FJXTMACU8

MACU8 アセンブラパッケージ
ユーザーズマニュアル

プログラム開発支援ソフトウェア

リロケータブルアセンブラ	RASU8
リンカ	RLU8
ライブラリアン	LIBU8
オブジェクトコンバータ	OHU8

2012 年 8 月発行

ご注意

本資料の一部または全部をラピスセミコンダクタの許可なく、転載・複写することを堅くお断りします。

本資料の記載内容は改良などのため予告なく変更することがあります。

本資料に記載されている内容は製品のご紹介資料です。ご使用にあたりましては、別途仕様書を必ずご請求のうえ、ご確認ください。

本資料に記載されております応用回路例やその定数などの情報につきましては、本製品の標準的な動作や使い方を説明するものです。したがいまして、量産設計をされる場合には、外部諸条件を考慮していただきますようお願いいたします。

本資料に記載されております情報は、正確を期すため慎重に作成したものですが、万が一、当該情報の誤り・誤植に起因する損害がお客様に生じた場合においても、ラピスセミコンダクタはその責任を負うものではありません。

本資料に記載されております技術情報は、製品の代表的動作および応用回路例などを示したものであり、ラピスセミコンダクタまたは他社の知的財産権その他のあらゆる権利について明示的にも黙示的にも、その実施または利用を許諾するものではありません。上記技術情報の使用に起因して紛争が発生した場合、ラピスセミコンダクタはその責任を負うものではありません。

本資料に掲載されております製品は、一般的な電子機器（AV 機器、OA 機器、通信機器、家電製品、アミューズメント機器など）への使用を意図しています。

本資料に掲載されております製品は、「耐放射線設計」はなされておりません。

ラピスセミコンダクタは常に品質・信頼性の向上に取り組んでおりますが、種々の要因で故障することもあり得ます。

ラピスセミコンダクタ製品が故障した際、その影響により人身事故、火災損害等が起こらないようご使用機器でのディレーティング、冗長設計、延焼防止、フェイルセーフ等の安全確保をお願いします。定格を超えたご使用や使用上の注意書が守られていない場合、いかなる責任もラピスセミコンダクタは負うものではありません。

極めて高度な信頼性が要求され、その製品の故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのある機器・装置・システム（医療機器、輸送機器、航空宇宙機、原子力制御、燃料制御、各種安全装置など）へのご使用を意図して設計・製造されたものではありません。上記特定用途に使用された場合、いかなる責任もラピスセミコンダクタは負うものではありません。上記特定用途への使用を検討される際は、事前にローム営業窓口までご相談願います。

本資料に記載されております製品および技術のうち「外国為替及び外国貿易法」に該当する製品または技術を輸出する場合、または国外に提供する場合には、同法に基づく許可が必要です。

Windows は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。また、その他の製品名や社名などは、一般に商標または登録商標です。

Copyright　2008 - 2012　LAPIS Semiconductor Co., Ltd.

ラピスセミコンダクタ株式会社

〒193-8550 東京都八王子市東浅川町 550 番地 1

http://www.lapis-semi.com/jp/

目次

はじめに 1-1

MACU8 アセンブラパッケージについて 1

必要なシステム 2

このマニュアルについて 3

表記法 5

1 序論

1.1 プログラムの開発の流れ 1-1

1.2 DCLファイル 1-3

1.2.1 マイクロコントローラの識別情報 1-3

1.2.2 使用可能なROMウィンドウ領域の範囲 1-3

1.2.3 使用可能なメモリ空間の範囲 1-3

1.2.4 SFR領域の範囲に許されるアクセス 1-4

1.2.5 アドレスを表す予約語 1-4

1.2.6 使用可能な命令 1-4

1.3 ファイル指定 1-5

1.4 環境変数 1-6

1.5 各ソフトウェアの使い方 1-7

1.5.1 プログラムをアセンブルする 1-7

1.5.2 オブジェクトファイルをライブラリファイルに登録する 1-7

1.5.3 複数のオブジェクトファイルをリンクする 1-8

1.5.4 オブジェクトファイルを変換する 1-8

1.5.5 Cソースレベルデバッグ情報の作成 1-9

1.5.6 アセンブリレベルデバッグ情報の作成 1-9

2 プログラミングの基礎知識

2.1 プログラムの作成 2-1

2.1.1 プログラムの記述 2-1

2.1.1.1 プログラムの構成 2-2

2.1.1.2 プログラムの最初に記述すること 2-3

2.1.1.3 リセットベクタの定義 2-4

2.1.1.4 プログラムの終了の指定 2-4

2.2 メモリ空間 2-5

2.2.1 メモリ空間の概要 2-5

2.2.2 特殊領域 2-6

2.2.2.1 ベクタ領域 2-6

2.2.2.2 ROMウィンドウ領域 2-7
2.2.2.3 SFR領域 2-7
2.3 アドレス空間 2-8
2.4 論理セグメント 2-9
2.4.1 論理セグメントの記述 2-9
2.4.2 論理セグメントに記述するソースステートメント 2-10
2.4.3 アブソリュートセグメントとリロケータブルセグメント 2-11
2.4.3.1 アブソリュートセグメント 2-11
2.4.3.2 リロケータブルセグメント 2-13
2.4.4 物理セグメント属性 2-15
2.4.5 ユーセージタイプとセグメントタイプ 2-15
2.4.6 セグメントの割り付け可能なアドレス範囲 2-16
2.4.7 特殊なリロケータブルセグメント 2-17
2.4.7.1 スタックセグメント 2-17
2.4.7.2 ダイナミックセグメント 2-18
2.5 ロケーションカウンタ 2-20
2.5.1 ロケーションカウンタの初期化 2-20
2.5.1.1 アブソリュートセグメントのロケーションカウンタの初期化 2-20
2.5.1.2 リロケータブルセグメントのロケーションカウンタの初期化 2-20
2.5.2 ロケーションカウンタの値の変化 2-20
2.5.3 ロケーションカウンタの値の参照 2-21
2.6 メモリモデル 2-22
2.7 データモデル 2-23
2.8 ROMウィンドウ領域の範囲指定について 2-26

3 プログラムの構成要素

3.1 プログラムの要素 3-1
3.1.1 文字セット 3-1
3.1.1.1 英字，数字，アンダスコア，疑問符，ドル記号 3-1
3.1.1.2 空白文字 3-1
3.1.1.3 改行コード，復帰コード 3-2
3.1.1.4 特殊文字 3-2
3.1.1.5 演算子 3-2
3.1.1.6 エスケープシーケンス 3-3
3.1.1.7 全角文字 3-4
3.1.2 定数 3-5
3.1.2.1 整定数 3-5
3.1.2.2 アドレス定数 3-6
3.1.2.3 文字定数 3-7
3.1.2.4 文字列定数 3-8

3.1.3 シンボル 3-9
3.1.3.1 ユーザシンボル 3-9
3.1.3.2 ローカルシンボルとパブリックシンボル 3-14
3.1.3.3 ユーザシンボルの参照 3-15
3.1.3.4 複数のソースファイルからのユーザシンボルの参照 3-16
3.1.3.5 マクロシンボル 3-16
3.1.3.6 予約語 3-17
3.1.4 ロケーションカウンタ記号 3-19
3.2 演算子と式 3-20
3.2.1 式の基本的な考え方 3-20
3.2.1.1 式が属性を持つ意味 3-20
3.2.1.2 アセンブル時に値が決まらない式 3-21
3.2.2 演算子 3-22
3.2.2.1 算術演算子 3-23
3.2.2.2 論理演算子 3-23
3.2.2.3 ビット論理演算子 3-24
3.2.2.4 関係演算子 3-24
3.2.2.5 ドット演算子 3-25
3.2.2.6 アドレス演算子 3-26
3.2.2.7 特殊演算子 3-27
3.2.3 式の種類 3-30
3.2.3.1 定数式 3-32
3.2.3.2 単純式 3-33
3.2.3.3 一般式 3-34
3.2.3.4 式の記述の制限 3-36
3.2.4 式の評価 3-38
3.2.4.1 演算子の優先順位 3-38
3.2.4.2 式の持つ数値の評価 3-39
3.2.4.3 式の属性の評価 3-39

4 アドレッシングと命令

4.1 アドレッシングの書式 4-1
4.1.1 表記について 4-1
4.1.2 レジスタアドレッシング 4-2
4.1.3 メモリアドレッシング 4-3
4.1.3.1 レジスタ間接アドレッシング 4-3
4.1.3.2 ダイレクトアドレッシング 4-8
4.1.4 即値アドレッシング 4-10
4.1.5 プログラムメモリアドレッシング 4-11
4.2 命令一覧 4-12
4.2.1 演算命令 4-12

4.2.2 シフト命令 4-12
4.2.3 ロード／ストア命令....4-13
4.2.4 コントロールレジスタアクセス命令....4-14
4.2.5 PUSH/POP命令 4-14
4.2.6 コプロセッサ転送命令 4-15
4.2.7 EAレジスタ転送命令....4-15
4.2.8 ALU命令....4-15
4.2.9 ビットアクセス命令 4-16
4.2.10 PSWアクセス命令 4-16
4.2.11 条件相対分岐命令 4-16
4.2.12 符号拡張命令....4-17
4.2.13 ソフトウェア割り込み命令 4-17
4.2.14 分岐命令....4-17
4.2.15 乗除算命令 4-17
4.2.16 その他....4-17

5 擬似命令の詳細

5.1 アセンブラ初期設定擬似命令 5-1
5.1.1 TYPE擬似命令 5-1
5.1.2 MODEL擬似命令 5-2
5.1.3 ROMWINDOW擬似命令....5-3
5.1.4 NOROMWIN擬似命令 5-4
5.2 プログラム終了宣言擬似命令....5-5
5.2.1 END擬似命令 5-5
5.3 シンボル定義擬似命令....5-6
5.3.1 EQU擬似命令 5-6
5.3.2 SET 擬似命令....5-7
5.3.3 CODE擬似命令 5-8
5.3.4 TABLE擬似命令 5-8
5.3.5 TBIT擬似命令 5-9
5.3.6 DATA擬似命令 5-10
5.3.7 BIT擬似命令 5-11
5.3.8 NVDATA擬似命令 5-12
5.3.9 NVBIT擬似命令 5-13
5.4 アブソリュートセグメント定義擬似命令....5-15
5.4.1 CSEG擬似命令....5-15
5.4.2 DSEG擬似命令....5-15
5.4.3 BSEG擬似命令....5-16
5.4.4 NVSEG擬似命令 5-16
5.4.5 NVBSEG擬似命令....5-16
5.4.6 TSEG擬似命令 5-17

5.4.7 アブソリュートセグメント定義擬似命令のパラメータ.....5-17

5.5 リロケータブルセグメント定義擬似命令 5-20
5.5.1 SEGMENT擬似命令 5-20
5.5.2 RSEG擬似命令.....5-23
5.5.3 STACKSEG擬似命令 5-24

5.6 アドレス制御擬似命令 5-26
5.6.1 ORG擬似命令.....5-26
5.6.2 ALIGN擬似命令 5-27
5.6.3 DS擬似命令.....5-28
5.6.4 DBIT擬似命令.....5-29

5.7 コード初期化擬似命令 5-30
5.7.1 DB擬似命令.....5-30
5.7.2 DW擬似命令.....5-31
5.7.3 CHKDBDW / NOCHKDBDW擬似命令.....5-32

5.8 最適化擬似命令 5-34
5.8.1 GJMP擬似命令.....5-34
5.8.2 GB*cond*擬似命令.....5-35

5.9 リンケージ制御擬似命令 5-37
5.9.1 複数のファイルによるプログラムの作成.....5-37
5.9.2 PUBLIC擬似命令.....5-37
5.9.3 EXTRN擬似命令.....5-38
5.9.4 COMM擬似命令 5-40
5.9.5 パブリック，イクスターナル，および共有シンボルの使用例.....5-43
5.9.5.1 パブリックシンボルをイクスターナルシンボルで参照する 5-43
5.9.5.2 複数のソースファイルで共通の共有シンボルを使用する 5-43
5.9.5.3 共有シンボルをイクスターナルシンボルで参照する.....5-44
5.9.6 パーシャルセグメントの使用 5-45

5.10 ファイル読み込み擬似命令.....5-48
5.10.1 INCLUDE擬似命令 5-48

5.11 マクロ定義擬似命令.....5-49
5.11.1 DEFINE擬似命令.....5-49

5.12 条件アセンブル擬似命令.....5-50
5.12.1 IF擬似命令.....5-50
5.12.2 IFDEF擬似命令 5-51
5.12.3 IFNDEF擬似命令.....5-52

5.13 Cデバッグ情報擬似命令 5-54
5.13.1 CFILE擬似命令.....5-54
5.13.2 CFUNCTION / CFUNCTIONEND擬似命令 5-54
5.13.3 CARGUMENT擬似命令.....5-54

5.13.4 CBLOCK / CBLOCKEND擬似命令 5-55
5.13.5 CLABEL擬似命令 5-55
5.13.6 CLINE / CLINEA擬似命令 5-55
5.13.7 CGLOBAL擬似命令 5-55
5.13.8 CSGLOBAL擬似命令 5-56
5.13.9 CLOCAL擬似命令 5-56
5.13.10 CSLOCAL擬似命令 5-56
5.13.11 CSTRUCTTAG / CSTRUCTMEM擬似命令 5-57
5.13.12 CUNIONTAG / CUNIONMEM擬似命令 5-57
5.13.13 CENUMTAG / CENUMMEM擬似命令 5-57
5.13.14 CTYPEDEF擬似命令 5-58
5.13.15 CVERSION擬似命令 5-58
5.13.16 CRET擬似命令 5-58

5.14 エミュレーションライブラリ指定擬似命令 5-59
5.14.1 FASTFLOAT擬似命令 5-59

5.15 リスティング制御擬似命令 5-60
5.15.1 OBJ / NOOBJ擬似命令 5-60
5.15.2 PRN / NOPRN擬似命令 5-60
5.15.3 ERR / NOERR擬似命令 5-61
5.15.4 DEBUG / NODEBUG擬似命令 5-62
5.15.5 LIST / NOLIST擬似命令 5-62
5.15.6 SYM / NOSYM擬似命令 5-63
5.15.7 REF / NOREF擬似命令 5-64
5.15.8 PAGE擬似命令 5-64
5.15.8.1 オペランドなしのPAGE擬似命令 5-64
5.15.8.2 オペランド付きのPAGE擬似命令 5-65
5.15.9 DATE擬似命令 5-65
5.15.10 TITLE擬似命令 5-66
5.15.11 TAB擬似命令 5-66

5.16 データアクセス制御擬似命令 5-67
5.16.1 NOFAR擬似命令 5-67

6 RASU8

6.1 概要 6-1

6.2 ファイル指定のデフォルト 6-2

6.3 RASU8 の操作方法 6-3

6.4 オプション定義ファイルによるオプションの指定 6-4
6.4.1 オプション定義ファイルの指定方法 6-4
6.4.2 オプション定義ファイルの書式 6-5

6.5 オプション 6-6

6.5.1 オプション一覧 6-6
6.5.2 各オプションの機能 6-10
6.5.2.1 /MS, /ML 6-10
6.5.2.2 /DN, /DF 6-10
6.5.2.3 /CD, /NCD 6-11
6.5.2.4 /SL 6-12
6.5.2.5 /W, /NW 6-13
6.5.2.6 /I 6-14
6.5.2.7 /DEF 6-14
6.5.2.8 /KE, /KEUC 6-15
6.5.2.9 /G 6-15
6.5.2.10 /PR, /NPR 6-17
6.5.2.11 /A 6-18
6.5.2.12 /L, /NL 6-19
6.5.2.13 /S, /NS 6-20
6.5.2.14 /R, /NR 6-20
6.5.2.15 /PW, /NPW 6-22
6.5.2.16 /PL, /NPL 6-23
6.5.2.17 /T 6-24
6.5.2.18 /O, /NO 6-25
6.5.2.19 /SD 6-25
6.5.2.20 /D, /ND 6-26
6.5.2.21 /E, /NE 6-27
6.5.2.22 /X 6-28
6.5.2.23 /BRAM, /BROM, /BNVRAM, /BNVRAMP 6-28
6.5.2.24 /ZC 6-29
6.6 終了コード 6-30
6.7 プリントファイル 6-31
6.7.1 アセンブリリストの読み方 6-32
6.7.2 クロスリファレンスリストの読み方 6-36
6.7.3 シンボルリストの読み方 6-36
6.7.3.1 シンボルインフォメーション 6-36
6.7.3.2 セグメントインフォメーション 6-39
6.7.4 終了メッセージの読み方 6-41
6.8 EXTRN宣言ファイル 6-42
6.8.1 EXTRN宣言ファイルとは 6-42
6.8.2 EXTRN宣言ファイルの作り方および使い方 6-42
6.9 エラーメッセージ 6-44
6.9.1 エラーメッセージの形式 6-45
6.9.2 エラーメッセージ一覧 6-45
6.9.2.1 フェイタルエラーメッセージ 6-45

6.9.2.2 アセンブルエラーメッセージ 6-50
6.9.2.3 ワーニングメッセージ 6-58
6.9.2.4 内部処理エラーメッセージ 6-61

7 RLU8

7.1 概要 7-1

7.2 RLU8 の操作方法 7-2
7.2.1 コマンドラインの書式 7-2
7.2.1.1 *object_files*フィールド 7-3
7.2.1.2 *absolute_file*フィールド 7-4
7.2.1.3 *map_file*フィールド 7-4
7.2.1.4 *libraries*フィールド 7-5
7.2.1.5 コマンドの例 7-6
7.2.2 実行方法 7-6
7.2.2.1 プロンプトで入力する方法 7-7
7.2.2.2 応答ファイルによる入力の指定 7-8

7.3 処理状態を示すメッセージ 7-10

7.4 終了コード 7-11

7.5 オプション 7-12
7.5.1 オプションの指定方法 7-12
7.5.1.1 構文 7-12
7.5.1.2 指定位置 7-12
7.5.1.3 名前の引数 7-12
7.5.1.4 アドレスの引数 7-13
7.5.2 オプション一覧 7-13
7.5.3 各オプションの機能 7-14
7.5.3.1 /D, /ND 7-14
7.5.3.2 /S, /NS 7-15
7.5.3.3 /CODE, /TABLE, /DATA, /BIT, /NVDATA, /NVBIT 7-15
7.5.3.4 /ORDER 7-18
7.5.3.5 /ROM, /RAM, /NVRAM, /NVRAMP 7-19
7.5.3.6 /CC 7-20
7.5.3.7 /SD, /NSD 7-21
7.5.3.8 /STACK 7-21
7.5.3.9 /A, /NA 7-21
7.5.3.10 /COMB 7-22
7.5.3.11 /EXC 7-23
7.5.3.12 /ROMWIN 7-23
7.5.3.13 /PDIF 7-23
7.5.3.14 /OVERLAY 7-23
7.5.3.15 /LA 7-24

7.5.3.16 /CP 7-24
7.6 リンク処理 7-26
7.6.1 モジュールのマッチングチェック 7-26
7.6.1.1 CPUコアのマッチングチェック 7-26
7.6.1.2 マイクロコントローラ名のマッチングチェック 7-26
7.6.1.3 メモリモデルのマッチングチェック 7-27
7.6.1.4 メモリ情報のマッチングチェック 7-27
7.6.1.5 ROMWINDOW属性のマッチングチェック 7-27
7.6.2 グローバルシンボルの対応付け 7-28
7.6.3 セグメントの結合 7-28
7.6.4 共有シンボルの結合 7-30
7.6.5 セグメントの参照関係のチェック 7-30
7.6.6 セグメントの割り付け 7-31
7.6.6.1 割り付け空間と領域 7-31
7.6.6.2 擬似セグメント 7-32
7.6.6.3 割り付けの優先度 7-33
7.6.7 フィックスアップ処理 7-35
7.7 マップファイル 7-36
7.8 エラーメッセージ 7-43
7.8.1 エラーメッセージの形式 7-43
7.9 エラーメッセージ一覧 7-45
7.9.1 コマンドラインエラーメッセージ 7-45
7.9.2 フェイタルエラーメッセージ 7-46
7.9.3 エラーメッセージ 7-49
7.9.4 ワーニングメッセージ 7-52
7.9.5 内部処理エラーメッセージ 7-56

8 LIBU8

8.1 概要 8-1
8.1.1 LIBU8 の機能 8-1
8.1.2 LIBU8 を使う利点 8-1
8.1.3 ファイル名とモジュール名の違い 8-1
8.2 LIBU8 の実行 8-3
8.2.1 コマンドラインによるライブラリ操作 8-3
8.2.1.1 *library_file*フィールド 8-4
8.2.1.2 *operations*フィールド 8-4
8.2.1.3 *list_file*フィールド 8-5
8.2.1.4 *output_library_file*フィールド 8-6
8.2.2 プロンプトを使っての実行 8-6
8.2.3 コマンドラインとプロンプトを組み合わせた使い方 8-8

8.2.4 応答ファイルを利用した使い方 8-8
8.3 出力されるメッセージのリダイレクト 8-10
8.4 終了コード 8-11
8.5 LIBU8 の操作 8-12
8.5.1 新規ライブラリの作成 8-12
8.5.2 モジュールの追加 8-14
8.5.3 ライブラリファイルの追加 8-15
8.5.4 モジュールの削除 8-16
8.5.5 モジュールの置換 8-17
8.5.6 モジュールのコピー 8-19
8.5.7 モジュールの抽出 8-20
8.5.8 操作の優先度 8-21
8.5.9 テンポラリファイル 8-21
8.6 リストファイルの形式 8-22
8.7 エラーメッセージ 8-24
8.7.1 エラーメッセージの書式 8-24
8.7.2 フェイタルエラーメッセージ 8-25
8.7.3 エラーメッセージ 8-27
8.7.4 ワーニングメッセージ 8-28

9 OHU8

9.1 概要 9-1
9.2 OHU8 の操作方法 9-4
9.2.1 コマンドラインを使用した起動方法 9-4
9.2.2 プロンプトを使用した起動方法 9-6
9.2.3 応答ファイルを利用した操作方法 9-6
9.3 OHU8 が出力するメッセージ 9-8
9.3.1 起動メッセージ 9-8
9.3.2 終了メッセージ 9-8
9.3.3 終了コード 9-9
9.4 オプション 9-10
9.5 OHU8 で使うファイル 9-12
9.5.1 入力ファイル 9-12
9.5.2 出力ファイル 9-12
9.5.3 インテルHEXファイル 9-12
9.5.3.1 コードセグメントレコード 9-13
9.5.3.2 データレコード 9-14
9.5.3.3 ファイル終了レコード 9-14
9.5.4 モトローラS2 フォーマット 9-15

9.5.4.1 S0 レコード 9-15
9.5.4.2 S2 レコード 9-15
9.5.4.3 S8 レコード 9-16
9.6 デバッグ情報 9-17
9.6.1 デバッグシンボルレコード 9-18
9.6.2 デバッグ情報終了レコード 9-18
9.7 入出力ファイル例 9-19
9.8 テンポラリファイル 9-22
9.9 エラーメッセージ 9-23
9.9.1 エラーメッセージの書式 9-23
10 オーバーレイ機能
10.1 概要 10-1
10.1.1 実配置アドレスと実行時アドレス 10-2
10.1.2 オーバーレイ領域の制限 10-2
10.1.3 オーバーレイユニットの配置可能領域 10-3
10.2 オーバーレイユニットの作成方法 10-4
10.2.1 アブソリュートセグメントで構成する方法 10-4
10.2.2 リロケータブルセグメントで構成する方法 10-5
10.3 オーバーレイローダ 10-8
11 アブソリュートリスティング機能
11.1 概要 11-1
11.2 アブソリュートプリントファイルの作成手順 11-2
11.3 リンク時に指定するオプション 11-4
11.4 再アセンブル時に指定するオプション 11-5
11.5 再アセンブル時のエラー 11-7
11.6 アブソリュートプリントファイルの出力例 11-8
11.7 Fatal Error 11 が発生した場合には 11-9
付録 11-1
付録A　擬似命令一覧 1
付録B　予約語一覧 8

はじめに

MACU8 アセンブラパッケージについて

MACU8 アセンブラパッケージは，8 ビット RISC プロセッサである nX-U8 コアを搭載したマイクロコントローラのアセンブリ言語プログラムを作成するために用意された，ソフトウェアパッケージです。

MACU8 アセンブラパッケージには，次のソフトウェアが含まれています。

RASU8（リロケータブルアセンブラ）

RASU8 は，アセンブリ言語で記述されたソースファイルから，オブジェクトファイルを作成します。オブジェクトファイルには，ソースファイルの記述に対応したオブジェクトコードと，リンクやデバッグなどで必要になる情報が入っています。プリントファイル，エラーファイルも作成します。

RLU8（リンカ）

RLU8 は，1 つ以上のオブジェクトモジュールを結合し，1 つのアブソリュートオブジェクトファイルを作成します。セグメントの割り付け状態とパブリックシンボルを示すマップファイルも作成します。

LIBU8（ライブラリアン）

LIBU8 は，ライブラリファイルの作成と管理を行うためのソフトウェアです。ライブラリファイルは複数のオブジェクトファイルを 1 つにまとめたもので，RLU8 によって使用されます。

OHU8（オブジェクトコンバータ）

OHU8 は，RLU8 または RASU8 が作成したアブソリュートなオブジェクトファイルを，インテル HEX フォーマット形式，またはモトローラ S2 フォーマット形式のファイルに変換します。

必要なシステム

MACU8 アセンブラパッケージの各ソフトウェアを動作させるためには，次の環境が必要です。

ハードウェア　　：IBM-PC / AT / Pentium 互換機およびクローン機

オペレーティングシステム　　：WindowsXP / Vista / 7

MACU8 アセンブラパッケージの各ソフトウェアは，Windows のコマンドプロンプトで動作するコマンドライン型ツールです。

このマニュアルについて

このマニュアルは，MACU8 アセンブラパッケージの各ソフトウェアについて説明しています。

このマニュアルは，ユーザがアセンブリ言語に習熟し，アセンブリ言語のソースファイルを作成，編集できることを前提として記述されています。

各章の概要は次のとおりです。

はじめに

この章です。

1 序論

MACU8 アセンブラパッケージの各ソフトウェアの機能概要と，プログラム開発の概要について説明します。

2 プログラミングの基礎知識

MACU8 アセンブラパッケージを使ってプログラムを開発する際に必要となる基礎知識について説明します。

3 プログラムの構成要素

プログラムを構成する要素について説明します。

4 アドレッシングと命令

アセンブリ言語プログラムで記述できるアドレッシングモードと，それぞれの命令で記述可能なアドレッシングモードについて説明します。

5 擬似命令の詳細

MACU8 アセンブラパッケージで用意している擬似命令について説明します。

6 RASU8

リロケータブルアセンブラ RASU8 の操作方法について説明します。

7 RLU8

リンカ RLU8 の操作方法について説明します。

8 LIBU8

ライブラリアン LIBU8 の操作方法について説明します。

9 OHU8

オブジェクトコンバータ OHU8 の操作方法について説明します。

10 オーバーレイ機能

オーバーレイ機能の使い方について説明します。

11 アブソリュートリスティング機能

アブソリュートリスティング機能の概要，およびアブソリュートプリントファイルの作成方法について説明します。

付録

擬似命令および予約語の一覧表を掲載しています。

表記法

このマニュアルでは，説明を分かりやすくするために，いくつかの記号を使用しています。このマニュアルで使用する記号と意味は次のとおりです。

記号	意味
SAMPLE	この文字は，画面に表示されるメッセージや，コマンドラインの入力例，作成されるリストファイルの例などを示します。
CAPITALS	英大文字であらわされた項目は，表示されているとおりにそのまま入力することを示します。
itarics	斜体で表示された項目は，そのまま文字を入力するのではなく，必要な情報に置き換えて入力することを示します。
[ ]	[ ]の中身は必要に応じて入力する項目です。省略することも可能です。
…	…の直前の項目を必要に応じて繰り返すことができます。
{*choice1*|*choice2*}	中カッコ（{}）の中の縦棒（|）で区切られた項目のうち，どれか１つを選んで入力することを示します。[ ]で囲まれていない限り，必ず１つは入力しなければなりません。
value1～*value2*	*value1* 以上，*value2* 以下の値を示します。
『マニュアル』	『』内に記述したものはマニュアル名を示します。
「参照先」	「」内に記述したものは，参照先を示します。
Ctrl+C	Ctrl キーと C キーを同時に押すことを表します。
PROGRAM ・ ・ ・ PROGRAM	縦に並んだ点は，プログラムの例が一部省略されていることを示します。

このマニュアルで数値の終わりに“H”が付加されている場合，その数値は 16 進数であることを示します。例えば，1000H と記述した場合は 16 進数の 1000（10 進数では 4096）を表します。

1 序論

1.1 プログラムの開発の流れ

ここでは，MACU8 アセンブラパッケージを使って，アセンブリ言語で作成したプログラムを開発するときの作業の流れを説明します。プログラムのデバッグについては，このマニュアルでは説明しませんので，使用するデバッガのマニュアルを参照してください。

図 1-1 にプログラム開発の流れを示します。この図において，四角はファイルを表します。ファイルのデフォルト拡張子が 1 つしかなければ，この中には，その拡張子が示されています。楕円はソフトウェアを表します。ひし形は作業時における判断を示します。

以下の説明は，図中の番号と対応させて読み進んでください。

(1) 市販のテキストエディタを使ってプログラムを記述します。プログラムが記述されたファイルをソースファイル（.ASM ファイル）と呼びます。

(2) RASU8 を使ってソースファイルをアセンブルし，オブジェクトファイル（.OBJ ファイル）を作成します。プリントファイル（.PRN ファイル）も作成します。エラーメッセージをファイルに出力することもできます。

(3) LIBU8 を使ってオブジェクトファイル（.OBJ ファイル）をライブラリファイル（.LIB ファイル）に登録することができます。汎用性のあるプログラムをライブラリファイルに登録することで，プログラムの開発効率を向上させることができます。ライブラリファイルは，RLU8 への入力として使用することができます。リストファイル（.LST ファイル）を作成することもできます。これは，ライブラリに登録されているオブジェクトモジュールとパブリックシンボルの一覧を含むファイルです。

(4) RLU8 を使ってプログラムを構成するすべてのオブジェクトファイルを結合し，1 つのアブソリュートオブジェクトファイル（.ABS ファイル）を作成します。RLU8 はオブジェクトファイル間の外部参照を解決したり，論理セグメントをメモリに割り付けたりします。また，マップファイル（.MAP）も作成します。

(5) OHU8 を使ってオブジェクトファイル（.ABS ファイル）を HEX ファイルに変換します。HEX ファイルの種類とフォーマットについては，「9 OHU8」を参照してください。

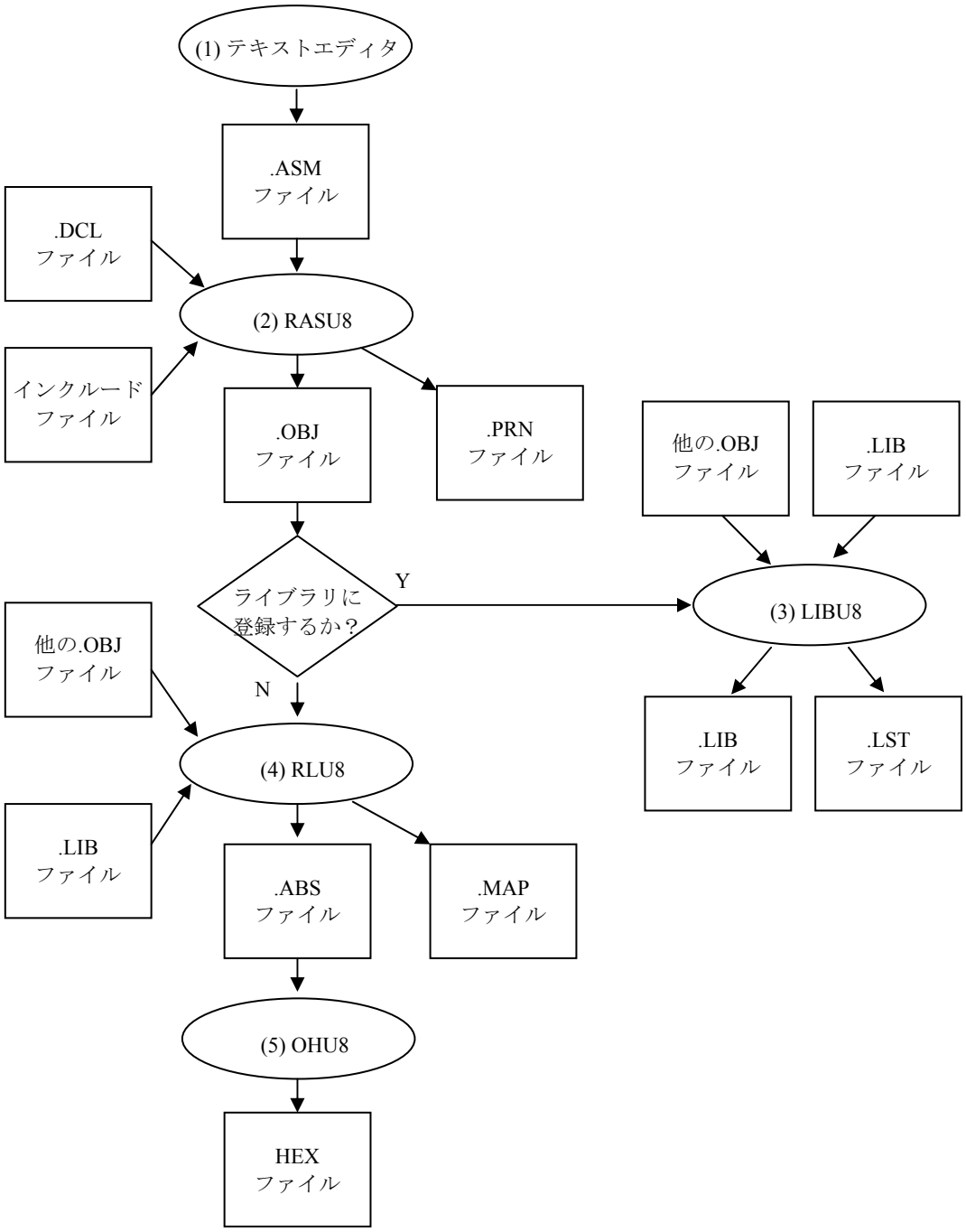


図 1-1 プログラム開発の流れ

1.2 DCL ファイル

MACU8 アセンブラパッケージのソフトウェアは，DCL ファイルと呼ばれる，対象のマイクロコントローラに固有な情報を持ったファイルを読み込みます。MACU8 アセンブラパッケージは，この DCL ファイルを交換することにより，複数のマイクロコントローラに対応できるようになっています。

DCL ファイルは，テキスト形式のファイルです。DCL ファイルの拡張子は，常に“.DCL”です。参照する DCL ファイルの名前は，ソースプログラム中に指定します。DCL ファイルを参照するソフトウェアは RASU8 です。RASU8 は，DCL ファイルの情報をオブジェクトファイルに格納します。RLU8 は，このオブジェクトファイルから DCL ファイルの情報を受け取ります。

DCL ファイルには，RASU8 を初期化するための重要な情報が記述されていますので，DCL ファイルの内容を書き換えるようなことは絶対にしないでください。DCL ファイルの内容を書き換えると，アセンブル処理が正常に行えなくなる可能性があります。

DCL ファイルに記述されている内容を，以下に説明します。

1.2.1 マイクロコントローラの識別情報

マイクロコントローラの名前が記述されています。RLU8 は，リンクを行うときにマイクロコントローラの識別情報を参照して，モジュールのリンクが可能かどうかをチェックします。

1.2.2 使用可能な ROM ウィンドウ領域の範囲

使用可能な ROM ウィンドウ領域の範囲と，その範囲をハードウェア上で設定するのに必要な値が記述されています。RASU8，RLU8 はこの情報に基づいて，指定した ROM ウィンドウ領域の範囲が有効かそうでないか，メモリに対するアクセスが適切なものであるかどうかをチェックします。

1.2.3 使用可能なメモリ空間の範囲

RASU8 は，この情報に基づいてプログラムメモリ空間，およびデータメモリ空間の有効範囲を決定します。そして，対象のメモリをアクセスするオペランドの値をチェックします。

メモリ空間に関する情報には，次の種類があります。

(1) プログラムメモリ空間，およびデータメモリ空間の物理セグメント数

(2) それぞれのメモリ（ROM, RAM, 不揮発性メモリ）が実装されるアドレスの範囲

(3) SFR 領域などの特殊領域の範囲

メモリが実装されるアドレスの範囲についてですが，DCL ファイルにはマイクロコントローラに内蔵されるメモリの範囲しか記述されていません。

RASU8 や RLU8 は，メモリが実装される領域だけを割り付け有効範囲として扱います。

ユーザが外部メモリを実装する場合，その外部メモリの情報を RASU8 や RLU8 に知らせる必

要があるのですが，DCL ファイルを変更することはできませんので，RASU8 と RLU8 では，外部メモリの実装範囲を指定するための起動オプションを用意しています。

オプションは次のとおりです。

RASU8 のオプション	RLU8 のオプション
/BROM(*start_address*,*end_address*)	/ROM(*start_address* , *end_address*)
/BRAM(*start_address*,*end_address*)	/RAM(*start_address* , *end_address*)
/BNVRAM(*start_address*,*end_address*)	/NVRAM(*start_address* , *end_address*)
/BNVRAMP(*start_address*,*end_address*)	/NVRAMP(*start_address* , *end_address*)

それぞれのオプションの詳細については，RASU8，および RLU8 の各オプションの説明を参照してください。

1.2.4 SFR 領域の範囲に許されるアクセス

RASU8 は，この情報に基づいて，SFR 領域へのアクセスをチェックします。

1.2.5 アドレスを表す予約語

RASU8 は，アドレスを表す予約語の値，および予約語の持つユーセージタイプをこの情報から得ます。具体的には，SFR 領域に割り当てられているレジスタ名などが記述されています。

これらは，オペランドでアドレスの代わりに使えます。

1.2.6 使用可能な命令

DCL ファイル中に定義されている命令だけが，対象のマイクロコントローラで使用可能です。DCL ファイル中に定義されていない命令をユーザプログラムで記述していた場合，RASU8 はエラーを表示します。

1.3 ファイル指定

MACU8 アセンブラパッケージのソフトウェアでは，入力や出力としてファイルを指定します。このマニュアルでは，このようなファイル指定を次のように定義します。

＜ドライブ：＞＜ディレクトリ＞＜ベース名＞＜．拡張子＞

また，ドライブとディレクトリの組み合わせをパスと呼びます。

例

```
C:¥U8¥MACU8¥SRC¥TEST.ASM
```

この例におけるファイル指定の各部分は，次のとおりです。

名称	ファイル指定の各部分
パス	C:¥U8¥MACU8¥SRC¥
ドライブ	C:
ディレクトリ	¥U8¥MACU8¥SRC¥
ベース名	TEST
拡張子	.ASM

ファイル指定は，最大 255 文字まで指定できます。ただし，ファイル指定に全角文字，空白文字を使用することはできませんのでご注意ください。

1.4 環境変数

MACU8 アセンブラパッケージのソフトウェアには，環境変数を使用するものがあります。環境変数は必ずしも設定しなければならないというものではありませんので，必要に応じて設定してください。

MACU8 アセンブラパッケージで使用する環境変数は，次のとおりです。

環境変数	説明
DCL	環境変数 DCL は，RASU8 が DCL ファイルをサーチするときに使用されます。RASU8 は，カレントディレクトリ，および RASU8.EXE が存在するディレクトリに DCL ファイルがない場合，DCL ファイルを探すために環境変数 DCL を使用します。 詳細については，「5.1.1 TYPE 擬似命令」を参照してください。
LIBU8	環境変数 LIBU8 は，RLU8 がライブラリファイルをサーチするときに使用されます。RLU8 はカレントディレクトリにライブラリファイルがない場合，ライブラリファイルを探すために環境変数 LIBU8 を使用します。 詳細については，「7.2.1.6 4 *libraries* フィールド」を参照してください。

環境変数の設定の例を以下に示します。

```
SET DCL=C:¥U8¥DCL
SET LIBU8=C:¥U8¥LIB
```

1.5 各ソフトウェアの使い方

ここでは，具体的な例を示しながら，それぞれの用途に応じてソフトウェアをどのように使うのかを説明します。

1.5.1 プログラムをアセンブルする

アセンブリ言語で記述したプログラムは RASU8 でアセンブルします。RASU8 を起動するコマンドラインの書式は，次のとおりです。

RASU8 *source_file*

source_file フィールドには，アセンブル対象のソースファイルを指定します。また，オプションをコマンドラインの任意の位置に指定できます。

RASU8 の使用方法や，オプションについての詳細は「6 RASU8」を参照してください。

例

```
RASU8 MAIN
RASU8 SUB
RASU8 PROC1
RASU8 PROC2
```

この例では，ソースファイル MAIN.ASM，SUB.ASM，PROC1.ASM，および PROC2.ASM をアセンブルし，オブジェクトファイル MAIN.OBJ，SUB.OBJ，PROC1.OBJ，および PROC2.OBJ を作成しています。

1.5.2 オブジェクトファイルをライブラリファイルに登録する

RASU8 で作成したオブジェクトファイルは，LIBU8 を使ってライブラリファイルに登録することができます。汎用性のあるプログラムをライブラリファイルに登録することで，プログラムの開発効率を向上させることができます。LIBU8 を起動するコマンドラインの書式は，次のとおりです。

LIBU8 *library_file* [*operations*] [, [*list_file*][, [*output_library*]]][;]

library_file フィールドには，操作の対象となる入力ライブラリファイルを指定します。*operations* フィールドには，*library_file* フィールドで指定したライブラリに対する操作を指定します。この操作には，追加（+），削除（-），置換（%），コピー（*），および抽出（&）があります。*list_file* フィールドには，リストファイル名を指定します。*output_library* フィールドには，出力ライブラリファイル名を指定します。

LIBU8 の使用方法についての詳細は「8 LIBU8」を参照してください。

例

```
LIBU8 MODULES +PROC1 +PROC2;
```

この例では，RASU8 で作成した 2 つのオブジェクトファイル PROC1.OBJ と PROC2.OBJ をライブラリファイル MODULES.LIB に追加しています。

1.5.3 複数のオブジェクトファイルをリンクする

RASU8 で作成したオブジェクトファイルをリンクするには，RLU8 を使用します。RLU8 を起動するコマンドラインの書式は，次のとおりです。

RLU8 *object_files* [, [*absolute_file*][, [*map_file*][, [*libraries*]]]][;]

object_files フィールドには，リンクするオブジェクトファイルおよびライブラリファイルの名前を指定します。*absolute_file* フィールドには，作成されるオブジェクトファイルの名前を指定します。*map_file* フィールドには，マップファイルの名前を指定します。*libraries* フィールドには，未解決な外部参照を解決するために使用されるライブラリファイルを指定します。コマンドラインの任意の場所にオプションを指定することができます。

RLU8 の使用方法や，オプションについての詳細は「7 RLU8」を参照してください。

例

```
RLU8 MAIN SUB ,TEST,, MODULES
```

この例では，オブジェクトファイル MAIN.OBJ と SUB.OBJ をリンクして，1 つのアブソリュートオブジェクトファイル TEST.ABS を作成しています。*libraries* フィールドにライブラリファイル MODULES.LIB が指定されていますが，未解決な外部参照シンボルが残っていたときに必要なモジュールが MODULES.LIB からサーチされ，リンクされます。

1.5.4 オブジェクトファイルを変換する

RLU8 によって作成されたオブジェクトファイルは，バイナリ形式のファイルになっているため，PROM ライタなどで ROM 化する場合，OHU8 を使用してファイルのフォーマット変換する必要があります。OHU8 を起動するコマンドラインの書式は，次のとおりです。

OHU8 *object_file* [*hex_file*][;]

object_file には，変換するオブジェクトファイルを指定します。*hex_file* には変換した内容を出力する HEX ファイルを指定します。また，コマンドラインの任意の場所にオプションを指定することができます。OHU8 は，変換フォーマットの形式として，インテル HEX フォーマットとモトローラ S2 フォーマットの 2 種類をサポートしています。変換フォーマットの種類を指定するには，オプションを指定します。デフォルトではインテル HEX フォーマットに変換します。

OHU8 の使用方法やオプション，または HEX ファイルの種類やフォーマットの形式について

の詳細は，「9 OHU8」を参照してください。

例

```
OHU8 TEST ;
```

この例では，アブソリュートオブジェクトファイル TEST.ABS をインテル HEX フォーマット形式に変換し，HEX ファイル TEST.HEX を作成しています。

1.5.5 C ソースレベルデバッグ情報の作成

C コンパイラ CCU8 を使用して C 言語でプログラムを開発する場合，C ソースレベルでデバッグを行うためには，C ソースレベルデバッグ情報を作成する必要があります。

C ソースレベルデバッグ情報を作成する方法を以下に示します。

例

```
CCU8 /TM610001 /SD HELLO.C
CCU8 /TM610001 /SD WORLD.C
RASU8 HELLO /SD
RASU8 WORLD /SD
RLU8 HELLO WORLD STARTUP /CC /SD;
```

この例では，C ソースファイル HELLO.C と WORLD.C を，C コンパイラ CCU8 でコンパイルしていますが，このときに/SD オプションを指定すると，CCU8 の出力するアセンブリソースファイル HELLO.ASM と WORLD.ASM に C ソースレベルのデバッグ情報が出力されます。そして，この情報をオブジェクトファイルに出力するために，RASU8 を使ってアセンブルするときに/SD オプションを指定しています。これで，RASU8 の出力するオブジェクトファイルにも C ソースレベルデバッグ情報が出力されます。このようにして作られた C ソースレベルデバッグ情報を含んだオブジェクトファイル HELLO.OBJ と WORLD.OBJ，そしてスタートアップファイル STARTUP.OBJ を，RLU8 を使ってリンクします。このときにも/SD オプションを指定します。このようにすることで，C ソースレベルデバッグ情報が HELLO.ABS に出力されます。

1.5.6 アセンブリレベルデバッグ情報の作成

アセンブリ言語でプログラムを開発する場合，アセンブリレベルでデバッグを行うためには，アセンブリレベルデバッグ情報を作成する必要があります。

アセンブリレベルデバッグ情報を作成する方法を以下に示します。

例

```
RASU8 HELLO /D
RASU8 WORLD /D
RLU8 HELLO WORLD /D ;
OHU8 HELLO /D;
```

アセンブリソースファイル HELLO.ASM と WORLD.ASM を，RASU8 でアセンブルするときに/D オプションを指定すると，HELLO.OBJ と WORLD.OBJ にアセンブリデバッグ情報が出力されます。HELLO.OBJ と WORLD.OBJ を，RLU8 を使ってリンクするときに，/D オプションを指定すると，HELLO.ABS にアセンブリレベルのデバッグ情報が出力されます。さらに，HELLO.ABS を OHU8 でフォーマット変換するときに/D オプションを指定すると，HELLO.HEX にデバッグ情報が出力されます。

2　プログラミングの基礎知識

2.1 プログラムの作成

ここでは，ソースプログラムの作成についての基本的なことがらを説明します。

2.1.1 プログラムの記述

はじめに，プログラムの記述例を示し，その記述例に沿って説明していきます。

```
 1: /************************************
 2:     SAMPLE PROGRAM
 3: ************************************/
 4:     TYPE (M610001)          ; DCL ファイルの指定
 5:     MODEL SMALL, NEAR       ; メモリモデルの指定
 6:     ROMWINDOW 0, 3FFFH      ; ROM ウィンドウ領域の指定
 7:
 8:     REL_CODE SEGMENT CODE   ; セグメントシンボルの定義
 9:     REL_DATA SEGMENT DATA   ; セグメントシンボルの定義
10:     STACK_SEG 100H           ; スタックセグメントの定義
11:
12:     EXTRN DATA: _$$SP       ; イクスターナルシンボルの定義
13:
14:     CSEG AT 0:0H            ; CODE セグメントの開始
15:     DW      _$$SP           ; スタックポインタ初期値
16:     DW      _START          ; リセット入力時の開始アドレス
17:
18:     RSEG    REL_CODE        ; CODE セグメントの開始
19: _START:
20:     MOV     R0,     #1CH
21:     ST      R0,     0FF00H
22:     MOV     ER0,    #00H
23:     MOV     ER2,    ER0
24:     LEA     OFFSET _BUF
25:     ST      XR0,    [EA+]
26:     ST      XR0,    [EA+]
27:     ST      XR0,    [EA+]
28:     ST      XR0,    [EA+]
29:     BAL     $
30:
31:     RSEG    REL_DATA        ; DATA セグメントの開始
32: _BUF:
33:     DS      10H
34:
35:     END                     ; プログラムの終了
```

プログラムの記述例の左端に記述されている行番号は，説明のために記述しているものです。

2.1.1.1 プログラムの構成

プログラムは，nX-U8 のアセンブリ言語で記述された一連のソースステートメントから構成されます。ソースステートメントとは，マイクロコントローラの命令，擬似命令，オペランド，およびコメントを組み合わせたものです。

ソースステートメントには，次の 3 種類があります。

(1) 擬似命令文

(2) 命令文

(3) 空文

それでは，それぞれのソースステートメントについて説明していきます。

2.1.1.1.1 擬似命令文

擬似命令文は，RASU8 の擬似命令を記述するソースステートメントです。擬似命令文の構文は，それぞれの擬似命令ごとに異なります。ただし，セミコロン（;）で始まり，改行コードで終了するコメントは，すべての擬似命令文の最後に記述できます。擬似命令文の終了は，改行コードです。

プログラムの記述例のうち，4 行目から 6 行目，8 行目から 10 行目，12 行目，14 行目から 16 行目，18 行目，31 行目，33 行目，35 行目はすべて擬似命令です。

それぞれの擬似命令文の記述方法については，「5 擬似命令の詳細」を参照してください。

2.1.1.1.2 命令文

命令文は，マイクロコントローラの命令を記述したソースステートメントです。命令文には，4 つのフィールドがあります。フィールドの順序を入れ替えることはできません。命令文の終了は，改行コードです。

