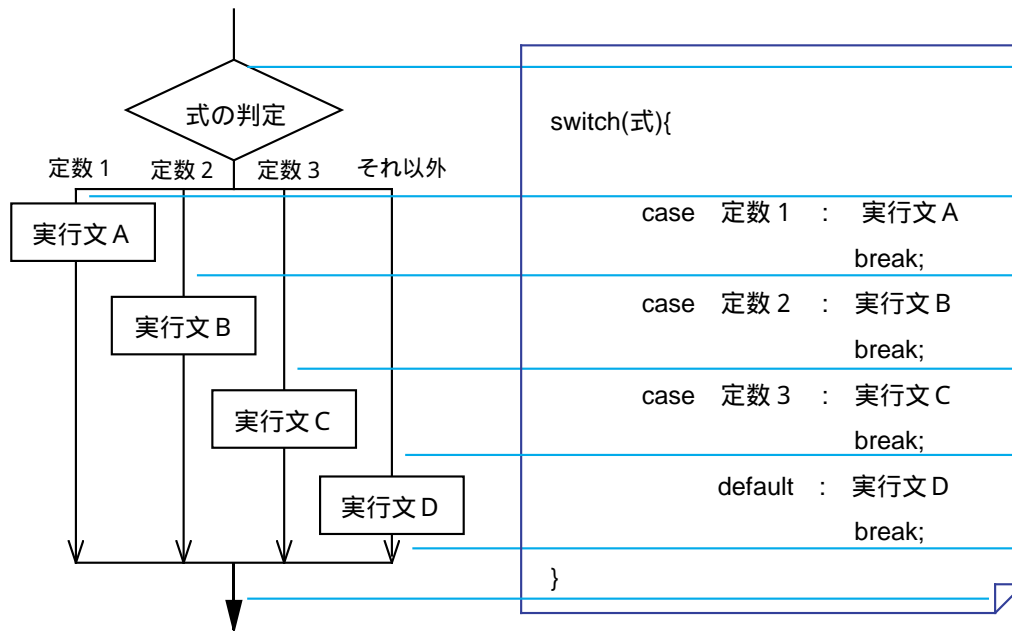


図 1.4.3 switch - case 文の記述例

例 1.4.3 四則演算の切り換え - 2 - (switch - case 文の記述例)

入力データ "sw" の内容により、実行する演算を切り換えます。

```

void select(void); ← "select"関数を宣言 (「1.5 関数」を参照)

int a = 29, b = 40; ← 使用する変数を宣言
long int ans;
char sw;

void select(void) ← "select"関数の定義
{
    switch(sw){ ← "sw"の内容を判定する
        case 0: ans = a + b; ← "sw"の内容が0であれば足し算を実行する
            break;
        case 1: ans = a - b; ← "sw"の内容が1であれば引き算を実行する
            break;
        case 2: ans = a * b; ← "sw"の内容が2であればかけ算を実行する
            break;
        case 3: ans = a / b; ← "sw"の内容が3であればわり算を実行する
            break;
        default: error(); ← "sw"の内容が4以上であればエラー処理を行う
            break;
    }
}
    
```

例 1.4.3 四則演算の切り換え - 2 - (switch - case 文の記述例)

コラム break を入れない switch - case 文

switch - case 文の各実行文の終わりには通常 break 文を記述します。

break 文を記述していないブロックがあれば、そのブロックを終了すると次のブロックへと上から順にブロックを実行していきます。つまり、式の値により処理のスタート位置を変えることができます。

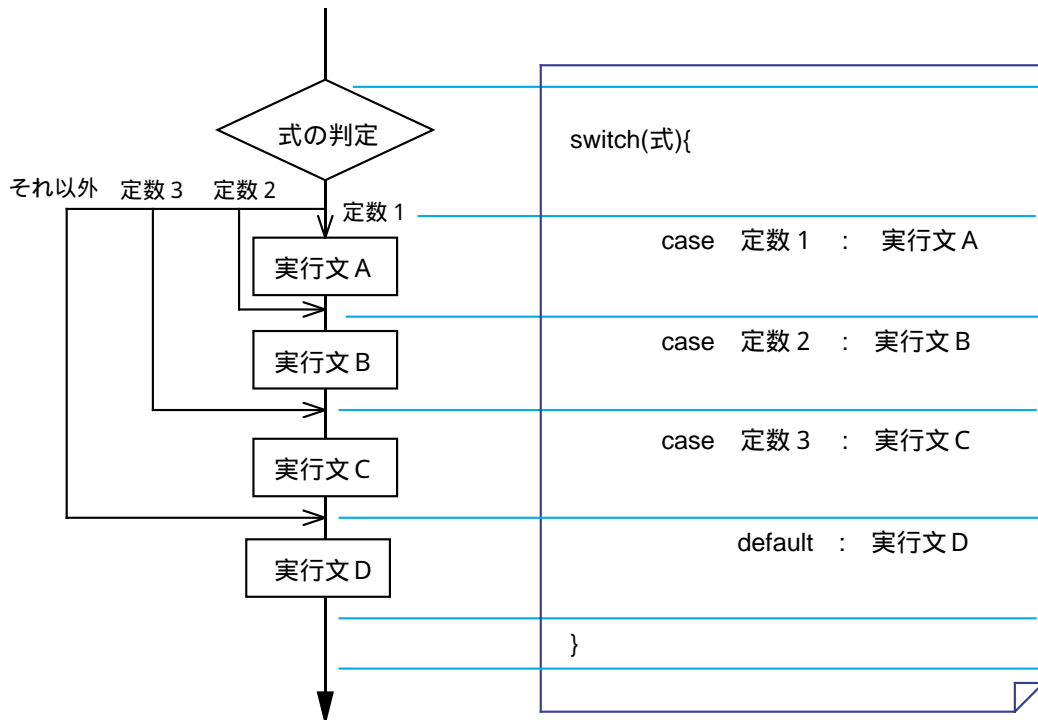


図 1.4.4 break を入れない switch - case 文

1.4.3 同じ処理の繰り返し (繰り返し処理)

繰り返し処理を記述するための制御文は「while 文」、「for 文」、「do - while 文」です。
この項では、これらの制御文の記述方法と使用例を説明します。

while 文

条件式が成立している間、ブロック内の処理を繰り返し実行します。条件式に 0 以外の定数を記述しておく、条件式が常に「真」となり、無限ループを実現できます。

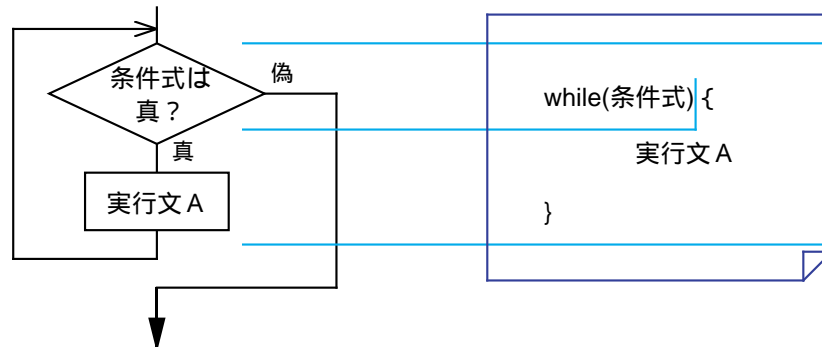


図 1.4.5 while 文の記述例

例 1.4.4 総和を求める - 1 - (while 文の記述例)

1 から 100 までの整数の和を求めます。

```

void sum(void); ← "sum"関数を宣言 (「1.5 関数」を参照)
unsigned int total = 0; ← 使用する変数を宣言
void sum(void) ← "sum"関数の定義
{
    unsigned int i = 1; ← カウンタ用の変数を宣言、初期化
    while(i <= 100){ ← カウンタの内容が100になるまでループする
        total += i; ← カウンタの内容を変化させる
        i++;
    }
}
    
```

例 1.4.4 総和を求める - 1 - (while 文の記述例)

for 文

例題 1.4.4 のようにカウンタを使って繰り返し処理を行うとき、条件の判定とともに必ず「カウンタの内容の初期化」と「カウンタの内容の変化」という作業が必要です。for 文ではこれらの作業が条件式とともに記述できます（図 1.4.6）。初期化（式 1）、条件式（式 2）、処理（式 3）はそれぞれ省略可能です。しかし、いずれの式を省略した場合でも間のセミコロン ';' は必ず記述してください。また、for 文と前述の while 文は常にかき換えられます。

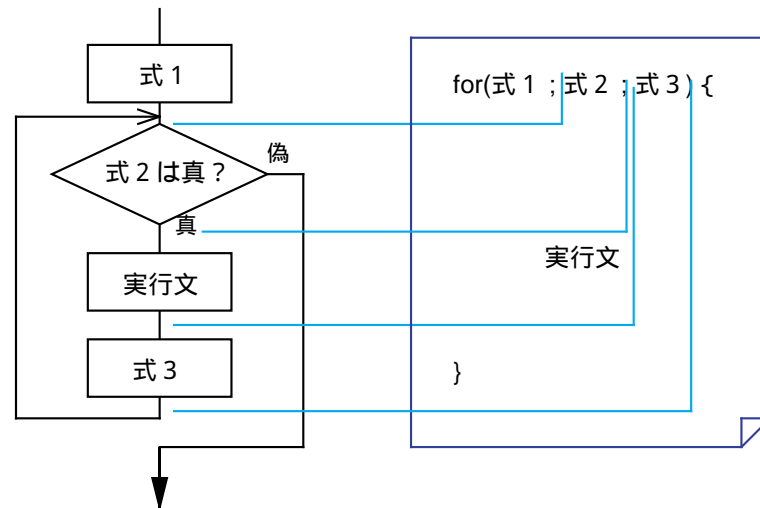


図 1.4.6 for 文の記述例

例 1.4.5 総和を求める - 2 - (for 文の記述例)

1 から 100 までの整数の和を求めます。

```

void sum(void); ← "sum"関数を宣言（「1.5 関数」を参照）
unsigned int total = 0; ← 使用する変数を宣言
void sum(void) ← "sum"関数の定義
{
    unsigned int i; ← カウンタ用の変数を宣言
    for(i = 1 ; i <= 100 ; i++){ ← カウンタの内容が 1 から 100 になるまでループする
        total += i;
    }
}
    
```

例 1.4.5 総和を求める - 2 - (for 文の記述例)

do - while 文

for文やwhile文とは異なり、処理を実行したのち条件判定を行います(「後判定」)。for文やwhile文では、条件によって1度も処理が実行されない場合がありますが、do - while文では必ず1度は処理が行われます。

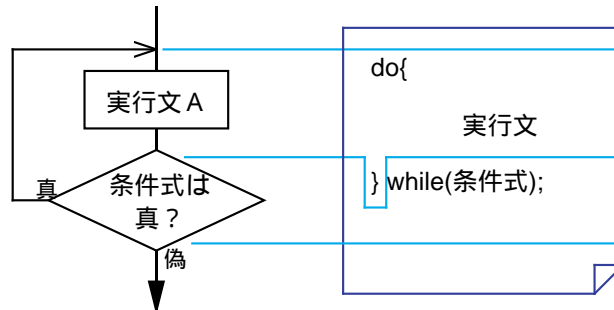


図 1.4.7 do - while 文の記述例

例 1.4.6 総和を求める - 3 - (do - while 文の記述例)

1 から 100 までの整数の和を求めます。

```

void sum(void); ← "sum"関数を宣言(「1.5 関数」を参照)
unsigned int total = 0; ← 使用する変数を宣言
void sum(void) ← "sum"関数の定義
{
    unsigned int i = 0; ← カウンタ用の変数を宣言、初期化
    do{
        i++;
        total += i;
    }while(i < 100); ← カウンタの内容が1から100になるまでループする
}
    
```

例 1.4.6 総和を求める - 3 - (do - while 文の記述例)

1.4.4 処理の中断

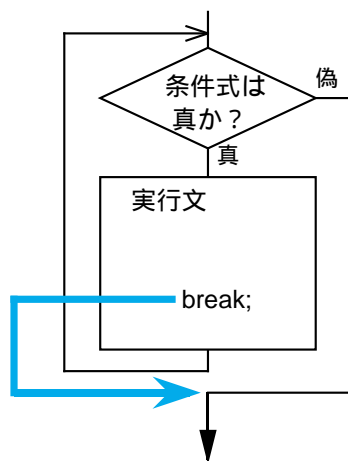
処理を中断して脱出するための制御文（「補助制御文」）には、「break 文」、「continue 文」、「goto 文」があります。

この項では、これらの制御文の記述方法と動作を説明します。

break 文

繰り返し処理や switch - case 文の中で使用します。「break ;」が実行されると、処理を中断して 1 ブロックだけ脱出します。

・ while 文に使用した場合



・ for 文に使用した場合

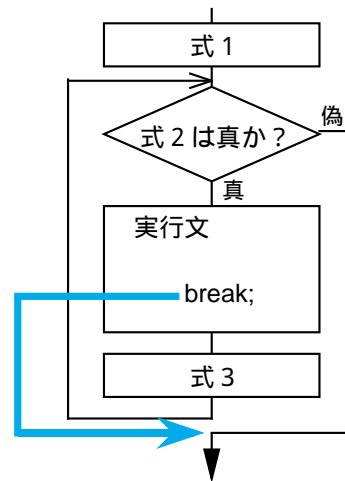
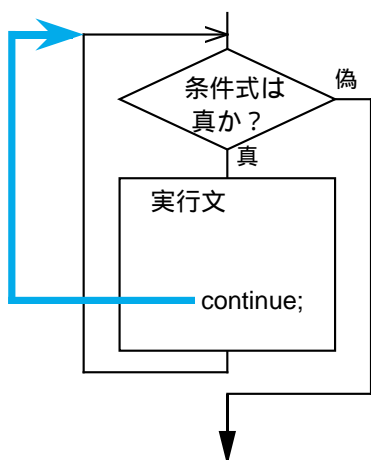


図 1.4.8 break 文の記述例

continue 文

繰り返し処理の中で使用します。「continue ;」が実行されると、処理を中断します。中断後 while 文では条件判定に戻り、for 文では式 3 を実行した後、条件判定に戻ります。

・ while 文に使用した場合



・ for 文に使用した場合

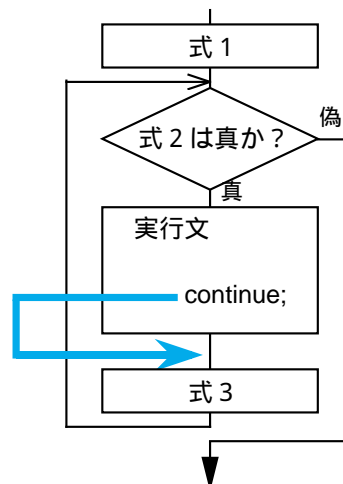


図 1.4.9 continue 文の記述例

goto 文

goto文が実行されると、goto文の後ろに記述されているラベルへ無条件に分岐します。break文やcontinue文とは違い、多重ブロックを一気に脱出して関数内のどの場所にも分岐することができます(図 1.4.10)。しかし、構造化プログラミングに反するため、エラー処理など緊急を要する場合にのみ、使用するほうがよいでしょう。

また、分岐先のラベルの後ろには必ず実行文を記述しなければなりません。なにもしたくない場合は空文(';'のみ)を記述します。

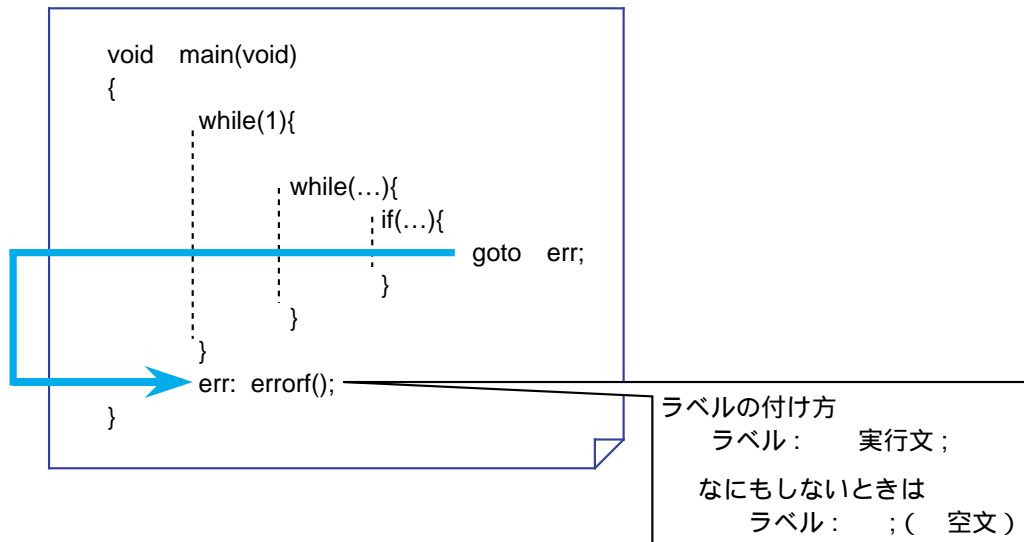


図 1.4.10 goto 文の働き

1.5 関数

1.5.1 関数とサブルーチン

アセンブリ言語のプログラムがサブルーチンを基本単位としているのと同じように、C 言語では「関数」がプログラムの基本単位となります。

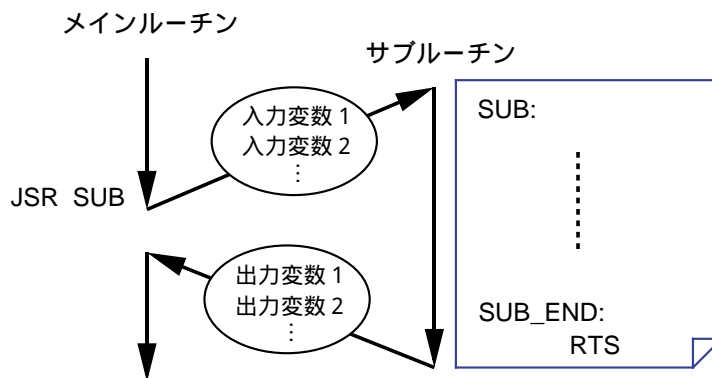
以下の項では、NC308 での関数の記述方法について説明します。

引数と戻り値

関数間のデータのやりとりは、サブルーチンの入力変数にあたる「引数」と出力変数にあたる「戻り値」で行います。

アセンブリ言語の場合は入力変数や出力変数の数に制約はありません。しかし、C 言語では戻り値については各関数につき 1 個と決められており、「return 文」を使って返します。引数については制約はありません。^(注)

・アセンブリ言語の「サブルーチン」



・C 言語の「関数」

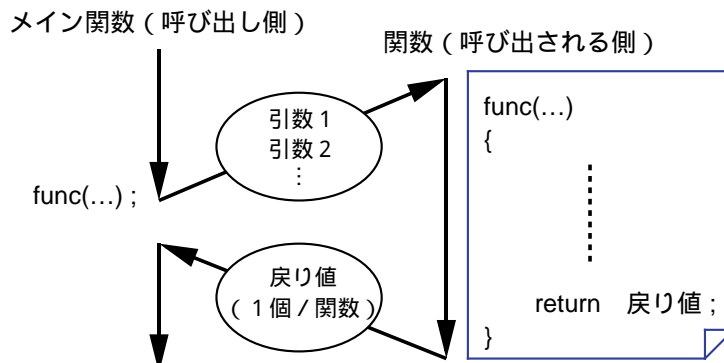


図 1.5.1 「サブルーチン」と「関数」

(注) ROM 化を目的とするコンパイラには、引数の数を制限しているものもあります。

1.5.2 関数の作成

関数を使用するためには「関数の型宣言 (プロトタイプ宣言)」、「関数の定義」、「関数の呼び出し」の3つの手続きが必要です。

この項では、これらの手続きの方法を説明します。

関数の型宣言 (プロトタイプ宣言)

C 言語で関数を使用する前には、必ず関数の型宣言 (プロトタイプ宣言) を行わなければなりません。関数の型とは、関数の引数と戻り値のデータ型です。

次に関数の型宣言 (プロトタイプ宣言) の書式を示します。

```
戻り値のデータ型 関数名 (引数のデータ型のならび);
```

戻り値や引数がないときは、空 (から) を意味する "void" という型を記述します。

関数の定義

関数本体では、引数を受け取るための「仮引数」のデータ型と名称を定義します。また戻り値は「return 文」を使って返します。

次に関数定義の書式を示します。

```
戻り値のデータ型 関数名 (仮引数 1 のデータ型 仮引数 1 ,  
{  
    :  
    return 戻り値 ;  
}
```

関数の呼び出し

関数を呼び出すとき、その関数に対する引数を記述します。また呼び出した関数からの戻り値は代入演算子を用いて受け取ります。

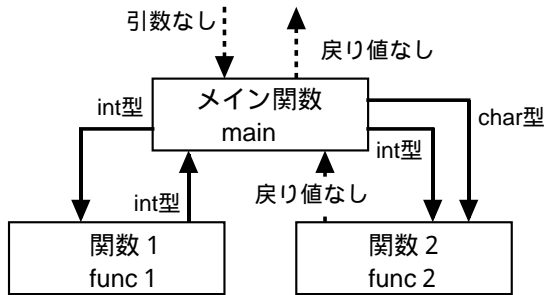
```
関数名 (引数 1 , );
```

戻り値があるとき

```
変数 = 関数名 (引数 1 , );
```

関数の記述例

次のような関係のある3つの関数を記述します。



```

/* プロトタイプ宣言 */
void main (void);
int func1 (int);
void func2 (int, char);

/* メイン関数 */
void main()
{
    int a = 40, b = 29;
    int ans;
    char c = 0xFF;

    ans = func1 ( a );
    func2 ( b, c );
}

/* 関数 1 の定義 */
int func1 (int x)
{
    int z;

    z = x + 1;
    return z;
}

/* 関数 2 の定義 */
void func2 (int y, char m)
{
    :
}
    
```

aを引数として関数 1 ("func1") を呼び出す
戻り値は"ans"に代入する

b,cを引数として関数 2 ("func2") を呼び出す
戻り値はなし

「return文」で戻り値を返す

図 1.5.2 関数の記述例

1.5.3 関数間でのデータの受け渡し

C 言語では引数と戻り値の受け渡しは、各変数の値をコピーして渡します(「Call by Value」)。したがって、関数を呼び出すときの引数の名前と、呼び出された関数が受け取る引数(仮引数)の名前を一致させる必要はありません。

呼び出された関数内での処理はコピーした仮引数を使って行われるので、呼び出し側の引数本体が破壊されることはありません。

これらの理由により、C 言語の関数は独立性が高く、関数の再利用も容易です。

この項では、関数間でのデータの受け渡しについて説明します。

例 1.5.1 和を求める(関数の記述例)

- 32768 ~ 32767 の範囲にある任意の 2 つの整数を引数として、その和を求める和算関数 "add" を作成し、メイン関数から呼び出します。

```

/* プロトタイプ宣言 */
void main ( void );
long add ( int , int );

/* メイン関数 */
void main ( void )
{
    long int answer;
    int a = 29, b = 40;

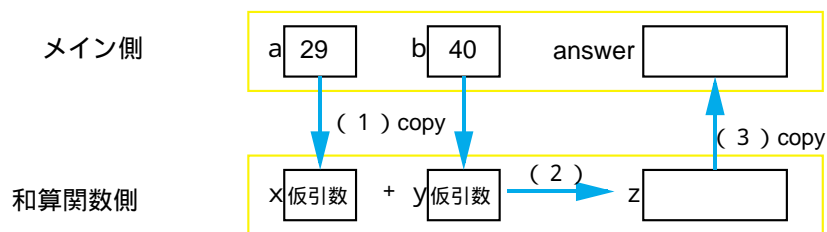
    answer = add ( a, b );
}

/* 和算関数 */
long add ( int x, int y )
{
    long int z;

    z = ( long int ) x + y;
    return z;
}

```

<データの流れ>



例 1.5.1 和を求める(関数の記述例)

1.6 記憶クラス

1.6.1 変数と関数の有効範囲

変数や関数はプログラム全体で使用するもの、1関数でのみ使用するものなど、各々の性質によりその有効範囲が異なります。このような変数や関数の有効範囲のことを「記憶クラス (Scope)」といいます。

以下の項では、変数と関数の記憶クラスの種類とその指定方法について説明します。

変数と関数の有効範囲

C言語のプログラムは複数のソースファイルから構成されています。さらに、1つのソースファイルは複数の関数によって構成されています。つまり、C言語のプログラムは、図 1.6.1に示すような階層構造になっています。

変数には、次の3段階の記憶クラスがあります。

- (1) 関数内でのみ有効
- (2) ファイル内でのみ有効
- (3) プログラム全体で有効

関数には、次の2段階の記憶クラスがあります。

- (1) ファイル内でのみ有効
- (2) プログラム全体で有効

C言語ではこれらの記憶クラスを変数、関数ごとに指定することができます。そして、この記憶クラスを効率よく利用することによって、自作の変数や関数にプロテクトをかけたり、逆にチーム内で共有したりすることができます。

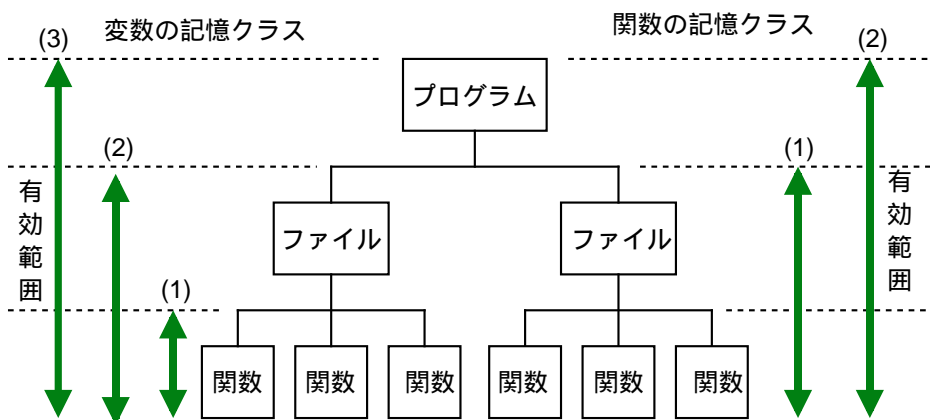


図 1.6.1 C言語プログラムの階層構造と記憶クラス

1.6.2 変数の記憶クラス

変数の記憶クラスは、型宣言を行うときに指定します。ポイントは次の2点です。

- (1) 外部変数と内部変数 (型宣言を行う場所)
- (2) 記憶クラス指定子 (型宣言に指定子を付加)

この項では、変数に対する記憶クラスの指定方法を説明します。

外部変数と内部変数

変数の有効範囲を指定するいちばん簡単な方法で、変数の型宣言を行う位置によって有効範囲が決まります。関数の外側で宣言した変数を「外部変数」、関数の内側で宣言した変数を「内部変数」といいます。「外部変数」は宣言以降どの関数からも参照できるグローバルな変数です。一方、「内部変数」は宣言以降その関数内でのみ有効なローカル変数となります。

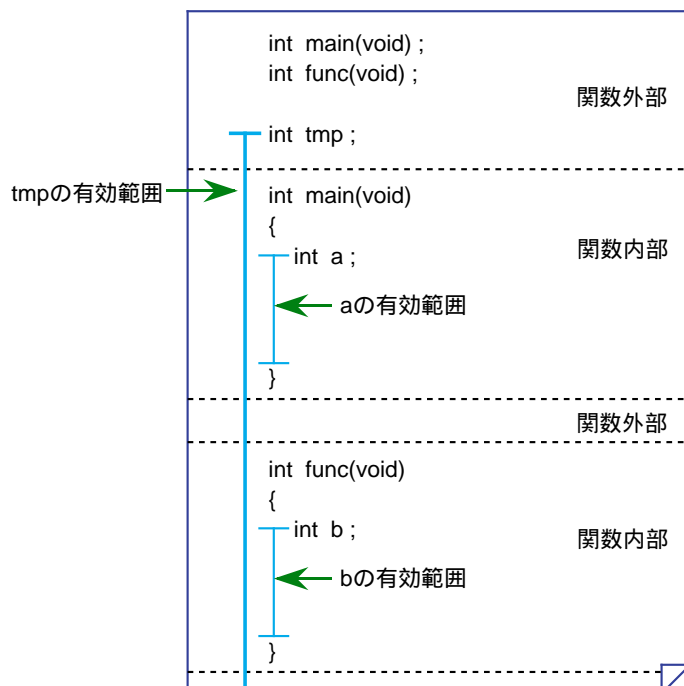


図 1.6.2 外部変数と内部変数

記憶クラス指定子

変数に対して使用する記憶クラス指定子は「auto」、「static」、「register」、「extern」があります。これらの記憶クラス指定子は外部変数に対して使用したときと、内部変数に対して使用したときでは働きが異なります。書式を次に示します。

記憶クラス指定子 データ型 変数名 ;

外部変数の記憶クラス

外部変数を宣言するときに、記憶クラス指定子をなにも付けなければプログラム全体で有効なグローバル変数となります。一方、宣言時に " static " を記述すると、宣言したファイル内のみ有効な変数となります。

図 1.6.3 のソースファイル 2 の " mode " のような、別ファイルで宣言した外部変数を使用する場合は " extern " を記述します。

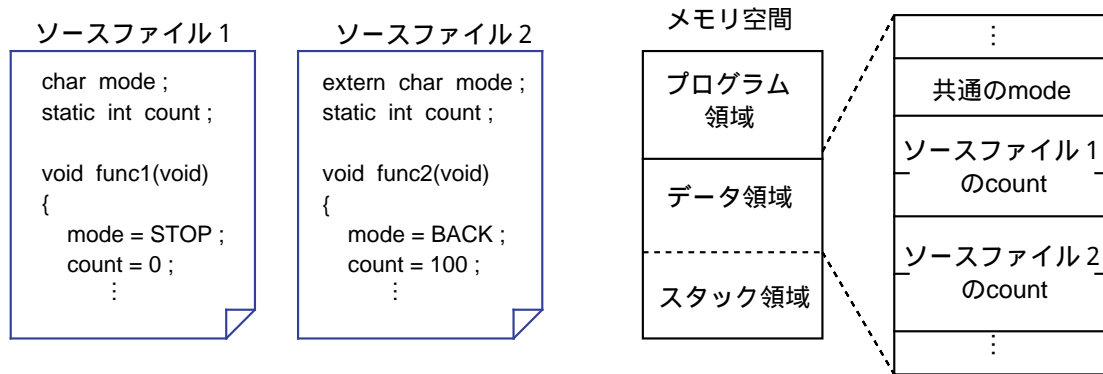


図 1.6.3 外部変数の記憶クラス

内部変数の記憶クラス

記憶クラス指定子を付けずに宣言した内部変数はスタック上に領域がとられるため、その関数が呼び出される度に初期化されます。一方、" static " をつけた内部変数はデータ領域へとられ、初期化されるのはプログラム起動時だけです。

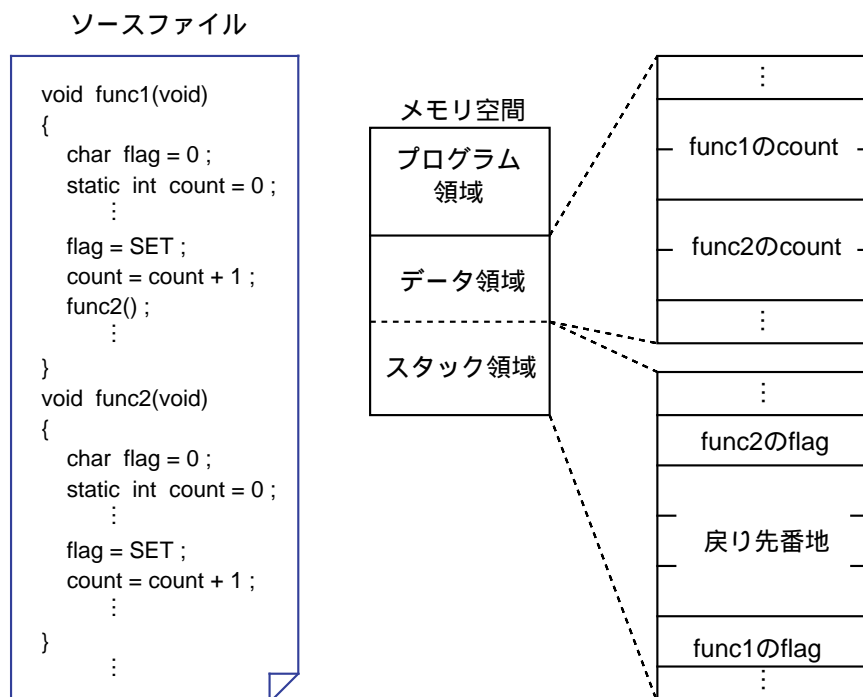


図 1.6.4 内部変数の記憶クラス

1.6.3 関数の記憶クラス

関数の記憶クラスは、関数の定義側と呼び出し側の両方で指定します。使用する記憶クラス指定子は、「static」と「extern」です。

この項では、関数に対する記憶クラスの指定方法を説明します。

グローバルな関数とローカルな関数

- (1) 関数の定義で記憶クラスを指定しないとき
この関数は他のソースファイルからも呼び出して使用できるグローバルな関数となります。
- (2) 関数の定義で「static」と宣言したとき
この関数は他のソースファイルからは呼び出せないローカルな関数になります。
- (3) 関数の型宣言で「extern」と宣言したとき
この関数は関数が記述されるソースファイルになく、他のソースファイルの関数を呼び出すことを明示しています。ただし、「extern」を付けなくても型宣言してあれば、関数がソースファイル内にないときは、自動的に「extern」をつけたのと同じ扱いになります。

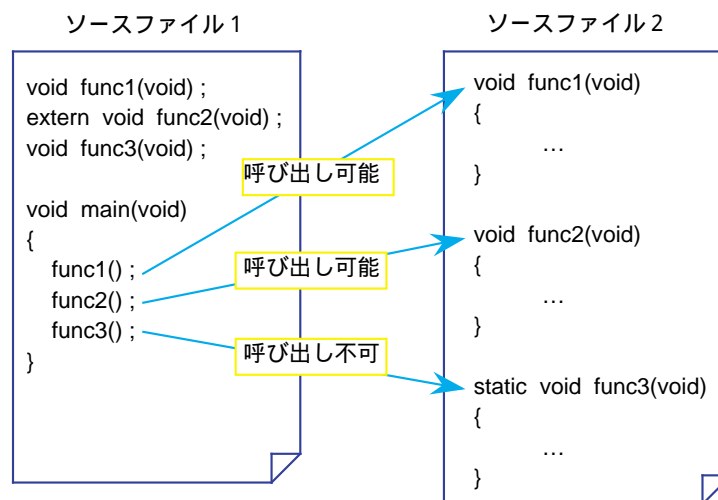


図 1.6.5 関数の記憶クラス

記憶クラスのまとめ

変数の記憶クラスを表 1.6.1 に、関数の記憶クラスを表 1.6.2 にまとめます。

表 1.6.1 変数の記憶クラス

記憶クラス	外部変数	内部変数
記憶クラス 指定子省略	他のソースファイルからも参照できる グローバルな変数 [データ領域上に割り付ける]	関数内でのみ有効な変数 [実行時にスタック上に割り付ける]
auto		関数内でのみ有効な変数 [実行時にスタック上に割り付ける]
static	他のソースファイルからは参照できな いローカルな変数 [データ領域上に割り付ける]	関数内でのみ有効な変数 [データ領域上に割り付ける]
register		関数内でのみ有効な変数 [実行時にレジスタ上に割り付ける] ただし、NC308では生成コード変更オ プション「-fenable_register(-fER)」を 指定していない場合は意味を持たない (コンパイル時に無視される)
extern	他のソースファイルの変数を参照する 変数 [割り付けない]	他のソースファイルの変数を参照する 変数 (他の関数からの参照はできない) [割り付けない]

表 1.6.2 関数の記憶クラス

記憶クラス	関数の種類
記憶クラス 指定子省略	他のソースファイルからも呼び出し実行できるグローバルな関数 [定義側で指定する]
static	他のソースファイルからは参照できないローカルな関数 [定義側で指定する]
extern	他のソースファイルにある関数を呼び出す [呼び出し側で指定する]

コラム レジスタ変数を使用するには

NC308では生成コード変更オプション「-fenable_register(-fER)」を指定することによりレジスタ記憶クラスを有効にします。このオプションを指定しない場合は、レジスタ記憶クラス指定子を記述しても意味を持たない(コンパイル時に無視される)ため注意してください。

```

void func1(void);

char array[10][10];

void func1(void)
{
    register int i,j;

    for(i = 0,j = 0 ; i < 0, j < 0 ; i++,j++){
        array[i][j] = 0;
    }
}

```

生成コード変更オプション「-fenable_register(-fER)」を指定すると実行時にレジスタ上に割り付けられる

図 1.6.6 レジスタ変数を使用するには

1.7 配列とポインタ

1.7.1 配列

「配列」と「ポインタ」はC言語の特徴ある機能です。

以下の項では、この配列の使用方法和、配列を扱う上で重要な手段となるポインタについて説明します。

配列とは何か

家族の年齢の合計を求めるプログラムを例とします。家族の構成は両親（父 = 29歳、母 = 24歳）と子ども1人（僕 = 4歳）です（例 1.7.1 参照）。

このプログラムでは、家族が増えると変数名が増えます。これに対処する手段として、C言語では「配列」という概念があります。配列とは、同じ型をもつデータ（int型）を1つの集合体として扱います。この例では父の年齢（papa）、母の年齢（mama）・・・と別の変数として扱うのではなく、家族の年齢（age）という集合体とします。各データは集合体の「要素」となります。つまり、0番目の要素が父、1番目の要素が母、2番目が僕です。

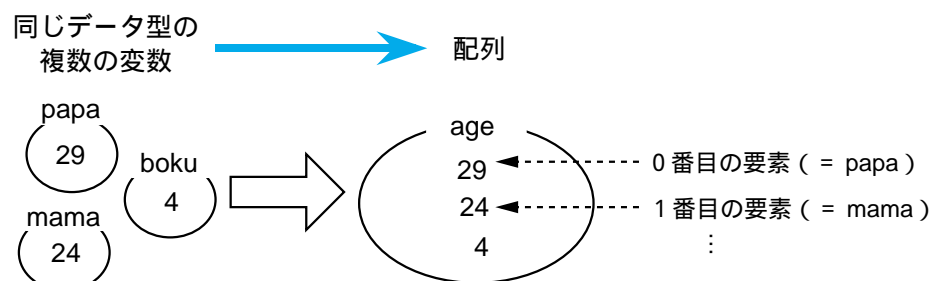


図 1.7.1 配列の概念

例 1.7.1 家族の年齢の合計を求める - 1 -

家族（父、母、僕）の年齢の合計を求めます。

```
void main(void)
{
    int papa = 29 ;
    int mama = 24 ;
    int boku = 4 ;
    int gokei ;

    gokei = papa + mama + boku ;
}
```

家族が増えると変数の型宣言、初期化の実行文が増え、演算式も長くなる。

```
void main(void)
{
    int papa = 29 ;
    int mama = 24 ;
    int boku = 4 ;
    int imouto1 = 1 ;
    int imouto2 = 1 ;
    :
    int gokei ;

    gokei = papa + mama + boku + imouto1 + imouto2 + ... ;
}
```

例 1.7.1 家族の年齢の合計を求め - 1 -

1.7.2 配列の作成

C 言語で扱う配列には、「1次元配列」と「2次元配列」の2種類があります。
この項では、各々の配列の作成と参照方法を説明します。

1 次元配列

1次元配列は、1次元の（直線的な）広がりをもつ配列です。1次元配列の宣言の書式を示します。

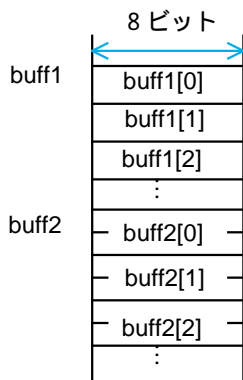
```
データ型 配列名 [要素数];
```

上の宣言を行うと、配列名を先頭ラベルとして要素数だけメモリ上に領域が確保されます。

1次元配列を参照するためには、配列名に要素番号を添字として付けます。ただし、要素番号は'0'から始まるので、最後の要素番号は「要素数 - 1」になります。

・ 1次元配列の宣言

```
char buff1[3];
int buff2[3];
```



・ 1次元配列の宣言と初期化

```
char buff1[] = {
    'a', 'b', 'c'
};

int buff2[] = {
    10, 20, 30
};
```

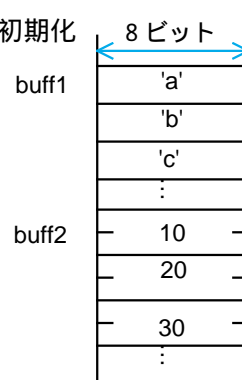


図 1.7.2 1次元配列の宣言とメモリ配置

例 1.7.2 家族の年齢の合計を求める - 2 -

配列を利用して家族の年齢の合計を求めます。

```
#define MAX 3 (注)
```

```
void main(void)
{
    int age[MAX];
    int gokei = 0;
    int i;

    age[0] = 29;
    age[1] = 24;
    age[2] = 4;

    for(i = 0; i < MAX; i++) {
        gokei += age[i];
    }
}
```

または

```
#define MAX 3
```

```
void main(void)
{
    int age[] = {
        29, 24, 4
    };

    int gokei = 0;
    int i;

    for(i = 0; i < MAX; i++) {
        gokei += age[i];
    }
}
```

宣言と同時に初期化している

配列にしておくと要素数を変数にして繰り返し文が利用できる

(注) #define MAX 3 : MAX = 3 と同義定義している
(「1.9 プリプロセスコマンド」参照)

例 1.7.2 家族の年齢の合計を求める - 2 -

2次元配列

2次元配列は「列」と「行」からなる平面的な広がりをもつ配列です。また、1次元配列の配列とみることできます。次に宣言の書式を示します。

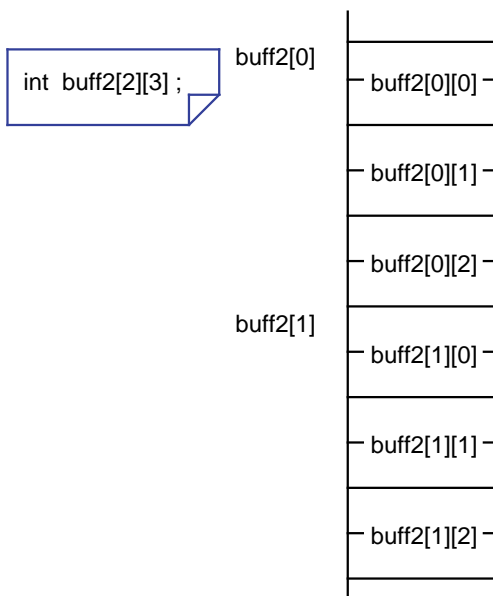
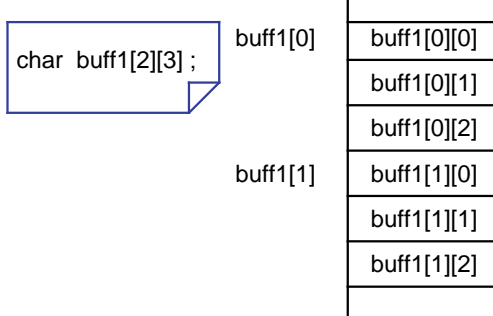
```
データ型 配列名 [行数][列数];
```

2次元配列を参照するときは配列名に「行番号」と「列番号」の2つを添字として付けます。行番号、列番号ともに'0'から始まるので、最後の番号は「行数(列数) - 1」になります。

・ 2次元配列の概念

行	0行0列	0行1列	0行2列	0行3列
	1行0列	1行1列	1行2列	1行3列
	2行0列	2行1列	2行2列	2行3列

・ 2次元配列の宣言



・ 2次元配列の宣言と初期化

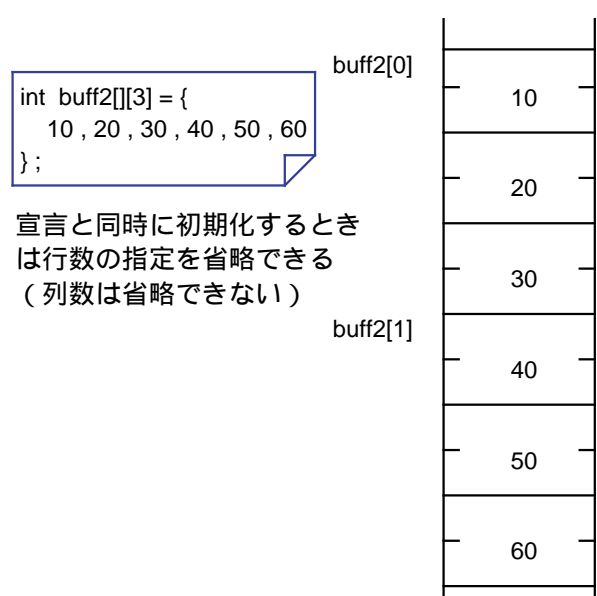
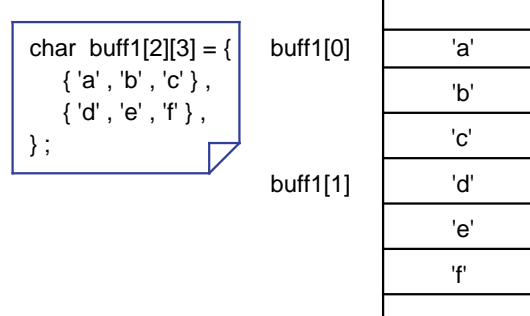


図 1.7.3 2次元配列の宣言とメモリ配置

1.7.3 ポインタ

ポインタ "pointer" とはデータを指し示すもの、つまりアドレスを意味します。

これから紹介する「ポインタ変数」は、データが格納されている「アドレス」を変数として扱います。アセンブリ言語でいうところの「間接アドレッシング」にあたるものです。

この項では、ポインタ変数の宣言と参照方法を説明します。

ポインタ変数の宣言

ポインタ変数は次のような書式で宣言します。

```
指し示すデータの型 *ポインタ変数名;
```

ただし、上の宣言で確保される領域はアドレスを格納する領域のみです。データ本体の領域を確保するには、別に型宣言する必要があります。

・ポインタ変数の宣言

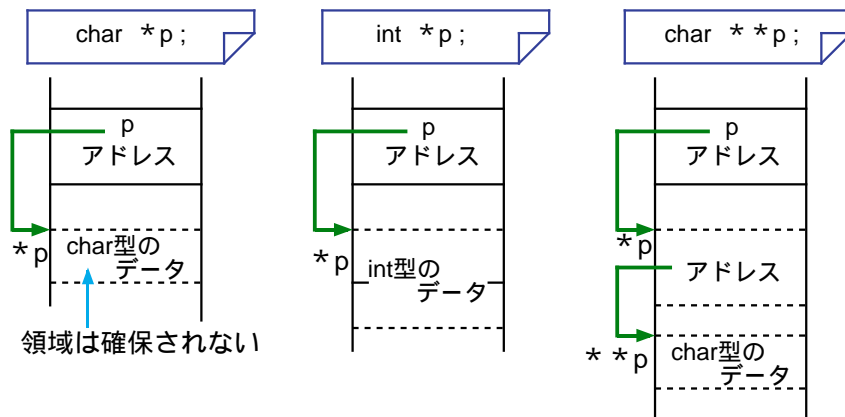


図 1.7.4 ポインタ変数の宣言とメモリ配置

ポインタと変数の関係

次にポインタ変数と変数の関係を、int 型の変数 'a' に対してポインタ変数 'p' を使って定数 '5' を代入する方法を例として説明します。

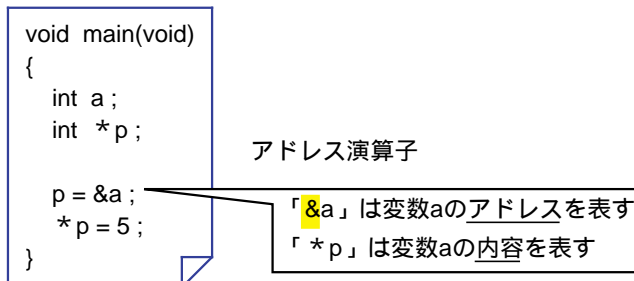


図 1.7.5 ポインタ変数と変数の関係

ポインタ変数の演算

ポインタ変数を加算、または減算することができます。ただし、このポインタ変数演算は整数の演算とは異なり、演算結果はアドレス値となります。したがって、ポインタ変数が指すデータのサイズによってアドレス値が異なります。

$$\begin{aligned} &\text{アドレス} + (\text{整数} \times \text{sizeof}(\text{型})) \\ &\text{アドレス} - (\text{整数} \times \text{sizeof}(\text{型})) \end{aligned}$$

ポインタ変数 ptr は int 型の変数を指す。
int 型変数のサイズを sizeof(int) で切り出すと
2 バイトになる。
したがって、p + 2 は 1004H 番地となる。

```

int * ptr;

ptr = (int *)0x001000;
ptr = p + 2;
    
```

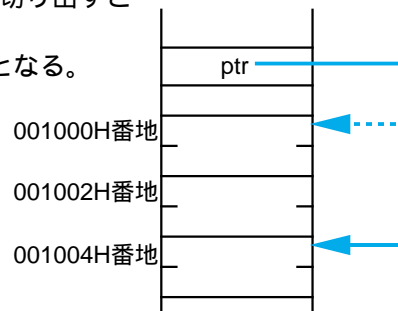


図 1.7.6 ポインタ変数の演算

コラム ポインタ変数のデータ長は？

C 言語プログラムの変数のデータ長は、データ型により決まります。ポインタ変数の場合、その内容はアドレスです。したがって、使用するマイクロプロセッサがアクセスできる全アドレス空間を表現できるだけのデータ長がポインタ変数に対して用意されることになります。

NC308 のポインタ変数は、該当するデータがデフォルトで far 領域にあるものとしているため、4 バイトのデータ長を持ちます。詳しくは「2.3.1 効率よいアドレッシング」を参照してください。

1.7.4 ポインタの活用

この項では、ポインタの活用例を説明します。

ポインタ変数と 1 次元配列

配列名に要素番号を示す添字を記述する方法では、「インデックスアドレッシング」としてコード化されます。このため、配列をアクセスするには常に「先頭から何番目」というアドレス計算が必要になります。一方、ポインタ変数を利用すると間接アドレッシングとなります。

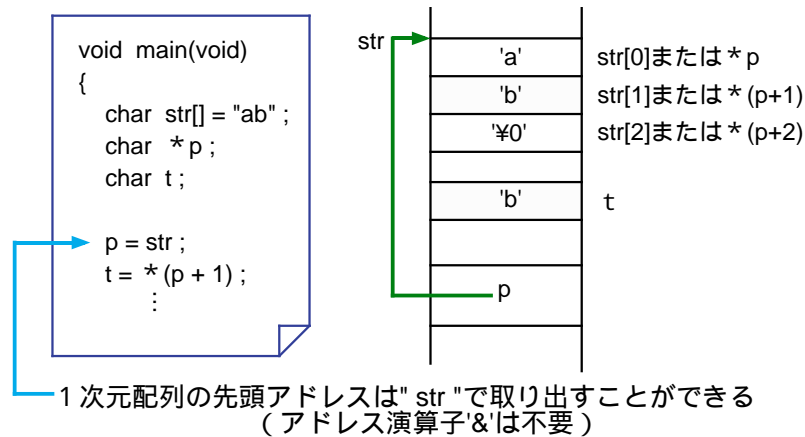


図 1.7.7 ポインタ変数と 1 次元配列

ポインタ変数と 2 次元配列

2次元配列も 1次元配列と同様、ポインタ変数を使ってアクセスできます。

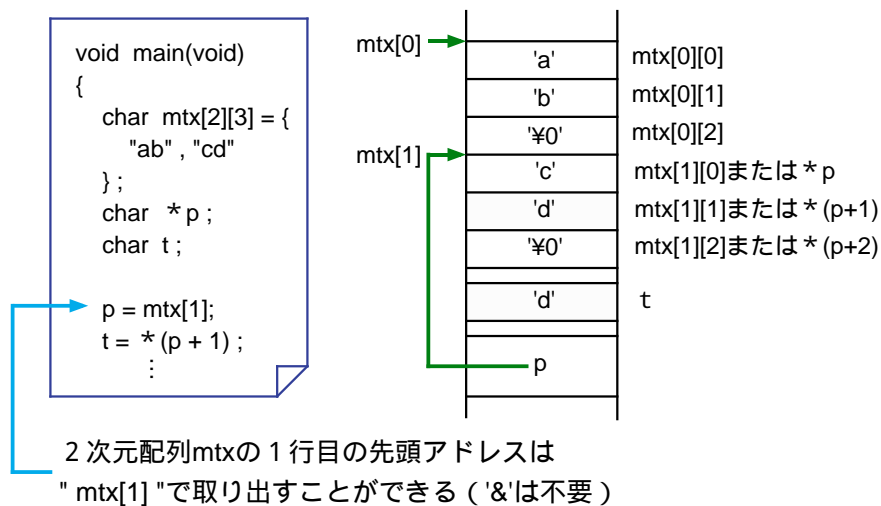


図 1.7.8 ポインタ変数と 2 次元配列

関数のアドレス渡し

C 言語の関数間の基本的なデータの受け渡しは「値渡し "Call by Value"」です。しかしこの方法では、配列や文字列を引数や戻り値にすることができません。

そこで使用される方法が、ポインタ変数を利用した「アドレス渡し "Call by Reference"」です。この方法は配列や文字列のアドレスを受け渡しする以外に、複数のデータを戻り値にしたいときにも利用できます。

ただし値渡しとは違い、アドレス渡しでは、呼び出し側のデータを直接書き換えることになるので、関数の独立性は低くなります。

図 1.7.8 にアドレス渡しにより配列を受け渡す例を示します。

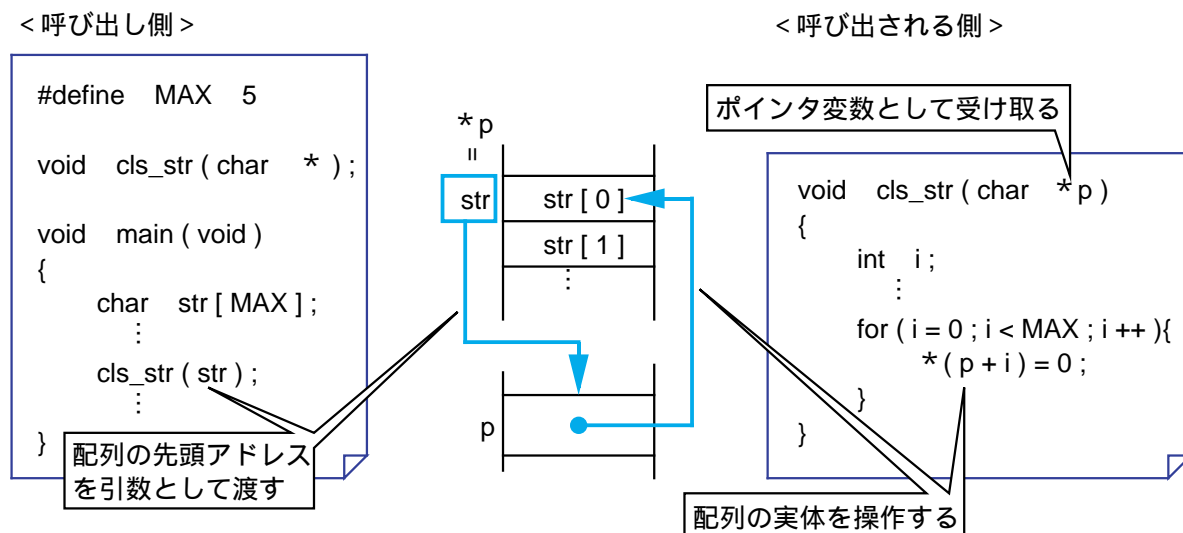


図 1.7.9 アドレス渡しの例 (配列を受け渡す)

コラム 関数間で高速にデータを受け渡す

関数間でデータを受け渡す方法として、値渡しとアドレス渡し以外に、受け渡すデータを外部変数にする方法があります。

この方法では関数の独立性が失われてしまうので、C 言語のプログラムとしては推奨できません。しかし関数呼び出し時の入口処理と出口処理 (引数と戻り値の受け渡し) がなくなるので、高速に関数を呼び出すことができるというメリットがあります。この性質を利用して、汎用性をあまり必要とせず、しかも高速処理を行いたい ROM 化プログラムではよく使用されます。

1.7.5 ポインタの配列化

この項では、ポインタ変数を配列にした「ポインタ配列」を説明します。

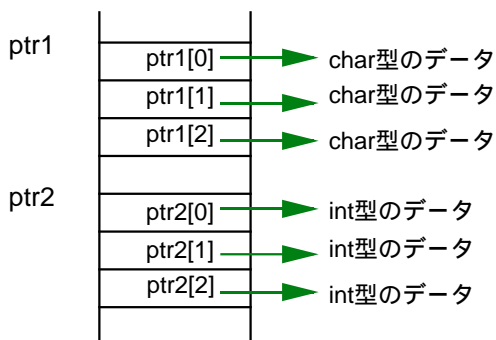
ポインタ配列の宣言

ポインタ配列の宣言方法を示します。

```
データ型 far(注) *配列名 [要素数];
```

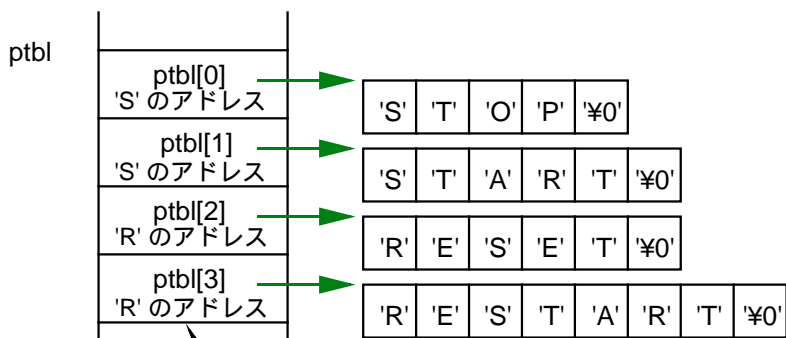
・ポインタ配列の宣言

```
char far *ptr1[3];
int far *ptr2[3];
```



・ポインタ配列の初期化

```
char far *ptbl[4] = {
    "STOP",
    "START",
    "RESET",
    "RESTART"
};
```



各文字列の先頭のアドレスが格納されている

図 1.7.10 ポインタ配列の宣言と初期化

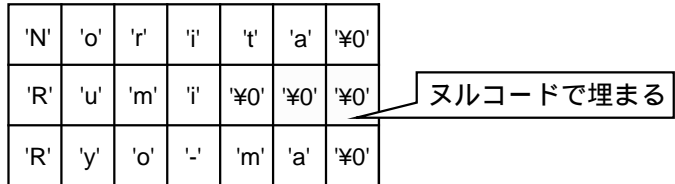
(注) NC308 では、ポインタ配列の実体データは far 領域に配置されます。また、ポインタ型変数はデフォルトで far 型(4 バイト)の変数となります。したがって通常ポインタに記述する "far" を省略します。詳しくは「2.3.1 効率よいアドレッシング」を参照してください。

ポインタ配列と 2 次元配列

ポインタ配列と 2 次元配列の違いを説明します。2 次元配列で文字数の異なる複数の文字列を宣言したとき、空いた領域はヌルコード'¥0'で埋まります。同じことをポインタ配列で定義するとメモリの空きが発生しません。このため、大量の文字列を操作するときや少しでもメモリ使用量を抑えたいときに有効な手法です。

・ 2 次元配列

```
char name[][7] = {
    "Norita",
    "Rumi",
    "Ryo-ma"
};
```



・ ポインタ配列

```
char far *name[3] = {
    "Norita",
    "Rumi",
    "Ryo-ma"
};
```

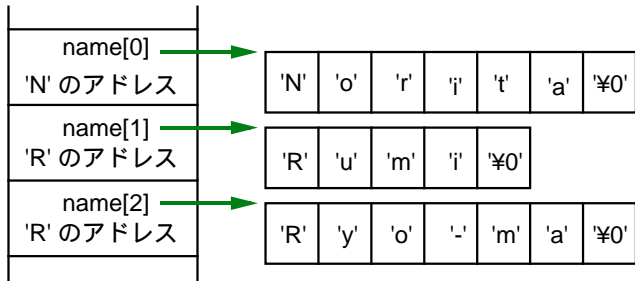


図 1.7.11 2 次元配列とポインタ配列の違い

1.7.6 関数ポインタを使ったテーブルジャンプ

アセンブリ言語のプログラムでは、あるデータの内容によって切り換えるべき処理が増えると「テーブルジャンプ」を使用します。前述のポインタ配列を利用すると、同様のことがC言語でも記述できます。

この項では、「関数ポインタ」を使ったテーブルジャンプの記述方法を説明します。

関数ポインタとは

前述のポインタと同様、関数の先頭アドレスを指し示すものが「関数ポインタ」です。関数ポインタを使用すると呼び出す関数をパラメータにできます。宣言と参照の書式を次に示します。

<宣言の書式> 戻り値の型 (*関数ポインタ名)(引数のデータ型);
<参照の書式> 戻り値を格納する変数 = (*関数ポインタ名)(引数);

例 1.7.3 テーブルジャンプを使った四則演算の切り換え

変数 "num" の内容によって演算方法を切り換えます。

```

/* プロトタイプ宣言 **** */
int  calc_f ( int , int , int );
int  add_f (int , int ) , sub_f ( int , int );
int  mul_f ( int , int ) , div_f ( int , int );

/* ジャンプテーブル **** */
int  (* const  jmptbl[] ) ( int , int ) = {
    add_f , sub_f , mul_f , div_f
};

void  main ( void )
{
    int  x = 10 , y = 2 ;
    int  num , val ;

    num = 2 ;
    if ( num < 4 ) {
        val = calc_f ( num , x , y );
    }
}

int  calc_f ( int  m , int  x , int  y )
{
    int  z ;
    int  (* p ) ( int , int );

    p = jmptbl [ m ] ;
    z = (* p ) ( x , y );
    return  z ;
}

```

関数ポインタの配列化

jmptbl[0]	"add_f"の 先頭アドレス
jmptbl[1]	"sub_f"の 先頭アドレス
jmptbl[2]	"mul_f"の 先頭アドレス
jmptbl[3]	"div_f"の 先頭アドレス

飛び先の設定

関数ポインタを使ったコール

例 1.7.3 テーブルジャンプを使った四則演算の切り換え

1.8 構造体と共用体

1.8.1 構造体と共用体

これまでのデータ型 (*char*型、*signed int*型、*unsigned long int*型など) は、コンパイラ仕様で決められた「基本データ型」といわれるものです。

C言語ではこれらの基本データ型をもとにして新しいデータ型を作ることができます。それが「構造体」と「共用体」です。

以下の項では、構造体と共用体の宣言方法と参照方法を説明します。

基本データ型から構造体へ

構造体や共用体を使用すれば、基本データ型をもとにして目的に応じた、より複雑なデータ型を作成できます。しかも、新たに作成されたデータ型は基本データ型と同じように参照したり、配列にすることができます。

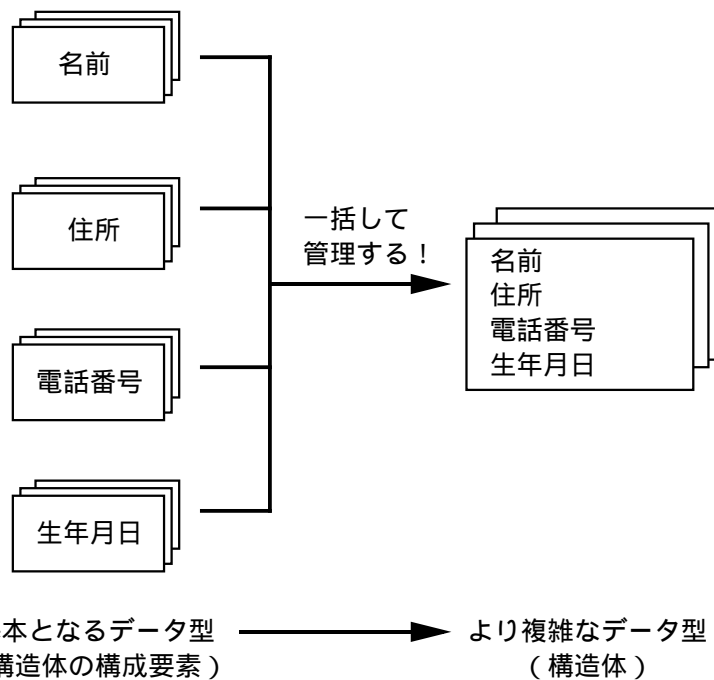


図 1.8.1 基本データ型から構造体へ

1.8.2 新しいデータ型の作成

新しいデータ型の基本となる構成要素のことを「メンバ」といいます。新しいデータ型を作るには、構成するメンバを定義します。この定義によって、これまでの変数と同じように、宣言して領域を確保し、必要に応じて参照できます。

この項では、構造体と共用体、各々の定義と参照方法を説明します。

構造体と共用体の違い

構造体と共用体では領域を確保する際、メンバの配置方法が異なります。

- (1) 構造体：メンバをシーケンシャルに配置します。
- (2) 共用体：メンバを同一アドレスから配置します。
(複数のメンバが同一メモリ領域を共有します)

構造体の定義と宣言

構造体を定義するためには「struct」を記述します。

```
struct 構造体タグ {
    メンバ1 ;
    メンバ2 ;
    :
};
```

上のように記述すると「struct 構造体タグ」というデータ型ができます。このデータ型を使って宣言すれば、通常の変数と同様、メモリ領域が確保されます。

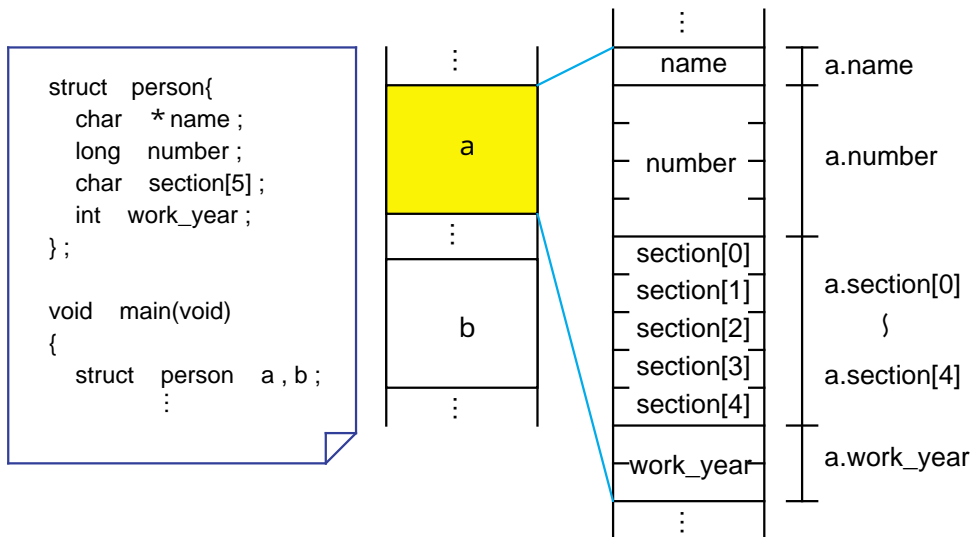
```
struct 構造体タグ 構造体変数名 ;
```

構造体の参照

構造体の各メンバを参照するときは、「構造体メンバ演算子'.'」を用います。

構造体変数名.メンバ名

構造体変数を初期化するときは、各メンバの初期化データを宣言順に、型を合わせて書き並べます。



nameの格納されている領域がnear領域の場合"struct person"は13バイトの型となり、far領域の場合15バイトの型となる

・ 構造体変数の初期化

```

struct person a = {
    "SATOH", 10025, "T511", 25
};
    
```

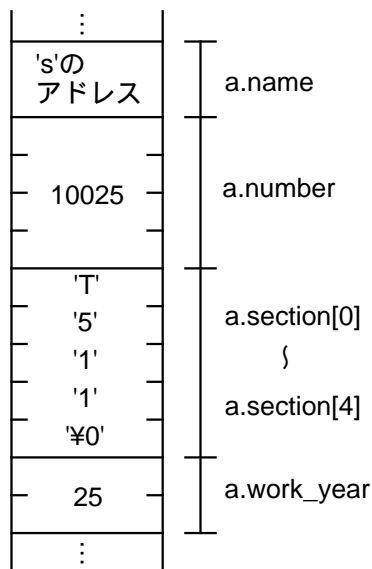


図 1.8.2 構造体の宣言とメモリ配置

ポインタを使った参照例

ポインタを使って各メンバを参照するときは、「->」を使用します。

ポインタ -> メンバ名

```

struct person{
  char far *name;
  long number;
  char section[5];
  int work_year;
};

struct person a={
  "SATOH", 10025, "T511", 25
};

void main(void)
{
  struct person *p;
  p = &a;
  :
}
    
```

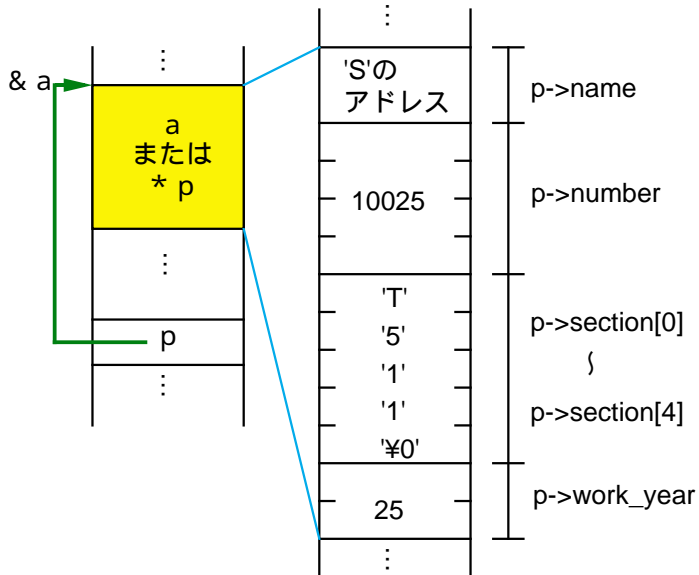


図 1.8.3 ポインタを使った参照例

共用体

共用体はメモリ上のある領域を全メンバで共有します。このため、絶対に同時に存在しない複数のデータを共用体にするメモリ使用量を節約できます。また、状況によって16ビット単位、8ビット単位など、扱う単位を切り換えたいデータに対して便利な機能です。

共用体を定義するためには「union」を記述します。この記述以外の定義、宣言、参照については構造体の場合と同じです。

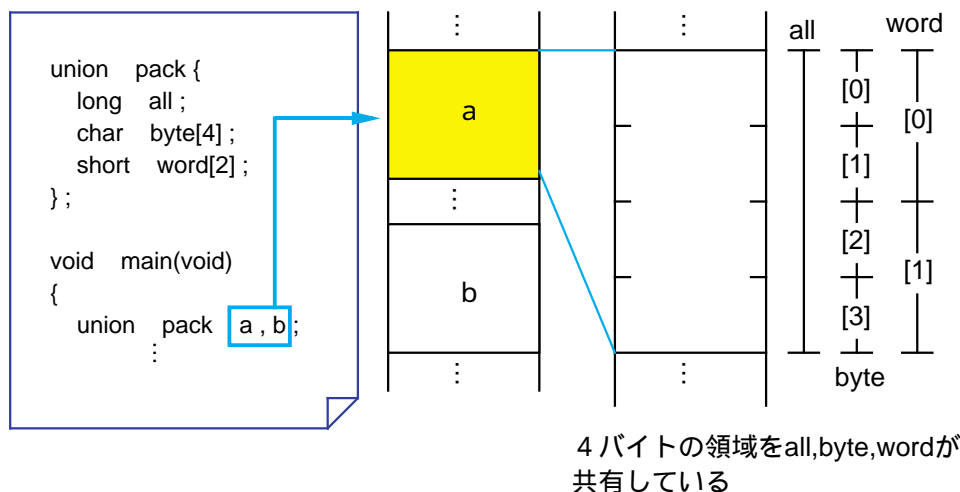


図 1.8.4 共用体の宣言と参照

コラム 型定義

構造体や共用体では「struct」や「union」のキーワードが必要なため、定義されたデータ型の文字数が増えてしまいます。これを回避する方法に「型定義 typedef」があります。

```
typedef 既存型名 新規型名;
```

のように記述すると、新規型名が既存型名と同義語とみなされ、プログラム中ではどちらの型名も使用できます。では実際に「typedef」を使用した例を図 1.8.5 に示します。

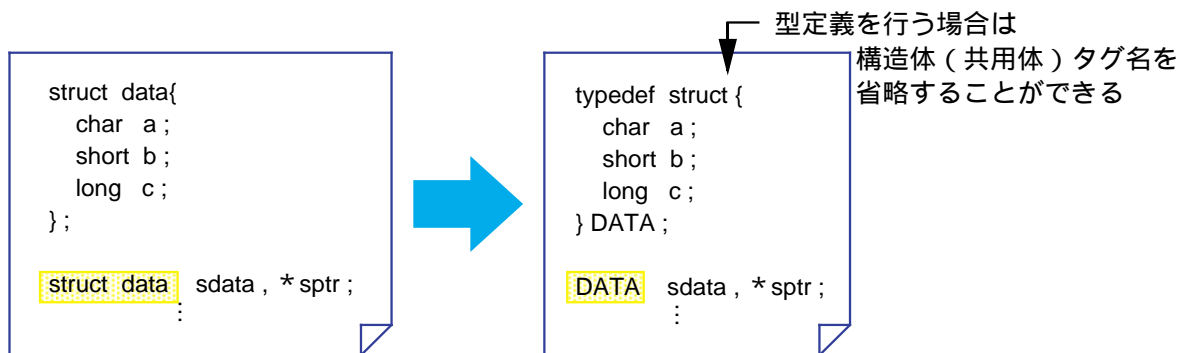


図 1.8.5 型定義 "typedef" の使用例

1.9 プリプロセスコマンド

1.9.1 NC308 のプリプロセスコマンド

C 言語では、ファイルの取り込み、マクロ機能、条件コンパイルのような機能を「プリプロセスコマンド」としてサポートしています。

以下の項では、NC308 で用意している主なプリプロセスコマンドを説明します。

NC308 のプリプロセスコマンド一覧

プリプロセスコマンドは他の実行文と区別するため、先頭が「'#」で始まる文字綴りになっています。記述位置は任意ですが、区切りを表すセミコロン';'はつけません。NC308 で使用できる主なプリプロセスコマンドを表 1.9.1 に示します。

表 1.9.1 NC308 の主なプリプロセスコマンド

記述	機能
#include	指定したファイルを取り込みます。
#define	文字列の置換およびマクロ定義を行います。
#undef	#defineによる定義を取り消します。
#if ~ #elif ~ #else ~ #endif	条件コンパイルを行います。
#ifdef ~ #elif ~ #else ~ #endif	条件コンパイルを行います。
#ifndef ~ #elif ~ #else ~ #endif	条件コンパイルを行います。
#error	メッセージを標準出力に出力し処理を中断します。
#line	ファイルの行番号を指定します。
#assert	定数式が偽の時に警告を出力します。
#pragma	NC308の拡張機能の処理を指示します。詳細は第2章以降で説明します。

1.9.2 ファイルの取り込み

別のファイルを取り込むためには「`#include`」を使用します。NC308では、検索するディレクトリによって記述方法が異なります。

この項では、目的別に「`#include`」の記述方法を説明します。

標準ディレクトリを検索する

```
#include <ファイル名>
```

起動オプション「`-I`」で指定したディレクトリ内のファイルを取り込みます。このディレクトリにファイルが存在しない場合は、NC308の環境変数「`INC308`」で設定した標準ディレクトリを検索し、ファイルを取り込みます。

通常、標準ディレクトリとして「標準インクルードファイル」が入っているディレクトリを指定します。

カレントディレクトリを検索する

```
#include "ファイル名"
```

カレントディレクトリのファイルを取り込みます。カレントディレクトリにファイルが存在しなければ、起動オプション「`-I`」で指定したディレクトリ、NC308の環境変数「`INC308`」で設定したディレクトリの順で検索し、ファイルを取り込みます。

標準インクルードファイルと区別するために独自に作成したインクルードファイルをカレントディレクトリに入れて、この記述方法で指定します。

"#include" 使用例

NC308の「`#include`」では8レベルまでネスティングが可能です。検索対象のどのディレクトリにも該当するファイルが存在しない場合はインクルードエラーを出力します。

```

/* インクルード * * * * * */
#include <stdio.h>
#include "usr_global.h"
/* メイン関数 * * * * * */
void main ( void )
{
    :
}
    
```

標準ディレクトリから
標準インクルードファイルを読み込む

カレントディレクトリから
グローバル変数のヘッダを読み込む

図 1.9.1 " #include " 記述例

1.9.3 マクロ定義

文字列の置換やマクロ定義には「#define 識別子」を使用します。識別子は通常、変数や関数と区別するために大文字を使用します。

この項では、マクロ定義と取り消し方法を説明します。

定数の定義

アセンブラの equ 文と同じように定数に名前を付けることができます。プログラム中のマジックナンバー（意味不明の即値）をなくすためや、定義を共通化するのに有効な方法です。

```
#define THRESHOLD 100
#define UPPER_LIMIT (THRESHOLD+50)
#define LOWER_LIMIT (THRESHOLD - 50)
```

しきい値を100に定義する
上限を+50におく
下限を - 50におく

図 1.9.2 定数の定義例

文字列の定義

```
#define TITLE "Position control program"
char mess[] = TITLE ;

#define void
void func()
{
    :
}
```

"TITLE"の位置に定義した文字列が入る
"void"が削除される
"void"をサポートしていないコンパイラではこの定義をしておけばソースファイルでの変更の必要なし

文字列に名前を付けたり、逆に文字列を削除する場合に使用します。

図 1.9.3 文字列の定義例

マクロ関数の定義

"#define" を使用するとマクロ関数を定義することもできます。このマクロ関数では通常の関数と同様、引数・戻り値の受け渡しができます。しかも通常の関数のように入力処理と出力処理がないために実行速度が速くなります。

また、マクロ関数の場合、引数のデータ型を宣言する必要はありません。

```
#define ABS(a) ((a) > 0 ? (a) : - (a))
```

引数の絶対値を返すマクロ関数

```
#define SEQN(a, b, c) {¥  
    func1(a); ¥  
    func2(b); ¥  
    func3(c); ¥  
}
```

'¥'は連続記述を意味する
この後改行して記述しても連続した文字列
として判断される

複文の場合は '{、}' で囲む

図 1.9.4 マクロ関数の定義例

定義の取り消し

```
#undef 識別子
```

"#define" で定義した識別子の置換を "#undef" で降りいませぬ。

ただし、以下の4つの識別子はコンパイラの予約語のため、"#undef" で無効にしないでください。

- __FILE__ ; ソースファイルの名前
- __LINE__ ; 現在のソースファイルの行番号
- __DATE__ ; コンパイルの日付
- __TIME__ ; コンパイルの時間

1.9.4 条件コンパイル

NC308 では 3 種類の条件でコンパイルを制御できます。
仕様による関数の切り換え、デバッグ用関数の組み込みの有無を制御するときなどに使用します。
この項では、条件コンパイルの種類と記述方法を説明します。

条件コンパイルいろいろ

NC308 で使用できる条件コンパイルの種類を表 1.9.2 に示します。

表 1.9.2 条件コンパイル一覧

記述方法	内容
<pre>#if 条件式 A #else B #endif</pre>	条件式が真 (0 でない) の場合は A のブロックをコンパイルし、真でない場合は B のブロックをコンパイルする
<pre>#ifdef 識別子 A #else B #endif</pre>	識別子が定義されている場合は A のブロックをコンパイルし、定義されていない場合は、 B のブロックをコンパイルする
<pre>#ifndef 識別子 A #else B #endif</pre>	識別子が定義されていない場合は A のブロックをコンパイルし、定義されている場合は B のブロックをコンパイルする

3 種類とも " #else " のブロックは省略可能です。また 3 つ以上のブロックに分類したい場合は " #elif " で条件を追加してください。

識別子の定義指定

識別子の定義指定は " #define " または NC308 の起動オプション ' - D ' によって指定します。

#define 識別子

" #define " による定義指定

%nc308 - D 識別子

起動オプションによる定義指定

条件コンパイル記述例

条件コンパイルを利用して、デバッグ用関数の組み込みを制御した例を図 1.9.5 に示します。

```

#define  DEBUG

void  main ( void )
{
    :
#ifdef  DEBUG
    check_output();
#else
    output();
#endif
    :
}

#ifdef  DEBUG
void  check_output ( void )
{
    :
}
#endif
    
```

識別子"DEBUG"を定義 (デバッグモードに設定)

デバッグモードであれば「デバッグ関数」を
でなければ「通常出力関数」を呼び出す
この場合は「デバッグ関数」が呼び出される

デバッグモードであれば「デバッグ関数」を組み込む

図 1.9.5 条件コンパイル記述例

第 2 章

ROM 化技術

- 2.1 メモリ配置
- 2.2 スタートアッププログラム
- 2.3 ROM 化のための拡張機能
- 2.4 アセンブリ言語とのリンク
- 2.5 割り込み処理

この章では組み込みプログラムを作成するための注意点を NC308 の拡張機能を中心に紹介しています。

2.1 メモリ配置

2.1.1 コード / データの種類

プログラムを構成するデータ / コードは、書き換えできるものやできないもの、初期値を持つものと持たないものなど様々です。すべてのデータ / コードを性質に応じて、ROM 領域、RAM 領域、スタック領域へと配置しなければなりません。

この項では、NC308 で生成されるデータ / コードの種類を説明します。

NC308 が生成するデータ / コード

NC308 が生成するデータ / コードの種類と配置領域を図 2.1.1 に示します。

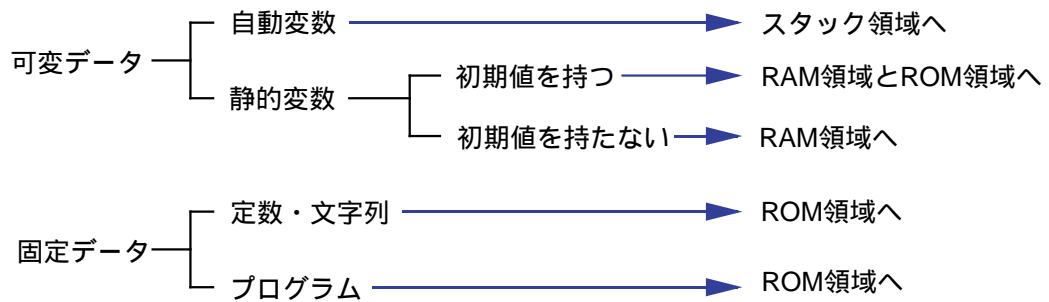


図 2.1.1 NC308 が生成するデータ / コードと配置領域

初期値を持つ静的変数の扱い

「初期値を持つ静的変数」は、書き換え可能なデータですから RAM 上になければなりません。しかし、RAM 上にあると初期値を設定することができません。

NC308 ではこの初期値を持つ静的変数に対しては RAM 上に領域を確保し、ROM 上に初期値を格納します。そしてスタートアッププログラムの中で ROM 上の初期値を RAM 上の領域にコピーします。