命令文の構文は，次のとおりです。

[*label*:]*nemonic* [*operand_field*][;*comment*]

label にはラベルを記述します。ラベルを記述した場合，ラベルの後ろには必ずコロン（:）を記述しなければなりません。ラベルは，そのラベルが所属する論理セグメント内のアドレスを持ちます。

nemonic には，マイクロコントローラの命令を記述します。命令の詳細については，関連するドキュメントを参照してください。

operand_field には，マイクロコントローラの命令が要求するオペランドを記述します。命令が要求するオペランドについては，関連するドキュメントを参照してください。オペランドの構文については，「4 アドレッシングと命令」を参照してください。

comment には，コメントを記述します。コメントはセミコロン（;）で始まり，改行コードまでがコメントとみなされます。

プログラムの記述例のうち，20 行目から 29 行目はすべて命令文です。

2.1.1.1.3 空文

空文は，命令を含まない文です。空文の構文は，次のとおりです。

[*label*:][;*comment*]

プログラムの記述例のうち，7 行目，11 行目，13 行目，17 行目，19 行目，30 行目，32 行目，34 行目はすべて空文です。

2.1.1.1.4 ブロックコメント

ブロックコメントを使用すると，複数の行をコメントにできます。ブロックコメントは，/*で始まり，*/で終了します。RASU8 は，このブロックコメントを無視してアセンブルします。/*と*/の間には，ブロックコメント終了記号（*/）を除いて，改行コードを含むどのような文字の組み合わせも記述できます。したがって，複数の行をブロックコメントにすることができます。ブロックコメントは，ネストすることができます。

```
/* これは
/*すべて*/
コメントになります*/
```

上の例は，すべてコメントになります。

プログラムの記述例のうち，1 行目から 3 行目はブロックコメントです。

2.1.1.2 プログラムの最初に記述すること

プログラムを作成する場合，アセンブラ初期設定用の擬似命令を使用して，対象のプログラムがどのようなマイクロコントローラに対するものなのか，どのようなメモリモデルで作成するのか，ROM ウィンドウ機能を利用するかしないかなどを RASU8 に対して知らせる必要があります。

例の 4 行目から 6 行目は，アセンブラ初期設定用の擬似命令です。

TYPE 擬似命令

TYPE 擬似命令を使用して，対象のマイクロコントローラに対応する DCL ファイルの名前を指定します。この指定は，ファイルごとで必ず行わなければなりません。

MODEL 擬似命令

MODEL 擬似命令を使用して，使用するメモリモデルの種類とデータモデルの種類を指定します。メモリモデルは，必要に応じて指定します。

ROMWINDOW 擬似命令

ROMWINDOW 擬似命令を使用して，ROM ウィンドウ領域の開始アドレスと終了アドレスを指定します。ROM ウィンドウ機能を使用しない場合は，NOROMWIN 擬似命令を指定します。ROM ウィンドウ領域は，必要に応じて指定します。

2.1.1.3 リセットベクタの定義

例の 14 行目から 16 行目までのアブソリュート CODE セグメントの内容について注目すると，物理セグメント#0 の 0 番地から順に，ワード単位で初期化を行っています。この領域は，リセットベクタ領域に相当する領域です。リセットベクタの定義は，いずれかのファイルで必ず記述する必要があります。リセットベクタの定義を記述しなかった場合，正常な動作は保証されません。

2.1.1.4 プログラムの終了の指定

例の 35 行目では，END 擬似命令を記述しています。END 擬似命令は，プログラムがその時点で終了することを RASU8 に知らせます。END 擬似命令が記述されると，その行以降の内容はアセンブルされません。プログラムに END 擬似命令が記述されていない場合，RASU8 はファイルの終わりまでをアセンブルします。

2.2 メモリ空間

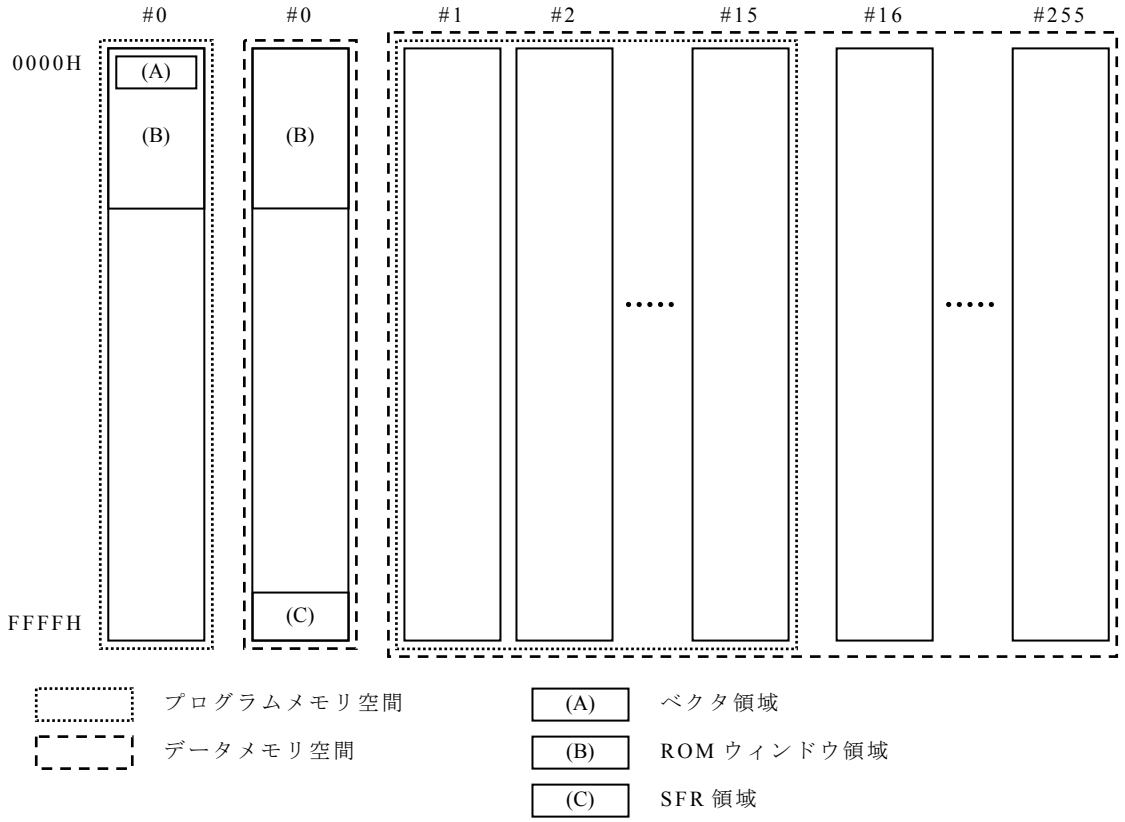
ここでは，nX-U8 コアのサポートするメモリ空間について説明します。

2.2.1 メモリ空間の概要

nX-U8 コアは，最大 1M バイトのプログラムメモリ空間と，最大 16M バイトのデータメモリ空間を持っています。プログラムメモリ空間は，主にプログラム実行に必要な命令コード（プログラムコード）が配置される空間です。データメモリ空間は，主にプログラム実行に必要な初期データや，実行時のデータを一時的に保存する領域が配置される空間です。

それぞれのメモリ空間は，64K バイト単位で区切られた，いくつかの物理セグメントから構成されています。また，物理セグメント#0 のメモリ空間と，物理セグメント#1 以上のメモリ空間とでは，メモリ空間の構造が異なっています。

nX-U8 のメモリ空間の概要を以下に示します。



物理セグメント#0 のメモリ空間は，プログラムメモリ空間とデータメモリ空間がそれぞれ独立した構造になっています。物理セグメント#1 から物理セグメント#15 までのメモリ空間は，プログラムメモリ空間としてもデータメモリ空間としても共通に使用できる構造になっています。物理セグメント#16 から物理セグメント#255 まではデータメモリ空間として使用可能です。

ユーザが物理セグメントを切り替えるためには，セグメントレジスタを操作するプログラムを記述しなければなりません。セグメントレジスタとは，現在の物理セグメントを指定するた

めのレジスタです。セグメントレジスタには，プログラムメモリ空間の物理セグメントを指定するためのコードセグメントレジスタ（CSR）と，データメモリ空間の物理セグメントを指定するためのデータセグメントレジスタ（DSR）が存在します。

CSR は 4 ビットのレジスタで，0 から 15 までの値を表現することができます。DSR は 8 ビットのレジスタで，0 から 255 までの値を表現することができます。したがって，プログラムメモリ空間で表現し得る範囲は 0:0000H から F:FFFFH までの 1M バイト，データメモリ空間で表現し得る範囲は 0:0000H から FF:FFFFH までの 16M バイトとなっています。

補足

このマニュアルでは，完全なアドレスの表現を，物理セグメントの番号（以降，物理セグメントアドレスと表現します）と物理セグメント内のオフセットアドレス（以降，オフセットアドレスと表現します）を用いて，次のように表現します。

physical_segment : *offset_address*

physical_segment は物理セグメントアドレスで，*offset_address* はオフセットアドレスを示します。例えば，“1:2345H”と表現した場合は，物理セグメント#1 のオフセットアドレス 2345H 番地であることを示します。

2.2.2 特殊領域

物理セグメント#0 のプログラムメモリ空間には，特殊領域としてベクタ領域と ROM ウィンドウ領域が存在します。物理セグメント#0 のデータメモリ空間には，特殊領域として ROM ウィンドウ領域と SFR 領域が存在します。

2.2.2.1 ベクタ領域

ベクタ領域は，リセット時や割り込み時に用いられる処理プログラムのエントリアドレス（ベクタ）を格納するための特殊な領域です。ベクタ領域は，さらに次の 3 種類の領域に分類されています。

ベクタ領域の種類	内容
リセットベクタ領域	リセット時のエントリアドレスを格納する領域です。 0 番地にはスタックポインタの初期値を，2 番地にはリセット時のエントリアドレスを必ず設定する必要があります。この設定を行わなかった場合，正常な動作は保証されません。
ハードウェア割り込み領域	ハードウェア割り込み発生時のエントリアドレスを格納する領域です。ハードウェア割り込みを使用しない場合は，この領域にも通常のプログラムコードを配置することができます。
ソフトウェア割り込み領域	ソフトウェア割り込みのエントリアドレスを格納する領域です。ソフトウェア割り込みを使用しない場合は，この領域にも通常のプログラムコードを配置することができます。

ベクタ領域の詳細については，ハードウェアのマニュアルをご参照ください。

2.2.2.2 ROM ウィンドウ領域

ROM ウィンドウ領域は，物理セグメント#0 のプログラムメモリ空間の中で，メモリアクセス命令を用いてアクセスできる領域です。物理セグメント#0 のデータメモリ空間において，内部 RAM がマッピングされていない領域でのみ窓が開き，この窓を通してプログラムメモリ空間の同じアドレスに配置されている読み出し専用のデータ（テーブルデータ）を読み出すことができます。なお，実際に ROM ウィンドウ領域を利用するには，ROM ウィンドウ機能をハードウェア的に制御するための設定が必要です。

プログラムメモリ空間の ROM ウィンドウ領域には，テーブルデータ，およびプログラムコードを配置することができます。

データメモリ空間の ROM ウィンドウ領域は，プログラムメモリ空間への窓として扱われる領域のため，この領域に対して読み書き可能なデータを配置することはできません。

2.2.2.3 SFR 領域

SFR 領域は，周辺機能を制御するための特殊機能レジスタ（SFR）が配置されている領域です。この領域には，データを配置することはできません。

2.3 アドレス空間

アドレス空間は，プログラムコードやデータなどを配置する領域を，RASU8 や RLU8 に分かるように用意した論理的な空間です。

アドレス空間は，後述する論理セグメントの再配置の対象となる空間であり，各メモリアドレッシングの対象となる空間でもあります。

アドレス空間には，次の種類があります。

アドレス空間	説明	アドレス単位
CODE アドレス空間	プログラムメモリ空間。 具体的には，プログラムコードを置くことができる空間です。	バイト
DATA アドレス空間	ROM ウィンドウ領域，および不揮発性メモリが実装される領域を除くデータメモリ空間。 具体的には，RAM データを置くことができる空間です。	バイト
BIT アドレス空間	ROM ウィンドウ領域，および不揮発性メモリが実装される領域を除くデータメモリ空間。 具体的には，RAM データを置くことができる空間です。	ビット
NVDATA アドレス空間	ROM ウィンドウ領域と重なる領域を除くデータメモリ空間上の不揮発性メモリが実装される領域。 具体的には，不揮発性のデータを置くことができる空間です。	バイト
NVBIT アドレス空間	ROM ウィンドウ領域と重なる領域を除くデータメモリ空間上の不揮発性メモリが実装される領域。 具体的には，不揮発性のデータを置くことができる空間です。	ビット
TABLE アドレス空間	プログラムメモリ空間上の ROM ウィンドウ領域，および物理セグメント#1 以上のデータメモリ空間。 具体的には，読み出し専用のデータ（すなわちテーブルデータ）を置くことができる空間です。	バイト

2.4 論理セグメント

nX-U8 には，CODE アドレス空間，DATA アドレス空間，BIT アドレス空間，NVDATA アドレス空間，NVBIT アドレス空間，そして TABLE アドレス空間という 6 つのアドレス空間があります。ユーザがアセンブリ言語でプログラムを記述する場合，プログラムのどの部分がどのアドレス空間に対応するのかを RASU8 と RLU8 に知らせる必要があります。そのために論理セグメントという概念を用います。論理セグメントとは，アドレスが連続したひとまとまりの領域を表しています。

nX-U8 では，すべてのプログラムの記述はこの論理セグメントに所属します。RLU8 は，この論理セグメントをメモリ空間のどこに割り付けるのかを決定します。

論理セグメントには，次の種類があります。

論理セグメント	説明
CODE セグメント	CODE アドレス空間に対応する論理セグメント。
DATA セグメント	DATA アドレス空間に対応する論理セグメント。ただし，SFR 領域を除きます。
BIT セグメント	BIT アドレス空間に対応する論理セグメント。ただし，SFR 領域を除きます。
NVDATA セグメント	NVDATA アドレス空間に対応する論理セグメント。
NVBIT セグメント	NVBIT アドレス空間に対応する論理セグメント。
TABLE セグメント	TABLE アドレス空間に対応する論理セグメント。

2.4.1 論理セグメントの記述

論理セグメントは，アドレス空間におけるひとまとまりの連続した領域です。nX-U8 のアセンブリ言語を使用してプログラムを記述する場合，すべてのソースステートメントは，この論理セグメントに所属します。ソースファイルのどの部分がどの論理セグメントとなるのかは，セグメント定義擬似命令を使用して定義します。

それぞれの論理セグメントを定義するための，セグメント定義擬似命令には，次の種類があります。

論理セグメント	セグメント定義擬似命令 （アブソリュートセグメント）	セグメント定義擬似命令 （リロケータブルセグメント）
CODE	CSEG	RSEG
DATA	DSEG	RSEG
BIT	BSEG	RSEG
NVDATA	NVSEG	RSEG

論理セグメント	セグメント定義擬似命令 （アブソリュートセグメント）	セグメント定義擬似命令 （リロケータブルセグメント）
NVBIT	NVBSEG	RSEG
TABLE	TSEG	RSEG

論理セグメントを定義する擬似命令には，それぞれ 2 種類の擬似命令があります。RSEG 擬似命令については後で説明します。ここでは，CSEG，DSEG，および BSEG 擬似命令を使用した例について説明します。

例

```
DSEG
.
.
.
```
DATA セグメントの範囲

```
BSEG
.
.
.
```
BIT セグメントの範囲

```
CSEG
.
.
.
```
CODE セグメントの範囲

この例では，DSEG 擬似命令を記述して DATA セグメントが開始されています。その後，BSEG 擬似命令の記述によって BIT セグメントが開始されています。最後に，CSEG 擬似命令の記述によって CODE セグメントが開始されています。

このように，論理セグメントの定義は，次に論理セグメントを定義するまで有効です。そして，プログラムの記述における論理セグメントとは，連続したソースステートメントのひとまとまりとなります。したがって，ソースステートメントは，必ずいずれかの論理セグメントに所属していることになります。

論理セグメントは、1 つの物理セグメント内に割り付けられます。複数の物理セグメントにまたがる論理セグメントは，定義できません。また，SFR 領域に論理セグメントを置くことはできません。

2.4.2 論理セグメントに記述するソースステートメント

アセンブリ言語によるプログラムを作成する場合，この論理セグメントを意識しなければなりません。すなわち，CODEアドレス空間に関するソースステートメントは，CODEセグメントの開始擬似命令を記述してから，ソースステートメントを記述しなければなりません。同様に，TABLEアドレス空間，DATAアドレス空間，BITアドレス空間，NVDATAアドレス空間，NVBITアドレス空間に関するソースステートメントも，対応するセグメント開始擬似命令を記述して

からソースステートメントを記述しなければなりません。アドレス空間についての詳しい説明は「2.3 アドレス空間」を参照してください。

各論理セグメントに主に記述するソースステートメントを以下に示します。

論理セグメント	論理セグメントに主に記述するソースステートメント
CODE	マイクロコントローラの命令文。
	指定した数値で初期化する DW 擬似命令文。
	最適な分岐命令に変換される GJMP 擬似命令文，および GB*cond* 擬似命令文。
DATA	バイト単位でデータ用のメモリを確保する DS 擬似命令文。
BIT	ビット単位でデータ用のメモリを確保する DBIT 擬似命令文。
NVDATA	指定した数値で初期化する DB 擬似命令文，または DW 擬似命令文。
	バイト単位でデータ用のメモリを確保する DS 擬似命令文。
NVBIT	ビット単位でデータ用のメモリを確保する DBIT 擬似命令文。
TABLE	指定した数値で初期化する DB 擬似命令文，または DW 擬似命令文。

2.4.3 アブソリュートセグメントとリロケータブルセグメント

それぞれの論理セグメントは，次の 2 種類に分類されます。

(1) アブソリュートセグメント

(2) リロケータブルセグメント

アブソリュートセグメントは，アセンブル時に RASU8 がアドレスを決定できる論理セグメントです。リロケータブルセグメントは，アセンブル時に RASU8 がアドレスを決定できない論理セグメントです。リロケータブルセグメントのアドレスは，RLU8 が決定します。

それぞれの論理セグメントは，擬似命令を使用して定義されます。

2.4.3.1 アブソリュートセグメント

アブソリュートセグメントは，アセンブル時に RASU8 がアドレスを決定できる論理セグメントです。アブソリュートセグメントは，物理セグメントごとに管理されます。アブソリュートセグメントを定義するときに，次の内容を指定できます。

(1) アブソリュートセグメントの先頭アドレス

(2) アブソリュートセグメントの所属する物理セグメントアドレス

これらの内容を指定しない場合，以前に指定したアブソリュートセグメントの指定を引き継ぎます。どのように引き継ぐのかについては，「5.4 アブソリュートセグメント定義擬似命令」を参照してください。

ひとつのプログラムに同じ物理セグメントアドレスを持つ複数のアブソリュートセグメントの記述がある場合，アブソリュートセグメントの先頭アドレスを指定していなければ，これらのアブソリュートセグメントは，1 つのアブソリュートセグメントになります。

プログラムの先頭から，最初に論理セグメントを定義するまでは，CODE アドレス空間に所属するアブソリュートセグメントになります。したがって，プログラムに論理セグメントを定義しない場合は，プログラムは，すべてこのセグメントになります。この場合，このセグメントは，物理セグメントアドレスは#0，オフセットアドレスは 0 で開始されます。

アブソリュートセグメントは，所属するアドレス空間によって次のように呼ばれます。

アブソリュートセグメント	説明
アブソリュート CODE セグメント	CODE アドレス空間に所属するアブソリュートセグメント
アブソリュート DATA セグメント	DATA アドレス空間に所属するアブソリュートセグメント
アブソリュート BIT セグメント	BIT アドレス空間に所属するアブソリュートセグメント
アブソリュート NVDATA セグメント	NVDATA アドレス空間に所属するアブソリュートセグメント
アブソリュート NVBIT セグメント	NVBIT アドレス空間に所属するアブソリュートセグメント
アブソリュート TABLE セグメント	TABLE アドレス空間に所属するアブソリュートセグメント

それぞれのアドレス空間に所属するアブソリュートセグメントは，次の擬似命令を使用して定義します。

擬似命令	説明
CSEG	アブソリュート CODE セグメントを定義します。
DESG	アブソリュート DATA セグメントを定義します。
BSEG	アブソリュート BIT セグメントを定義します。
NVSEG	アブソリュート NVDATA セグメントを定義します。
NVBSEG	アブソリュート NVBIT セグメントを定義します。
TSEG	アブソリュート TABLE セグメントを定義します。

例

```
        CSEG    #1 AT 100H  ;アブソリュートCODEセグメントの定義(1)
        NOP

        DSEG    #0          ;アブソリュートDATAセグメントの定義(1)
LABEL7: DS      2

        CSEG    #1          ;アブソリュートCODEセグメントの定義(2)
        MOV     ER0, #0

        DSEG                ;アブソリュートDATAセグメントの定義(2)
LABEL8: DS      2
```

この例では，4つのアブソリュートセグメントを定義しています。アブソリュートCODEセグメントの定義(1)では，先頭アドレスを100H，所属する物理セグメントアドレスを#1と指定しています。アブソリュートCODEセグメントの定義(2)では，先頭アドレスを指定しないで，所属する物理セグメントアドレスだけを#1としています。先頭アドレスを指定していないので，同じ物理セグメントに所属するアブソリュートCODEセグメント(1)のアドレスを引き継ぎます。すなわち，NOP命令の次の番地に“MOV ER0, #0”命令が置かれることになります。

アブソリュートDATAセグメントの定義(1)では，先頭アドレスを指定しないで，所属する物理セグメントアドレスだけを#0としています。アブソリュートDATAセグメントの定義(2)では，先頭アドレスも所属する物理セグメントアドレスも指定していません。したがって，このセグメントは，アブソリュートDATAセグメント(1)のアドレスを引き継ぎます。

2.4.3.2 リロケータブルセグメント

リロケータブルセグメントは，アセンブル時にRASU8がアドレスを決定できない論理セグメントです。RLU8がアドレスを決定します。RASU8はリロケータブルセグメントをセグメントシンボルで管理します。1つのソースファイルに同じセグメントシンボルを使用して定義した，複数のリロケータブルセグメントがある場合，これらのリロケータブルセグメントは，連続してメモリに割り付けられます。先に定義したリロケータブルセグメントに続いて，次に定義したリロケータブルセグメントが割り付けられます。

異なるソースファイル中に，同じセグメントシンボルを使用して定義したリロケータブルセグメントがある場合，RLU8は，これらのリロケータブルセグメントを結合してメモリに割り付けます。セグメントシンボルの表す値は，結合されたセグメントの先頭アドレスです。このRLU8で結合されるリロケータブルセグメントを，それぞれパーシャルセグメントと呼びます。リロケータブルセグメントのアドレスの決定と結合順序については，「7.65.3 セグメントの結合」を参照してください。

リロケータブルセグメントは，所属するアドレス空間によって，次のように呼ばれます。

リロケータブルセグメント	説明
リロケータブル CODE セグメント	CODE アドレス空間に所属するリロケータブルセグメント
リロケータブル DATA セグメント	DATA アドレス空間に所属するリロケータブルセグメント
リロケータブル BIT セグメント	BIT アドレス空間に所属するリロケータブルセグメント
リロケータブル NVDATA セグメント	NVDATA アドレス空間に所属するリロケータブルセグメント
リロケータブル NVBIT セグメント	NVBIT アドレス空間に所属するリロケータブルセグメント
リロケータブル TABLE セグメント	TABLE アドレス空間に所属するリロケータブルセグメント

リロケータブルセグメントを定義する擬似命令は，次のとおりです。

擬似命令	説明
RSEG	リロケータブルセグメントを定義します。

RSEG のオペランドには，セグメントシンボルを指定します。セグメントシンボルは，SEGMENT 擬似命令を使用して定義されます。このセグメントシンボルを定義するときに，リロケータブルセグメントの所属するアドレス空間の種類を指定します。

例

```
CODESEG2    SEGMENT CODE    #1   ;セグメントシンボル(CODESEG2)の定義
DATASEG2    SEGMENT DATA         ;セグメントシンボル(DATASEG2)の定義
            RSEG    CODESEG2     ;リロケータブル CODE セグメントの定義
            NOP
            RSEG    DATASEG2     ;リロケータブル DATA セグメントの定義
LABEL9:     DS      2
            RSEG    CODESEG2     ;リロケータブル CODE セグメントの定義
            MOV     ER0, #0
```

この例では，2 つのセグメントシンボル(CODESEG2 と DATASEG2)を定義し，これらを使用して，リロケータブルセグメントを定義しています。セグメントシンボルは，SEGMENT 擬似命令を使用して定義されています。リロケータブルセグメントは，RSEG 擬似命令を使用して定義されています。セグメントシンボルを定義するときに，そのセグメントシンボルに対応するリロケータブルセグメントをどのアドレス空間に割り付けるのかを指定します。また，割り付ける物理セグメントも指定できます。

この例では、セグメントシンボル CODESEG2 を定義するときに，CODE アドレス空間に割り付けることと物理セグメント#1 に割り付けることを指定しています。また，セグメントシンボル DATASEG2 を定義するときに，DATA アドレス空間に割り付けることを指定しています。どの物理セグメントに割り付けるのかは指定していません。

また，この例では，ひとつのセグメントシンボル CODESEG2 を使用して，リロケータブル CODE セグメントが 2 つ定義されています。この場合，この 2 つのリロケータブル CODE セグメントは，連続してメモリに割り付けられます。すなわち，NOP 命令の次の番地に“MOV ER0, #0”が置かれることになります。

この例から分かるように，リロケータブルセグメントを割り付ける絶対アドレスを指定することはできません。なぜなら，リロケータブルセグメントとは，その論理セグメントが置かれる絶対アドレスに影響されないプログラムを記述すためのものだからです。リロケータブルセグメントを割り付ける絶対アドレスを指定するには，RLU8 の起動時にオプションで指定します。

2.4.4 物理セグメント属性

各論理セグメントのうち，アブソリュートセグメントに関しては，最初から割り付けるアドレスが決定していますので物理セグメントアドレスは確定しています。しかし，リロケータブルセグメントに関しては，物理セグメントアドレスが確定しているものと，物理セグメントアドレスが未確定のものが存在することになります。物理セグメントの確定状態を表す属性のことを，物理セグメント属性と呼びます。

物理セグメント属性は，次のとおりです。

物理セグメント属性	意味
確定（#*n*）	物理セグメントアドレスが確定していることを示します。
ANY	どの物理セグメントに所属するかが確定していないことを示します。リロケータブルセグメントが持ち得る属性です。どの物理セグメントに割り付けるかは RLU8 が決定します。

論理セグメントに限らず，共有シンボルやイクスターナルシンボルも物理セグメント属性を持ちます。

2.4.5 ユーセージタイプとセグメントタイプ

ユーセージタイプとは，値がどのような利用目的を持っているのかを表す属性です。ユーセージタイプには，次の種類があります。

ユーセージタイプ	意味
NUMBER	数値を持つことを示します。
CODE	CODE アドレス空間のアドレス値を持つことを示します。

ユーセージタイプ	意味
DATA	DATA アドレス空間のアドレス値を持つことを示します。
BIT	BIT アドレス空間のアドレス値を持つことを示します。
NVDATA	NVDATA アドレス空間のアドレス値を持つことを示します。
NVBIT	NVBIT アドレス空間のアドレス値を持つことを示します。
TABLE	TABLE アドレス空間のアドレス値を持つことを示します。
TBIT	TABLE アドレス空間のビットアドレス値を持つことを示します。
NONE	アドレス値を持ちますが，どのアドレス空間であるかは確定していないことを示します。

ユーセージタイプ NUMBER は数値を表しますので，値のユーセージタイプが NUMBER であることと，値が数値型であることは，まったく同じことを意味します。

ユーセージタイプの中で，CODE，DATA，BIT，NVDATA，NVBIT，および TABLE を特にセグメントタイプと呼びます。このマニュアルでは，アドレスの所属するアドレス空間の種類をセグメントタイプと呼びます。また，数値やアドレスの利用目的の種類をユーセージタイプと呼びます。

ユーセージタイプは，アドレッシングの保護のために重要な役割を持っています。多くの命令や擬似命令のオペランドには値を記述することができますが，それぞれのオペランドにどのような種類の値を記述すべきなのかは，あらかじめ決まっています。RASU8 は，オペランドに使用すべきユーセージタイプと，実際に使用されたユーセージタイプを比較してチェックします。

2.4.6 セグメントの割り付け可能なアドレス範囲

各論理セグメントは，そのセグメントの種類，および実装されるメモリの種類によって，割り付け可能なアドレスの範囲が制限されます。RASU8 は，アブソリュートセグメントに対して，対象のセグメントが割り付け可能なアドレス範囲内にあるかどうかをチェックします。RLU8 は，リロケータブルセグメントに対して，対象のセグメントが割り付け可能なアドレスの範囲内に納まるかどうかをチェックしながら，そのセグメントを空き領域に割り付けていきます。

以下に，各セグメントの割り付け可能なアドレス範囲を示します。

論理セグメント	物理セグメントの範囲	オフセットアドレスの範囲
CODE	#0～#15	各物理セグメントに実装される ROM または，不揮発性メモリの範囲内。

論理セグメント	物理セグメントの範囲	オフセットアドレスの範囲
DATA / BIT	#0～#255	各物理セグメントに実装される RAM の範囲内。 ただし，物理セグメント#0 に関しては，ROM ウィンドウ領域，不揮発性メモリ領域，および SFR 領域と重なる範囲を除きます。
NVDATA / NVBIT	#0～#255	各物理セグメントに実装される不揮発性メモリの範囲内。 ただし，物理セグメント#0 に関しては，SFR 領域と ROM ウィンドウ領域と重なる範囲を除きます。
TABLE	#0～#255	各物理セグメントに実装される ROM の範囲内。 ただし，物理セグメント#0 に関しては，ROM ウィンドウ領域の範囲内に限られます。

メモリが実装されていない領域に対しては，RASU8，および RLU8 は，論理セグメントの割り付け対象外領域とみなします。DCL ファイルには，マイクロコントローラに内蔵されるメモリの範囲しか記述されていませんので，外部メモリを実装する場合には，RASU8，または RLU8 の起動時に，外部メモリの実装範囲を指定するオプションを指定してください。

2.4.7 特殊なリロケータブルセグメント

リロケータブルセグメントの中には，通常のリロケータブルセグメントとは扱いの異なる特殊なセグメントがあります。

特殊なリロケータブルセグメントは，次のとおりです。

(1) スタックセグメント

(2) ダイナミックセグメント

それぞれの特殊なリロケータブルセグメントについて，以下に説明します。

2.4.7.1 スタックセグメント

スタックセグメントとは，スタック領域を表す特殊なリロケータブルセグメントです。スタックセグメントは，必ず物理セグメント#0 のデータメモリ空間に配置されます。

スタックセグメントを定義するには，STACKSEG 擬似命令を使用します。STACKSEG 擬似命令のオペランドには，スタックサイズを指定します。

スタックセグメントは，リロケータブルな DATA セグメントとして扱われます。このため，スタックセグメントの開始アドレスと終了アドレスを直接得ることはできません。しかし，シンボルを参照することによって，これらのアドレスを得ることができます。

スタックセグメントの開始アドレスは，$STACK というシンボルを参照することによって得ることができます。$STACK はスタックセグメントの名前であり，スタックセグメントを定義し

たときに，RASU8 によって自動的に生成されます。

スタックポインタの初期値（スタックセグメントの終了アドレスに 1 を加算した値）は，_$$SP というシンボルを参照することによって得ることができます。ただし，_$$SP は自動的に生成されるものではありません。_$$SP を参照するためには，EXTRN 擬似命令を用いて_$$SP を使用することを宣言しておく必要があります。

nX-U8 では，リセットベクタ領域の 0 番地にスタックポインタの初期値を必ず配置しなければなりません。以下に，スタックセグメントの定義，およびスタックポインタの初期値定義の記述例を示します。

```
STACKSEG  200H
EXTRN DATA NEAR:_$$SP
    CSEG  AT  00H
    DW  _$$SP
```

上の例では，200H バイトのサイズを持つスタックセグメントを定義しています。そして，スタックポインタの初期値をリセットベクタ領域の 0 番地に定義しています。例えば，RLU8 によって，スタックセグメントが 0:8000H 番地に配置されたとすると，_$$SP の値は 8200H になります。

スタックセグメントは，通常のリロケータブルセグメントとは異なり，リンク時にスタックセグメントのサイズを変更することができます。スタックセグメントのサイズを変更する場合，RLU8 の起動オプションで“/STACK(*stack_size*) ”を指定します。*stack_size* には，変更するサイズを指定します。

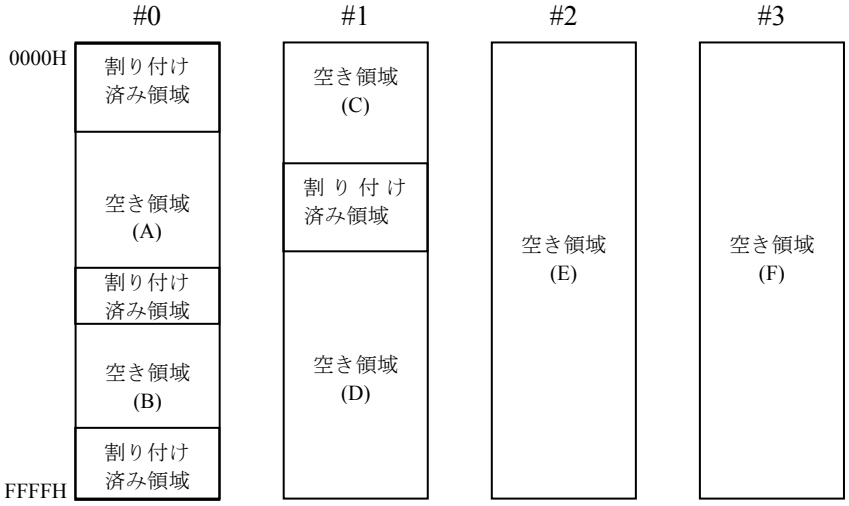
2.4.7.2 ダイナミックセグメント

ダイナミックセグメントとは，アセンブル時にはサイズが決まっていないセグメントで，リンク時にサイズが決定する特殊なリロケータブルセグメントです。このセグメントは，高水準言語における動的メモリ割り付け機能を実現するために用意されているものです。実際には，リンク時にすべての論理セグメントをメモリ空間に割り付けた後，データメモリ空間に残っている最大の空き領域が，ダイナミックセグメントとして割り当てられます。ただし，複数の物理セグメントにまたがって割り当てられることはありません。したがって，一つのダイナミックセグメントのサイズは，最大 64K バイトとなります。ダイナミックセグメントは，物理セグメントごとに設定できます。

以下に，ダイナミックセグメント定義の記述例を示します。

```
dynamic_0   segment data #0 dynamic
dynamic_2   segment data #2 dynamic
dynamic_any segment data any dynamic
```

この例では，dynamic_0 は物理セグメント#0 に配置するダイナミックセグメント，dynamic_2 は物理セグメント#2 に配置するダイナミックセグメント，そして dynamic_any はリンク時に物理セグメントが決定するダイナミックセグメントとして定義されます。



データメモリ空間

たとえば，ダイナミックセグメント以外のすべての論理セグメントが割り付けられた後の空き領域の状態が，上の図のようになっていたとします。空き領域には，すべて RAM が実装されているものと仮定します。このとき，dynamic_0 は空き領域(A)に割り付けられます。dynamic_2 は空き領域(E)に割り付けられます。そして，dynamic_any は空き領域(F)に割り付けられます。

2.5 ロケーションカウンタ

RASU8 は，現在アセンブルしている論理セグメントのアドレスをアセンブル処理の間，常に保持しています。このアドレスを保持するカウンタをロケーションカウンタと呼びます。

リロケータブルセグメントは，それぞれ独自のロケーションカウンタを持っています。また，アブソリュートセグメントのロケーションカウンタは，アドレス空間の物理セグメントごとに用意されています。

2.5.1 ロケーションカウンタの初期化

2.5.1.1 アブソリュートセグメントのロケーションカウンタの初期化

アブソリュートセグメントのロケーションカウンタは，アドレス空間の物理セグメントごとに用意されています。そして，それぞれのロケーションカウンタは RASU8 の起動時に初期化されます。それぞれの物理セグメントに所属するアブソリュートセグメントは，このロケーションカウンタの初期値を開始アドレスとします。

アブソリュートセグメントのロケーションカウンタは，次のとおり初期化されます。

セグメントの種類	アブソリュートセグメントのオフセットアドレス初期値
CODE	該当する物理セグメントに実装される ROM の最小アドレス。
DATA	該当する物理セグメントに実装される RAM の最小アドレス。ただし，物理セグメント#0 の場合は，内蔵 RAM の最小アドレスとなります。
BIT	該当する物理セグメントに実装される RAM の最小アドレス（ビットアドレス）。ただし，物理セグメント#0 の場合は，内蔵 RAM の最小アドレスとなります。
NVDATA	該当する物理セグメントに実装される不揮発性メモリの最小アドレス。
NVBIT	該当する物理セグメントに実装される不揮発性メモリの最小アドレス（ビットアドレス）。
TABLE	該当する物理セグメントに実装される ROM の最小アドレス。

2.5.1.2 リロケータブルセグメントのロケーションカウンタの初期化

リロケータブルセグメントのロケーションカウンタは，それぞれ独自のロケーションカウンタを持っています。セグメントシンボルを定義するときに，対応するロケーションカウンタは 0 に初期化されます。

2.5.2 ロケーションカウンタの値の変化

それぞれのロケーションカウンタの値は，以下に説明するマイクロコントローラの命令，ま

たは擬似命令を使用すると変化します。

アブソリュートセグメント定義時の開始アドレスの指定

アブソリュートセグメントを定義するときに開始アドレスを定義すると，ロケーションカウンタは，そのアドレス値に変更されます。

マイクロコントローラの命令

CODE セグメントのロケーションカウンタの値は，命令のバイト数だけ増加します。

GJMP，GB*cond* 擬似命令

CODE セグメントのロケーションカウンタの値は，変換された分岐命令のバイト数だけ増加します。

DS 擬似命令

DATA，TABLE，NVDATA，または CODE セグメントのロケーションカウンタの値は，オペランドの値だけ増加します。

DBIT 擬似命令

BIT または NVBIT セグメントのロケーションカウンタの値は，オペランドが示す値だけ増加します。

DB, DW 擬似命令

CODE セグメント，TABLE セグメント，または NVDATA セグメントのロケーションカウンタの値は，オペランドの総バイト数だけ増加します。

ORG 擬似命令

セグメントのロケーションカウンタの値は，オペランドの示す値になります。

2.5.3 ロケーションカウンタの値の参照

ドル記号（$）を使用すれば，ソースステートメントが所属する論理セグメントの，現在のロケーションカウンタの値を参照できます。このドル記号（$）をロケーションカウンタ記号と呼びます。

2.6 メモリモデル

nX-U8 では，ハードウェア的にプログラムメモリとしてアクセス可能な物理セグメント数を制御する機能を持っています。RASU8 は，このハードウェア仕様をサポートするために，メモリモデルの概念を持っています。

メモリモデルとは，DCL ファイルの内容とは無関係に，使用可能なプログラムメモリ空間の物理セグメント数，および命令を制限するものです。

メモリモデルの指定は，RASU8 の起動オプション/MS，/ML を指定するか，MODEL 擬似命令を使用して指定します。

この指定は，RASU8 に知らせるためだけのものであり，ハードウェア上のメモリモデル設定を行うものではありませんのでご注意ください。実際には，メモリモデルをハードウェア的に制御するための設定が必要です。

RASU8 の初期設定は，SMALL モデルになっています。

アクセス可能な物理セグメントの範囲は，次のとおりです。

メモリモデル	意味
SMALL	プログラムメモリ空間の物理セグメントは#0 のみに制限されます。 SEGMENT 擬似命令を用いて，物理セグメント属性の指定を行うときに，#1 以上を指定するとエラーになります。 分岐先アドレスに，物理セグメント#1 以上のアドレスを指定するとエラーになります。
LARGE	プログラムメモリ空間の物理セグメントは，#0 から#15 までとなります。

複数のモジュール間で，メモリモデルは一致しなければなりません。異なるメモリモデルのモジュールをリンクしようとした場合には，RLU8 はエラーを表示して終了します。

2.7 データモデル

データメモリ空間に関しては，ハードウェア的にアクセス可能な物理セグメントの数を制御するような機能はありません。メモリアクセス命令を使用するときに DSR を設定するコードを挿入すれば，いつでも物理セグメント#1 以上のデータメモリ空間をアクセスすることができるようになっています。

nX-U8 のアセンブリ言語仕様では，メモリアクセスを行う際に，DSR を設定するコードを挿入するかどうかを RASU8 に知らせるために，アドレッシング指定子と呼ばれるものが用意されています。

アドレッシング指定子を以下に示します。

アドレッシング指定子	意味
NEAR	メモリアクセスの対象を物理セグメント#0 に限定します。DSR を設定するコードは挿入されません。
FAR	メモリアクセスの対象は，物理セグメントを限定しません。すなわち，メモリアクセス命令の直前に DSR を設定するコードを挿入します。

メモリアクセス命令を使用するときに，上記のアドレッシング指定子を明記している場合，RASU8 はその記述に従い命令コードを生成します。

上記のアドレッシング指定子は必ずしも記述する必要はなく，省略することもできます。

そして，アドレッシング指定子を省略したメモリアクセス命令に対し，DSR の設定コードを挿入するかどうかを決定するのが，データモデルです。

データモデルには NEAR モデルと FAR モデルがあります。

NEAR モデル

NEAR モデルでは，RASU8 は，アドレッシング指定子が省略されたメモリアクセス命令のアクセス対象を NEAR とみなします。ただし，メモリアクセス命令のオペランドに記述されたものが，アドレッシング指定子が省略されたものであっても，前方参照を含まない式でその物理セグメント属性が ANY であれば FAR とみなします。

NEAR モデルが指定された場合，RASU8 は，物理セグメントの指定が省略されたセグメントシンボル，共有シンボルに対し，それらが物理セグメント#0 に割り付けられるように，物理セグメント属性#0 を与えます。そして，イクスターナルシンボルに対しては，対応するリロケータブルシンボルが物理セグメント#0 に割り付けられているものと仮定して扱います。したがって，イクスターナルシンボルに対応するリロケータブルシンボルが，物理セグメント#0 に確定していないことがリンク時に判明した場合には，RLU8 はエラーを表示します。

FAR モデル

FAR モデルでは，RASU8 は，アドレッシング指定子が省略されたメモリアクセス命令のアク

セス対象を FAR とみなします。ただし，メモリアクセス命令のオペランドに記述されたものが，アドレッシング指定子が省略されたものであっても，前方参照を含まない式で，その物理セグメント属性が#0 であれば NEAR とみなします。

RASU8 は，物理セグメントの指定が省略されたセグメントシンボル，共有シンボルに対し，物理セグメント属性 ANY を与えます。そして，イクスターナルシンボルに対しては，対応するリロケータブルシンボルの物理セグメント属性は ANY であると解釈します。

NEAR モデルを指定するには，RASU8 の起動オプションで/DN を指定するか，ソースファイル中で“MODEL NEAR”を記述します。FAR モデルを指定するには，RASU8 の起動オプションで/DF を指定するか，ソースファイル中で“MODEL FAR”を記述します。

```
TYPE (M610001)
MODEL NEAR
REL_DATA_SEG SEGMENT DATA
    RSEG    REL_DATA_SEG
VAR1:
    DS      2

    CSEG AT 1000H
    MOV     ER0, #00H
    ST      ER0, VAR1  ; ”ST  ER0, NEAR VAR1”と同等
```

上の例の場合，データモデルを NEAR に設定しているので，セグメント REL_DATA_SEG に対して，物理セグメント#0 に配置されるように物理セグメント属性が与えられます。そして，メモリアクセス命令を用いて REL_DATA_SEG に所属する VAR1 をアクセスする場合，REL_DATA_SEG は必ず物理セグメント#0 に配置されることがわかっているので，NEAR アクセスとみなされます。

```
TYPE (M610001)
MODEL FAR
REL_DATA_SEG SEGMENT DATA
    RSEG    REL_DATA_SEG
VAR1:
    DS      2

    CSEG AT 1000H
    MOV     ER0, #00H
    ST      ER0, VAR1  ; ”ST  ER0, FAR VAR1”と同等
```

上の例の場合，データモデルを FAR に設定しているので，セグメント REL_DATA_SEG に対して，物理セグメント#1 以上にも配置されるように物理セグメント属性 ANY が与えられます。そして，メモリアクセス命令を用いて REL_DATA_SEG に所属する VAR1 をアクセスする場合，REL_DATA_SEG は物理セグメント#1 以上にも配置される可能性があるため，FAR アクセスとみなされます。

データモデルを指定しない場合，デフォルトのデータモデルは，NEAR モデルとなります。

補足

NEAR モデルの場合，メモリアクセス命令において，オペランドに前方参照シンボルを使用すると，エラーになる場合があります。それは次のような場合です。

```
    TYPE (M610001)
    MODEL   NEAR
BWD_FAR EQU 1:8000H
    L       R0, FWD_FAR     ;エラーになる
    L       R1, BWD_FAR     ;L R1, FAR BWD_FAR として扱われる
FWD_FAR EQU 1:8000H
```

FWD_FAR は前方参照シンボルで，BWD_FAR は後方参照シンボルです。FWD_FAR も BWD_FAR も同じアドレス値 1:8000H を定義しています。

NEAR モデルの場合，メモリアクセス命令のオペランドに前方参照シンボルを記述しているものに対しては，RASU8 は NEAR アドレッシングと仮定して処理します。このため，前方参照シンボルが物理セグメント#0 のアドレスを示していなかった場合には，RASU8 はエラーを表示します。

2.8 ROM ウィンドウ領域の範囲指定について

ROM ウィンドウ領域は，ユーザがソースプログラム中で範囲を指定することが可能です。ROMWINDOW 擬似命令は，ROM ウィンドウ機能を利用することをアセンブラに宣言し，その範囲を特定します。NOROMWIN 擬似命令は ROM ウィンドウ機能を利用しないことをアセンブラに知らせます。

実際に ROM ウィンドウ領域を利用するには，ROMWIDOW 擬似命令，または NOROMWIN 擬似命令による ROM ウィンドウ領域の範囲指定のほかに，ROM ウィンドウ機能をハードウェア的に制御するための設定が必要です。

ROMWINDOW 擬似命令，NOROMWIN 擬似命令は，必ずしも記述する必要はありませんが，記述した場合と記述しなかった場合とで制約が異なります。

ROMWINDOW 擬似命令も NOROMWIN 擬似命令も記述しなかった場合

ROM ウィンドウ機能を使用することが指定されたことになりますが，ROM ウィンドウの範囲は決まっていません。このため，物理セグメント#0 のメモリ空間に対して，RASU8 は正しい範囲チェックを行うことができません。物理セグメント#0 のデータメモリ空間において，内蔵 RAM の配置される領域，SFR 領域を除いて，アドレス空間を特定することができません。したがって，アセンブル時において，内蔵 RAM の配置される領域，および SFR 領域以外の領域に対しては，アドレスチェックが行われるたびにワーニングが表示されます。

ROMWINDOW 擬似命令を記述した場合

アセンブル時に ROM ウィンドウ領域の範囲が確定しますので，RASU8 は正しく範囲チェックを行うことができます。ただし，ROM ウィンドウ領域の範囲は，リンクするすべてのモジュールで一致しなければなりません。ROM ウィンドウ領域の範囲が異なるモジュールをリンクしようとした場合には，RLU8 はエラーを表示し，強制終了します。

NOROMWIN 擬似命令を記述した場合

ROM ウィンドウ機能を使用しないという指定のため，RASU8 は物理セグメント#0 には TABLE アドレス空間が存在しないものと認識します。したがって，物理セグメント#0 に TABLE タイプの領域を確保しようとした場合には，RASU8 はエラーを表示します。

NOROMWIN 擬似命令で ROM ウィンドウ機能を使用しないことが指定されたモジュールは，ROM ウィンドウ機能を使用するモジュールとリンクすることはできません。ROM ウィンドウ機能を使用しないモジュールと，ROM ウィンドウ機能を使用するモジュールをリンクしようとした場合には，RLU8 はエラーを表示し，強制終了します。

3 プログラムの構成要素

3.1 プログラムの要素

プログラムの要素とは，RASU8 がプログラムに使用できる文字セット，定数，シンボル，およびロケーションカウンタ記号です。それぞれの要素について以下に説明します。

3.1.1 文字セット

プログラムに使用できる文字には，次の種類があります。

1. 英字， 数字，アンダスコア，疑問符，ドル記号
2. 空白文字
3. 改行コード，復帰コード
4. 特殊文字
5. 演算子
6. エスケープシーケンス
7. 全角文字

ただし，文字定数，文字列定数，およびコメントには，1 バイトのコードで表現されるすべての文字（00H～0FFH）を使用できます。

3.1.1.1 英字，数字，アンダスコア，疑問符，ドル記号

大文字および小文字の英字，10 進数のアラビア数字，アンダスコア（_），疑問符（?），およびドル記号（$）をプログラムに使用できます。これをまとめると次のようになります。

	使用可能な文字
大文字の英字	ABCDEFGHIJKLMNOPQRSTUVWXYZ
小文字の英字	abcdefghijklmnopqrstuvwxyz
10 進数の数字	0123456789
アンダスコア	_
疑問符	?
ドル記号	$

3.1.1.2 空白文字

スペース（20H）およびタブ（09H）は，ソースステートメント中の隣接する要素を区切る役割を持っています。これらを空白文字と呼びます。連続した空白文字と 1 つの空白文字は，同じ意味になります。