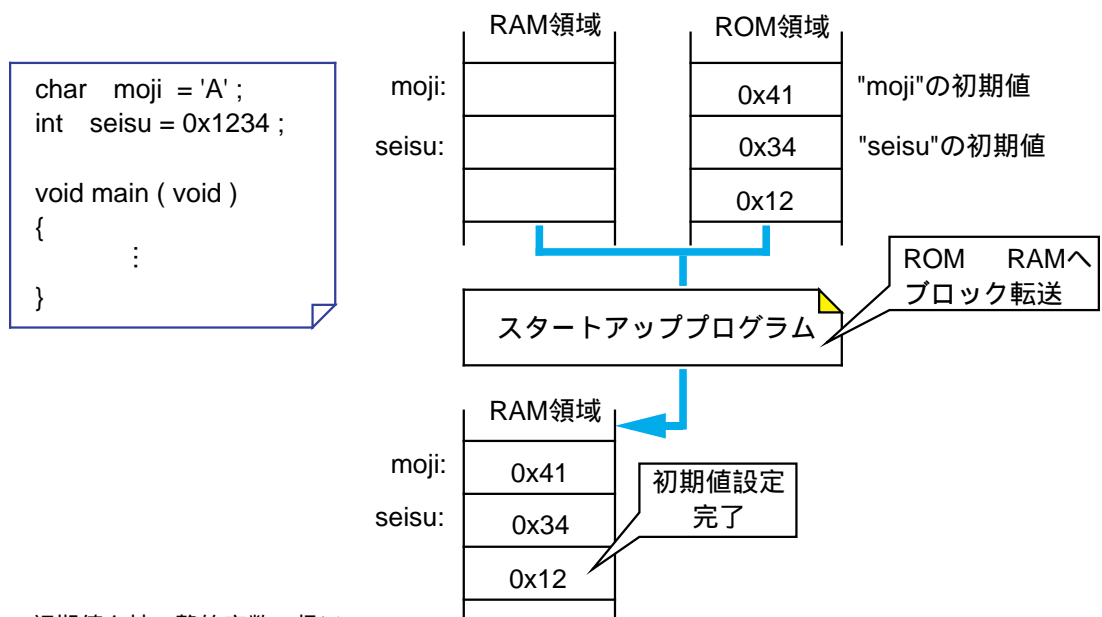


図 2.1.2 初期値を持つ静的変数の扱い

2.1.2 NC308 が管理するセクション

NC308 はデータ/コードの配置領域を「セクション」として管理しています。
この項では、NC308 が生成および管理するセクションの種類と管理方法を説明します。

セクションの構成

NC308 ではデータを種類別にセクションに分けて管理します (図 2.1.3)。NC308 が管理するセクションの構成を表 2.1.1 に示します。

表 2.1.1 NC308 のセクション構成

セクションベース名	内容
data	初期値を持つ静的変数を格納
bss	初期値を持たない静的変数を格納
rom	文字列・定数を格納
program	プログラムを格納
program_s	#pragma SPECIALで指定したプログラムを格納
vector	可変ベクタ領域 (コンパイラは生成しない)
fvector	固定ベクタ領域 (コンパイラは生成しない)
stack	スタック領域 (コンパイラは生成しない)
heap	ヒープ領域 (コンパイラは生成しない)

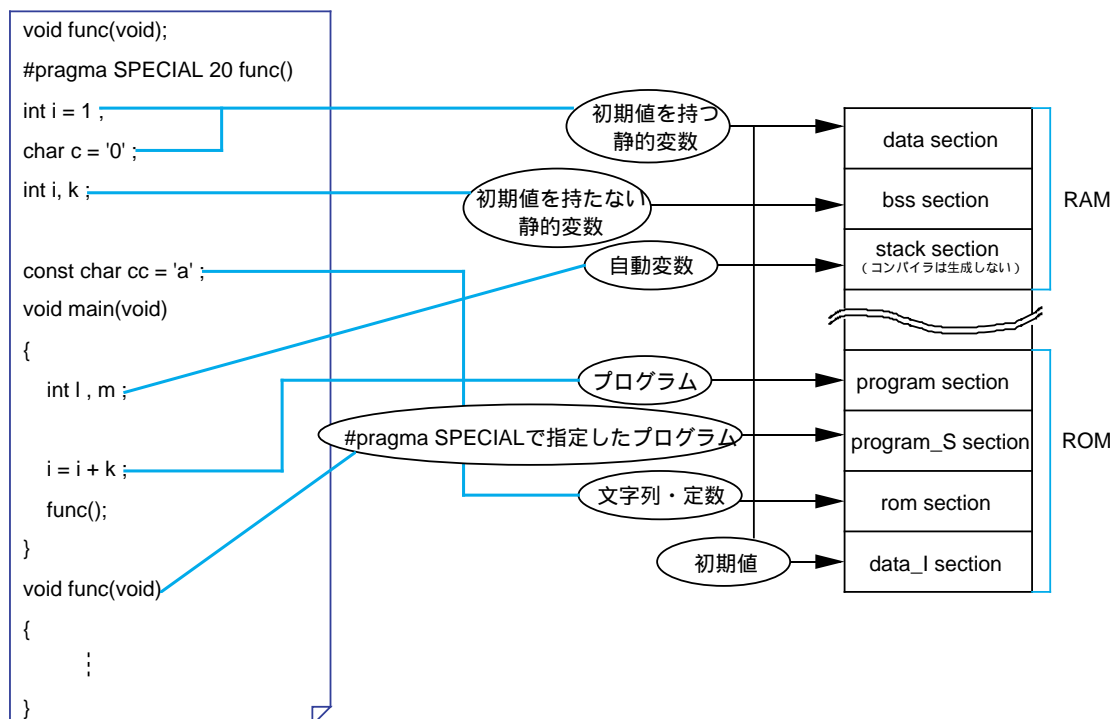


図 2.1.3 データの種類によるセクションへの配置

セクションの属性

NC308 が生成するセクションは、初期値の有無、配置される領域、データサイズなどの「属性」によって、さらに細かく分類されます。

属性の指定方法については「2.3.1 効率よいアドレッシング」を参照してください。

表 2.1.2 に各属性を表す記号と内容を示します。

表 2.1.2 セクションの属性

属性	内容	対象セクションベース名
I	データの初期値を保持するセクション	data
N / F / S	N - near属性 (絶対番地000000H ~ 00FFFFH番地の64Kバイトの領域)	data、 bss、 rom
	F - far属性 (000000H ~ FFFFFFFH番地の16Mバイト全メモリ領域)	data、 bss、 rom
	S - SBDATA属性 (SB相対アドレッシングを使用できる領域)	data、 bss
E / O	E - データサイズが偶数	data、 bss、 rom
	O - データサイズが奇数	data、 bss、 rom

セクションの命名規則

NC308 が生成するセクションの名前はセクションベース名と属性によって決められます。

各セクションベース名と属性の組み合わせを図 2.1.4 に示します。

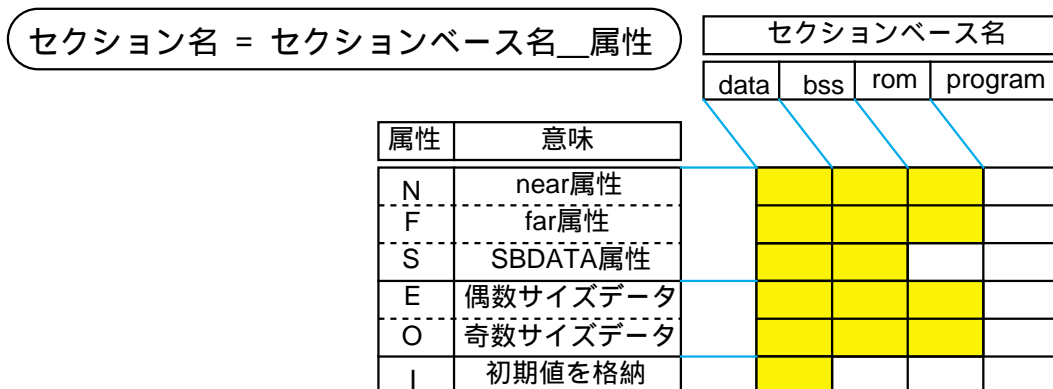


図 2.1.4 セクション名の命名規則

2.1.3 メモリ配置の制御

NC308 ではユーザのシステムに合わせて効率よくメモリ配置ができる拡張機能を用意しています。
この項では、メモリ配置のための拡張機能を説明します。

セクション名を変更する (#pragma SECTION)

#pragma SECTION 規定セクションベース名 変更セクションベース名

NC308 が生成するセクションベース名を変更します。" program " を変更した場合とそれ以外のセクションベース名を変更した場合は、変更名の有効範囲が異なります。

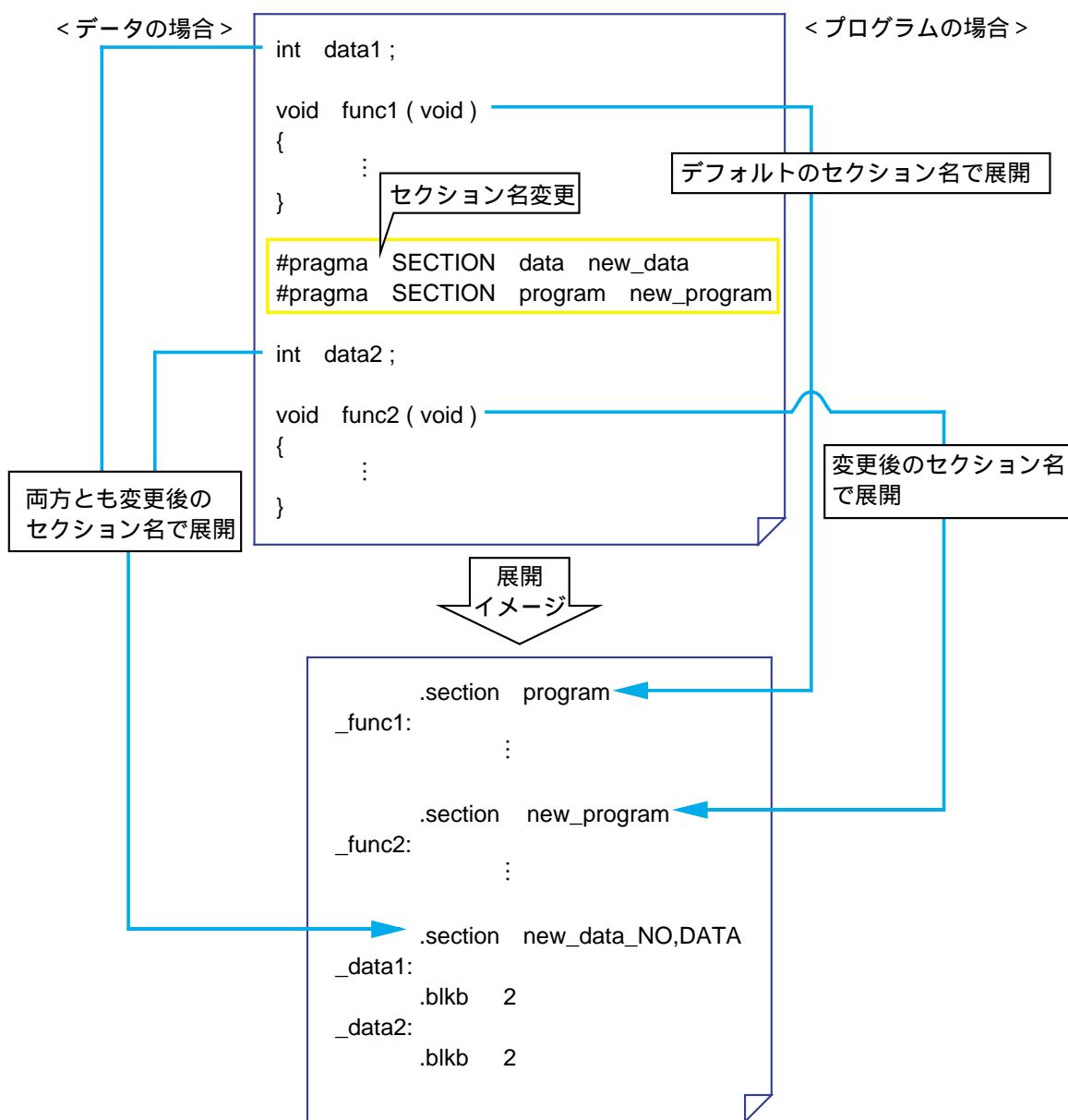


図 2.1.5 "#pragma SECTION" の記述例

セクション名の追加 ("sect308.inc")

NC308が生成するセクションはセクション定義ファイル"sect308.inc"に定義されています。"#pragma SECTION"でセクション名を変更するということは、NC308が生成するセクションベース名が追加されたということになります。そのため、必ずセクション定義ファイル "sect308.inc" でセクション名を追加定義しなければなりません。

```

; -----
; Arrangement of section
; -----
; Near RAM data area
; -----
; SBDATA area
    .section      data_SE,DATA
    .org          400H
data_SE_top:
;
    .section      bss_SE,DATA
bss_E_top:
    :
    .section      data_NO,DATA
data_NO_top:
;
    .section      new_data_NO,DATA
new_data_NO_top:
;
    :
; -----
; code area
; -----
    .section      interrupt
;
    .section      program
;
    .section      new_program
    :

```

"#pragma SECTION"で変更したセクション名を追加定義する

"#pragma SECTION"で変更したセクション名を追加定義する

図 2.1.6 セクション名の追加("sect308.inc")

強制的に ROM セクションへ配置する (const 修飾子)

変数の型宣言時に初期データを記入すると、RAM 領域と ROM 領域の両方が確保されます。しかし、このデータがプログラム実行中変化しない固定データであるとき、型宣言時に「const 修飾子」を記述します。すると ROM 領域のみが確保され、RAM 領域は使用しませんのでメモリ使用量を節約できます。また、コンパイル時に明示的な代入をチェックするため、書き換えミスをチェックできます。

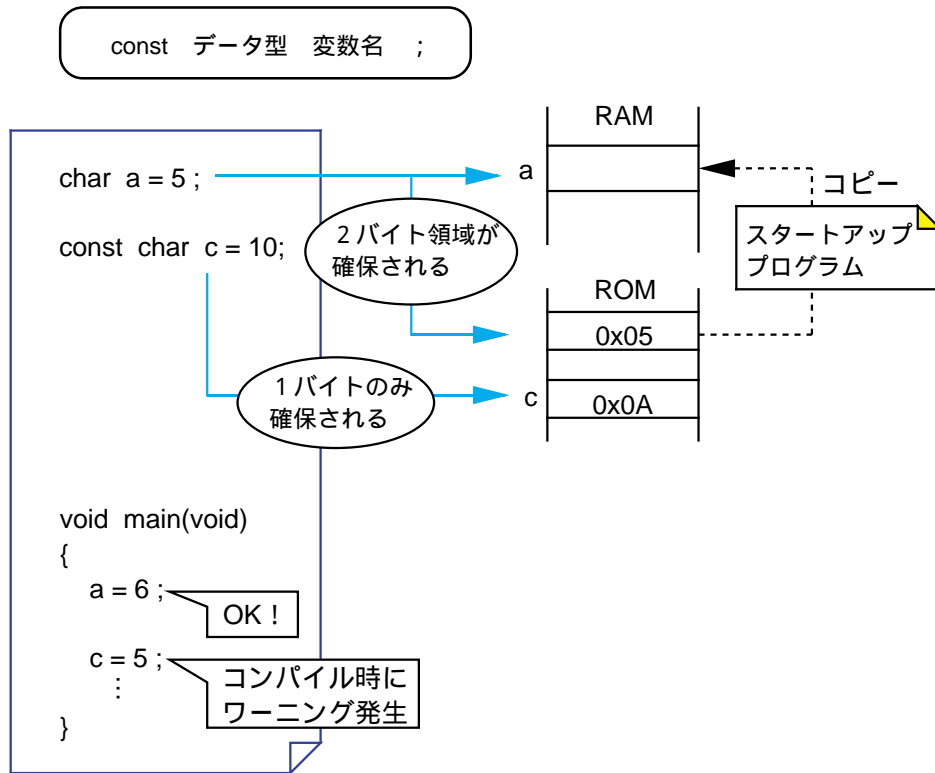


図 2.1.7 const 修飾子とメモリ配置

外部変数の参照を定数に置き換える最適化

最適化オプション'-Oconst'をつけると、コンパイル時に「const 修飾子」で宣言した外部変数の参照を定数に置き換える最適化を行います。最適化を行う外部変数は、構造体、共用体、および配列を除く外部変数です。また、同一の C 言語ソースファイル中で初期化を記述している外部変数に限られます。

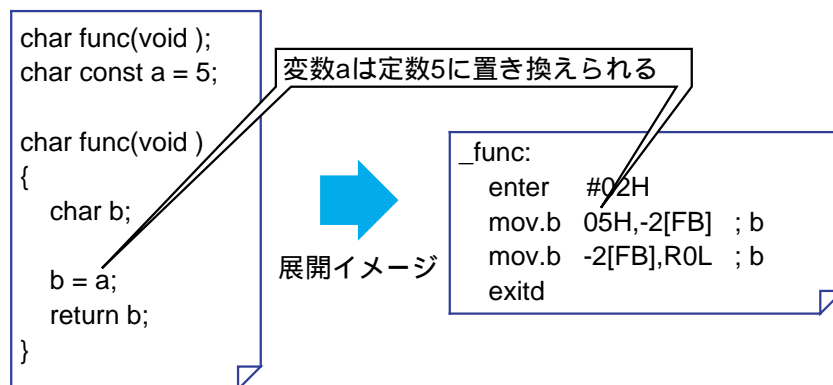


図 2.1.8 外部変数の参照を定数に置き換える最適化

2.1.4 構造体のメモリ配置の制御

NC308では構造体を配置する場合、メモリ使用量が最小になるように宣言順につめて(パックして)配置します。しかし、処理スピードを重視するときなどは「`#pragma STRUCT`」を記述し、構造体の配置方法を制御できます。

この項では、構造体の配置に関する拡張機能を説明します。

NC308 の構造体配置規則

NC308では以下の規則に基づいて構造体のメンバを配置します。

- (1) 構造体はパックします。構造体の内部にパディング(隙間)は発生しません。
- (2) メンバの宣言順に配置します。

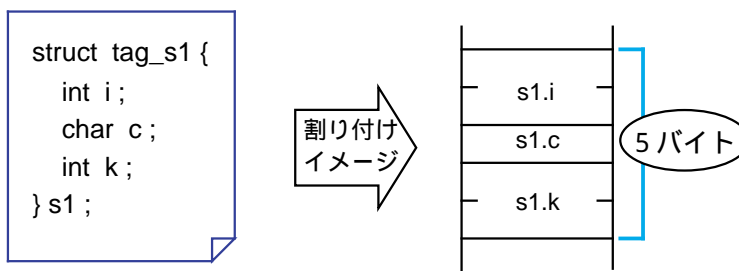
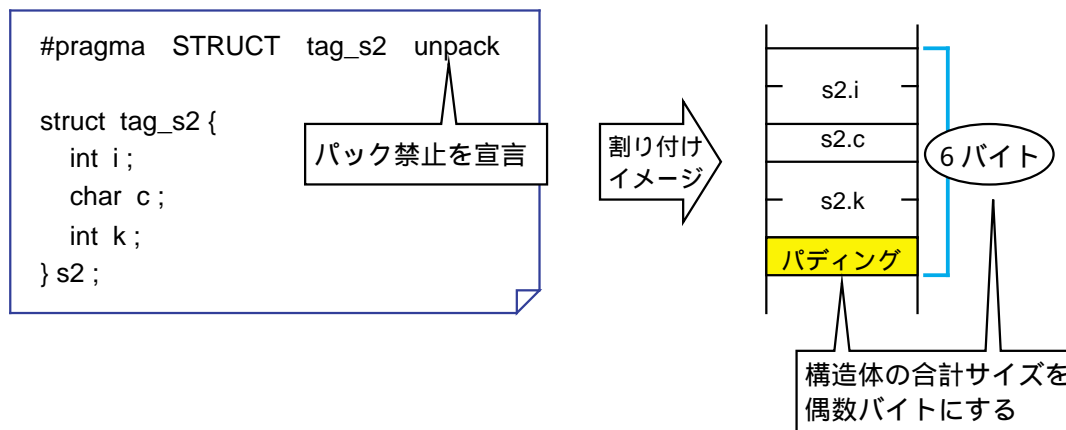


図 2.1.9 NC308 デフォルトの構造体の割り付けイメージ

構造体メンバのパックの禁止 (`#pragma STRUCT tag名 unpack`)



構造体メンバの合計サイズが偶数バイトになるようにパディング(隙間)を入れます。アクセススピードを優先するときに指定します。

図 2.1.10 構造体メンバのパックの禁止

構造体メンバの配置の最適化 (#pragma STRUCT tag 名 arrange)

構造体メンバの宣言順にとらわれず、偶数サイズのメンバを先に配置します。前述のパック禁止 " #pragma STRUCT unpack " と合わせて使用すると、偶数サイズの各メンバが偶数番地から配置されることになるので、効率の良いメモリアクセスを実現できます。

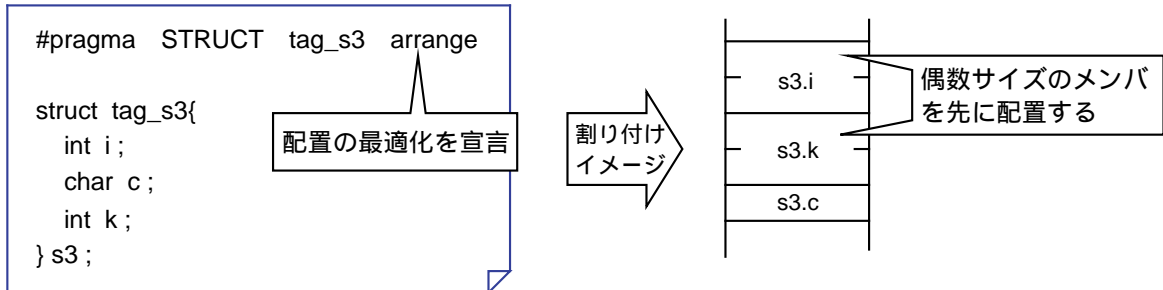


図 2.1.11 構造体メンバの配置の最適化

2.2 スタートアッププログラム

2.2.1 スタートアッププログラムの役割

組み込み型のプログラムを正常に動作させるためには、処理の前にマイコンの初期化やスタック領域の設定を行わなければなりません。通常、これらの処理はC言語では記述できません。そこでC言語のソースプログラムとは別に、アセンブリ言語で初期設定用のプログラムを記述します。これが「スタートアッププログラム」です。

以下の項では、NC308 が用意しているサンプルスタートアッププログラム「ncrt0.a30」と「sect308.inc」について説明します。

スタートアッププログラムの役割

スタートアッププログラムの役割を示します。

- (1) スタック領域の確保
- (2) マイコンの初期設定
- (3) 静的変数領域の初期化
- (4) 割り込みテーブルレジスタ "INTB" の設定
- (5) main 関数の呼び出し
- (6) 割り込みベクタテーブルの設定

サンプルスタートアッププログラムの構成

NC308のサンプルスタートアッププログラムは "ncrt0.a30" と "sect308.inc" の2つのファイルで構成されています。

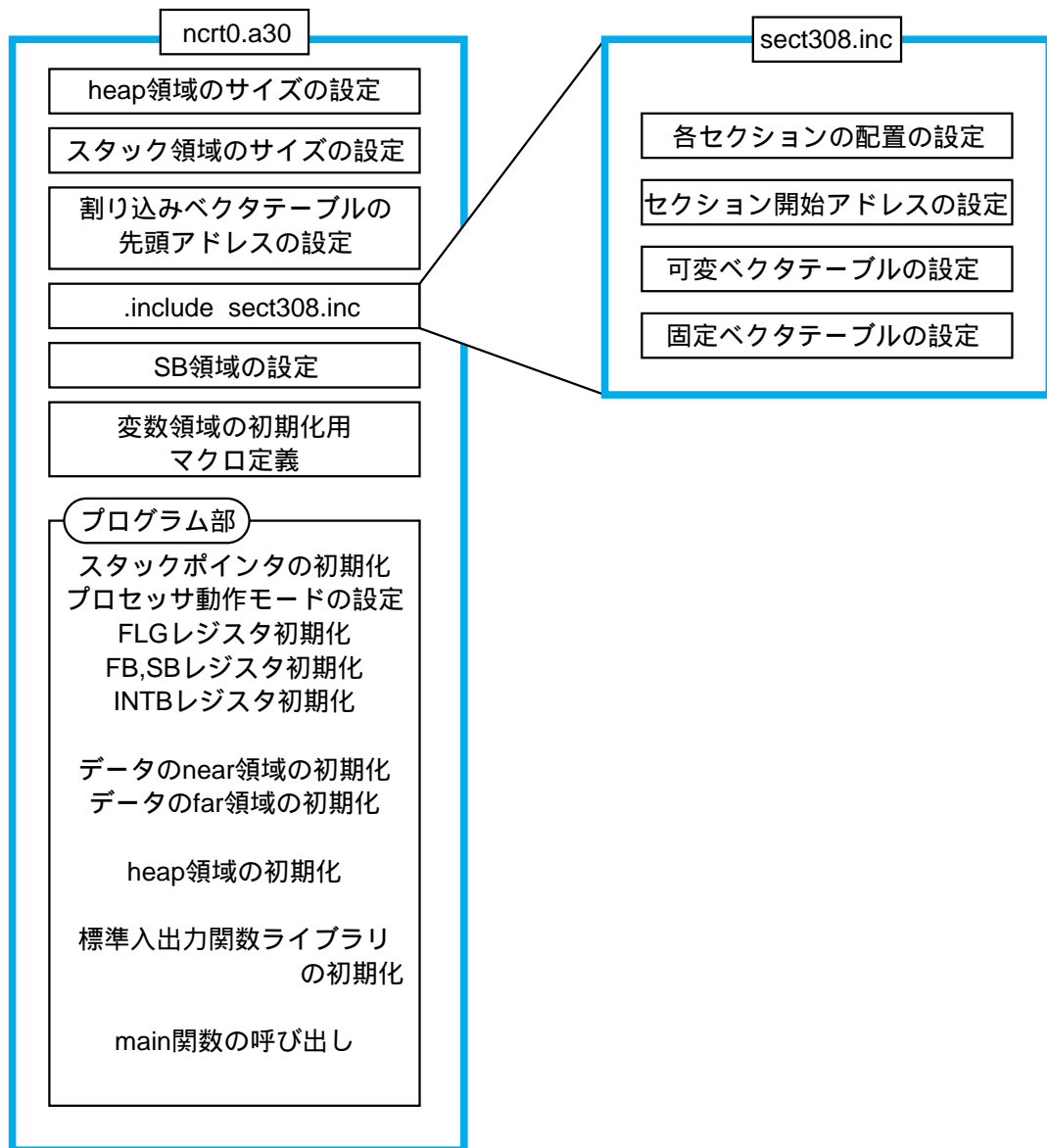


図 2.2.1 サンプルスタートアッププログラムの構成

2.2.2 スタック使用量の見積もり

スタートアッププログラムの中で適切なスタックサイズを設定します。スタックサイズが小さすぎればシステムの暴走につながりますし、大きすぎればメモリの無駄遣いです。

この項では、適切なスタックサイズの見積もり方について説明します。

スタックを使用するもの

スタックを使用するものには次のようなものがあります。

- (1) 自動変数領域
- (2) 複雑な演算などに使用するテンポラリ領域
- (3) 戻り先番地
- (4) 旧フレームポインタ
- (5) 関数への引数
- (6) 戻り値が構造体、または共用体の場合の格納アドレス

スタック使用量表示ファイル

個々の関数のスタック使用量を計算します。プログラムリストから見積もることも可能ですが、NC308起動時に起動オプション " - fshow_stack_usage(-fSSU) " を指定するとスタック使用量情報ファイル " xxx.stk " を生成します。ただし、アセンブリ言語のサブルーチン呼び出しおよびインラインアセンブラによって使用したスタックについては出力されません。これらは、プログラムリストから算出してください。

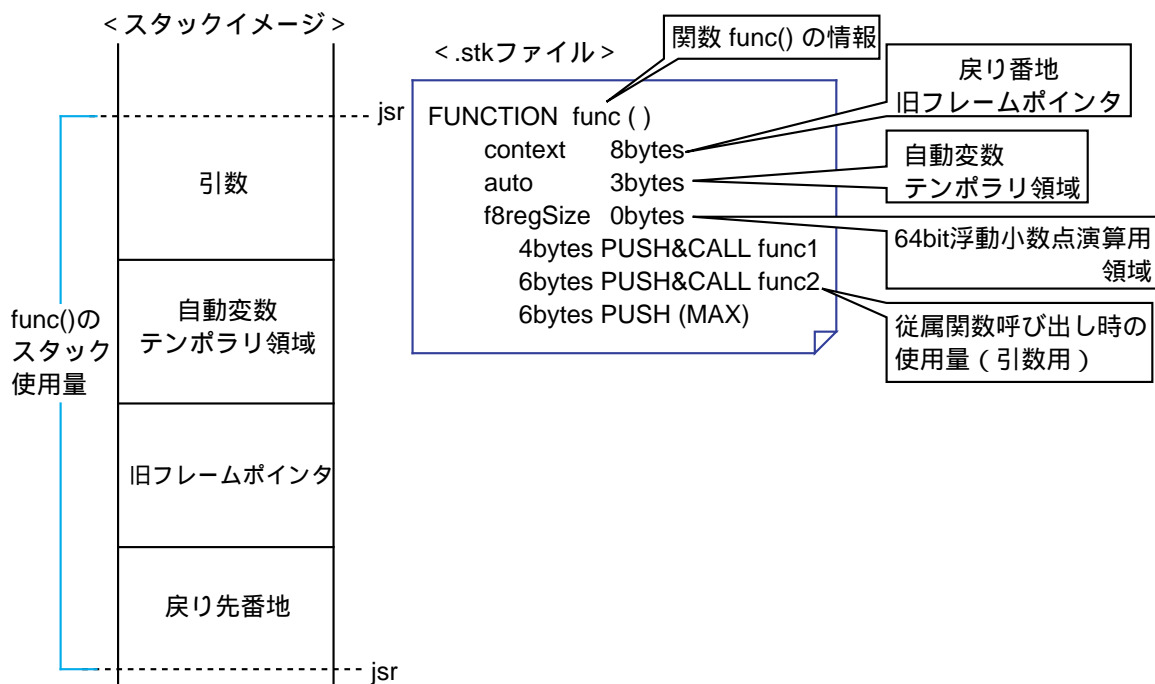


図 2.2.2 スタック使用量情報ファイル

最大スタック使用量の計算

個々の関数のスタック使用量から、関数の呼び出し関係や割り込みを考慮し、最大のスタック使用量を求めます。

図 2.2.3 にサンプルプログラムを使った最大スタック使用量の計算例を示します。

<ソースファイル" sample.c ">

```
void main ( void );
int func1 ( int , int );
int func2 ( int , int );
int func3 ( int );

void main ( void )
{
    int m , n ;
    long kekka1 , kekka2 ;

    kekka1 = func1 ( m , n );
    kekka2 = func2 ( m , n );
}

int func1 ( int x , int y )
{
    int z1 , z2 ;

    z1 = x + y ;
    z2 = func3 ( z1 );

    return z2 ;
}

int func2 ( int x , int y )
{
    int z ;

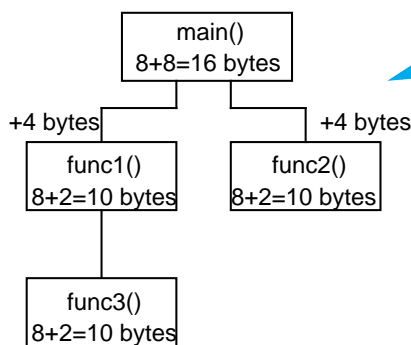
    z = x - y ;
    return z ;
}

int func3 ( int x )
{
    return x ;
}
```

%nc308 -fshow_stack_usage sample.c

<スタック使用量情報ファイル" sample.stk ">

FUNCTION main			
context	8 bytes	↑ 個々の関数で使用 するスタック量	
auto	8 bytes		
f8regSize	0 bytes		
	4 bytes PUSH &	CALL func1	
	4 bytes PUSH &	CALL func2	
	4 bytes PUSH (MAX)		
=====			
FUNCTION func1			
context	8 bytes	↑ 関数呼び出し時に 使用するスタック量	
auto	2 bytes		
f8regSize	0 bytes		
	0 bytes PUSH &	CALL func3	
	0 bytes PUSH (MAX)		
=====			
FUNCTION func2			
context	8 bytes		
auto	2 bytes		
f8regSize	0 bytes		
	0 bytes PUSH (MAX)		
=====			
FUNCTION func3			
context	8 bytes		
auto	2 bytes		
f8regSize	0 bytes		
	0 bytes PUSH (MAX)		
=====			



のパスの時：16+4+10+10=40 bytes
のパスの時：16+4+10=30 bytes

最大スタック使用量は40 bytes

図 2.2.3 最大スタック使用量の計算方法

スタックの最大使用量の自動算出

プログラム構成が単純であれば前述の方法でスタックを見積もることが可能です。しかし、プログラムの構成が複雑である場合、内部関数を使用している場合などは、計算が複雑になります。そのような場合は、NC308 付属の「スタックサイズ算出ユーティリティ "stk308"」を使用します。stk308 は、コンパイル時に出力したスタック使用量情報ファイル "xxx.stk" をもとに、自動的に最大スタック使用量を算出し、標準出力に出力します。また、起動オプション '-o' をつけると、計算結果と関数の呼び出し関係を「算出結果表示ファイル "xxx.siz"」に出力します。

割り込みスタックサイズを見積もるためには、割り込み関数とその割り込み関数から呼び出される関数のスタック使用量を算出する必要があります。この場合は起動オプション '-e 関数名' を使用します。'-o' と合わせて使用すると、指定した関数以下でのスタック使用量と関数の呼び出し関係を出力します。

前述のサンプルプログラムを例に、stk308 の処理結果を図 2.2.4 に示します。

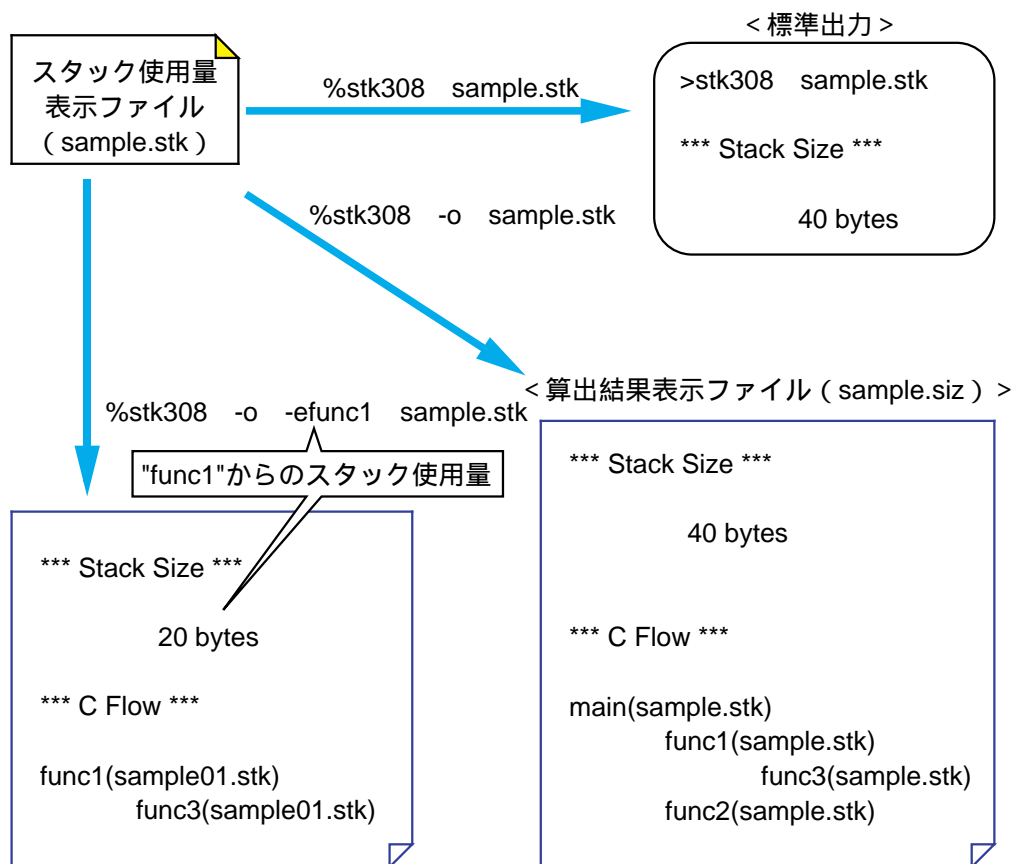


図 2.2.4 スタックサイズ算出ユーティリティ "stk308"

2.2.3 スタートアッププログラムの作成

サンプルスタートアッププログラムは、作成するプログラムに合わせて変更する必要があります。
この項では、具体的なサンプルスタートアッププログラムの変更方法を説明します。

サンプルスタートアッププログラムの変更

作成するプログラムに合わせて、以下の点を変更してください。

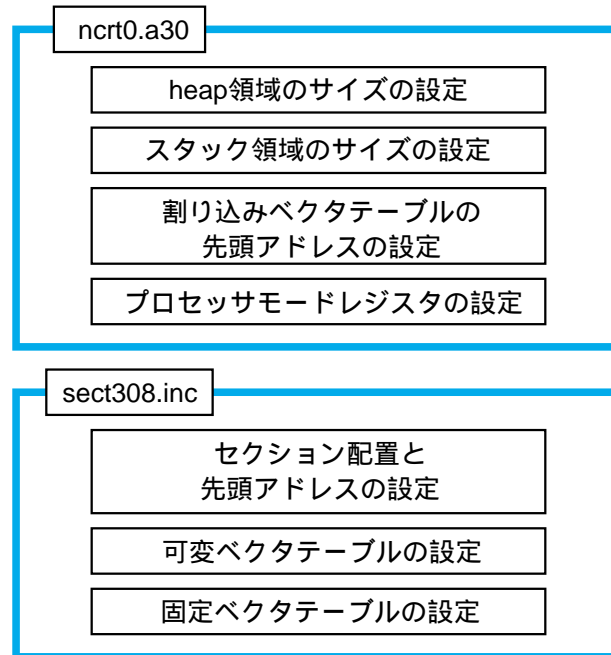


図 2.2.5 サンプルスタートアッププログラムの変更点

heap 領域のサイズの設定 (" ncr0.a30 ")

メモリ管理関数 (calloc、malloc) を使用する際に確保するメモリ使用量を設定します。メモリ管理関数を使用しない場合は、'0' を設定してください。このとき、" ncr0.a30 " 内で heap 領域の初期化を行っている部分をコメントにすると、不要なライブラリがリンクされず、ROM サイズを節約できます。

```

;-----
; HEAP SIZE definition
;-----
HEAPSIZE          .equ    0
    
```

メモリ管理関数を使用しない場合は
'0'を設定し、heap領域の初期化部
をコメントにする

```

;-----
; heap area initialize
;-----
;
; .glb    __mbase
; .glb    __mnext
; .glb    __msize
;
; mov.l   #(heap_top&0FFFFFFH),__mbase
; mov.l   #(heap_top&0FFFFFFH),__mnext
; mov.l   #(heap_top&0FFFFFFH),__msize
    
```

図 2.2.6 heap 領域の設定

スタックサイズの設定 (" ncr0.a30 ")

スタックサイズ算出ユーティリティ " stk308 " などから得られた結果をもとに、ユーザースタックサイズと割り込みスタックサイズを設定します。

多重割り込みを使用する場合は、該当する割り込みのスタック使用量の和を割り込みスタックサイズとして設定してください。

```

;-----
; STACK SIZE definition
;-----
STACKSIZE         .equ    300H
;
;-----
; INTERRUPT STACK SIZE definition
;-----
ISTACKSIZE        .equ    300H
    
```

多重割り込みを使用する場合は該当する
割り込みのスタック使用量の和を設定

図 2.2.7 スタックサイズの設定

割り込みベクタテーブルの先頭アドレスの設定 (" ncrct0.a30 ")

割り込みベクタテーブルの先頭アドレスを設定します。この設定値は " ncrct0.a30 " 内で割り込みテーブルレジスタ " INTB " に設定されます。

```

;-----
; INTERRUPT VECTOR ADDRESS definition
;-----
VECTOR_ADR      .equ  0FFFD00H
;-----
; interrupt section start
;-----
                .glb      start
                .section  interrupt
start:
;-----
; after reset , this program will start
;-----
                ldc      #VECTOR_ADR,intb
;-----

```

割り込みテーブルレジスタ " INTB " に設定

図 2.2.8 割り込みベクタテーブルの先頭アドレスの設定

プロセッサ動作モードの設定 (" ncrct0.a30 ")

プロセッサ動作モードの設定を行います。同様にシステムクロックの設定などM16C/80の動作を直接制御する命令はここに追加します。追加場所および記述方法を図 2.2.9 に示します。

```

;-----
; interrupt section start
;-----
                .glb      start
                .section  interrupt
start:
;-----
; after reset , this program will start
;-----
                ldc      #istack_top-1,isp
;-----
                mov.b   #0000011B,000AH ; disable register protect
                mov.b   #10000111B,0004H ; processor mode register 0
                mov.b   #00001000B,0006H ; system clock control register 0
                mov.b   #00100000B,0007H ; system clock control register 1
                mov.b   #00000000B,000AH ; enable register protect
;-----
                ldc     #0080H,flg
                ldc     #stack_top-1,sp
;-----
                ldc     #stack_top-1,fb
                ldc     #data_SE_top,sb
;-----
                ldc     #VECTOR_ADR,intb
;-----

```

リセット後プログラムはこのラベルからスタートする

システムに合わせた設定を追加する

図 2.2.9 プロセッサ動作モードの設定

各セクションの配置と先頭アドレスの設定 (" sect308.inc ")

NC308が生成するセクションの配置と先頭アドレスを設定します。セクションの先頭アドレスは疑似命令 ".org" を用いて指定します。

先頭アドレス指定がないセクションについては、前に定義したセクションに連続したメモリに配置されます。

```

; -----
; Arrangement of section
; -----
; Near RAM data area
; -----
; SBDATA area
    .section      data_SE,DATA
    .org          400H
data_SE_top:
;
    .section      bss_SE,DATA,ALIGN
bss_SE_top:
    :
; -----
; Far RAM data area
; -----
    .section      data_FE,DATA
    .org          10000H
data_FE_top:
    :
; -----
; Far ROM data area
; -----
    .section      rom_FE,ROMDATA
    .org          0FF0000H
rom_FE_top:
    :

```

メモリマップに合わせて各領域の先頭番地を指定する

図 2.2.10 セクションの先頭アドレスの設定

可変ベクタテーブルの設定 (" sect308.inc ")

可変ベクタテーブルに関する設定事項をセクション定義ファイル " sect308.inc " に追加します。設定例を図 2.2.11 に示します。

```

; -----
;
; variable vector section
; -----
;
; .section          vector          ; variable vector table
; .org             VECTOR_ADR
;
; .lword           dummy_int        ; vector 0 ( BRK )
; .org             ( VECTOR_ADR + 32 )
; .lword           dummy_int        ; DMA0 ( software int 8 )
; .lword           dummy_int        ; DMA1 ( software int 9 )
; .lword           dummy_int        ; DMA2 ( software int 10 )
; .lword           dummy_int        ; DMA3 ( software int 11 )
; .lword           dummy_int        ; TIMER A0 ( software int 12 )
; .lword           dummy_int        ; TIMER A1 ( software int 13 )
; .lword           dummy_int        ; TIMER A2 ( software int 14 )
; .lword           dummy_int        ; TIMER A3 ( software int 15 )
; .lword           dummy_int        ; TIMER A4 ( software int 16 )
; .lword           dummy_int        ; UART0 trance (software int 17)
; .lword           dummy_int        ; UART0 receive (software int 18)
; .lword           dummy_int        ; UART1 trance ( software int 19 )
; .lword           dummy_int        ; UART1 receive (software int 20)
; .lword           dummy_int        ; TIMER B0 ( software int 21)
; .lword           dummy_int        ; TIMER B1 ( software int 22)
; .lword           dummy_int        ; TIMER B2 ( software int 23)
; .lword           dummy_int        ; TIMER B3 ( software int 24)
; .lword           dummy_int        ; TIMER B4(software int 25)
; .lword           dummy_int        ; INT5 (software int 26)
; .lword           dummy_int        ; INT4 (software int 27)
; .lword           dummy_int        ; INT3 ( software int 28)
; .lword           dummy_int        ; INT2(software int 29)
; .lword           dummy_int        ; INT1 (software int 30)
; .lword           dummy_int        ; INT0 ( software int 31)
; .lword           dummy_int        ; TIMER B5(software int 32)
; .lword           dummy_int        ; uart2 trance/NACK(software int 33)
; .lword           dummy_int        ; uart2 receive/ACK(software int 34)
; .lword           dummy_int        ; uart3 trance/NACK(software int 35)
; .lword           dummy_int        ; uart3 receive/ACK(software int 36)
; .lword           dummy_int        ; uart4 trance/NACK(software int 37)
; .lword           dummy_int        ; uart4 receive/ACK(software int 38)
; .lword           dummy_int        ; uart2 bus collision(software int 39)
; .lword           dummy_int        ; uart3 bus collision(software int 40)
; .lword           dummy_int        ; uart4 bus collision(software int 41)
; .lword           dummy_int        ; AD Convert ( software int 42)
; .lword           dummy_int        ; input key ( software int 43 )
; .lword           dummy_int        ; software int (software int 44 )
; .lword           dummy_int        ; software int (software int 45 )
; .lword           dummy_int        ; software int (software int 46 )
; .lword           dummy_int        ; software int (software int 47 )
; .lword           dummy_int        ; software int (software int 48 )
; .lword           dummy_int        ; software int (software int 49 )
; .lword           dummy_int        ; software int (software int 50 )
; .lword           dummy_int        ; software int (software int 51 )
; .lword           dummy_int        ; software int (software int 52 )
; .lword           dummy_int        ; software int (software int 53 )
; .lword           dummy_int        ; software int (software int 54 )
; .lword           dummy_int        ; software int (software int 55 )
; .lword           dummy_int        ; software int (software int 56 )
; .lword           dummy_int        ; software int (software int 57 )
; .lword           dummy_int        ; software int (software int 58 )
; .lword           dummy_int        ; software int (software int 59 )
; .lword           dummy_int        ; software int (software int 60 )
; .lword           dummy_int        ; software int (software int 61 )
; .lword           dummy_int        ; software int (software int 62 )
; .lword           dummy_int        ; software int (software int 63 )
; to vector 63 from vector 44 is used for MR308

```

図 2.2.11 可変ベクタテーブルの設定

固定ベクタテーブルの設定 (" sect308.inc ")

固定ベクタテーブルの先頭アドレスと、各割り込みのベクタアドレスを設定します。記述例を図 2.2.12 に示します。

```

;-----
;
; fixed vector section
;-----
;
;
;-----
;
; .section      fvector          ; fixed vector table
; .org          0FFFE00H
;
;-----
;
; fixed vector section
;-----
;
; .org          0FFFFDCH
;
UDI:
; .lword
OVER_FLOW:
; .lword
B_R_K:
; .lword
ADDRESS_MATCH:
; .lword
; .lword
WDT:
; .lword
; .lword
NMI:
; .lword
RESET:
; .lword
start
    
```

固定ベクタテーブルの先頭アドレスを設定

使用する機能のベクタアドレスを設定する
使用しない場合は " dummy_int " を設定しておく

割り込みベクタとしては何も配置されていない領域

"dummy_int" の処理 (" ncr0.a30 ")

```

;=====
; dummy interrupt function
;-----
dummy_int:
    reit
    :
    
```

図 2.2.12 固定ベクタテーブルの設定

シングルチップモードで使用する時の注意点

M16C/80 シリーズをシングルチップモードで使用する場合は、" near ROM " と " far RAM " 領域は使用しません。図 2.2.13 に示す " ncrct0.a30 " および " sect308.inc " のブロックを削除するか、コメント文にしてください。

ncrct0.a30 : far 領域の初期化プログラム (" FAR area initialize ")
sect308.inc : near ROM 領域の確保 (" Near ROM data area ")
far RAM 領域の確保 (" Far RAM data area ")

```

(" ncrct0.a30 ")
:
;
;-----
; FAR area initialize.
;-----
;
; bss zero clear
;-----
;
; BZERO      bss_FE_top,bss_FE
; BZERO      bss_FO_top,bss_FO
;-----
; Copy edata_E(O) section from edata_EI(OI) section
;-----
;
; BCOPY      data_FEI_top,data_FE_top,data_FE
; BCOPY      data_FOI,data_FO_top,data_FO
;-----
;
; ldc  #stack_top-1,sp
;
;
;-----
(" sect308.inc ")
:
;-----
; Near ROM data area
;-----
;
; .section    rom_NE,ROMDATA,ALIGN
; rom_NE_top:
;
;
; .section    rom_NO,ROMDATA
; rom_NO_top:
;
;-----
; Far RAM data area
;-----
;
; .section    data_EI,DATA
; .org       10000H
; data_FE_top:
;
;
; .section    bss_FE,DATA,ALIGN
; bss_FE_top:
;
;
; .section    data_FO,DATA
; data_FO_top:
;
;
; .section    bss_FO,DATA
; bss_FO_top:
;
;-----

```

コメント文にしておく

図 2.2.13 シングルチップモードで使用する時の記述例

2.3 ROM 化のための拡張機能

2.3.1 効率よいアドレッシング

M16C/80 シリーズのアクセス領域は最大 16M バイトです。NC308 ではこの領域を 000000H ~ 00FFFFH 番地の「near 領域」、000000H ~ FFFFFFFH 番地の領域を「far 領域」と分割して管理しています。この項では、これらの領域への変数、関数の配置方法とアクセス方法を説明します。

near 領域と far 領域

NC308 では最大 16M バイトのアクセス空間を「near 領域」と「far 領域」の 2 つの領域で管理しています。それぞれの領域の特徴を表 2.3.1 に示します。

表 2.3.1 near 領域と far 領域

領域名	内容
near 領域	M16C/80 シリーズがデータを効率よくアクセスできる空間。 絶対番地 000000H ~ 00FFFFH の 64K バイトの領域でスタックや内部 RAM などが配置される。
far 領域	M16C/80 シリーズがアクセスできる絶対番地 000000H ~ FFFFFFFH の 16M バイトの全メモリ空間。 内部 ROM などが配置される。

near / far 属性のデフォルト

NC308 では near 領域に配置される変数、関数を「near 属性」、far 領域に配置される変数、関数を「far 属性」と区別しています。変数、関数のデフォルトの属性を表 2.3.2 に示します。

表 2.3.2 near / far 属性のデフォルト

分類	属性
プログラム	far 固定
RAM データ	near (ただし、ポインタ型は far ^(注))
ROM データ	far
スタックデータ	near 固定

near / far の属性をデフォルトから変更したい場合は、NC308 起動時に次の起動オプションを指定してください。

- ffar_RAM (- fFRAM) ; RAM データのデフォルト属性を " far " に変更
- fnear_ROM (- fNROM) ; ROM データのデフォルト属性を " near " に変更
- fnear_pointer (- fNP) ; ポインタ型のデフォルト属性を "near" に変更

(注) NC308 ではポインタ型変数のサイズはデフォルトで far 型 (4 バイト) の変数となります。NC30 ではポインタ型変数のサイズはデフォルトで near 型 (2 バイト) の変数となります。

関数の near / far

M16C/80 シリーズのアーキテクチャ上、NC308 の関数の属性は far 領域固定です。near を指定した場合、NC308 はコンパイル時にワーニングを出力し、強制的に far 領域に配置します。

変数の near / far

[記憶クラス] 型指定子 near / far 変数名 ;

型宣言時に near / far を指定しなければ、RAM データは near 領域に配置され、const 修飾子を指定したものや ROM データは far 領域へ配置されます。

```
static int data ;
static int near n_data ;
static int far f_data ;
static const int c_data = 0x1234 ;
```

図 2.3.1 静的変数の near / far

自動変数に対しては near / far を指定しても配置には全く影響を及ぼしません (すべてスタック領域に配置されます)。

スタック領域がnear/farどちらの領域に割り付けられるかはシステムに依存する

```
void func(void)
{
    int near i_near ;
    int far i_far ;

    int near * addr_near ;
    int * addr_far ;
    ...
}
```

図 2.3.2 自動変数の near / far

ポインタの near / far

ポインタの near / far 指定により、ポインタに格納するアドレスのサイズやポインタ自身を配置する領域を指定します。

- (1) ポインタに格納するアドレスのサイズを指定します。
なにも指定しなければ far 領域にある変数を指し示す 32 ビット長 (4 バイト) のポインタ変数として扱われます。

```
[ 記憶クラス ] 型指定子 near / far *変数名 ;
```

near ポインタ変数に格納するアドレスのサイズ 16 ビット長

far ポインタ変数に格納するアドレスのサイズ 32 ビット長

```
int near *near_data ;
int far *far_data ;
```

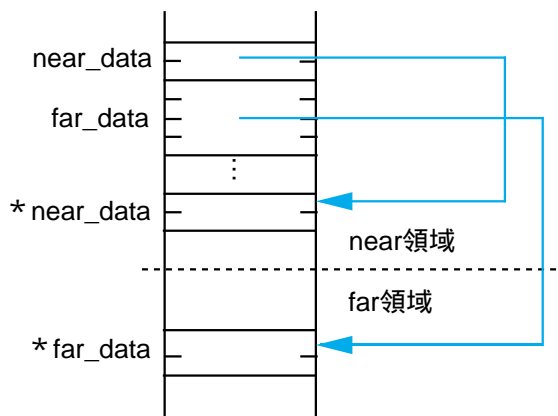


図 2.3.3 ポインタに格納するアドレスサイズを指定

- (2) ポインタ自身が配置される領域を指定します。
なにも指定しなければポインタ変数自身は near 領域に配置されます。

```
[ 記憶クラス ] 型指定子 * near / far 変数名 ;
```

near ポインタ変数自身の領域を near 領域へ配置

far ポインタ変数自身の領域を far 領域へ配置

```
int *near near_data ;
int *far far_data ;
```

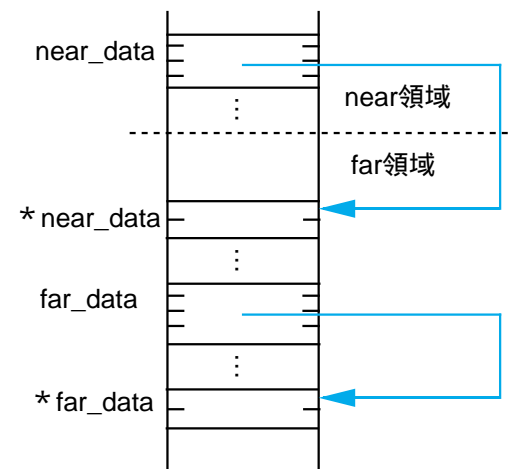


図 2.3.4 ポインタの配置領域を指定

ポインタのnear/far指定におけるNC308とNC30の相違点

M16C/60、M16C/20シリーズ用のCコンパイラNC30ではポインタのnear/farをなにも指定していなければすべてnear属性として扱われていました。NC308ではポインタに格納するアドレスのサイズを指定するときに、なにも指定していなければポインタ変数のサイズは32ビット長(4バイト)となり、far領域にある変数を指し示すポインタ変数として扱われます。

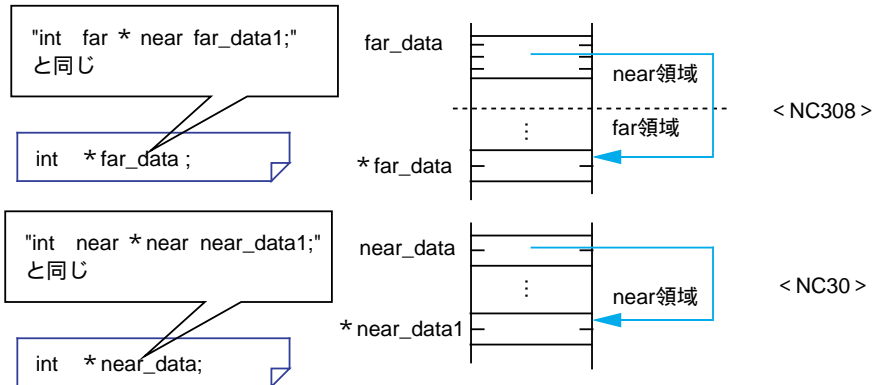


図 2.3.5 ポインタの near/far における NC308 と NC30 の相違点

far領域に配置された変数アドレスのnearポインタへの代入

NC308 は far 領域に配置された変数のアドレスを near ポインタへ代入しようとする、アドレスの上位が無視されて代入が行われることを示すワーニングメッセージを出力します。

また、明示的あるいは暗黙的に far ポインタを near ポインタに変換されたことを示すワーニングメッセージを出力します。

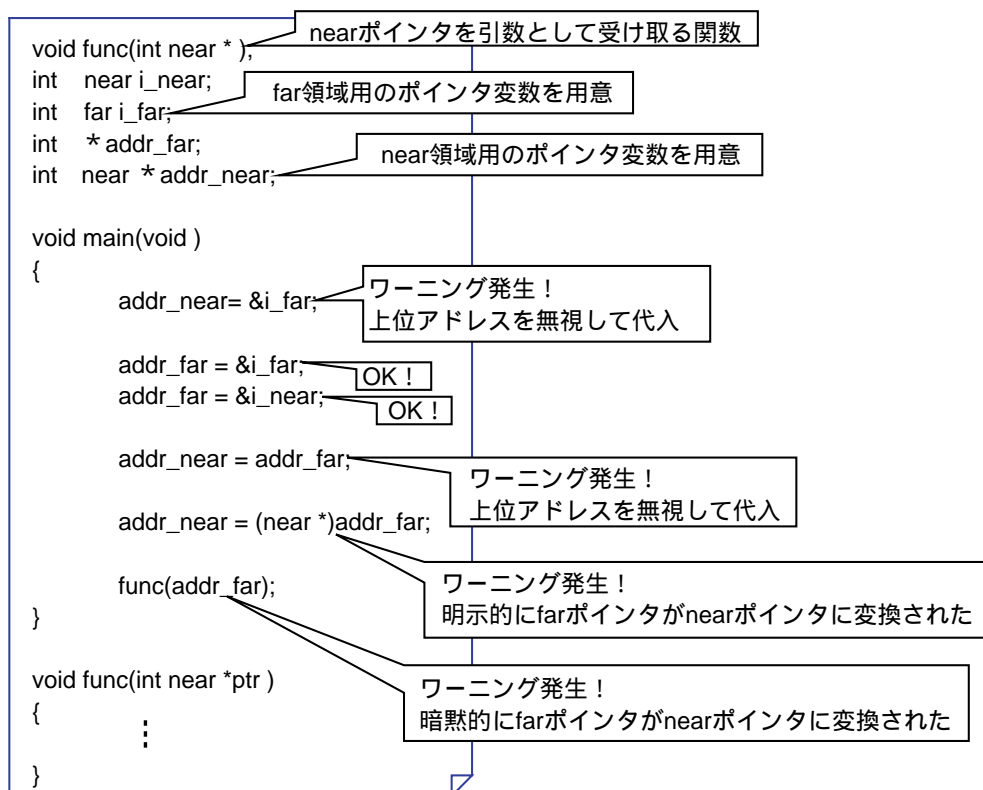


図 2.3.6 far 領域に配置された変数アドレスの near ポインタへの代入

SB 相対アドレッシングを使用する (#pragma SBDATA)

#pragma SBDATA 変数名

この宣言を行った変数については、AS308の疑似命令 ".SBSYM" を生成し、参照時にSB相対アドレッシングモードを使用します。これにより、ROM効率のより高いコードを生成することが可能となります。

```
#pragma SBDATA m
static int m, n;

void main ( void )
{
    m = m + n;
}
```



```
.SBSYM _m
.SECTION program
.glb _main
_main:
add.W _n, _m
rts

.SECTION bss_NE,DATA
_n: .blkb 2

.SECTION bss_SE,DATA
_m: .blkb 2

.end
```

変数mに対して疑似命令 ".SBSYM" を生成

SBアドレッシングを使用するかはアセンブラに依存

変数mはSBDATA属性に配置

図 2.3.7 "#pragma SBDATA" の展開イメージ

2.3.2 ビットの扱い

NC308ではビット単位でデータを扱うことができます。方法は構造体の応用である「ビットフィールド」と拡張機能の2通りです。

この項では、それぞれの方法について説明します。

ビットフィールド

NC308ではビットを扱う方法として、ビットフィールドをサポートしています。ビットフィールドとは、構造体を利用してビットシンボルを割り付ける方法です。書式を以下に示します。

```
struct tag {
    型指定子 ビットシンボル : ビット数 ;
} ;
```

参照するときは構造体、共用体と同様、ピリオド '.' で区切って指定します。

変数名 . ビットシンボル

ビットフィールドを宣言した際のメモリへの割り付け方法はコンパイラによって異なります。NC308の割り付けルールは以下の2つです。図 2.3.6 に具体的な割り付け例を紹介します。

- 下位ビットから順に割り付ける
- データ型の異なるデータは次のアドレスに配置する
(型により確保される領域のサイズが異なる。)

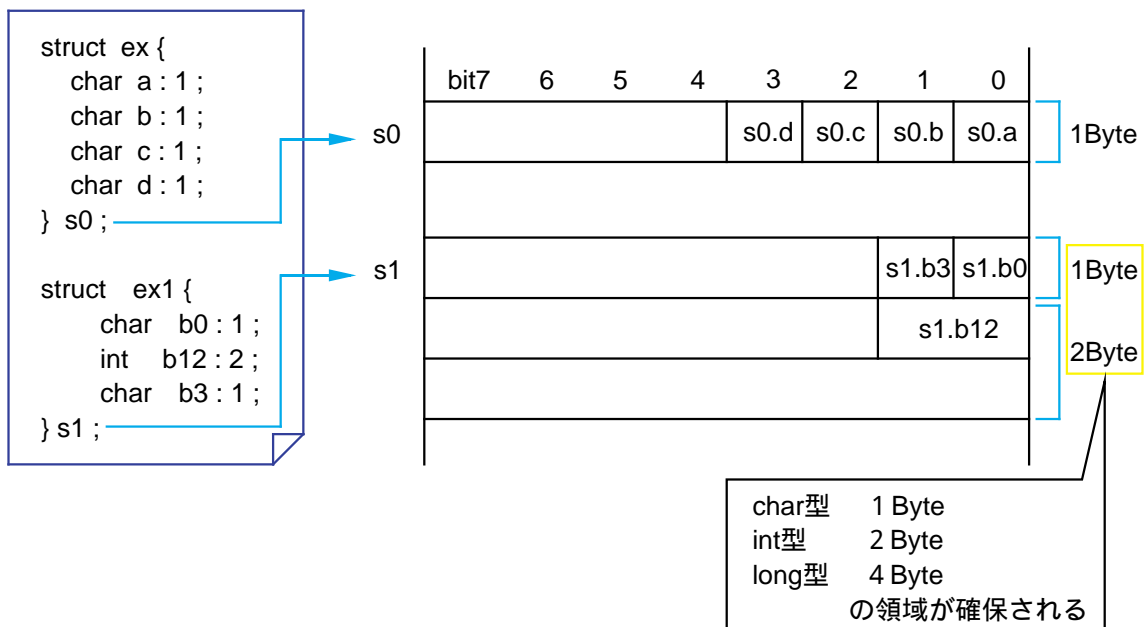


図 2.3.8 ビットフィールドの割り付け例

2.3.3 I/Oインタフェースの制御

組み込み型システムでI/Oインタフェースを制御する際は、変数に対して絶対アドレスを指定します。NC308で絶対アドレスを指定する方法としてはポインタの利用と、拡張機能の利用の2通りの方法があります。この項では、それぞれの方法について説明します。

ポインタによる絶対アドレス指定

ポインタを利用すると絶対アドレス指定を行うことができます。記述例を図 2.3.8 に示します。

例) 00000a番地に0xefを代入する

```
char * point ;
point = (char *)0x00000a ;
*point = 0xef ;
```

|| 1行にまとめると

```
*(char *)0x00000a = 0xef ;
```

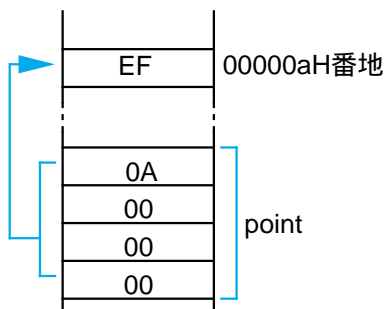
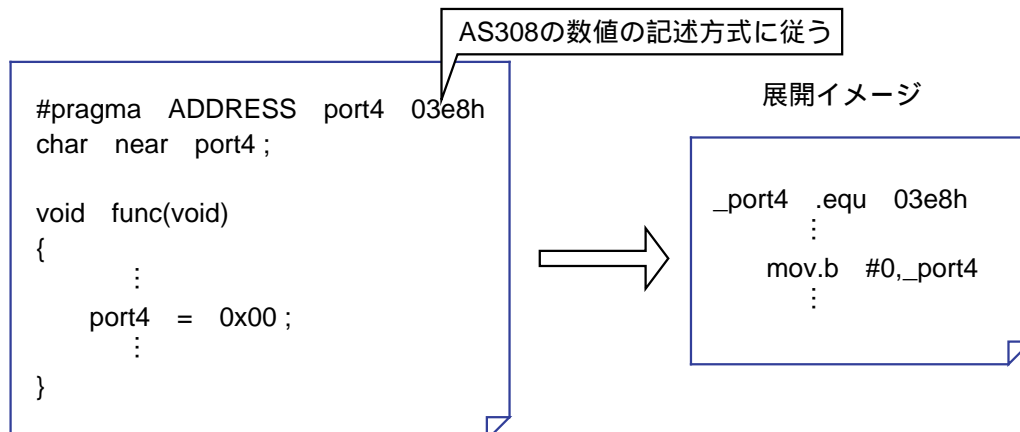


図 2.3.9 ポインタによる絶対アドレス指定

拡張機能による絶対アドレス指定 (#pragma ADDRESS)

```
#pragma ADDRESS 変数名 絶対アドレス
```

上記の宣言により変数名を、絶対アドレスに配置します。この方法では変数名が絶対アドレスとして同義定義されているため、前述の方法のようにポインタ変数領域をとる必要がなく、メモリ使用量を節約できます。



"#pragma ADDRESS"は、関数外で定義された変数にのみ有効

図 2.3.10 "#pragma ADDRESS"による絶対アドレス指定

例 2.3.1 "#pragma ADDRESS" を利用した SFR 領域の定義

SFR 領域の設定を "#pragma ADDRESS" を用いて行います。この SFR の設定は通常、別ファイルとして用意しておき、ソースプログラムの中でインクルードします。

SFR 領域定義ファイルの一例を次に示します。

SFR 領域定義ファイル <M30800.h>

```

/*-----*/
/*   type definition   */
/*-----*/
union {
    struct {
        char b0 : 1;
        char b1 : 1;
        char b2 : 1;
        char b3 : 1;
        char b4 : 1;
        char b5 : 1;
        char b6 : 1;
        char b7 : 1;
    }bit;
    unsigned char all;
}SFR;
:

/*-----*/
/*   input/output port   */
/*-----*/
#pragma ADDRESS P6      03c0h
static SFR P6;
#pragma ADDRESS P7      03c1h
static SFR P7;
#pragma ADDRESS PD6     03c2h
static SFR PD6;
#pragma ADDRESS PD7     03c3h
static SFR PD7;
#pragma ADDRESS P8      03c4h
static SFR P8;
#pragma ADDRESS P9      03c5h
static SFR P9;
#pragma ADDRESS PD8     03c6h
static SFR PD8;
#pragma ADDRESS PD9     03c7h
static SFR PD9;
:

/*-----*/
/*   Port P6 direction register bit   */
/*-----*/
#define PD6_0 PD6.bit.b0 /* P6 direction register bit0 */
#define PD6_1 PD6.bit.b1 /* P6 direction register bit1 */
#define PD6_2 PD6.bit.b2 /* P6 direction register bit2 */
#define PD6_3 PD6.bit.b3 /* P6 direction register bit3 */
#define PD6_4 PD6.bit.b4 /* P6 direction register bit4 */
#define PD6_5 PD6.bit.b5 /* P6 direction register bit5 */
#define PD6_6 PD6.bit.b6 /* P6 direction register bit6 */
#define PD6_7 PD6.bit.b7 /* P6 direction register bit7 */
:
    
```

ビット操作のための型宣言

絶対アドレスの設定

SFR 領域定義ファイルを読み込む

```

<ソースファイル>
#include "M30800.h"
:
void main ( void )
{
:
P6.all = 0x00 ;
PD6_0 =1;
:
    
```

SFR 領域を参照し、バイトアクセス

SFR 領域を参照し、ビットアクセス

例 2.3.1 "#pragma ADDRESS" を利用した SFR 領域の定義

2.3.4 C 言語で書けないときの対処策

「処理時間が間に合わない」、「Cフラグを直接制御したい」などハードウェアに関する処理はC言語では記述できない場合があります。この場合NC308では、C言語のソースプログラム中にアセンブリ言語を直接書き込むことができます（「インラインアセンブル機能」）。インラインアセンブルの方法は「asm関数」を使用する方法と、「#pragma ASM」を使用する方法の2通りがあります。

この項では、それぞれの方法について説明します。

1行だけアセンブリ言語で記述する（asm関数）

asm("文字列")

上記のように記述すると"（ダブルクォーテーション）で囲まれた文字列が、そのまま（スペース、タブも含めて）アセンブリ言語ソースプログラムに展開されます。

この記述は関数の内外問わず記述できるので、フラグやレジスタを直接操作したいときや高速な処理が必要なときに使用します。

記述例を図 2.3.10 に示します。

```
void main (void)
{
    initialize();
    asm(" FSET I");
    ...
}
```

割り込み許可フラグをセット

図 2.3.11 asm 関数の記述例

アセンブリ言語で自動変数をアクセスする（asm関数）

関数内部において自動変数をアクセスしたいときは、"\$\$[FB]"を利用して図 2.3.11 のように記述します。すると、"\$\$"をコンパイラがFBレジスタのオフセット値に置き換えるため、C言語での自動変数名をアセンブリ言語プログラムで使用できます。

```
void main (void)
{
    unsigned int m;
    m = 0x07;
    asm(" MOV.W $$[FB],R0",m);
}
```

自動変数'm'を定義

展開イメージ

```
_main:
    enter          #02H
    mov.w         #0007H,-2[FB] ; m
    ##### ASM START
    MOV.W        -2[FB] R0
    ##### ASM END
    exitd
```

'm'のFBオフセット値は'-2'

FB相対アドレッシングを使用

<書式>

asm("アセンブリ言語", 自動変数名);

図 2.3.12 asm 関数で自動変数を使用する

モジュールごとアセンブリ言語で記述する (#pragma ASM)

埋め込むアセンブリ言語が複数行にわたる場合、" #pragma ASM " を使用します。この拡張機能では、" #pragma ASM " ~ " #pragma ENDASM " で囲まれた部分をアセンブリ言語で記述された領域と判断し、アセンブリ言語ソースプログラムにそのまま出力します。

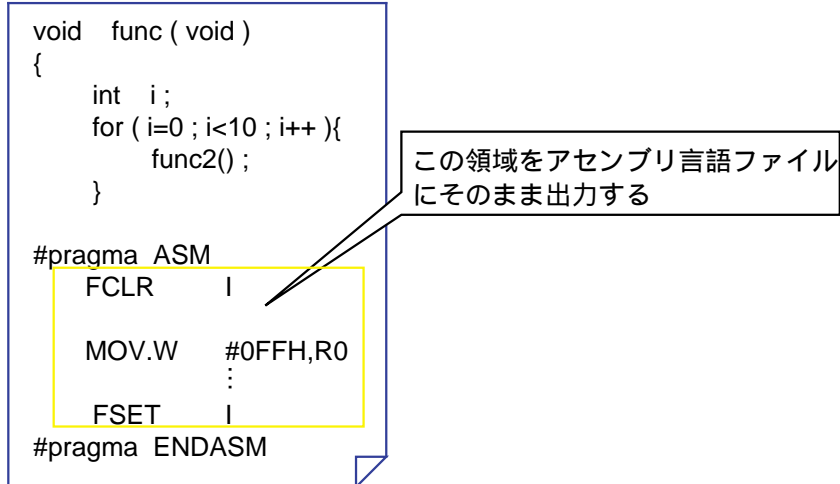


図 2.3.13 " #pragma ASM " 機能使用例

コラム asm 関数を利用した部分的な最適化の抑止

NC308 では最適化オプション ' -O ' をつけると、コンパイル時に生成コードの最適化を行います。しかし、最適化によって割り込み発生時などに不具合が発生する場合、asm 関数を利用して部分的に最適化を抑止することができます。また、同様に最適化オプション " -Ono_bit - (ONB) " を指定すればビット操作をまとめる最適化を抑止することができます。使用例を図 2.3.14 に示します。

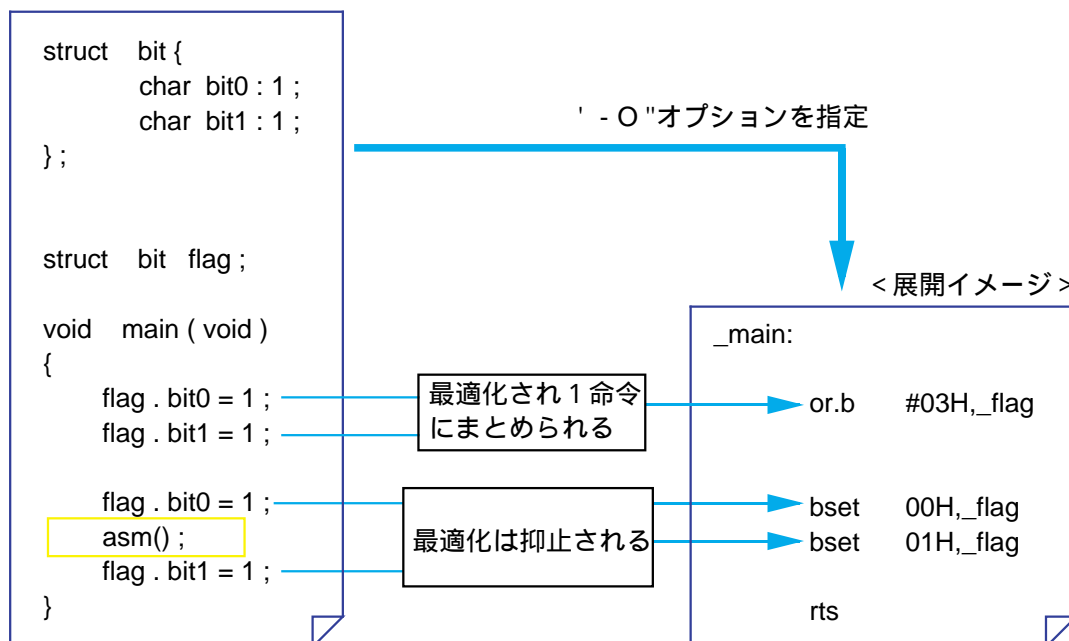


図 2.3.14 asm 関数を利用した部分的な最適化の抑止

2.3.5 アセンブルマクロ関数の使用

NC308 ではアセンブリ言語命令の一部を C 言語の関数として記述することができます (「アセンブルマクロ関数機能」)。

通常の C 言語の記述では NC308 が展開しないアセンブリ言語命令を直接 C 言語のプログラム上に記述できるので、プログラムのチューンナップが行いやすくなります。

この項では、アセンブルマクロ関数の記述方法と使用例について説明します。

アセンブルマクロ関数で記述できるアセンブリ言語命令(1)

NC308 ではアセンブルマクロ関数で 18 種類のアセンブリ言語命令を記述することができます。

アセンブルマクロ関数名はアセンブリ言語命令を半角小文字で表します。演算時のビット長は "_b"、"_w"、"_l" で表します。アセンブルマクロ関数で記述できるアセンブリ言語命令を表 2.3.3、表 2.3.4 に示します。

表 2.3.3 アセンブルマクロ関数で記述できるアセンブリ言語命令(1)

アセンブリ言語命令	アセンブルマクロ関数名	機能	書式
DADD	dadd_b	val1 と val2 の 10 進加算結果を返す	char dadd_b(char val1, char val2);
	dadd_w		int dadd_w(int val1, int val2);
DADC	dadc_b	val1 と val2 のキャリー付き 10 進加算結果を返す	char dadc_b(char val1, char val2);
	dadc_w		int dadc_w(int val1, int val2);
DSUB	dsub_b	val1 と val2 の 10 進減算結果を返す	char dsub_b(char val1, char val2);
	dsub_w		int dsub_w(int val1, int val2);
DSBB	dsbb_b	val1 と val2 のボロー付き 10 進減算結果を返す	char dsbb_b(char val1, char val2);
	dsbb_w		int dsbb_w(int val1, int val2);
RMPA	rmpa_b	初期値 int、回数 count、乗数の格納されている先頭アドレスを p1、p2 として積和演算結果を返す	long rmpa_b(long init, int count, char *p1, char *p2);
	rmpa_w		long rmpa_w(long init, int count, int *p1, int *p2);
MAX	max_b	val1 と val2 の最大値を選択し結果を返す	char max_b(char val1, char val2);
	max_w		int max_w(int val1, int val2);
MIN	min_b	val1 と val2 の最小値を選択し結果を返す	char min_b(char val1, char val2);
	min_w		int min_w(int val1, int val2);

アセンブラマクロ関数で記述できるアセンブリ言語命令(2)

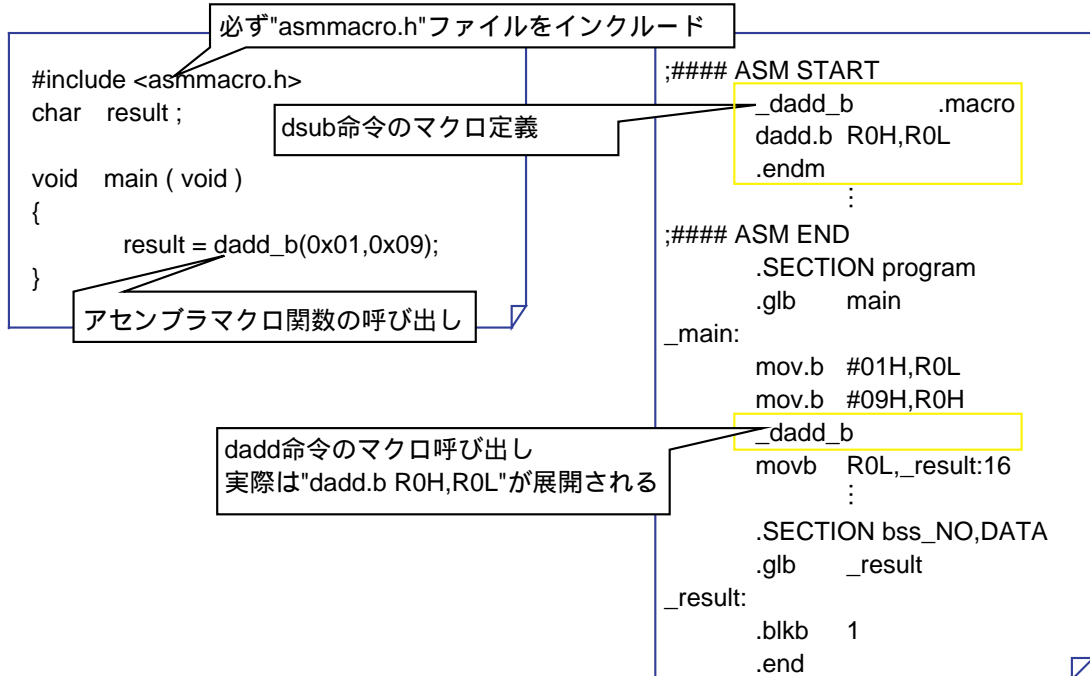
表 2.3.4 アセンブラマクロ関数で記述できるアセンブリ言語命令(2)

アセンブリ言語命令	アセンブラマクロ関数名	機能	書式
SMOVB	smovb_b	転送番地p1から転送番地p2にcount回数分逆方向のストリング転送を行う	void smovb_b(char *p1, char *p2, unsigned int count);
	smovb_w		void smovb_w(int *p1, int *p2, unsigned int count);
SMOVF	smovf_b	転送番地p1から転送番地p2にcount回数分順方向のストリング転送を行う	void smovf_b(char *p1, char *p2, unsigned int count);
	smovf_w		void smovf_w(int *p1, int *p2, unsigned int count);
SMOVU	smovu_b	転送番地p1から転送番地p2に順方向で0が検出されるまでストリング転送を行う	void smovu_b(char *p1, char *p2);
	smovu_w		void smovu_w(int *p1, int *p2);
SIN	sin_b	固定の転送番地p1から転送番地p2にcount回数分順方向のストリング転送を行う	void sin_b(char *p1, char *p2, unsigned int count);
	sin_w		void sin_w(int *p1, int *p2, unsigned int count);
SOUT	sout_b	転送番地p1から転送番地p2にcount回数分順方向のストリング転送を行う	void sout_b(char *p1, char *p2, unsigned int count);
	sout_w		void sout_w(int *p1, int *p2, unsigned int count);
SSTR	sstr_b	ストアするデータval、転送番地p、転送回數countとしてストリングストアを行う	void sstr_b(char val, char *p, unsigned int count);
	sstr_w		void sstr_w(int val, int *p, unsigned int count);
ROLC	rolc_b	valをキャリーを含めて1ビット左回転した結果を返す	unsigned char rolc_b(unsigned char val);
	rolc_w		unsigned int rolc_w(unsigned int val);
RORC	rorc_b	valをキャリーを含めて1ビット右回転した結果を返す	unsigned char rorc_b(unsigned char val);
	rorc_w		unsigned int rorc_w(unsigned int val);
ROT	rot_b	valをcount回数分回転した結果を返す	unsigned char rot_b(signed char val, unsigned char count);
	rot_w		unsigned int rot_w(signed char val, unsigned int count);
SHA	sha_b	valをcount回数分算術シフトした結果を返す	unsigned char sha_b(signed char count, unsigned char val);
	sha_w		unsigned int sha_w(signed char count, unsigned int val);
	sha_l		unsigned long sha_l(signed char count, unsigned long val);
SHL	shl_b	valをcount回数分論理シフトした結果を返す	unsigned char shl_b(signed char count, unsigned char val);
	shl_w		unsigned int shl_w(signed char count, unsigned int val);
	shl_l		unsigned long shl_l(signed char count, unsigned long val);

例2.3.2 アセンブラマクロ関数"dadd_b"を使用した10進加算

NC308のアセンブラマクロ関数を呼び出して使用するときには、アセンブラマクロ関数定義ファイル"asmmacro.h"を必ずインクルードしてください。

アセンブラマクロ関数"dadd_b"を使用した10進加算の例を例2.3.2に示します。



<展開イメージ>

例 2.3.2 アセンブラマクロ関数 "dadd_b" を使用した 10 進加算

例2.3.3 アセンブラマクロ関数"smovf_b"を使用したstring転送

アセンブラマクロ関数 "smovf_b" を使用したstring転送の例を例 2.3.3 に示します。

```

#include <asmmacro.h>
char src_string[] = "ABCDEFGG";
char dest_string[8];

void main ( void )
{
    smovf_b( src_string, dest_string, sizeof(src_strig));
}

;#### ASM START
_smovf_b .macro
pushm R3,A0,A1
smovf.b
popm R3,A0,A1
.endm
:
;#### ASM END
.SECTION program
.glob main
_main:
pushm R3,A0,A1
mov.w #(_dest_string&0FFFFH),A1
mov.w #(_src_string&0FFFFH),A0
_smovf_b
popm R3,A0,A1
rts
:
.SECTION data_NE,DATA
.glob _src_string
_src_string:
.blkb 8

.SECTION data_NEI,ROMDATA
.byte 41H ;'A'
.byte 42H ;'B'
.byte 43H ;'C'
:
.SECTION bss_NE,DATA
.glob _dest_string
_dest_string:
.blkb 8
.end
    
```