3.1.1.3 改行コード，復帰コード

改行コード（0AH）は，ソースステートメントの終了を意味します。復帰コード（0DH）は，文法的に意味を持ちません。RASU8 は，復帰コードを読み飛ばします。

3.1.1.4 特殊文字

特殊文字は，前後の要素に特殊な意味を与える文字です。特殊文字は，次のとおりです。

文字	説明
#	イミディエイトアドレッシングを指定する。
	物理セグメントアドレスを指定する。
,	オペランドの区切りに使用する。
:	ラベルを指定する。
	EXTRN，COMM 擬似命令のユーセージタイプとシンボルの区切りに使用する。
	アドレス定数の，物理セグメントアドレスとオフセットアドレスの区切りに使用する。
;	コメントを開始する。
[ ]	レジスタ間接アドレッシングを指定する。
'	文字定数の開始および終了を指定する。
"	文字列定数の開始および終了を指定する。

3.1.1.5 演算子

演算子は，ひとつの文字または文字の組み合わせで指定されます。それぞれの演算子の機能については，「3.2.2 演算子」を参照してください。演算子には次の種類があります。

(	)	.	!	~	::
+	-	*	/	%	
<<	>>				
<	<=	>	>=	==	!=
&	^	|	&&	||	
BYTE1	BYTE2	BYTE3	BYTE4	WORD1	WORD2
SEG	OFFSET	BPOS	SIZE		
OVL_SEG	OVL_OFFSET	OVL_ADDRESS			

3.1.1.6 エスケープシーケンス

文字列と文字定数には，エスケープシーケンスを使用できます。エスケープシーケンスは，円記号（¥）と文字または数字を組み合わせたものです。エスケープシーケンスには，次の種類があります。

構文	説明
¥nnn	nnn は，1 桁から 3 桁までの 8 進数を表し，値はこの 8 進数の値になる。nnn は 8 進数で 0 から 377 の範囲内でなければならない。
¥xnn , ¥Xnn	nn は，1 桁または 2 桁の 16 進数を表し，値はこの 16 進数の値になる。nnn は 16 進数で 0 から 0FFH の範囲内でなければならない。
¥a	07H に変換される。
¥b	08H に変換される。
¥f	0CH に変換される。
¥n	0AH に変換される。
¥r	0DH に変換される。
¥t	09H に変換される。
¥v	0BH に変換される。
¥*char*	*Cchar* は，a，b，f，n，r，t，v 以外の ASCII 文字を表し，この文字に対応する 1 バイトのコードに変換される。
¥ *J*	*J* は，全角文字を表わし，この全角文字に対応する 2 バイトのコードに変換される。このエスケープシーケンスは，文字列定数には使用できるが，文字定数には使用できない。

例

エスケープシーケンスの例を以下に示します。エスケープシーケンスがどのような値になるのかを 16 進数の整定数で示しています。

エスケープシーケンス	値
¥0	00H
¥47	27H
¥377	0FFH
¥8	38H
¥047	27H
¥x0	00H
¥xA	0AH
¥xFF	0FFH
¥x0F	0FH
¥F	46H
¥a	07H
¥n	0AH
¥あ	82H，0A0H

3.1.1.7 全角文字

漢字などの全角文字をプログラムに使用できます。RASU8 では全角文字は SJIS コードと EUC コードを認識できます。DOS のプロンプトから RASU8 を実行する時に/KE，または/KEUC オプションを指定した場合は EUC コードを全角文字として解釈し，これらのオプションをつけなかった場合は SJIS コードを全角文字として解釈します。どちらの全角文字も 1 文字あたり 2 バイトのコードから構成されており，RASU8 では次の順序で現れた文字を全角文字として認識します。

	SJIS コード	EUC コード （/KE，/KEUC 指定時）
全角文字の第 1 バイト	81H～9FH 0E0H～0FCH	0B0H～0F4H 0A1H～0A8H
全角文字の第 2 バイト	40H～0FCH	0A0H～0FEH

全角文字は次の位置にだけ使用できます。

1. 文字列定数
2. コメント
3. ブロックコメント

3.1.2 定数

定数は，プログラム中で一定の値として使用される数値，文字，および文字列です。本アセンブラパッケージでは整定数，アドレス定数，文字定数，および文字列定数を定数として認識します。以下に各定数についての説明します。

3.1.2.1 整定数

構文

ddigits

*hdigits*H

*ddigits*D

*odigits*O

*odigits*Q

*bdigits*B

説明

整定数は 32 ビットで表現できる整数です。整定数として，2，8，10，および 16 進数が使用できます。整定数の基数を指定するには，数値の後に基数指定子を付けます。基数指定子を省略した場合は，10 進数になります。

hdigits には 16 進数，*ddigits* には 10 進数，*odigits* には 8 進数，そして *bdigits* には 2 進数を記述します。シンボルと区別するため整定数の先頭文字は，0 から 9 の数字でなければなりません。したがって，16 進数の記述において，その先頭文字が英字となる場合には，先頭に数字 0 を付けなければなりません。

プログラムの読みやすさのために，数値を表す文字列中の任意の位置に任意の回数アンダスコア（_）を記述できます。アンダスコアをいくつ記述しても，整定数の意味は変わりません。ただし，整定数の先頭文字にアンダスコア（_）を記述することはできません。

各基数の数値に使用できる文字および各基数の基数指定子は，次のとおりです。

基数指定子	説明	使用可能な文字
H，h	16 進数	0123456789ABCDEFabcdef_
D，d	10 進数	0123456789_
O，o，Q，q	8 進数	01234567_
B，b	2 進数	01_

基数指定子は，大文字，小文字のどちらでも使用できます。また，16 進数の場合，数値には，大文字，小文字のどちらの英字でも使用できます。

例

10 進数 256 を，16 進数，8 進数，および 2 進数で指定する場合は，次の記述になります。

	記述
16 進数	100H
10 進数	256　　256D
8 進数	400O　　400Q
2 進数	100000000B

整定数の先頭に数字 0 をいくつ記述しても，整定数の意味は変わりません。次の記述も 10 進数 256 の記述になります。

	記述
16 進数	00100H
10 進数	0256　　00256D
8 進数	000400O　　00400Q
2 進数	000100000000B

アンダスコア（_）を使用して 10 進数 256 を記述した例を以下に示します。

	記述
16 進数	1_ 00H　　1_00_H
2 進数	1_00000000B　　1_0000_0000_B

3.1.2.2 アドレス定数

構文

integer_constant1: *integer_constant2*

説明

アドレス定数は，アドレス空間上のアドレスを直接表現します。

アドレス定数は，物理セグメントアドレスとオフセットアドレスの 2 つのフィールドで表わされます。*integer_constant1* は，物理セグメントアドレスを表わす整定数で，その値は 0 以上 0FFH 以下でなければなりません。*integer_constant2* は，オフセットアドレスを表わす整定数で，その値は 0 以上 7FFFFH 以下でなければなりません。

integer_constant1 と *integer_constant2* は，コロン（:）で区切ります。コロン（:）の前後に，空白文字を挿入することはできません。

アドレス定数そのものは，アドレス空間の種類までは表現していません。アドレス定数が，どのアドレス空間上のアドレスを示すのかは，アドレス定数を記述するステートメントの命令や擬似命令の種類をもとに，RASU8 が決定します。

例 1

DATA アドレス空間の物理セグメントアドレスが 2，オフセットアドレスが 1000H であるアドレスを表現する例を以下に示します。

```
     L    R0, 2:1000H
D_ADR     DATA 2:1000H
```

例 2

CODE アドレス空間の物理セグメントアドレスが 2，オフセットアドレスが 1000H であるアドレスを表現する例を次に示します。

```
     B    2:1000H
C_ADR     CODE 2:1000H
```

3.1.2.3 文字定数

構文

'*char*'

説明

文字定数は，指定した文字の表す 1 バイトのコードに変換されます。*char* には，1 バイトのコードで表される文字を指定します。また，1 バイトのコードを表すエスケープシーケンスも指定できます。文字およびエスケープシーケンスを省略した場合の値は，0H になります。全角文字は 2 バイトのコードですから指定できません。

例

文字定数の例を以下に示します。文字定数がどのような値になるのかを 16 進数の整定数で示しています。

文字定数	値
''	00H
'A'	41H
'¥0'	00H
'¥47'	27H
'¥377'	0FFH
'¥8'	38H
'¥047'	27H
'¥x0'	00H
'¥xA'	0AH
'¥xFF'	0FFH
'¥x0F'	0FH
'¥F'	46H
'¥''	27H

3.1.2.4 文字列定数

構文

"*characters*"

説明

文字列定数は，コードメモリの初期化などに使用される，二重引用符（"）で囲まれた文字列です。*characters* には，文字列を指定します。文字列中には，エスケープシーケンス，1 バイトのコードで表される文字，および全角文字を記述できます。文字列は，255 バイト以内のコードを表すものでなければなりません。

例

DB 擬似命令のオペランドに文字列定数を使用した例を以下に示します。コメントにコードの値を 16 進数の整定数で示しています。

```
DB   "STRING"       ;53H, 54H, 52H, 49H, 4EH, 47H
DB   "¥111¥222"     ;49H, 92H
DB   "¥x10¥XFF"     ;10H, 0FFH
```

3.1.3 シンボル

シンボルは，次のものを表す名前です。

1. 数値
2. アドレス
3. リロケータブルセグメント
4. 命令
5. 擬似命令
6. レジスタ
7. レジスタアドレス
8. 演算子
9. アドレッシングの種類
10. 命令の特殊オペランド
11. 擬似命令の特殊オペランド
12. マクロ

シンボルには，プログラマが定義するシンボルと，アセンブラがあらかじめ用意しているシンボルがあります。プログラマが定義するシンボルをユーザシンボルと呼び，アセンブラがあらかじめ用意しているシンボルを予約語と呼びます。

数値およびアドレスを表すシンボルには，ユーザシンボルと予約語の両方があります。リロケータブルセグメントを表すシンボルとマクロを表わすシンボルは，すべてユーザシンボルです。それ以外のシンボルは，すべて予約語です。

シンボルは，英字，数字，アンダスコア（_），疑問符（?），およびドル記号（$）から構成される 1 文字以上 32 文字以内の文字列です。先頭文字は，英字，アンダスコア（_），疑問符（?），またはドル記号（$）でなければなりません。32 文字を越えるシンボルの記述があった場合，33 文字目以降の文字は無視されます。SET 擬似命令を使用して定義する場合を除いて，シンボルは，ひとつのソースファイル中で 1 回だけ定義できます。ひとつのソースファイル中に同じ名前のシンボルを 2 回以上定義した場合は，エラーになります。

3.1.3.1 ユーザシンボル

ユーザシンボルは，プログラム中にプログラマが定義するシンボルです。予約語は，ユーザシンボルとして定義できません。

ユーザシンボルを構成する英字の大文字，小文字を区別するかどうかは，RASU8 の/CD オプションと/NCD オプションで制御できます。プログラマが/CD オプションを指定する場合，は大文字と小文字を区別します。プログラマが/NCD オプションを指定する場合，RASU8 は大文字と小文字を区別しません。デフォルトでは，RASU8 は大文字と小文字を区別します。

ユーザシンボルは，ラベル定義の記述，起動オプション，または擬似命令を使用して定義します。それぞれのユーザシンボルの定義方法は，次のとおりです。

ユーザシンボル	定義に使用する擬似命令，起動オプションまたはシンボルの種類
数値を表すユーザシンボル	EQU 擬似命令，SET 擬似命令，EXTRN 擬似命令
アドレスを表すユーザシンボル	ラベル，EQU 擬似命令，SET 擬似命令 CODE 擬似命令， DATA 擬似命令，BIT 擬似命令 NVDATA 擬似命令，NVBIT 擬似命令 TABLE 擬似命令，TBIT 擬似命令 COMM 擬似命令，EXTRN 擬似命令
リロケータブルセグメントを表すユーザシンボル	SEGMENT 擬似命令
マクロを表わすユーザシンボル	DEFINE 擬似命令，RASU8 の/DEF 起動オプション

マクロを表わすシンボルは，テキスト文字列を持ちます。マクロ以外のユーザシンボルは，値を持ちます。この値は，シンボルを定義するときに与えられます。数値を表すシンボルは，その数値を値として持ちます。アドレスを表すシンボルは，そのアドレスを値として持ちます。リロケータブルセグメントを表すシンボルは，そのリロケータブルセグメントが割り付けられる領域の先頭アドレスを値として持ちます。

例 1

数値を表すユーザシンボルの定義例を以下に示します。この例では，SYMEQU1，SYMSET1，および SYM_EXT_NUM1 が，数値を表すユーザシンボルになります。

```
SYMEQU1   EQU  0FFH
SYMSET1   SET  100H
EXTRN     NUMBER:SYM_EXT_NUM1
```

例 2

アドレスを表すユーザシンボルの定義例を以下に示します。

```
EXTINT0   CODE 3H
EXTINT1   EQU  EXTINT0+1
SYMDAT1   DATA 80H
SYMBIT1   BIT  80H.0
SYMSET2   SET  10H+SYMDAT1
SYM_COMM_DAT1  COMM DATA 2
EXTRN     BIT:SYM_EXT_BIT1
```

この例では，EXTINT0，EXTINT1，SYMDAT1，SYMBIT1，SYMSET2，SYM_COMM_DAT1，および SYM_EXT_BIT1 がアドレスを表すユーザシンボルになります。

例 3

リロケータブルセグメントを表すユーザシンボルの定義例を以下に示します。MAINCOD，TABLE1，および DATCOMBUF1 がリロケータブルセグメントを表すユーザシンボルになります。

```
MAINCOD   SEGMENT   CODE #0
     RSEG MAINCOD
     .
     .
     .
TABLE1    SEGMENT   TABLE #1
     RSEG TABLE1
     DW   0000H
     DW   0001H
     .
     .
     .
DATBUF1   SEGMENT   DATA 2 ANY
     RSEG DATBUF1
     DS   2
```

例 4

マクロを表わすユーザシンボルの定義例を以下に示します。この例では，MCRSYM がマクロを表わすユーザシンボルになります。

```
DEFINE    MCRSYM    "MACRO BODY"
```

以下に，ユーザシンボルをいくつかの種類に分類して説明します。このように分類して説明する理由は，プログラム中のどの位置にシンボルを使用できるのかが，このシンボルの種類によって決定するからです。なお，以下の分類には，マクロを表わすシンボルは含まれません。

ユーザシンボルには，アセンブル時に RASU8 が値を決定できるシンボルと出来ないシンボルがあります。アセンブル時に値を決定できるシンボルをアブソリュートシンボルと呼び，アセンブル時に値は決定できないが，RLU8 によって値を決定できるシンボルをリロケータブルシンボルと呼びます。

3.1.3.1.1 アブソリュートシンボル

アブソリュートシンボルには，数値を表すアブソリュートシンボルとアドレスを表すアブソリュートシンボルがあります。アブソリュートシンボルは，次のように定義します。

1. ローカルシンボル定義擬似命令（EQU，SET，CODE，DATA，BIT，NVDATA，NVBIT，TABLE，TBIT）のオペランドに定数式を記述して，シンボルを定義する。
2. アブソリュートセグメントに所属するラベルを定義する。

例

アブソリュートシンボルを定義する例を以下に示します。この例において，SYMCOD，

SYMDAT，SYMBIT，SYMEQU，SYMSET，DATALABEL，および CODELABEL は，アブソリュートシンボルになります。

```
SYMCOD    CODE 100H
SYMDAT    DATA 80H
SYMBIT    BIT  80H.0
SYMEQU    EQU  (-1)
SYMSET    SET  10H+SYMDAT

     DSEG #0 AT 280H
DATALABEL:
     DS   2

     CSEG #0 AT 100H
CODELABEL:
     NOP
```

3.1.3.1.2 リロケータブルシンボル

リロケータブルシンボルには，次の種類があります。

1. 単純リロケータブルシンボル
2. セグメントシンボル
3. 共有シンボル
4. イクスターナルシンボル

次に，それぞれのリロケータブルシンボルについて説明します。

(1) 単純リロケータブルシンボル

単純リロケータブルシンボルは，同じソースファイル中にあるリロケータブルセグメントのアドレスを表すシンボルです。ただし，リロケータブルセグメントを表すシンボル（セグメントシンボル）は，単純リロケータブルシンボルではありません。単純リロケータブルシンボルは，次のように定義します。

1. リロケータブルセグメントに所属するラベルを定義する。
2. ローカルシンボル定義擬似命令（EQU，SET，CODE，DATA，BIT，NVDATA，NVBIT，TABLE）のオペランドに単純リロケータブルシンボルを使用した式を記述して，シンボルを定義する。

単純リロケータブルシンボルは，リロケータブルセグメントに所属します。単純リロケータブルシンボルが所属するリロケータブルセグメントは，以下のようにして決定されます。

a. リロケータブルセグメントに定義するラベルの場合，そのリロケータブルセグメントに所属する。
b. ローカルシンボルを定義する擬似命令（EQU，SET，CODE，DATA，BIT，NVDATA，NVBIT，TABLE，TBIT）を使用して定義する単純リロケータブルシンボルの場合，オペラ

ンドに指定する単純リロケータブルシンボルと同じセグメントに所属する。

例

単純リロケータブルシンボルを定義する例を以下に示します。

```
DATSEG    SEGMENT   DATA
     RSEG DATSEG
LBUF:
     DS   1
HBUF:
     DS   1

CODSEG    SEGMENT   CODE
     RSEG CODSEG
START:
     NOP

SIMCOD    CODE START+1
SIMDAT    DATA LBUF+2
SIMBIT    BIT  (LBUF+2).0
SIMEQU    EQU  HBUF+2
SIMSET    SET  LBUF+4
```

この例において，LBUF，HBUF，START，SIMCOD，SIMDAT，SIMBIT，SIMEQU，および SIMSET は，単純リロケータブルシンボルになります。DATSEG および CODSEG は，リロケータブルセグメントを表すシンボルなので，単純リロケータブルシンボルではありません。

(2) セグメントシンボル

セグメントシンボルは，リロケータブルセグメントを表すシンボルです。SEGMENT 擬似命令を使用してセグメントシンボルを定義します。RSEG 擬似命令のオペランドにセグメントシンボルを記述すると，そのセグメントシンボルを名前とするリロケータブルセグメントを定義できます。この意味で，セグメントシンボルのことをセグメント名と呼ぶこともあります。

セグメントシンボルは，そのリロケータブルセグメントが割り付けられる領域の先頭アドレスを値として持ちます。他のソースファイルのセグメントシンボルを参照するには，SEGMENT 擬似命令を使用して，そのセグメントシンボルを定義することが必要です。

例

```
CHARBUF   SEGMENT   DATA #2
SUB1 SEGMENT   CODE
```

この例では，セグメントシンボル CHARBUF，および SUB1 を定義しています。

(3) 共有シンボル

共有シンボルは，複数のソースファイルが共有するデータ領域の先頭アドレスを表わします。

他のソースファイル中に同じ名前の共有シンボルが宣言されている場合，同じ名前の共有シンボルの宣言の中で最大のサイズが RLU8 によってメモリに割り付けられます。共有シンボルは，その領域の先頭アドレスを表します。

他のソースファイル中に同じ名前の共有シンボルが宣言されていない場合，宣言時に指定するサイズの領域が RLU8 によってメモリに割り付けられます。共有シンボルは，その領域の先頭アドレスを表します。

共有シンボルは，COMM 擬似命令を使用して宣言されます。

例

共有シンボルを宣言する例を以下に示します。

```
COMMSYM   COMM DATA 10H
```

この例において，COMMSYM は，DATA アドレス空間に割り付けられる共有シンボルです。共有シンボルのサイズ指定は，10H バイトです。

(4) イクスターナルシンボル

イクスターナルシンボルを使用して，他のソースファイルよって定義されたパブリックシンボル，および共有シンボルを参照できます。イクスターナルシンボルは，EXTRN擬似命令を使用して宣言します。パブリックシンボルついては「3.1.3.2 ローカルシンボルとパブリックシンボル」を参照してください。

例

イクスターナルシンボルを宣言する例を以下に示します。

```
EXTRN DATA:EXTSYM1 EXTSYM2
EXTRN NUMBER:EXTSYM3 CODE:EXTSYM4
```

この例において，EXTSYM1 および EXTSYM2 は，データアドレスを表すイクスターナルシンボルです。EXTSYM3 は，数値を表すイクスターナルシンボルです。そして，EXTSYM4 は，コードアドレスを表すイクスターナルシンボルです。

3.1.3.2 ローカルシンボルとパブリックシンボル

アブソリュートシンボルと単純リロケータブルシンボルは，そのシンボルに対して何も宣言しないと，そのシンボルが定義されているソースファイル内だけで参照できます。この意味で，アブソリュートシンボルと単純リロケータブルシンボルを合わせて，ローカルシンボルと呼びます。

ローカルシンボルを他のソースファイルから参照する場合には，PUBLIC 擬似命令を使用して，

ローカルシンボルをパブリック宣言します。このパブリック宣言されたローカルシンボルをパブリックシンボルと呼びます。

ローカルシンボルが，そのシンボルが定義されているソースファイル中だけで参照できるということは重要です。一般に，それぞれのソースファイルは，ある独立した機能に対応しています。そして，それぞれのソースファイルは，別々に作成されます。このため，他のソースファイルで同じ名前のローカルシンボルが使用されているとしても，別のシンボルとして扱われなければ，シンボルの名前の管理が大変になってしまいます。

例

パブリックシンボルを宣言する例を以下に示します。

```
PUBLIC    SYMEQU    DATABUF1        ;パブリックシンボルの宣言

SYMEQU    EQU  1

DATSEG2   SEGMENT   DATA
     RSEG DATSEG2
DATABUF1:
     DS   2
```

この例において，SYMEQU はアブソリュートシンボルです。そして，DATABUF1 は単純リロケータブルシンボルです。したがって，この 2 つのシンボルはローカルシンボルになります。この例では，ローカルシンボル SYMEQU および DATABUF1 を PUBLIC 擬似命令を使用してパブリック宣言しています。

3.1.3.3 ユーザシンボルの参照

定義するユーザシンボルを，同じソースファイル中の命令および擬似命令のオペランドから参照できます。

ユーザシンボルの参照には，次の種類があります。

1. 後方参照
   ソースファイル中の記述順序において，参照時点より前に定義したシンボルを参照すること。
2. 前方参照
   ソースファイル中の記述順序において，参照時点より後に定義したシンボルを参照すること。

マイクロコントローラの命令のオペランドで，ユーザシンボルを参照する場合は，後方参照および前方参照の両方とも許されます。

擬似命令のオペランドでユーザシンボルを参照する場合，後方参照は常に許されますが，擬似命令によっては前方参照が許されない場合があります。前方参照が許されない擬似命令のオペランドについては，「5. 擬似命令の詳細」の各擬似命令の説明を参照してください。

例

この例では，2 つのマイクロコントローラの命令（B，MOV）と 2 つの擬似命令（DW，ORG）のオペランドに，後方参照と前方参照の 2 種類のシンボルを記述しています。

```
     MOV  ER0, #FORWARD_SYM
     DW   FORWARD_VALUE

BACKWARD_VALUE EQU  10H
BACKWARD_SYM   EQU  28H
BACKWARD_LABEL:
     B    FORWARD_LABEL
     ORG  FORWARD_VALUE            ;エラー

     ORG  BACKWARD_VALUE
     DW   BACKWARD_VALUE
     MOV  ER0, #BACKWARD_SYM
     B    BACKWARD_LABEL
FORWARD_LABEL:
FORWARD_SYM    EQU  30H
FORWARD_VALUE  EQU  20H
```

後方参照するシンボルは BACKWARD_VALUE，BACKWARD_SYM，および BACKWARD_LABEL です。前方参照するシンボルは，FORWARD_LABEL，FORWARD_SYM，および FORWARD_VALUE です。マイクロコントローラの命令および DW 擬似命令のオペランドには，後方参照と前方参照の両方が許されます。ただし，ORG 擬似命令のオペランドには，前方参照は許されません。したがって，ソースステートメント“ORG　FORWARD_VALUE”は，エラーになります。

3.1.3.4 複数のソースファイルからのユーザシンボルの参照

複数のソースファイルから同じユーザシンボルを参照するには，次のシンボルを使用します。

1.パブリックシンボル

2.イクスターナルシンボル

3.共有シンボル

4.セグメントシンボル

これらのシンボルを使用する方法については，「5.9 リンケージ制御擬似命令」を参照してください。

3.1.3.5 マクロシンボル

マクロシンボルは，他のユーザシンボルと比べて特殊です。

通常ユーザシンボルには，値が与えられますが，マクロシンボルには，テキスト文字列が与えられます。また，あるソースファイルで定義したマクロシンボルを，別なソースファイルで参照するために，マクロシンボルをパブリック宣言することはできません。

マクロシンボルのもっとも大きな特徴は，マクロシンボル自体ではなく，マクロシンボルが与えるテキスト文字列の方が，アセンブリ言語上の意味を持っていることです。

このような理由で，このマニュアルでは，マクロシンボルと他のユーザシンボルは，はっきり区別されています。このマニュアルの中の“ユーザシンボルは・・・”という表現は，ほとんどの場合マクロシンボルを除くシンボルのことを言っています。

3.1.3.6 予約語

予約語は，MACU8 アセンブラパッケージがあらかじめ用意しているシンボルです。予約語には，次の種類があります。

1. 命令
2. 擬似命令
3. レジスタ
4. 演算子
5. SFR シンボル
6. アドレッシング指定子
7. 命令の特殊オペランド
8. 擬似命令の特殊オペランド

SFR シンボルは，ユーザシンボルと同様に大文字と小文字の区別を/CD オプションと/NCD オプションを使って制御することができます。あとのすべての予約語は，大文字と小文字の区別は行われません。

予約語の中には，同じシンボルで複数の用途を持つものもあります。それらはプログラム中に記述した文脈によってどちらの意味を持つかが判断されます。

予約語の中には，特定のコアを持つマイクロコントローラに特有のものもあります。そのような予約語は，対象の CPU コア以外を使用する場合には開放され，ユーザシンボルとして使用することができます。

「付録 B 予約語一覧」に，SFR シンボルを除く，すべての予約語とその用途，そのシンボルを予約語とする CPU コアの種類を示しています。

次に，それぞれの予約語の説明をします。

3.1.3.6.1 命令

RASU8 がアセンブルできるマイクロコントローラの命令です。命令の機能については，関連するドキュメントを参照してください。

3.1.3.6.2 擬似命令

RASU8 が用意している擬似命令です。擬似命令の機能については，「5. 擬似命令の詳細」を参照してください。

3.1.3.6.3 レジスタ

レジスタを表すシンボルです。マイクロコントローラの命令のオペランドに使用します。レジスタを用いたオペランドの書き方については，「4.1 アドレッシングの書式」を参照してください。レジスタを表す予約語は，次のとおりです。

```
R0      R1      R2      R3      R4      R5      R6      R7
R8      R9      R10     R11     R12     R13     R14     R15
ER0     ER2     ER4     ER6     ER8     ER10    ER12    ER14
XR0     XR4     XR8     XR12    QR0     QR8
CR0     CR1     CR2     CR3     CR4     CR5     CR6     CR7
CR8     CR9     CR10    CR11    CR12    CR13    CR14    CR15
CER0    CER2    CER4    CER6    CER8    CER10   CER12   CER14
CXR0    CXR4    CXR8    CXR12   CQR0    CQR8
BP      DSR     EA      ECSR    ELR     EPSW    FP      LR
SP      PC      PSW
```

3.1.3.6.4 演算子

演算子を表すシンボルです。式の記述に使用します。演算子の機能と使用方法については，「3.2.2 演算子」を参照してください。演算子を表す予約語は次のとおりです。

```
BYTE1       BYTE2       BYTE3       BYTE4       WORD1       WORD2
SEG         OFFSET      BPOS        SIZE
OVL_SEG     OVL_OFFSET  OVL_ADDRESS
```

3.1.3.6.5 SFR シンボル

対象のマイクロコントローラ固有のアドレス値を持つシンボルです。SFR 領域の各レジスタ名やレジスタのビット名などがこれに該当します。RASU8 では，この予約語をアブソリュートシンボルとしてアセンブルします。この予約語は DCL ファイルで定義されています。

例

以下の例は，SFR シンボルである“P0CON0”と“P00”を利用してポート 0.0 をハイインピーダンス入力にセットし，ポート 0.0 のビットをテストしています。

```
L    R0, P0CON0
OR   R0, #03H
ST   R0, P0CON0
BTST P00
```

SFR シンボルは，ほかの予約語と違って，/CD オプションと/NCD オプションの影響を受けます。デフォルト，または/CD を指定する場合，シンボルの綴りと，英大文字と英小文字の使い分

けが同じ場合だけ，同じシンボルとしてアセンブルされます。/NCD オプションを指定する場合は，シンボルの綴りが同じであれば，英大文字と英小文字の使い分けが異なっていても同じシンボルとしてアセンブルされます。

3.1.3.6.6 アドレッシング指定子

アドレッシング指定子はアクセスの対象となるメモリの領域を明示するためのシンボルです。アドレッシング指定子を用いたオペランドの書き方については，「4.1.3 メモリアドレッシング」を参照してください。アドレッシング指定子を表す予約語は次のとおりです。

```
NEAR        FAR
```

3.1.3.6.7 命令の特殊オペランド

条件分岐命令の分岐条件や，フラグ名等のような，命令のオペランドに指定する，アドレッシングとは異なる意味を持つシンボルです。具体的な意味や使用方法などについては，nX-U8 コアのインストラクションマニュアルを参照してください。命令の特殊オペランドを表す予約語は次のとおりです。

```
GT          GE          LT          LE          PS          NS
EQ          NE          ZF          NZ          CY          NC
GTS         GES         LTS         LES         OV          NV
AL
```

3.1.3.6.8 擬似命令の特殊オペランド

擬似命令のオペランドに指定する，特別な意味を持つシンボルです。それぞれのシンボルの意味や使用方法については，「5. 擬似命令の詳細」を参照してください。

```
CODE        DATA        BIT         NVDATA      NVBIT       TABLE
DYNAMIC     NVRAM       UNIT        WORD        DUP
NEAR        FAR         SMALL       LARGE
```

3.1.4 ロケーションカウンタ記号

構文

$

説明

ロケーションカウンタ記号を使用すると，現在アセンブルしている論理セグメントのロケーションカウンタの値を参照できます。。

ロケーションカウンタについては，「2.5 ロケーションカウンタ」を参照してください。

3.2 演算子と式

命令および擬似命令のオペランドには，式を使用できます。式とは，いくつかの定数およびアドレスや数値を表わすシンボルを演算子で結合したものです。RASU8 は，ソースステートメントに記述された式を評価して，ひとつの値に変換します。プログラマは，プログラムに直接値を記述するかわりに，その値の本来の意味を式として記述することができます。

この章では，最初にアセンブリ言語における式の基本的な考え方を説明したあと，RASU8 が用意する演算子の種類とそれぞれの機能，値の性質による式の分類，RASU8 の式の評価のしかた仕方を説明します。

3.2.1 式の基本的な考え方

3.2.1.1 式が属性を持つ意味

定数やシンボルのような値を表現する要素は，さまざまな属性を持っています。例えば CODE セグメントの領域で記述されたラベルは CODE アドレス空間上のアドレスを示しているという属性を持っています。リロケータブルセグメントの領域で記述されたラベルはそのリロケータブルセグメントの属性を持っています。

RASU8 は，式の演算結果もまた，定数やシンボルと同じように数値型とアドレス型に分類し，さらにユーセージタイプと呼ばれる値の利用目的を表わす属性を与えます。式の属性は，演算子の種類や演算の対象となる定数やシンボルの種類によって決定します。つまり，RASU8 は，式の値だけではなく式の意味も管理しています。

上記の説明は，漠然としていますが，実際の例を見れば，この考え方が簡単であり，また自然であることが理解できます。

例 1

```
L    ER0, TBL+2
```

この例では，TBL は DATA アドレス空間のアドレスと仮定します。このとき，“TBL+2”という表現は，TBL から 2 バイト目のアドレスであることがわかります。すなわち，“TBL+2”は，DATA アドレス空間のアドレスを表わす式であると言えます。

例 2

```
MOV  R0, #END_ADR-START_ADR
```

この例で，START_ADR と END_ADR は DATA アドレス空間上に用意したバッファの開始アドレスと終了アドレスと仮定します。このとき，START_ADR と END_ADR それぞれは，アドレスを表わしますが，“END_ADR-START_ADR”という表現は，バッファのサイズを表わす数値であることがわかります。

例 3

```
SB   D_ADR.4
```

この例で，D_ADR は DATA アドレス空間のアドレスと仮定します。このとき，“D_ADR.4”

という表現は，D_ADR のビット 4 を表わしていることがわかります。すなわち，“D_ADR.4”は，BIT アドレス空間のアドレスを表わす式であると言えます。

このように，式が属性を持つ理由は次の 2 つです。

1. 命令や擬似命令のオペランドに記述された式が正しい使われ方をされているかどうかを RASU8 が監視すること。
2. 式自体の記述が意味的に矛盾していないかどうかを RASU8 が監視すること。

特に，1.が重要です。RASU8 は，命令や擬似命令のオペランドに本来指定されるはずのない式が記述された場合，ワーニングを表示してオペランドの指定が誤りである可能性があることをプログラマに知らせます。

次に，使用目的の点で間違った式の記述の例を示します。

例 4

```
B     DATA_ADR
```

この例で，DATA_ADR はデータメモリ空間上のアドレスと仮定します。このとき，B 命令の第 2 オペランドには，コードメモリ空間上のアドレスが指定されるべきです。したがって，DATA_ADR は，明らかに間違った使い方がされています。このとき，RASU8 は，このソースステートメントに対して，ワーニングを報告します。

例 5

```
MOV  R0, END_ADR+START_ADR
```

この例は，演算結果に意味のない式の例です。この例で，END_ADR と START_ADR は，どちらもデータメモリ空間上のアドレスと仮定します。このとき，“END_ADR+START_ADR”という表現は，アドレス同士の加算ですので，この演算自体にあまり意味がありません。場合によっては，プログラマが意識的にこのような記述をするかもしれませんが，間違った使い方をしている可能性の方が大きいと言えます。この場合も，RASU8 はワーニングを報告します。

このように，式が属性を持ち，式の記述にある程度の制限を与えることは，プログラムの安全性を高める上で重要です。

3.2.1.2 アセンブル時に値が決まらない式

整定数やアブソリュートシンボルなど，値が確定している要素だけからなる式は，アセンブル時に式の値を決定できます。このような式を定数式と呼びます。定数式には，すべての演算子を使用することができます。

これに対して，リロケータブルシンボルを含んだ式は，アセンブル時には，式の値が決定できない場合があります。このような式をリロケータブル式と呼びます。リロケータブル式の情報はオブジェクトファイルに出力され，RLU8 によって解決されます。

次にリロケータブル式の例を，いくつか示します。

例 1

```
     EXTRN     DATA:GL_TBL
     L    ER0, GL_TBL+2
```

この例では，イクスターナルシンボル GL_TBL に 2 を加算しています。GL_TBL の値は，アセンブル時には確定しません。したがって，“GL_TBL+2”という式の値も，アセンブル時には確定しません。

例 2

```
GL_TBL    COMM DATA 10H
     SB   GL_TBL.4
```

この例では，共有シンボル GL_TBL が示すデータメモリ空間上のアドレスのビット 4 を参照しています。例 1 と同じ様に，GL_TBL の値はアセンブル時には確定しません。したがって，“GL_TBL.4”という式の値も，アセンブル時には確定しません。

このような，リロケータブルシンボルに対する演算は，ごく一部の演算子しか使用することができず，また記述形式にも制限があります。リロケータブル式の記述のしかたと制限については「3.2.3　式の種類」を参照してください。

3.2.2 演算子

ここでは RASU8 で使用可能な演算子の機能を説明します。RASU8 が提供する演算子には次の種類があります。

1. 算術演算子
2. 論理演算子
3. ビット演算子
4. 関係演算子
5. ドット演算子
6. 特殊演算子

演算子には単項演算子と二項演算子があります。単項演算子の右側，および二項演算子の両側に，式を使用できます。

以下の説明の中で，“真”とは 0 以外の数値を示し，“偽”は 0 を意味します。また，構文の説明に使用する *expression*，*expression1*，および *expression2* は式を意味しています。

3.2.2.1 算術演算子

一般的な算術演算のための演算子です。

演算子	構文	意味
+	*expression1* + *expression2*	加算
	+ *expression1*	正数（単項演算子）
-	*expression1* - *expression2*	減算
	- *expression1*	負数（単項演算子）
*	*expression1* * *expression2*	乗算
/	*expression1* / *expression2*	除算
%	*expression1* % *expression2*	モジュロ算（*expression1* を *expression2* で割った余り）

例

```
VALUE     EQU  30H
     ADD  R0, #VALUE+1
BUFSIZE   EQU  1024H
     DS   BUFSIZE*4
```

この例では ADD 命令のオペランドに+演算子を，DS 擬似命令のオペランドに*演算子を使用しています。

3.2.2.2 論理演算子

論理演算子は，左辺と右辺，または右辺の式の真偽を評価して，条件に応じた真偽値を与えます。

演算子	構文	意味
&&	*expression1* && *expression2*	両方とも真なら 1，それ以外は 0
||	*expression1* || *expression2*	どちらかが真なら 1，それ以外は 0
!	!*expression*	*expression* が真なら 0，偽なら 1 になる。

例

```
SW1  EQU   0
SW2  EQU   2

IF  SW1&&SW2
   BUSIZE   EQU  1024
ELSE
   BUSIZE   EQU  2048
ENDIF
```

この例では，IF 擬似命令のオペランドに&&演算子を使用しています。

3.2.2.3 ビット論理演算子

ビット論理演算子は，式の各ビットに対して，論理演算を実行します。

演算子	構文	意味
&	*expression1* & *expression2*	論理積
|	*expression1* | *expression2*	論理和
^	*expression1* ^ *expression2*	排他的論理和
<<	*expression1* << *expression2*	*expression2* が示す値だけ，*expression1* を左へビットシフトする。最下位ビット側から 0 がシフトインされる。
>>	*expression1* >> *expression2*	*expression2* が示す値だけ，*expression1* を右へビットシフトする。最上位ビット側から 0 がシフトインされる。
～	～ *expression*	ビット反転

例

```
SCB_MSK   EQU  1111_1000B
BCB_MSK   EQU  1100_1111B
     AND  R0, #BCB_MSK & SCB_MSK
```

この例では，AND 命令のオペランドに，&演算子を使用しています。

3.2.2.4 関係演算子

関係演算子は，2 つの式の値の大小関係を比較します。条件を満足する場合は 1 になり，条件を満足しない場合は 0 になります。関係演算子は定数式だけに使用できます。アドレスを表わす式同士どうしの演算では，物理セグメントアドレスも比較の対象になります。

演算子	構文	意味
>	*expression1* > *expression2*	*expression1* が *expression2* よりも大きければ 1 を返す。そうでなければ 0 を返す。
>=	*expression1* >= *expression2*	*expression1* が *expression2* 以上であれば 1 を返す。そうでなければ 0 を返す。
<	*expression1* < *expression2*	*expression1* が *expression2* よりも小さければ 1 を返す。そうでなければ 0 を返す。
<=	*expression1* <= *expression2*	*expression1* が *expression2* 以下であれば 1 を返す。そうでなければ 0 を返す。
==	*expression1* == *expression2*	*expression1* と *expression2* が等しければ 1 を返す。そうでなければ 0 を返す。
!=	*expression1* != *expression2*	*expression1* と *expression2* が異なっていれば 1 を返す。そうでなければ 0 を返す。

例

```
IF   VALUE1 >= VALUE2
      .
      .
      .
ENDIF
```

この例では，IF 擬似命令のオペランドに>=演算子を使用して条件アセンブルを行っています。

3.2.2.5 ドット演算子

ドット演算子は，データアドレスとビットオフセットから，ビットアドレスを算出します。

構文

expression1.*expression2*

説明

expression1 にはデータアドレス値を指定します。*expression2* には，データアドレス内のビット位置を指定します。上記の構文は次の式と同じ値を持ちます。

((*expression1* << 3) + *expression2*)

RASU8 は，ドット演算子（.）を算術演算子と同様に扱います。すなわち，*expression1* と *expression2* の値の範囲はチェックされません。

例

ドット演算子を使用した例を以下に示します。この例では，ユーザシンボル DATSYM1 と DATSYM2 は，ユーセージタイプ DATA を持っています。また，ユーザシンボル EXTNUM1 は，ユーセージタイプ NUMBER を持っています。そして，SB 命令のオペランドにこれらのシンボルをアドレスとするデータメモリのビット 0 を指定しています。

```
DATSYM1   DATA 8000H
EXTRN     NUMBER:EXTNUM1

     DSEG #0 AT 8800H
DATSYM2:
     DS   1

     CSEG
     SB   DATSYM1.0
     SB   DATSYM2.0
     SB   EXTNUM1.0
```

3.2.2.6 アドレス演算子

アドレス演算子は，左辺の値を物理セグメントとし，右辺の値をオフセットアドレスとするアドレス型の値を求める演算子です。

構文

expression1::*expression2*

説明

expression1 にアドレス型の値を指定した場合，その物理セグメントアドレスの値を結果の物理セグメントアドレスとします。数値型の値を指定した場合，その値そのものを結果の物理セグメントアドレスとします。

expression2 にアドレス型の値を指定した場合，そのオフセットアドレスを結果のオフセットアドレスとします。数値型の値を指定した場合，その値そのものを結果のオフセットアドレスとします。

例

以下に::演算子を利用した例を示します。1 番目の L 命令では，DATSYM と同じ物理セグメントの FFFFH にアクセスしています。2 番目の L 命令では，物理セグメントアドレス#1 の DATSYM と同じオフセットアドレスにアクセスしています。

```
     DSEG
DATASYM:  DS   10H

     CSEG
     L    R0,  DATASYM::0FFFFH
     L    R1,  1::DATASYM
```

3.2.2.7 特殊演算子

式の値から，ある連続したビットの取り出しや，セグメントシンボルからそのセグメントが実際に配置されるアドレスを取得を行う演算子です。特殊演算子には次の種類があります。

演算子	構文	意味
BYTE1	BYTE1 *expression*	(*expression* & 0FFH)と等価である。
BYTE2	BYTE2 *expression*	((*expression* >> 8) & 0FFH)と等価である。
BYTE3	BYTE3 *expression*	((*expression* >> 16) & 0FFH)と等価である。
BYTE4	BYTE4 *expression*	((*expression* >> 24) & 0FFH)と等価である。
WORD1	WORD1 *expression*	((*expression*) & 0FFFFH)と等価である。
WORD2	WORD2 *expression*	((*expression* >> 16)& 0FFFFH) と等価である。
SEG	SEG *expression*	アドレス型の式 *expression* の物理セグメントアドレスを得る。
OFFSET	OFFSET *expression*	アドレス型の式 *expression* のオフセットアドレスを得る。
SIZE	SIZE *segment_symbol*	*segment_symbol* からそのセグメントのサイズを得る。
BPOS	BPOS *expression*	ビットアドレスを表す式 *expression* からビットオフセット値，つまり下位 3 ビットを得る。
OVL_SEG	OVL_SEG *segment_symbol*	*segment_symbol* からそのセグメントが実際に配置される物理セグメントアドレスを得る
OVL_OFFSET	OVL_OFFSET *segment_symbol*	*segment_symbol* からそのセグメントが実際に配置される開始位置のオフセットアドレスを得る。
OVL_ADDRESS	OVL_ADDRESS *segment_symbol*	*segment_symbol* からそのセグメントが実際に配置される開始アドレスを得る。

他の演算子が汎用的であるのに対して，特殊演算子は， nX-U8 のインストラクションセット

を有効に利用する目的で用意されたものであり，その用途が限定されています。したがって，演算の対象となる式にもいくつかの制限があります。

BYTE1 ～ BYTE4 演算子は，数値の特定の連続する 8 ビットのデータを得るための演算子です。また，WORD1 演算子と WORD2 演算子は，数値の特定の連続する 16 ビットのデータを得るための演算子です。アドレス型の式に対してこれらの演算子を使用することもできますが，この場合はワーニングになります。

例 1

BYTE1 ～ BYTE4 演算子，WORD1 演算子および WORD2 演算子を使用した例を次に示します。

```
VALUE     EQU  12345678H

     CSEG
     MOV  R0, #BYTE1 VALUE         ;78H
     MOV  R1, #BYTE2 VALUE         ;56H
     MOV  R2, #BYTE3 VALUE         ;34H
     MOV  R3, #BYTE4 VALUE         ;12H

     TSEG
     DW   WORD1 VALUE         ;5678H
     DW   WORD2 VALUE         ;1234H
```

SEG 演算子はアドレス式から物理セグメントアドレスを取り出すための演算子です。従って数値型の式に対して使用することはできません。

OFFSET 演算子は，アドレス式からオフセットアドレスを取り出すための演算子です。アドレス型の式を数値型に型変換するためにも使用できます。数値型の式に対して OFFSET 演算子を使用することもできますが，この場合はワーニングになります。

例 2

この例では，リロケータブルセグメント DATA_SEG の物理セグメントアドレスとセグメントの開始アドレスを求めるために，SEG 演算子と OFFSET 演算子を使用しています。

```
DATA_SEG  SEGMENT   DATA
     MOV  R2, #SEG DATA_SEG        ;物理セグメントアドレス
     MOV  ER0, #OFFSET DATA_SEG    ;オフセットアドレス
```

BPOS 演算子は，ビットアドレスのビットオフセットを得るための演算子です。連続データ領域の特定のビット位置だけにアクセスするような場合を想定して用意された演算子です。BPOS 演算は，次の演算と同じ働きをします。

expression & 7

BPOS 演算子をユーセージタイプ BIT，NVBIT，または TBIT 以外の式に使用した場合，ワーニングになります。

例 3

この例では，D_TBL 上の FLG1 と同じビット位置にアクセスするために，BPOS 演算子を使用しています。

```
     BSEG
FLG1:     DBIT 1

     DSEG #0
D_TBL:    DS   10H

     CSEG
     MOV  R10, #BYTE1 D_TBL
     MOV  R11, #BYTE2 D_TBL
LOOP:
     L    R0, [ER10]
     SB   R0.(BPOS FLG1) ;ビット位置を得る
     .
     .
     .
```

SIZE 演算子は，リロケータブルセグメントのサイズを得るための演算子です。したがって，SIZE 演算をセグメントシンボル以外に使用することはできません。

例 4

この例では，リロケータブルセグメント DATA_SEG のセグメントサイズを SIZE 演算子を使用して求めています。

```
DATA_SEG  SEGMENT   DATA WORD
     DW   SIZE DATA_SEG  ;セグメントサイズを得る
```

OVL_SEG 演算子，OVL_OFFSET 演算子および OVL_ADDRESS 演算子はセグメントシンボルからオーバーレイ機能使用時にそのセグメントが退避されているアドレスを取り出すための演算子です。従ってこれらの演算をセグメントシンボル以外に使用することは出来ません。

例 5

この例では，OVL_SEG と OVL_OFFSET を使用してプログラムを退避しているアドレスからプログラムを QR0 に取り出しています。

```
     CSEG
     MOV  R8, #OVL_SEG CODE_SEG2
     LEA  OVL_OFFSET CODE_SEG2
LOOP2:
     L    QR0, R8:[EA+]
     .
     .
     .
CODE_SEG2 SEGMENT   CODE NVRAM
     RSEG CODE_SEG2
START2:
     .
     .
     .
```

例 6

この例では，OVL_ADDRESS を使用してプログラムを退避しているアドレスからプログラムを ER2 に取り出しています。

```
     CSEG
     MOV  ER0, #0
LOOP1:
     L    ER2, OVL_ADDRESS CODE_SEG1[ER0]
     .
     .
     .
CODE_SEG1 SEGMENT   CODE NVRAM
     RSEG CODE_SEG1
START1:
     .
     .
     .
```

3.2.3 式の種類

RASU8 によって式の値がどの程度確定できるか，という視点から，式は大きく次の 3 種類に分類できます。

1. 定数式
2. リロケータブル式
3. 単純リロケータブル式

定数式とは，RASU8 が値を確定できる式のことです。

リロケータブル式とは，RASU8 が値を確定することができず，RLU8 によって値が確定される式のことです。リロケータブル式には，文法上許される範囲内で，すべてのリロケータブルシンボルを使用することができます。

単純リロケータブル式とは，リロケータブル式の一種であり，セグメントベースからのオフセットアドレスは確定できる式のことです。単純リロケータブル式には，リロケータブルシン

ボルの中でも単純リロケータブルシンボルに限って使用することができます。

例

```
ABS_DATA  DATA 1000H
EXTRN     DATA:EXT_DATA

DATA_SEG  SEGMENT   DATA WORD
     RSEG DATA_SEG
D_TBL:    DS   10H

     CSEG
     L    R0, ABS_DATA+10H
     L    R1, EXT_DATA+10H
     L    R2, D_TBL+10H
```

この例の L 命令の第 2 オペランドを順番に見ていきます。最初の“ABS_DATA+10H”は，ABS_DATA の値が 1000H に確定していますので，式の値は 1010H であることも確定します。ですから，この式は定数式です。

2 番目の“EXT_DATA+10H”は，イクスターナルシンボルが表わすアドレスへの加算ですから，RASU8 はこの式の値は確定できません。ですから，この式はリロケータブル式です。

3 番目の“D_TBL+10H”は，D_TBL 自体のアドレスは確定しませんが，このプログラムの中でのセグメントベースからのオフセット値は 0 です。したがって，式の値は，少なくともこのプログラム中では，10H であることが確定します。ですから，この式は単純リロケータブル式です。

このように式を分類する理由は，命令や擬似命令の種類によっては，オペランドに指定できる式の種類に制限を与えているからです。命令や擬似命令のオペランドに記述する式は，どれだけ自由に式が使えるか，という尺度で次の 3 つに分類されます。

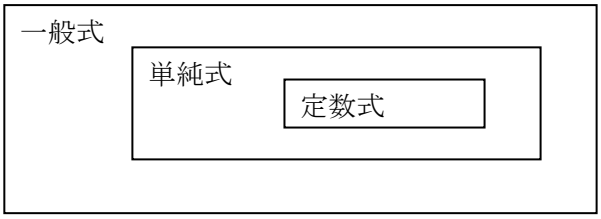
1. 定数式
2. 単純式
3. 一般式

定数式とは，すでに述べたとおり RASU8 が値を確定できる式のことであり，記述の自由度がもっとも低い式です。

単純式とは，定数式に単純リロケータブル式を加えたもので，少なくともプログラム内でのセグメントベースからのオフセットアドレスは確定できる式のことです。

一般式とは，定数式とリロケータブル式の総称です。つまり，文法上許されるすべての式のことであり，記述の自由度がもっとも高い式です。

定数式，単純式，一般式の関係を図で表すと以下の様になります。



基本命令のオペランドのほとんどには，一般式を記述することができますが，擬似命令のオペランドの多くは，定数式かもしくは単純式しか認めていません。このような制限が存在する理由は，擬似命令の種類によっては，RASU8 の処理の都合上，定数式や単純式でなければ不都合な場合があるからです。具体的な実例を，「3.2.3.4 式の記述の制限」に示しています。また，命令のオペランドに使用できる式については「4. アドレッシングと命令」を参照してください。擬似命令のオペランドに使用できる式については「5. 擬似命令の詳細」を参照してください。

次に，定数式，単純式，一般式の書式を説明します。

3.2.3.1 定数式

定数式は RASU8 が値を決定できる式です。具体的には，整定数，文字定数，アブソリュートシンボル，またはアブソリュートセグメントに使用したロケーションカウンタ記号を演算子で結合したものです。定数式にはすべての演算子を使用できます。定数式には，リロケータブルシンボルを使用できません。これは，RASU8 が定数式の値を決定しなければならないからです。ただし，次の場合には，リロケータブルシンボルを含む式であっても定数式になります。

1. 同じリロケータブルセグメントに所属する単純リロケータブルシンボル同士の減算を行う場合。
2. 同じリロケータブルセグメントに所属する単純リロケータブルシンボルにドット演算（.）を行った式同士どうしの減算を行う場合。
3. リロケータブルシンボルにドット演算（.）を行い，さらにその式に BPOS 演算を行う場合。
4. 物理セグメント属性が ANY でないリロケータブルシンボルに SEG 演算を行う場合。

例

定数式を使用した例を以下に示します。

```
ABSSYM1  EQU      100H

DATSEG4  SEGMENT  DATA #2
         RSEG     DATSEG4
LABEL1:
         DS       2
LABEL2:
         TSEG
         DW       100H                      ;100H
         DW       'A'                       ;41H
         DW       ABSSYM1                   ;100H
         DW       1+2                       ;3H
         DW       +(1+2*ABSSYM1)            ;201H
         DW       100H*'A'                  ;4100H
         DW       (LABEL2-LABEL1)           ;2H
         DW       (LABEL2.0-LABEL1.0)       ;10H
         DW       BPOS(LABEL1.3)            ;3H
         DW       SEG LABEL2                ;2H
```

上記の記述で，DW 擬似命令のオペランドに記述された式は全て定数式で，結果はコメントに記述された値になります。

3.2.3.2 単純式

単純リロケータブルシンボル以外のリロケータブルシンボルを含まない式です。単純リロケータブルシンボルとして，リロケータブルセグメントに所属するロケーションカウンタシンボルを使用できます。単純式には定数式が含まれます。単純式の構文は次のとおりです。

式	定義
単純式	定数式
	| 単純リロケータブル式
単純リロケータブル式	単純リロケータブルシンボル
	| (単純リロケータブル式)
	| + 単純リロケータブル式
	| 単純リロケータブル式 + 定数式
	| 定数式 + 単純リロケータブル式
	| 単純リロケータブル式 - 定数式
	| 単純リロケータブル式 . 定数式
	| OFFSET 単純リロケータブル式

この構文の定義において，縦棒（｜）は，いくつかの項目の中から 1 つだけ指定することを表します。

定数式でない単純式にドット演算子が使用されている場合，その式に対してさらにドット演算子または OFFSET 演算子は使用できません。また，単純式が単純リロケータブルシンボルを含む場合，RASU8 は単純式の値を決定できません。この場合，単純式の最終的な値は RLU8 が決定します。

例

定数式以外の単純式の例を以下に示します。

```
DATSEG5   SEGMENT   DATA
     RSEG DATSEG5
     DS   2
LABEL3:
     ORG  LABEL3
```

上記のとおり定義されたシンボルを使用した単純式の例を次の表に示します。

```
(LABEL3)
+LABEL3
LABEL3-1
LABEL3.4
OFFSET LABEL3
```

3.2.3.3 一般式

一般式は，セグメントシンボル，イクスターナルシンボル，および共有シンボルを含む式を単純式に追加したものです。一般式の構文は次のとおりです。

式	定義
一般式	定数式
	｜ リロケータブル式
	｜ リロケータブル演算式
リロケータブル式	リロケータブルシンボル
	｜ (リロケータブル式)
	｜ + リロケータブル式
	｜ リロケータブル式 + 定数式
	｜ 定数式 + リロケータブル式
	｜ リロケータブル式 - 定数式
	｜ リロケータブル式 . 定数式
	｜ OFFSET リロケータブル式
リロケータブル演算式	BYTE1 リロケータブル式
	｜ BYTE2 リロケータブル式
	｜ BYTE3 リロケータブル式
	｜ BYTE4 リロケータブル式
	｜ WORD1 リロケータブル式
	｜ WORD2 リロケータブル式
	｜ SEG リロケータブル式
	｜ BPOS リロケータブル式
	｜ SIZE リロケータブル式
	｜ OVL_SEG セグメントシンボル
	｜ OVL_OFFSET セグメントシンボル
	｜ OVL_ADDRESS セグメントシンボル
	｜ (リロケータブル演算式)

この構文の定義において，縦棒（｜）は，いくつかの項目の中から 1 つだけ指定することを表します。リロケータブル式の構文が一般式の構文のあとに示されています。ドット演算子を含むリロケータブル式に対してさらにドット演算子を使用したり，特殊演算子を使用することはできません。

RASU8 はリロケータブル式の値を決定できません。リロケータブル式の最終的な値は RLU8 が決定します。

例

定数式と単純式ではない一般式の例を以下に示します。

```
SEGSYM    SEGMENT DATA
COMMSYM   COMM DATA 2
EXTRN     DATA:EXTSYM BIT:BITSYM
```

上記のとおり定義されたシンボルを使用した一般式の例を次の表に示します。

```
EXTSYM
(COMMSYM)
+EXTSYM
COMMSYM-1
EXTSYM.1
(EXTSYM+1).1
EXTSYM.0+10.0
HIGH EXTSYM
LOW(COMMSYM)
SEG (+EXTSYM)
OFFSET (COMMSYM-1)
PAGE SEGSYM
LREG (COMMSYM)
BPOS BITSYM
SIZE SEGSYM
```

3.2.3.4 式の記述の制限

それぞれの式の記述位置には制限があります。また，式中のユーザシンボルに前方参照が許されない場合があります。このような制限には次の種類があります。

(1) ORG 擬似命令のオペランドでの制限

アブソリュートセグメントに ORG 擬似命令を記述する場合，オペランドには定数式だけが指定できます。これは，オペランドのアドレスをアセンブル時に確定しなければならないためです。

それに対して，リロケータブルセグメントに ORG 擬似命令を記述する場合，オペランドには単純式を指定できます。ただし，単純式に含まれる単純リロケータブルシンボルは，現在のリロケータブルセグメントに所属していなければなりません。なぜなら，RASU8 において，リロケータブルセグメントのアドレスは確定できなくても，その相対的なアドレスの関係は確定できなければならないからです。相対的なアドレスの関係が確定すれば，リロケータブルセグメントのサイズを確定できます。この確定したサイズを使用して，RLU8 は論理セグメントをメモリに割り付けます。

例

```
     TYPE(M610001)
XCODSEG   SEGMENT   CODE
     RSEG XCODSEG
XLABEL:
     .
     .
     .

CODESEG   SEGMENT   CODE
     RSEG CODESEG
     ORG  10H
     .
     .
     .
LABEL:
     .
     .
     .
     ORG  LABEL+100H
     .
     .
     .
     ORG  XLABEL          ;エラー
     .
     .
     .
```

リロケータブルセグメント CODESEG に所属する ORG 擬似命令文のオペランドには，定数式の“10H”，単純式の“LABEL+100H”，および単純式の“XLABEL”が指定されています。オペランドが現在のリロケータブルセグメントに所属しない単純リロケータブルシンボルである“XLABEL”の場合だけエラーになります。

(2) ローカルシンボルを定義する擬似命令のオペランドでの制限

EQU 擬似命令や CODE 擬似命令など，ローカルシンボルを定義する擬似命令のオペランドには，単純式が指定できます。ただし，単純式ではない一般式は指定できません。また，これらの擬似命令のオペランドに指定されるユーザシンボルには，前方参照は許されません。

(3) その他の擬似命令のオペランドでの制限

(1) および(2) 以外の擬似命令のオペランドでも，記述できる式が制限される場合があります。このような制限については，それぞれの擬似命令の説明を参照してください。

(4) マイクロコントローラの命令のオペランドにおける制限

マイクロコントローラの命令のオペランドのうち，ローテートシフト命令のシフト幅とビットアドレッシングのビット位置には，定数式だけが指定できます。それ以外のアドレッシングモードには，一般式を使用することができます。

3.2.4 式の評価

3.2.4.1 演算子の優先順位

演算子の優先度は，式の評価順序を決定します。演算子はすべて優先度の高いものから評価されます。優先度の等しい演算子は，式中左から記述されている順に評価されます。

演算子の優先度は次のとおりです。優先度の高いものほど，優先順位の番号が小さくなります。

優先順位	演算子
1	( ) OVL_ADDRESS OVL_SEG OVL_OFFSET
2	. ::
3	! ~ +（単項） -（単項）BYTE1 BYTE2 BYTE3 BYTE4 WORD1 WORD2 SEG OFFSET BPOS SIZE
4	* / %
5	+（二項） -（二項）
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	|
12	&&
13	||

例

```
LABEL     DATA 200H

     L    R0, LABEL+2*8
     SB   (LABEL+2).7
     SB   LABEL+2.7
```

この例では，ユーセージタイプ DATA を持つアブソリュートシンボル LABEL を定義しています。その後に，このシンボルを使用した式をオペランドとする 3 つの命令文があります。

3 行目の命令の第 2 オペランドの値は，210H になります。4 行目の命令のオペランドは，アドレスが LABEL+2 のデータメモリのビット 7 を表します。注意しなければならないのは，5 行目最後の命令のオペランドは，アドレスが LABEL+2 のデータメモリのビット 7 を表さないことです。なぜなら，+演算子よりも，ドット演算子（.）の優先度の方が高いので，“LABEL+(2.7)”と評価されるからです。

3.2.4.2 式の持つ数値の評価

RASU8 は数値を符号なし 32 ビット整数として扱います。式の持つ数値を計算する場合，演算中も値を符号なし 32 ビット整数として扱います。アドレスを表す式では，物理セグメントアドレスを符号なし 8 ビットで扱います。RASU8 は，評価順序にしたがって演算し，式の持つ数値を計算します。そして，それぞれのオペランドに応じて，計算結果の有効範囲がチェックされます。

3.2.4.3 式の属性の評価

ここでは，RASU8 が式の属性をどのように評価するのかを説明します。

シンボルだけからなる式の場合，その式の持つ属性は，シンボルの持つ属性になります。

演算子を使用した式の場合，その式の属性は，使用する演算子の種類と演算の対象となる式の属性によって異なります。演算子によっては，演算の対象となる式の種類に制限を与えるものがあり，RASU8 は制限に違反する式を，エラーまたはワーニングにします。エラーになるのは，RASU8 が演算結果を出せないような式です。ワーニングとなるのは，演算そのものはできても，その演算に意味がないような式です。

ここでは，演算子の種類と演算の対象となる式の種類によって，演算結果の属性がどうなるか，またどのような記述がエラーまたはワーニングになるかを演算子ごとに説明します。説明の中で，次のような式の種類を表わす記号を使用します。

記号	意味
address_expression	アドレス型の式を表わします。すなわち，ユーセージタイプ NONE，CODE，DATA，NVDATA，TABLE，BIT，NVBIT，または TBIT を持つ式です。*segment_symbol* もこの中に含まれます。
number_expression	数値型の式，すなわちユーセージタイプ NUMBER を持つ式を表します。
code_expression	ユーセージタイプ CODE を持つ式です。
data_expression	ユーセージタイプ DATA を持つ式です。
nvdata_expression	ユーセージタイプ NVDATA を持つ式です。
table_expression	ユーセージタイプ TABLE を持つ式です。
bit_expression	ユーセージタイプ BIT を持つ式です。
nvbit_expression	ユーセージタイプ NVBIT を持つ式です。
tbit_expression	ユーセージタイプ TBIT を持つ式です。
none_expression	ユーセージタイプ NONE を持つ式です。
segment_symbol	セグメントシンボル単体です。

(1) ( )の属性評価

演算子( )を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ　エラーの有無
(*address_expression*)	*address_expression* と同じ
(*number_expression*)	*number_expression* と同じ

説明

カッコで囲まれた式の属性は全く変化しません。

(2) +，-演算の属性評価

演算子+，-を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
+ *number_expression*	*number_expression* の属性	
+ *address_expression*	*address_expression* の属性	
number_expression + *number_expression*	数値型	
number_expression + *address_expression*	*address_expression* の属性	
address_expression + *number_expression*	*address_expression* の属性	
address_expression + *address_expression*	数値型	ワーニング
- *number_expression*	*number_expression* の属性	
- *address_expression*	*address_expression* の属性	
number_expression – *number_expression*	数値型	
number_expression – *address_expression*	*address_expression* の属性	
address_expression – *number_expression*	*address_expression* の属性	
address_expression – *address_expression*	数値型	条件によってはエラー

説明

単項の+演算と-演算では，指定された式の属性をそのまま引き継ぎます。

二項の+と-演算では，左辺と右辺のどちらか一方がアドレス型であれば，アドレスの属性を引き継ぎます。

アドレス型同士の加算は，ワーニングになります。

アドレス型同士の減算は，同じ論理セグメント内のアドレス同士の減算のみが許されます。

(3) *，/，%演算の属性評価

演算子*，/，%を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
number_expression * *number_expression*	数値型	
number_expression * *address_expression*	数値型	ワーニング
address_expression * *number_expression*	数値型	ワーニング
address_expression * *address_expression*	数値型	ワーニング
number_expression / *number_expression*	数値型	
number_expression / *address_expression*	数値型	ワーニング
address_expression / *number_expression*	数値型	ワーニング
address_expression / *address_expression*	数値型	ワーニング
number_expression % *number_expression*	数値型	
number_expression % *address_expression*	数値型	ワーニング
address_expression % *number_expression*	数値型	ワーニング
address_expression % *address_expression*	数値型	ワーニング

説明

これらの演算を，アドレス型を対して使用した場合は，ワーニングになります。

演算結果はどの演算も数値型になります。

(4) 論理演算の属性評価

論理演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
number_expression && *number_expression*	数値型	
number_expression && *address_expression*	数値型	ワーニング
address_expression && *number_expression*	数値型	ワーニング
address_expression && *address_expression*	数値型	ワーニング
number_expression || *number_expression*	数値型	
number_expression || *address_expression*	数値型	ワーニング
address_expression || *number_expression*	数値型	ワーニング
address_expression || *address_expression*	数値型	ワーニング
! *number_expression*	数値型	
! *address_expression*	数値型	ワーニング

説明

これらの演算子を，アドレス型に対して使用した場合は，ワーニングになります。

演算結果はどの演算も数値型になります。

(5) ビット論理演算の属性評価

ビット論理演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
number_expression & *number_expression*	数値型	
number_expression & *address_expression*	数値型	ワーニング
address_expression & *number_expression*	数値型	ワーニング
address_expression & *address_expression*	数値型	ワーニング
number_expression | *number_expression*	数値型	
number_expression | *address_expression*	数値型	ワーニング
address_expression | *number_expression*	数値型	ワーニング
address_expression | *address_expression*	数値型	ワーニング
number_expression ^ *number_expression*	数値型	
number_expression ^ *address_expression*	数値型	ワーニング
address_expression ^ *number_expression*	数値型	ワーニング
address_expression ^ *address_expression*	数値型	ワーニング
number_expression << *number_expression*	数値型	
number_expression << *address_expression*	数値型	ワーニング
address_expression << *number_expression*	数値型	ワーニング
address_expression << *address_expression*	数値型	ワーニング
number_expression >> *number_expression*	数値型	
number_expression >> *address_expression*	数値型	ワーニング
address_expression >> *number_expression*	数値型	ワーニング
address_expression >> *address_expression*	数値型	ワーニング
~ *number_expression*	数値型	
~ *address_expression*	数値型	ワーニング

説明

これらの演算子を，アドレス型に対して使用した場合は，ワーニングになります。

演算結果はどの演算も数値型になります。

(6) 関係演算の属性評価

関係演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
number_expression > *number_expression*	数値型	
number_expression > *address_expression*	数値型	ワーニング
address_expression > *number_expression*	数値型	ワーニング
address_expression > *address_expression*	数値型	
number_expression >= *number_expression*	数値型	
number_expression >= *address_expression*	数値型	ワーニング
address_expression >= *number_expression*	数値型	ワーニング
address_expression >= *address_expression*	数値型	
number_expression < *number_expression*	数値型	
number_expression < *address_expression*	数値型	ワーニング
address_expression < *number_expression*	数値型	ワーニング
address_expression < *address_expression*	数値型	
number_expression <= *number_expression*	数値型	
number_expression <= *address_expression*	数値型	ワーニング
address_expression <= *number_expression*	数値型	ワーニング
address_expression <= *address_expression*	数値型	
number_expression == *number_expression*	数値型	
number_expression == *address_expression*	数値型	ワーニング
address_expression == *number_expression*	数値型	ワーニング
address_expression == *address_expression*	数値型	
number_expression != *number_expression*	数値型	
number_expression != *address_expression*	数値型	ワーニング
address_expression != *number_expression*	数値型	ワーニング
address_expression != *address_expression*	数値型	

説明

これらの演算を数値型とアドレス型の間で行った場合はワーニングになります。

アドレス型同士の比較では，物理セグメントアドレスも比較の対象になります。

演算結果はどの演算も数値型になります。

(7) アドレス演算の属性評価

アドレス演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ　エラーの有無
number_expression :: *number_expression*	NONE 型
number_expression :: *address_expression*	NONE 型
address_expression :: *number_expression*	NONE 型
address_expression :: *address_expression*	NONE 型

説明

上記の組み合わせでアドレス演算を行った場合，エラー，ワーニングとも発生しません。

演算結果はどの演算も NONE 型になります。

(8) ドット演算の属性評価

ドット演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
number_expression.number_expression	数値型	
code_expression.number_expression	数値型	ワーニング
data_expression.number_expression	BIT 型	
nvdata_expression.number_expression	NVBIT 型	
table_expression.number_expression	TBIT 型	
bit_expression.number_expression	数値型	ワーニング
nvbit_expression.number_expression	数値型	ワーニング
tbit_expression.number_expression	数値型	ワーニング
none_expression.number_expression	NONE 型	
number_expression.address_expression	数値型	ワーニング
code_expression.address_expression	数値型	ワーニング
data_expression.address_expression	BIT 型	ワーニング
nvdata_expression.address_expression	NVBIT 型	ワーニング
table_expression.address_expression	TBIT 型	ワーニング
bit_expression.address_expression	数値型	ワーニング
nvbit_expression.address_expression	数値型	ワーニング
tbit_expression.address_expression	数値型	ワーニング
none_expression.address_expression	NONE 型	ワーニング

説明

ドット演算子の右辺にアドレス型を記述した場合は，ワーニングになります。また，左辺にユーセージタイプ CODE，BIT，NVBIT，TBIT のアドレス型を記述した場合も，ワーニングになります。

演算結果の属性は，左辺に記述した式のユーセージタイプによって異なります。

(9) 特殊演算の属性評価

特殊演算子を使用する演算の場合，次のとおり評価されます。

式の記述形式	演算結果のユーセージタイプ	エラーの有無
BYTE1 *number_expression*	数値型	
BYTE1 *address_expression*	数値型	ワーニング
BYTE2 *number_expression*	数値型	
BYTE2 *address_expression*	数値型	ワーニング
BYTE3 *number_expression*	数値型	
BYTE3 *address_expression*	数値型	ワーニング
BYTE4 *number_expression*	数値型	
BYTE4 *address_expression*	数値型	ワーニング
WORD1 *number_expression*	数値型	
WORD1 *address_expression*	数値型	ワーニング
WORD2 *number_expression*	数値型	
WORD2 *address_expression*	数値型	ワーニング
SEG *address_expression*	数値型	
OFFSET *number_expression*	数値型	ワーニング
OFFSET *address_expression*	数値型	
BPOS *number_expression*	数値型	ワーニング
BPOS *code_expression*	数値型	ワーニング
BPOS *data_expression*	数値型	ワーニング
BPOS *nvdata_expression*	数値型	ワーニング
BPOS *table_expression*	数値型	ワーニング
BPOS *bit_expression*	数値型	
BPOS *nvbit_expression*	数値型	
BPOS *tbit_expression*	数値型	
BPOS *none_expression*	数値型	ワーニング
SIZE *segment_symbol*	数値型	
OVL_ADDRESS *segment_symbol*	NONE 型	
OVL_SEG *segment_symbol*	数値型	
OVL_OFFSET *segment_symbol*	数値型	

説明

BYTE1 演算，BYTE2 演算，BYTE3 演算，BYTE4 演算，WORD1 演算，WORD2 演算を，アドレス型に対して使用した場合，ワーニングになります。

SEG 演算子は物理セグメントアドレスを求める演算子なので，数値型には使用できません。

OFFSET 演算子，BPOS 演算子は，アドレスに関する演算を行うので，数値型に対して使用するとワーニングになります。また，BPOS 演算はビットオフセットを求める演算なので，ビットを単位とするアドレス型以外(ユーセージタイプ NONE，CODE，DATA，TABLE)に使用した場合は，ワーニングになります。

SIZE 演算，OVL_ADDRESS 演算，OVL_SEG 演算，OVL_OFFSET 演算はセグメントシンボルにしか記述できません。

演算結果は，OVL_ADDRESS 演算以外は数値型になります。OVL_ADDRESS 演算の演算結果は NONE 型になります。

４ アドレッシングと命令

4.1 アドレッシングの書式

基本命令のオペランドに記述できるアドレッシングの一覧表を種類別に示します。各アドレッシング機能の詳細は，『nX-U8 コア インストラクションマニュアル』を参照してください。

アドレッシングに関しては，アセンブリソースの記述性を高めるため『nX-U8 コア インストラクションマニュアル』に記述されていないアセンブラ独自のアドレッシングもあります。アセンブラ独自のアドレッシングは，RASU8 によって『nX-U8 コア インストラクションマニュアル』に記述されているアドレッシングに変換されます。アセンブラ独自のアドレッシングについては，その都度説明します。

注意

アドレッシングの記述の中で[ ]を使用するものがありますが，これは[ ]内の記述が省略可能であることを示しているわけではありませんので，ご注意ください。

4.1.1 表記について

アドレッシングの記述を示す前に，アドレッシングの記述で用いる記号とその意味を以下に示します。

記号	意味
R*n*	バイト型汎用レジスタ R0, R1, …, R14, R15 のいずれかを示す。
ER*n*	ワード型汎用レジスタ ER0, ER2, …, ER12, ER14 のいずれかを示す。
XR*n*	ダブルワード型汎用レジスタ XR0, XR4, XR8, XR12 のいずれかを示す。
QR*n*	クワッドワード型汎用レジスタ QR0, QR8 のいずれかを示す。
Disp16	16 ビットサイズのディスプレースメントを表す-65535～+65535 の数値型一般式，または物理セグメントアドレスをもつ一般式。
Disp6	6 ビットサイズのディスプレースメントを表す-32～+31 の数値型一般式，または物理セグメントアドレスをもつ一般式。
width	ビットシフトのシフト幅を表す 0～7 の数値型定数式。
imm8	バイト型の即値を表す 0～0FFH の数値型一般式。
imm7	符号付き 7 ビットの即値を表す-64～+63 の数値型一般式。
bit_offset	ビットオフセットを表す 0～7 の数値型定数式。
Radr	相対分岐命令または条件相対分岐命令の分岐先を表す一般式。
snum7	SWI 命令のベクタ番号を表す 0～+63 の数値型一般式。
signed8	符号付きバイト型の即値を表す-128～+127 の数値型一般式(*imm8* の派生)。
Sadr	SWI 命令のベクタアドレスを表す一般式(*snum7* の派生)。

記号	意味
unsigned8	符号なしバイト型の即値を表す 0～0FFH の数値型一般式(*imm8* の派生)。
Dadr	データメモリ空間上のバイトアドレスを表す一般式。
Cadr	プログラムメモリ空間上のバイトアドレスを表す一般式。
Dbitadr	データメモリ空間上のビットアドレスを表す一般式。
pseg_addr	物理セグメントアドレスを表す一般式。

命令のオペランドに記述する式については，次のような規則があります。

(1) すべてのアドレッシングで，シンボルの前方参照が許されます。ただし，アドレッシングの最適化は，前方参照を含む式には適用されません。

(2) *bit_offset*，*width* 以外は一般式を使用することができます。

(3) 数値式を記述すべきアドレッシングに，アドレス式を記述してもかまいません。この場合，物理セグメントアドレスは無視され，オフセットアドレスのみが有効になります。

(4) アドレス式を記述すべきアドレッシングに，数値式を記述してもかまいません。この場合，物理セグメント#0 のメモリ空間上のアドレスとして扱われます。

(5) アドレス式を記述すべきアドレッシングは，式のユーセージタイプに制限を設けています。許されないユーセージタイプのアドレス式を記述する場合，ワーニングになります。

4.1.2 レジスタアドレッシング

指定したレジスタそのものがアクセスの対象となります。レジスタアドレッシングの記述は，以下のとおりです。

アドレッシング記述	機能
R*n*	バイト型の汎用レジスタ（R*n*）がアクセスの対象となります。
ER*n*	ワード型の汎用レジスタ（ER*n*）がアクセスの対象となります。インストラクション表の記述で，オペランドに ER*n* が記されている場合，ER12 の代用として BP，ER14 の代用として FP の記述が可能です。
XR*n*	ダブルワード型の汎用レジスタ（XR*n*）がアクセスの対象となります。
QR*n*	クワッドワード型の汎用レジスタ（QR*n*）がアクセスの対象となります。
CR*n*	バイト型のコプロセッサレジスタ（CR*n*）がアクセスの対象となります。
CER*n*	ワード型のコプロセッサレジスタ（CER*n*）がアクセスの対象となります。

アドレッシング記述	機能
CXR*n*	ダブルワード型のコプロセッサレジスタ（CXR*n*）がアクセスの対象となります。
CQR*n*	クワッドワード型のコプロセッサレジスタ（CQR*n*）がアクセスの対象となります。
PC	プログラムカウンタがアクセスの対象となります。
LR	リンクレジスタがアクセスの対象となります。
EA	EA レジスタがアクセスの対象となります。
SP	スタックポインタがアクセスの対象となります。
PSW	プログラムステータスワードがアクセスの対象となります。
ELR	例外処理用のリンクレジスタがアクセスの対象となります。
ECSR	CSR 退避レジスタがアクセスの対象となります。
EPSW	PSW 退避レジスタがアクセスの対象となります。
R*n*.*bit_offset*	汎用レジスタ R*n* の *bit_offset* で示されるビット位置のデータがアクセスの対象となります。

4.1.3 メモリアドレッシング

メモリアドレッシングは，データメモリ空間上のメモリをアクセスの対象とするものです。

4.1.3.1 レジスタ間接アドレッシング

レジスタの内容をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。レジスタ間接アドレッシングの記述は以下のとおりです。

アドレッシング記述	機能
[EA]	EA レジスタの内容をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。 DSR プリフィックス命令の代替記述がない場合，物理セグメント#0 のデータメモリ空間がアクセスの対象となります。
pseg_addr:[EA]	物理セグメント指定付きのアドレッシングです。物理セグメント#*pseg_addr* のデータメモリ空間がアクセスの対象となります。 *pseg_addr* のユーセージタイプは NUMBER でなければなりません。NUMBER 以外の場合は，エラーになります。
DSR:[EA]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。

アドレッシング記述	機能
R*n*:[EA]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。
[EA+]	EA レジスタの内容をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。対象のメモリにアクセスしたあと，EA レジスタの内容がインクリメントされます。 EA レジスタに加算される値は，バイト型の命令では 1，ワード型の命令では 2，ダブルワード型の命令では 4，クワッドワード型の命令では 8 となります。 DSR プリフィックス命令の代替記述がない場合，物理セグメント#0 のデータメモリ空間がアクセスの対象となります。
pseg_addr:[EA+]	物理セグメント指定付きのアドレッシングです。物理セグメント#*pseg_addr* のデータメモリ空間が指定されます。 *pseg_addr* のユーセージタイプは NUMBER でなければなりません。NUMBER 以外の場合は，エラーになります。
DSR:[EA+]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間が指定されます。
R*n*:[EA+]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間が指定されます。
[ER*n*]	ワード型汎用レジスタ ER*n* の内容をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。[ER12]の代用として[BP]，[ER14]の代用として[FP]を記述することも可能です。 DSR プリフィックス命令の代替記述がない場合，物理セグメント#0 のデータメモリ空間がアクセスの対象となります。
pseg_addr:[ER*n*]	物理セグメント指定付きのアドレッシングです。物理セグメント#*pseg_addr* のデータメモリ空間がアクセスの対象となります。 *pseg_addr* のユーセージタイプは NUMBER でなければなりません。NUMBER 以外の場合は，エラーになります。 LEA 命令では，このアドレッシングは記述できません。
DSR:[ER*n*]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
R*n*:[ER*m*]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。
Disp16[ER*n*]	*Disp16*+ER*n* をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。 このアドレッシング記述はデータモデルに依存します。NEAR モデルの場合には NEAR *Disp16*[ER*n*]になります。FAR モデルの場合には，*Disp16* が前方参照を含まない式で，物理セグメント属性が#0 であれば NEAR *Disp16*[ER*n*]に，それ以外の場合は FAR *Disp16*[ER*n*]になります。 *Disp16* のユーセージタイプは NUMBER, DATA, NVDATA, TABLE, NONE のいずれかでなければなりません。その他の場合はワーニングとなります。
NEAR *Disp16*[ER*n*]	物理セグメント#0 に限定したアドレッシングになります。 LEA 命令では，このアドレッシングは記述できません。
FAR *Disp16*[ER*n*]	物理セグメントを限定しないアドレッシングになります。 *Disp16* の属する物理セグメント#(SEG *Disp16*)のデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。
DSR:*Disp16*[ER*n*]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。*Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
R*n*:*Disp16*[ER*m*]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 *Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
Disp16[BP]	このアドレッシングは，『nX-U8 コア インストラクションマニュアル』には記述されていない独自のアドレッシングです。 *Disp16*+BP をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。 このアドレッシング記述はデータモデルに依存します。NEAR モデルの場合には NEAR *Disp16*[BP]になります。FAR モデルの場合には，*Disp16* が前方参照を含まない式で，物理セグメント属性が#0 であれば NEAR *Disp16*[BP]に，それ以外の場合は FAR *Disp16*[BP]になります。 *Disp16*[BP]を記述した場合，*Disp16* の値により最適化が行われます。*Disp16* が前方参照シンボルを含まないアブソリュート式で，かつ *Disp16* のオフセットアドレスが-32 以上，+31 以下である場合，*Disp6*[BP]に置き換えられます。条件を満たさない場合には，*Disp16*[ER12]に置き換えられます。 *Disp16* のユーセージタイプは NUMBER, DATA, NVDATA, TABLE, NONE のいずれかでなければなりません。その他の場合はワーニングとなります。
NEAR *Disp16*[BP]	物理セグメント#0 に限定したアドレッシングになります。 LEA 命令では，このアドレッシングは記述できません。
FAR *Disp16*[BP]	物理セグメントを限定しないアドレッシングになります。 *Disp16* の属する物理セグメント#(SEG *Disp16*) のデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。
DSR:*Disp16*[BP]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 *Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
R*n*:*Disp16*[BP]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 *Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
Disp16[FP]	このアドレッシングは，『nX-U8 コア インストラクションマニュアル』には記述されていない独自のアドレッシングです。 *Disp16*+FP をアドレスとするデータメモリ空間上のメモリが，アクセスの対象となります。 このアドレッシング記述はデータモデルに依存します。NEAR モデルの場合には NEAR *Disp16*[FP]になります。FAR モデルの場合には，*Disp16* が前方参照を含まない式で，物理セグメント属性が#0 であれば NEAR *Disp16*[FP]に，それ以外の場合は FAR *Disp16*[FP]になります。 *Disp16*[FP]を記述した場合，*Disp16* の値により最適化が行われます。*Disp16* が前方参照シンボルを含まないアブソリュート式で，かつ *Disp16* のオフセットアドレスが-32 以上，+31 以下である場合，*Disp6*[FP]に置き換えられます。条件を満たさない場合には，*Disp16*[ER14]に置き換えられます。 *Disp16* のユーセージタイプは NUMBER, DATA, NVDATA, TABLE, NONE のいずれかでなければなりません。その他の場合はワーニングとなります。
NEAR *Disp16*[FP]	物理セグメント#0 に限定したアドレッシングになります。 LEA 命令では，このアドレッシングは記述できません。
FAR *Disp16*[FP]	物理セグメントを限定しないアドレッシングになります。*Disp16* の属する物理セグメント#(SEG *Disp16*) のデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
DSR:*Disp16*[FP]	物理セグメント指定付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 *Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
R*n*:*Disp16*[FP]	物理セグメント指定付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。 *Disp16* の属する物理セグメントは無視されます。 *Disp16* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。

4.1.3.2 ダイレクトアドレッシング

記述した値をアドレスとするデータメモリ空間上のメモリをアクセスの対象とします。ダイレクトアドレッシングの記述は以下のとおりです。

アドレッシング記述	機能
Dadr	記述した値をバイトアドレスとするデータメモリ空間のメモリがアクセスの対象となります。 このアドレッシングはデータモデルに依存します。NEAR モデルの場合には NEAR *Dadr* になります。FAR モデルの場合には，*Dadr* が前方参照を含まない式で，物理セグメント属性が#0 であれば NEAR *Dadr* に，それ以外の場合は FAR *Dadr* になります。*Dadr* のユーセージタイプは DATA, NVDATA, TABLE, NUMBER, NONE のいずれかでなければなりません。その他の場合はワーニングとなります。
NEAR *Dadr*	物理セグメント#0 に限定したアドレッシングになります。 LEA 命令では，このアドレッシングは記述できません。
FAR *Dadr*	物理セグメントを限定しないアドレッシングになります。 *Dadr* の属する物理セグメント#(SEG *Dadr*)のデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
DSR:*Dadr*	物理セグメント付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間がアクセスの対象となります。*Dadr* はオフセットアドレスのみが有効となり，*Dadr* の持つ物理セグメントアドレスは無視されます。 *Dadr* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
R*n*:*Dadr*	物理セグメント付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間が指定されます。*Dadr* はオフセットアドレスのみが有効となり，*Dadr* の持つ物理セグメントアドレスは無視されます。 *Dadr* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
Dbitadr	記述した値をビットアドレスとするデータメモリ空間のメモリがアクセスの対象となります。 このアドレッシングはデータモデルに依存します。NEAR モデルの場合には NEAR *Dbitadr* になります。FAR モデルの場合には，*Dbitadr* が前方参照を含まない式で，物理セグメント属性が#0 であれば NEAR *Dbitadr* に，それ以外の場合は FAR *Dbitadr* になります。 *Dbitadr* のユーセージタイプは BIT, NVBIT, TBIT, NUMBER, NONE のいずれかでなければなりません。その他の場合はワーニングとなります。
NEAR *Dbitadr*	物理セグメント#0 に限定したビットアドレッシングになります。 LEA 命令では，このアドレッシングは記述できません。
FAR *Dbitadr*	物理セグメントを限定しないビットアドレッシングになります。 *Dbitadr* の属する物理セグメント#(SEG *Dbitadr*)のデータメモリ空間がアクセスの対象となります。 LEA 命令では，このアドレッシングは記述できません。

アドレッシング記述	機能
DSR:*Dbitadr*	物理セグメント付きのアドレッシングです。DSR によって示される物理セグメントのデータメモリ空間が指定されます。*Dbitadr* はオフセットアドレスのみが有効となり，*Dbitadr* の持つ物理セグメントアドレスは無視されます。 *Dbitadr* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。
R*n*:*Dbitadr*	物理セグメント付きのアドレッシングです。汎用レジスタ R*n* によって示される物理セグメントのデータメモリ空間が指定されます。*Dbitadr* はオフセットアドレスのみが有効となり，*Dbitadr* の持つ物理セグメントアドレスは無視されます。 *Dbitadr* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。 LEA 命令では，このアドレッシングは記述できません。

4.1.4 即値アドレッシング

記述した値そのものが対象となります。即値アドレッシングの記述は，以下のとおりです。

アドレッシング記述	機能
#imm8	記述した値は，8 ビット即値として扱われます。 *imm8* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。
#signed8	記述した値は，符号付きの 8 ビット即値として扱われます。 ADD SP, *#imm8* を記述した場合に，*imm8* は *signed8* として扱われます。 値の範囲は，-128 <= *signed8* <= +127 となります。 *signed8* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。
#unsigned8	記述した値は，符号なしの 8 ビット即値として扱われます。 MOV PSW, *#imm8* を記述した場合に，*imm8* は *unsigned8* として扱われます。 値の範囲は，0 <= *unsigned8* <= 0FFH となります。 *unsigned8* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。

アドレッシング記述	機能
#width	記述した値は，シフト幅として扱われます。 値の範囲は，0 <= *width* <= 7 となります。 *width* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。
#snum7	記述した値は，SWI 命令のベクタ番号として扱われます。 値の範囲は，0 <= *snum7* <= 63 となります。 *snum7* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。
#imm7	記述した値は，符号付きの 7 ビット即値として扱われます。 値の範囲は，-64 <= *imm7* <= +63 となります。 *imm7* のユーセージタイプは NUMBER でなければなりません。その他の場合はワーニングとなります。
Sadr	SWI 命令のベクタアドレスとして扱われます。この記述は，*#snum7* に置き換えられます。置き換え式は次のとおりです。 *snum7* = (*Sadr* – *swi_vector_start*)/2 *swi_vector_start* は，SWI ベクタアドレスの開始アドレスを示します。 *Sadr* のユーセージタイプは CODE, NONE, NUMBER のいずれかでなければなりません。その他の場合はワーニングとなります。

4.1.5 プログラムメモリアドレッシング

プログラムメモリ空間上のメモリがアクセスの対象となります。プログラムメモリアドレッシングの記述は，以下のとおりです。

アドレッシング記述	機能
Cadr	B, BL の分岐先アドレスとなります。*Cadr* は物理セグメントアドレスを含みますので，異なる物理セグメントアドレスへの分岐が可能です。
Radr	条件分岐，または最適化分岐擬似命令の分岐先アドレスとなります。分岐先アドレスは，同一物理セグメント内に限定されます。
ER*n*	ワード型汎用レジスタ ER*n* の内容が分岐先アドレスとなります。B, BL で用いられます。分岐先アドレスは，同一物理セグメント内に限定されます。

4.2 命令一覧

ここでは，nX-U8 で使用できる命令，および各命令に対する記述可能なアドレッシングの一覧を示します。

各命令の詳細については，『nX-U8 コア インストラクションマニュアル』を参照してください。アドレッシングの表記の詳細については，「4.1 アドレッシングの書式」を参照してください。

4.2.1 演算命令

ニーモニック	オペランド 1	オペランド 2	備考
MOV ADD AND OR XOR CMPC ADDC CMP	R*n*	*#imm8* R*m*	*imm8* のユーセージタイプは，NUMBER でなければなりません。
SUB SUBC	R*n*	R*m*	
MOV ADD	ER*n*	ER*m* *#imm7*	*imm7* のユーセージタイプは，NUMBER でなければなりません。
CMP	ER*n*	ER*m*	

4.2.2 シフト命令

ニーモニック	オペランド 1	オペランド 2	備考
SLL SRL SRA SLLC SRLC	R*n*	R*m* *#width*	*width* のユーセージタイプは NUMBER でなければなりません。

4.2.3 ロード／ストア命令

ニーモニック	オペランド 1	オペランド 2	備考
L ST	R*n* ER*n* XR*n* QR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*n*:[EA] [EA+] *pseg_addr*:[EA+] DSR:[EA+] R*n*:[EA+]	*pseg_addr* のユーセージタイプは，NUMBER でなければなりません。
	R*n* ER*n*	[ER*m*] *pseg_addr*:[ER*m*] DSR:[ER*m*] R*n*:[ER*m*]	*pseg_addr* のユーセージタイプは，NUMBER でなければなりません。
		Disp16[ER*m*] NEAR *Disp16*[ER*m*] FAR *Disp16*[ER*m*] *Disp16*[BP] NEAR *Disp16*[BP] FAR *Disp16*[BP] *Disp16*[FP] NEAR *Disp16*[FP] FAR *Disp16*[FP]	*Disp16* のユーセージタイプは，DATA，NVDATA，TABLE，NONE，NUMBER のいずれかでなければなりません。
		DSR:*Disp16*[ER*m*] R*n*:*Disp16*[ER*m*] DSR:*Disp16*[BP] R*n*:*Disp16*[BP] DSR:*Disp16*[FP] R*n*:*Disp16*[FP]	*Disp16* のユーセージタイプは，NUMBER でなければなりません。
		Dadr NEAR *Dadr* FAR *Dadr*	*Dadr* のユーセージタイプは，DATA，NVDATA，TABLE，NONE，NUMBER のいずれかでなければなりません。
		DSR:*Dadr* R*m*:*Dadr*	*Dadr* のユーセージタイプは，NUMBER でなければなりません。

4.2.4 コントロールレジスタアクセス命令

ニーモニック	オペランド 1	オペランド 2	備考
MOV	R*n*	PSW EPSW ECSR	
	PSW EPSW ECSR	R*m*	
	ER*n*	ELR SP	ER12 の代わりに BP を，ER14 の代わりに FP を記述することが可能です。
	ELR SP	ER*m*	ER12 の代わりに BP を，ER14 の代わりに FP を記述することが可能です。
	PSW	*#unsigned8*	*unsigned8* のユーセージタイプは，NUMBER でなければなりません。
ADD	SP	*#signed8*	*signed8* のユーセージタイプは，NUMBER でなければなりません。

4.2.5 PUSH/POP 命令

ニーモニック	オペランド 1	オペランド 2	備考
PUSH	R*n* ER*n* XR*n* QR*n* *register_list*		*register_list* には EPSW, ELR, LR, EA のすべて，または一部を指定できます。
POP	R*n* ER*n* XR*n* QR*n* *register_list*		*register_list* には PSW, LR, PC, EA のすべて，または一部を指定できます。

4.2.6 コプロセッサ転送命令

ニーモニック	オペランド 1	オペランド 2	備考
MOV	CR*n* CER*n* CXR*n* CQR*n*	R*m* [EA] *pseg_addr*:[EA] DSR:[EA] R*m*:[EA] [EA+] *pseg_addr*:[EA+] DSR:[EA+] R*m*:[EA+]	*pseg_addr* のユーセージタイプは，NUMBER でなければなりません。
	R*n* [EA] *pseg_addr*:[EA] DSR:[EA] R*n*:[EA] [EA+] *pseg_addr*:[EA+] DSR:[EA+] R*n*:[EA+]	CR*m* CER*m* CXR*m* CQR*m*	*pseg_addr* のユーセージタイプは，NUMBER でなければなりません。

4.2.7 EA レジスタ転送命令

ニーモニック	オペランド 1	オペランド 2	備考
LEA	[ER*n*] *Disp16*[ER*n*] *Dadr*		*Disp16*，*Dadr* のユーセージタイプは，NUMBER でなければなりません。

4.2.8 ALU 命令

ニーモニック	オペランド 1	オペランド 2	備考
DAA DAS NEG	R*n*		

4.2.9 ビットアクセス命令

ニーモニック	オペランド 1	オペランド 2	備考
SB TB RB	R*n.bit_offset* *Dbitadr*		*Dbitadr* のユーセージタイプは BIT, NVBIT, TBIT, NONE, NUMBER のいずれかでなければなりません。

4.2.10 PSW アクセス命令

ニーモニック	オペランド 1	オペランド 2	備考
EI DI SC RC CPLC			

4.2.11 条件相対分岐命令

ニーモニック	オペランド 1	オペランド 2	備考
BEQ BNE BLT BLE BGT BGE BLTS BLES BGTS BGES BZ BNZ BCY BNC BOV BNV BPS BNS BAL	*Radr*		*Radr* のユーセージタイプは CODE, NONE, NUMBER のいずれかでなければなりません。

4.2.12 符号拡張命令

ニーモニック	オペランド 1	オペランド 2	備考
EXTBW	ER*n*		

4.2.13 ソフトウェア割り込み命令

ニーモニック	オペランド 1	オペランド 2	備考
SWI	*#snum7*		*snum7* のユーセージタイプは NUMBER でなければなりません。
	Sadr		*Sadr* のユーセージタイプは CODE, NONE, NUMBER のいずれかでなければなりません。
BRK			

4.2.14 分岐命令

ニーモニック	オペランド 1	オペランド 2	備考
B BL	*Cadr* ER*n*		*Cadr* のユーセージタイプは CODE, NONE, NUMBER のいずれかでなければなりません。

4.2.15 乗除算命令

ニーモニック	オペランド 1	オペランド 2	備考
MUL DIV	ER*n*	R*m*	

4.2.16 その他

ニーモニック	オペランド 1	オペランド 2	備考
INC DEC	[EA]		

ニーモニック	オペランド 1	オペランド 2	備考
RTI			
RT			
NOP			

5 擬似命令の詳細

5.1 アセンブラ初期設定擬似命令

アセンブラ初期設定擬似命令は，RASU8 に対してどのような条件でアセンブルするのかを設定するために用意されている擬似命令です。したがって，アセンブラ初期設定擬似命令は，プログラムの最初に記述する必要があります。

5.1.1 TYPE 擬似命令

構文

TYPE (*dcl_name*)

説明

TYPE 擬似命令は，対象のマイクロコントローラに対応する DCL ファイル名を指定するための擬似命令です。RASU8 は，*dcl_name* をベース名，“.DCL”を拡張子とする DCL ファイルの情報を読み込みます。マイクロコントローラの名前と DCL ファイルのベース名はよく似ていますが，マイクロコントローラ名が“ML”で始まるのに対して，DCL ファイルのベース名は“M”で始まる点が異なります。例えば，マイクロコントローラ名が ML610001 であれば，TYPE 擬似命令には“M610001”を指定します。

DCL ファイルは，次の順序でサーチされます。

1. カレントディレクトリ
2. RASU8.EXE が存在するディレクトリ
3. 環境変数 DCL に指定されているディレクトリ

RASU8 は，アセンブル処理の前に DCL ファイルの内容を読み込みます。DCL ファイルの内容に誤りがあっても，DCL ファイルの終わりまで読み込みます。そして，発生するすべての DCL エラーを表示し強制終了します。DCL ファイルの読み込みが正常であれば，続いて RASU8 はソースファイルをアセンブルします。

補足

TYPE 擬似命令は，必ず指定しなければなりません。TYPE 擬似命令を指定しない場合や，指定される DCL ファイルを見つけることができない場合，RASU8 はソースファイルのアセンブル処理を行わず強制終了します。

TYPE 擬似命令はプログラムの先頭で指定してください。

TYPE 擬似命令を 2 回以上指定することはできません。

例

```
        TYPE(M610001)

        EXTRN DATA: _$$SP

        CSEG AT 0H
        DW  _$$SP
        DW  START
        CSEG AT 1000H
    START:
        .
        .
        .
```

この例では，対象のマイクロコントローラは ML610001 です。RASU8 は DCL ファイル M610001.DCL を読み込みます。

5.1.2 MODEL 擬似命令

構文

MODEL　*memory_model* [, *data_model*]

または

MODEL　*data_model* [, *memory_model*]

説明

MODEL 擬似命令は，使用するメモリモデル，およびデータモデルを RASU8 に知らせるための擬似命令です。

memory_model には，次のいずれかを指定します。

memory_model	メモリモデルの種類	意味
SMALL	SMALL モデル	プログラムコードを，すべて物理セグメント#0 に配置することを RASU8 に知らせます。
LARGE	LARGE モデル	プログラムコードを，物理セグメント#1 以上にも配置する可能性があることを RASU8 に知らせます。

data_model には，次のいずれかを指定します。

data_model	データモデルの種類	意味
NEAR	NEAR モデル	アドレッシング指定子が指定されていないデータを，すべて物理セグメント#0 に配置することを RASU8 に知らせます。
FAR	FAR モデル	アドレッシング指定子が指定されていないデータを，物理セグメント#1 以上にも配置する可能性があることを RASU8 に知らせます。

memory_model を省略した場合には， SMALL モデルが設定されます。

data_model を省略した場合には，NEAR モデルが設定されます。

```
TYPE (M610001)
MODEL LARGE, NEAR
REL_CODE_SEG SEGMENT CODE
REL_DATA_SEG SEGMENT DATA
```

この例では，メモリモデルを LARGE モデル，データモデルを NEAR モデルに設定しています。

補足

MODEL 擬似命令は，RASU8 に対してメモリモデル，およびデータモデルの種類を知らせるものであり，ハードウェア的にメモリモデルを設定するわけではありません。実際のメモリモデル設定は，メモリモデル制御用のレジスタに値を書き込む必要があります。

5.1.3 ROMWINDOW 擬似命令

構文

ROMWINDOW　*base_address*, *end_address*

説明

ROMWINDOW 擬似命令は，ROM ウィンドウ領域の範囲を指定します。

base_address は ROMWINDOW 領域のベースアドレスを表す定数式です。*end_address* は，ROM ウィンドウ領域のトップアドレスを表す定数式です。