必ず"asmmacro.h"ファイルをインクルード

smovf命令のマクロ定義

アセンブラマクロ関数の呼び出し

smovf命令のマクロ呼び出し
実際は
pushm R3,A0,A1
smovf.b
popm R3,A0,A1
が展開される

<展開イメージ>

例 2.3.3 アセンブラマクロ関数 "smovf_b" を使用したstring転送

例 2.3.4 アセンブラマクロ関数 "rmpa_w" を使用した積和演算

アセンブラマクロ関数 "rmpa_w" を使用した積和演算の例を例 2.3.4 に示します。

```

#include <asmmacro.h>
int str1[10] = {0,1,2,3,4,5,6,7,8,9};
int str2[10] = {0,1,2,3,4,5,6,7,8,9};
long result;

void main (void)
{
    result = rmpa_w(0, 9, str1, str2);
}
        
```

必ず"asmmacro.h"ファイルをインクルード

rmpa命令のマクロ定義

初期値 積和回数

アセンブラマクロ関数の呼び出し

```

;#### ASM START
:
_rmpa_w .macro
pushm R1,R3,A1,A0
mov.w #00H,R1
rmpa.w
popm R1,R3,A1,A0
.endm

;#### ASM END
.SECTION program
.glob main
_main:
mov.w #(_str2&0FFFFH),A1
mov.w #(_str1&0FFFFH),A0
mov.w #0009H,R3
mov.l #00000000H,R2R0
_rmpa_w
mov.l R2R0,_result:16
:
.SECTION data_NE,DATA
.glob _str1
_str1:
.blkb 20
.glob _str2
_str2:
.blkb 20
:
.SECTION data_NEI,ROMDATA
.word 0000H
.word 0001H
.word 0002H
.word 0003H
:
.SECTION bss_NO,DATA
.glob _result
_result:
.blkb 4
.end
        
```

rmpa命令のマクロ呼び出し
実際は
pushm R1,R3,A1,A0
mov.w #00H,R1
rmpa.w
popm R1,R3,A1,A0
が展開される

<展開イメージ>

例 2.3.4 アセンブラマクロ関数 "rmpa_w" を使用した積和演算

(注) マクロアセンブラ関数定義ファイル"asmmacro.h"はNC308の環境変数"INC308"で設定した標準ディレクトリに格納されてい
ます。

2.4 アセンブリ言語とのリンク

2.4.1 関数間のインタフェース

モジュールサイズが小さい場合はインラインアセンブルで対応できます。しかし、モジュールサイズが大きい場合や、既存のモジュールを流用するとき、アセンブリ言語のサブルーチンをC言語プログラムから呼び出すことができます。また、その逆も可能です。

この項では、NC308 の関数間のインタフェースを説明します。

関数の入口処理と出口処理

NC308 での関数を呼び出す際の主な処理は次の3つです。

- (1) スタックフレームの構築と解除
- (2) 引数の受け渡し
- (3) 戻り値の受け渡し

これらの動作の手順を図 2.4.1 に示します。

```

int func ( int , int ) ;
void main ( void )
{
    int a = 3 , b = 5 ;
    int c ;
    ⋮
    c = func ( a , b ) ;
    ⋮
}
int func( int x , int y )
{
    ⋮
}
    
```

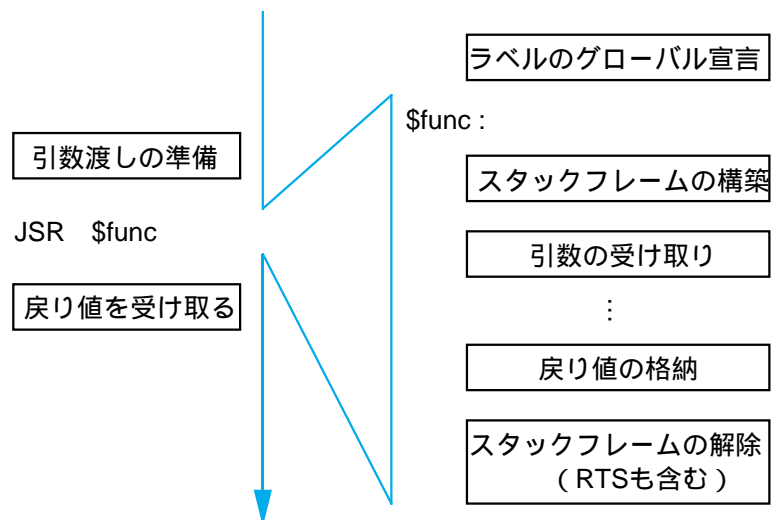


図 2.4.1 関数の呼び出し動作

関数呼び出し時の ROM サイズ圧縮方法

NC308 では、別ファイルに定義される関数を呼び出すとき、jsr.a 命令 (4 バイトで構成) に展開して関数呼び出しを行います。したがって、64K バイト内(-32768 ~ +32767)に存在する関数を呼び出す場合もフルアドレス指定 (オペランド 3 バイト) で展開するため、ROM 効率が悪くなります。

これを生成コード変更オプション「-fJSRW」を指定してコンパイルを行うと、関数呼び出し命令を jsr.w 命令 (3 バイトで構成) に展開して関数呼び出しを行うことができ、ROM サイズの圧縮を行うことができます。

リンク時にエラーが発生した場合 (16 ビット相対で届かなかった場合) はその関数に対して "#pragma JSRA" 宣言することで、jsr.a 命令に展開して関数呼び出しを行うことができますので、64K バイトを超えるプログラムにも対応することができます。したがって、プログラムが 64K バイト内に納まっている場合は生成コード変更オプション「-fJSRW」を使用することを推奨します。

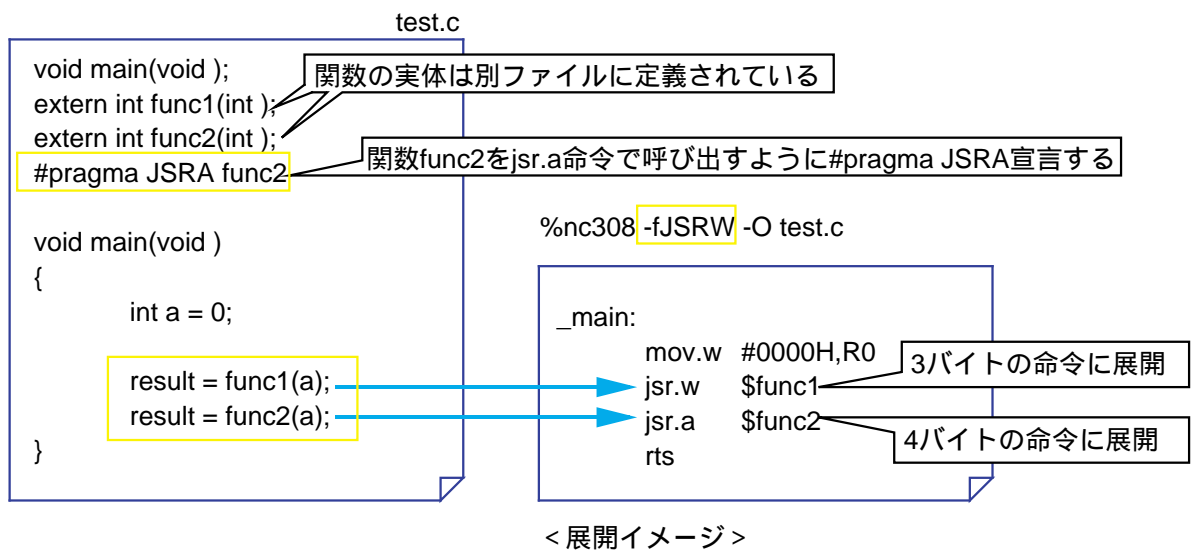
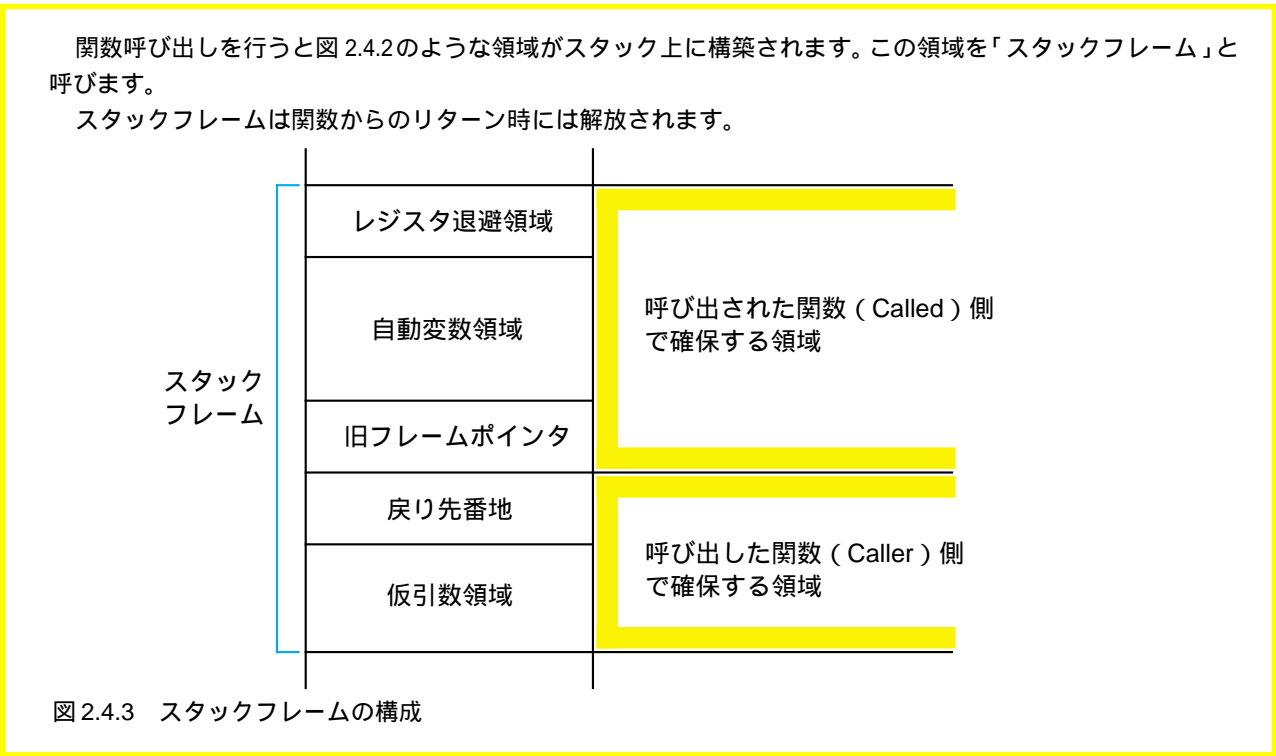
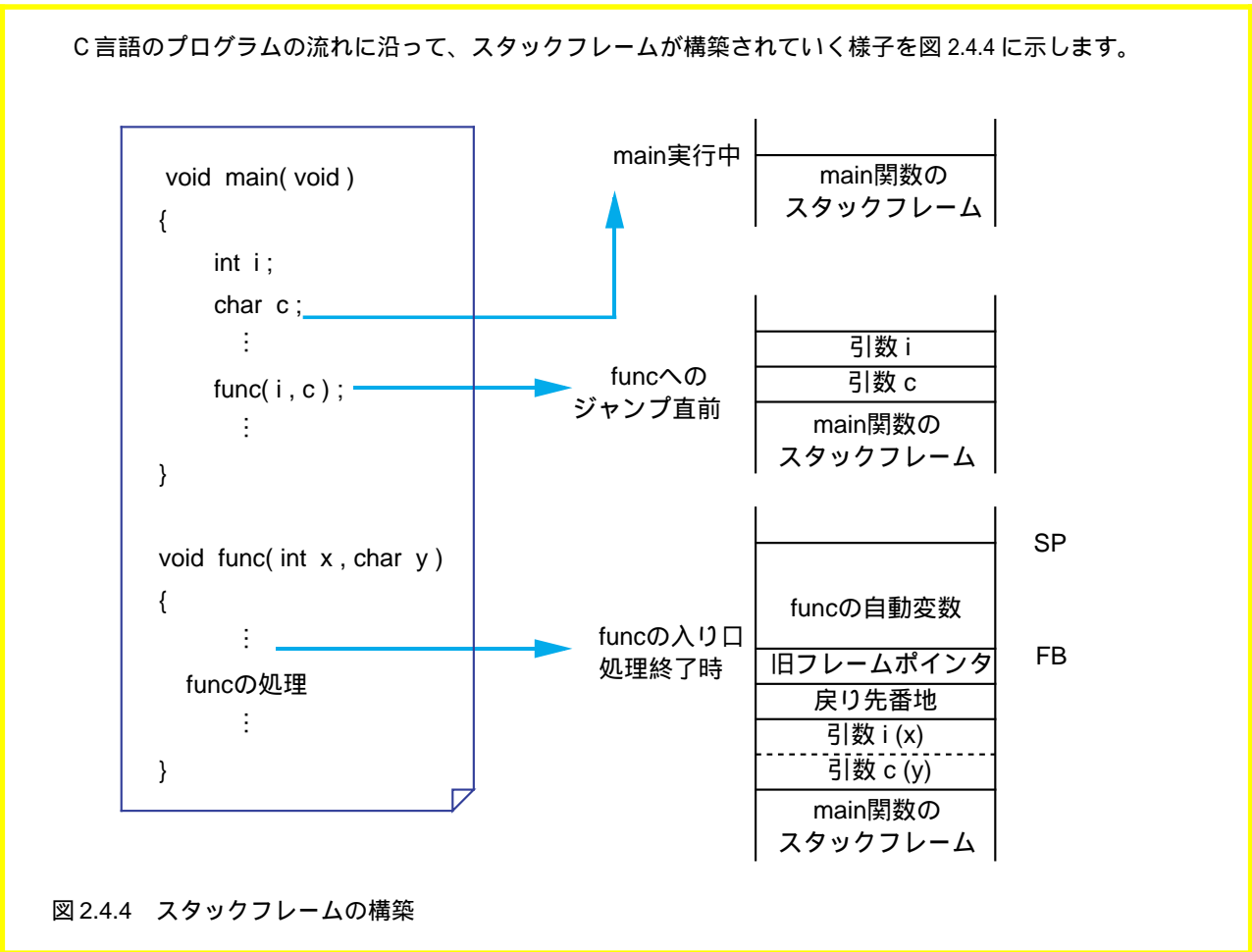


図 2.4.2 関数呼び出し時の ROM サイズ圧縮方法

スタックフレームの構成



スタックフレームの構築



引数の引き渡し規則

NC308 では関数へ引数を渡す方法は「レジスタ渡し」と「スタック渡し」の2通りがあります。
次の3つの条件がそろえばレジスタ渡しになり、それ以外はスタック渡しになります。

- (1) 関数の引数の型がプロトタイプ宣言されていること
- (2) 引数の型として表 2.4.1 に示す引数と引数の型が一致していること
- (3) プロトタイプ宣言の引数部分に可変引数 "..." を使用していないこと

表 2.4.1 引数の引き渡し例

引数の型	第 1 引数	第 2 引数以降
char型	R0L	スタック
short、int型 nearポインタ型	R0	スタック
その他の型	スタック	スタック

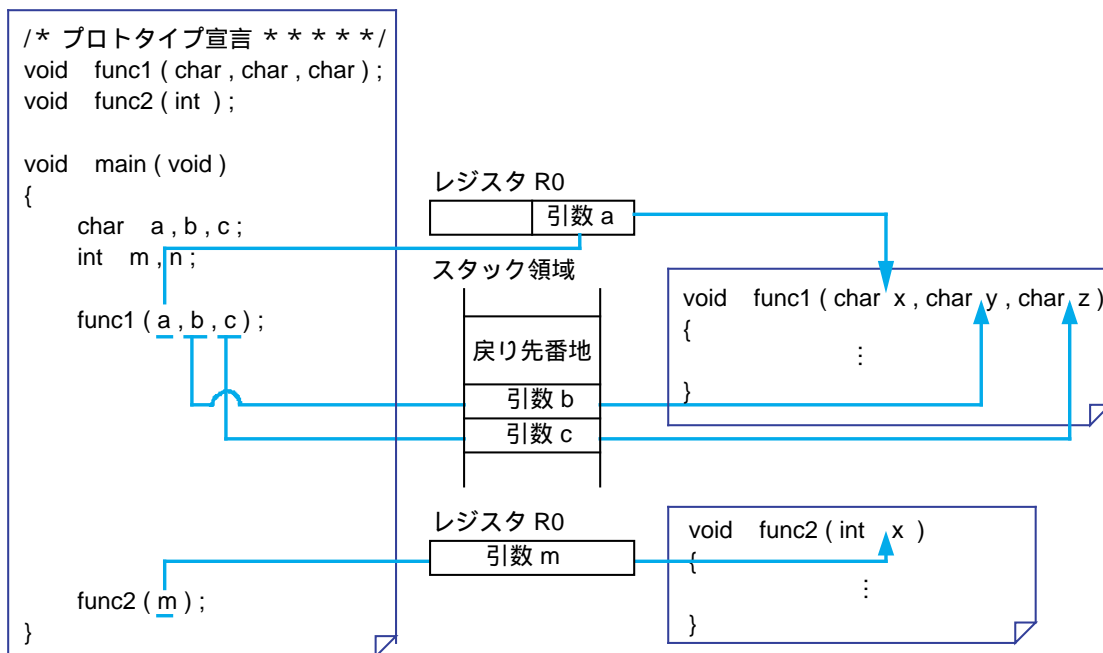


図 2.4.5 引数の引き渡し例

戻り値の引き渡し規則

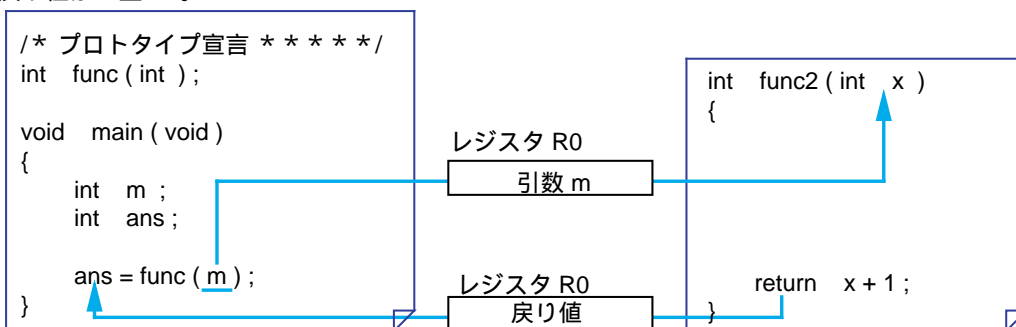
戻り値は、構造体・共用体を除き、すべてレジスタに格納されます。ただし戻り値のデータ型によって格納されるレジスタが異なります。

構造体・共用体が戻り値の場合は「格納アドレススタック渡し」となります。つまり、関数呼び出し時に戻り値を格納する領域を用意し、そのアドレスを隠れた引数としてスタック渡しを行います。呼び出された関数側ではリターン時に、スタックに積まれたアドレスが示す領域に戻り値を書き込みます。

表 2.4.2 戻り値の引き渡し規則

データ型	返し方
char	R0L
int short	R0
long float	R2R0
double	R3R2R1R0
nearポインタ	R0
farポインタ	R2R0
struct union	格納アドレスをスタック渡し

・戻り値がint型の時



・戻り値が構造体の時

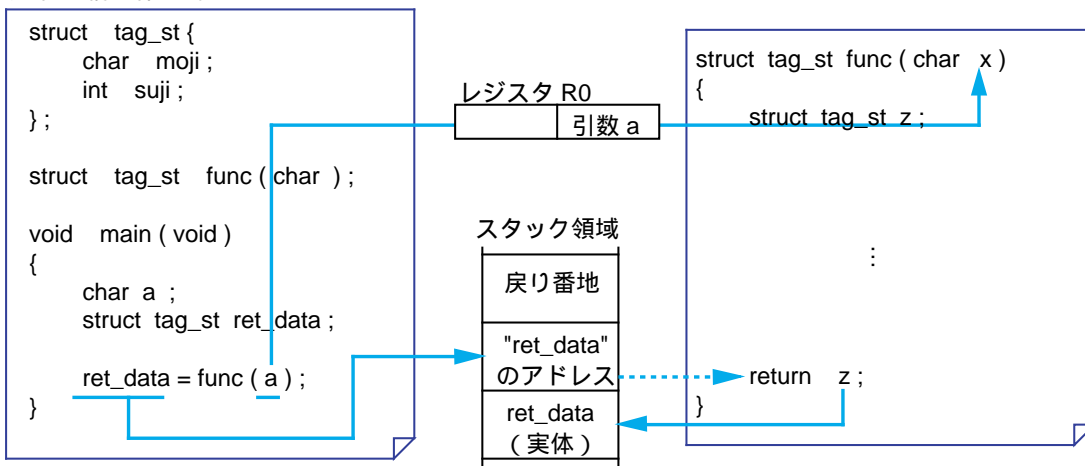


図 2.4.6 戻り値の引き渡し例

関数のアセンブリ言語へのシンボル変換規則

NC308では関数の性質によって、変換されるシンボルが異なります。シンボル変換規則を表 2.4.3に示します。

表 2.4.3 シンボル変換規則

関数の種類	変換方法
引数レジスタ渡し	関数の先頭に「\$」を付加
引数スタック渡し 引数なし #pragma INTERRUPT[B/E/F] #pragma PARAMETER[C]	関数の先頭に「_」を付加

コラム より速く関数を呼び出すためには

関数を呼び出す際には戻り値や引数の引き渡しなどでスタック操作が必要となり、実際の処理が行われるまでに時間がかかります。したがって、スタック操作の少ないレジスタ渡しの方が、関数呼び出しから処理までの時間が短くなります。

NC308ではこの時間差をさらに短くするために「インライン記憶クラス」を用意しています。インライン記憶クラスを指定した関数は、コンパイル時にマクロ関数としてコード生成され、関数を呼び出したところに直接コードが展開されます。このため関数呼び出し時間、および通常のスタック操作が無くなり、実行速度を向上させることができます(注)。

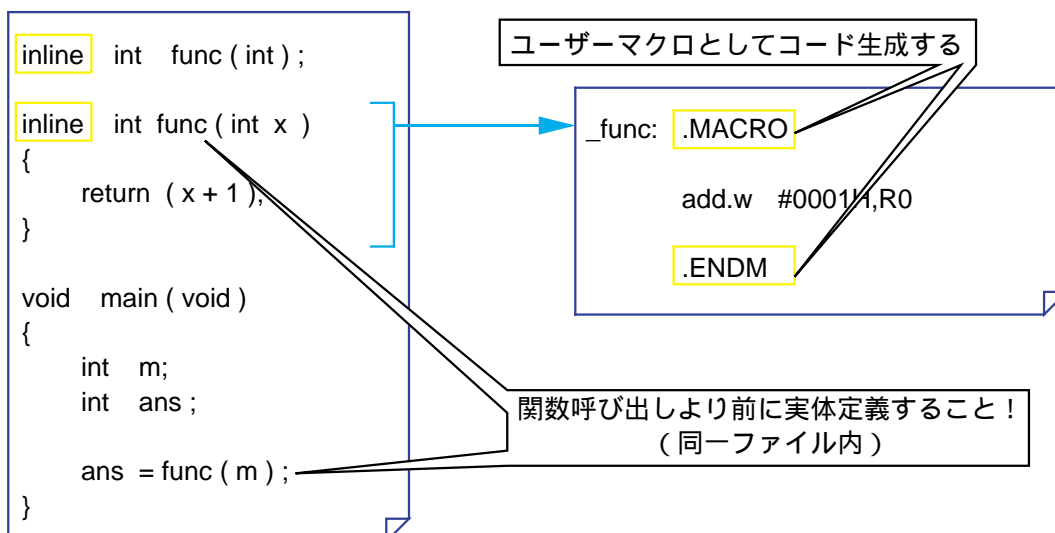


図 2.4.7 インライン記憶クラスの記述例

(注) 実行速度は上がりますが、関数を呼び出すところすべてにコードを生成するため、呼び出し頻度が高い関数の場合はROM効率は悪くなります。

2.4.2 C 言語からのアセンブリ言語の呼び出し

この項では、アセンブリ言語のサブルーチンをC言語の関数として呼び出す具体的な記述方法について説明します。

アセンブリ言語へ引数を渡す (#pragma PARAMETER)

```
#pragma PARAMETER 関数名 (レジスタ名 ; ...)
```

上のように記述した関数は通常の引き渡し規則によらず、指定されたレジスタに引数をセットして呼び出し動作を行います。

この機能を利用すると引数渡しのスタック操作がないため、関数呼び出しの際のオーバーヘッドが小さくなります。ただし、この機能を使用するときには次のような注意が必要です。

- (1) "#pragma PARAMETER" を記述する前に指定関数のプロトタイプ宣言を行ってください。
- (2) プロトタイプ宣言において以下の項目を守ってください。
 - ・関数の引数は必ず char 型、int 型、long 型、float 型、near ポインタ型、far ポインタ型にしてください。構造体型、共用体型、double 型は宣言できません。
 - ・関数の戻り値として構造体、共用体は宣言できません。
 - ・レジスタのサイズと引数のサイズを合わせてください。
 - ・レジスタ名の大文字 / 小文字は区別しません。
 - ・本 #pragma で指定した関数の実体定義を C 言語で行った場合はエラーとなります。

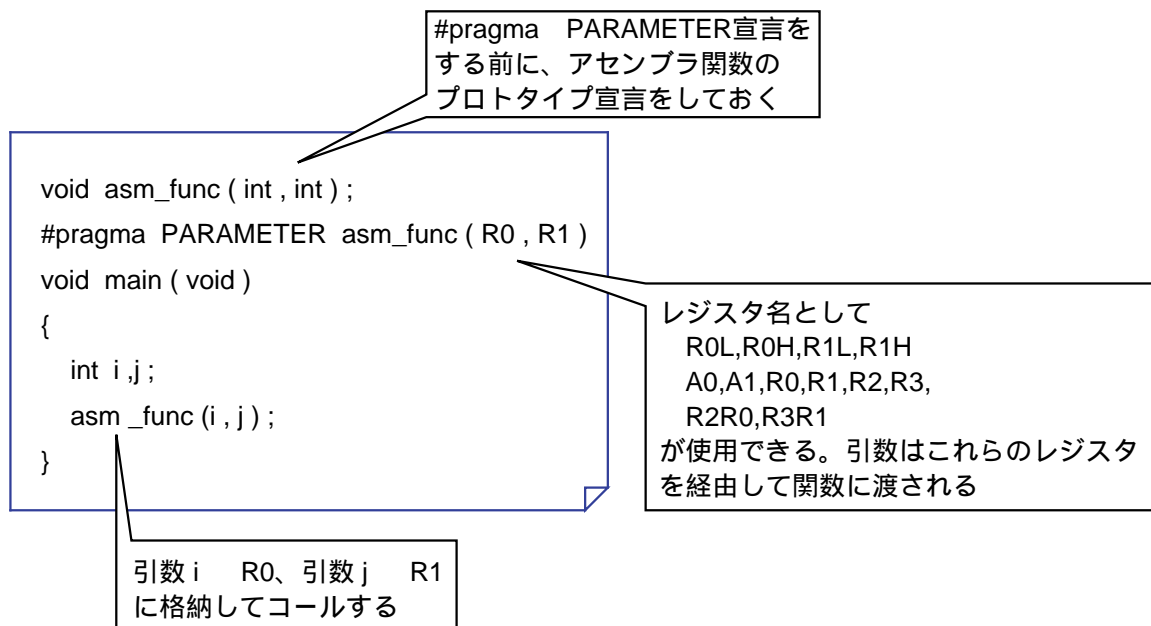


図 2.4.8 "#pragma PARAMETER" の記述例

アセンブリ言語のサブルーチンを呼び出す

C 言語プログラムからアセンブリ言語のサブルーチン (アセンブリ言語関数) を呼び出すときは、以下の規則に従ってください。

- (1) サブルーチンは C 言語プログラムとは別のファイルに記述します。
- (2) サブルーチン名はシンボル変換規則に従ってください。
- (3) 呼び出す側の C 言語プログラム中で、サブルーチン (アセンブリ言語関数) のプロトタイプ宣言を行います。このとき、記憶クラス指定子 "extern" で外部参照宣言を行います。
- (4) サブルーチン (アセンブリ言語関数) 内で FB 相対アドレッシングを使用して自動変数にアクセスする場合は ".fb" 宣言を行い、値は 0 を設定してください。
- (5) 通常、サブルーチン (アセンブリ言語関数) 内では NC308 が占有して使用する B フラグ、U フラグ、SB レジスタ、FB レジスタの値の変更は行わないでください。変更が必要な場合は、関数の入口でスタックに値を退避し、関数の出口でスタックから値を復帰するようにしてください。

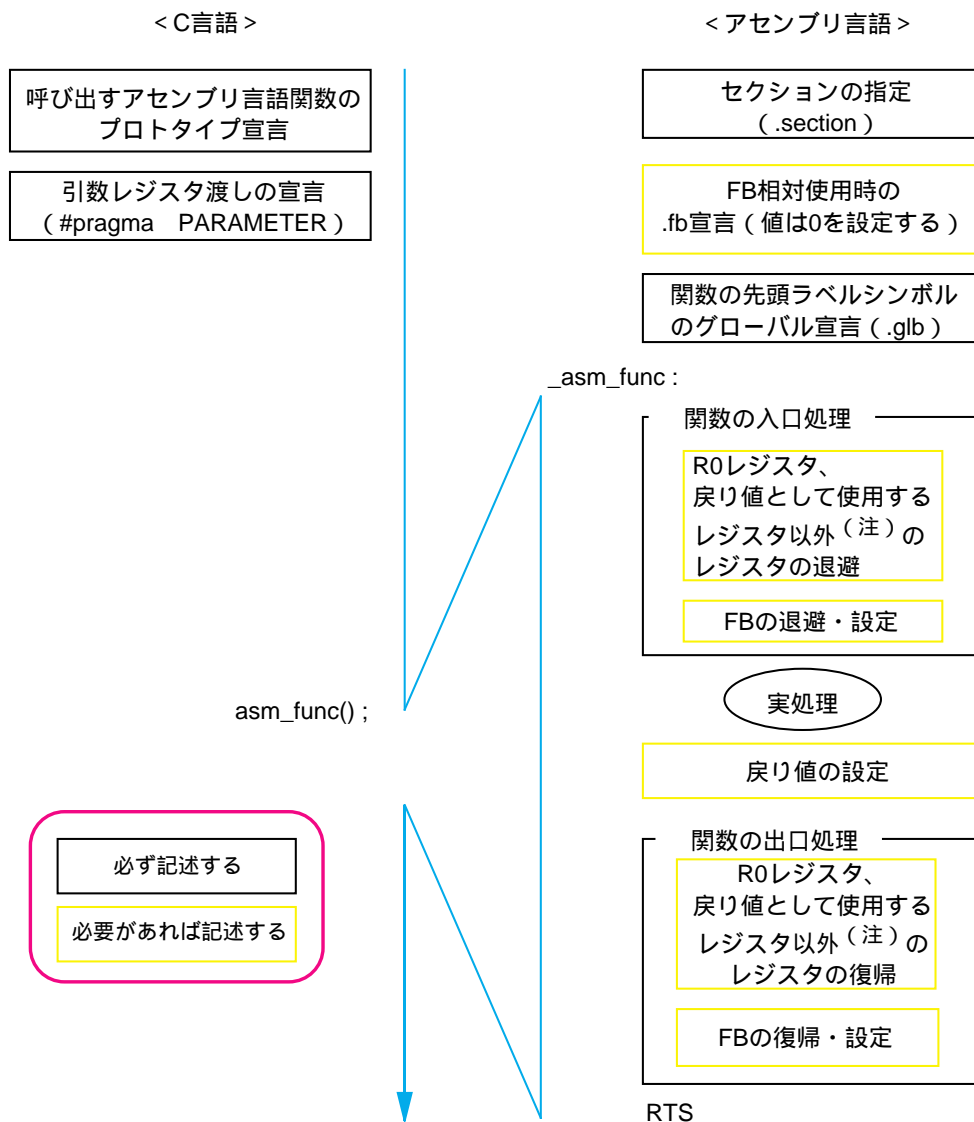


図 2.4.9 サブルーチンの呼び出し

(注) R0 レジスタおよび、戻り値に使用するレジスタの退避については、退避が必要な場合は NC308 が関数の呼び出し側 (Caller 側) で退避するためのコードをします。このため、R0 レジスタおよび、戻り値に使用するレジスタは退避 / 復帰する必要がありません。

コラム NC308 と NC30 のコーリングコンベンションの違い

NC30 では関数呼び出し時のレジスタ退避を関数の呼び出し側 (Caller) で行いますが、NC308 では呼び出された関数側 (Called) で行います。そのため、C 言語関数からアセンブリ言語関数を呼び出すときは、C 言語関数内で使用しているレジスタの内容を破壊しないように、アセンブリ言語関数の入口で関数内で破壊されるレジスタ (R0 レジスタ、戻り値に使用するレジスタ以外) の退避 ("PUSHM" 命令を使用) を行い、アセンブリ言語関数の出口でレジスタ (R0 レジスタ、戻り値に使用するレジスタ以外) 復帰 ("POPM" 命令を使用) を行う処理を必ず記述してください。

NC30 とコーリングコンベンションを同じにするにはスイッチ "/C" を追加した "#pragma PARAMETER /C" を記述して関数呼び出し時にレジスタ退避を関数の呼び出し側 (Caller) で行うようにします。"#pragma SPECIAL" にも同様のスイッチ "/C" を用意しており、NC30 のコーリングシーケンスと互換性をとることができます。

図 2.4.10 に #pragma PARAMETER /C の記述例を示します。

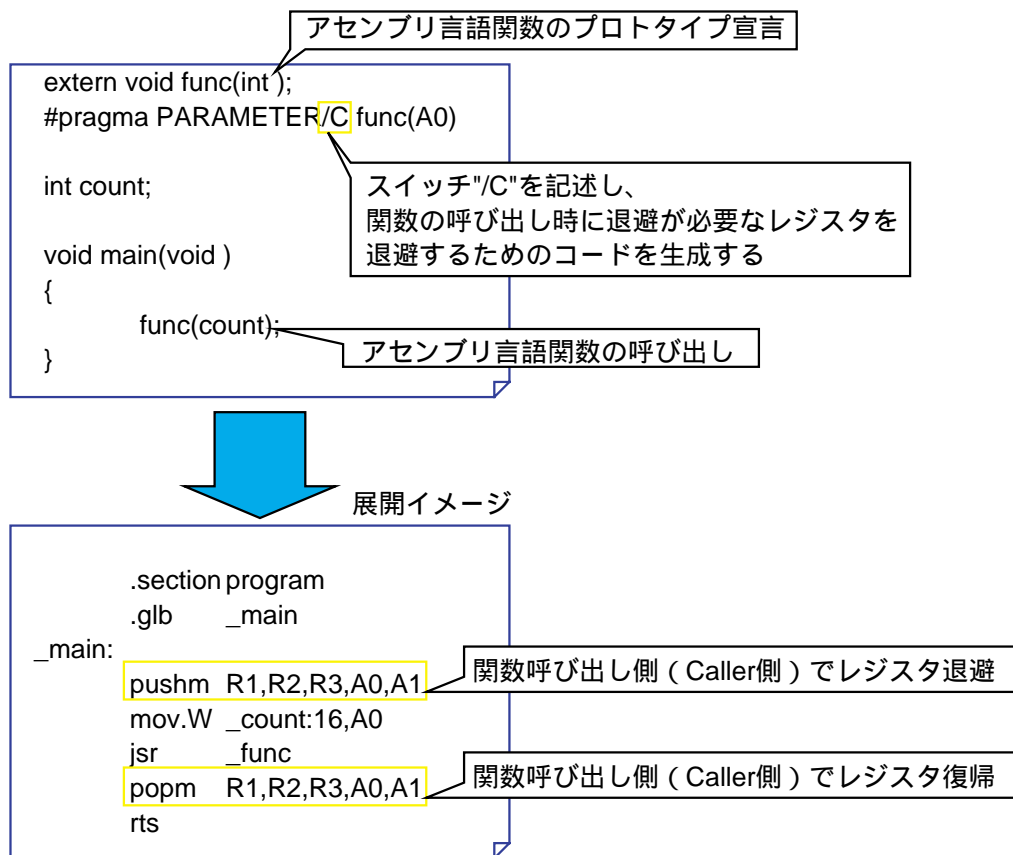


図 2.4.10 #pragma PARAMETER /C の記述例

例 2.4.1 サブルーチン呼び出す

カウントアップした結果を LED に表示します。LED 表示部をアセンブリ言語で、カウントアップ部を C 言語で作成しリンクします。

< カウントアップ部 >

```

/* プロトタイプ宣言 */
void led (int);
#pragma PARAMETER led (A0)

/* SB相対アドレッシング使用変数指定 */
#pragma SBDATA counter

static int counter;

void main (void)
{
    if ( counter < 9 ) {
        counter ++;
    }
    else {
        counter = 0;
    }

    led ( counter );
}
        
```

< LED表示部 >

```

P7 .equ 03c1h

.section program
.glob led

_led :
    mov.b    table[a0], P7
    rts
;-----
; LED表示用データテーブル
;-----

.section rom_FE, ROMDATA
table :
    .byte   0c0h, 0f9h, 0a4h, 0b0h, 099h
    .byte   092h, 082h, 0f8h, 080h, 090h

.end
        
```

アセンブリ言語関数の引数レジスタ渡しの宣言

配置領域の設定

サブルーチン名をグローバル宣言

#pragma PARAMETER 指定のためサブルーチン名は関数名の先頭に「_」を付けて変換

アセンブリ言語関数 "led()" を呼び出す

例 2.4.1 サブルーチン呼び出す

スペシャルページサブルーチンを呼び出す(#pragma SPECIAL[/C])

```
#pragma SPECIAL[/C] 呼び出し番号 関数名 ( )
```

上のように記述した関数の呼び出しは jsrs 命令 (スペシャルページサブルーチン呼び出し命令) に展開されます。

この機能を利用すると、呼び出し回数の多い関数の呼び出し時の命令バイト数を削減することができます(注)。この機能を使用するときには次のような注意が必要です。

- (1)"pragma SPECIAL" を記述する前に必ず関数のプロトタイプ宣言を行ってください。
- (2)"#pragma SPECIAL" で宣言した関数は program_S セクションに配置されます。したがってスタートアッププログラムの中で program_S セクションを F00000H から FFFFFFFH 番地の領域に配置してください。
- (3)呼び出し番号は 18 から 255 までの 10 進数のみ指定可能です。
- (4)"#pragma SPECIAL" で宣言した関数の先頭アドレスには "_SPECIAL_呼び出し番号:" がラベルとして出力されます。したがって、スペシャルページベクタテーブルにこの関数の出力ラベル名を設定する必要があります。

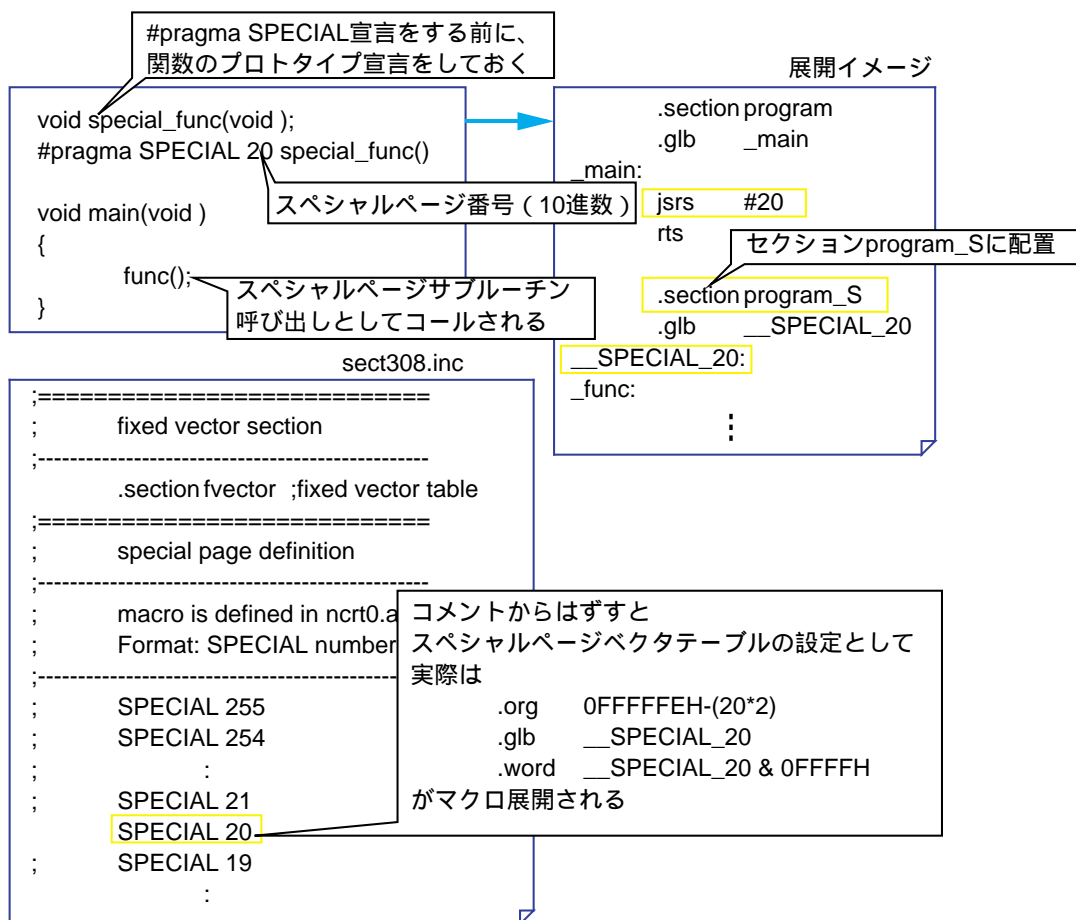


図 2.4.11 スペシャルページサブルーチンを呼び出す

(注) jsrs 命令を使用する場合は命令バイト数として 2 バイト必要となります。さらにスペシャルページベクタテーブルとして 2 バイト必要になるため、3 回以上呼び出される関数に対して "#pragma SPECIAL" 宣言するのが有効です。