設定できる *base_address* と *end_address* は，対象となるマイクロコントローラに依存します。設定できる範囲は，DCL ファイル中の ROMWINDOW 文を参照することにより，知ることができます。

補足

ROMWINDOW 擬似命令は，RASU8 に ROM ウィンドウ領域の範囲を知らせるためだけのもので，ハードウェア的に ROM ウィンドウ領域を設定するわけではありません。実際に ROM ウィンドウ機能を使用する場合には，ROM ウィンドウ機能をハードウェア的に制御するための設定

が必要です。

ROMWINDOW 擬似命令を 2 回以上指定することはできません。NOROMWIN 擬似命令が先に指定されていると，本擬似命令は指定できません。

例

```
TYPE (61xxxx)
ROMWINDOW 0, 3FFFH
 .
 .
 .
```

この例では，ROMWINDOW 擬似命令を使用して，ROM ウィンドウの領域を 0 から 3FFFH 番地に設定しています。

5.1.4 NOROMWIN 擬似命令

構文

NOROMWIN

説明

NOROMWIN 擬似命令は，ROM ウィンドウ機能を使用しないことを指定します。

NOROMWIN 擬似命令を指定した場合，RASU8 は物理セグメント#0 に ROM ウィンドウ領域が存在しないものと認識します。したがって，物理セグメント#0 に TABLE タイプの領域を確保しようとした場合は，RASU8 はエラーを表示します。

NOROMWIN 擬似命令で ROM ウィンドウ機能を使用しないことが指定されたモジュールは，ROM ウィンドウ機能を使用するモジュールとリンクすることはできません。ROM ウィンドウ機能を使用しないモジュールと，ROM ウィンドウ機能を使用するモジュールをリンクしようとした場合には，RLU8 はエラーを表示し，強制終了します。

補足

NOROMWIN 擬似命令を 2 回以上指定することはできません。ROMWINDOW 擬似命令が先に指定されていると，本擬似命令は指定できません。

5.2 プログラム終了宣言擬似命令

プログラム終了宣言擬似命令は，プログラムの終了を RASU8 に知らせるための擬似命令です。

5.2.1 END 擬似命令

構文

END

説明

END 擬似命令は，プログラムの終了を RASU8 に知らせるための擬似命令です。RASU8 は，END 擬似命令までをアセンブルします。END 擬似命令の後にソースステートメントを記述していても，RASU8 はそれを無視します。また，インクルードファイル中に END 擬似命令がある場合，RASU8 はそのインクルードファイル内の END 擬似命令以降のアセンブルを中止し，インクルードファイルを呼び出したソースファイルのアセンブル処理に戻ります。

5.3 シンボル定義擬似命令

シンボル定義擬似命令は，シンボルを定義し，そのシンボルに対して値，またはアドレス値を与えるための擬似命令です。

5.3.1 EQU 擬似命令

構文

symbol EQU *simple_expression*

説明

EQU 擬似命令は，ローカルシンボルを定義します。*symbol* には，定義するシンボルを指定し，*simple_expression* には，前方参照を含まない単純式を指定します。

EQU 擬似命令によって定義されるシンボルには，*simple_expression* の値と属性がそのまま与えられます。すなわち，*simple_expression* が定数式の場合，定義するシンボルはアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，定義するシンボルは単純リロケータブルシンボルになります。また，*simple_expression* が数値型の式であれば，シンボルのユーセージタイプは NUMBER になり，*simple_expression* がアドレス型であれば，シンボルには *simple_expression* のアドレスの性質がそのまま与えられます。

補足

すでに定義されているシンボルを *symbol* に指定することはできません。

この擬似命令により定義されるシンボルのユーセージタイプが NUMBER，NONE 以外の場合，RASU8 は，そのアドレスのメモリの種類とシンボルのユーセージタイプの整合がとれているかどうかをチェックします。メモリの種類とユーセージタイプの整合がとれていない場合，RASU8 はワーニングを表示します。

例

```
SEGSYM   SEGMENT DATA 2
         RSEG    SEGSYM
BUF1:    DS      4

BASE     EQU     10H
BUFSIZE  EQU     4H
VALUE    EQU     BASE+BUFSIZE
BUFX     EQU     BUF1+BUFSIZE
```

この例では，EQU 擬似命令を使用して 4 つのローカルシンボルを定義しています。BASE，BUFSIZE，および VALUE は，ユーセージタイプ NUMBER を持つアブソリュートシンボルになります。BUFX は，ユーセージタイプ DATA を持つ単純リロケータブルシンボルになります。

5.3.2 SET 擬似命令

構文

symbol SET *simple_expression*

説明

SET 擬似命令は，ローカルシンボルを定義します。*symbol* には，定義するシンボルを指定し，*simple_expression* には，前方参照を含まない単純式を指定します。

機能的には EQU 擬似命令と同じですが，SET 擬似命令で定義したシンボルは，SET 擬似命令を使用して何度も再定義することができます。

SET 擬似命令によって定義されるシンボルには，*simple_expression* の値と属性がそのまま与えられます。すなわち，*simple_expression* が定数式の場合，定義するシンボルはアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，定義するシンボルは単純リロケータブルシンボルになります。また，*simple_expression* が数値型の式であれば，シンボルのユーセージタイプは NUMBER になり，*simple_expression* がアドレス型であれば，シンボルには *simple_expression* のアドレスの性質がそのまま与えられます。

補足

すでに SET 擬似命令以外で定義されているシンボルを *symbol* に定義することはできません。

この擬似命令により定義されるシンボルのユーセージタイプが NUMBER，NONE 以外の場合，RASU8 は，そのアドレスのメモリの種類とシンボルのユーセージタイプの整合がとれているかどうかをチェックします。メモリの種類とユーセージタイプの整合がとれていない場合，RASU8 はワーニングを表示します。

例

```
SETSYM  SET 10H
        MOV R0,     #SETSYM
        .
        .
        .
SETSYM  SET 20H
        MOV R0,     #SETSYM
        .
        .
        .
```

この例では，SET 擬似命令を使用して，ユーセージタイプ NUMBER を持つアブソリュートシンボル SETSYM を定義しています。最初の SET 擬似命令の直後にある MOV 命令では，SETSYM の値は 10H になりますが，2 つ目の SET 擬似命令の直後にある MOV 命令では，SETSYM の値は 20H になります。

5.3.3 CODE 擬似命令

構文

symbol CODE *simple_expression*

説明

CODE 擬似命令は，CODE アドレス空間のバイトアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，CODE アドレス空間のアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ CODE が与えられます。

RASU8 は，*simple_expression* の値が CODE アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が CODE アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，CODE，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

例

```
CODE_SYM1 CODE 1000H
CODE_SYM2 CODE 2:2000H
         CSEG #3 AT  3000H
LABEL:   DW  1000H
CODE_SYM3 CODE LABEL+100H
```

この例では，CODE 擬似命令を使用して，ユーセージタイプ CODE を持つ 3 つのアブソリュートシンボルを定義しています。CODE_SYM1 は，物理セグメント#0 のオフセットアドレス 1000H を表すシンボルとなります。CODE_SYM2 は，物理セグメント#2 のオフセットアドレス 2000H を表すシンボルとなります。CODE_SYM3 は，物理セグメント#3 のオフセットアドレス 3100H を表すシンボルとなります。

5.3.4 TABLE 擬似命令

構文

symbol TABLE *simple_expression*

説明

TABLE 擬似命令は，TABLE アドレス空間のバイトアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，TABLE アドレス空間のアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ TABLE が与えられます。

RASU8 は，*simple_expression* の値が TABLE アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が TABLE アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

NOROMWIN 擬似命令が指定されていた場合で，*simple_expression* の値が物理セグメント#0 を示していた場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，TABLE，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合で，*simple_expression* の値が物理セグメント#0 を示している場合，ROM ウィンドウ領域の範囲が特定できないため，RASU8 はワーニングを表示します。

例

```
TABLE_SYM1 TABLE 1000H
TABLE_SYM2 TABLE 2:2000H
         TSEG #3 AT  3000H
LABEL:   DW  1234H
TABLE_SYM3 TABLE LABEL+100H
```

この例では，TABLE 擬似命令を使用して，ユーセージタイプ TABLE を持つ 3 つのアブソリュートシンボルを定義しています。TABLE_SYM1 は，物理セグメント#0 のオフセットアドレス 1000H を表すシンボルとなります。TABLE_SYM2 は，物理セグメント#2 のオフセットアドレス 2000H を表すシンボルとなります。TABLE_SYM3 は，物理セグメント#3 のオフセットアドレス 3100H を表すシンボルとなります。

5.3.5 TBIT 擬似命令

構文

symbol TBIT *simple_expression*

説明

TBIT 擬似命令は，TABLE アドレス空間のビットアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，TABLE アドレス空間のビットアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ TBIT が与えられます。

RASU8 は，*simple_expression* の値が TABLE アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が TABLE アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

NOROMWIN 擬似命令が指定されていた場合で，*simple_expression* の値が物理セグメント#0 を示していた場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，TBIT，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合で，*simple_expression* の値が物理セグメント#0 を示している場合，ROM ウィンドウ領域の範囲が特定できないため，RASU8 はワーニングを表示します。

例

```
TBIT_SYM1 TBIT 1000H.1
TBIT_SYM2 TBIT 2:2000H.4
         TSEG #3 AT  3000H
LABEL:   DB   0CAH
TBIT_SYM3 TBIT LABEL.3
```

この例では，TBIT 擬似命令を使用して，ユーセージタイプ TBIT を持つ 3 つのアブソリュートシンボルを定義しています。TBIT_SYM1 は，物理セグメント#0 のオフセットアドレス 1000H のビット 1 を表すシンボルとなります。TBIT_SYM2 は，物理セグメント#2 のオフセットアドレス 2000H のビット 4 を表すシンボルとなります。TBIT_SYM3 は，物理セグメント#3 のオフセットアドレス 3000H のビット 3 を表すシンボルとなります。

5.3.6 DATA 擬似命令

構文

symbol DATA *simple_expression*

説明

DATA 擬似命令は，DATA アドレス空間のバイトアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，DATA アドレス空間のアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ DATA が与えられます。

RASU8 は，*simple_expression* の値が DATA アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が DATA アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，DATA，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合，*simple_expression* の値が内部 RAM 領域，SFR 領域以外の物理セグメント#0 を示している場合，RASU8 はワーニングを表示します。

例

```
DATA_SYM1 DATA 0A000H
DATA_SYM2 DATA 4:2000H
         DSEG #5 AT  3000H
LABEL:   DS   2
DATA_SYM3 DATA LABEL+100H
```

この例では，DATA 擬似命令を使用して，ユーセージタイプ DATA を持つ 3 つのアブソリュートシンボルを定義しています。DATA_SYM1 は，物理セグメント#0 のオフセットアドレス A000H を表すシンボルとなります。DATA_SYM2 は，物理セグメント#4 のオフセットアドレス 2000H を表すシンボルとなります。DATA_SYM3 は，物理セグメント#5 のオフセットアドレス 3100H を表すシンボルとなります。

5.3.7 BIT 擬似命令

構文

symbol BIT *simple_expression*

説明

BIT 擬似命令は，BIT アドレス空間のアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，BIT アドレス空間のア

ドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ BIT が与えられます。

RASU8 は，*simple_expression* の値が BIT アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が BIT アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，BIT，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合，*simple_expression* の値が内部 RAM 領域，SFR 領域以外の物理セグメント#0 を示している場合，RASU8 はワーニングを表示します。

例

```
BIT_SYM1 BIT  0A000H.1
BIT_SYM2 BIT  4:2000H.2
         DSEG #5 AT  3000H
LABEL:   DS   2
BIT_SYM3 BIT  LABEL.3
```

この例では，BIT 擬似命令を使用して，ユーセージタイプ BIT を持つ 3 つのアブソリュートシンボルを定義しています。BIT_SYM1 は，物理セグメント#0 のオフセットアドレス A000H のビット 1 を表すシンボルとなります。BIT_SYM2 は，物理セグメント#4 のオフセットアドレス 2000H のビット 2 を表すシンボルとなります。BIT_SYM3 は，物理セグメント#5 のオフセットアドレス 3000H のビット 3 を表すシンボルとなります。

5.3.8 NVDATA 擬似命令

構文

symbol NVDATA *simple_expression*

説明

NVDATA 擬似命令は，NVDATA アドレス空間のバイトアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，NVDATA アドレス空間のアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになり

ます。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ NVDATA が与えられます。

RASU8 は，*simple_expression* の値が NVDATA アドレス空間の範囲内にあるかどうかをチェックします。指定した *simple_expression* の値が NVDATA アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，NVDATA，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合で，*simple_expression* の値が物理セグメント#0 を示している場合，ROM ウィンドウ領域の範囲が特定できないため，RASU8 はワーニングを表示します。

例

```
DATA_SYM1 NVDATA 8000H
DATA_SYM2 NVDATA 4:6000H
         NVSEG #5 AT 4000H
LABEL:   DS   2
NVDATA_SYM3 NVDATA LABEL+100H
```

この例では，NVDATA 擬似命令を使用して，ユーセージタイプ NVDATA を持つ 3 つのアブソリュートシンボルを定義しています。NVDATA_SYM1 は，物理セグメント#0 のオフセットアドレス 8000H を表すシンボルとなります。NVDATA_SYM2 は，物理セグメント#4 のオフセットアドレス 6000H を表すシンボルとなります。NVDATA_SYM3 は，物理セグメント#5 のオフセットアドレス 4100H を表すシンボルとなります。

5.3.9 NVBIT 擬似命令

構文

symbol NVBIT *simple_expression*

説明

NVBIT 擬似命令は，NVBIT アドレス空間のビットアドレスを表すローカルシンボルを定義します。*symbol* には，定義するローカルシンボルを指定します。*simple_expression* には，NVBIT アドレス空間のビットアドレスを表す，前方参照を含まない単純式を指定します。

simple_expression が定数式の場合，*symbol* はアブソリュートシンボルになります。*simple_expression* が単純リロケータブル式の場合，*symbol* は単純リロケータブルシンボルになります。*symbol* には，*simple_expression* のアドレス値とユーセージタイプ NVBIT が与えられます。

RASU8 は，*simple_expression* の値が NVBIT アドレス空間の範囲内にあるかどうかをチェック

します。指定した *simple_expression* の値が NVBIT アドレス空間の範囲外であった場合，RASU8 はワーニングを表示します。

NOROMWIN 擬似命令が指定されていた場合で，*simple_expression* の値が物理セグメント#0 を示していた場合，RASU8 はワーニングを表示します。

補足

すでに定義されているシンボルを *symbol* に定義することはできません。

simple_expression のユーセージタイプは，NVBIT，NONE または NUMBER だけが許され，それ以外はエラーになります。*simple_expression* のユーセージタイプが NUMBER の場合，*symbol* の物理セグメントアドレスは 0 になります。

ROM ウィンドウ領域が未定の場合で，*simple_expression* の値が物理セグメント#0 を示している場合，ROM ウィンドウ領域の範囲が特定できないため，RASU8 はワーニングを表示します。

例

```
NVBIT_SYM1 NVBIT  8000H.1
NVBIT_SYM2 NVBIT  4:6000H.2
         NVSEG #5 AT  4000H
LABEL:   DS   2
NVBIT_SYM3 NVBIT  LABEL.3
```

この例では，NVBIT 擬似命令を使用して，ユーセージタイプ NVBIT を持つ 3 つのアブソリュートシンボルを定義しています。NVBIT_SYM1 は，物理セグメント#0 のオフセットアドレス 8000H のビット 1 を表すシンボルとなります。NVBIT_SYM2 は，物理セグメント#4 のオフセットアドレス 6000H のビット 2 を表すシンボルとなります。NVBIT_SYM3 は，物理セグメント#5 のオフセットアドレス 4000H のビット 3 を表すシンボルとなります。

5.4 アブソリュートセグメント定義擬似命令

nX-U8 のアセンブリ言語で作成されるプログラムは，複数（1 つ以上）の論理セグメントの集まりとして定義されます。プログラムの中で論理セグメントの種類を切り替えるときに，それを RASU8 に知らせる必要があります。

ここで説明する擬似命令は，これからアブソリュートな論理セグメントに属する記述をしようという場合に指定する命令です。これに対して，リロケータブルセグメントを開始させるには，RSEG擬似命令を使用します。RSEG擬似命令の使用方法は，「5.5 リロケータブルセグメント定義擬似命令」で説明しています。

5.4.1 CSEG 擬似命令

構文

CSEG [#*pseg_addr*][AT *start_address*]

CSEG [#*pseg_addr*] AT *overlay_address* OVL *allocation_address*

説明

CSEG 擬似命令は，アブソリュート CODE セグメントの定義の開始を宣言します。

OVL 記述子付きの定義は，アブソリュートなオーバーレイ用のセグメントを定義する場合に記述します。アブソリュートなオーバーレイ用のセグメントの定義，およびオーバーレイについては，「10 オーバーレイ機能」を参照してください。

CSEG 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラメータ」を参照してください。

5.4.2 DSEG 擬似命令

構文

DSEG [#*pseg_addr*][AT *start_address*]

説明

DSEG 擬似命令は，アブソリュート DATA セグメントの定義の開始を宣言します。

DSEG 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラ

メータ」を参照してください。

5.4.3 BSEG 擬似命令

構文

BSEG [#*pseg_addr*][AT *start_address*]

説明

BSEG 擬似命令は，アブソリュート BIT セグメントの定義の開始を宣言します。

BSEG 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラメータ」を参照してください。

5.4.4 NVSEG 擬似命令

構文

NVSEG [#*pseg_addr*][AT *start_address*]

説明

NVSEG 擬似命令は，アブソリュート NVDATA セグメントの定義の開始を宣言します。

NVSEG 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラメータ」を参照してください。

5.4.5 NVBSEG 擬似命令

構文

NVBSEG [#*pseg_addr*][AT *start_address*]

説明

NVBSEG 擬似命令は，アブソリュート NVBIT セグメントの定義の開始を宣言します。

NVBSEG 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラメータ」を参照してください。

5.4.6 TSEG 擬似命令

構文

TSEG [#*pseg_addr*][AT *start_address*]

説明

TSEG 擬似命令は，アブソリュート TABLE セグメントの定義の開始を宣言します。

TABLE 擬似命令には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。#*pseg_addr* には，定義する論理セグメントの物理セグメントアドレスを指定します。*start_address* には，定義する論理セグメントの開始アドレスを指定します。

パラメータの指定の詳細については，「5.4.7 アブソリュートセグメント定義擬似命令のパラメータ」を参照してください。

5.4.7 アブソリュートセグメント定義擬似命令のパラメータ

それぞれのアブソリュートセグメント定義擬似命令（CSEG，DSEG，BSEG，NVSEG，NVBSEG，TSEG）には，パラメータとして#*pseg_addr*，AT *start_address* を指定することができます。

pseg_addr には，定義する論理セグメントの物理セグメントアドレスを指定します。*pseg_addr* は，物理セグメントアドレスを表す，前方参照を含まない定数式です。

AT に続けて記述する *start_address* には，定義する論理セグメントの開始アドレスを指定します。*start_address* は前方参照を含まない定数式です。AT *start_address* を指定することによって，ロケーションカウンタの値は指定されたアドレス値に更新されます。

AT *start_address* の意味は，*start_address* が数値型の式であるか，アドレス型の式であるかで意味が異なります。*start_address* が数値型の式である場合，この指定はオフセットアドレスだけを指定することを意味します。一方，*start_address* がアドレス型の式である場合，この指定は物理セグメントアドレスとオフセットアドレスの両方を指定することを意味します。このとき，#*pseg_addr* を同時に記述することはできません。言い換えれば，#*pseg_addr* を記述した場合，AT *start_address* に指定する *start_address* は数値型の式に限定されます。

例えば，物理セグメント#1 のオフセットアドレス 1000H 番地を開始アドレスとするアブソリュート CODE セグメントを定義する場合，

```
CSEG #1 AT 1000H
CSEG    AT 1:1000H
```

上の例のような記述は，正しい記述として認められますが，

```
CSEG #1 AT 1:1000H  ;ERROR
```

上の例のような記述をした場合，RASU8 はエラーを表示します。

#pseg_addr および AT *start_address* の指定は，それぞれ省略可能です。これらを省略した場合，同じセグメントタイプの論理セグメントの，直前の設定を引き継ぐという性質があります。

(1) *#pseg_addr* を省略する場合

#pseg_addr を省略すると，同じセグメントタイプの，直前の物理セグメントの設定を引き継ぎます。ただし，これは *start_address* が数値型の式の場合に限られます。

```
CSEG    #2  AT  1000H
.
.
.
CSEG    AT  3000H ; 物理セグメント#2 を継承する
```

最初の CSEG 擬似命令で，物理セグメント#2 を指定しています。次の CSEG 擬似命令では，物理セグメントに関する指定はなく，開始アドレス 3000H だけを指定します。この場合，物理セグメントは#2 に設定されます。

(2) AT *start_address* を省略する場合

AT *start_address* を省略すると，同じセグメントタイプの同じ物理セグメントの，直前のオフセットアドレスを引き継ぎます。

```
DSEG    #4  AT  1000H
DS      10H     ;1010H
CSEG    #1  AT  8000H
.
.
.
DSEG    #4 ; start address 1010H
```

この例では，最初の DSEG 擬似命令で，物理セグメント#4 の開始アドレス 1000H を指定しています。DS 擬似命令で 10H バイトを確保しているので，#4 のロケーションカウンタは 1010H に更新されています。次の DSEG 擬似命令では，物理セグメントアドレスの指定はありますが，開始アドレスの指定はありません。このとき，開始アドレスは直前のロケーションカウンタの値 1010H に設定されます。

(3) パラメータを指定しない場合

パラメータを何も指定しない場合，同じセグメントタイプの物理セグメントの設定とオフセットアドレスを引き継ぎます。

```
DSEG    #4  AT  1000H
DS      10H     ;1010H
DSEG    #1  AT  8000H
DS      20H     ;8020H
.
.
.
DSEG    ; start address 1:8020H
```

最後の DSEG 擬似命令では，物理セグメントも開始アドレスも指定していません。この場合，直前の DSEG 擬似命令の設定をそのまま引き継ぎ，物理セグメント#1 のオフセットアドレス 8020H が設定されます。

5.5 リロケータブルセグメント定義擬似命令

リロケータブルセグメントは，アセンブル時にはその絶対アドレスが確定せず，リンク時にアドレスが確定する論理セグメントです。

リロケータブルセグメントは，次の手順で定義します。

(1) SEGMENT 擬似命令を用いてセグメントシンボルを定義します。

(2) RSEG 擬似命令のオペランドにセグメントシンボルを指定して，リロケータブルセグメントを定義します。

5.5.1 SEGMENT 擬似命令

構文

segment_symbol SEGMENT *segment_type* [*boundary_attr*][*seg_attr*][*relocation_attr*]

説明

SEGMENT 擬似命令はセグメントシンボルを定義します。1 つのソースファイル中に最大 65535 個までのセグメントシンボルを定義できます。

segment_symbol には，定義するセグメントシンボルを定義します。*segment_symbol* は，リロケータブルセグメントの識別に用いられます。RSEG 擬似命令のオペランドには，この *segment_symbol* を指定します。また，*segment_symbol* を命令のオペランドに使用することもできます。この場合，*segment_symbol* はリロケータブルセグメントのベースアドレスを表し，アセンブル時の値は 0 になります。

SEGMENT 擬似命令には，4 つのパラメータがあります。*segment_type* は必ず指定しなければなりません。*boundary_attr* と *seg_attr*，および *relocation_attr* は，必要でなければ省略してもかまいません。次に，それぞれのパラメータの意味を説明します。

segment_type

segment_type には，リロケータブルセグメントを割り付けるアドレス空間の種類を表すセグメントタイプを指定します。次のセグメントタイプの中から 1 つだけ指定できます。

segment_type	意味
CODE	CODE アドレス空間に割り当てます。
DATA	DATA アドレス空間に割り当てます。ただし，SFR 領域を除きます。
BIT	BIT アドレス空間に割り当てます。ただし，SFR 領域を除きます。
NVDATA	NVDATA アドレス空間に割り当てます。
NVBIT	NVBIT アドレス空間に割り当てます。
TABLE	TABLE アドレス空間に割り当てます。

boundary_attr

boundary_attr には，リロケータブルセグメントが割り付けられるときの先頭アドレスの境界値を指定します。これを論理セグメントの境界値属性といいます。*boundary_attr* には，境界値の種類を表すシンボルか，または整定数を指定します。*boundary_attr* 指定の種類とその意味，指定できるセグメントタイプを次に示します。

boundary_attr	意味	セグメントタイプ
UNIT	CODE セグメントは 2 バイト境界。 TABLE，DATA，NVDATA は 1 バイト境界。 BIT，NVBIT は 1 ビット境界。	制限はありません。
WORD	2 バイト境界。	CODE ， TABLE ， DATA，NVDATA
整定数	指定する値を境界とします。 CODE，TABLE，DATA，NVDATA はバイト単位，BIT，NVBIT はビット単位です。 指定できる値は，次の中の 1 つです。 1，2，4，8，16，32，64，128，256，512，1024，2048	制限はありません。 ただし，CODE の場合，1 を指定することはできません。指定した場合にはエラーとなります。

boundary_attr の指定を省略すると，UNIT が自動的に割り当てられます。

seg_attr

seg_attr には，論理セグメントの物理セグメント属性を指定します。

seg_attr	意味
#pseg_addr	対象の論理セグメントを物理セグメント*#pseg_addr* に割り付けます。*#pseg_addr* は，定数式です。
ANY	対象の論理セグメントを割り付ける物理セグメントを限定しません。

#pseg_addr を指定した場合，その物理セグメントには，対象の論理セグメントが割り付けられるメモリが存在しなければなりません。例えば，物理セグメント#1 に ROM だけが実装されているときに，物理セグメント#1 に DATA セグメントや NVDATA セグメントを割り付けようとすると，RASU8 はワーニングを表示します。

seg_attr の指定を省略すると，論理セグメントに割り当てられる物理セグメント属性は，メモリモデル，およびデータモデルの指定により決定します。

セグメントタイプ	論理セグメントに割り当てられる物理セグメント属性
CODE	SMALL モデルの場合，#0 が割り当てられます。 LARGE モデルの場合，ANY が割り当てられます。
TABLE， DATA， BIT，NVDATA，NVBIT	NEAR モデルの場合，#0 が割り当てられます。 FAR モデルの場合，ANY が割り当てられます。

NOROMWIN 擬似命令を指定した場合，TABLE タイプのセグメントシンボルを定義するときに物理セグメント属性に#0 を指定すると，RASU8 はワーニングを表示します。

relocation_attr

relocation_attr には，リロケータブルセグメントを割り付ける領域を指定します。これを論理セグメントの特殊領域属性と呼びます。特殊領域属性の種類とその意味，指定できるセグメントタイプを次に示します。

relocation_attr	意味	セグメントタイプ
DYNAMIC	対象のリロケータブルセグメントを，ダイナミックセグメントとして定義します。この属性が指定されると，すべてのセグメントを割り付けた後，空き領域の中で最大の連続領域が，ダイナミックセグメントに割り当てられます。	DATA
NVRAM	セグメントの割り付け対象領域を不揮発性メモリ領域に指定します。この属性のないリロケータブル CODE セグメントは，ROM 上に割り付けられます。	CODE

例

```
CODE_SEG    SEGMENT CODE
DATA_SEG    SEGMENT DATA
NVDATA_SEG  SEGMENT NVDATA
```

上の例は，もっともシンプルな定義の例です。CODE_SEG は CODE アドレス空間の ROM が実装されている領域に割り付けられます。DATA_SEG は DATA アドレス空間の RAM が実装されている領域に割り付けられます。NVDATA_SEG は NVDATA アドレス空間の不揮発性メモリが実装されている領域に割り付けられます。

```
DATA_SEG1   SEGMENT DATA 1
DATA_SEG2   SEGMENT DATA 8
```

上の例は，境界値属性を指定する例です。DATA_SEG1 では 1 バイト境界上に，DATA_SEG2 は 8 バイト境界上にそれぞれ割り付けられます。DATA_SEG1，DATA_SEG2 ともに RAM が実装されている領域に割り付けられます。

```
CODE_SEG    SEGMENT CODE #2 NVRAM
TABLE_SEG   SEGMENT TABLE 8 #1
DYNAMIC_SEG SEGMENT DATA 2 #3 DYNAMIC
```

上の例では，境界値属性や特殊領域属性などを組み合わせて指定しています。CODE_SEG は，物理セグメント#2 の不揮発性メモリが実装されている領域に割り付けられます。TABLE_SEG は，物理セグメント#1 の ROM が実装されている領域の 8 バイト境界上に割り付けられます。DYNAMIC_SEG は，物理セグメント#3 の RAM が実装されている領域の 2 バイト境界上に割り付けられます。

5.5.2 RSEG 擬似命令

構文

RSEG *segment_symbol*

説明

RSEG 擬似命令は，リロケータブルセグメントを定義します。

segment_symbol には，定義するリロケータブルセグメントを表すセグメントシンボルを指定します。*segment_symbol* は，RSEG 擬似命令の記述位置より前で，SEGMENT 擬似命令で定義されていなければなりません。

リロケータブルセグメントのアセンブル時の開始アドレスは，必ず 0 に設定されます。したがって，RSEG 擬似命令でリロケータブルセグメントの記述を開始するとき，ロケーションカウンタは 0 にセットされます。

1 つのリロケータブルセグメントを複数のブロックに分けて定義することもできます。その場合は，その都度 RSEG 擬似命令を使用して，リロケータブルセグメントを定義することを宣言します。すでに一度以上定義されたリロケータブルセグメントを再び定義すると，ロケーションカウンタの値は，直前の定義の最終アドレスを引き継ぎます。

例

```
VAR_DATA  SEGMENT DATA WORD   ;セグメントシンボルの定義
CODE_0    SEGMENT CODE #0     ;セグメントシンボルの定義
CODE_1    SEGMENT CODE #1     ;セグメントシンボルの定義

        RSEG    VAR_DATA    ;リロケータブル DATA セグメントの定義
BUF1:   DS      4H
```

```
        RSEG    CODE_0       ;リロケータブル CODE セグメントの定義
SUB1:   MOV     ER0,    #00H
        MOV     ER2,    ER0
        LEA     BUF1
        ST      XR0,    [EA]
        RT
        .
        .
        .
        RSEG    VAR_DATA    ;すでに定義されているリロケータブル
                            ;セグメント VAR_DATA を継続します。
BUF2:   DS      10H

        RSEG    CODE_1      ;リロケータブル CODE セグメントの定義
SUB2:   MOV     ER0,    #00H
        ST      ER0,    BUF2
        .
        .
        .
```

この例では，1 つのリロケータブル DATA セグメントと 2 つのリロケータブル CODE セグメントを定義しています。

リロケータブル DATA セグメント VAR_DATA は，DATA アドレス空間の 2 バイト境界に割り付けられます。VAR_DATA は，RSEG 擬似命令を使用して 2 回定義されていますので，2 回目に定義される VAR_DATA の開始アドレスは， 最初に定義された VAR_DATA の最後のアドレスを引き継ぎます。したがって，ラベル BUF2 のアドレスは 4H となります。

リロケータブル CODE セグメント CODE_0 は，CODE アドレス空間の物理セグメント#0 に割り付けられます。

リロケータブル CODE セグメント CODE_1 は，CODE アドレス空間の物理セグメント#1 に割り付けられます。

5.5.3 STACKSEG 擬似命令

構文

STACKSEG *stack_size*

説明

STACKSEG 擬似命令は，スタックセグメントを定義します。

stack_size には，スタックセグメントのサイズを指定します。*stack_size* は，前方参照を含まない定数式です。*stack_size* には偶数値を指定しなければなりません。奇数値を指定した場合，RASU8 はワーニングを表示して，スタックセグメントのサイズを *stack_size*+1 に補正します。

STACKSEG 擬似命令が指定されると，RASU8 は$STACK という名前のスタックセグメントを自動的に生成します。STACKSEG 擬似命令のオペランド *stack_size* に 0 を指定した場合，セグメントシンボル$STACK が定義されるだけでスタック領域は確保されません。$STACK は，リロケータブルセグメントの 1 つですが，RSEG 擬似命令のオペランドに$STACK を指定することはできません。

補足

スタックポインタの初期値，すなわちスタックセグメントの終了アドレスに 1 を加算したアドレスは，_$$SP で参照することができます。リセットベクタの 0 番地には，スタックポインタの初期値を定義しておく必要があります。このときに EXTRN 擬似命令で外部参照宣言を行い，_$$SP を参照します。

```
STACKSEG    200H          ;スタックセグメント($STACK)の定義
EXTRN DATA NEAR:_$$SP     ;_$$SP の外部参照宣言
CSEG    AT  00H           ;リセットベクタの 0 番地に
DW      _$$SP             ;スタックポインタの初期値を定義
```

この例では，スタック領域として 200H バイトを確保し，スタックセグメントの終了アドレスを参照するために，EXTRN 擬似命令で_$$SP を外部参照宣言し，リセットベクタの 0 番地にスタックポインタの初期値を定義しています。

5.6 アドレス制御擬似命令

5.6.1 ORG 擬似命令

構文

ORG *address*

説明

ORG 擬似命令は，所属する論理セグメントのロケーションカウンタの値を，*address* の値に再設定します。所属する論理セグメントがアブソリュートセグメントなのか，リロケータブルセグメントなのかによって，ORG 擬似命令の機能が異なります。それぞれの場合について以下に示します。

(1) アブソリュートセグメントに所属する場合

address には，前方参照を含まない定数式でロケーションカウンタの値を設定します。

定数式は，所属する論理セグメントの先頭アドレス以上の値でなければなりません。また，定数式の値は，対象となるアドレス空間内でなければなりません。定数式がアドレス式の場合，物理セグメントアドレスは，アブソリュートセグメントの物理セグメントアドレスと同じでなければなりません。

```
CSEG #1 AT 1000H
.
.
.
ORG 1030H
.
.
.
ORG 1100H
.
.
.
ORG 200H    ;エラー
```

この例では，物理セグメント#1 のアブソリュート CODE セグメント中で，ORG 擬似命令を使用しています。開始アドレスは，1000H です。したがって，ORG 擬似命令のオペランドの値は，1000H 以上の値でなければなりません。最後の ORG 擬似命令のオペランドの値は 200H ですから，RASU8 はエラーを表示します。

(2) 論理セグメントがリロケータブルセグメントに所属する場合

address には，前方参照を含まない単純式でロケーションカウンタの値を指定します。

単純リロケータブルシンボルが単純式に含まれる場合，その単純リロケータブルシンボルの所属するリロケータブルセグメントは，現在のリロケータブルセグメントでなければなりません。

オペランドが定数式の場合，オペランドの値は，所属するリロケータブルセグメントの先頭アドレスからのオフセットをあらわしています。

```
DATASEG SEGMENT DATA
        RSEG    DATASEG
LABEL1: DS      10H
        ORG     LABEL1+30H
LABEL2: DS      10H
        ORG     100H
LABEL3: DS      10H
```

この例では，リロケータブルセグメント中で，2 つの ORG 擬似命令が使用されています。最初の ORG 擬似命令のオペランドには，所属するリロケータブルセグメントのラベルが使用されています。2 番目の ORG 擬似命令のオペランドは定数式です。この 100H は所属するリロケータブルセグメントの先頭アドレスからのオフセットを表しています。

5.6.2 ALIGN 擬似命令

構文

ALIGN

説明

ALIGN 擬似命令は，カレントロケーションの調整を行います。

カレントロケーションが奇数の場合，領域を 1 バイト確保し，次に続くワード長のデータやラベルが偶数アドレスから割り付けられるように調整します。カレントロケーションが偶数の場合は何もしません。

ALIGN 擬似命令は，CODE セグメント，TABLE セグメント，DATA セグメント，または NVDATA セグメントで記述できます。

この擬似命令は，アブソリュートセグメント，および境界値属性が 2 以上のリロケータブルセグメントでのみ記述可能です。境界値属性が 1 のリロケータブルセグメントの場合，カレントロケーションが奇数か偶数かを特定することができないため，ALIGN 擬似命令を記述した場合，RASU8 はエラーを表示します。

```
Address    Source
-------            CSEG    AT  200H
00:0200    START:  L       ER0,    STR1START
00:0204            L       R1,     [ER0]
```

```
-------             TSEG    AT  300H
00:0300    STR1:    DB      "OddSize"
00:0307             ALIGN
00:0308    STR1START:
00:0308             DW      STR1
```

この例では，TABLE セグメントの中で DB 擬似命令を使用して，奇数バイトの初期化を行っています。そして，その直後に ALIGN 擬似命令を記述しておくと，ラベル STR1START のアドレスは偶数アドレスに調整されます。

補足（CODE セグメントの自動 ALIGNMENT 機能）

CODE セグメントの場合，ALIGN 擬似命令を記述しなくても CODE セグメントのロケーションカウンタは，必ず偶数となるように調整されます。これを自動 ALIGNMENT 機能といいます。

CSEG 擬似命令や ORG 擬似命令によって，ロケーションカウンタに奇数アドレスを指定した場合や，DB 擬似命令や DS 擬似命令で奇数バイト長の領域を確保しようとした場合には，RASU8 はワーニングを表示し，カレントロケーションに 1 を加算して偶数アドレスに調整します。

5.6.3 DS 擬似命令

構文

[*label* :] DS *size*

説明

DS 擬似命令は，*size* で指定するバイト数の領域を，所属する論理セグメントに割り当てます。所属する論理セグメントのロケーションカウンタの値には，指定したサイズが加算されます。

size には，アドレス空間に割り当てるバイト数を定数式で指定します。定数式は，前方参照を含んではいけません。

DS 擬似命令は，CODE セグメント，TABLE セグメント，DATA セグメント，または NVDATA セグメントで記述できます。

CODE セグメントの場合，*size* には偶数を指定しなければなりません。奇数を指定すると RASU8 は，ワーニングを表示します。

```
DATASEG SEGMENT DATA
        RSEG    DATASEG
BUF:    DS      10H
```

```
CODESEG SEGMENT CODE
        RSEG    CODESEG
        MOV     ER0,    #00H
        ST      ER0,    BUF
```

この例では，リロケータブル DATA セグメント DATASEG に，10H バイトの領域を割り当てています。

5.6.4 DBIT 擬似命令

構文

[*label* :] DBIT *size*

説明

DBIT 擬似命令は，*size* で指定するビット数の領域を，所属する論理セグメントに割り当てます。所属する論理セグメントのロケーションカウンタの値には，指定したサイズが加算されます。

size には，アドレス空間に割り当てるビット数を定数式で指定します。定数式は，前方参照を含んではいけません。

DBIT 擬似命令は，BIT セグメント，または NVBIT セグメントで記述できます。

```
BITSEG  SEGMENT BIT
        RSEG    BITSEG
FLAG:   DBIT    8

CODESEG SEGMENT CODE
        RSEG    CODESEG
        SB      FLAG
```

この例では，リロケータブルセグメント BITSEG に，8 ビットの領域を割り当てています。

5.7 コード初期化擬似命令

5.7.1 DB 擬似命令

構文

[*label* :] DB { *expression* | *string_constant* | *duplicate_expression*}

[, { *expression* | *string_constant* | *duplicate_expression* }] …

説明

DB 擬似命令は，バイト単位でメモリを初期化します。オペランドには，*expression*（一般式），*string_constant*（文字列定数），または *duplicate_expression*（デュプリケート式）を指定することができます。オペランドの数に制限はありません。

expression は前方参照を含んでいてもかまいません。*expression* には，1 バイトのデータを指定します。*expression* の値は，-255 から+255 までの範囲内でなければなりません。

オペランドに *string_constant*（文字列定数）を指定した場合，各文字は文字の並びの順にコード化されます。

duplicate_expression は，連続したアドレスの範囲を同じ値で初期化する場合に指定します。

duplicate_expression の構文は，次のようになっています。

duplicate_expression の構文

repeat DUP *expression*

repeat には繰り返す回数を定数式で指定します。*expression* には，初期化を行うための 1 バイトのデータを指定します。

補足

DB 擬似命令は，CODE セグメント，TABLE セグメント，および NVDATA セグメントで記述できます。

例

```
TABLESEG SEGMENT TABLE
         RSEG    TABLESEG
CHAR_TABLE:
         DB      'A', 'B', 'C', 'D', 'E', 'F'
STRING:
         DB      "String"
INIT_TABLE:
         DB      4 DUP 10H  ;DB  10H,10H,10H,10H と同じ
```

この例では，一般式を用いて初期化する例と文字列定数を用いて初期化する例，およびデュ

プリケート式を用いて初期化する例を示しています。

5.7.2 DW 擬似命令

構文

[*label* :] DW { *expression* | *duplicate_expression*} [, { *expression* | *duplicate_expression* }] …

説明

DW 擬似命令は，ワード単位でメモリを初期化します。オペランドには，*expression*（一般式），または *duplicate_expression*（デュプリケート式）を指定することができます。オペランドの数に制限はありません。

expression は前方参照を含んでいてもかまいません。*expression* には，1 ワード（2 バイト）のデータを指定します。*expression* の値は，-65535 から+65535 までの範囲内でなければなりません。

duplicate_expression は，連続したアドレスの範囲を同じ値で初期化する場合に指定します。

duplicate_expression の構文は，次のようになっています。

duplicate_expression の構文

repeat DUP *expression*

repeat には繰り返す回数を定数式で指定します。*expression* には，初期化を行うための 1 ワードのデータを指定します。

補足

DW 擬似命令は，CODE セグメント，TABLE セグメント，および NVDATA セグメントで記述できます。

例

```
TABLESEG SEGMENT TABLE
         RSEG    TABLESEG
TABLE1:
         DW      -3, -2, -1, 0, 1, 2, 3
TABLE2:
         DW      3 DUP 0FFFFH  ;DW  0FFFFH,0FFFFH,0FFFFH と同じ
```

この例では，一般式を用いて初期化する例とデュプリケート式を用いて初期化する例を示しています。

5.7.3 CHKDBDW / NOCHKDBDW 擬似命令

構文

CHKDBDW

NOCHKDBDW

対応する RASU8 のオプション

/ZC

説明

CHKDBDW 擬似命令および NOCHKDBDW 擬似命令は，DB 擬似命令および DW 擬似命令でプログラムコードがアクセス不可能な範囲に配置される場合または配置される可能性がある場合に，ワーニングを出力するかどうかを設定します。

CHKDBDW 擬似命令を指定すると，次の行から，次に NOCHKDBDW 擬似命令を指定する行までの範囲内で，プログラムコードがアクセス不可能な範囲に配置される可能性があるかどうかのチェックを行います。/ZC オプションの指定，未指定に関係なくチェックを行います。

NOCHKDBDW 擬似命令を指定すると，次の行から，次に CHKDBDW 擬似命令を指定する行までの範囲内で，プログラムコードがアクセス不可能な範囲に配置されるかどうかのチェックは行いません。/ZC オプションの指定，未指定に関係なくチェックは行いません。

例

```
CSEG AT 0:0000H
NOCHKDBDW          ; 以降の DB/DW 擬似命令のチェックを行いません。
DW     0F000H      ; SP(スタックポインタ)の初期値
DW     PROG_ENTRY  ; PC(プログラムカウンタ)の初期値
DW     BRK_ENTRY   ; ELEVEL が 0 または 1 の状態での brk 命令実行時の分岐先

CSEG AT 0:0008H
DW     NMI_ENTRY   ; ノンマスカブル割り込み
```

この例は，ベクタテーブルの記述方法です。ベクタテーブルは、L 命令でアクセスする必要はありません。したがって、NOCHKDBDW 擬似命令を記述してワーニングのチェックを抑止して下さい。

例

```
        CHKDBDW             ; 以降の DB/DW 擬似命令をチェックします
        CSEG   AT 0:9000H
SUB_ROUTINE1:
        MOV  R0, #3         ; 入力値(本例では 0～7 を想定)

        /* ジャンプテーブルのアドレスを ER2 へ格納 */
        /* 物理セグメントアドレスは#0 固定          */
        MOV  R2, #BYTE1 OFFSET JMP_TABLE
        MOV  R3, #BYTE2 OFFSET JMP_TABLE

        ADD  R0, R0         ; 1 データあたり 2 バイトのため入力値を 2 倍
        ADD  R2, R0         ; 入力値 × 2 をアドレスに加算
        ADDC R3, #0
        L    ER0, [ER2]     ; ジャンプアドレスを読み込み完了
        B    ER0

JMP_TABLE:              ; ジャンプテーブル
        DW   JMP_ADDR0      ; Warning 42 発生
        DW   JMP_ADDR1      ; Warning 42 発生
        DW   JMP_ADDR2      ; Warning 42 発生
        DW   JMP_ADDR3      ; Warning 42 発生
        DW   JMP_ADDR4      ; Warning 42 発生
        DW   JMP_ADDR5      ; Warning 42 発生
        DW   JMP_ADDR6      ; Warning 42 発生
        DW   JMP_ADDR7      ; Warning 42 発生
             :
```

ジャンプテーブル等のテーブルデータを CODE セグメントに記述する場合プログラムからアクセスする必要のあるデータは CHKDBDW 擬似命令を記述してワーニングのチェックを行います。上記の例ではラベル JMP_TABLE 以降の DW 擬似命令は、ROM WINDOW の範囲外に配置されるのでワーニングが出力されます。この場合、ジャンプテーブルを ROM WINDOW 領域か、物理セグメントアドレス#1 に移動すことで回避できます。

5.8 最適化擬似命令

nX-U8 をコアとするマイクロコントローラには，いくつかの分岐命令があります。マイクロコントローラの命令を直接記述する代わりに，GJMP 擬似命令や GB*cond* 擬似命令を使用すれば，RASU8 は分岐先のアドレス値や分岐先の距離に応じた最適な命令に変換します。

5.8.1 GJMP 擬似命令

構文

[*label* :] GJMP *symbol*

説明

GJMP 擬似命令は，最適な分岐命令に変換されます。

symbol は，分岐先を表すシンボルです。RASU8 は，GJMP 擬似命令を *symbol* が示す分岐先に応じた最適な分岐命令（B 命令または BAL 命令）に変換します。

ただし，RASU8 起動時に/G オプションが指定されていない場合，*symbol* が前方参照シンボルであった場合には，最適化は行われず B 命令に変換されます。/G オプションが指定されている場合には，*symbol* が前方参照シンボルであっても最適化が行われます。

補足

GJMP 擬似命令は CODE セグメントにしか記述できません。

symbol のユーセージタイプは，CODE，NONE，または NUMBER でなければなりません。

例

```
        CSEG  AT  1000H
LABEL1:
        DS      40H
        GJMP    LABEL1  ;BAL 命令に変換される
        .
        .
        .
        CSEG  AT  2000H
        GJMP    LABEL1  ;B 命令に変換される
```

この例では，最初の GJMP 擬似命令では LABEL1 までの距離が-128 ワードから+127 ワードの範囲にあるため，BAL 命令に変換されます。2 番目の GJMP 擬似命令では，LABEL1 までの距離が-128 ワードから+127 ワードの範囲にないため，B 命令に変換されます。

5.8.2 GB*cond* 擬似命令

構文

[*label* :] GBGT *symbol*
[*label* :] GBGE *symbol*
[*label* :] GBNC *symbol*
[*label* :] GBEQ *symbol*
[*label* :] GBZ *symbol*
[*label* :] GBNE *symbol*
[*label* :] GBNZ *symbol*
[*label* :] GBLE *symbol*
[*label* :] GBLT *symbol*
[*label* :] GBCY *symbol*
[*label* :] GBPS *symbol*
[*label* :] GBNS *symbol*
[*label* :] GBLTS *symbol*
[*label* :] GBLES *symbol*
[*label* :] GBGTS *symbol*
[*label* :] GBGES *symbol*
[*label* :] GBOV *symbol*
[*label* :] GBNV *symbol*

説明

GB*cond* 擬似命令は，最適な条件分岐命令に変換されます。この最適化は/G オプションが指定されているときのみ有効となります。GB*cond* 擬似命令が記述されているときに，/G オプションを指定せずにアセンブルすると，RASU8 はエラーを表示します。

symbol は，分岐先を表すシンボルです。RASU8 は，GB*cond* 擬似命令を *symbol* が示す分岐先に応じた最適な条件分岐命令に変換します。

GB*cond* 擬似命令は，次の命令に変換されます。

B*cond* *symbol*

または

B*revcond* *branch_address*

B *symbol*

branch_address:

ここで，B*revcond* とは B*cond* の条件と正反対の条件分岐命令を示します。B*revcond* と B*cond* の対応は，次のとおりです

条件分岐命令	正反対の条件分岐命令
BGT	BLE
BGE or BNC	BLT or BCY
BEQ or BZ	BNE or BNZ
BPS	BNS
BLTS	BGES
BLES	BGTS
BOV	BNV

補足

GB*cond* 擬似命令は CODE セグメントにしか記述できません。

symbol のユーセージタイプは，CODE，NONE，または NUMBER でなければなりません。

例

```
        CSEG  AT  1000H
LABEL1:
        DS      40H
        GBLT    LABEL1  ;BLT 命令に変換される
        .
        .
        .
        CSEG  AT  2000H
        GBLT    LABEL1  ;BGE 2006H, B LABLE1 に変換される
```

この例では，最初の GBLT 擬似命令では LABEL1 までの距離が-128 ワードから+127 ワードの範囲にあるため，BLT 命令に変換されます。2 番目の GBLT 擬似命令では，LABEL1 までの距離が-128 ワードから+127 ワードの範囲にないため，BGE 命令と B 命令に変換されます。

5.9 リンケージ制御擬似命令

リンケージ制御擬似命令は，主にプログラムを複数のファイルに分割して作成する場合に使用します。

5.9.1 複数のファイルによるプログラムの作成

1 つのプログラムを複数のソースファイルに分けて開発する場合，あるソースファイルで定義されるシンボルを，他のソースファイルから参照できるようにするためには，次の宣言が必要です。

(1) シンボルを定義する側のソースファイルで，シンボルを他のソースファイルからも参照できるようにする宣言（パブリック宣言）

(2) シンボルを参照する側のソースファイルで，他のソースファイルで定義されているシンボルを参照する宣言（イクスターナル宣言）

パブリック宣言するシンボルのことを，パブリックシンボルといいます。そして，イクスターナル宣言するシンボルのことをイクスターナルシンボルといいます。

パブリック宣言するためには，PUBLIC 擬似命令を使用します。そして，イクスターナル宣言するためには，EXTRN 擬似命令を使用します。

あるソースファイルにイクスターナルシンボルが宣言されていれば，同じ名前のパブリックシンボルが他のソースファイルに必ず存在しなければなりません。

パブリックシンボルとイクスターナルシンボルの両方の特徴を持ったシンボルとして，共有シンボルがあります。共有シンボルを定義するためには，COMM 擬似命令を使用します。複数のソースファイルで同じ名前の共有シンボルを定義すると，それらの共有シンボルは，共通のメモリ領域を表します。

パブリックシンボルと共有シンボルは，複数のファイルから参照することができます。この意味から，この 2 つのシンボルを合わせて，グローバルシンボルと呼びます。

SEGMENT 擬似命令で定義するセグメントシンボルを，EXTRN 擬似命令でイクスターナル宣言することはできません。あるセグメントシンボルを，複数のソースファイルで使用するためには，それぞれのソースファイルで SEGMENT 擬似命令を使用して，そのシンボルを定義しなければなりません。

パブリックシンボル，イクスターナルシンボル，共有シンボル，およびセグメントシンボルについて、以下に説明します。

5.9.2 PUBLIC 擬似命令

構文

PUBLIC *symbol* [*symbol* …]

説明

PUBLIC 擬似命令は，ローカルシンボルをパブリックシンボルとして宣言します。ローカルシンボルをパブリック宣言することによって，そのシンボルを他のソースファイルから使用できるようになります。

symbol には，ローカルシンボルを指定します。ローカルシンボルの定義と，そのシンボルのパブリック宣言とは，どちらを先に行ってもかまいません。

PUBLIC 擬似命令のオペランドには，複数のシンボルを指定することができます。

補足

パブリックシンボルを他のソースファイルから参照するためには，参照する側のソースファイル中で，EXTRN 擬似命令を使用して同じ名前のイクスターナルシンボルを宣言しなければなりません。

複数のソースファイルで同じ名前のパブリックシンボルを定義することはできません。

SET 擬似命令で再定義しているユーザシンボルをパブリック宣言すると，最後に定義した値を持つパブリックシンボルになります。

例

```
GLOBAL_NUMBER   EQU  1
DATASEG SEGMENT DATA 2
        RSEG    DATASEG
GLOBAL_DATA:
        DS      2
PUBLIC  GLOBAL_NUMBER GLOBAL_DATA
```

この例では，アブソリュートシンボル GLOBAL_NUMBER と単純リロケータブルシンボル GLOBAL_DATA をパブリック宣言しています。

5.9.3 EXTRN 擬似命令

構文

EXTRN *usage_type* [*attribute*] : *symbol* [*symbol* …] [*usage_type* [*attribute*] : *symbol* [*symbol* …]] …

説明

EXTRN 擬似命令はイクスターナルシンボルを宣言します。

usage_type には，イクスターナルシンボルのユーセージタイプを指定します。*usage_type* は，オペランドに別の *usage_type* を指定するまで有効です。次のユーセージタイプの中から，１つだけ指定できます。

usage_type	意味
CODE	CODE アドレス空間上のアドレスを表すシンボル
DATA	DATA アドレス空間上のアドレスを表すシンボル
BIT	BIT アドレス空間上のアドレスを表すシンボル
NVDATA	NVDATA アドレス空間上のアドレスを表すシンボル
NVBIT	NVBIT アドレス空間上のアドレスを表すシンボル
TABLE	TABLE アドレス空間上のアドレスを表すシンボル
TBIT	TABLE アドレス空間上のビットアドレスを表すシンボル
NONE	アドレス空間を指定しないアドレスシンボル
NUMBER	数値を表すシンボル

symbol には，外部参照するシンボルを指定します。イクスターナルシンボルは，他のソースファイル中で宣言されるパブリックシンボル，または共有シンボルを参照します。

attribute には，シンボルの物理セグメント属性を記述します。

attribute の種類と意味は次のとおりです。

attribute	意味
NEAR	対象のシンボルの物理セグメント属性は#0
FAR	対象のシンボルの物理セグメント属性は ANY

attribute は，ユーセージタイプ NUMBER 以外のシンボルで記述ができます。ユーセージタイプ NUMBER のシンボルは数値型であるため，*attribute* を指定することができません。

ユーセージタイプ TABLE，TBIT，DATA，BIT，NVDATA，NVBIT，および NONE のイクスターナルシンボルを定義するときに *attribute* を省略した場合，イクスターナルシンボルの物理セグメント属性はデータモデルに依存します。NEAR モデルの場合は，物理セグメント属性は#0 となります。FAR モデルの場合は，物理セグメント属性は ANY となります。

ユーセージタイプ CODE のイクスターナルシンボルを定義するときに *attribute* を省略した場合，イクスターナルシンボルの物理セグメント属性はメモリモデルに依存します。SMALL モデルの場合は，物理セグメント属性は#0 となります。LARGE モデルの場合は，物理セグメント属性は ANY となります。

補足

すでに定義されているシンボルを，*symbol* に指定することはできません。セグメントシンボルを EXTRN 擬似命令で参照することはできません。

イクスターナルシンボルと，それに対応するパブリックシンボルのユーセージタイプ，および物理セグメント属性は一致していなければなりません。一致していない場合，RLU8 はエラー

を表示します。

例

```
;ファイル 1
    DSEG    AT  0:8000H
NEAR_VAR:
    DS      2H
    DSEG    AT  7:1000H
FAR_VAR:
    DS      2H
PUBLIC NERA_VAR FAR_VAR
```

```
;ファイル 2
EXTRN DATA NEAR : NEAR_VAR
EXTRN DATA FAR  : FAR_VAR

    CSEG    AT  1000H
    MOV     ER0,    #00H
    ST      ER0,    NEAR_VAR
    ST      ER0,    FAR_VAR
```

この例では，ファイル 1 でパブリック宣言したシンボルを，ファイル 2 でイクスターナル宣言しています。

5.9.4 COMM 擬似命令

COMM 擬似命令は共有シンボルを定義します。共有シンボルは，複数のソースファイルで共通のデータ領域を定義するものです。

複数のソースファイルで定義される共有シンボルは，共通のデータ領域の先頭アドレスを表します。共有シンボルが定義するデータ領域のアドレスは，RLU8 が決定します。

共有シンボルは，リロケータブルセグメントと似ている部分がありますが，共有シンボルが確保する領域にラベルを定義したり，初期化を行うことはできません。また，複数のソースファイルで定義されるリロケータブルセグメントが，それぞれのソースファイルで独立した領域を表すのに対して，複数のソースファイルで定義される共有シンボルは，それぞれのソースファイルで共通の領域を表します。

COMM 擬似命令の構文を以下に示します。

構文

communal_symbol COMM *segment_type size* [*seg_attr*]

説明

COMM 擬似命令の構文は，SEGMENT 擬似命令とよく似ています。ただし，*segment_type* の直

後には，確保する領域のサイズを表す *size* を指定します。また，境界値属性の指定はありません。

segment_type

segment_type には，リロケータブルセグメントを割り付けるアドレス空間の種類を表すセグメントタイプを指定します。次のセグメントタイプの中から 1 つだけ指定できます。

segment_type	意味
DATA	DATA アドレス空間に割り当てます。ただし，SFR 領域を除きます。
BIT	BIT アドレス空間に割り当てます。ただし，SFR 領域を除きます。
NVDATA	NVDATA アドレス空間に割り当てます。
NVBIT	NVBIT アドレス空間に割り当てます。
TABLE	TABLE アドレス空間に割り当てます。

seg_attr

seg_attr には，共有シンボルの物理セグメント属性を指定します。

seg_attr	意味
#pseg_addr	対象の共有シンボルを物理セグメント *#pseg_addr* に割り付けます。*#pseg_addr* は，定数式です。
ANY	対象の共有シンボルを割り付ける物理セグメントを限定しません。

#pseg_addr を指定した場合，その物理セグメントには，対象の共有シンボルが割り付けられるメモリが存在しなければなりません。例えば，物理セグメント#1 に ROM だけが実装されているときに，物理セグメント#1 に DATA タイプの共有シンボルや NVDATA タイプの共有シンボルを割り付けようとすると，RASU8 はワーニングを表示します。

seg_attr の指定を省略すると，論理セグメントに割り当てられる物理セグメント属性は，データモデルの指定により決定します。NEAR モデルの場合，#0 が割り当てられます。FAR モデルの場合，ANY が割り当てられます。

NOROMWIN 擬似命令を指定した場合，TABLE タイプの共有シンボルを定義するときに物理セグメント属性に#0 を指定すると，RASU8 はワーニングを表示します。

size

size は共有シンボルが確保する領域のサイズをあらわす整定数です。共有シンボルのサイズの単位は，*segment_type* の種類によって異なります。*segment_type* が DATA，NVDATA，または TABEL の場合はバイト単位であり，BIT または NVBIT の場合はビット単位となります。

COMM 擬似命令には SEGMENT 擬似命令とは異なり，境界値属性の指定はありません。共有シンボルによって確保する領域は，セグメントタイプが TABLE，DATA，または NVDATA であれば *size* が 1 ならば 1 バイト境界に割り付けられ，*size* が 2 以上ならば 2 バイト境界に割り付け

られます。セグメントタイプが BIT または NVBIT ならば 1 ビット境界に割り付けられます。

複数のソースファイルで同じ名前の共有シンボルを定義すると，それぞれのソースファイルで共通の領域を確保することになります。例えば，

```
COM_AREA COMM DATA 2
```

というソースステートメントが，複数のソースファイルで指定される場合，それぞれのソースファイルが，DATA アドレス空間の 2 バイトの領域を共有することになり，そのアドレスを表すシンボルが COM_AREA になります。ここで重要なのは，複数のソースファイルで COM_AREA を定義しても，確保される領域はあくまでも 2 バイトであることです。

補足

1 つのソースファイル中で，同じ共有シンボルを 2 回以上宣言するとエラーになります。

もし，各ソースファイルの共有シンボルのサイズが異なっていれば，指定するサイズの中で最大の領域がメモリに割り付けられます。

他のソースファイルで宣言される共有シンボルを，EXTRN 擬似命令を使用して外部参照してもかまいません。

他のソースファイルでパブリック宣言されているシンボルを，COMM 擬似命令で定義することができますが，この場合はオペランドに指定するサイズの領域は確保されず，単に“他のソースファイルのシンボルを参照する”ことを宣言することになります。すなわち，これは EXTRN 擬似命令を記述することとまったく同等になります。このようなケースは，C コンパイラ CCU8 が作成するソースファイルに存在することもありますが，アセンブリ言語でプログラミングする場合には，あまりお薦めできる使い方ではありません。

例

```
         TYPE (M610001)
GL_BUF1 COMM DATA 100H
GL_BITF COMM BIT  4
         .
         .
         .
         L       ER0,    GL_BUF1
         SB      GL_BITF+2
```

この例では，共有シンボル GL_BUF1 および GL_BITF を宣言し，それらをマイクロコントローラの命令のオペランドに指定しています。GL_BUF1 は DATA アドレス空間に 100H バイトの領域を確保し，GL_BITF は BIT アドレス空間に 4 ビットの領域を確保しています。

5.9.5 パブリック，イクスターナル，および共有シンボルの使用例

5.9.5.1 パブリックシンボルをイクスターナルシンボルで参照する

あるソースファイルで定義されるシンボルを，他のソースファイルでの使用するために，PUBLIC 擬似命令と EXTRN 擬似命令を用いる例を示します。

```
;ソースファイル 1
        TYPE (M610001)
PUBLIC BUF_SIZE
PUBLIC BUF
BUF_SIZE EQU 20H

DATA_SEG SEGMENT DATA 2
        RSEG    DATA_SEG
BUF:    DS      BUF_SIZE
```

```
;   ソースファイル 2
        TYPE (M610001)
EXTRN NUMBER: BUF_SIZE
EXTRN DATA NEAR: BUF

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
PROG1:
        LEA     OFFSET BUF
LOOP:
        MOV     R0,     #BUF_SIZE
        MOV     R1,     #00H
        ST      R1,     [EA+]
        ADD     R0,     #-1
        BNZ     LOOP
        RT
```

この例では，ソースファイル 1 で定義している BUF_SIZE と BUF をソースファイル 2 で使用する例を示しています。

ソースファイル 1 では，2 つのシンボルを PUBLIC 擬似命令でパブリック宣言しています。ソースファイル 2 では，2 つのシンボルを EXTRN 擬似命令でイクスターナル宣言しています。

5.9.5.2 複数のソースファイルで共通の共有シンボルを使用する

複数のソースファイルで，共通のデータ領域を使用するために，それぞれのソースファイルで COMM 擬似命令を用いて共有シンボルを定義する例を示します。

```
;ソースファイル 1
        TYPE (M610001)

GL_BUF1 COMM DATA 2
GL_BUF2 COMM DATA 2
GL_BUF3 COMM DATA 2

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        ST      ER0,    GL_BUF2
        ST      ER0,    GL_BUF3
```

```
;ソースファイル 2
        TYPE (M610001)

GL_BUF1 COMM DATA 2
GL_BUF2 COMM DATA 2
GL_BUF3 COMM DATA 4  ;ソースファイル 1 とサイズが異なる

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        L       ER2,    GL_BUF2
        ST      ER0,    GL_BUF3
        ST      ER2,    GL_BUF3+2
```

この例では，2 つのソースファイルで 3 つの共有シンボル（GL_BUF1，GL_BUF2，GL_BUF3）を定義しています。これらのシンボルは，それぞれ 2 つのソースファイル間で共通のデータ領域を確保します。GL_BUF1 と GL_BUF2 には，それぞれ 2 バイトが割り当てられます。GL_BUF3 は，ソースファイル 1 とソースファイル 2 で，指定するサイズが異なります。この場合には，大きいほうのサイズである 4 バイトが割り当てられます。

5.9.5.3 共有シンボルをイクスターナルシンボルで参照する

あるソースファイルで定義される共有シンボルを，他のソースファイルでも使用するために，EXTRN 擬似命令を用いる例を示します。

```
;ソースファイル 1
        TYPE (M610001)

GL_BUF1 COMM DATA 2 #0
GL_BUF2 COMM DATA 2 #0
GL_BUF3 COMM DATA 2 ANY
```

```
;ソースファイル 2
        TYPE (M610001)

EXTRN   DATA NEAR : GL_BUF1 GL_BUF2
EXTRN   DATA FAR  : GL_BUF3

CODE_SEG SEGMENT CODE
        RSEG    CODE_SEG
        L       ER0,    GL_BUF1
        L       ER2,    GL_BUF2
        ST      ER0,    GL_BUF3
```

この例では，ソースファイル 1 で定義している 3 つの共有シンボル（GL_BUF1，GL_BUF2，GL_BUF3）をソースファイル 2 でも使用するために，ソースファイル 2 で EXTRN 擬似命令を用いてイクスターナル宣言しています。

5.9.6 パーシャルセグメントの使用

複数のソースファイル間で，同じ名前をもつリロケータブルセグメントを定義することができます。複数のソースファイルで，同じ名前をもつリロケータブルセグメントのことをパーシャルセグメントと呼びます。

パーシャルセグメントは，RLU8 により 1 つの論理セグメントに結合されます。この機能を利用することにより，複数のファイルに分割されて割り当てられている領域を，1 つの連続した領域として扱うことができます。以下に例を示します。

```
;ソースファイル 1
        TYPE (M610001)
        ROMWINDOW 0, 3FFFH
DATA_VAR    SEGMENT DATA  2 #0
INIT_TABLE  SEGMENT TABLE 2 #0

        RSEG    DATA_VAR   ; ソースファイル 2 の DATA_VAR と結合される
VAR1:   DS      4
VAR2:   DS      10H

        RSEG    INIT_TABLE ; ソースファイル 2 の INIT_TABLE と結合される
        DW      5678H, 1234H
        DB      0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
PUBLIC  VAR1 VAR2
```

```
;ソースファイル 2
        TYPE (M610001)
        ROMWINDOW 0, 3FFFH
DATA_VAR    SEGMENT DATA  2 #0
INIT_TABLE  SEGMENT TABLE 2 #0
CODE_SEG    SEGMENT CODE

        RSEG    DATA_VAR   ;ソースファイル 1 の DATA_VAR と結合される
VAR3:   DS      2
VAR4:   DS      4

        RSEG    INIT_TABLE ;ソースファイル 1 の INIT_TABLE と結合される
        DW      0
        DW      0FFFFH, 0FFFFH

        RSEG    CODE_SEG
        L       ER0,    SIZE DATA_VAR ;結合後のセグメントのサイズ
        LEA     OFFSET  DATA_VAR      ;結合後の先頭アドレス
        MOV     R2,     #BYTE1 OFFSET INIT_TABLE ;結合後の先頭アドレス
        MOV     R3,     #BYTE2 OFFSET INIT_TABLE ;結合後の先頭アドレス
LOOP:
        L       ER4,    [ER2]
        ST      ER4,    [EA+]
        ADD     ER2,    #2
        ADD     ER0,    #-2
        BNE     LOOP
```

この例では，セグメントシンボル DATA_VAR と INIT_TABLE を，ソースファイル 1 とソースファイル 2 で定義しています。

そして，この 2 つのソースファイルを，RLU8 を使ってリンクすると，論理セグメント DATA_VAR，および INIT_TABLE は，それぞれパーシャルセグメントとして扱われ，連続したアドレス空間に割り付けられます。パーシャルセグメントの結合の順序は，RLU8 のコマンドラインに指定するオブジェクトファイルの順番どおりになります。

また，リロケータブルセグメント CODE_SEG の中では，セグメントシンボル DATA_VAR，および INIT_TABLE を参照しています。これらのシンボルは，結合後のセグメントの先頭アドレスを示します。

5.10 ファイル読み込み擬似命令

5.10.1 INCLUDE 擬似命令

構文

INCLUDE (*include_file*)

説明

INCLUDE 擬似命令は，*include_file* で指定されるインクルードファイルを読み込みます。INCLUDE 擬似命令が記述されると，その位置にインクルードファイルの内容が挿入されます。

インクルードファイルの中でさらに INCLUDE 擬似命令を使用して，別のファイルを挿入できます。最大 8 レベルまで，INCLUDE 擬似命令をネストできます。

インクルードファイル中に END 擬似命令がある場合，RASU8 はそのインクルードファイル内の END 擬似命令以降のアセンブルを中止し，インクルードファイルを呼び出したソースファイルのアセンブル処理に戻ります。

補足

RASU8 の/I オプションを使用して，インクルードファイルを格納しているディレクトリを指定することができます。

例

```
;ソースファイル
    INCLUDE(DEFINE.INC)
    CSEG AT 0:0H
    DW      _$$SP
    DW      START
    .
    .
    .
```

```
;インクルードファイル（DEFINE.INC）
    TYPE (M610001)
    MODEL SMALL, NEAR
    ROMWINDOW 0, 7FFFH
```

この例では，インクルードファイル DEFINE.INC の中で，アセンブラ初期設定の擬似命令（TYPE 擬似命令，MODEL 擬似命令，ROMWINDOW 擬似命令）を記述しています。

5.11 マクロ定義擬似命令

5.11.1 DEFINE 擬似命令

構文

DEFINE *symbol* ”*macro_body*”

説明

DEFINE 擬似命令は，マクロシンボルを定義します。

DEFINE 擬似命令は，*symbol* に対して *macro_body* を割り当てます。この定義以降で，ソースステートメント中に *symbol* が現れると，RASU8 はそれを *macro_body* に置き換えてアセンブルを行います。*macro_body* に指定できる文字数は，最大 255 バイトです。

macro_body の中に，別のマクロシンボルを記述してもかまいません。この場合，RASU8 は，最初のマクロを置きかえる過程で，さらにマクロの置き換えを行います。マクロのネストは，最大 8 レベルまで許されています。

補足

マクロシンボルは，DEFINE 擬似命令で定義した後でないと参照できません。

例

```
DEFINE  CODESEG     ”SEGMENT CODE”
DEFINE  CLR4BYTE    ”MOV ER0, #0¥nST ER0, [EA+]¥nST ER0, [EA]”

PROG1   CODESEG     ;PROG1   SEGMENT CODE

        RSEG    PROG1
        LEA     8000H
        CLR4BYTE    ;MOV ER0, #0
                    ;ST  ER0, [EA+]
                    ;ST  ER0, [EA]
```

この例では，CODESEG と CLR4BYTE という 2 つのマクロシンボルを定義しています。

CLR4BYTE のマクロ定義の例で示すように，複数の命令を 1 つのマクロシンボルとして定義することもできます。

RASU8 は，マクロを使用した記述を，コメントで示すとおりに解釈します。

5.12 条件アセンブル擬似命令

条件アセンブル機能を使用すると，プログラムのあるブロックを，ある一定の条件が満たされるときだけアセンブルするような制御を行うことができます。その結果，1 つのソースプログラムを複数の目的に利用することができるようになります。

条件アセンブル機能は，条件アセンブル擬似命令を記述することによって実現します。条件アセンブル擬似命令の構文は次のとおりです。

IF*xxx conditional_operand*

true_conditional_body

[ELSE

false_conditional_body]

ENDIF

IF*xxx* は，次の条件アセンブル擬似命令のうちのどれかを表します。

IF　　IFDEF　　IFNDEF

conditional_operand は，条件アセンブルの真偽条件を与える式やシンボルです。*conditional_operand* に指定する内容は，条件アセンブル擬似命令によって異なります。

true_conditional_body と *false_conditional_body* は，ソースステートメントのブロックを表します。条件が真であった場合，*true_conditional_body* のステートメントブロックがアセンブルされます。条件が偽であった場合，*true_conditional_body* のステートメントブロックは読み飛ばされます。このとき，ELSE 擬似命令がある場合は，*false_conditional_body* がアセンブルされます。

true_conditional_body や *false_conditional_body* に，さらに条件アセンブル擬似命令を記述してもかまいません。条件アセンブル擬似命令は，最大 15 レベルまでネストすることができます。

5.12.1 IF 擬似命令

構文

IF *expression*

説明

expression は，前方参照を含まない定数式です。

expression の値が 0 以外であれば条件は真と判定されます。*expression* の値が 0 であれば条件は偽と判定されます。また，*expression* が前方参照を含む場合や *expression* に文法的な誤りがある場合，条件は偽と判定されます。

例

```
;PROG_SW EQU 0
;PROG_SW EQU 1
PROG_SW EQU 2
        .
        .
        .
IF PROG_SW == 2
        BUF_SIZE1 EQU   100H
        BUF_SIZE2 EQU   200H
ELSE
        BUF_SIZE1 EQU   200H
        BUF_SIZE2 EQU   400H
ENDIF

        DSEG  AT  8000H
BUF1:   DS      BUF_SIZE1
BUF2:   DS      BUF_SIZE2
```

この例では，条件判定の式として PROG_SW == 2 を指定しています。プログラムの最初で PROG_SW には 2 を与えているので，この条件は真となります。したがって，BUF_SIZE1 には 100H，BUF_SIZE2 には 200H がセットされます。

5.12.2 IFDEF 擬似命令

構文

IFDEF *symbol*

説明

symbol は，予約語を除くシンボルです。

symbol がこのソースステートメント以前に定義されていれば，条件は真になります。*symbol* がプログラムの中で定義されていないか，またはこのソースステートメント以降で定義されていれば，条件は偽になります。

例

```
PROG_SW2 EQU 0
        .
        .
        .
IFDEF PROG_SW1
        INCLUDE (INIT1.INC)
ELSE
        INCLUDE (INIT2.INC)
ENDIF
```

この例では，IFDEF 擬似命令のオペランドの PROG_SW1 は定義されていないので，この条件は偽になります。したがって，INIT2.INC がインクルードされることになります。

注意

マクロシンボルを条件アセンブル擬似命令のオペランドに指定するときは，注意が必要です。例えば，次の様にプログラムを記述した場合，

```
DEFINE  SW      "SYM1"
IFDEF   SW  ; IFDEF SYM1 と解釈される
        INCLUDE (FILE1.INC)
ELSE
        INCLUDE (FILE2.INC)
ENDIF
```

このプログラムは，IFDEF SW を“シンボル SW が定義されていれば”という意図で記述されていたとします。しかし，実際は，SW はマクロシンボルなので，RASU8 は SW の本体である SYM1 が定義されているかどうかで条件を判定してしまいます。SYM1 は定義されていないので，条件は偽となり，FILE2.INC がインクルードされます。

5.12.3 IFNDEF 擬似命令

構文

IFNDEF *symbol*

説明

symbol は，予約語を除くシンボルです。

IFNDEF 擬似命令の条件判定は，IFDEF 擬似命令とまったく逆です。

symbol がこのソースステートメント以前に定義されていれば，条件は偽になります。*symbol* がプログラムの中で定義されていないか，またはこのソースステートメント以降で定義されていれば，条件は真になります。

例

```
PROG_SW2 EQU 0
        .
        .
        .
IFNDEF PROG_SW1
        INCLUDE (INIT1.INC)
ELSE
        INCLUDE (INIT2.INC)
ENDIF
```

この例では，IFNDEF 擬似命令のオペランドの PROG_SW1 は定義されていないので，この条件は真になります。したがって，INIT1.INC がインクルードされることになります。

5.13 C デバッグ情報擬似命令

C デバッグ情報擬似命令は，ソースプログラムを C 言語で記述した場合の，C ソースレベルのデバッグ情報を定義するものです。C デバッグ情報擬似命令は，C コンパイラ CCU8 で/SD オプションを指定したときに，CCU8 が自動的に生成するものであり，プログラマが使用するものではありません。

ここでは，情報公開の意味でそれぞれの C デバッグ情報擬似命令について説明していきますが，通常のアセンブリソースに，これらの擬似命令を記述したり，CCU8 が生成するソースプログラムに含まれるこれらの擬似命令を書き換えたりした場合，正常なアセンブル結果が得られるかどうか，またその後のデバッグ作業が正常に行えるかどうかは保証できませんのでご注意ください。

5.13.1 CFILE 擬似命令

構文

CFILE *file_id total_line* ”*filename*”

説明

CFILE 擬似命令は，C ソースファイルのファイルに関する情報を定義するものです。

5.13.2 CFUNCTION / CFUNCTIONEND 擬似命令

構文

CFUNCTION *fn_id*

CFUNCTIONEND *fn_id*

説明

これらの擬似命令は，C ソースプログラムの関数に関する情報を定義するものです。CFUNCTION 擬似命令は関数の開始を示し，CFUNCTIONEND 擬似命令は関数の終了を示します。

5.13.3 CARGUMENT 擬似命令

構文

CARGUMENT *attrib size offset* ”*variable_name*” *hierarchy*

説明

CARGUMENT 擬似命令は，C ソースプログラムの関数の引数に関する情報を定義するものです。

5.13.4 CBLOCK / CBLOCKEND 擬似命令

構文

CBLOCK *fn_id block_id c_source_line*

CBLOCKEND *fn_id block_id c_source_line*

説明

これらの擬似命令は，C ソースプログラム中での 1 つの関数，および関数内のブロック記述の開始と終了を定義するものです。

CBLOCK 擬似命令はブロックの開始位置を定義します。CBLOCKEND 擬似命令はブロックの終了位置を定義します。

5.13.5 CLABEL 擬似命令

構文

CLABEL *label_no* ”*label_name*”

説明

CLABEL 擬似命令は，C ソースプログラム上に記述されたラベルと，C コンパイラ CCU8 が出力するアセンブリソース中のラベルとを関連付けるものです。

5.13.6 CLINE / CLINEA 擬似命令

構文

CLINE *line_attr line_no start_column end_column*

CLINEA *file_id line_attr line_no start_column end_column*

説明

これらの擬似命令は，C ソースプログラムの行番号に関する情報を定義するものです。

5.13.7 CGLOBAL 擬似命令

構文

CGLOBAL *usg_typ attrib size* ”*variable_name*” *hierarchy*

説明

CGLOBAL 擬似命令は，C ソースプログラム中で定義したグローバル変数に関する情報を定義するものです。

variable_name は C ソースプログラム上のグローバル変数の名前です。*variable_name* の先頭にアンダースコア（_）を付けたシンボルが，パブリックシンボルまたは共有シンボルとしてアセンブリソースファイルに出力されます。

5.13.8 CSGLOBAL 擬似命令

構文

CSGLOBAL *usg_typ attrib size* ”*variable_name*” *hierarchy*

説明

CSGLOBAL 擬似命令は，C ソースプログラム中で定義した静的グローバル変数に関する情報を定義するものです。

variable_name は C ソースプログラム上の静的グローバル変数の名前です。*variable_name* の先頭にアンダースコア（_）を付けたシンボルが，ローカルシンボルとしてアセンブリソースファイルに出力されます。

5.13.9 CLOCAL 擬似命令

構文

CLOCAL *attrib size offset block_id* ”*variable_name*” *hierarchy*

説明

CLOCAL 擬似命令は，C ソースプログラム中で定義したローカル変数に関する情報を定義するものです。

5.13.10 CSLOCAL 擬似命令

構文

CSLOCAL *attrib size alias_no block_id* ”*variable_name*” *hierarchy*

説明

CSLOCAL 擬似命令は，C ソースプログラム中で定義した静的ローカル変数に関する情報を定義するものです。

variable_name は C ソースプログラム上の静的ローカル変数の名前です。アセンブリソース上のシンボルは，“_$ST”の後ろに *alias_no* で示される 10 進数を付加したものとなります。例えば，

```
CSLOCAL 43H 0002H 000AH 0001H "static_local" 02H 00H 01H
```

と記述されていた場合，静的ローカル変数 static_local は，アセンブリソース上のローカルシン

ボル“_$ST10”に対応していることを示します。

5.13.11 CSTRUCTTAG / CSTRUCTMEM 擬似命令

構文

CSTRUCTTAG *fn_id block_id st_id total_mem total_size ”tag_name”*

CSTRUCTMEM *attrib size offset ”member_name” hierarchy*

説明

これらの擬似命令は，C ソースプログラム上に記述された構造体に関する情報を定義するものです。

CSTRUCTTAG 擬似命令は構造体のタグについて定義するもので，CSTRUCTMEM 擬似命令は直前に記述された CSTRUCTTAG のメンバを定義するものです。

5.13.12 CUNIONTAG / CUNIONMEM 擬似命令

構文

CUNIONTAG *fn_id block_id un_id total_mem total_size ”tag_name”*

CUNIONMEM *attrib size ”member_name” hierarchy*

説明

これらの擬似命令は，C ソースプログラム上に記述された共用体に関する情報を定義するものです。

CUNIONTAG 擬似命令は共用体のタグについて定義するもので，CUNIONMEM 擬似命令は直前に記述された CUNIONTAG のメンバを定義するものです。

5.13.13 CENUMTAG / CENUMMEM 擬似命令

構文

CENUMTAG *fn_id block_id emu_id total_mem ”tag_name”*

CENUMMEM *value ”member_name”*

説明

これらの擬似命令は，C ソースプログラム上に記述された列挙型に関する情報を定義するものです。

CENUMTAG 擬似命令は列挙型のタグについて定義するもので，CENUMMEM 擬似命令は直前に記述された CENUMTAG の列挙子を定義するものです。

tag_name は共用体のタグ名を示します。*member_name* は構造体中のメンバ名を示します。

5.13.14 CTYPEDEF 擬似命令

構文

CTYPEDEF *fn_id block_id attrib* ”*type_name*” *hierarchy*

説明

CTYPEDEF 擬似命令は，C ソースプログラム中の typedef によって定義された型に関する情報を定義するものです。

5.13.15 CVERSION 擬似命令

構文

CVERSION *version_number*

説明

CVERSION 擬似命令は，C コンパイラ CCU8 のバージョン情報を定義するものです。

5.13.16 CRET 擬似命令

構文

CRET *offset*

説明

CRET 擬似命令は，戻り番地がスタックに退避された場合のスタックポインタからのオフセット情報を定義するものです。

5.14 エミュレーションライブラリ指定擬似命令

エミュレーションライブラリ指定擬似命令は，C コンパイラ CCU8 により自動的に出力されるものです。プログラマが直接指定するものではありません。

ここでは，情報公開の意味でエミュレーションライブラリ指定擬似命令について説明していきますが，通常のアセンブリソースに，これらの擬似命令を直接記述したり，CCU8 が生成するソースプログラムに含まれるこれらの擬似命令を書き換えたりした場合，正常なリンク結果が得られるかどうかは保証できませんのでご注意ください。

5.14.1 FASTFLOAT 擬似命令

構文

FASTFLOAT

説明

この擬似命令は，リンクする浮動小数演算用のエミュレーションライブラリの種類を，RLU8 に対して指示するものです。

この擬似命令が記述されると，高速化した浮動小数演算用のエミュレーションライブラリがリンクされます。ただし，この場合，浮動小数点演算をすべて単精度演算とするため，演算精度は落ちることになります。

5.15 リスティング制御擬似命令

リスティング制御擬似命令は，出力ファイルを生成するかしないか，ファイルを生成する場合の出力先のファイル指定，プリントファイルの各リストの出力制御，および書式制御などを指定します。

5.15.1 OBJ / NOOBJ 擬似命令

構文

OBJ [(*object_file*)]

NOOBJ

対応する RASU8 のオプション

/O[(*object_file*)]

/NO

説明

これらの擬似命令は，オブジェクトファイルの作成の制御を行います。

OBJ 擬似命令を指定すると，オブジェクトファイルが作成されます。*object_file* には，オブジェクトファイル名を指定します。

NOOBJ 擬似命令を指定すると，オブジェクトファイルは作成されません。

OBJ 擬似命令と NOOBJ 擬似命令を省略した場合，オブジェクトファイルは作成されます。このとき，オブジェクトファイル名は，ソースファイル名の拡張子を“.OBJ”に変えたものになります。

補足

これらの擬似命令は，どちらかをただ一度しか指定できません。そして，最初に指定されたものが有効になります。また，擬似命令の指定よりもオプション指定が優先されます。

5.15.2 PRN / NOPRN 擬似命令

構文

PRN [(*print_file*)]

NOPRN

対応する RASU8 のオプション

/PR[(*print_file*)]

/NPR

説明

これらの擬似命令は，プリントファイルの作成の制御を行います。

PRN 擬似命令を指定すると，プリントファイルが作成されます。*print_file* には，プリントファイル名を指定します。

NOPRN 擬似命令を指定すると，プリントファイルは作成されません。

PRN 擬似命令と NOPRN 擬似命令を省略した場合，プリントファイルは作成されます。このとき，プリントファイル名は，ソースファイル名の拡張子を“.PRN”に変えたものになります。

補足

これらの擬似命令は，どちらかをただ一度しか指定できません。そして，最初に指定されたものが有効になります。また，擬似命令の指定よりもオプション指定が優先されます。

5.15.3 ERR / NOERR 擬似命令

構文

ERR [(*error_file*)]

NOERR

対応する RASU8 のオプション

/E[(*error_file*)]

/NE

説明

これらの擬似命令は，エラーファイルの作成の制御を行います。

ERR 擬似命令を指定すると，エラーファイルが作成され，そのファイルにエラーメッセージが出力されます。*error_file* には，エラーファイル名を指定します。

NOERR 擬似命令を指定すると，エラーファイルは作成されません。エラーメッセージは標準エラー出力に出力されます。

ERR 擬似命令と NOERR 擬似命令を省略した場合，エラーファイルは作成されず，エラーメッセージは標準エラー出力に出力されます。

補足

これらの擬似命令は，どちらかをただ一度しか指定できません。そして，最初に指定されたものが有効になります。また，擬似命令の指定よりもオプション指定が優先されます。

5.15.4 DEBUG / NODEBUG 擬似命令

構文

DEBUG

NODEBUG

対応する RASU8 のオプション

/D

/ND

説明

これらの擬似命令は，オブジェクトファイルに対するアセンブリレベルデバッグ情報の出力の制御を行います。

DEBUG 擬似命令を指定すると，オブジェクトファイルにアセンブリレベルデバッグ情報が出力されます。

NODEBUG 擬似命令を指定すると，アセンブリレベルデバッグ情報は出力されません。

DEBUG 擬似命令が指定されないかぎり，アセンブリレベルデバッグ情報はオブジェクトファイルに出力されません。

補足

これらの擬似命令は，どちらかをただ一度しか指定できません。そして，最初に指定されたものが有効になります。また，擬似命令の指定よりもオプション指定が優先されます。

5.15.5 LIST / NOLIST 擬似命令

構文

LIST

NOLIST

対応する RASU8 のオプション

/L

/NL

説明

これらの擬似命令は，プリントファイルに対するアセンブリリストの出力の制御を行います。

アセンブリリストは，プログラムとそれに対応するオブジェクトコードのリストです。LIST 擬似命令，NOLIST 擬似命令を使用して，プログラムのどの範囲をアセンブルリストに出力するかを指定することができます。

LIST 擬似命令を記述すると，次の行からのプログラムがアセンブルリストに出力されます。

NOLIST 擬似命令を記述すると，次の行からのプログラムはアセンブルリストに出力されなくなります。ただし，エラーまたはワーニングを含むソースステートメントはアセンブルリストに出力されます。

RASU8 は，プログラムの先頭に LIST 擬似命令が指定されているものとしてアセンブルします。したがって，LIST 擬似命令と NOLIST 擬似命令を指定していない場合，すべてのプログラムはアセンブリリストに出力されます。

補足

RASU8 で/L オプションを指定した場合，プログラム中で NOLIST 擬似命令が現れるまでの各ステートメントがアセンブルリストに出力されます。すなわち，プログラムの先頭に LIST 擬似命令を記述したのと同じ意味を持ちます。

RASU8 で/NL オプションを指定した場合，プログラム中で LIST 擬似命令が現れるまでの各ステートメントはアセンブルリストに出力されません。すなわち，プログラムの先頭に NOLIST 擬似命令を記述したのと同じ意味を持ちます。

5.15.6 SYM / NOSYM 擬似命令

構文

SYM

NOSYM

対応する RASU8 のオプション

/S

/NS

説明

これらの擬似命令は，プリントファイルに対するシンボルリストの出力の制御を行います。

シンボルリストは，プログラムに使用しているユーザシンボルの情報を内容とするリストです。SYM 擬似命令，NOSYM 擬似命令を使用して，シンボルリストを出力するかどうかを指定します。

SYM 擬似命令を指定すると，すべてのユーザシンボルの情報がシンボルリストに出力されます。NOSYM 擬似命令を指定すると，シンボルリストは作成されません。

SYM 擬似命令も NOSYM 擬似命令も指定しない場合，シンボルリストは出力されません。

補足

これらの擬似命令は，最初に指定されたものが有効になります。また，擬似命令の指定よりもオプション指定が優先されます。

5.15.7 REF / NOREF 擬似命令

構文

REF

NOREF

対応する RASU8 のオプション

/R

/NR

説明

これらの擬似命令は，プリントファイルに対するクロスリファレンスリストの出力の制御を行います。

クロスリファレンスリストは，プログラムで定義されるユーザシンボルとそれぞれのユーザシンボルが使用される行番号を示すリストです。REF 擬似命令，NOREF 擬似命令を使用して，クロスリファレンスリストに出力するユーザシンボルを制御します。

REF 擬似命令を指定すると，次の行から，次に NOREF 擬似命令を指定する行までの範囲内で，定義または参照しているシンボルについて，クロスリファレンスリストを作成します。

NOREF 擬似命令を指定すると，次の行から，次に REF 擬似命令を指定する行までの範囲内で，定義または参照しているシンボルについては，クロスリファレンスリストを作成しません。

補足

RASU8 で/R オプションを指定した場合，プログラム中で NOREF 擬似命令が現れるまでの各ステートメントがクロスリファレンスリストに出力されます。すなわち，プログラムの先頭に REF 擬似命令を記述したのと同じ意味を持ちます。

RASU8 で/NR オプションを指定した場合，プログラム中で REF 擬似命令が現れるまでの各ステートメントはクロスリファレンスリストに出力されません。すなわち，プログラムの先頭に NOREF 擬似命令を記述したのと同じ意味を持ちます。

5.15.8 PAGE 擬似命令

PAGE 擬似命令は，オペランドがない場合と，オペランドがある場合とで異なった機能を持ちます。ここでは，それぞれについて説明します。

5.15.8.1 オペランドなしの PAGE 擬似命令

構文

PAGE

説明

オペランドのない PAGE 擬似命令は，プリントファイルを強制的に改ページします。

PAGE 擬似命令の置かれている行から次のページになります。

補足

NOLIST 擬似命令が有効である範囲では，PAGE 擬似命令は無視されます。

5.15.8.2 オペランド付きの PAGE 擬似命令

構文

PAGE [*page_length*][, *page_width*]

対応する RASU8 のオプション

/PL*page_length*

/PW*page_width*

説明

オペランドのある PAGE 擬似命令は，プリントファイルの各ページの行数と各行の文字数を指定します。*page_length* には 1 ページの行数を指定し，*page_width* には 1 行の文字数を指定します。*page_length* と *page_width* は前方参照を含まない定数式です。

page_length と *page_width* は，どちらか一方だけを指定してもかまいませんが，両方を省略して PAGE だけを指定すると，改ページを行ってしまいます。

page_length の値の範囲は，10 から 65535 までです。10 より小さい値を指定すると 10 を指定したものとみなされ，65535 より大きい値を指定すると 65535 を指定したものとみなされます。*page_length* の初期設定値は 60 です。

page_width の値の範囲は，79 から 132 までです。79 より小さい値を指定すると 79 を指定したものとみなされ，132 より大きい値を指定すると 132 を指定したものとみなされます。*page_width* の初期設定値は 79 です。

補足

オペランド付きの PAGE 擬似命令はプログラムで一度だけしか指定できません。

5.15.9 DATE 擬似命令

構文

DATE ”*charcter_string*”

説明

DATE 擬似命令は，プリントファイルの日時の欄に出力される文字列を指定します。

charcter_string には，25 文字以内の文字列を指定します。*charcter_string* に 25 文字を超える文字列を指定すると，26 文字目以降は無視されます。

プログラム中に，複数の DATE 擬似命令を使用できますが，最後に指定したものだけが有効になります。DATE 擬似命令を省略すると，プリントファイルの日時の欄には RASU8 を起動したときの日時が出力されます。

5.15.10 TITLE 擬似命令

構文

TITLE ”*character_string*”

説明

TITLE 擬似命令は，プリントファイルのタイトルを指定します。このタイトルは，プリントファイルの各ページのヘッダに出力されます。

character_string には，タイトルとする 70 文字以内の文字列を指定します。 *charcter_string* に 70 文字を超える文字列を指定すると，71 文字目以降は無視されます。

プログラム中に，複数の TITLE 擬似命令を使用できますが，最後に指定したものだけが有効になります。TITLE 擬似命令を省略すると，プリントファイルのヘッダには，タイトルとしては，なにも出力されません。

5.15.11 TAB 擬似命令

構文

TAB [*tab_width*]

対応する RASU8 のオプション

/T[*tab_width*]

説明

TAB 擬似命令を指定すると，プログラムに使用されているタブコードを，指定したタブの幅に見合った文字数分のスペースに置き換えて，アセンブルリストを出力します。

tab_width には，タブ幅を 1 から 15 までの定数式で指定します。*tab_width* を省略すると，“TAB 8”を指定したのと同じになります。また，*tab_width* の値が 1 から 15 までの範囲になかった場合も，“TAB 8”を指定したのと同じになります。

TAB 擬似命令は，プログラム中に何度でも指定できますが，最初の指定だけが有効になります。また，擬似命令の指定よりも，オプション指定が優先されます。

5.16 データアクセス制御擬似命令

データアクセス制御擬似命令は，使用可能なデータメモリ空間を制御します。物理セグメント#0 のみに限定します。

5.16.1 NOFAR 擬似命令

構文

NOFAR

説明

NOFAR 擬似命令を指定すると、アセンブリファイルに対し、RASU8 はデータメモリ空間を 1 つに限定します。すなわち、オブジェクトファイルへ出力されるデータメモリ空間は物理セグメント#0 のみに限定されます。

NOFAR 擬似命令が記述されたアセンブリファイルに対し、RASU8 は FAR データアクセスを許可しません。FAR データアクセスを検出した場合、RASU8 はエラーを出力します。

6 RASU8

6.1 概要

RASU8 は 8 ビット RISC プロセッサである nX-U8 コアを搭載したマイクロコントローラ対応のリロケータブルアセンブラです。この章では，RASU8 の操作方法および機能について説明します。

RASU8 は DCL ファイルの内容を参照してソースファイルをアセンブルします。DCL ファイルは，対象のマイクロコントローラに固有な情報を持ったファイルです。このファイルを交換することにより，RASU8 は，それぞれのマイクロコントローラに対応できます。ソースファイルとは，nX-U8 コア対応のアセンブリ言語で記述されたプログラムです。

アセンブルの結果，RASU8 は次の 4 つのファイルを作成します。

1.オブジェクトファイル

2.プリントファイル

3.エラーファイル

4.EXTRN 宣言ファイル

オブジェクトファイルには，再配置可能なオブジェクトコードと，リンク，デバッグに必要な情報が含まれています。

プリントファイルには，ソースファイルの内容と，生成されたオブジェクトコードが含まれています。さらに，ソースファイルで使用されているシンボルの名前と値を示すことができます。

エラーファイルは，エラーメッセージとエラーの生じたソースステートメントからなるもので，指定がなければ画面に表示されます。

EXTRN 宣言ファイルは，プログラムで定義されるパブリックシンボルに対応する EXTRN 宣言のリストを内容とするファイルです。

6.2 ファイル指定のデフォルト

RASU8 を使用する場合，入力および出力ファイルの指定が必要です。ファイルは，コマンドラインや擬似命令のオペランドで指定します。RASU8 のファイル指定には次の種類があります。

1. ソースファイルの指定
2. インクルードファイルの指定
3. オプション定義ファイルの指定
4. ABL ファイルの指定
5. オブジェクトファイルの指定
6. プリントファイルの指定
7. エラーファイルの指定
8. EXTRN 宣言ファイルの指定
9. DCL ファイルの指定

上記のファイル指定において，ドライブとディレクトリは省略できます。ソースファイル，インクルードファイル，およびオプション定義ファイルの指定以外では，ベース名も省略できます。ドライブ，ディレクトリ，ベース名，または拡張子を省略した場合のデフォルトは次のとおりです。

<table>
<tr><th>ファイル指定</th><th>ドライブ</th><th>ディレクトリ</th><th>ベース名</th><th>拡張子</th></tr>
<tr><td>ソースファイル</td><td rowspan="2">カレントドライブ</td><td rowspan="2">ドライブのカレントディレクトリ</td><td rowspan="4">省略不可</td><td>.ASM</td></tr>
<tr><td>オプション定義ファイル</td><td rowspan="2">省略不可</td></tr>
<tr><td>インクルードファイル</td><td rowspan="2">カレントドライブ(注 1)</td><td rowspan="2">ドライブのカレントディレクトリ(注 1)</td></tr>
<tr><td>DCL ファイル</td><td>.DCL(固定)</td></tr>
<tr><td>ABL ファイル</td><td rowspan="5">カレントドライブ
(注 2)</td><td rowspan="5">ドライブのカレントディレクトリ(注 2)</td><td rowspan="5">ソースファイル指定のベース名</td><td>.ABL</td></tr>
<tr><td>オブジェクトファイル</td><td>.OBJ</td></tr>
<tr><td>プリントファイル</td><td>.PRN</td></tr>
<tr><td>エラーファイル</td><td>.ERR</td></tr>
<tr><td>EXTRN 宣言ファイル</td><td>.EXT</td></tr>
</table>

（注1） ドライブ，ディレクトリが共に省略された場合，またはドライブを省略しディレクトリが円記号(¥)以外の文字で始まる場合，各ファイルのサーチパスが適用されます。インクルードファイルのサーチパスについては，「6.5.2.6 /I」を，DCL ファイルのサーチパスについては「5.1.1 TYPE 擬似命令」を参照してください。

（注2） ドライブ，ディレクトリ，ベース名，および拡張子がすべて省略された場合には，ソースファイル指定のドライブ，またはソースファイル指定のディレクトリになります。

6.3 RASU8 の操作方法

ここでは，RASU8 を実行する方法を説明します。

DOS のプロンプトが表示されている状態で RASU8 とタイプし，その後にソースファイルとオプションを指定しリターンキーを押します。コマンドラインの書式は次のとおりです。

RASU8 [*options*] *source_file* [*options*]

source_file には，アセンブルするソースファイルを指定します。*options* には，オプションまたは応答ファイル指定を組み合わせて使用します。 オプションを表す英字の前には，スラッシュ（/）を付けなければなりません。オプションとソースファイル，オプションとオプションの間には空白文字を挿入してください。

source_file を指定せずに，RASU8 とだけタイプしてリターンキーを押すと，RASU8 の使い方とオプションの一覧が画面に表示されたあと DOS プロンプトに戻ります。

例

/S オプションをつけて，ソースファイル MAIN.ASM をアセンブルする場合は，次のようにタイプします。

```
RASU8  MAIN.ASM /S
```

ソースファイル名の拡張子が省略された場合には，RASU8 は拡張子“.ASM”を付けて処理します。ソースファイル名のドライブが省略された場合には，RASU8 はカレントドライブにソースファイルがあるとみなします。ソースファイル名のディレクトリが省略された場合には，RASU8 はカレントディレクトリにソースファイルがあるとみなします。コマンドラインを正しく入力すると，RASU8 の起動メッセージが画面に表示されます。その後，次のメッセージが順番に表示されます。

```
[dcl_file] loading...
  pass1...
  pass2...
```

また，/ G オプションをつけてアセンブルすると，画面表示は次のようになります。

```
[dcl_file] loading...
  pass1...
  branch optimization...
  pass2...
```