間接アドレッシングでサブルーチンを呼び出す

通常、C 言語からアセンブリ言語のサブルーチンを呼び出すときの命令は "jsr" が生成されます。"jsri" による間接アドレッシングを使用する場合は「関数ポインタ」を使用します。ただし、関数ポインタを使用した場合 "#pragma PARAMETER" による引数渡しのレジスタ指定はできません。記述例を図 2.4.12 に示します。

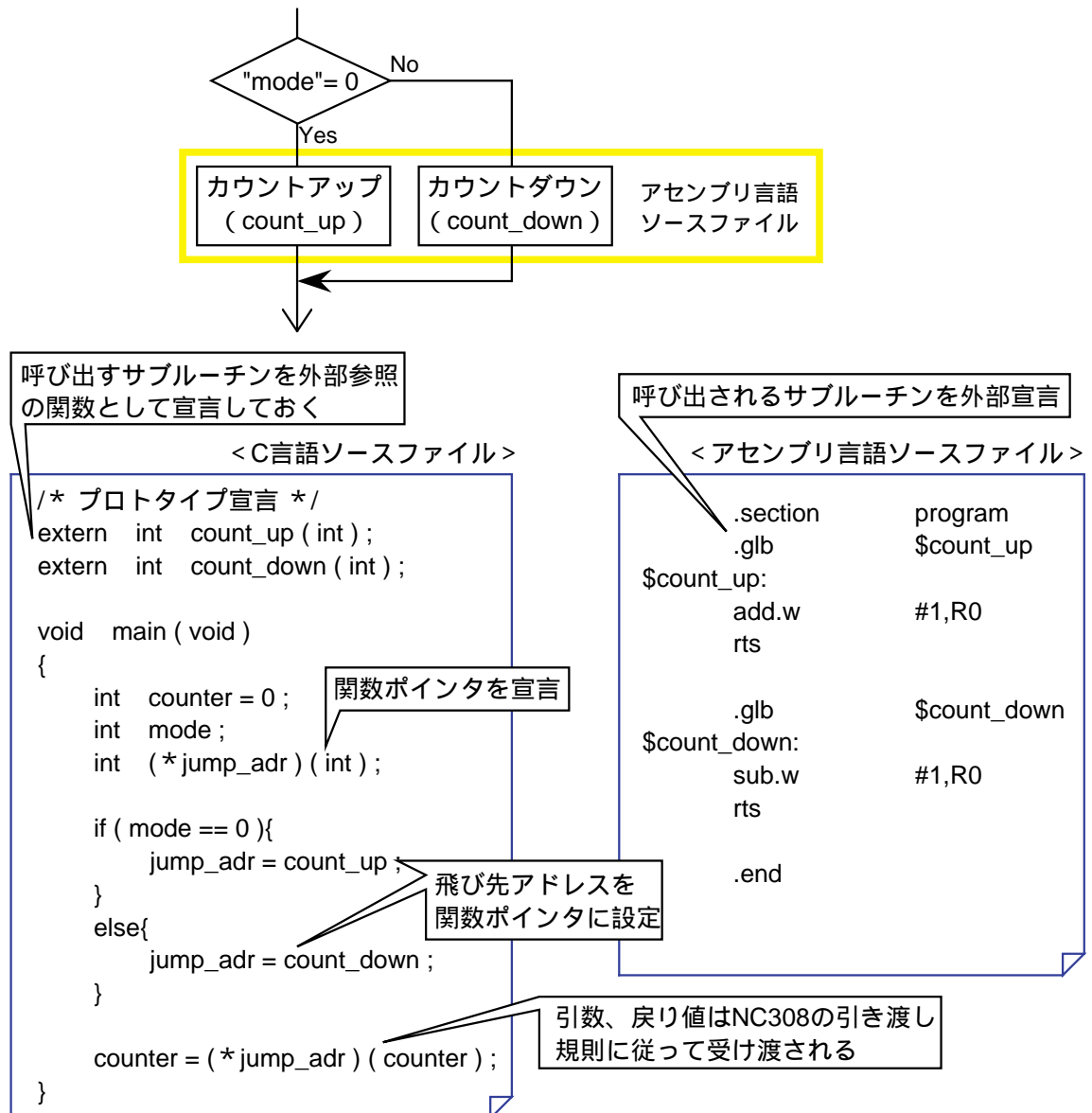
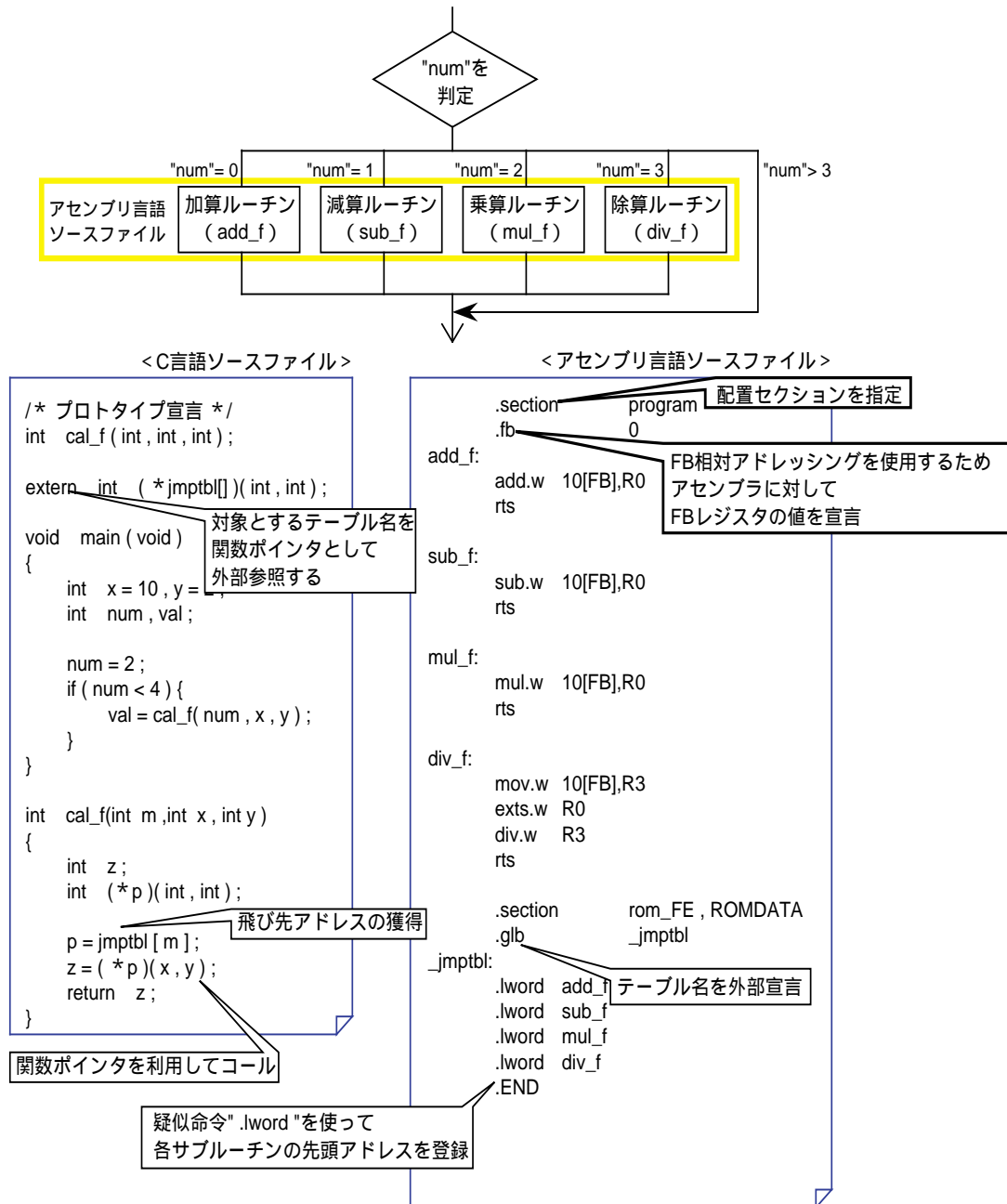


図 2.4.12 間接アドレッシングでサブルーチンを呼び出す

例 2.4.2 テーブルジャンプでサブルーチンを呼び出す

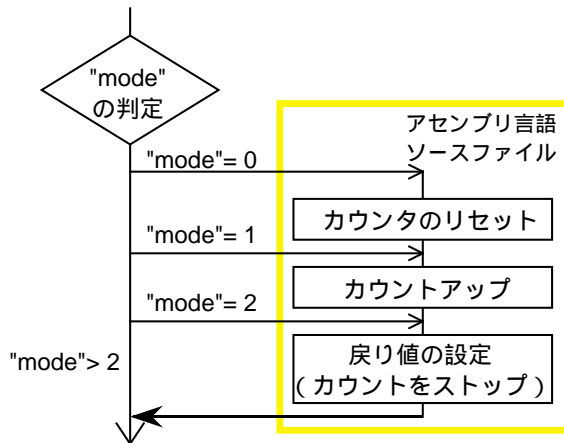
"num" の値に従って、異なるサブルーチンを C 言語プログラムから呼び出します。このように多数の分岐がある場合にテーブルジャンプを利用すると、どのサブルーチンでも同じ処理時間で呼び出すことができます。ただし "#pragma PARAMETER" による引数渡しのレジスタ指定はできません。



例 2.4.2 テーブルジャンプでサブルーチンを呼び出す

例 2.4.3 少し変わったテーブルジャンプの使い方

サブルーチン内のラベルをジャンプテーブルに登録すると、サブルーチンの先頭アドレスをモードによって変えることができます。1つのサブルーチンで複数の処理を実現できるのでROM容量の節約につながります。



<C言語ソースファイル>

```

/* プロトタイプ宣言 */
int clock ( int , int );

extern int ( * clock_mode [ ] ) ( int );

void main ( void )
{
    int mode ;
    int counter = 0 ;

    mode = 2 ;
    if ( mode < 3 ) {
        counter = clock( mode , counter ) ;
    }
}

int clock( int m , int x )
{
    int z ;
    int ( * p ) ( int ) ;

    p = clock_mode [ m ] ;
    z = ( * p ) ( x ) ;
    return z ;
}
  
```

<アセンブリ言語ソースファイル>

```

.section      program
reset:
    mov.w    #0FFFFH,R0

count:
    add.w    #1,R0

stop:
    rts

.section      rom_FE,ROMDATA
.glb
_clock_mode:
.lword    reset
.lword    count
.lword    stop
.END
  
```

サブルーチン内のラベルを
ジャンプテーブルに登録

例 2.4.3 少し変わったテーブルジャンプの使い方

2.4.3 アセンブリ言語からの C 言語の呼び出し

この項では、アセンブリ言語プログラムから C 言語の関数を呼び出す方法を説明します。

C 言語関数の呼び出し

アセンブリ言語から C 言語関数を呼び出すときには以下の規則に従ってください。

- (1) 呼び出すサブルーチンのラベルは NC308 のシンボル変換規則に従ってください。
- (2) C 言語関数はアセンブリ言語のプログラムとは別のファイルに記述してください。
- (3) アセンブリ言語のファイルでは C 言語関数を呼び出す前に AS308 の疑似命令 ".glb" で外部参照宣言をしてください。
- (4) C 言語関数は関数の入り口処理で R0 レジスタ、および戻り値に使用するレジスタの退避を行いません。このため、C 言語関数を呼び出す前に必ず R0 レジスタ、および戻り値に使用するレジスタの退避を行ってください。

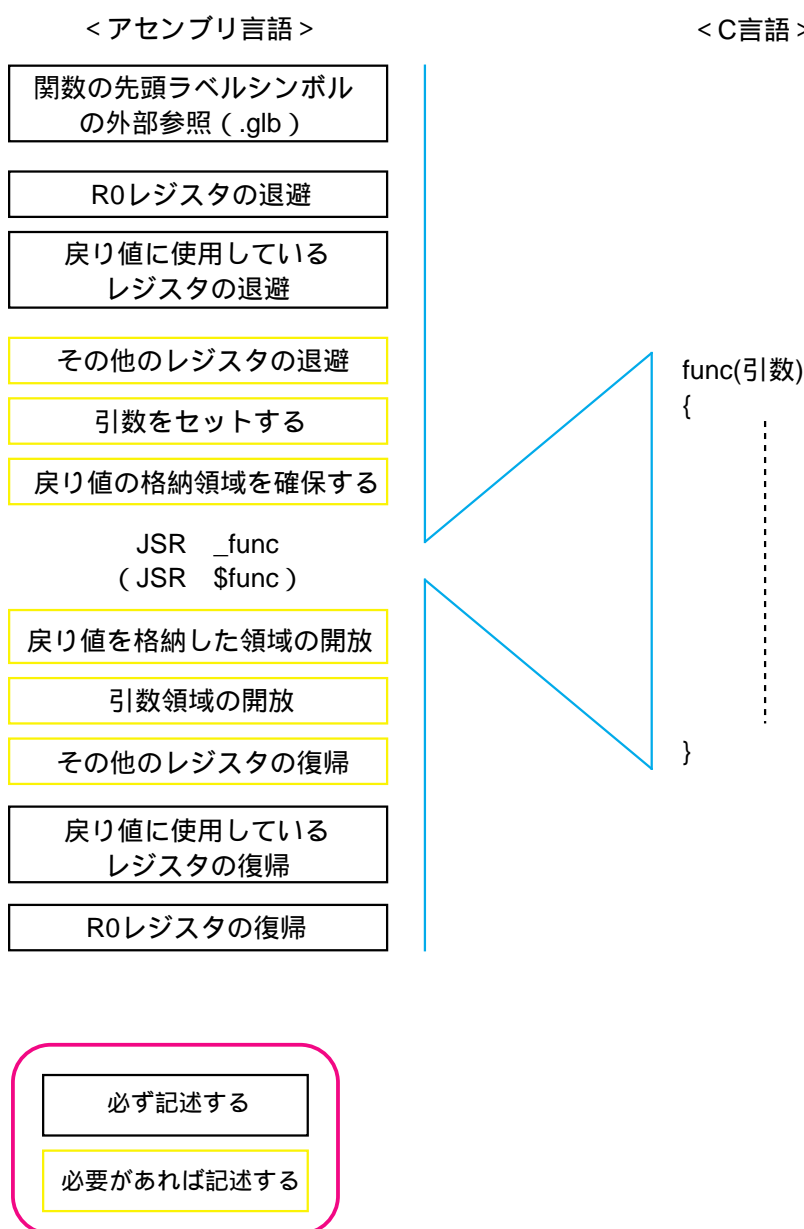


図 2.4.13 C 言語関数の呼び出し

2.5 割り込み処理

2.5.1 割り込み処理関数の記述

NC308 では割り込み処理を C 言語関数として記述することができます。手順は次の 4 つです^(注)。

- (1) 割り込み処理関数の記述
- (2) 割り込みベクタテーブルへの登録
- (3) 割り込み許可フラグ (Iフラグ) の設定
 - ・インラインアセンブル機能により行います。
- (4) 使用する割り込みの割り込み優先レベルの設定
 - ・アセンブラ使用時と同様に割り込みを許可する前に設定します。

この項では、割り込み処理の種類別に関数の記述方法を説明します。

ハードウェア割り込みの記述 (#pragma INTERRUPT 割り込み関数名)

上のように宣言すると、指定した関数の入口と出口において、通常の間数の手続き以外に M16C/80 シリーズの全レジスタの退避・復帰と reit 命令を生成します。割り込み処理関数の型は、引数 / 戻り値ともに void 型のみ有効です。それ以外の型を宣言した場合は、コンパイル時にワーニングを出力します。

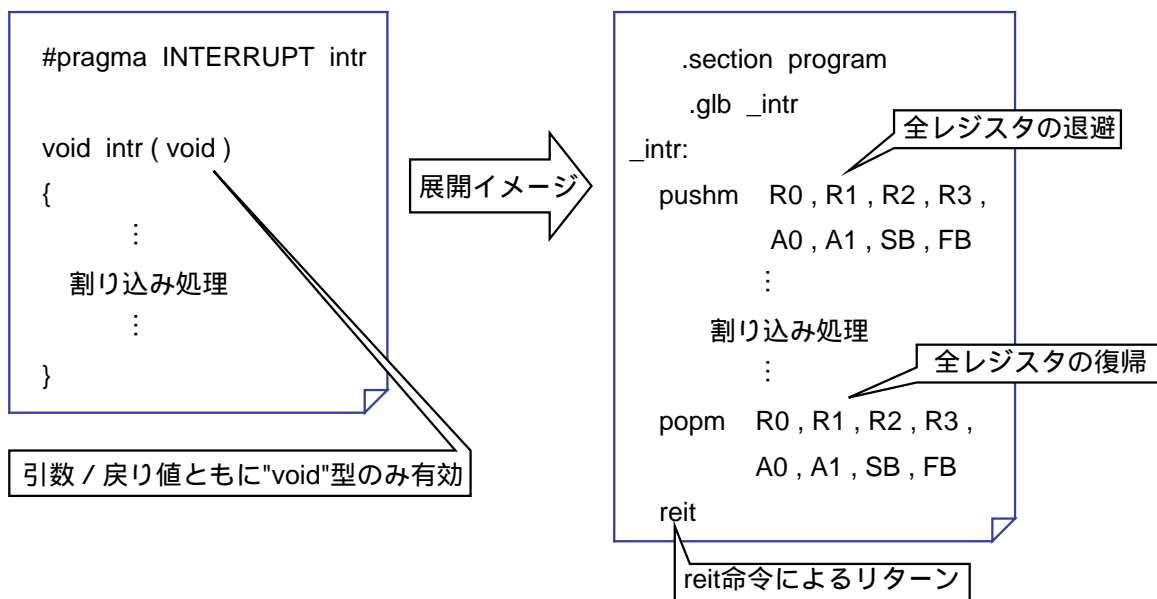


図 2.5.1 割り込み処理関数の展開イメージ

(注) 詳しくは「2.5.5 割り込み処理関数の記述例」を参照してください。

レジスタバンクを使用した割り込みの記述 (#pragma INTERRUPT/B)

M16C/80シリーズではレジスタバンクを切り換えることによって、レジスタの内容などを保護しつつ割り込み処理が起動されるまでの時間を短縮することができます。この機能を使用したい場合は以下のように記述します。

#pragma INTERRUPT/B 割り込み関数名

上のように記述するとレジスタ退避 / 復帰の命令の代わりに、レジスタバンクを切り換える命令が生成されます。ただし、M16C/80シリーズのレジスタバンクはレジスタバンク 0, 1 の 2 セットですので、指定できる割り込みはひとつです^(注)。短時間での起動が必要な割り込みに対してこの機能を使用するようにしてください。

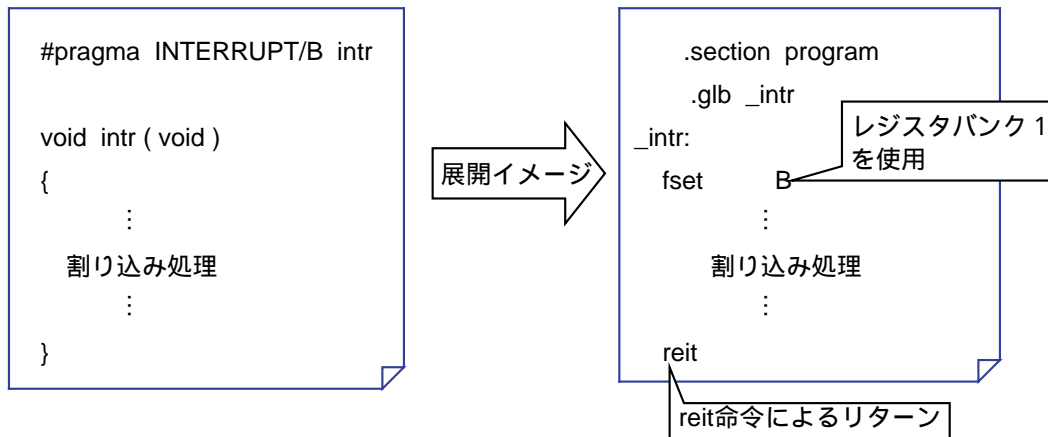


図 2.5.2 レジスタバンクを使用した割り込み処理関数の展開イメージ

(注) 多重割り込みを使用しない場合はすべての割り込みで使用できます。

多重割り込みを許可にする割り込みの記述 (#pragma INTERRUPT/E)

M16C/80 シリーズでは、割り込み要求が受け付けられると割り込み許可フラグ (Iフラグ) が "0" になり割り込み禁止の状態となります。そのため、割り込み処理関数の入り口 (割り込み処理関数に入った直後) で割り込み許可フラグ (Iフラグ) を "1" にして多重割り込みを許可の状態にすると、割り込みの応答性を向上させることができます。この機能を使用したい場合は以下のように記述します。

```
#pragma INTERRUPT/E 割り込み関数名
```

上のように記述すると割り込み処理関数の入り口 (割り込み処理関数に入った直後) で割り込み許可フラグ (Iフラグ) を "1" にする命令が生成されます。ただし、割り込み処理関数の途中で多重割り込み許可にしたい場合は、"#pragma INTERRUPT" 宣言を行い、割り込み処理関数の途中で asm()関数を使用して割り込み許可フラグ (Iフラグ) を "1" に設定してください。

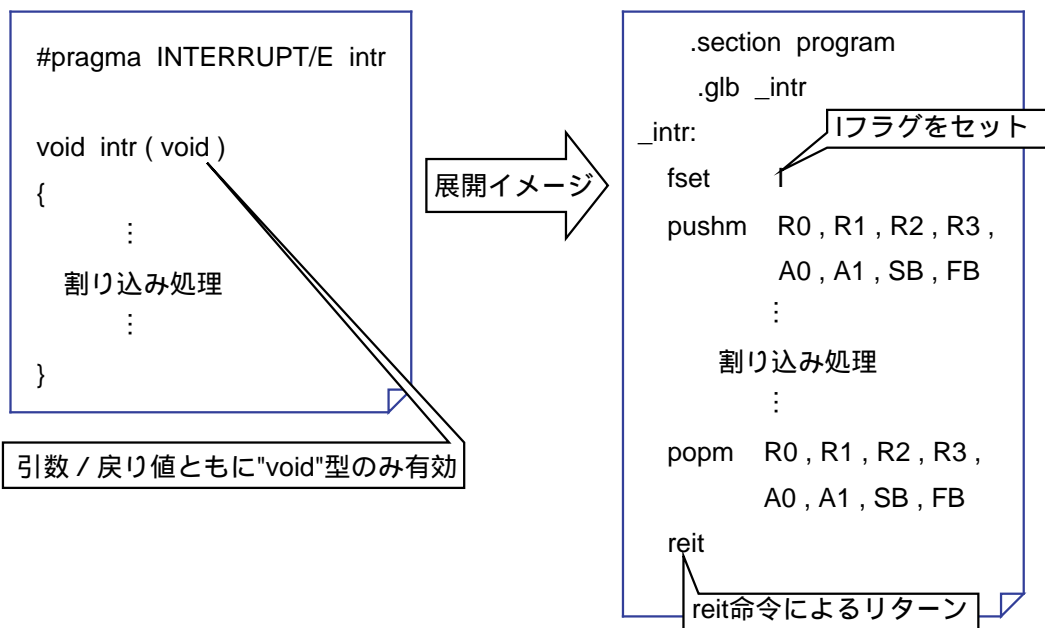


図 2.5.3 多重割り込みを許可にする割り込み処理関数の展開イメージ

2.5.2 高速割り込み処理関数の記述

NC308では割り込みの応答を5サイクルで、また割り込みの復帰を3サイクルで実行できる高速割り込み処理をC言語関数として記述することができます。ただし、高速割り込みに設定できる割り込みは割り込み優先レベル7の1つの割り込みだけです。手順は次の5つです^(注)。

- (1)高速割り込み処理関数の記述
- (2)使用する高速割り込みの割り込み優先レベルの設定
 - ・アセンブラ使用時と同様に割り込みを許可する前に設定します。
- (3)高速割り込み指定ビットの設定
- (4)ベクタレジスタ(VCT)の設定
- (5)割り込み許可フラグ(Iフラグ)の設定
 - ・インラインアセンブル機能により行います。

この項では、高速割り込み処理の種類別に関数の記述方法を説明します。

高速ハードウェア割り込みの記述 (#pragma INTERRUPT/F)

```
#pragma INTERRUPT/F 割り込み関数名
```

上のように宣言すると、指定した関数の入口と出口において、通常の間数の手続き以外にM16C/80シリーズの全レジスタの退避・復帰と高速割り込みルーチンからの復帰命令freitを生成します。割り込み処理関数の型は、引数/戻り値ともにvoid型のみ有効です。それ以外の型を宣言した場合は、コンパイル時にワーニングを出力します。

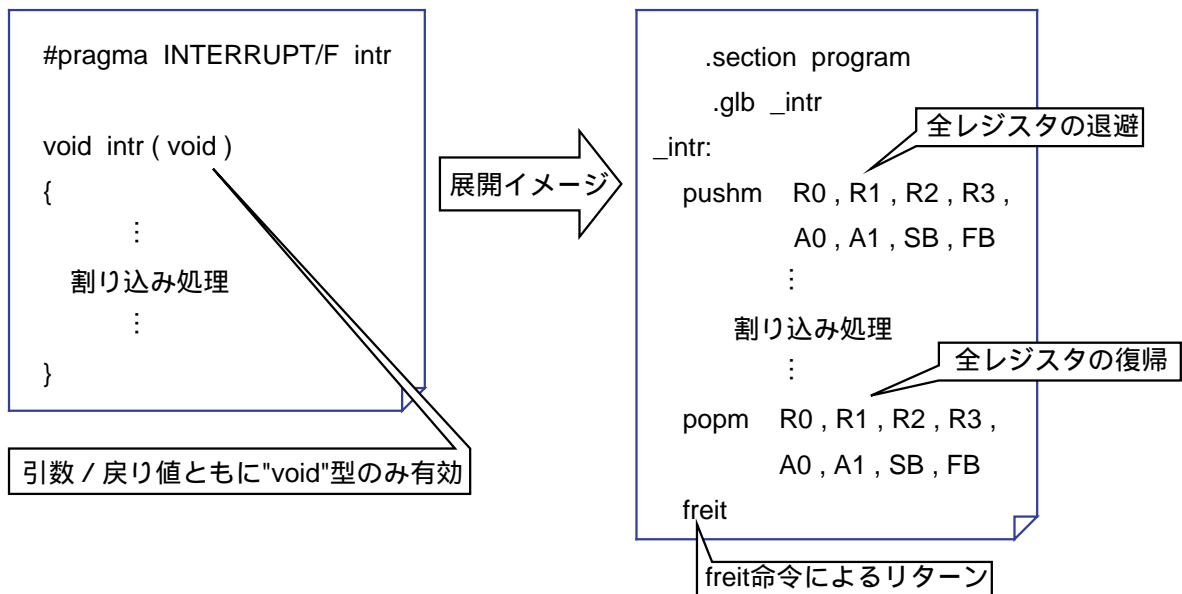


図 2.5.4 高速割り込み処理関数の展開イメージ

(注) 詳しくは「2.5.5 割り込み処理関数の記述例」を参照してください。

レジスタバンクを使用した高速割り込みの記述 (#pragma INTERRUPT/F/B)

M16C/80シリーズではレジスタバンクを切り換えることによって、レジスタの内容などを保護しつつ高速割り込み処理が起動されるまでの時間を短縮することができます。この機能を使用したい場合は以下のように記述します。

```
#pragma INTERRUPT/F/B 割り込み関数名
```

上のように記述するとレジスタ退避 / 復帰の命令の代わりに、レジスタバンクを切り換える命令が生成されます。ただし、M16C/80シリーズのレジスタバンクはレジスタバンク 0, 1 の 2 セットですので、指定できる割り込みはひとつです^(注)。最も短時間での起動が必要な割り込みに対してこの機能を使用するようにしてください。

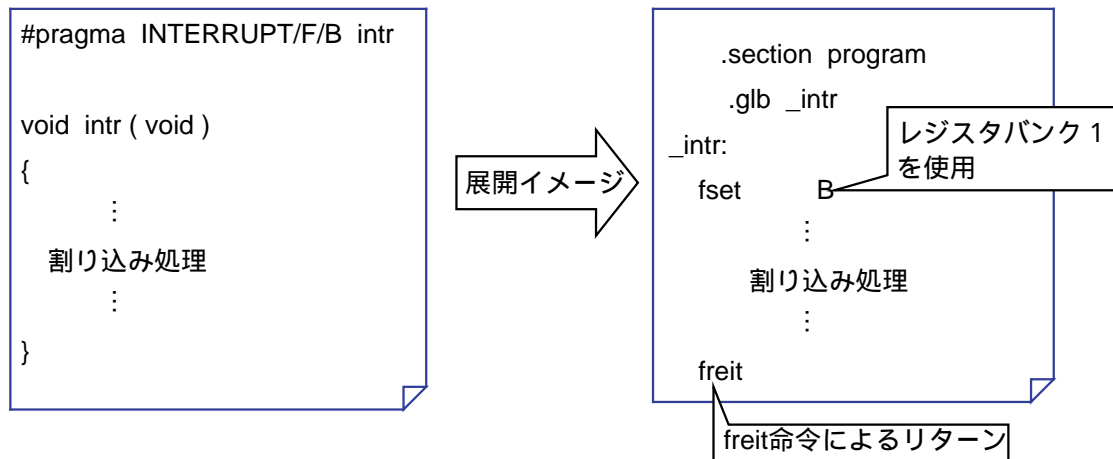


図 2.5.5 レジスタバンクを使用した高速割り込み処理関数の展開イメージ

(注) 多重割り込みを使用しない場合はすべての割り込みで使用できます。

2.5.3 ソフトウェア割り込み (INT 命令) 処理関数の記述

アセンブリ言語関数を呼び出すソフトウェア割り込みの記述 (#pragma INTCALL)

M16C/80 シリーズのソフトウェア割り込み (INT 命令) を使用する場合は "#pragma INTCALL" を記述します。この機能を使用するとデバッグ時に擬似的に割り込みを発生させることができます。また、INT 命令の必要バイト数が 2 バイトのため、ROM 効率が向上します^(注)。

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合と C 言語で記述されている場合で記述方法が異なります。

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合は以下のように記述します。

```
#pragma INTCALL ソフトウェア割り込み番号 アセンブリ言語関数名(レジスタ名, レジスタ, ...)
```

ソフトウェア割り込みで呼び出す関数の実体がアセンブリ言語で記述されている場合は、レジスタを経由して引数を渡すことができます。また構造体・共用体以外の戻り値を受け取ることができます。

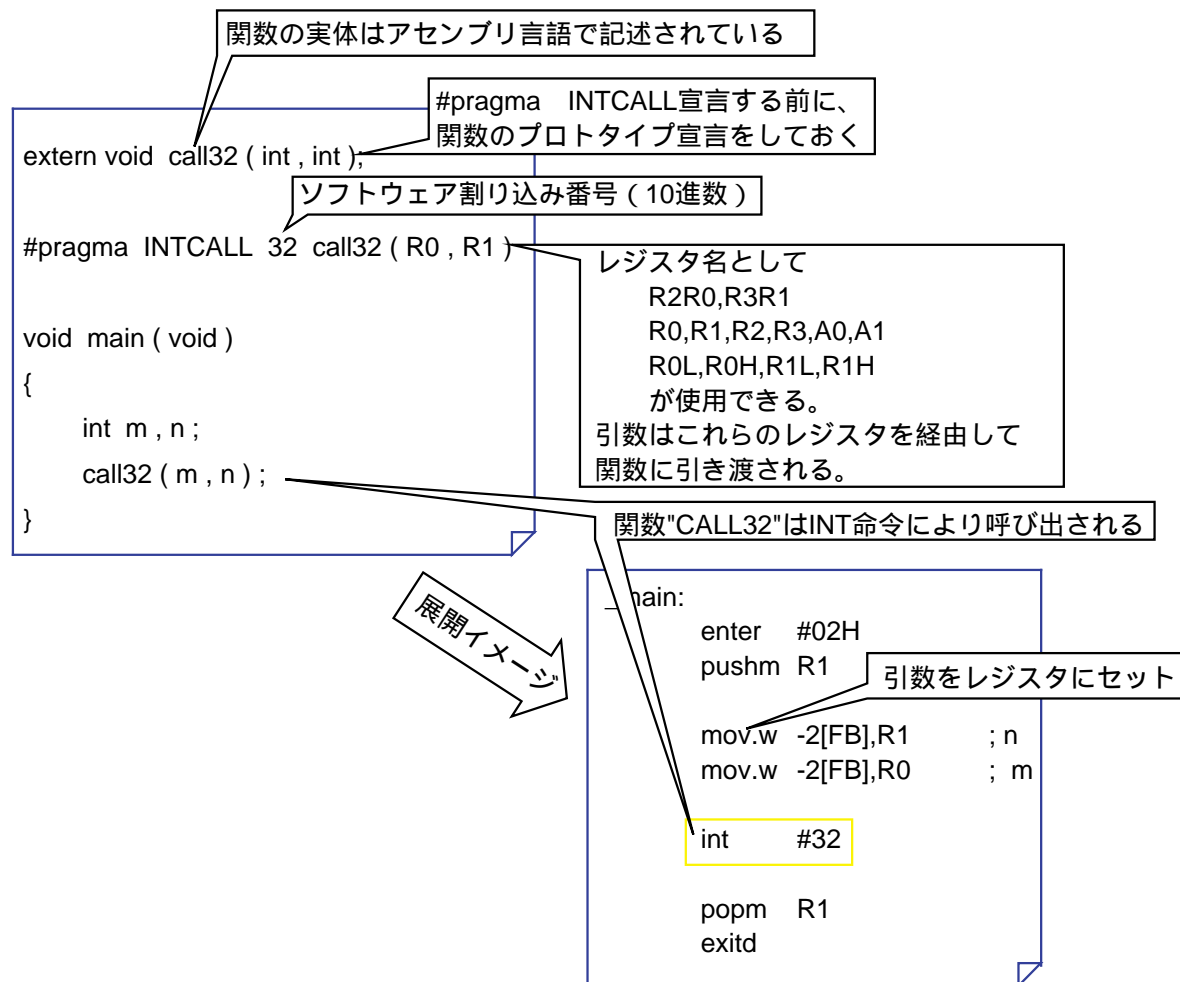


図 2.5.6 アセンブリ言語関数を呼び出す "#pragma INTCALL" 記述例

(注) INT 命令の必要サイクル数 1 2 サイクル + 割り込みシーケンス実行時間が必要となるため、関数が実行されるまでの時間はかかります。

C 言語関数を呼び出すソフトウェア割り込みの記述 (#pragma INTCALL)

ソフトウェア割り込み (INT 命令) で呼び出す関数の実体が C 言語で記述されている場合は以下のように記述します。

```
#pragma INTCALL ソフトウェア割り込み番号 C言語関数名()
```

ソフトウェア割り込みで呼び出す関数の実体が C 言語で記述されている場合は、引数の引き渡し規則によって、すべての引数がレジスタ渡しとなる関数のみ指定することができます。"#pragma INTCALL" 宣言を行う関数の引数は記述できません。また構造体・共用体以外の戻り値を受け取ることができます。

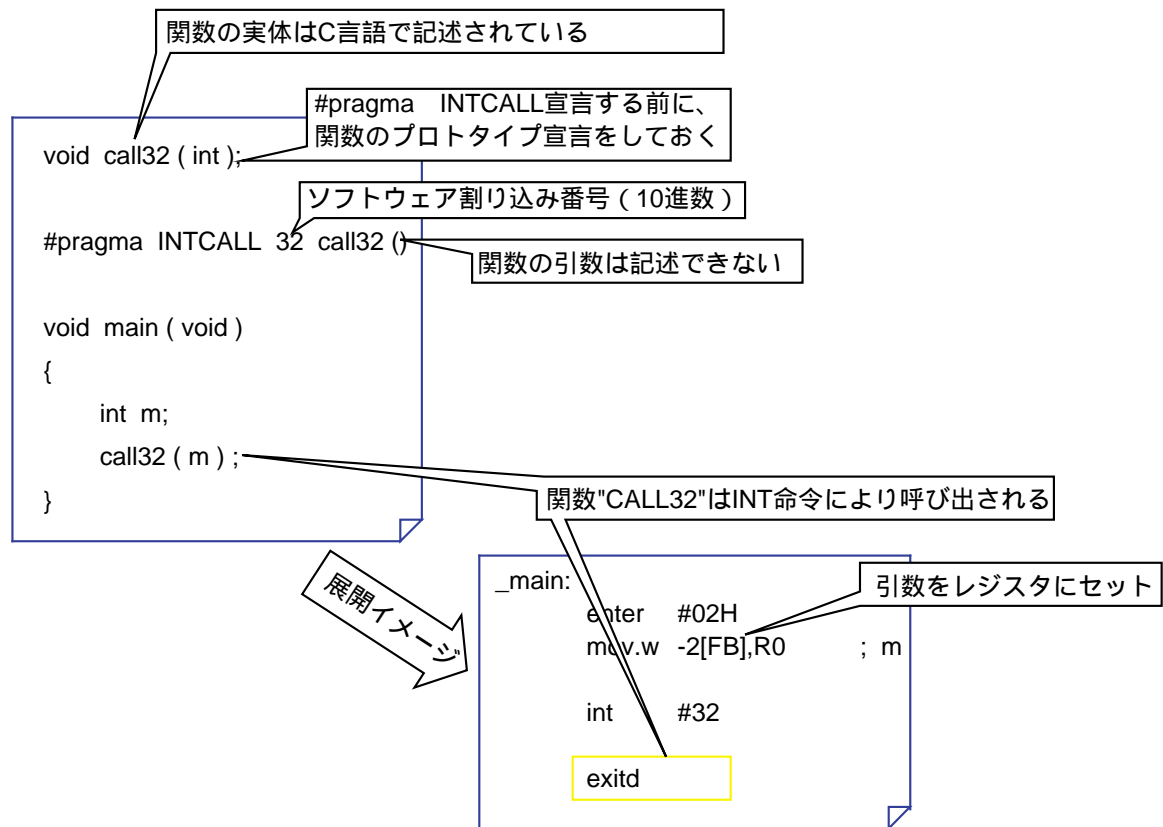


図 2.5.7 C 言語関数を呼び出す "#pragma INTCALL" 記述例

2.5.4 割り込み処理関数を登録する

割り込みを正常に使用するためには、割り込み処理関数を記述するとともに割り込みベクタテーブルに登録する必要があります。

この項では、割り込みベクタテーブルへの登録方法を説明します。

割り込みベクタテーブルへの登録

割り込み処理関数を記述する場合、サンプルスタートアッププログラム "sect308.inc" 中の割り込みベクタテーブルを変更することにより、割り込み処理関数を登録します。

割り込みベクタテーブルの変更は以下の手順で行います。

- (1) 割り込み処理関数名を指示命令 ".glb" で外部参照宣言する。
- (2) 使用する割り込みのダミー関数名 "dummy_int" を、割り込み処理関数名に変更する。

```

;-----
; variable vector section
;-----
.section      vector          ;variable vector table
.org         VECTOR_ADR
;
.word       dummy_int       ; vector (BRK)
.org       ( VECTOR_ADR + 32 )
.word       dummy_int       ; DMA0 (software int 8)
.word       dummy_int       ; DMA1 (software int 9)
.word       dummy_int       ; DMA2 (software int 10)
.word       dummy_int       ; DMA3 (software int 11)
.glb       _ta0
.word       _ta0           ; TIMER A0 (software int 12)
.word       dummy_int       ; TIMER A1 (software int 13)
.word       dummy_int       ; TIMER A2 (software int 14)
.word       dummy_int       ; TIMER A3 (software int 15)
.word       dummy_int       ; TIMER A4 (software int 16)
          :

```

TA0割り込みに関数"ta0()"を登録する

図 2.5.8 割り込みベクタテーブル ("sect308.inc")

2.5.5 割り込み処理関数の記述例

この項では、INT0 割り込みが発生するたびに "counter" の内容を 0 クリアし、INT1 割り込みが発生するたびに "counter" の内容をカウントアップさせるプログラムの記述例を示します。

割り込み処理関数の記述

図 2.5.9 にソースファイルの記述例を示します。

```

/* プロトタイプ宣言 * * * * * */
void int0 ( void );
void int1 ( void );
#pragma INTERRUPT/F int0
#pragma INTERRUPT int1
/* * * * * * */

unsigned int counter ;

void int0 ( void )      /* 高速割り込み関数 */
{
    counter = 0 ;
}

void int1 ( void )      /* 割り込み関数 */
{
    if ( counter < 9 ) {

        counter ++ ;

    }
    else {
        counter = 0 ;
    }
}

void main ( void )
{

    INTOIC = 0x07;          /* 高速割り込み優先レベルの設定 */
    RLVL = 0x08;          /* 高速割り込み指定 */
    asm( " LDC  #_int0,VCT " ); /* ベクタレジスタの設定 */
    INT1IC = 0x01 ;       /* 割り込み優先レベルの設定 */

    asm ( " fset  i " );   /* 割り込み許可 */

    while (1);            /* 割り込み待ちループ */

}