RASU8 はアセンブル処理の最初に，DCL ファイルをロードします。DCL ファイルをロードしている間は，“[*dcl_file*] loading...”が表示されます。*dcl_file* は，実際にロードしている DCL ファイル名です。

RASU8 のアセンブル処理は，パス 1，パス 2 と呼ばれる処理に分かれています。RASU8 は，パス 1 処理において，シンボルの値とプログラムのアドレスを決定しようとします。パス 2 の処理では，パス 1 の結果を使用して，オブジェクトファイルを作成します。パス 1 の処理が始まる

と，“pass1...”が表示され，パス 2 の処理が始まると，“pass2...”が表示されます。また，/G オプションが指定されると，パス 1 終了後，パス 2 の処理開始前に分岐命令の最適化が始まり，“branch optimization...”が表示されます。

作成したプログラムにエラーがあれば，その後にエラーメッセージが表示されます。エラーメッセージについては，「6.9 エラーメッセージ」を参照してください。

アセンブルが終了すると，RASU8 は次のようなメッセージを表示し DOS プロンプトに戻ります。

```
Print  File : MAIN.prn
Object File : MAIN.obj
Error  File : Console

Errors   : 0
Warnings : 0  (/Wrpeast)
Lines    : 100
Assembly End.
```

最初の 3 行は作成された各ファイルの名前です。Print File に続けてプリントファイル名が，Object File に続けてオブジェクトファイル名が，Error File に続けてエラーファイル名（通常は画面表示を表わす“Console”）が表示されます。

ファイル名の表示に続いて，アセンブル結果の情報が表示されます。Errors の後にはエラーの総数が，Warnings の後にはワーニングの総数が表示されます。/W の後には，RASU8 がチェックしたワーニングの種類が表示されます。Lines の後には，ソースファイルの行数が表示されます。

参考

RASU8 が画面に表示するメッセージは，すべて標準出力デバイスに出力されています。DOS のリダイレクト機能を使用すれば，メッセージをファイルに出力することができます。

また，ソースプログラムのエラーメッセージおよびワーニングメッセージだけをファイルに出力する場合には，/E オプションまたは ERR 擬似命令を使用して下さい。

6.4 オプション定義ファイルによるオプションの指定

ソースファイルの指定やオプションをコマンドラインに記述する代わりに，テキストファイルからオプションを読み込む方法もあります。このテキストファイルのことをオプション定義ファイルと呼びます。

6.4.1 オプション定義ファイルの指定方法

オプション定義ファイルを指定するには，単価記号(@)に続けてオプション定義ファイルを指定します。単価記号(@)とオプション定義ファイルの指定の間には空白文字を挿入する事は出来

ません。

例 1

ソースファイル MAIN.ASM をアセンブルするのに必要なオプションがオプション定義ファイル FOO.OPT に記述されている場合，次のようにタイプします。

RASU8 MAIN.ASM @FOO.OPT

例 2

ソースファイル MAIN.ASM をアセンブルするために必要なソースファイルの指定，オプションの記述が BAR.OPT に記述されておりいるが，それに加えて/S オプションを追加してアセンブルしたい場合，次のようにタイプします。

RASU8 @BAR.OPT /S

6.4.2 オプション定義ファイルの書式

オプション定義ファイルでは次の要素の記述が可能です。

1. ソースファイルの指定
2. 各種オプションの指定
3. コメント記述

各要素は空白(20H)，TAB(09H)，LF（0AH）のいずれかで区切ります。CR(0DH)は読み飛ばされます。

１行に記述できるオプションの数や文字数に制限はありません。

コメントを記述する事も出来ます。ファイル中にセミコロン(;)，シャープ(#)，// のいずれか現れると以降，LF(0AH)までをコメントと解釈して読み飛ばします。 ブロックコメントは使用できません。

例

以下は MAIN.ASM を/E，/R，/NL，/Xextrn.ext オプション付きでアセンブルするためのオプション定義ファイルの一例です。

```
;----------------------------------------
;   オプション定義ファイルのサンプル(BAR.OPT)
;----------------------------------------
  MAIN.ASM        ;ソースファイルの指定
  /E              ;エラーファイルの出力を有効にする
  /R /NL          ;プリントファイルの出力項目変更
  /Xextrn.ext     ;EXTRN ファイルを"extrn.ext"で作成する
```

6.5 オプション

オプションを指定することにより，RASU8 の動作や出力ファイルの形式を制御することができます。すべてのオプションは，オプション先頭文字で始まり，オプション名が続きます。オプションの種類によってはその後にパラメータを指定できるものもあります。

オプション先頭文字はスラッシュ（/）またはハイフォン(-)のどちらを指定してもかまいません。便宜のため，以降の文章ではスラッシュ(/)を使用して説明していきます。

オプション名は，大文字，小文字のどちらでも使用できます。オプション先頭文字とオプション名，オプション名とパラメータの間にスペースを挿入することはできません。いくつかのオプションについては，まったく同じ機能を持つ擬似命令が存在します。

6.5.1 オプション一覧

RASU8 が用意するオプションを以下に示します。

オプションとそれに対応する擬似命令の両方を指定しない場合の動作は，デフォルトの欄に示されています。アスタリスク（*）は，そのオプションの機能がデフォルトで指定されることを表します。数値の場合は，オプションの機能の設定値がその数値であることを表します。

オプション	デフォルト	対応する擬似命令	
/MS	*	MODEL SMALL	メモリモデルを SMALL モデルにする。
/ML		MODEL LARGE	メモリモデルを LARGE モデルにする。
/DN	*	MODEL NEAR	データモデルを NEAR モデルにする,
/DF		MODEL FAR	データモデルを FAR モデルにする。
/CD	*		ユーザシンボルの大文字と小文字の区別を行う。
/NCD			ユーザシンボルの大文字と小文字の区別を行わない。
/SL[*symbol_length*]	32		アセンブラが認識するユーザシンボルの文字数を指定する。
/W[*warning_type*]	*		チェックするワーニングの種類を指定する。
/NW[*warning_type*]			チェックしないワーニングの種類を指定する。
/I*include_path*			インクルードファイルの検索パスを指定する
/DEF*symbol*[=*body*]		DEFINE	マクロシンボルを定義する。
/KE /KEUC			ソースファイル中の全角文字 EUC コードでチェックする。
/G			GJMP 擬似命令の前方参照最適化を有効にし, GB*cond* 擬似命令を記述可能にする。

オプション	デフォルト	対応する擬似命令	
/PR[*print_file*]	*	PRN	プリントファイルを作成する。
/NPR		NOPRN	プリントファイルを作成しない。
/A[*abl_file*]			アブソリュートプリントファイルを作成する。
/L	*	LIST	アセンブリリストを作成する。
/NL		NOLIST	アセンブリリストを作成しない。
/S		SYM	シンボルリストを作成する。
/NS	*	NOSYM	シンボルリストを作成しない。
/R		REF	クロスリファレンスリストを作成する
/NR	*	NOREF	クロスリファレンスリストを作成しない。
/PW*page_width*	79	PAGE , *page_width*	プリントファイルの 1 行あたりの文字数を指定する。
/NPW			プリントファイルの 1 行あたりの文字数制限を解除する。
/PL*page_length*	60	PAGE *page_length*	プリントファイルの 1 ページあたりの行数を指定する。
/NPL			プリントファイルの 1 ページあたり行数制限を解除する。
/T[*tab_width*]	8	TAB [*tab_width*]	タブコードを置き換える。

オプション	デフォルト	対応する擬似命令	
/O[*object_file*]	*	OBJ[(*object_file*)]	オブジェクトファイルを作成する。
/NO			オブジェクトファイルを作成しない
/SD			C ソースレベルのデバッグ情報をオブジェクトファイルに出力する。
/D			アセンブリレベルデバッグ情報をオブジェクトファイルに出力する。
/ND	*		アセンブリレベルデバッグ情報をオブジェクトファイルに出力しない。
/E[*error_file*]		ERR[(*error_file*)]	エラーメッセージをファイルに出力する。
/NE	*	NOERR	エラーメッセージを画面に出力する。
/X[*extrn_file*]			EXTRN 宣言ファイルを作成する。
/BRAM(*start_address*,*end_address*)			データメモリ空間に追加した RAM の領域を指定する。
/BROM(*start_address*,*end_address*)			プログラムメモリ空間に追加した ROM の領域を指定する。
/BNVRAM(*start_address*,*end_address*)			データメモリ空間に追加した不揮発性メモリの領域を指定する。
/BNVRAMP(*start_address*,*end_address*)			プログラムメモリ空間の物理セグメントアドレス #0 に追加した不揮発性メモリの領域を指定する。
/ZC			DB 擬似命令および DW 擬似命令でプログラムコードがアクセス不可能な範囲に配置されるかチェックする

6.5.2 各オプションの機能

6.5.2.1 /MS，/ML

構文

/MS

/ML

説明

これらのオプションは，アプリケーションプログラムで使用するメモリモデルの種類を設定します。

/MS オプションは SMALL メモリモデル，/ML オプションは LARGE メモリモデルを設定します。メモリモデルの設定を行わない場合，SMALL メモリモデルが選択されます。

メモリモデルについては，「2.6 メモリモデル」を参照してください。

対応する擬似命令

これらのオプションを指定する代わりに，MODEL 擬似命令を使ってメモリモデルを設定することもできます。オプションと MODEL 擬似命令の両方でメモリモデルを設定する場合，オプションの設定が優先されます。

MODEL 擬似命令については，「5.1.2 MODEL 擬似命令」を参照してください。

例

ソースファイル FOO.ASM を LARGE メモリモデルでアセンブルする場合は次のようにタイプします。

```
RASU8  FOO.ASM  /ML
```

補足

/ML と/MS を同時に指定する事は出来ません。

6.5.2.2 /DN，/DF

構文

/DN

/DF

説明

これらのオプションは，アプリケーションプログラムで使用するデータモデルの種類を設定します。

/DN オプションは NEAR データモデル，/DF オプションは FAR データモデルを設定します。データモデルの設定を行わない場合，NEAR データモデルが選択されます。

データモデルについては，「2.7 データモデル」を参照してください。

対応する擬似命令

これらのオプションを指定する代わりに，MODEL 擬似命令を使ってデータモデルを設定することもできます。オプションと MODEL 擬似命令の両方でデータモデルを設定する場合，オプションの設定が優先されます。

MODEL 擬似命令については，「5.1.2 MODEL 擬似命令」を参照してください。

例

ソースファイル FOO.ASM を FAR データモデルでアセンブルする場合は次のようにタイプします。

```
RASU8  FOO.ASM  /DF
```

補足

/DN と/DF を同時に指定する事は出来ません。

6.5.2.3 /CD，/NCD

構文

/CD

/NCD

説明

/CD オプションを指定すると，ソースファイルにおいて，シンボルに使われている英字の大文字と小文字は区別されます。この場合，シンボルの綴りが同じでも大文字と小文字の使いわけが 1 文字でも異なっていれば，異なるシンボルになります。

/NCD オプションを指定すると，シンボルに使われている英字の大文字と小文字は区別されません。この場合，シンボルの綴りが同じであれば，大文字と小文字の使いわけが異なっていても同じシンボルになります。/NCD オプションを指定すると，RASU8 はシンボルに使われている英字を大文字に変換してから取り扱います。プリントファイルやオブジェクトファイルには，このように変換された名前でシンボルの情報が格納されます。

デフォルトでは，英字の大文字と小文字は区別されます。

英字の大文字と小文字の区別の制御ができるのは，ラベルやセグメント名などプログラムの中で定義されるユーザシンボル，および DCL ファイルの中で定義される SFR シンボルだけです。

命令や擬似命令などの予約語は，オプションの指定に関らず，英字の大文字と小文字は区別されません。

例

ソースファイル FOO.ASM を大文字と小文字を区別しないでアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /NCD
```

このときに，ソースファイル FOO.ASM が次の記述を含む場合，シンボル未定義のエラーは発生しません。

```
     CSEG
     L    R0,   UCSYM
     SB
     DSEG AT  0:200H
UcSym:
     DS   10H
```

これは，シンボル UcSym，および UCSYM は英字の大文字と小文字の組み合わせが異なりますが，綴りは同じのため，RASU8 がこれらを同じシンボルとして扱うからです。/NCD オプションを指定しないでアセンブルする場合は，未定義エラーが発生します。

補足

/CD と/NCD を同時に指定する事は出来ません。

6.5.2.4 /SL

構文

/SL[*symbol_length*]

説明

RASU8 V1.60 からは，ソースファイルにおけるシンボルとして認識するデフォルトの文字数を 32 文字から 255 文字に拡張しました。

これに伴い，/SL オプションで指定する文字数は無視され，RASU8 は常に 255 文字までを認識します。

6.5.2.5 /W，/NW

構文

/W[*warning_type*]

/NW[*warning_type*]

説明

RASU8 がアセンブル時にチェックするワーニングは，いくつかの種類に分類されています。/W オプションと/NW オプションは，特定の種類のワーニングを有効にしたり，逆に特定の種類のワーニングを無効にしたりする場合に使用します。

/W オプションは，*warning_type* で示される種類のワーニングを有効にします。

/NW オプションは，*warning_type* で示される種類のワーニングを無効にします。

warning_type は，ワーニングの種類を表わす文字の組み合わせです。ワーニングの種類と，そのワーニングを表わす文字の関係を以下に示します。

ワーニングの種類を表す文字	ワーニングの内容
R	リロケータブルセグメントの定義に関するチェック
P	擬似命令の記述に関するチェック
E	式の記述に関するチェック
A	アドレッシングの記述に関するチェック
S	SFR アクセス属性に関するチェック
T	ROMWINDOW 領域の指定に関するチェック

warning_type を省略する場合は，すべてのワーニングの種類を指定することと同じです。つまり/W だけを指定する場合は，すべてのワーニングがチェックされ，/NW だけを指定する場合は，ワーニングのチェックはいっさい行われません。

デフォルトでは全てのワーニングがチェックされます。

具体的なワーニングメッセージの内容と，それぞれが属するワーニングの種類については，「6.9.2.3 ワーニングメッセージ」を参照してください。

例

ソースファイル FOO.ASM を，擬似命令の記述に関するワーニングチェックと式に関するワーニングチェックを行わずにアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM  /NWPE
```

6.5.2.6 /I

構文

/I*include_path*

説明

/I オプションは，INCLUDE 擬似命令で読み込むファイルのパスを指定します。/I オプションを複数記述することにより，複数のパスを指定できます。

RASU8 は，次の順番でインクルードファイルをサーチします。

(1) カレントディレクトリからインクルードファイルをサーチします。カレントディレクトリに目的のファイルが存在すれば，そのファイルを読み込みます。

(2) カレントディレクトリに目的のファイルが存在しなかった場合，もし/I オプションでインクルードファイルのパスが指定されていたら，そのパスから目的のファイルをサーチします。もし/I オプションが複数指定されていた場合，記述した順にファイルをサーチします。

INCLUDE 擬似命令についての詳細は，「5.10.1 INCLUDE 擬似命令」を参照してください。

例

ソースファイル FOO.ASM をアセンブルする時に，インクルードファイルをカレントディレクトリ，C:¥USR¥SHARE¥INC，C:¥USR¥PRV¥INC という順にサーチして読み込む場合は，次のようにタイプします。

```
RASU8  FOO.ASM  /IC:¥USR¥SHARE¥INC  /IC:¥USR¥PRV¥INC
```

6.5.2.7 /DEF

構文

/DEF*symbol*[=*body*]

説明

/DEF オプションはマクロシンボルを定義します。*symbol* と等記号（=）の間，等記号（=）と *body* の間に空白文字を挿入する事は出来ません。

=*body* を指定すると，マクロシンボル *symbol* に対して *body* を割り当てます。=*body* を省略すると，マクロボディには”1”が割り当てられます。

対応する擬似命令

このオプションを指定する代わりに，DEFINE 擬似命令を使ってマクロシンボルを定義することも可能です。

例

ソースファイル FOO.ASM をアセンブルする時に，マクロシンボル READDCL にマクロボディ“TYPE(M610001)”を定義し，マクロシンボル ONE に“1”を定義場合は，次のようにタイプします。

```
RASU8  FOO.ASM  /DEFREADDCL=TYPE(M610001)  /DEFONE
```

6.5.2.8 /KE，/KEUC

構文

/KE

/KEUC

説明

これらのオプションを指定すると，ソースファイルに含まれている全角文字は EUC コードである事を前提として アセンブルを行います。

このオプションを指定しなかった場合，ソースファイルに含まれている全角文字はシフト JIS コードであると解釈します。

本アセンブラでは次の順序で現れる文字を全角文字と解釈します。

	シフト JIS コード	EUC コード （/KE，/KEUC 指定時）
全角文字の第 1 バイト	81H～9FH 0E0H～0FCH	0B0H～0F4H 0A1H～0A8H
全角文字の第 2 バイト	40H～0FCH	0A0H～0FEH

6.5.2.9 /G

構文

/G

説明

/G オプションは，GB*cond* 擬似命令を利用可能にし，GJMP 擬似命令と GB*cond* 擬似命令の最適化処理を行うことを指定します。

/G オプションを指定しない場合，GJMP 擬似命令のオペランドに指定したシンボルが後方参照の時，分岐元であるカレントロケーションから分岐先のアドレスが相対分岐の範囲内にあるかどうかチェックを行います。そうでない場合は B 命令と同等のコードが生成されます。

/G オプションを指定した場合の最適化アルゴリズムはシンボルの参照方向に依存しません。分岐元と分岐先が連続した論理セグメントであれば相対範囲内であるかどうかのチェックを行います。

ここでいう「連続した論理セグメント」とは，どちらも同じセグメントタイプのアブソリュートセグメントか同じリロケータブルセグメントで，その間にそのセグメントのカレントロケーションが AT オペランド付きの CSEG 擬似命令や ORG 擬似命令によって設定されていない状態を指します。

例 1

以下に示した例に現れる GJMP 擬似命令とその分岐先ラベルは，全て「連続した論理セグメント」です。GJMP CLAB3 とラベル CLAB3 の間には ORG 擬似命令が存在しますが，これは DATA セグメントのカレントロケーションを変更しているだけなので，これも連続しているといえます。

```
     TYPE(M610001)
     CSEG AT 0:1000H
     MODEL LARGE
CLAB1:
     .
     .
     .
     GJMP CLAB1

COD_SEG   SEGMENT CODE
     RSEG COD_SEG
     GJMP CLAB2
     GJMP CLAB3
CLAB2:

     DEG AT 8000H
     ORG 100H
     .
     .
     .

     RSEG COD_SEG
CLAB3:
     .
     .
     .
```

例 2

以下の例に現れる GJMP 擬似命令と，その分岐先ラベルは全て連続していません。したがって，これらの GJMP 擬似命令は全て B 命令と同等のコードが生成されます。

```
CLAB1    CODE 0:1060H

     CSEG AT 0:1000H
CLAB2:
     GJMP CLAB1

     ORG 1020H
     GJMP CLAB2
     GJMP CLAB3

     CSEG AT 0:1040H
CLAB3:

COD_SEG1 SEGMENT CODE
     RSEG COD_SEG1
     GJMP CLAB4
     GJMP CLAB1

COD_SEG2 SEGMENT CODE
     RSEG    COD_SEG2
CLAB4:
     .
     .
     .
```

6.5.2.10 /PR，/NPR

構文

/PR[*print_file*]

/NPR

説明

/PR オプションを使用すると，プリントファイルが作成されます。*print_file* には，プリントファイル名を指定します。オペランドを省略する場合，およびファイル指定の一部を省略する場合のデフォルトについては，「6.2　ファイル指定のデフォルト」を参照してください。

/NPR オプションを使用すると，プリントファイルは，作成されません。ただし，/A オプションを同時に指定している場合には，たとえ/NPR の指定があっても，プリントファイルは作成されます。

/PR オプションと/NPR オプションを省略する場合は，プリントファイルは作成され，プリン

トファイル名は，ソースファイル名の拡張子を“.PRN”に変えたものになります。

対応する擬似命令

/PR オプションを指定する代わりに，プログラム中に PRN 擬似命令を記述してもかまいません。また，/NPR オプションを指定する代わりに，プログラム中に NOPRN 擬似命令を記述してもかまいません。オプションと擬似命令の両方を指定する場合には，オプションの指定が優先されます。

PRN 擬似命令と NOPRN 擬似命令については，「5.15.2 PRN / NOPRN 擬似命令」を参照してください。

例

```
RASU8  FOO.ASM /PROUTPUT.LST
```

この例では，OUTPUT.LST というプリントファイルの作成を指定しています。

```
RASU8  FOO.ASM /NPR
```

この例では，プリントファイルを作成しないことを指定しています。

補足

/PR と/NPR を同時に指定する事は出来ません。

6.5.2.11 /A

構文

/A[*abl_file*]

説明

/A オプションは，アブソリュートプリントファイルを作成する場合に指定します。

アブソリュートプリントファイルとは，不確定なマシンコード情報やアドレス情報をまったく含まず，すべての情報が確定しているプリントファイルのことです。

abl_file には，ABL ファイル名を指定します。ABL ファイルとは，アブソリュートプリントファイルを作成するために必要な情報を持つバイナリ形式のファイルで，RLU8 によって作成されます。

/A オプションを使用する場合でも，/PR オプションによるファイル名の指定方法は変わりません。ただし，通常のプリントファイルのデフォルト拡張子は“.PRN”ですが，アブソリュートプリントファイルのデフォルト拡張子は“.APR”になります。

アブソリュートプリントファイルの作り方の詳細は，「11. アブソリュートリスティング機能」を参照してください。

例

ソースファイル FOO.ASM のアブソリュートプリントファイルを作成する場合は，次のようにタイプします。この例では，ABL ファイル APRINFO.ABL を読み込みます。

```
RASU8  FOO.ASM /AAPRINFO
```

6.5.2.12 /L，/NL

構文

/L

/NL

説明

/L オプションを指定すると，プログラム中で NOLIST 擬似命令が現れるまでの各ステートメントがアセンブリリストに出力されます。

/NL オプションを指定すると，プログラム中で LIST 擬似命令が現れるまでの各ステートメントはアセンブリリストに出力されません。ただし，エラーを含むステートメントは，たとえ/NL オプションが指定されていてもアセンブリリストに出力されます。

デフォルトでは，/L が指定されます。

アセンブリリストはプリントファイルに出力されます。出力される内容は「6.7 プリントファイル」にて説明されています。LIST 擬似命令と NOLIST 擬似命令については，「5.15.5 LIST / NOLIST 擬似命令」を参照してください。

対応する擬似命令

これらのオプションと LIST 擬似命令，NOLIST 擬似命令の機能はほとんど同じです。ただしオプションの指定が，プログラムの先頭から有効であるのに対して，LIST 擬似命令と NOLIST 擬似命令は，記述した行以降のステートメントに対して効力を持ちます。

例

ソースファイル FOO.ASM の内容をアセンブリリストに出力して，アセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /L
```

ソースファイル FOO.ASM の内容をアセンブリリストに出力しないでアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /NL
```

補足

/L と/NL を同時に指定する事は出来ません。

6.5.2.13 /S，/NS

構文

/S

/NS

説明

/S オプションを指定すると，すべてのユーザシンボルの情報がシンボルリストに出力されます。/NS オプションを指定すると，シンボルリストは作成されません。

デフォルトでは，シンボルリストは作成されません。

シンボルリストはプリントファイルに出力されます。出力される内容は「6.7 プリントファイル」に説明されています。

対応する擬似命令

/S オプションを指定する代わりに，プログラム中に SYM 擬似命令を記述してもかまいません。また，/NS オプションを指定する代わりに，プログラム中に NOSYM 擬似命令を記述してもかまいません。オプションと擬似命令の両方を指定した場合には，オプションの指定が優先されます。SYM 擬似命令と NOSYM 擬似命令については，「5.15.6 SYM / NOSYM 擬似命令」を参照してください。

例

ソースファイル FOO.ASM に使用されているすべてのシンボルをシンボルリストに出力してアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /S
```

シンボルリストを作成しないで，ソースファイル FOO.ASM の内容をアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /NS
```

補足

/S と/NS を同時に指定する事は出来ません。

6.5.2.14 /R，/NR

構文

/R

/NR

説明

/R オプションを指定すると，すべてのユーザシンボルについて，そのシンボルが使用されている行番号がクロスリファレンスリストに出力されます。/NR オプションを指定すると，クロスリファレンスリストは作成されません。

正確に言えば，クロスリファレンスリストの作成には，プログラム中に記述される REF 擬似命令と NOREF 擬似命令が影響します。/R オプションが指定されていても，プログラム中に NOREF 擬似命令が記述された場合には，その行から REF 擬似命令が現れるまでの行の間の行番号は，クロスリファレンスリストには出力されません。逆に，/NR オプションが指定されていても，プログラム中に REF 擬似命令が記述された場合には，その行から NOREF 擬似命令が現れるまでの行の間の行番号は，クロスリファレンスリストに出力されます。つまり，REF 擬似命令と NOREF 擬似命令は，クロスリファレンスリストの出力のスイッチの役割を持っています。

しかし，通常上記のような使い方をすることはほとんどありません。したがって，/R オプションはクロスリファレンスリストを作成し，/NR オプションはクロスリファレンスリストを作成しない，と考えて差し支えありません。

デフォルトでは，クロスリファレンスリストは作成されません。

クロスリファレンスリストはプリントファイルに出力されます。出力される内容は「6.7 プリントファイル」に説明されています。REF 擬似命令と NOREF 擬似命令については，「5.15.7 REF / NOREF 擬似命令」を参照してください。

対応する擬似命令

これらのオプションと REF 擬似命令，NOREF 擬似命令の機能はほとんど同じです。ただしオプションの指定が，プログラムの先頭から有効であるのに対して，REF 擬似命令と NOREF 擬似命令は，記述した行以降のステートメントに対して効力を持ちます。

例

ソースファイル FOO.ASM に使用されているシンボルのクロスリファレンスリストを作成してアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /R
```

クロスリファレンスリストを作成しないで，ソースファイル FOO.ASM をアセンブルする場合は，次のようにタイプします。

```
RASU8  FOO.ASM /NR
```

補足

/R と/NR を同時に指定する事は出来ません。

6.5.2.15 /PW，/NPW

構文

/PW*page_width*

/NPW

説明

/PW オプションは，プリントファイルの各行の文字数を指定します。

page_width には，各行の文字数を整定数で指定します。ここでいう文字数とは，1 バイト文字の数です。*page_width* には，次の範囲の整定数を指定できます。

79 ～ 132

79 より小さい値を指定する場合には，プリントファイルの各行の文字数は，79 になります。132 より大きい値を指定する場合には，プリントファイルの各行の文字数は，132 になります。

/NPW オプションは，プリントファイルの各行の文字数制限による改行を抑制します。このオプションにより，1 行が 132 文字よりも多い行でも改行なしに出力が可能です。

デフォルトでは，プリントファイルの各行の文字数は，79 に設定されています。

対応する擬似命令

/PW オプションを指定する代わりに，プログラム中に PAGE 擬似命令を記述してもかまいません。PAGE 擬似命令の第 2 オペランドに文字数を指定することによって，/PW オプションと同じ設定を行うことができます。

/NPW オプションを指定する代わりに，プログラム中に NOPAGE 擬似命令を記述する事も可能です。ただし，NOPAGE 擬似命令は，同時に各ページの行数による改ページ制御までも抑制してしまいます。NOPAGE 擬似命令では，各ページの行数制限はそのままで各行の文字数制限だけを解除することは出来ないので注意が必要です。

PAGE 擬似命令と，/PW または/NPW オプションの両方を指定する場合には，オプションの指定が優先されます。

例

```
RASU8  FOO.ASM /PW132
```

この例では，プリントファイルの 1 行あたりの文字数を 132 文字に指定しています。

```
RASU8  FOO.ASM /NPW
```

この例では，プリントファイルの各行の文字数制限による改行を抑制しています。

補足

/PW と/NPW を同時に指定する事は出来ません。

6.5.2.16 /PL，/NPL

構文

/PL*page_length*

/NPL

説明

/PL オプションは，プリントファイルの各ページの行数を指定します。

page_length には，各ページの行数を整定数で指定します。この行数は，プリントファイルのヘッダやその前後の空白行なども含んだものです。*page_length* には，次の範囲の整定数を指定できます。

10 ～ 65535

10 より小さい値を指定する場合には，プリントファイルの各ページの行数は，10 になります。65535 より大きい値を指定する場合には，プリントファイルの各ページの行数は，65535 になります。

/NPL オプションは各ページの行数制限を抑制します。これにより，各リスト出力が 65535 行よりも多くなってもページヘッダやページフッタが出力されず，連続したリスト出力が可能です。ただし，/NPL オプションは各ページの行数による改ページ制御を解除するだけで，オペランドを持たない PAGE 擬似命令による改ページや，各リスト出力の終了に行われる改ページは抑制できません。

デフォルトでは，プリントファイルの各ページの行数は，60 に設定されています。

対応する擬似命令

/PL オプションを指定する代わりに，プログラム中に PAGE 擬似命令を記述してもかまいません。PAGE 擬似命令の第 1 オペランドに文字数を指定することによって，/PL オプションと同じ設定を行うことができます。

/NPL オプションを指定する代わりに，プログラム中に NOPAGE 擬似命令を記述する事も可能です。ただし，NOPAGE 擬似命令は，同時に 1 行あたりの文字数による改行の制限も解除してしまいます。NOPAGE 擬似命令では，各行の文字数制限はそのままで各ページの行数制限だけを解除することは出来ないので注意が必要です。

PAGE 擬似命令または NOPAGE 擬似命令のどちらかと，/PW または/NPW オプションのどちらかの擬似命令とオプションの両方を指定する場合には，オプションの指定が優先されます。

例

```
RASU8  FOO.ASM /PL100
```

この例では，プリントファイルの各ページの行数を 100 行に指定しています。

```
RASU8  FOO.ASM /NPL
```

この例では，プリントファイルの各ページの行数制限を抑制しています。

補足

/PL と/NPL を同時に指定する事は出来ません。

6.5.2.17 /T

構文

/T[*tab_width*]

説明

/T オプションを使用すると，プログラムに使用されているタブコードを適切な文字数分のスペースに置き換え，その後の文字を *tab_width* の倍数桁目に表示します。したがって，プリンタがタブコードを認識しない場合でも，/T オプションを使用して整形されたプリントファイルをリスト出力できます。

tab_width には，1 つのタブコードに対応するスペースの数を，1 から 15 までの整定数で指定します。*tab_width* を省略する場合，8 を指定することになります。

対応する擬似命令

/T オプションを指定する代わりに，プログラム中に TAB 擬似命令を記述してもかまいません。/T オプションと TAB 擬似命令の両方を指定する場合には，/T オプションの指定が優先されます。

/T オプションも TAB 擬似命令も指定しなかった場合，タブコードはそのままプリントファイルに出力されます。

例

```
RASU8  FOO.ASM /T4
```

この例では，プログラムに使用されているタブコードを最大 4 バイトのスペースに置き換え，プリントファイルを整形します。

6.5.2.18 /O，/NO

構文

/O[*object_file*]

/NO

説明

/O オプションを使用すると，オブジェクトファイルが作成されます。*object_file* には，オブジェクトファイル名を指定します。オペランドを省略する場合，およびファイル指定の一部を省略する場合のデフォルトについては，「6.2 ファイル指定のデフォルト」を参照してください。

/NO オプションを使用すると，オブジェクトファイルは作成されません。

/O オプションと/NO オプションを省略する場合は，プリントファイルは作成され，オブジェクトファイル名は，ソースファイル名の拡張子を“.OBJ”に変えたものになります。

対応する擬似命令

/O オプションを指定する代わりに，プログラム中に OBJ 擬似命令を記述してもかまいません。また，/NO オプションを指定する代わりに，プログラム中に NOOBJ 擬似命令を記述してもかまいません。オプションと擬似命令の両方を指定する場合には，オプションの指定が優先されます。

OBJ 擬似命令と NOOBJ 擬似命令については，「5.15.1 OBJ / NOOBJ 擬似命令」を参照してください。

例

```
RASU8  FOO.ASM /OOUTPUT.OBJ
```

この例では，OUTPUT.OBJ というオブジェクトファイルの作成を指定しています。

```
RASU8  FOO.ASM /NO
```

この例では，オブジェクトファイルを作成しないことを指定しています。

補足

/O と/NO を同時に指定する事は出来ません。

6.5.2.19 /SD

構文

/SD

説明

/SD オプションは，ソースファイルが CCU8 が作成したファイルである場合に指定します。このオプションを指定すると，RASU8 は CCU8 が作成したアセンブリソースファイルに埋め込まれた C デバッグ擬似命令を解析し，C ソースレベルデバッグ情報を含んだオブジェクトファイルを作成します。このオプションを指定することにより，C ソースレベルでのデバッグが可能になります。/SD オプションを指定しない場合，C ソースレベルでのデバッグはできません。

例

```
RASU8  CCFOO /SD
```

この例では，CCU8 の作成したソースファイル CCFOO.ASM をアセンブルし，C ソースレベルデバッグ情報を含んだオブジェクトファイルを作成します。

6.5.2.20 /D，/ND

構文

/D

/ND

説明

/D オプションを使用すると，オブジェクトファイルにアセンブリレベルデバッグ情報が出力されます。オブジェクトファイルにこのデバッグ情報が含まれていると，プログラムをシンボリックにデバッグできます。

/ND オプションを使用すると，オブジェクトファイルにアセンブリレベルデバッグ情報は，出力されません。

デフォルトでは，デバッグ情報はオブジェクトファイルに出力されません。

対応する擬似命令

/D オプションを指定する代わりに，プログラム中に DEBUG 擬似命令を記述してもかまいません。また，/ND オプションを指定する代わりに，プログラム中に NODEBUG 擬似命令を記述してもかまいません。オプションと擬似命令の両方を指定する場合には，オプションの指定が優先されます。

DEBUG 擬似命令と NODEBUG 擬似命令については，「5.15.4 DEBUG / NODEBUG 擬似命令」を参照してください。

例

```
RASU8  FOO.ASM /D
```

この例では，アセンブリレベルデバッグ情報をオブジェクトファイルに出力しています。

補足

/D と/ND を同時に指定する事は出来ません。

6.5.2.21 /E，/NE

構文

/E[*error_file*]

/NE

説明

/E オプションは，エラーメッセージの出力先を RASU8 に指示します。*error_file* にエラーファイル名を指定すると，そのファイルにエラーメッセージが出力されます。オペランドを省略する場合，およびエラーファイルの一部を省略する場合のデフォルトについては，「6.2 ファイル指定のデフォルト」を参照してください。

/NE オプションは，エラーメッセージを画面（標準出力）に表示することを RASU8 に指示します。

デフォルトでは，エラーメッセージは画面に表示されます。

/E オプションによってエラーメッセージの出力先を制御できるのは，アセンブルエラーメッセージとワーニングメッセージだけです。フェイタルエラーメッセージや内部処理エラーメッセージも合わせてファイルに出力するときは，DOS のリダイレクト機能を使用してください。

対応する擬似命令

/E オプションを指定する代わりに，プログラム中に ERR 擬似命令を記述してもかまいません。また，/NE オプションを指定する代わりに，プログラム中に NOERR 擬似命令を記述してもかまいません。オプションと擬似命令の両方を指定する場合には，オプションの指定が優先されます。

ERR 擬似命令と NOERR 擬似命令については，「5.15.3 ERR / NOERR 擬似命令」を参照してください。

例

```
RASU8  FOO.ASM /EERROR.LST
```

この例では，ERROR.LST というエラーファイルの作成を指定しています。

補足

/E と/NE を同時に指定する事は出来ません。

6.5.2.22 /X

構文

/X[*extrn_file*]

説明

/X オプションを使用すると，EXTRN 宣言ファイルが作成されます。*extrn_file* には，EXTRN 宣言ファイルを指定します。オペランドを省略する場合，およびファイル指定の一部を省略する場合のデフォルトについては，「6.2　ファイル指定のデフォルト」を参照してください。

デフォルトでは，EXTRN 宣言ファイルは作成されません。

EXTRN 宣言ファイルについては，「6.8 EXTRN 宣言ファイル」を参照してください。

例

```
RASU8  FOO.ASM /XEXTRN.INC
```

この例では，EXTRN.INC という EXTRN 宣言ファイルの作成を指定しています。

6.5.2.23 /BRAM，/BROM，/BNVRAM，/BNVRAMP

構文

/BRAM(*start_address*,*end_address*)

/BROM(*start_address*,*end_address*)

/BNVRAM(*start_address*,*end_address*)

/BNVRAMP(*start_address*,*end_address*)

説明

これらのオプションは，ユーザが RASU8 が管理できるアドレス空間上にメモリを追加した場合に，そのメモリの種類と領域を指定するためのオプションです。マイクロコントローラに搭載されているメモリの種類を再定義する事は出来ません。*start_address* と *end_address* には，それぞれ領域の開始アドレスと終了アドレスを指定します。

/BRAM オプションは，RAM の追加を行います。指定できるアドレスの範囲は 0:0000H から 255:0FFFFH の任意の領域が可能です。指定した範囲が物理セグメントアドレス#0 の領域を含んでいた場合，該当する領域はデータメモリ空間に配置されます。

/BROM は，ROM の追加を行います。指定できるアドレスの範囲は 0:0000H から 255:0FFFFH までの任意の領域が可能です。指定した範囲が物理セグメントアドレス#0 の領域を含んでいた場合，該当する領域はプログラムメモリ空間に配置されます。

/BNVRAM は，不揮発性メモリを追加します。指定できるアドレスの範囲は 0:0000H から 255:0FFFFH までの任意の領域が可能です。指定した範囲が物理セグメントアドレス#0 の領域を含んでいた場合，該当する領域はデータメモリ空間に配置されます。

/BNVRAMP はプログラムメモリ空間上の物理セグメントアドレス#0 に不揮発性メモリを追加します。指定できるアドレスの範囲は 0:0000H から 0:0FFFFH までの任意の領域が可能です。

6.5.2.24 /ZC

構文

/ZC

説明

/ZC オプションを使用すると，DB 擬似命令および DW 擬似命令でプログラムコードを配置する際，アクセス不可能な範囲に配置される場合または配置される可能性がある場合にワーニングを出力します。

このオプションは、CHKDBDW 擬似命令および NOCHKDBDW 擬似命令により部分的にワーニングチェックの有効および無効が設定できます。

6.6 終了コード

RASU8 は，アセンブル終了時に終了状態に応じた値を返します。この値を終了コードと呼びます。終了コードはバッチファイルなどを使用して検査できます。終了コードは，次のとおりです。

終了コード	説明
0	エラーはありません。
1	ワーニングが発生しました。
2	アセンブルエラーが発生しました。
3	フェイタルエラー，内部処理エラーまたは DCL エラーが発生して，強制終了しました。

6.7 プリントファイル

ここでは，RASU8 が作成するプリントファイルの書式と読み方を説明します。プリントファイルは，次のリストから構成されています。

1. アセンブリリスト

アセンブリリストは，プログラムとそれに対応するオブジェクトコードのリストです。

2. クロスリファレンスリスト

クロスリファレンスリストは，それぞれのユーザシンボルがプログラム中のどの行で定義され，参照されているかを示すリストです。

3. シンボルリスト

シンボルリストは，プログラムに使用しているユーザシンボルに関する情報のリストです。

4. 終了メッセージ

終了メッセージは，アセンブル中に検出したエラーとワーニングの総数，およびアドレス空間に関する情報を示すリストです。

次のオプションまたは擬似命令を使用して，プリントファイルの出力を制御できます。

	擬似命令	オプション
プリントファイル全体	PRN，NOPRN	/PR，/NPR
アセンブリリスト	LIST，NOLIST	/L，/NL
クロスリファレンスリスト	REF，NOREF	/R，/NR
シンボルリスト	SYM，NOSYM	/S，/NS

6.7.1 アセンブリリストの読み方

アセンブリリストの例を以下に示します。なお，説明の便宜上アセンブリリストの左側に番号が付いています。

アセンブリリストの番号にしたがって説明をしていきます。

```
(1) RASU8(ML610001)Relocatable Assembler, Ver.1.00.11   assemble list. page:   1
(2)  Source File: SAMPLE.asm
(3)  Object File: SAMPLE.obj
(4)  Date  : 2000/05/13 Thu.[16:53]
(5)  Title :
(6) ## Loc. Object                    Line   Source Statements

                                        1   ;*********************************
                                        2   ; sample program
                                        3   ;*********************************
                                        4           TYPE (M610001)
                                        5           MODEL SMALL, NEAR
                                        6           ROMWINDOW  0, 7FFFH
                                        7   ;--------------------------------
(7) -------- I N C L U D E ---------    8           INCLUDE(DEFINE.H)
                                        9   ;   include
                                       10           TAB     8
                                       11           PAGE    60, 80
                                       12           SYM
                                       13           REF
                                       14           ERR
                                       15           EXTRN   NUMBER: EXT_NUM
                                       16   ;   end of include
(8) ------  END OF INCLUDE  --------
(9)   --------------------------       17           CSEG    AT 2000H
                                       18   ;--------------------------------
 ┌ 00:2000                             19   LABEL:
 │   --------------------------       20           DSEG
(10)00:8000                            21           DS      100H
 │   --------------------------       22           BSEG
 │ 00:40000 (8000.0)                   23           DBIT    200H
 └ 00:41000 (8200.0)                   24           ORG     8200H.0
                                       25   ;--------------------------------
     --------------------------        26           CSEG
 ┌ 00:2000 12-90 00-30                 27           L       ER0, 3000H
(11)00:2004 FF 02                      28           MOV     R2, #0FFH
 └ 00:2006 00'03                       29           MOV     R3, #EXT_NUM
                                       30   ;--------------------------------
 ┌     = 0000FFFFH                     31   NUM_SYM         EQU     0FFFFH
(12)    = 00:FFFFH                     32   DATA_SYM        DATA    0FFFFH
 └     = 00:47FFFH (8FFFH.7)           33   BIT_SYM         BIT     08FFFH.7
                                       34   ;--------------------------------
     --------------------------        35           CSEG
 ┌ 00:2008 01 02 03 04 05 06 07 08     36           DB      1,2,3,4,5,6,7,8,
(13)00:2010 09 0A                     >>>   9,10
 │ 00:2012 01-00 02-00 03-00 04-00     37           DW      1,2,3,4,
 └ 00:201A 05-00 06-00 07-00          >>>   5,6,7
                                       38   ;--------------------------------
                                       39           L       R0, R0
 ┌  ** Error 00: bad operand
 │ 00:2020 11-90 FF-7F                 40           ST      R0, BIT_SYM
(14) ** Error 29: out of range
 │  ** Error 29: out of range
 └  ** Warning 12: usage type mismatch
                                       41
```

(1) 各ページの先頭には，対応するマイクロコントローラの機種，アセンブラの内部バージョン，

およびページ番号が表示されます。

(2) ソースファイル名です。

(3) オブジェクトファイル名です。

(4) アセンブルをした日時です。

(2)，(3)，および(4)は最初のページだけに表示されます。

(5) TITLE 擬似命令で指定したタイトルが表示されます。TITLE 擬似命令を省略した場合，タイトルの欄には何も表示されません。

(6) アセンブリリストのフィールドの見出しです。それぞれのフィールドには，次の意味があります。

フィールド	意味
##	物理セグメントアドレスを表示するフィールド。 物理セグメントアドレスが確定している場合は，そのアドレスを 16 進数で表示する。なお，物理セグメント属性が未確定の場合は“??”を表示する。 ##フィールドと Loc.フィールドの間はコロン（:）で区切られる。
Loc.	ロケーションカウンタを 16 進数で表示するフィールド。 アブソリュートセグメントの場合は，絶対アドレスを表し，リロケータブルセグメントの場合は，そのセグメントが割り付けられる先頭アドレスからの変位を表す。ビット型のセグメントの場合，ビットアドレスが表示された後，ドット演算子を用いた式で次のように表示する。 01000 (0200.0)
Object	オブジェクトコードを 16 進数で表示するフィールド。 オブジェクトコードは 1 バイト単位で表示され，アセンブル時に確定できない場合は，その右側に一重引用符（'）が付く。
Line	ソースファイル中にインクルードファイルの内容を挿入したときの行番号（10 進数）を表示するフィールド。したがって，インクルードファイルがある場合，この行番号の内容と，ソースファイルの対応する行番号の内容は一致しない。
Source Statements	ソースファイル，インクルードファイルの内容を表示するフィールド。

##フィールド，Loc.フィールド，および Object フィールドには，上記以外に特殊なメッセージやエラーメッセージを表示する場合もあります。

(7) INCLUDE 擬似命令を使用してインクルードファイルを挿入した場合に表示されます。そして，この次の行からインクルードファイルの内容が Source Statements フィールドに表示されます。

(8) インクルードファイルの表示がすべて終了した後に表示されます。

(9) CSEG，DSEG，RSEG 擬似命令などによりセグメントが切りかわるときに表示されます。

(10) ラベル，DS 擬似命令，DBIT 擬似命令，および ORG 擬似命令の記述の場合には，そのステートメントのロケーションが表示されます。

(11) オブジェクトコードの表示です。2 バイトのオブジェクトコードはハイフン（-）でつながれます。オブジェクトコードが未確定の場合，その未確定のコードの右端に一重引用符（'）が付けられます。

(12) ローカルシンボル定義擬似命令（EQU，SET，CODE，DATA，NVDATA，TABLE，BIT，NVBIT）の記述の場合には，シンボルに与えられる値（またはアドレス）が表示されます。

(13) DB 擬似命令，DW 擬似命令が記述されていた場合，1 ライン 8 バイト単位でコードが表示されます。1 ステートメントのコードが 8 バイトを超える場合，改行して表示されます。2 行目以降は Line フィールドに“>>>”マークが表示されます。

(14) ソースステートメントにエラーまたはワーニングがあった場合，そのラインの直後にエラー内容を表示します。

6.7.2 クロスリファレンスリストの読み方

クロスリファレンスリストは，プログラムの中に現れたシンボルの行番号の一覧を示すものです。

クロスリファレンスリストの例を以下に示します。

```
RASU8(ML610001)Relocatable Assembler, Ver.1.00.11   C-Ref list.    page:   2

symbol       lines ( #:definition line)
------------ ------------------------------------------------------------------
APSW ....... (SFR) 43
ASSP ....... (SFR) 42
BITSYM ..... 26#
CODESYM .... 25#
COMSYM ..... 27#
EXTSYM ..... 22#
MACROSYM ... 29#
NUMSYM ..... 23#
PUBSYM ..... 24# 30
SEG000 ..... 11# 18
SEG001 ..... 12# 18
SEG010 ..... 13# 19
SEG011 ..... 14# 19 32
SEG020 ..... 15# 20 35
SEG021 ..... 16# 20 38
UNDEF ...... 30
```

上記のように，クロスリファレンスリストは，2 つのフィールドから構成されています。それぞれのフィールドには，次の意味があります。

symbol フィールドには，ユーザシンボルの名前が表示されます。

lines フィールドには，シンボルが現れた行番号が表示されます。#が付く行番号は定義行であることを表します。SET 擬似命令で定義されたシンボルの場合，一番最後に定義した行番号に#が付けられます。DCL ファイルで定義された SFR アドレスシンボルの場合は，行番号の直前に(SFR)と表示されます。

6.7.3 シンボルリストの読み方

シンボルリストは，プログラムの中に現れたシンボルの詳細な情報を示すものです。シンボルリストは，シンボルインフォメーション，セグメントインフォメーションで構成されます。

6.7.3.1 シンボルインフォメーション

シンボルインフォメーションには，プログラムで定義されるすべてのシンボルと，1 回以上参照された SFR シンボルの詳細な情報を表示します。

以下にシンボルインフォメーションの例を示します。

```
----- symbol information -----

symbol       type usgtyp physeg     value    ID
------------ ---- ------ ------ ----------- ----
APSW ....... sfr  DATA   #00           4H      0
ASSP ....... sfr  DATA   #00           0H      0
BITSYM ..... loc  BIT    #00         200H.0    0
CODESYM .... loc  CODE   #00        1000H      0
COMSYM ..... com  DATA   ANY           0H      1
EXTSYM ..... ext  CODE   ANY           0H      1
MACROSYM ... mcr  Macro body
NUMSYM ..... loc  NUMBER ---    10000000H      0
PUBSYM ..... pub  DATA   #00        1000H      0
SEG000 ..... seg  CODE   ANY           0H      1
SEG001 ..... seg  CODE   ANY           0H      2
SEG010 ..... seg  CODE   #00           0H      3
SEG011 ..... seg  CODE   #00           0H      4
SEG021 ..... seg  BIT    ANY           0H.0    6
UNDEF ...... ***
```

それぞれのフィールドについて説明します。

symbol フィールドにはユーザシンボルの名前が表示されます。

type フィールドには，シンボルの種類を表す記号が次のとおり表示されます。

表示	説明
***	未定義シンボル
sfr	SFR シンボル
loc	ローカルシンボル
pub	パブリックシンボル
seg	セグメントシンボル
com	共有シンボル
ext	イクスターナルシンボル
mcr	マクロシンボル この場合は，mcr に続いてマクロ本体の文字列が表示される

usgtyp フィールドには，ユーセージタイプが次のとおり表示されます。

表示	説明
NUMBER	ユーセージタイプ NUMBER
NONE	ユーセージタイプ NONE
CODE	ユーセージタイプ CODE
DATA	ユーセージタイプ DATA
NVDATA	ユーセージタイプ NVDATA
TABLE	ユーセージタイプ TABLE
BIT	ユーセージタイプ BIT
NVBIT	ユーセージタイプ NVBIT
TBIT	ユーセージタイプ TBIT

physeg フィールドには物理セグメント属性を表す以下の記号が表示されます。

表示	説明
---	数値型
#*XX*	物理セグメントアドレス *XX*（16 進表示）
ANY	物理セグメントアドレスは未確定

value フィールドには，シンボルの値が 16 進数で表示されます。

ID フィールドには，セグメントシンボル，共有シンボル，イクスターナルシンボルの場合は，それぞれ定義された順番が表示されます。単純リロケータブルシンボルの場合は，そのシンボルが所属するセグメントの ID が表示されます。それ以外では，何も表示されません。

6.7.3.2 セグメントインフォメーション

セグメントインフォメーションには，プログラムで定義されるリロケータブルセグメントの詳細な情報を表示します。

以下に例を示します。

```
----- segment information -----

S-ID symbol       segtyp physeg  size  bound reltype
---- ------------ ------ ------ ------ ----- -------
   1 SEG000 ..... CODE   ANY        0H PAGE
   2 SEG001 ..... CODE   ANY        0H OCT
   3 SEG010 ..... CODE   #00        0H WORD
   4 SEG011 ..... CODE   #00       10H UNIT
   5 SEG021 ..... BIT    ANY      200H UNIT   NVRAM
```

それぞれのフィールドについて説明します。

S-ID フィールドにはセグメントシンボルの定義順番が表示されます。

symbol フィールドにはそのセグメントシンボルの名前が表示されます。

segtyp フィールドにはセグメントタイプが次のとおり表示されます。

表示	説明
CODE	ユーセージタイプ CODE
DATA	ユーセージタイプ DATA
NVDATA	ユーセージタイプ NVDATA
TABLE	ユーセージタイプ TABLE
BIT	ユーセージタイプ BIT
NVBIT	ユーセージタイプ NVBIT

physeg フィールドには物理セグメント属性が表示されます。

size フィールドにはセグメントサイズが 16 進数で表示されます。このサイズの単位はビット型の場合はビット単位，それ以外のセグメントではバイト単位です。

bound フィールドには境界値属性が表示されます。境界値属性は次のとおり表示されます。

表示	説明
UNIT	BIT, NVBIT セグメントでは 1 ビット境界 それ以外では 1 バイト境界
WORD	2 バイト境界
OCT	8 バイト境界
PAGE	256 バイト境界
整定数（16 進表示）	整定数の値が n のとき，BIT, NVBIT セグメントなら n ビット境界 それ以外なら n バイト境界

reltyp フィールドには指定した特殊領域属性が表示されます。特殊領域属性が指定されていなければ，何も表示されません。

6.7.4 終了メッセージの読み方

終了メッセージは，CPU コアの種類によって表示のされ方が異なります。

以下に例を示します。

例

```
Target       : ML610001 (nX-U8/100)  ← (1)
Memory Model : LARGE                 ← (2)
Data   Model : FAR                   ← (3)
ROM WINDOW   :    0H to 7FFFH        ← (4)
Internal RAM : 8000H to 8FFFH        ← (5)

Errors   : 0                         ← (6)
Warnings : 0  (/Wrpeast)             ← (7)
Lines    : 62                        ← (8)
```

(1) マイクロコントローラの機種名およびコア名が表示されます。

(2) メモリモデルの種類が表示されます。

(3) データモデルの種類が表示されます

(4) ROMWINDOW 擬似命令が指定されていればその領域の範囲表示されます。，NOROMWIN 擬似命令が指定されていれば，“None”が，どちらも指定されていない場合“(not specified)”が表示されます。

(5) 内部 RAM 領域の範囲が表示されます。

(6) エラーの総数が表示されます。

(7) ワーニングの総数が表示されます。

(8) 処理した行数が表示されます。

6.8 EXTRN 宣言ファイル

ここでは，RASU8 が作成する EXTRN 宣言ファイルの使い方と作り方を説明します。

6.8.1 EXTRN 宣言ファイルとは

EXTRN 宣言ファイルとはプログラムのパブリック宣言に対応するイクスターナル宣言を内容とするファイルであり，/X オプションを指定したときに作成されます。

モジュール分割を行う場合，シンボルを定義するモジュールではパブリック宣言，参照するモジュールではイクスターナル宣言を行う必要があります。通常，この宣言はプログラマが行うのですが，シンボルの宣言の管理はシンボルが多くなった場合や，モジュール数が多くなった場合にわずらわしいものとなるでしょう。

EXTRN 宣言ファイル作成機能は，このような問題を解決するための 1 つの手段として用意されたものです。

6.8.2 EXTRN 宣言ファイルの作り方および使い方

簡単な例として，モジュール F001.ASM の中でサブルーチン SUB00 と SUB01 を，DATA アドレス空間のシンボルとして BUF00 と BUF01 を定義してモジュール F002.ASM, F003.ASM でそれらを参照する場合を考えてみましょう。

F001.ASM

```
        TYPE (M610001)
        PUBLIC  SUB00 SUB01 BUF00 BUF01

        CSEG    AT 0:1000H
SUB00:
        ; sub routine SUB00
        RT

R_CODE  SEGMENT CODE
        RSEG    R_CODE
SUB01:
        ; sub routine SUB01
        RT

        DSEG    AT 0:8000H
BUF00:  DS      10H

R_DATA  SEGMENT DATA
        RSEG    R_DATA
BUF01:  DS      100H
```

このファイルを次のようにアセンブルします。

```
RASU8 F001 /X
```

すると，次のように EXTRN 宣言ファイル F001.EXT が自動的に作成されます。

F001.EXT

```
;; External symbol declaration file.