```

図 2.5.9 割り込み処理関数の記述例

割り込みベクタテーブルへの登録

図 2.5.10 に割り込みベクタテーブルへの登録例を示します。

```

;-----
; variable vector section
;-----
.section          vector          ; variable vector table
.org              VECTOR_ADR
                :
.org              ( VECTOR_ADR + 32 )
.lword           dummy_int        ; DMA0 (software int 8)
.lword           dummy_int        ; DMA1 (software int 9)
.lword           dummy_int        ; DMA2 (software int 10)
.lword           dummy_int        ; DMA3 (software int 11)
.lword           dummy_int        ; TIMER A0 (software int 12)
.lword           dummy_int        ; TIMER A1 (software int 13)
.lword           dummy_int        ; TIMER A2 (software int 14)
.lword           dummy_int        ; TIMER A3 (software int 15)
.lword           dummy_int        ; TIMER A4 (software int 16)
.lword           dummy_int        ; uart0 trance (software int17)
.lword           dummy_int        ; uart0 receive (software int18)
.lword           dummy_int        ; uart1 trance (software int19)
.lword           dummy_int        ; uart1 receive (software int 20)
.lword           dummy_int        ;TIMER B0 (software int 21)
.lword           dummy_int        ;TIMER B1 (software int 22)
.lword           dummy_int        ;TIMER B2 (software int 23)
.lword           dummy_int        ;TIMER B3 (software int 24)
.lword           dummy_int        ;TIMER B4 (software int 25)
.lword           dummy_int        ; INT5 (software int 26)
.lword           dummy_int        ; INT4 (software int 27)
.lword           dummy_int        ; INT3 (software int 28)
.lword           dummy_int        ; INT2 (software int 29)
.glb             _int1
.lword           _int1             ; INT1 (software int 30)
.lword           dummy_int        ; INT0 (software int 31)
.lword           dummy_int        ; TIMER B5 (software int 32)
                :

```

図 2.5.10 割り込みベクタテーブルへの登録例

第 3 章

リアルタイム OS(MR308)の使用

- 3.1 リアルタイム OS の基礎
- 3.2 システムコールの利用法
- 3.3 MR308 を用いた開発手順
- 3.4 NC308 を用いた MR308 の組み込み

この章では、M16C/80 シリーズ用リアルタイム OS (MR308) の概略機能について説明するとともに、NC308 を用いてリアルタイム OS を使用するための注意事項について説明します。。

3.1 リアルタイム OS の基礎

3.1.1 リアルタイム OS とタスク

リアルタイム OS を用いたプログラムは、タスク単位に構成します。
この項では、リアルタイム OS (MR308) でのタスクの取り扱いについて説明します。

タスクによるプログラムの構成

タスクとは、機能、処理時間などの単位で分けられたプログラムモジュールです。1つのタスクが1関数となる場合もありますが、1つのタスクが複数の関数から構成される場合もあります。

MR308では、これらのタスク管理を、タスクごとに異なる番号「ID」を用いて行います。タスクのIDは任意に設定できます。



図 3.1.1 プログラムの複数タスクによる構成図

タスクのスタイル

タスクは、表 3.1.1 に示す 3 つのうち、いずれかのスタイルをとります。

表 3.1.1 タスクのスタイル

終了するスタイル	ある条件で終了するスタイル	無限ループスタイル
<pre>void task1 (void) { </pre>	<pre>void task2 (void) { for (;;) { if () { break ; } } }</pre>	<pre>void task3 (void) { for (;;) { </pre>

タスクの状態

タスクはすべてリアルタイムOSに管理されます。リアルタイムOSは、各タスクからの要求である「システムコール」をもとに、実行するタスクを決定します。そして、各タスクの状態もリアルタイムOSが管理します。図 3.1.2 に MR308 におけるタスクの状態について示します。

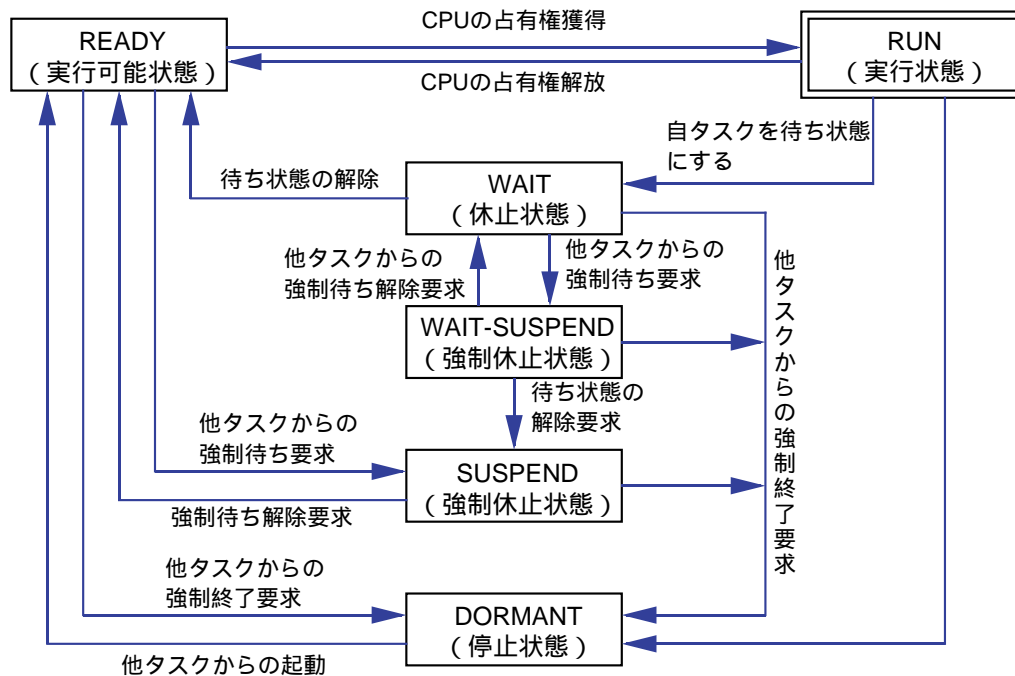


図 3.1.2 各タスク状態 (遷移図含む)

この状態の中で「RUN」、「READY」、「WAIT」の3つの状態は特に重要です。

- RUN : タスク内の処理を実行できる状態を示します。このRUN状態となるタスクは1つだけです。
- READY : RUN状態になることを待っているタスクの状態を示します。RUN状態のタスクが別の状態に変わるにより、READY状態の中の1タスクが次にRUN状態になります。
- WAIT : RUN状態のタスクがある要因によって処理を休止した状態を示します。RUN状態のタスクが休止するとリアルタイムOSは、READY状態のタスクのうち1つをRUN状態にします。

タスクの状態の切り換わり

タスクの状態が切り換わるイベントは次の3つです。

- ・ RUN タスクがシステムコールを発行した場合
- ・ 割り込みプログラム内でシステムコールを発行した場合
- ・ リアルタイム OS が持つ割り込みプログラム内でシステムコールを発行した場合

このようにシステムコールを発行することによりタスクの状態が切り替わり、RUN状態のタスクが次々と変更されていきます。そしてリアルタイム OS は、プログラム内で待ち時間が発生した場合に、待ちに関係のない別の処理を実行します。

コラム MR308 と μ ITRON 仕様^(注)

MR308 は、「 μ ITRON 仕様」に準拠したリアルタイム OS です。 μ ITRON 仕様とは、リアルタイム OS のうち、マイクロコンピュータ制御用のリアルタイム OS に関する標準規格です。主な仕様の項目を以下に示します。

- 1 . システムコールの名称の統一
- 2 . タスクの状態定義 (RUN、WAIT、READY 状態は必須)

(注) μ ITRON 仕様の著作権は、東京大学坂村健博士に属します。

3.1.2 リアルタイム OS の機能

リアルタイム OS の主な機能は「タスクのスケジューリング」、「タスクのディスパッチ」、「オブジェクトの管理」の 3 つです。

この項では、これらの機能について説明します。

タスクのスケジューリング

あるシステムの中で、READY 状態のタスクは 1 つとは限りません。しかし、RUN 状態のタスクは唯一です。そのためリアルタイム OS は、READY 状態にあるタスクの中から、次にどのタスクを RUN 状態にするのかを選択しなければなりません。この選択方式のことを「スケジューリング」といいます。スケジューリング方式はいくつかありますが、MR308 では「優先度方式」を用いています。

優先度方式：タスクにある優先度を付け（重み付け）、優先度の高いものを先に RUN 状態にします。また同じ優先度を持つタスクが存在する場合は、先に READY 状態になったタスクが優先されます。

タスクの優先度はユーザが任意に設定するものであり、リアルタイム OS が設定するものではありません。この優先度の決定がリアルタイム OS を使用する上で最も重要なポイントになります。

コンテキストとタスクコントロールブロック (TCB)

リアルタイム OS が READY 状態にあるタスクを RUN 状態にすることを「ディスパッチ」といいます。このディスパッチを行う場合、RUN 状態のタスクは中断することになります。

そのためタスクの資源（各レジスタの内容など）をどこかに保存しておく必要があります。このタスクの資源のことを「コンテキスト」といいます。コンテキスト管理のためにリアルタイム OS は、設定するタスクの数だけ「TCB(タスクコントロールブロック)」を用意します。

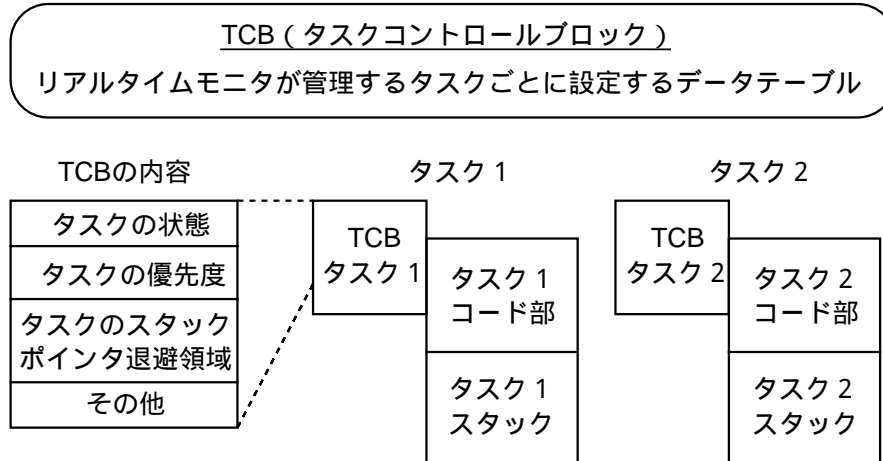


図 3.1.3 " TCB " の主な構造

タスクのディスパッチ

ディスパッチの流れを次に示します。

1. ディスパッチ発生
2. RUN 状態のタスクのコンテキストをスタックに退避
3. 現在のスタックポインタを TCB 内の領域に退避
4. 次に RUN 状態にするタスクの ID を確認
5. ID をもとに次に RUN 状態にするタスクの TCB からスタックポインタを確保
6. スタックからコンテキストを確保
7. スタックポインタをもとにリターンする (RUN 状態のタスクに切り替わる)

オブジェクトの種類

システムコールを用いて操作できる対象を「オブジェクト」といいます。タスク自身もシステムコールによって操作できますので、オブジェクトの一部です。表 3.1.2 に MR308 が用意している、タスク以外のオブジェクトについて示します。

表 3.1.2 MR308 のオブジェクト

オブジェクト名	機能
イベントフラグ	タスク間で同期をとるために用います。ビットごとの設定が可能です。(1ワード長)
セマフォ	タスク間で同期をとるために用います。主にタスク間での排他制御に使用します。セマフォは、セマフォカウンタをもとに排他制御を行います。
メールボックス	タスク間での通信(データの受け渡し)を行うために用います。メールボックスには、1ワード長のデータもしくは、データブロックのアドレスを送受信できます。

オブジェクトではありませんが、TCB 内部にタスク間での同期をとるためのカウンタが用意されます。また、各オブジェクトはタスク同様、任意の番号 " ID " によって管理されます。ID はユーザが任意の値を設定します。

コラム スケジューリング方式あれこれ

スケジューリング方式には、優先度方式のほかに以下のものがあります。

- ・ FCFS 方式(First Come First Service)
 - READY 状態になった順番にタスクが RUN 状態に切り換わる方式です。
- ・ ラウンドロビン方式 - FCFS 方式と同様に順番に RUN 状態に切り替わります。この方式では、一定時間ごとに強制的に RUN 状態のタスクがリアルタイム OS によって切り換えられます。

オブジェクトの管理

リアルタイム OS ではシステムコールを使ってオブジェクトの管理を行います。
表 3.1.3 に、タスクおよび各オブジェクトを操作するためのシステムコールと機能を示します。

表 3.1.3 オブジェクト操作のための主なシステムコール

分類	オブジェクト	システムコール	機能
タスク管理	タスク	sta_tsk()	タスクを起動 (READY状態) します。
		ext_tsk()	自タスクを正常終了 (DORMANT状態) します。
タスク付属同期	タスク	slp_tsk()	自タスクをWAIT状態にします。
		wup_tsk()	WAITタスクをREADY状態にします。
同期・通信	イベントフラグ	set_flg()	イベントフラグをセットします。イベントフラグを待っているタスクがあればそのタスクを起動 (READY状態) します。
		wai_flg()	イベントフラグを待ち (WAIT状態) ます。すでにイベントフラグがセットされている場合は処理を続けます。
	セマフォ	sig_sem()	セマフォを解放します (セマフォカウンタを+1します)。セマフォを待っているタスクがある場合は、そのタスクを起動 (READY状態) します。その場合、セマフォは変化しません。
		wai_sem()	セマフォカウンタがすでに 0 の場合、待ち (WAIT状態) ます。0 以外の場合はセマフォカウンタを - 1し、処理を続けます。
	メールボックス	snd_msg()	メッセージをメールボックスに送信します。メッセージを待っているタスクがあればそのタスクを起動 (READY状態) にし、メッセージを渡します。待っているタスクがない場合、メッセージはメールボックス内に保管されます。
		rcv_msg()	メールボックスからメッセージを受信します。メッセージがない場合は、待ち (WAIT状態) ます。メッセージがすでにある場合は、メッセージを受け取り、処理を続けます。

3.1.3 割り込み管理

MR308 では、割り込みプログラムを「割り込みハンドラ」と呼びます。

この項では、MR308 の割り込みハンドラの種類と、割り込みハンドラの 1 つである「OS 依存割り込みハンドラ」の管理方法を説明します。

割り込みハンドラの種類

MR308では割り込みハンドラを、内部でシステムコールを使用するか、使用しないかによって区別しています。前者を「OS 依存割り込みハンドラ」、後者を「OS 独立割り込みハンドラ」と呼びます。この項ではOS 依存割り込みハンドラについて説明します。

表 3.1.4 割り込みハンドラの種類

割り込みハンドラ	意味
OS 依存割り込みハンドラ	MR308の持つシステムコールを使用する割り込みハンドラです。割り込みプログラムと異なり、システムコールを使用するための処理が必要になります。
OS 独立割り込みハンドラ	MR308のシステムコールを使用しない割り込みハンドラです。割り込みプログラムと同じ動作をします。

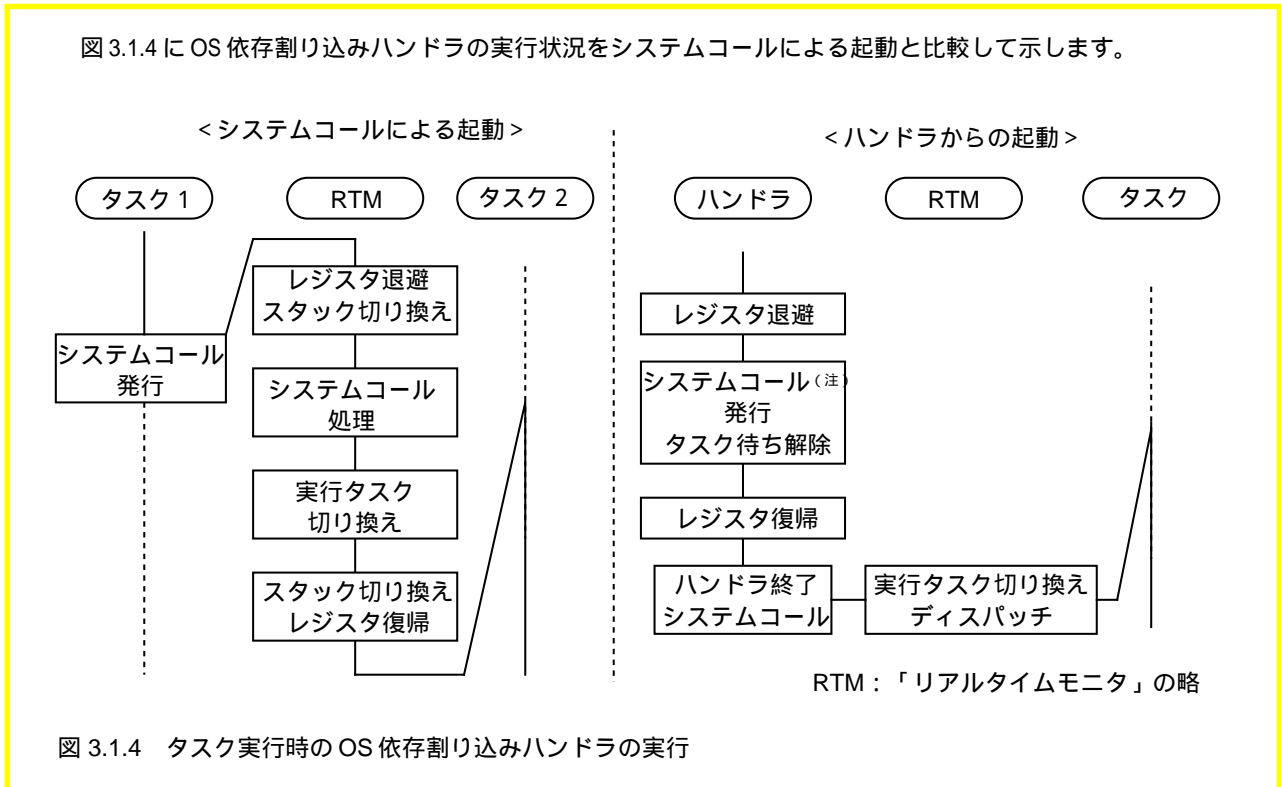
OS 依存割り込みハンドラ

OS 依存割り込みハンドラはタスクとは異なり、ディスパッチやスケジューリングの対象とはなりませんので TCB は作成されません。

OS 依存割り込みハンドラの処理手順は、以下の通りです。

1. レジスタの退避
2. ハンドラの処理 (システムコール使用)
3. レジスタの復帰
4. OS 依存割り込みハンドラ終了システムコール "ret_int"
 - *) OS 依存割り込みハンドラの終了時には OS 依存割り込みハンドラ終了用システムコールを用います。このシステムコール内でスケジューリング、ディスパッチが行われます。
 - OS 依存割り込みハンドラ終了時にディスパッチが行われるため、ハンドラ終了後 RUN 状態になるタスクは、割り込み発生時に RUN 状態にあったタスクと同じとは限りません。

OS 依存割り込みハンドラの実行



(注) 割り込みハンドラ内で使用できるシステムコールは限られています。必ず割り込みハンドラでの使用が可能なシステムコールを使用してください。

多重割り込み管理

割り込みは、多重(OS依存割り込みハンドラ実行時に割り込み許可レベルの高い割り込み発生)に起こる場合があります。

図 3.1.5 に多重割り込みの場合の OS 依存割り込みハンドラの動作を示します。

< タスク 1 > < OS依存割り込みハンドラ 1 > < OS依存割り込みハンドラ 2 >
(優先度低) (優先度高)

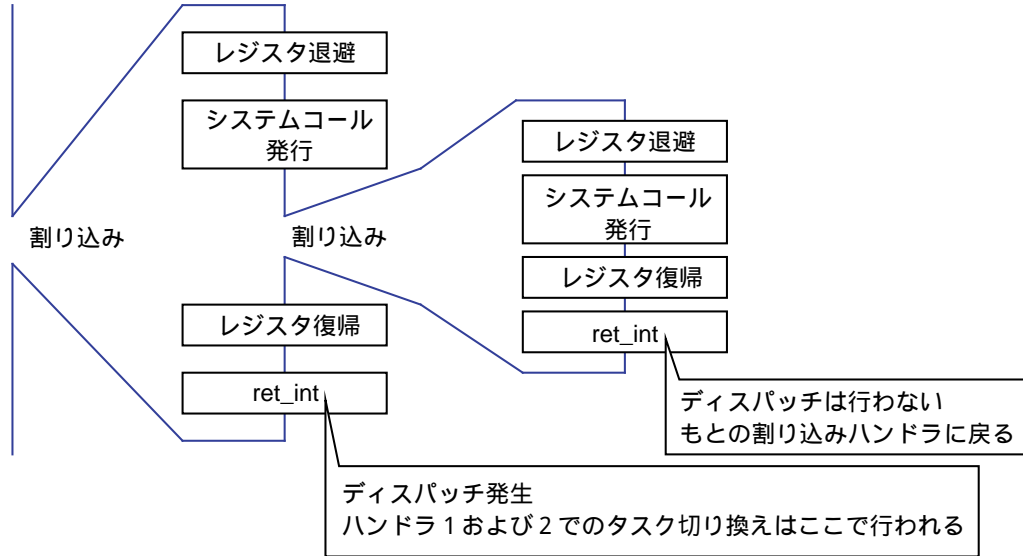


図 3.1.5 多重割り込みでの OS 依存割り込みハンドラの実行

多重割り込みが発生した場合、高い割り込みレベルによって起動した OS 依存割り込みハンドラ内のシステムコール "ret_int" は、ディスパッチを行いません。これは、OS 依存割り込みハンドラの処理をすべて終了してからタスクに戻る必要があるためです。

3.1.4 特殊なハンドラ

ハンドラには、割り込みハンドラのほかリアルタイム OS の持つ機能を利用したものがあります。
この項では、割り込みハンドラ以外の特殊なハンドラについて説明します。

システムクロック割り込みハンドラ

システムクロック割り込みハンドラはリアルタイム OS が用意するハンドラです。ハードウェアタイマ 1 本を「システムクロック」として占有して時間管理を行います。

表 3.1.5 リアルタイム OS が用意する割り込みハンドラ

ハンドラ名	機能	備考
システムクロック 割り込みハンドラ	リアルタイムモニタが用意するタイマ割り込み用のハンドラです。使用するタイマは、任意に指定します。時間管理機能を用いる場合は必要となります。	タイマを 1 本占有します。また使用不可とすることもできます。

システムクロック割り込みハンドラの周期時間（タイマ割り込み発生周期）は、任意に設定できます。

特殊なハンドラ

表 3.1.6 に示すハンドラはすべて、システムクロック割り込みハンドラの一部として呼び出されます。そのため、これらのハンドラ内ではシステムコールを使用できます。

ハンドラ名	機能	備考
周期起動ハンドラ	設定した時間ごとにシステムクロック割り込みハンドラ内から起動します。システムクロック割り込みハンドラの一部として動作しますのでサブルーチン形式となります。	ユーザが用意します。
アラームハンドラ	設定した時間に 1 度だけシステムクロック割り込みハンドラ内から起動します。システムクロック割り込みハンドラの一部として動作しますのでサブルーチン形式となります。	ユーザが用意します。

表 3.1.6 特殊なハンドラ

3.2 システムコールの利用法

3.2.1 MR308 のシステムコール

この項では、リアルタイム OS を利用する上で必要となるシステムコールについて、MR308 の提供形式と、システムへの組み込まれ方を説明します。

MR308 の提供形式

MR308 は、ライブラリ形式として提供されます。ライブラリとして提供されるということはリンク時のみ、このライブラリが組み込まれることを意味します。

また、MR308 を構成する各システムコールはライブラリのモジュールとなります。

図 3.2.1 に MR308 が提供するシステムコールライブラリを示します。

タスク管理	メールボックス	セマフォ	時間管理
タスク付属同期	イベントフラグ	メモリプール管理	バージョン管理
スケジューラ			割り込み処理

図 3.2.1 MR308 が提供するシステムコールライブラリ

システムへの組み込まれ方

MR308 は、各システムコールをライブラリ化したものです。そのためシステム全体をリンクするときには、プログラム内に記述したシステムコールだけが組み込まれます。MR308 すべてが組み込まれることはありません。

また、各システムコールはプログラムから見た場合、すべて外部関数として扱われます (MR308 が用意する関数となります)。

3.2.2 システムコールの記述方法

この項では、C 言語を用いてリアルタイム OS を使用する上でのシステムコールの記述方法を説明します。

基本的なシステムコールの記述方法

システムコールはすべて関数として扱われます。ですから、プログラム内でシステムコールを利用する方法は、通常の間数呼び出しと同じです。

```
#include <mr30.h>
void task1 ( void )
{
    slp_tsk();
}
```

MR308を利用する上で必要となるインクルードファイル

自タスクをWAIT状態にする

図 3.2.2 システムコールの記述

システムコールのパラメータ

システムコールへのパラメータは、関数の引数として記述します。

```
#include <mr308.h>
#include "id.h"
void task2 ( void )
{
    wup_tsk ( ID_task1 );
}
```

MR308を利用する上で必要となるインクルードファイル

オブジェクト操作を行う上で必要となるインクルードファイル

タスクを起動 (READY状態) する

図 3.2.3 パラメータを持つシステムコールの記述

オブジェクトの指定

MR308でオブジェクト操作を行うシステムコールを利用する場合、オブジェクトのIDを指定します。ただし、MR308ではこのIDをオブジェクト名を利用して視覚的にわかりやすく表現できます。

IDを数値のみで指定することも可能ですが、プログラムの可読性をよくするためには、この指定方法を推奨します。

IDの指定方法 -- ID_[オブジェクト名]
オブジェクト名は任意に設定します。

システムコールのエラーコード

システムコールの戻り値がすべてシステムコールのエラーコードとなっています。このエラーコードも特定の文字列を使用して判別できます。

表 3.2.1 にエラーコード一覧を示します。

表 3.2.1 エラーコード一覧 (注)

文字列	意味
E_OK	正常終了
E_OBJ	オブジェクトの状態が不正
E_QOVR	キューイングまたはネストのオーバーフロー
E_TMOUT	ポーリング失敗またはタイムアウト
E_RLWAI	待ち状態強制解除

このエラーコードを用いて、システムコールを利用した後の処理を選択することができます。利用例を図 3.2.4 に示します。

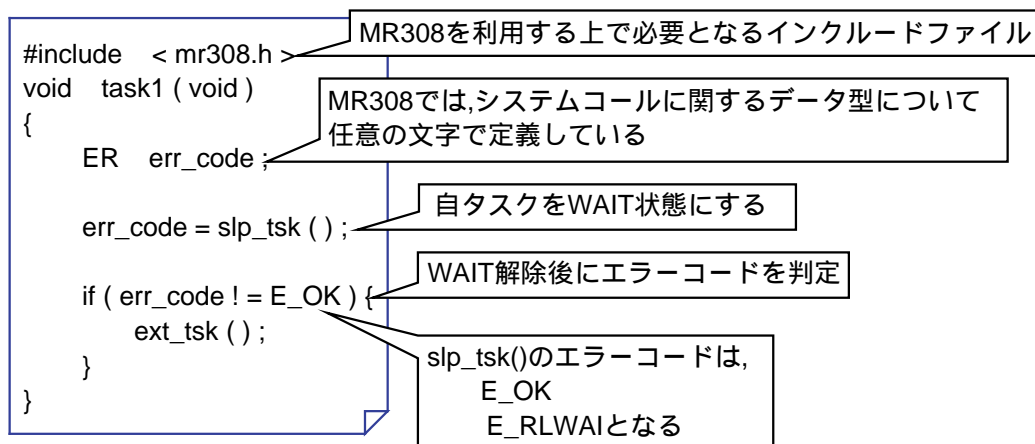


図 3.2.4 エラーコードの利用

(注) システムコールによって使用できるエラーコードは異なります。

コラム 定義されている文字列

MR308では、システムコールのパラメータに関するデータ型や特定のデータ型について文字列を定義しています。この文字列はμITRON仕様のリアルタイムOS同士の互換性をとるために統一されています。

表 3.2.2 データタイプと文字

特定データ					
符号付き8ビット整数	B	符号付き16ビット整数	H	符号付き32ビット整数	W
符号なし8ビット整数	UB	符号なし16ビット整数	UH	符号なし32ビット整数	UW
データタイプが一致しないものへのポインタ	*VP				
パラメータデータ					
オブジェクトID	ID	エラーコード	ER	タスク優先度	PRI

3.3 MR308 を用いた開発手順

3.3.1 開発時に必要なファイル

MR308 を使用してプログラムを開発する場合、プログラムの他に「スタートアッププログラム」、「オブジェクト定義ファイル」が必要です。

この項では、それぞれのファイルの内容について説明します。

MR308 用スタートアッププログラム

スタートアッププログラムの必要性は「2.2 スタートアッププログラム」で説明しています。ここではMR308 用スタートアッププログラムについて簡単に説明します。

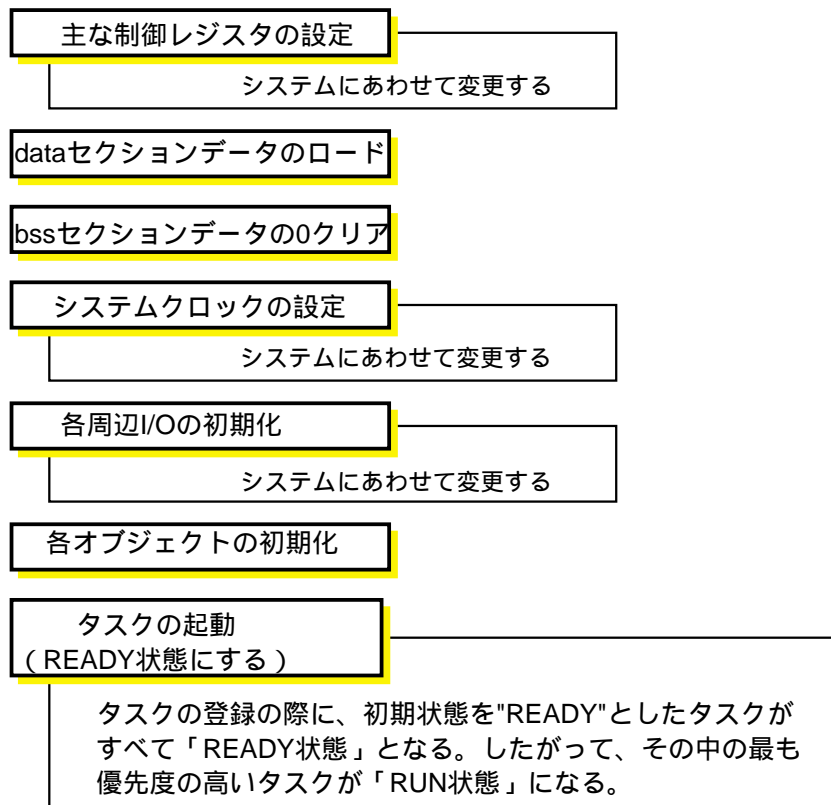


図 3.3.1 MR308 用スタートアッププログラムの処理概要

スタートアッププログラムの変更点

MR308 を用いてプログラム開発を行うためには、MR308 が提供するスタートアッププログラムを変更する必要があります。

主な変更内容を次に示します。

- ・プロセッサモードレジスタの設定
- ・割り込みベクタテーブルの先頭アドレスの設定
- ・使用する周辺 I / O の初期化
- ・メモリ配置の変更 (注)

プロセッサモードレジスタの設定 (" crt0mr.a30 ")

プロセッサモードレジスタなどM16C/80シリーズを直接制御するレジスタの初期化を行います。変更場所および記述方法を図 3.3.2 に示します。

リセット後プログラムはこのラベルからスタートする

```

=====
; Interrupt section start
; -----
;
; .glb          __SYS_INITIAL
; .section      MR_KERNEL, CODE, ALIGN
__SYS_INITIAL:
; -----
; after reset, this program will start
; -----
;
;   ldc   #__Sys_Sp, ISP           ; set initial ISP
;
;   mov.b #02H, PMOD              ; Set Processor Mode Register
;   mov.b #00H, 000AH            システムに合わせてプロセッサモードレジスタ
;   ldc   #0000h, flg             を設定する
;   ldc   #__Sys_Sp, fb
;   ldc   #data_SE_top, sb
    
```

図 3.3.2 M16C/80 シリーズ制御レジスタの初期化

(注) メモリ配置の変更は、スタートアッププログラム内では行えません。MR308 用セクション定義ファイル "c_sec.inc" を修正する必要があります。

割り込みベクタテーブルの先頭アドレスの設定 (" crt0mr.a30,c_sec.inc ")

割り込みベクタテーブルの先頭アドレスを設定します。この設定値は " crt0mr.a30 " 内で、割り込みテーブルレジスタ " INTB " に設定します。

(" crt0mr.a30 ")

```

;-----
; Set System IPL and Set Interrupt Vector
;-----
mov.b #0,R0L
mov.b #_SYS_IPL,R0H
ldc R0,FLG ; set system IPL
ldc #__INT_VECTOR,INTB
    
```

(" c_sec.inc ")

```

;-----
; VECTOR TABLE
;-----
.glb __INT_VECTOR
.section INTERRUPT_VECTOR ;Interrupt vector table
.org OFFFD00H
__INT_VECTOR:
    
```

" c_sec.inc "内で定義した値を割り込み
テーブルレジスタ" INTB "に設定する

図 3.3.3 割り込みベクタテーブルの先頭アドレスの設定

使用する周辺 I / O の初期化 (" crt0mr.a30 ")

周辺 I / O の初期設定は " crt0mr.a30 " 内に追加記述してください。追加場所を図 3.3.4 に示します。

```

;+-----+
;| User Initial Routine ( if there are ) |
;+-----+
;Initialize standard I/O
.glb _init
JSR.A _init
    
```

使用する周辺 I / O の初期設定プログラムを追加する

図 3.3.4 周辺 I / O の初期化

メモリ配置の変更 (" c_sec.inc ")

各セクションの先頭アドレスを疑似命令 ".org" を用いて設定します。先頭アドレス指定がないセクションについては、前に定義したセクションに連続してメモリに配置されます。

```

; -----
; Arrangement of section
; -----
; RAM SB data area
; -----
;
SECTION    data_SE,DATA
.ORG      400H
__MR_SB:
data_SE_TOP:

SECTION    bss_SE,DATA,ALIGN
bss_SE_top:

SECTION    data_SO,DATA
data_SO_top:

SECTION    bss_SO,DATA
bss_SO_top:
:
; -----
; Near data area
; -----
SECTION    data_NE,DATA,ALIGN
data_NE_top:

SECTION    bss_NE,DATA,ALIGN
bss_NE_top:

SECTION    data_NO,DATA
data_NO_top:

SECTION    bss_NO,DATA
bss_NO_top:
:
; -----
; Far RAM data area
; -----
SECTION    data_FE,DATA
.org      100000H
data_FE_top:
:
; -----
; Far ROM data area
; -----
SECTION    rom_FE,ROMDATA
.org      0FF0000H
rom_FE_top:
:

```

メモリマップに合わせて各領域の先頭番地を設定する

図 3.3.5 メモリ配置の変更

オブジェクト定義ファイル(コンフィグレーションファイル)

各オブジェクトの定義は「コンフィグレーションファイル」と呼ばれるファイルに記述します。このコンフィグレーションファイルは、MR308が提供するコンフィグレーションファイルのテンプレートファイル" default.cfg " をもとにして作成します。

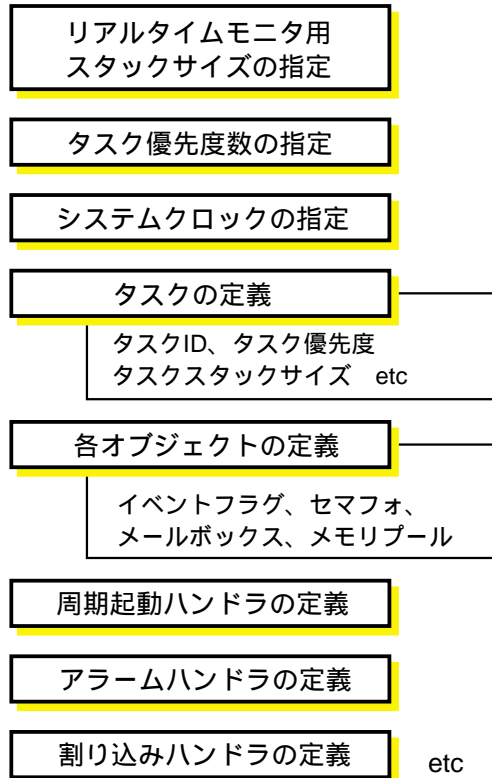


図 3.3.6 コンフィグレーションファイルの概要

作成したコンフィグレーションファイルは、MR308が提供する「コンフィグレータ" cfg308 "」によって、MR308を組み込む際に必要となるファイルに展開されます。

コラム MR308 用メモリ配置設定ファイル

MR308が提供するスタートアッププログラムには、メモリ配置を決定するインクルードファイルが含まれています。メモリ配置を変更する場合は、これらのインクルードファイルを修正する必要があります。ここでは、メモリ配置に関連するファイルについて説明します。

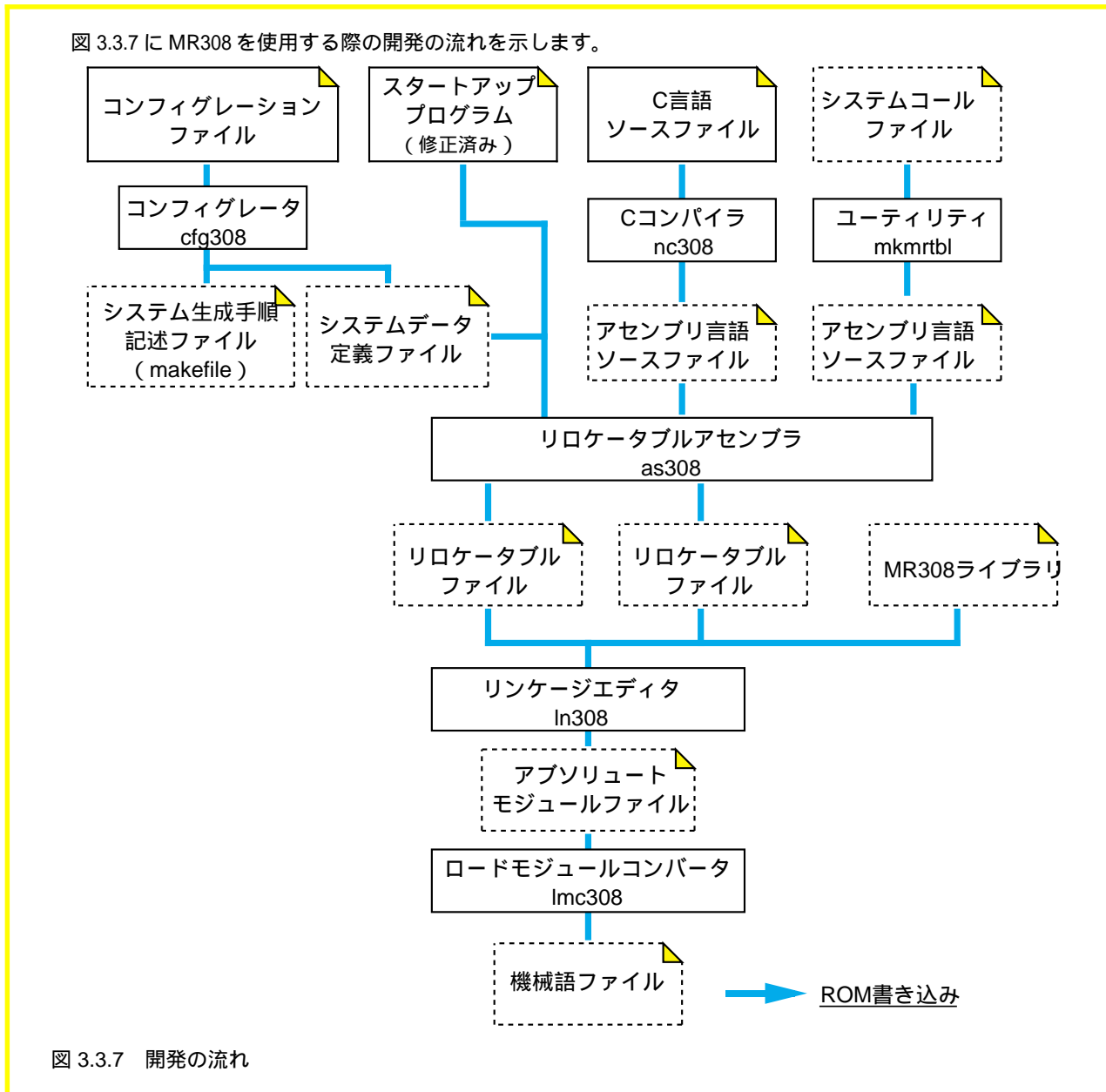
表 3.3.1 MR308 用メモリ配置関連ファイル

ファイル名	機能	備考
c_sec.inc	NC308を用いた場合のメモリ配置を行う インクルードファイル	C言語による開発用
asm_sec.inc	AS308のみを用いた場合のメモリ配置を行う インクルードファイル	アセンブリ言語による開発用

3.3.2 MR308 を用いた開発の流れ

この項では、MR308 を組み込んだプログラム開発の流れを説明します。

MR308 を用いた開発の流れ



開発の手順

以下の手順でプログラムを開発します。

1. 各タスク、ハンドラの設計, 作成
2. スタートアッププログラムの修正
3. メモリ配置の修正
4. コンフィギュレーションファイルの作成
5. コンフィグレータの起動
6. オブジェクトの作成

3.4 NC308 を用いた MR308 の組み込み

3.4.1 NC308 を用いたプログラムの記述

MR308 を組み込むために NC308 では拡張機能を用意しています。MR308 のための拡張機能は、MR308 のコンフィグレータを用いることにより特定ファイルに書き込まれます。そのためプログラム内に特定ファイルをインクルードすれば、拡張機能を既存のプログラム内に記述する必要はありません。しかし、拡張機能の意味は理解してください。

以下の項では、NC308 を用いた MR308 の組み込み方を説明します。

インクルードするファイル

MR308 を用いたプログラムを作成するためには、プログラムの先頭で必要なファイルをインクルードします。このインクルードファイルには MR308 をプログラム内に組み込むための必要な定義事項が書かれています。

表 3.4.1 MR308 を使用するためのインクルードファイル

ファイル名	機能
mr308.h	MR308が必要とする定義事項とシステムコールのプロトタイプ宣言を行います。
id.h	プログラム内で使用するオブジェクトのIDの定義を行います。 MR308用拡張機能を用いた宣言を行います。 (このファイルはコンフィグレータを起動した際にコンフィグレーションファイルをもとに自動作成されます。)

```
#include <mr308.h>
#include "id.h"
```

上の例は、" mr308.h " が標準ディレクトリ(環境変数 INC308 で指定されたディレクトリ)に、" id.h " がカレントディレクトリに置かれている場合の記述例です。

" id.h " は、コンフィグレータを起動することにより、カレントディレクトリに作成されます。

MR308 用拡張機能

MR308を使用するための拡張機能は、プリプロセスコマンドである #pragma 命令を使用します。これらの拡張機能は、指定する関数より前に記述しなければなりません。

表 3.4.2 に MR308 用拡張機能を示します。

表 3.4.2 MR308 用拡張命令一覧

拡張命令	意味
#pragma TASK	タスクとする関数を指定します。
#pragma INTHANDLER	OS依存割り込みハンドラとする関数を指定します。
#pragma HANDLER	INTHANDLERの省略形です。
#pragma CYCHANDLER	周期起動ハンドラとする関数を指定します。
#pragma ALMHANDLER	アラームハンドラとする関数を指定します。

ただし MR308 が持つコンフィグレータを使用すると、必要となる MR308 用拡張機能は自動的に組み込まれます。したがって、これらの拡張命令を記述する必要はありません。

3.4.2 NC308 を用いたタスクの記述

この項では、NC308 を用いてタスクを記述する方法と注意点を説明します。

タスクの記述方法

タスクに指定した関数およびその関数から呼び出される関数内では、MR308のシステムコールを使用できます。タスクの記述例を図 3.4.1 に示します。

```
#include < mr308.h >
#include " id.h "

void task1 ( void )
{
    for ( ;; ) {
        ⋮
    }
}
```

図 3.4.1 タスクの記述例

タスク指定による命令展開の特徴

タスクに指定した関数は以下の点で、通常関数の命令展開と異なります。

- ・関数を終了する場合、ext_tsk システムコールを出力します。

タスクを記述するときの注意点 - 1 -

タスクは、関数スタイルで記述します。その際、以下の点に注意してください。

- ・リターン値は、void 型とします。
- ・引数は、void 型または int 型引数を 1 つ持ちます。指定できる引数は 1 つだけです。MR308 ではタスクが最初に起動する場合 (sta_tsk システムコールの引数として)、スタートコードとして整数型のデータを 1 つ受け取ることができます。
- ・static 型の関数をタスクとして定義できません (図 3.4.3 参照)。
- ・タスクを再スタートさせた場合、タスク内で使用している外部変数と static 変数は初期化されません。再度初期化してください (図 3.4.4 参照)。

記述例と注意点を図 3.4.2、図 3.4.3、図 3.4.4 に示します。

```
#include < mr308.h >
#include " id.h "

void task1 ( void )
{
    :
}

void task2 ( int code )
{
    switch ( code ) {
        :
    }
}
```

引数に1個の整数型を指定できる

スタートコードを用いて処理を切り替えることができる

図 3.4.2 タスクの記述例

```
#include < mr308.h >
#include " id.h "

static void task3 ( void )
{
    :
}
```

static型の関数はタスクとして使用できない

図 3.4.3 タスクを記述する際の注意点 - 1 - (static 型の関数について)

タスクを記述するときの注意点 - 2 -

```
#include < mr308.h >
#include " id.h "

char mode = 0;

void task1 ( void )
{
    for ( ;; ) {
        if ( mode ) {
            ⋮
        }
    }
}
```

このタスクが一度終了（休止状態）となり
再度スタート（RUN状態）となった場合
外部変数（mode）は初期化されない

➔
正しくは

```
#include < mr308.h >
#include " id.h "

char mode = 0;

void task1 ( void )
{
    mode = 0;

    for ( ;; ) {
        if ( mode ) {
            ⋮
        }
    }
}
```

再スタートするタスクの中で外部変数とstatic変数を使用する場合はタスク関数内で初期化する

図 3.4.4 タスクを記述する際の注意点 - 2 - （再スタートするタスクの変数の初期化）

コラム 変数の参照範囲 (スコープ)

変数は、記憶クラスにより参照範囲が異なります。記憶クラスによる変数の参照範囲を表 3.4.3、図 3.4.5 に示します。

表 3.4.3 変数の参照範囲

変数の記憶クラス	参照範囲
外部変数	すべてのタスク、ハンドラで参照ができます。
タスク、ハンドラ外static変数	同一ファイル内のタスク、ハンドラで参照できます。
タスク、ハンドラ内static変数	1つのタスク、ハンドラ内で参照できます。
内部変数	1つのタスク、ハンドラ内で参照できます。
レジスタ変数	

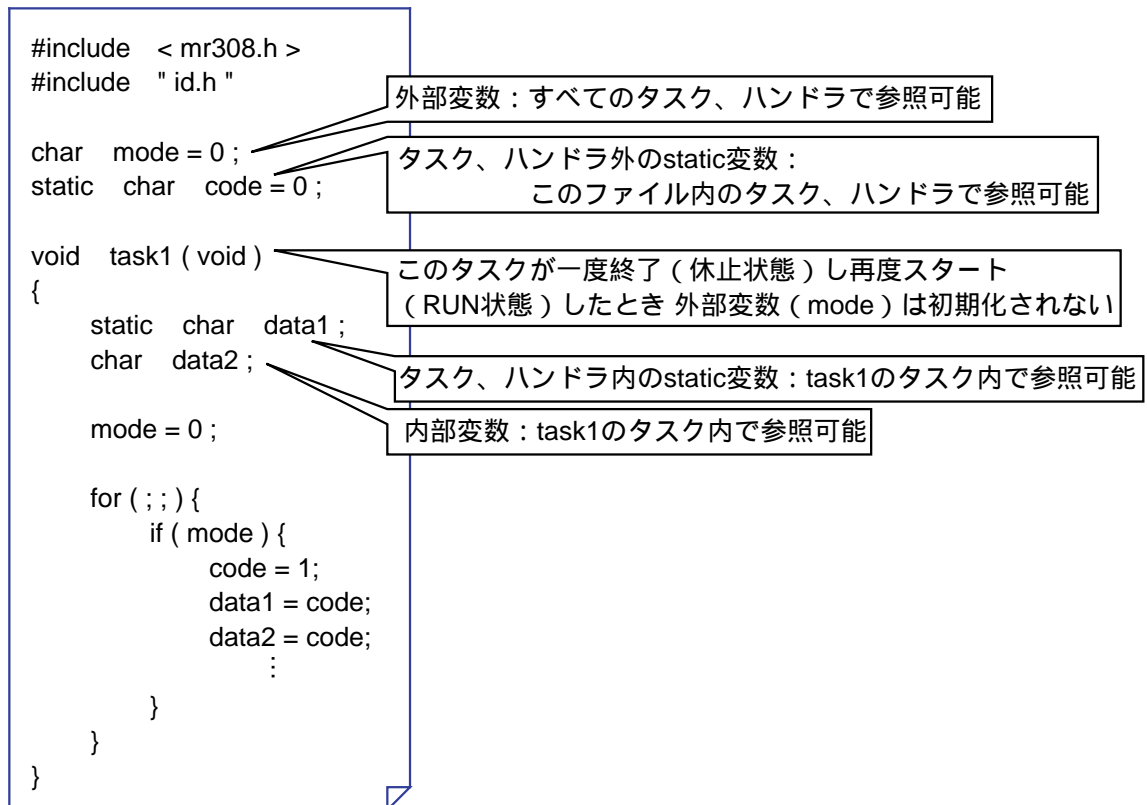


図 3.4.5 変数の参照範囲例

3.4.3 割り込みハンドラの記述

MR308 では、割り込みハンドラは「OS 依存割り込みハンドラ」と「OS 独立割り込みハンドラ」の 2 種類に分類されています。

この項では、OS 依存割り込みハンドラの記述方法と注意点を説明します。(注)

OS 依存割り込みハンドラを C 言語によって記述する

OS 依存割り込みハンドラと指定した関数内では、OS 依存割り込みハンドラ内で使用可能なシステムコールを使用することができます。

記述例を図 3.4.6 に示します。

```
#include < mr308.h >
#include " id.h "

void int_hand ( void )
{
    :
}
```

図 3.4.6 OS 依存割り込みハンドラの記述例

OS 依存割り込みハンドラの命令展開の特徴

OS 依存割り込みハンドラと指定した関数は、以下の処理を行う命令に展開されます。

- ・全レジスタのスタックへの退避を行います。
- ・MR308 用割り込みハンドラ入口処理を行います。
- ・終了時に全レジスタのスタックからの復帰を行います。
- ・ret_int システムコールを用いて終了します。

(注) 「OS 独立割り込みハンドラ」の記述方法は、「2.5 割り込み処理」に記述している内容と同様です。

OS 依存割り込みハンドラを記述するための注意点

OS 依存割り込みハンドラは関数スタイルで記述します。その際、次のような点に注意してください。

- ・リターン値は void 型のみ有効です。
- ・引数は void 型のみ有効です。
- ・static 型の関数を定義することはできません。
- ・使用できるシステムコールは、ハンドラ内で使用可能なもののみです。

```
#include < mr308.h >
#include " id.h "

void int_hand ( void )
{
    iwup_tsk ( ID_task1 );
    ...
}
```

OS依存割り込みハンドラはリターン値引数ともvoid型のみ

OS依存割り込みハンドラ内ではハンドラ内で使用可能なシステムコールを使用する

図 3.4.7 OS 依存割り込みハンドラの記述例

```
#include < mr308.h >
#include " id.h "

static void int_hand ( void )
{
    ...
}
```

static型の関数をOS依存割り込みハンドラとして定義できない

図 3.4.8 OS 依存割り込みハンドラを記述する際の注意点 (static 型の関数について)

OS 依存割り込みハンドラとタスクのデータのやり取り

OS 依存割り込みハンドラとタスク間でデータをやり取りする方法は、外部変数を用いる方法とメールボックスを用いる方法の 2 通りがあります。

図 3.4.9 に外部変数を利用した例を示します。

```

#include < mr308.h >
#include " id.h "

char data1 ;
void int_hand ( void )
{
    data1 = 0x10 ;
    iwup_tsk ( ID_task1 ) ;
    ...
}

void task1 ( void )
{
    for ( ; ; ) {
        slp_tsk();
        if ( data1 ) {
            ...
        }
    }
}

```

タスクとやり取りを行う場合外部変数を宣言する

OS 依存割り込みハンドラからのデータを使用する

図 3.4.9 外部変数を利用したデータのやり取りの記述例

コラム ハンドラ内で使用できるシステムコール

OS 依存割り込みハンドラ、周期起動ハンドラ、アラームハンドラ内では特定のシステムコールのみ使用できます。使用できないシステムコールを用いた場合、プログラムが正常に動作しませんので注意してください。また、ixxx_xxx形式のシステムコールはハンドラ専用のシステムコールです。システムコールの詳細な機能については、MR308 のマニュアルを参照してください。

ista_tsk()	ichg_pri()	irot_rdq()	irel_wai()	get_tid()	isus_tsk()
irsm_tsk()	iwup_tsk()	iset_flg()	clr_flg()	pol_flg()	isig_sem()
preq_sem()	isnd_msg()	prcv_msg()	set_tim()	get_tim()	act_cyc()
ret_int()	get_ver()	can_wup()	ref_tsk()	ref_flg()	ref_sem()
ref_mbx()	ref_mpf()	ref_mpl()	ref_cyc()	ref_alm()	pget_blf()
relblf()					

メールボックスを利用したデータのやり取り

図 3.4.10 にメールボックスを利用して OS 依存割り込みハンドラとタスク間でデータのやり取りをする例を示します。この記述例では 16 ビット長のデータをメッセージとしています。ほかに、16 ビット長のアドレスをメッセージとすることもできます。

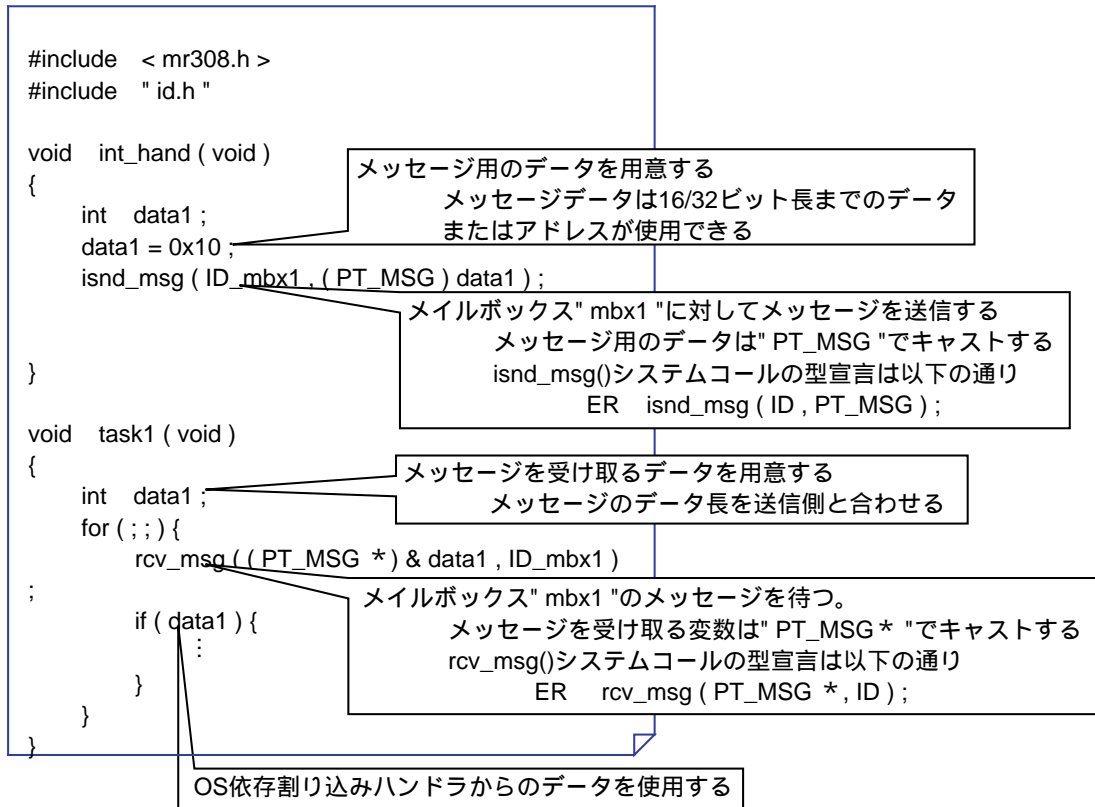


図 3.4.10 メールボックスを利用したデータのやり取りの記述例

3.4.4 周期起動ハンドラ、アラームハンドラの記述

この項では、周期ハンドラおよびアラームハンドラの記述方法と注意点を説明します。

周期起動ハンドラ、アラームハンドラの記述方法

周期起動ハンドラ、アラームハンドラに指定した関数内では、ハンドラ内で使用可能なシステムコールを使用することができます。

記述例を図 3.4.11 に示します。

```
#include < mr308.h >
#include " id.h "

void cyc_hand(void)
{
    :
}
```

図 3.4.11 周期起動ハンドラ、アラームハンドラの記述例

周期起動ハンドラ、アラームハンドラは、MR308の用意するシステムクロック割り込みハンドラ内で呼び出される関数になります。

命令展開の特徴

周期起動ハンドラ、アラームハンドラに指定した関数は、以下の処理を行う命令に展開されます。

- ・ rts 命令 (M16C/80 シリーズ用サブルーチン復帰命令) または、 exitd 命令 (M16C/80 シリーズ用関数復帰命令) を用いて終了します。

周期起動ハンドラおよびアラームハンドラを記述するための注意点

周期起動ハンドラ、アラームハンドラは、関数スタイルで記述します。その際、以下の点に注意してください。

- ・リターン値は void 型のみ有効です。
- ・引数は void 型のみ有効です。
- ・static 型の関数を周期起動ハンドラ、アラームハンドラとして定義できません。
- ・使用できるシステムコールは、ハンドラ内で使用可能なもののみです。

```
#include < mr308.h >
#include " id.h "

void cyc_hand ( void )
{
    iwup_tsk ( ID_task1 );
    :
}
```

周期起動ハンドラは、リターン値引数ともvoid型とする

周期起動ハンドラ内ではハンドラ内で使用可能なシステムコールを使用する

図 3.4.12 周期起動ハンドラの記述例

```
#include < mr308.h >
#include " id.h "

static void cyc_hand ( void )
{
    :
}
```

static型の関数を周期起動ハンドラとして定義できない

図 3.4.13 周期起動ハンドラの記述例（誤りの例）

周期起動ハンドラ、アラームハンドラとタスクのデータのやり取り方法

周期起動ハンドラ、アラームハンドラが、タスクとデータのやり取りを行う場合は、OS依存割り込みハンドラとタスクのデータのやり取りと同様の方法を用います。

付録

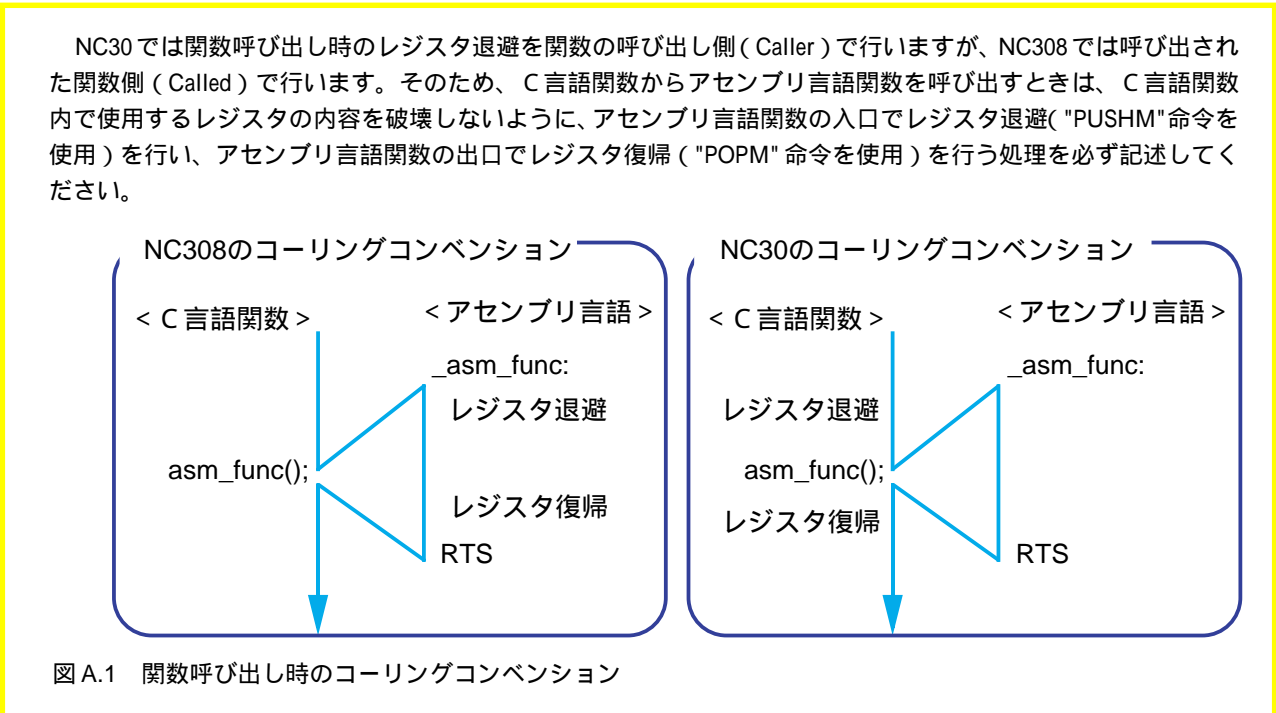
付録 A NC308 と NC30 の機能比較

付録 B NC308 コマンドリファレンス

付録 C Q & A

付録A NC308 と NC30 の機能比較

関数呼び出し時のコーリングコンベンション



引数の引き渡し規則

NC308 と NC30 の引数引き渡し規則の相違点を表 A.1 と表 A.2 に示します。

表 A.1 NC308 の引数の引き渡し規則

引数の型	第1引数	第2引数	第3引数以降
char型	R0L	スタック	スタック
short、int型 nearポインタ型	R0	スタック	スタック
その他の型	スタック	スタック	スタック

表 A.2 NC30 の引数の引き渡し規則

引数の型	第1引数	第2引数	第3引数以降
char型	R1L	スタック	スタック
short、int型 nearポインタ型	R1	R2	スタック
その他の型	スタック	スタック	スタック

アセンブルマクロ関数

NC308ではアセンブリ言語命令の一部をC言語の関数として記述することができる「アセンブルマクロ関数機能」をサポートしています。

この機能を使用することにより、アセンブリ言語命令の一部を直接的にC言語のプログラムに記述することができるので、プログラムのチューンナップが行いやすくなります。

詳しくは「2.3.5 アセンブルマクロ関数の使用」を参照してください。

表 A.3 アセンブルマクロ関数で記述可能なアセンブリ言語命令

DADD	DADC	DSUB	DSBB
RMPA	MAX	MIN	SMOVB
SMOVF	SMOVU	SIN	SOUT
SSTR	ROLC	RORC	ROT
SHA	SHL		

機能変更された拡張機能

near / far 修飾子の定義およびポインタ変数のアドレスサイズのデフォルトが変わります。

表 A.4 機能変更された拡張機能

項目	NC308	NC30
near・far 修飾子	データをアクセスするアドレッシングモードを指定する。 near;000000H~00FFFFH番地のアクセス far;000000H~FFFFFFH番地のアクセス	データをアクセスするアドレッシングモードを指定する。 near;000000H~0FFFFH番地のアクセス far;000000H~FFFFFFH番地のアクセス
ポインタ変数のデフォルトアドレスサイズ	farポインタ (4 バイト)	nearポインタ (2 バイト)

追加された拡張機能

NC308 では M16C/80 シリーズより機能追加となった、「高速割り込み」に対応するための拡張機能を追加しました。

表 A.5 追加された拡張機能

項目	NC308
#pragma INTERRUPT /F	高速割り込み関数呼び出し時に、全レジスタの退避 / 復帰を行い、高速割り込みルーチンからの復帰命令 FREIT 命令を生成する。
#pragma INTERRUPT /F/B	高速割り込み関数呼び出し時に、レジスタをスタックに退避する代わりにレジスタバンクを切り替える。また、高速割り込みルーチンからの復帰命令 FREIT 命令を生成する。
#pragma INTERRUPT /F/E	高速割り込み関数呼び出し時に、割り込み処理関数の入り口 (割り込み処理関数の直後) で割り込み許可フラグ (I フラグ) を "1" にして多重割り込みを許可の状態にする。また、高速割り込みルーチンからの復帰命令 FREIT 命令を生成する。

削除された拡張機能

表 A.6 に示す NC30 の拡張機能は、NC308 ではサポートしていません。

表 A.6 NC308 ではサポートしていない拡張機能

項目	機能
#pragma BIT	16ビット絶対アドレッシングモードによる、1ビット操作命令を使用できる領域にある変数であることを宣言する。
#pragma EXT4MPTR	4 Mバイトの拡張領域に配置する変数（ポインタ）を宣言する。

付録 B NC308 コマンドリファレンス

NC308 コマンドの入力書式

%nc308 [起動オプション] [アセンブリ言語ソースファイル名]
[リロケートブルオブジェクトファイル名] <C 言語ソースファイル名 >

% : プロンプトを示します。

< > : 必須項目を示します。

[] : 必要に応じて記述する項目を示します。

: スペースを示します。

複数のオプションを記述する場合はスペースキーで区切ってください。

コンパイルドライバの制御に関するオプション

表 B.1 コンパイルドライバの制御に関するオプション

オプション	機能
- c	リロケートブルファイル (属性 .r30) を作成し処理を終了します。
- D識別子	識別子を定義します。#defineと同じ機能です。
- Iディレクトリ名	#includeで指定するファイルが存在するディレクトリ名を指定します。 ディレクトリは最大8個まで指定可能です。
- E	プリプロセスコマンドのみを起動し結果を標準出力に出力します。
- P	プリプロセスコマンドのみを起動しファイル (属性 .i) を作成します。
- S	アセンブリ言語ソースファイル (属性 .a30) を作成し、処理を終了します。
- Uプリディファインドマクロ名	指定したプリディファインドマクロを未定義にします。
- silent	起動時のコピーライトメッセージを出力しません。

起動オプション - c、- E、- Pおよび - Sを指定しない場合、NC308 は In308 まで制御を行い、アプソリュートモジュールファイル (属性 .x30) まで作成します。

出力ファイル指定オプション

表 B.2 出力ファイル指定オプション

オプション	機能
- oファイル名	ln308が生成するファイル（アブソリュートモジュールファイル、マップファイルなど）の名称を指定します。また、ディレクトリ名も指定できます。ファイルの拡張子は記述しないでください。
- dirディレクトリ名	ln308が生成するファイル（アブソリュートモジュールファイル、マップファイルなど）の出力先ディレクトリを指定します。

バージョン情報表示オプション

表 B.3 バージョン情報表示オプション

オプション	機能
- v	実行中のコマンドプログラム名およびコマンドラインを表示します。
- V	コンパイラの各プログラムの起動時にメッセージを表示し、処理を終了します（コンパイル処理は行いません）。

デバッグ用オプション

表 B.4 デバッグ用オプション

オプション	機能
- g	デバッグ情報をアセンブリ言語ソースファイル（属性 .a30）に出力します。
- genter	関数呼び出し時に必ずenter命令を出力します。 デバッガのスタックトレース機能を使用するときは必ずこのオプションを指定してください。
- gno_reg	レジスタ変数に関するデバッグ情報の出力を抑止します。

最適化オプション

表 B.5 最適化オプション

オプション	短縮形	機能
- O	なし	速度およびROM容量ともに最小にする最適化を行います。
- OR	なし	速度よりもROM容量を重視した最適化を行います。
- OS	なし	ROM容量よりも速度を重視した最適化を行います。
- Oconst	- OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
- Ono_bit	- ONB	ビット操作をまとめる最適化を抑止します。
- Ono_break_source_debug	- ONBSD	ソース行情報に影響する最適化を抑止します。
- Ono_float_const_fold	- ONFCF	浮動小数点の畳み込み処理を抑止します。
- Ono_stdlib	- ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更などを抑止します。
- Osp_adjust	- OSA	スタック補正コードを取り除く最適化を行います。これによりROM容量を削減することができます。ただし、使用するスタック量が多くなる可能性があります。

ライブラリ指定オプション

表 B.6 ライブラリ指定オプション

オプション	機能
-lライブラリファイル名	リンク時に使用するライブラリを指定します。

生成コード変更オプション

表 B.7 生成コード変更オプション

オプション	短縮形	機能
- fansi	なし	- fnot_reserve_asm、 - fnot_reserve_far_and_near、 - fnot_reserve_inline、 および - fextend_to_intを有効にします。
- fnot_reserve_asm	- fNRA	asmを予約語にしません。(_asmのみ有効になります。)
- fnot_reserve_far_and_near	- fNRFAN	far、nearを予約語にしません。(_far、_nearのみ有効になります。)
- fnot_reserve_inline	- fNRI	inlineを予約語にしません。(_inlineのみ有効になります。)
- fextend_to_int	- fETI	char型データをint型に拡張し演算を行います (ANSI規格で定められた拡張を行います) (注)。
- fchar_enumerator	- fCE	enumerator (列挙子) の型をint型ではなく unsigned char型で扱います。
- fno_even	- fNE	データ出力時に奇数データと偶数データを分離しないで、すべてodd属性のセクションに配置します。
- fshow_stack_usage	- fSSU	スタックの使用状況をファイル (拡張子 .stk) に出力します。
- ffar_RAM	- fFRAM	RAMデータのデフォルト属性をfarにします。
- fnear_ROM	- fNROM	ROMデータのデフォルト属性をnearにします。
- fnear_pointer	- fNP	ポインタ型のデフォルトをnearとして扱います。
- fconst_not_ROM	- fCNR	constで指定した型をROMデータとして扱いません。
- fnot_address_volatile	- fNAV	#pragma ADDRESS (#pragma EQU) で指定した変数をvolatileで指定した変数とみなしません。
- fsmall_array	- fSA	far型の配列を参照する場合、その総サイズが64Kバイト以内であれば、添字の計算を16ビットで行います。
- fenable_register	- fER	レジスタ記憶クラスを有効にします。
- falign	なし	関数の先頭アドレスを常に偶数に配置します。
- fJSRW	なし	関数呼び出しの命令のデフォルトをJSR.W命令に変更します。
- fuse_DIV	- fUD	除算に対するコード生成を変更します。

(注) NC308 ではデフォルトで char 型データはデータ拡張をせずに評価を行います。ANSI 規格では char 型データまたは int 型データを評価するときには必ず int 型にデータ拡張を行います。

警告オプション

表 B.8 警告オプション

オプション	短縮形	機能
- Wnon_prototype	- WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
- Wunknown_pragma	- WUP	サポートしていない#pragmaを使用した場合、警告を出します。
- Wno_stop	- WNS	エラーが発生してもコンパイル作業を停止しません。
- Wstdout	なし	エラーメッセージをホストマシンの標準出力 (stdout) に出力します。
- Werror_fileタグファイル名	- WEF	タグファイルを出力します。
- Wstop_at_warning	- WSAW	ワーニング発生時にコンパイル処理を停止します。
- Wnesting_comment	- WNC	コメント中に*を記述した場合に警告を出します。
- Wccom_max_warnings	- WCMW	ccomnoccの出力するwarningの回数の上限を指定できます。
- Wall	なし	検出可能な警告をすべて表示します。
- Wmake_tagfile	- WMT	タグファイルに生成メッセージの内容を出力します。

アセンブル・リンクオプション

表 B.9 アセンブル・リンクオプション

オプション	機能
- as308 <オプション>	アセンブルコマンドas308のオプションを指定します。 2個以上のオプションを渡すときは、" (ダブルクォーテーション) で囲んでください。
- ln308 <オプション>	リンクコマンドln308のオプションを指定します。 2個以上のオプションを渡す場合は、" (ダブルクォーテーション) で囲んでください。

その他のオプション

表 B.10 その他のオプション

オプション	短縮形	機能
- dsource	- dS	出力するアセンブリ言語ソースファイルリスト中にC言語ソースリスティングをコメントとして出力します。

コマンド入力例

スタートアッププログラム (ncrt0.a30) と C 言語ソースプログラム (c_src.c) をリンクし、 アブソリュートモジュールファイル (test.x30) を作成する。

```
%nc308 -otest ncr0.a30 c_src.c
      出力ファイル名を指定
```

アセンブラリストファイルとマップファイルを生成します。

```
%nc308 -as308 "-" -ln308 "-M" c_src.c
      "as308"、"ln308" のオプションを指定
```

デバッグ情報をアセンブリ言語ソースファイル (属性 .a30) に出力します。

```
%nc308 -g -S ncr0.a30 c_src.c
```

付録C Q & A

構造体の転送 (コピー)

< Question >

構造体を転送 (コピー) するには、どのような方法がありますか？

< Answer >

- (1) 同じ定義の構造体を転送する場合
構造対変数名と代入演算子を用いて転送します。
- (2) 異なる定義の構造体を転送する場合
メンバごとに、代入演算子を用いて転送します。

```

struct tag1 { /* 構造体の定義 */
    int    mem1 ;
    char   mem2 ;
    int    mem3 ;
};

struct tag2 {
    int    mem1 ;
    char   mem2 ;
    int    mem3 ;
};

near struct tag1 near_s1t1,near_s2t1 ;
near struct tag2 near_s1t2 ;
far struct tag1 far_s1t1,far_s2t1 ;

main()
{
    near_s1t1.mem1 = 0x1234 ;
    near_s1t1.mem2 = 'A' ;
    near_s1t1.mem3 = 0x5678 ;

    /* 同じ定義の構造体の転送 ----- */
    near_s2t1 = near_s1t1 ; /* near -> near */
    far_s1t1 = near_s1t1 ; /* near -> far */
    near_s2t1 = far_s1t1 ; /* far -> near */
    far_s2t1 = far_s1t1 ; /* far -> far */

    /* 異なる定義の構造体の転送 ----- */
    near_s1t2.mem1 = near_s1t1.mem1 ;
    near_s1t2.mem2 = near_s1t1.mem2 ;
    near_s1t2.mem3 = near_s1t1.mem3 ;
}
    
```

(1) 同じ定義の構造体の場合
配置領域にかかわらず、構造体変数名
と代入演算子で転送できる

(2) 異なる定義の構造体の場合
メンバごとに転送する

図 C.1 構造体の転送の記述例

生成コードの削減 (1)

< Question >

生成コードを減らしたいのですが、どのような点をチェックすればよいでしょうか？

< Answer >

以下の点をチェックしてください。

【データ宣言時に・・・】

(1) int 型を宣言しているデータで、次のようなデータ範囲に納まるものではありませんか。

なお、() 内は省略可能です。

unsigned int 型で、0 ~ 255 以内 (unsigned) char 型に修正してください。

(signed) int 型で、- 128 ~ 127 以内 signed char 型に修正してください。

(2) unsigned / signed 修飾子を省略している int 型以上のデータで、負の数をもたないものではありませんか。

unsigned 修飾子を付けてください。

(NC308 では、int 型以上のデータは "signed" がデフォルトになります)

【far 型配列宣言時に・・・】

(1) サイズが 64K バイトを超えない範囲で、且つ extern 宣言している far 型配列で、要素数を省略しているものはありませんか。

配列の要素数を省略せずに明記してください。または生成コード変更オプション " - fsmall_array (- fSA)" を指定してください。

(NC308 では生成コード変更オプション " - fsmall_array(- fSA)" を指定すると、要素数の計算を 16 ビットで行います。)

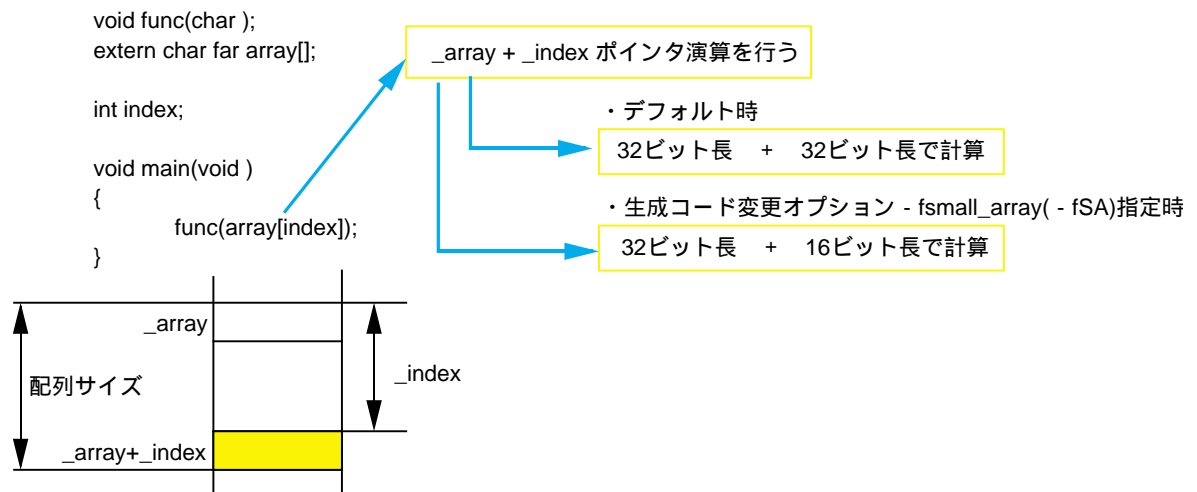


図 C.2 生成コード変更オプション ' - fsmall_array(- fSA)' 使用例

【コンパイル時に・・・】

(1) 最適化オプション "- OR" を指定していますか。 "- OR" を指定してください。

(NC308 では最適化オプション "- OR" を指定すると、ROM効率を重視した最適化を行います。)

(2) 最適化オプション "- Osp_adjust(- OSA)" を指定していますか。 "- Osp_adjust(- OSA)" を指定してください。

(NC308 では最適化オプション "- Osp_adjust(- OSA)" を指定すると、スタック補正コードを取り除く最適化を行います。これにより、ROM容量を削減することができます。ただし、使用するスタック量が多くなる可能性があります。)

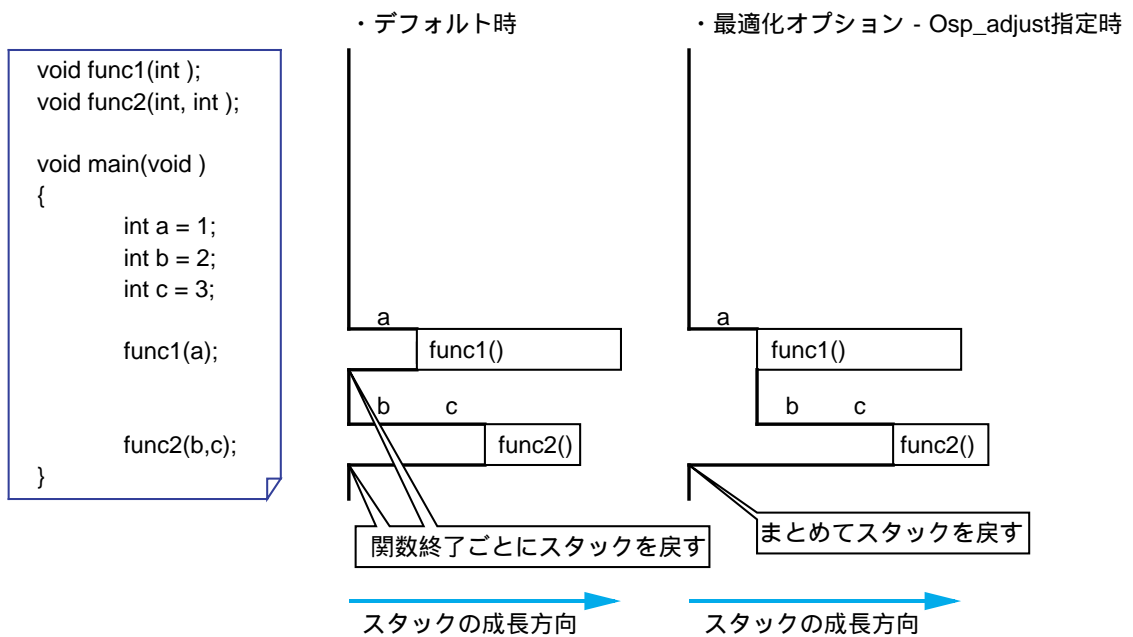


図 C.3 最適化オプション "- Osp_adjust(- OSA)" 使用例

生成コードの削減 (2)

< Question >

ファイル分割をしています。
生成コードを削減するためには、どのような点に注意すればよいでしょうか。

< Answer >

以下の点に注意してください。

【SB 相対アドレッシング内にあるデータを参照するとき・・・】

(1) SB 相対アドレッシング内にあるデータを参照する場合、必ず "#pragma SBDATA" を宣言してください。

<p>< ソースファイル 1 > "mode" を定義</p> <pre>void func1(void); char mode; #pragma SBDATA mode void main(void) { mode = 1; func1(); }</pre>	<p>< ソースファイル 2 > "mode" を参照</p> <pre>extern void func(void); extern char mode; #pragma SBDATA mode void func1(void) { mode = mode + 1; }</pre>
--	--

"mode" を SB 相対でアクセスするためには参照側で "#pragma SBDATA" を宣言する

図 C.4 "#pragma SBDATA" の記述例

【生成コードが 64k バイト以下のプログラムのとき・・・】

(1) 生成コード変更オプション " - fJSRW" を指定してください。

リンク時エラーが発生した場合は、エラーが発生した関数のみを "#pragma JSRA " 宣言してください。

リンクエラーが発生した場合は
関数func2をjsr.a命令で呼び出すように#pragma JSRA宣言する

<p>test1.c</p> <pre>void main(void); extern void func1(void); extern void func2(void); #pragma JSRA func2 void main(void) { ... }</pre>	<p>test2.c</p> <pre>void func1(void); void func2(void); void func1(void) { ... } void func2(void) { ... }</pre>	<p>%nc308 -fJSRW test1.c test2.c</p>
--	---	--------------------------------------

図 C.5 " - fJSRW" の使用例と "#pragma JSRA" の記述例

改訂記録	M16C/80 シリーズプログラム作成の手引き < C 言語編 > アプリケーションノート
------	--

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2003.09.25	-	初版発行

安全設計に関するお願い

- ・弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご注意ください。

本資料ご利用に際しての留意事項

- ・本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
- ・本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
- ・本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ (<http://www.renesas.com>) などを通じて公開される情報に常にご注意ください。
- ・本資料に記載した情報は、正確を期すため、慎重に制作したものです。万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
- ・本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任を負いません。
- ・本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際は、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
- ・本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
- ・本資料に関し詳細についてのお問い合わせ、その他お気づきの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。