     EXTRN     CODE NEAR : SUB00
     EXTRN     CODE NEAR : SUB01
     EXTRN     DATA NEAR : BUF00
     EXTRN     DATA NEAR : BUF01
;; End of listing.
```

ここで，それらを参照する F002.ASM および F003.ASM ではプログラムの最初で次のように指定します。

```
     INCLUDE (F001.EXT)
```

この 1 行をプログラムに記述することにより，F001.ASM で定義したパブリックシンボルをすべて参照することができます。

また，F001.ASM をもう一度，次のようにアセンブルします。

```
RASU8 F001 /X /DF /ML
```

すると，今度は次のように EXTRN 宣言ファイル F001.EXT が自動的に作成されます。

```
;; External symbol declaration file.

     EXTRN     CODE FAR  : SUB00
     EXTRN     CODE FAR  : SUB01
     EXTRN     DATA NEAR : BUF00
     EXTRN     DATA FAR  : BUF01
;; End of listing.
```

このように，EXTRN 宣言ファイルを使用することによって，プログラマはイクスターナル宣言の管理から解放されるばかりか，プログラムが読みやすくなり，保守性も向上します。また，例で示した F001.EXT の SUB00，SUB01 や BUF01 からも分かるように，メモリモデルやデータモデルに応じて各シンボルの物理セグメント属性も付加されますので，確実なアドレッシングの最適化を行えます。

6.9 エラーメッセージ

RASU8 は，アセンブル処理に関するエラーを報告します。エラーの報告には，次の種類があります。

1. 画面またはエラーファイルにエラーメッセージを出力する。
2. プリントファイルにエラーの番号を出力する。

RASU8 のエラーには，次の種類があります。

1. フェイタルエラー
2. アセンブルエラー
3. ワーニング
4. 内部処理エラー

フェイタルエラーは，RASU8 がアセンブルを続行できない致命的なエラーです。フェイタルエラーが発生すると，RASU8 はアセンブル処理を停止します。

アセンブルエラーは，ソースファイルの解析上発生したエラーです。アセンブルエラーが発生してもアセンブル処理を続行し，プリントファイルとオブジェクトファイルを作成します。

ワーニングは，プログラムに問題があるかもしれないということを示します。ワーニングが発生してもアセンブル処理を続行し，プリントファイルとオブジェクトファイルを作成します。

内部処理エラーは，RASU8 の内部処理の不具合が検出された場合に発生するエラーです。内部処理エラーが発生すると，RASU8 はアセンブル処理を停止します。

通常これらのエラーメッセージは，画面に表示されます。エラーメッセージをファイルに出力させたいときには，DOS のリダイレクト機能を使用してください。また，アセンブルエラーとワーニングのメッセージだけをファイルに出力させたいときには，/E オプションまたは ERR 擬似命令を使用してください。

6.9.1 エラーメッセージの形式

画面またはエラーファイルに出力されるエラーメッセージの形式は，次のとおりです。

構文

filename(*line1*) : *line2* : *type number* : *message*

filename には，エラーが発生したファイルの名前が表示されます。*line1* には，エラーが発生した箇所を示すソースファイル上の行番号が表示されます。*line2* には，エラーが発生した箇所を，プリントファイルの Line フィールドの値で表示します。

type には，エラーの種類が次のとおり表示されます。

type	エラーの種類
Fatal Error	フェイタルエラーであることを示す。
Error	アセンブルエラーであることを示す。
Warning	ワーニングであることを示す。

number にはエラー番号が，*message* にはエラーメッセージが表示されます。エラー番号とエラーメッセージの一覧は，「6.9.2 エラーメッセージ一覧」に説明されています。

6.9.2 エラーメッセージ一覧

RASU8 が表示するエラーメッセージの一覧を以下に示します。エラーメッセージの左には，エラーメッセージの番号が記載されています。エラーメッセージの下には，その内容が説明されています。

6.9.2.1 フェイタルエラーメッセージ

F00　insufficient memory

処理を継続するためのメモリが足りません。このエラーの原因は，Windows の仮想メモリの領域不足が考えられます。ハードディスクの空き容量を増やしてみる，仮想メモリの上限を増やす，他のアプリケーションが起動している場合は終了してみる，等の操作を行って仮想メモリの空き領域を増やしてみてください。また/R オプション（もしくは REF 擬似命令）を指定している場合は，その指定を外してみてください。それでもなお，このエラーが発生する場合は，プログラムを分割するかシンボル数を減らす対策を行ってください。

F01　file not found : *file_name*

file_name で示されるソースファイル，インクルードファイル，または DCL ファイルが見つかりません。

F02 cannot open file : *file_name*

file_name で示されるファイルが作成できません。
file_name はオブジェクトファイル，プリントファイルまたはエラーファイルです。名前の指定に無効な文字を使っていないか，または存在しないディレクトリを指定していないかを確認してください。

F03 cannot close file : *file_name*

ファイルをクローズできません。
もっとも考えられる原因は，ディスク容量の不足です。

F04 error(s) found in DCL file

DCL ファイルになんらかの文法エラーが 1 つ以上存在しました。
この場合，アセンブル結果が保証できないためアセンブラはこのエラーメッセージを出力して処理を終了します。弊社が提供するオリジナル DCL ファイルを使う限りは，このエラーが発生することはありません。

F05 file seek error

ファイルシークができません。

F06 too many INCLUDE nesting levels

インクルードファイルのネスティングレベルが 8 を越えています。

F07 line number overflow

1 つのソースプログラムの行数（インクルードファイルも含めた総数）が 9,999,999 を越えています。

F08 I/O error writing file

オブジェクトファイルへの書き込みができません。

F09 TYPE directive missing

ソースプログラム中に TYPE (機種名) の指定がありません。
または，TYPE 擬似命令より前に別の命令を記述しています。

F10 unclosed block comment

ブロックコメント /* ... */ が閉じていません。

F11　illegal reading binary file

ABL ファイルの内容が正しくありません。
このエラーが発生した場合，上記のエラーメッセージに続けて次のメッセージも表示されます。

ABL file : *message*

message にはそのエラーの内容が表示されます。このエラーが発生した場合，まずは次のことを確認してください。

最初のアセンブルでエラー（ワーニングは除く）が発生していませんか。
最初のアセンブル後にソースファイルを編集していませんか。
最初のアセンブルと再アセンブルでメモリモデルの指定，データモデルの指定，分岐最適化オプションの有無，ROMWINDOW 領域の指定，/B オプションの指定，シンボルの大文字小文字の区別の指定，インクルードパス指定は完全に一致していますか。
リンク時にアドレッシング関連以外の致命的なエラーが発生していませんか。
リンクするときに/A オプションを指定していますか。
最新版のリンカを使用して ABL ファイルを作成していますか。

これらのことを確認し，必要があればソースファイルを再度アセンブル，リンクして ABL ファイルを作成した後，再度アブソリュートプリントファイルを作成してください。それでもエラーが発生する場合は弊社までご連絡ください。
以下に，メッセージの種類とその意味を示します。

以下の“ABL file :”で始まるメッセージは，ABL ファイルに問題があるときに，前ページのフェイタルエラーメッセージ「F11　illegal reading binary file」に続いて表示されるものです。

ABL file : module information is not found

ABL ファイルには，現在アセンブルしているプログラムの情報は含まれていません。

ABL file : CORE ID mismatch

ABL ファイルで定義されるモジュール情報と，現在アセンブルしているプログラムの対象の CPU コアが一致していません。

ABL file : Target Machine mismatch

ABL ファイルで定義されるモジュール情報と，現在アセンブルしているプログラムの対象のマイクロコントローラの名前が一致していません。

ABL file : Memory Model mismatch

ABL ファイルで定義されるモジュール情報と，現在アセンブルしているプログラムのメモリモデルの種類が一致していません。

ABL file : DATA Model mismatch

ABL ファイルで定義されるモジュール情報と，現在アセンブルしているプログラムのデータモデルの種類が一致していません。

ABL file : branch optimization mismatch

ABL ファイルで定義されるモジュール情報と，現在アセンブルしているプログラムの分岐最適化オプションが一致していません。

ABL file : symbol is not entry

ABL ファイルに含まれるシンボルが，現在アセンブルしているプログラムで定義されていません。

ABL file : illegal physical segment attribute

セグメントシンボルの属性情報が異常な値を保持しています。

ABL file : illegal segment type

セグメントシンボルのセグメントタイプが異常な値を保持しています。

ABL file : illegal usage type

シンボルのユーセージタイプが異常な値を保持しています。

ABL file : symbol type mismatch

シンボルの種類（セグメントシンボル，共有シンボルなど）が，ABL ファイルと現在アセンブルしているプログラムで一致していません。

ABL file : symbol usage type mismatch

シンボルのユーセージタイプ（CODE ,DATA, BIT など）が，ABL ファイルと現在アセンブルしているプログラムで一致していません。

ABL file : local symbol value mismatch : *symbol*

ラベルなどのローカルシンボルが，正しくない値を保持しています。

ABL file : file format is illegal

ABL ファイルの構造に欠陥があります。

ABL file : absolute machine code mismatch.(line *xxxx*)

再アセンブルで生成したマシンコードと，RLU8 がフィックスアップしたマシンコードが一致していません。

ABL file : location of absolute machine code mismatch.(line *xxxx*)

再アセンブルで生成したマシンコードと，RLU8 がフィックスアップしたマシンコードのアドレスが一致していません。

F12 checksum error reading file

ABL ファイルのレコードのチェックサムが正しくありません。

F13　I/O error reading file

ABL ファイルを読み込むことができません。

F14　old DCL file

古い RASU8 用の DCL ファイルを使用しています。

F16　source code filename not specified

RASU8 起動時にソースファイルの指定がありません。

6.9.2.2 アセンブルエラーメッセージ

E00　bad operand

オペランドの記述が間違っています。
マイクロコントローラの命令の場合，アドレッシング指定が間違っているか，またはオペランドの数が多いか少ないかが考えられます。
擬似命令の場合，各擬似命令のフォーマットと記述が一致していないことが考えられます。

E01　bad syntax

命令を認識する以前の基本的な構文ミスです。

E03　physical segment address out of range

物理セグメントアドレスが，実際に使用できるセグメント数を越えています。対象のマイクロコントローラの物理セグメント数の範囲内でありながらこのエラーが発生する場合は，メモリモデルが SMALL になっている可能性があります。

E04　bad character : *c* (*XX*)

文字 *c*（ASCII コード *XX*）はプログラムで使用することはできません。

E05　illegal integer constant

整定数またはアドレス定数の記述が間違っています。

E06　illegal escape sequence

文字定数または文字列定数中のエスケープシーケンスの記述が間違っています。

E07　unexpected EOL
E08　unexpected EOF

文字定数（'*c*'）または文字列定数（"..."）が閉じていません。

E09　illegal string constant

文字列定数に無効な記述があります。

E10　string constant too long

文字列定数の文字数が 256 文字を越えています。

E11　illegal option : *option*

option はオプションとして認められません。この指定は無視されます。

E12　constant required

命令のオペランドかオプションに整定数の指定が必要です。

E13　declaration duplicated

同じ擬似命令またはオプションが 2 度以上指定されています。

E14　location out of range

ロケーションが規定範囲を越えています。
このエラーは，セグメント開始指定（CSEG 擬似命令など）の AT アドレス，または ORG 擬似命令の開始アドレスがセグメント制限の上限または下限を越えている場合や，命令や DS，DBIT， GJMP，GCAL，DB，および DW の各擬似命令により更新されるロケーションがセグメント上限を越える場合に起こります。

E17　AT address must be NUMBER

アブソリュートセグメント開始時に AT と # を指定した場合，AT 指定はオフセットアドレスと解釈されます。この場合，AT 指定にアドレス式を指定することはできません。
例えば，次のような場合にこのエラーが発生します。

```
CSEG  AT  2:3000H #4
```

E18　segment / usage type mismatch

命令が要求するセグメントタイプまたはユーセージタイプと，指定したタイプが一致していません。このエラーは次のような場合に発生します。
セグメント開始指定（ORG 擬似命令，CSEG 擬似命令など）の開始アドレスのユーセージタイプが，カレントセグメントのセグメントタイプと一致していない。
シンボル定義擬似命令（CODE 擬似命令など）のオペランドのユーセージタイプが，命令のタイプと一致しない。
DS 擬似命令をビット系セグメント中に記述している。
DBIT 擬似命令をバイト系セグメント中に記述している。
DB 擬似命令，DW 擬似命令を CODE，NVDATA，TABLE 以外のセグメントに記述している。

E19　undefined symbol : *symbol*

symbol は定義されていません。

E20　segment symbol required

SIZE，OVL_ADDRESS，OVL_SEG，OVL_OFFSET の各演算子の右辺，RSEG 擬似命令のオペランドにはセグメントシンボルが必要です。

E21　forward reference not allowed

前方参照を行っています。
多くの擬似命令は，オペランドに前方参照を許していません。
また RSEG 擬似命令に指定するセグメント名は，必ずそれ以前に定義しておかなければなりません。

E22 stack segment not allowed

スタックセグメント$STACK を使用することはできません。

E23 symbol redefinition : *symbol*

symbol はすでに定義されています。

E25 segment ID mismatch

リロケータブルセグメントで ORG 擬似命令のオペランドにリロケータブルアドレスを指定する場合，その式はカレントセグメントのアドレスを表わしていなければなりません。

E26 address not allowed

カレントセグメントが ANY 型リロケータブルセグメントの場合，ORG 擬似命令のアドレスは数値型（NUMBER）でなければなりません。

E27 physical segment address mismatch

物理セグメントアドレスが一致していません。
このエラーは，ROMWINDOW 擬似命令のオペランドに物理セグメント#1 以上のアドレスを指定した場合や，物理セグメントの確定したセグメント内で ORG 擬似命令を指定して，そのアドレスの物理セグメントアドレスがカレントセグメントと一致しない場合に発生します。

E28 local symbol required : *symbol*

パブリック宣言する *symbol* は，ローカルシンボルとして定義されていなければなりません。

E29 out of range : message

オペランドの値が規定の範囲を越えています。
message には，具体的な領域の名前などが表示されます。

E30 illegal boundary
E31 illegal relocation type

SEGMENT 擬似命令または COMM 擬似命令の境界値指定，または特殊領域属性の指定が間違っています。

E33 entry overflow

セグメントシンボル，共有シンボル，またはイクスターナルシンボルの数が 65535 を越えています。または/BRAM，/BROM，/BNVRAM，/BNVRAMP により追加された領域が多すぎます。

E34 string constant required

DATE 擬似命令または TITLE 擬似命令のオペランドに文字列定数が必要です。または C デバッグ擬似命令のオペランドの書式が間違っています。

E35 absolute expression required

オペランドは定数式でなければなりません。
このエラーは多くの擬似命令のオペランドや SWI 命令の割り込み番号，シフト系命令のシフト幅が，定数式でない場合に発生します。

E36 simple relocatable expression required

シンボル定義擬似命令（EQU 擬似命令など）または ORG 擬似命令のオペランドは，定数式または単純リロケータブル式でなければなりません。

E37 expression is unresolved

未解決な演算に対して，さらに演算を行っています。
もしくはシンボル定義擬似命令（EQU 擬似命令など）または ORG 擬似命令のオペランドに，未解決演算を含む式を指定しています。

E38 illegal expression format

式の基本的な構文ミスです。
例えば，カッコのバランスが合っていない場合などがこれに該当します。

E39 invalid relocatable expression

リロケータブルシンボルに対して許されていない演算を行っています。

E40 division by zero

0 による除算またはモジュロ演算を行っています。

E41 illegal bit offset

ドット演算子の右辺のビットオフセットが定数ではありません。
またはビットアドレッシングのビットオフセットに定数ではない値かまたは 8 以上の値を指定しています。

E42 right expression of SEG operator must be address

SEG 演算子のオペランドには，アドレスを指定しなければなりません。

E44 illegal core name

DCL ファイルの#CORE 文の CPU コア名が間違っています。

E46 mnemonic required

DCL ファイルの#INSTRUCTION 文以降には命令ニーモニックが必要です。

E48　#ENDCASE without #CASE

DCL ファイル中で#ENDCASE 文とつりあう#CASE 文がありません。

E51　CODE segment only

マイクロコントローラの命令，GJMP 擬似命令，GCAL 擬似命令，CLINE 擬似命令，CLINEA 擬似命令は CODE セグメントにしか記述できません。

E52　GJMP/GBcond operand must be symbol

GJMP 擬似命令または GB*cond* 擬似命令のオペランドは，シンボルしか指定できません。

E54　out of relative jump range

カレントロケーションと分岐先アドレスの物理セグメントアドレスが異なるか，オフセットアドレスの差が－128～＋127 ワードの範囲にありません。

E55　LABEL or NAME format error

命令とシンボル，またはラベルの構文関係が間違っています。
例えば，次のような場合にこのエラーが発生します。

```
LABEL:   EQU   100H
NAMES    DS    100H
```

E56　invalid CPU instruction

DCL ファイルの#INSTRUCTION 文で定義されていない命令を，プログラム中で使用しています。

E57　invalid initialization directive

アセンブラ初期設定擬似命令（ROMWINDOW，NOROMWIN，MODEL）の記述位置が正しくありません。このエラーはこれらの擬似命令より前には指定できない命令を記述した場合に発生します。例えば，次のような場合にこのエラーが発生します。

```
EXTRN   NUMBER : MAXADDRESS  ;ROMWINDOW や MODEL の前には指定できない。
NOROMWIN

MODEL   LARGE
```

E58　illegal SFR word/byte attribute

DCL ファイルの SFR アクセス属性定義文の中のワード／バイトアクセス属性フィールドのフォーマットが間違っています。

E59　illegal SFR bit attribute

DCL ファイルの SFR アクセス属性定義文の中のビットアクセス属性フィールドのフォーマットが間違っています。

E60 out of SFR address range

DCL ファイルの SFR アクセス属性定義文の中の SFR アドレスが，SFR キーワードで定義された SFR 領域の範囲に入っていません。

E61 misplaced ENDIF directive

ENDIF 擬似命令に対応する条件アセンブル開始擬似命令(IF, IFDEF, IFNDEF)がありません。

E62 misplaced ELSE directive

ELSE 擬似命令に対応する条件アセンブル開始擬似命令(IF, IFDEF, IFNDEF)がありません。

E63 unexpected end of file in conditional directive

条件アセンブル開始擬似命令(IF, IFDEF, IFNDEF)に対応する ENDIF 擬似命令がありません。このエラーは，常にプログラムの終わりの行に対して発生します。

E64 too many conditional directive nesting levels

条件アセンブル命令のネスティングのレベルが 15 を越えています。

E65 too many macro nesting levels

マクロのネスティングのレベルが 8 を越えています。

E67 symbol for CDB directives not defined

CSLOCAL 擬似命令，CSGLOBAL 擬似命令，CLABEL 擬似命令が検索しているローカルシンボルがありません。

E71 label or '$' is not allowed

分岐最適化オプション指定時には CODE セグメント内で定義されたラベルやカレントロケーションシンボルは前方参照シンボルと同じ扱いになります。したがって，CSEG 擬似命令や EQU 擬似命令のオペランドなど，前方参照を許さないオペランドでは記述できません。

E72 invalid NEAR/FAR

NEAR アドレス指定子の記述が間違っています。例えば物理セグメント#1 以上のアドレス式に対し NEAR アドレス指定子を付加した場合などです。

E73 usage type NUMBER expected

数値型の式が要求されています。次のような記述をした場合にこのエラーが発生します。

```
ADDR  EQU     1:1234H
      MOV     QR0, ADDR:[EA+]
```

E74 cannot write to ROM

ROM 領域に対して書き込み命令を実行しようとしています。

E75 invalid fn_id

C デバッグ情報擬似命令において fn_id オペランドの値が正しくありません。

E76 invalid block_id

C デバッグ情報擬似命令において，block_id オペランドの値が正しくありません。

E77 cfunction cannot nest

CFUNCTION 擬似命令のネストをしようとしています。例えば間違って CFUNCTIONEND 擬似命令を削除した場合や fn_id が異なる場合にこのエラーが発生します。

E78 invalid position

C デバッグ情報擬似命令の記述位置が間違っています。この擬似命令以前に，対応すべき C デバッグ擬似命令が存在しない場合に発生します。

E79 overlay location out of range

オーバーレイ機能を使用した CODE セグメントの実配置アドレスが ROM の存在しないアドレスを指しています。

E80 illegal range

指定した領域が間違っています。これは/BROM，/BRAM，/BNVRAM，/BNVRAMP オプションで指定できない範囲を指定した場合，開始アドレスが終了アドレスよりも大きいアドレスであった場合，既存の領域と重なる領域を指定した場合にこのエラーが発生します。

E81 missing physical segment address

物理セグメントアドレスの記述がありません。このエラーは，CSEG 擬似命令で OVL オペランド指定してありながら実行時アドレスの物理セグメントアドレスを指定しなかった場合に発生します。

E82 missing member directives for previous CxxxTAG directive

CSTRUCTTAG 擬似命令に対し CSTRUCTMEM 擬似命令が足りない場合，CENUMTAG 擬似命令に対し CENUMMEM 擬似命令が足りない場合にこのエラーが発生します。

E84 unclosed CFUNCTION directive exists

CFUNCTION 擬似命令と CFUNCTIONEND 擬似命令の対応が取れていません。CFUNCTION 擬似命令に対応する CFUNCTIONEND 擬似命令が無いままソースが終了した場合に発生します。

E85 segment address mismatch

CFUNCTION 擬似命令から，対応する CFUNCTIONEND 擬似命令までの間にセグメントが変化しています。RASU8 は正しい C デバッグ情報を出力する事が出来ません。

E89 DSR access prohibited

NOFAR 擬似命令によって DSR レジスタの使用が禁止されたソース内において，DSR レジスタの内容を変更しようとしています。

E90 duplicated CRET directive between CFUNCTION and CFUNCTIONEND

CFUNCTION 擬似命令と CFUNCTIONEND 擬似命令との間で，CRET 擬似命令が 2 回以上記述されています。RASU8 は正しい C デバッグ情報を出力する事が出来ません。

6.9.2.3 ワーニングメッセージ

ワーニングは 6 つの種類に分類されています。ワーニングチェックは/NW オプションで禁止させることができます。また，/W の後にワーニングの種類を表す文字を指定することによって，特定の種類のチェックだけを行うようにすることもできます。/W の後に指定できる文字とその意味を以下に示します。

文字	チェックの内容
R	リロケータブルセグメントの定義に関するチェック
P	擬似命令の記述に関すチェック
E	式の評価に関するチェック
A	アドレッシングチェックに関するチェック
S	SFR アクセス属性に関するチェック
T	ROMWINDOW 状態に関するチェック

例えば R，E，T のワーニングチェックを行う場合は，起動オプションとして/WRET を指定します。

次にワーニングメッセージとその意味を示します。ワーニング番号の後に示す文字がワーニングの種類を表します。

W01(R) stack size must be even

スタックセグメントのサイズは偶数でなければなりません。RLU8 は指定した値に 1 加算したサイズでスタック領域を確保します。

W02(P) duplicate option or directive

既に指定した擬似命令やオプションを再び指定しています。この指定は無視されます。

W05(E) expression of type address required

アドレス式が必要です。
この警告は，OFFSET 演算子の右辺に数値式を指定した場合に発生します。

W06(E) expression of type NUMBER required

数値式が必要です。
この警告は，BYTE1，BYTE2，BYTE3，BYTE4，WORD1，および WORD2 演算子の右辺や，数値型のみを許すアドレッシングに，アドレス式を指定した場合に発生します。以下に例を示します。

```
ADRSYM  EQU     0:2345H
IS78H   EQU     BYTE1 01:5678H   ; ワーニング 06
        LEA     2:3000H     ; ワーニング 06
        L       R0, R5:ADRSYM    ; ワーニング 06
        ADD     SP, #0:12H       ; ワーニング 06
        SWI     #0:12H           ; ワーニング 06
```

1 行目以外はすべてこのワーニングが発生します。

W08(E) segment address mismatch

アドレス同士の演算において，左辺と右辺のセグメントアドレスが一致していません。

W09(E) address attribute not inherited

式は数値として扱われます。アドレスとしての属性は失われます。

W10(E) cannot check physical segment address

物理セグメントアドレスが一致しているかどうか保証できません。

W11(E) right expression of operator must be NUMBER

シフト演算または右辺は数値型でなければなりません。

W12(A) usage type mismatch

命令が要求するユーセージタイプと，指定したタイプが一致していません。

W13(E) left expression of bit operator must be byte address

ドット演算の左辺は，バイト型の式でなければなりません。

W16(E) BPOS operator should be used only on bit address

BPOS 演算子の右辺は，ビット型でなければなりません。

W25(S) illegal access to SFR

SFR 領域に対するアクセスが無効です。
このワーニングは，書き込み禁止の SFR に対して書きこんだ場合や，ワードアクセス禁止の SFR にワードアクセスした場合などに発生します。

W26(S) cannot access to high byte in SFR word

ワードアクセスだけ可能な SFR の上位バイトにワードアクセスしています。

W28(A) cannot access to high byte

RAM の奇数アドレスに対してワードアクセスしています。

W29(A) cannot write to ROM window

ROM WINDOW 領域に対して書き込みを行なっています。

W31(E) reference before first definition

SET 擬似命令で定義されたシンボルを最初の定義より前で参照した場合，シンボルには最後に定義された値がセットされます。

W35(A) branch address must be even

相対分岐命令の分岐先アドレスのオフセットは偶数でなくてはなりません。

W36(A) physical segment address not determined

物理セグメントアドレスが決められません。
アセンブラは NEAR アドレッシングか FAR アドレッシングか決める事が出来ないので保留とし，RLU8 でアドレスチェックを行います。

W37(T) ROMWINDOW/NOROMWIN is not specified

ソースプログラム中で ROMWINDOW 擬似命令も NOROMWINDOW 擬似命令も指定されていません。RASU8 は物理セグメントアドレス #0 のデータメモリ空間上のメモリの種類を特定できません。

W38(A) current location aligned

直前の命令によって CODE セグメントのカレントロケーションが奇数になりました。
アセンブラはオフセットアドレスに 1 を加算して偶数アドレスになるよう補正します。

W39(A) address out of range

命令のオペランドによって指定されたアドレスは対象となるメモリが存在しません。
CODE，DATA 等のシンボル定義擬似命令で対象メモリの存在しないアドレスを指定した場合や，/BRAM，/BROM，/BNVRAM，/BNVRAMP オプションで領域を確保し忘れた場合などによく発生します。

W40(A) physical segment address out of range

この警告は，SEGMENT 擬似命令で物理セグメントアドレスを指定した時，セグメントタイプに対応するメモリがその物理セグメントに存在しない場合に発生します。
/BRAM，/BROM，/BNVRAM，/BNVRAMP オプションで領域を確保し忘れた場合などによく発生します。

W42(A) cannot load this data by L instruction

データとしてアクセスできない、またはアクセスできるかどうか判別できないメモリ上に定数コードを記述しました。定数コードの配置アドレスを変更するか、NOCHKDBDW 擬似命令でチェックを抑止して下さい。

W48(A) DSR prefix generated

データメモリの物理セグメントが#0 だけの機種において、DSR プリフィックスコードが使われています。データメモリの物理セグメントが#0 だけの場合、DSR プリフィックスコードは不要です。

W49(R) ABL file format is old. Please rebuild ABL file by the latest linker

古いバージョンのリンカで作成した ABL ファイルが指定されています。最新のリンカを使用して再ビルドして ABL ファイルを作成してください。

6.9.2.4 内部処理エラーメッセージ

**** RASU8 Internal Error : Process [*function*] ****

RASU8 の内部処理の不具合が検出された場合に発生するエラーです。*function* は内部処理位置を表わす文字列です。通常このエラーが発生することはありませんが，もしこのエラーが発生した場合は弊社までご連絡くださるようお願い致します。

7 RLU8

7.1 概要

リンカ RLU8 は，リロケータブルアセンブラ RASU8 が作成した複数のオブジェクトファイルを結合して 1 つのアブソリュートオブジェクトファイルを作成するためのユーティリティです。

RLU8 には，オブジェクトファイルの他に，LIBU8 によって作成されたライブラリファイルを入力として指定することができます。ライブラリファイルが指定されると，RLU8 はライブラリファイル中のオブジェクトモジュールを取り出してリンクします。ライブラリファイル中のオブジェクトモジュールをリンクする方法には，次の 3 つがあります。

(1) すべてのオブジェクトモジュールを取り出してリンクする方法

(2) 指定したオブジェクトモジュールだけを取り出してリンクする方法

(3) 未解決な外部参照を解決するオブジェクトモジュールだけを取り出してリンクする方法

本文書では，オブジェクトモジュールを単にモジュールとも呼びます。

RLU8 によって作成されるアブソリュートオブジェクトファイルには，リロケータブルな部分がすべて解決されたオブジェクトコードが含まれています。オプションを使って，このファイルにデバッグ情報を出力することもできます。

RLU8 はマップファイルも作成します。これは，セグメントの割り付け状態とパブリックシンボルのリストを内容とするもので，プログラムをデバッグするときに，セグメントの開始アドレスを知る場合などに使用します。

本文書では，位置関係を表すために，アドレス 0 方向を下位メモリ，アドレス 0FFFFH 方向を上位メモリと表現します。

7.2 RLU8 の操作方法

7.2.1 コマンドラインの書式

RLU8 のコマンドラインの書式は次のとおりです。

RLU8 *object_files* [, [*absolute_file*] [, [*map_file*][, [*libraries*]]]] [;]

コンマ（,）で区切られている各フィールドは，次のように使われます。

object_files フィールドは，リンクするオブジェクトファイルおよびライブラリファイルの名前を指定するのに使用します。

absolute_file フィールドは，デフォルトの出力ファイル名を他の名前に変更するのに使用します。

map_file フィールドは，デフォルトのマップファイル名を他の名前に変更するのに使用します。

libraries フィールドは，未解決な外部参照を解決するために使用されるライブラリファイルを指定するのに使用します。

object_files フィールドへの入力以外は，省略することができます。

フィールドへの入力を省略する場合は，そのフィールドの直後のコンマだけを入力します。コンマの代わりにリターンキーだけを入力すると，RLU8 はプロンプトを表示し，そのフィールドへの入力を促します。

RLU8 は，デフォルトの処理を変更させるためにいくつかのオプションを用意しています。オプションは，どのフィールドにも指定できます。

コマンドラインの最後にあるセミコロン（;）は，コマンドの終了を意味します。セミコロンを指定すると， RLU8 は，残りのフィールドへの入力を求めるプロンプトを表示しません。この場合， RLU8 は，省略されたフィールドに対して，デフォルト値を使用します。セミコロンを使用することによって，不要なプロンプトを出力させないようにすることができます。

RLU8 は，入力として与えられたファイル名にパスが指定されていなければ，そのファイルのデフォルトパスとしてカレントドライブのカレントディレクトリを仮定します。したがって，読み込みたいファイルがこのデフォルトパス以外にあったり，デフォルトパス以外にファイルを作成したい場合，ファイル指定にパス名を与えなければなりません。

拡張子を持たないファイル名を指定するときは，その名前の直後にピリオド（.）を付けます。ピリオドを付けないと，そのフィールドが持つデフォルトの拡張子が付けられます。

object_files フィールド以外の 3 つのフィールドでは，パス名だけを指定することができます。この場合，パス名の最後に円記号（¥）を付けなければなりません。たとえば，“¥USR¥APDIR¥”のように指定します。そうしないと，RLU8 は，パス名の最後のディレクトリをベース名と見なし，そのフィールドのデフォルトの拡張子を付けてしまいます。

ファイル名はパスを含めて最大 255 文字のロングファイル名を許しています。ただし，ファイル名に全角文字および空白文字を使用することはできませんのでご注意ください。また，拡張

子は最後のドット（.）以降を拡張子とみなします。

次に，各フィールドの使い方を説明します。

7.2.1.1 *object_files* フィールド

object_files フィールドは，リンクするオブジェクトファイルを指定するのに使用します。少なくとも 1 つのファイル名を指定しなければなりません。拡張子を指定しなければ，RLU8 は拡張子をデフォルトの“.OBJ”と見なします。

複数のファイルを指定するときには，ファイル名前の間をスペースかプラス記号（+）で区切ります。*object_files* フィールドへの入力を次の行にまたがって指定するときは，現在の行の最後の文字としてプラス記号（+）を置いてリターンキーを押し，残りの入力を続けます。ただし，1 つの名前を 2 行に分割して指定することはできません。

object_files フィールドには，ライブラリファイルも指定できます。このフィールドでは，ファイル名の拡張子が“.LIB”のファイルだけをライブラリファイルとして扱います。このフィールドのデフォルトの拡張子は“.OBJ”となっていますので，ライブラリファイルを指定するときは，拡張子“.LIB”を必ず指定してください。拡張子を指定しなかったり，拡張子が“.LIB”でないとき，RLU8 はそのファイルをオブジェクトファイルとして扱います。

ライブラリファイルが指定されると，RLU8 は，未解決の外部参照を解決するかどうかにかかわらず，ライブラリファイル中のすべてのオブジェクトモジュールを取り出しリンクします。これは，そのライブラリファイル中のオブジェクトモジュールをすべて *object_files* フィールドに指定したのと同じことになります。この方法によって，RLU8 を起動するたびに多くのオブジェクトファイル名をタイプすることを避けることができます。

さらに，ライブラリファイル中の特定のモジュールだけをリンクすることもできます。この場合，次のようにライブラリファイル名（*library_filename*）に続いて，そのモジュールの名前（*module_name*）を指定します。

library_filename (*module_name* …)

複数のモジュール名を指定するときは，それらのモジュール名名前の間をスペースで区切ります。RLU8 は，指定されたモジュールだけをライブラリから取り出してリンクします。このとき，そのライブラリ中の残りのモジュールはリンクされません。

```
RLU8 MAIN PROJECT.LIB ( GETDATA CALC DISPLAY );
```

この例では，ライブラリ PROJECT.LIB の中からモジュール GETDATA，CALC，および DISPLAY だけを取り出して MAIN.OBJ とリンクします。

7.2.1.1.1 ファイルのサーチ方法

object_files フィールドで指定されたオブジェクトファイルおよびライブラリファイルをサーチする場合，RLU8 は以下の場所をサーチします。

(1) ファイル名がパス指定を含む場合，RLU8 はそのディレクトリ内でファイルをサーチします。ファイルが見つからなければ，サーチは終了します。

(2) ファイル名がパス指定を含まない場合，RLU8 はカレントドライブのカレントディレクトリ内でファイルをサーチします。ファイルが見つからなければ，サーチは終了します。

(3) 上記のいずれのサーチにおいてもファイルが見つからない場合，RLU8 はエラーメッセージを表示して処理を終了します。

7.2.1.1.2 起動方法の表示

object_files フィールドに入力を何も指定せず，リターンキーだけを入力するとプロンプトが表示されます。このプロンプトに対してリターンキーだけを入力すると，RLU8 は起動方法を表示して終了します。

7.2.1.2 *absolute_file* フィールド

absolute_file フィールドは，出力となるアブソリュートオブジェクトファイルの名前を指定するために使用します。拡張子を指定しなかった場合，RLU8 は拡張子をデフォルトの“.ABS”と見なします。

absolute_file フィールドに何も指定しなければ，RLU8 はオブジェクトファイルにデフォルトの名前を付けます。この名前は，*object_files* フィールドの先頭のファイル名に，拡張子“.ABS”を付けたものになります。

absolute_file フィールドにパスだけが指定されると，RLU8 はそのディレクトリに，デフォルトの名前でアブソリュートオブジェクトファイルを作成します。

アブソリュートオブジェクトファイルを作成するディレクトリは，明確にパスを指定しない限り，カレントディレクトリになります。

7.2.1.3 *map_file* フィールド

*map_file*フィールドは，マップファイルのファイル名を指定したり，マップファイルを作成しないようにするときに使用します。マップファイルは，リンク結果を示すテキストファイルです。マップファイルの形式については「7.7 マップファイル」を参照してください。

map_file フィールドに何も指定しなければ，RLU8 はマップファイルにデフォルトの名前を付けます。この名前は，アブソリュートオブジェクトファイル名の拡張子を“.MAP”にしたものになります。

map_file フィールドにパスだけが指定されると，RLU8 はそのディレクトリに，デフォルトの名前でマップファイルを作成します。

マップファイルを作成するディレクトリは，明確にパスを指定しない限り，カレントディレクトリになります。

マップファイルを作成しないようにするには，このフィールドに“NUL”を指定してください。

7.2.1.4 *libraries* フィールド

libraries フィールドには，ライブラリファイルを指定することができます。複数のファイルを指定するときには，名前の間をスペースかプラス記号（+）で区切ります。*libraries* フィールドへの入力を次の行にまたがって指定するときは，現在の行の最後の文字としてプラス記号（+）を置いてリターンキーを押し，残りの入力を続けます。ただし，1 つの名前を 2 行に分割して指定することはできません。拡張子を付けずにライブラリのベース名だけを指定すると，RLU8 はその拡張子をデフォルトの“.LIB”と見なします。

libraries フィールドで指定されたライブラリファイルは，未解決の外部参照を解決するために使用されます。RLU8 は，*libraries* フィールドに指定されている順に，ライブラリファイルをサーチします。ファイル名に明確にパスが指定されていれば，そのディレクトリをサーチしますが，パスが指定されていなければ，次の順でディレクトリをサーチします。

(1) カレントディレクトリ

(2) 環境変数 LIBU8 に定義されているディレクトリ

指定されたすべてのライブラリファイルを使っても，未解決な外部参照がある場合，デフォルトでは，それらは未解決のまま残りますが，/CC オプションが指定されていると，残りの外部参照を解決するために，RLU8 は，C 言語プログラムのためのエミュレーションライブラリをサーチします。RLU8 がサーチするエミュレーションライブラリは次のとおりです。

エミュレーションライブラリ名	内容	備考
LONGU8.LIB	整数演算ライブラリ	
DOUBLEU8.LIB	倍精度浮動小数点ライブラリ	CCU8 のオプションによって DOUBLEU8.LIB をサーチするか FLOATU8.LIB をサーチするか決定します。
FLOATU8.LIB	単精度浮動小数点ライブラリ	

libraries フィールドにパスを指定することによって，エミュレーションライブラリをサーチするディレクトリを RLU8 に知らせることができます。パス名を指定する場合，パス名の最後に円記号（¥）を付けなければなりません。パス名の最後に円記号（¥）が指定されなかった場合，RLU8 は，パス名の最後の名前をライブラリファイルのベース名と見なして，“.LIB”拡張子を持つファイルをサーチします。

RLU8 は，エミュレーションライブラリをサーチするとき，次の順に調べます。

(1) カレントディレクトリ

(2) *libraries* フィールドで指定されたディレクトリ

(3) 環境変数 LIBU8 に定義されているディレクトリ

RLU8 は，これらの場所でライブラリファイルを見つけられなかった場合，フェイタルエラーを表示して処理を終了します。

7.2.1.5 コマンドの例

ここでは，例を用いて，RLU8 コマンドラインの使い方を解説します。

例

```
RLU8 MAIN CALC DISP , , MAINLIST , USER.LIB
RLU8 MAIN + CALC + DISP , , MAINLIST , USER.LIB
```

この 2 つのコマンドはいずれも同じことを RLU8 に指示します。MAIN.OBJ，CALC.OBJ，および DISP.OBJ をリンクし，アブソリュートオブジェクトファイル MAIN.ABS が作成されます。マップファイルの名前は MAINLIST.MAP となります。そして，外部参照を解決するために USER.LIB を調べます。

例

```
RLU8 MAIN CALC DISP , , NUL ;
```

この例では MAIN.OBJ，CALC.OBJ，および DISP.OBJ をリンクし，アブソリュートオブジェクトファイル MAIN.ABS を作成します。*map_file* フィールドに NUL が指定されていますので，マップファイルは作成されません。

例

```
RLU8 PROJECT1.LIB;
```

この例では，ライブラリ PROJECT1.LIB の中のすべてのモジュールをリンクして，アブソリュートオブジェクトファイル PROJECT1.ABS を作成します。マップファイルの名前は PROJECT1.MAP となります。

例

```
C> RLU8 MAIN GETDATA +
INPUT FILES [.OBJ]: CALC ERRHDL +
INPUT FILES [.OBJ]: DISPLAY USER.LIB ;
```

この例では，*object_files* フィールドへの入力を 3 行にわたって指定しています。最初の行（コマンドライン）では，MAIN.OBJ と GETDATA.OBJ を入力しています。この行の最後の文字がプラス記号になっていますので，RLU8 は引き続き *object_files* フィールドへの入力を要求するプロンプトを表示します。次に，CALC.OBJ と ERRHDL.OBJ を入力し，最後にプラス記号を置きます。すると，RLU8 は再び *object_files* フィールドへの入力を要求するプロンプトを表示します。次に，DISPLAY.OBJ と USER.LIB に続けてセミコロンを入力します。セミコロンが指定されると，RLU8 はコマンドが終了したものとみなし，残りのフィールドに対するプロンプトを表示せずにリンク処理を開始します。

7.2.2 実行方法

RLU8 は，少なくとも 1 つのオブジェクトファイルを指定することによって処理を開始します。

RLU8 が必要とする入力を指定する方法には，次の 3 つがあります。

(1) 必要な入力をすべてコマンドラインで直接指定する。

(2) RLU8 が表示するプロンプトに対して指定する。

(3) 応答ファイルに指定を入力し，応答ファイルの名前をコマンドラインの定められた位置に指定する。

これらの方法は，組み合わせて使用することができます。コマンドラインで直接指定する方法については，「7.2.1 コマンドラインの書式」を参照してください。ここでは，その他の 2 つの方法について説明します。

7.2.2.1 プロンプトで入力する方法

コマンドラインのフィールドの一部が省略されていて，コマンドラインがセミコロンで終了していない場合，RLU8 は省略された入力に対してプロンプトを表示します。RLU8 は，以下の行を 1 行ずつ表示することによって，必要な入力をうながします。

```
INPUT FILES [.OBJ]:
OUTPUT FILE [base_name.ABS]:
MAP FILE [base_name.MAP]:
LIBRARIES [.LIB]:
```

RLU8 は，それぞれのプロンプトに応答されるまで，次の行を表示しません。これらのプロンプトは，「7.2.1 コマンドラインの書式」で説明したコマンドラインのフィールドに対応しています。これらの対応は，次のとおりです。

プロンプト	コマンドラインのフィールド
`INPUT FILES`	*object_files* フィールド
`OUTPUT FILE`	*absolute_file* フィールド
`MAP FILE`	*map_file* フィールド
`LIBRARIES`	*libraries* フィールド

すべてのフィールドに対してプロンプトを使って指定するときは，DOS のプロンプトに対して RLU8 とだけ入力します。

オプションは，セミコロンが入力される前であれば，任意のフィールドのどの位置にでも指定することができます。

各フィールドのデフォルト値は，角カッコ内に表示されます。デフォルト値でよければ，リターンキーを入力します。デフォルト値以外に変更したい場合，そのファイル名をタイプします。*base_name* は，*object_files* フィールドで最初に指定したファイルのベース名となります。残りのプロンプトすべてに対して，デフォルトを指定し，プロンプトを表示しないようにするには，セミコロンを入力してからリターンキーを押します。

拡張子を付けずにファイル名を指定すると，RLU8 はデフォルトの拡張子を付け加えます。拡張子を持たないファイル名を指定するには，その名前の直後にピリオド（.）を付けます。

object_files や *libraries* フィールドに複数のファイルやパスを指定するときは，名前の間をスペースまたはプラス記号（+）で区切ります。*object_files* や *libraries* に対する応答が長くて次の行にわたる場合，現在の行の最後の文字としてプラス記号を置いてリターンキーを押し，残りの入力を続けます。新しい行に同じプロンプトが表示されれば，応答を続けて入力することができます。ただし，1 つのファイル名やパス名を 2 行に分割して指定することはできません。

7.2.2.2 応答ファイルによる入力の指定

応答ファイルを使って RLU8 に入力を与えることができます。応答ファイルとは，コマンドラインやプロンプトに対する入力を含んだテキストファイルのことです。応答ファイルを利用すると，頻繁に使うオプションや入力を保存したり，DOS のコマンドラインの制限である 127 文字を越えて指定することができます。

注意

RLU8 は応答ファイルから取り出した内容をコマンド解析用のバッファにコピーします。このため，応答ファイルに記述できる内容はコマンド解析用のバッファサイズに依存します。コマンド解析用のバッファは 32K バイト確保しています。コマンド解析用のバッファをオーバーした時点で，RLU8 はエラーを表示し強制終了します。

応答ファイルの使い方

コマンドラインの任意の位置や任意のプロンプトに，応答ファイルの名前を指定します。ファイル名は@記号の直後に指定します。応答ファイルにはデフォルトの拡張子はありませんので，応答ファイルに拡張子がある場合には必ず指定してください。ファイル名にはパスを指定することができます。

応答ファイルは，どのフィールド（コマンドラインのものや各プロンプトに対するもの）に対しても指定でき，1 つまたは複数のフィールドや残りのフィールドへの指定として使うことができます。RLU8 は，応答ファイルの内容については特に規定していません。RLU8 は応答ファイルからフィールドを読み取り，入力されていないフィールドに対して順に割り当てていきます。RLU8 は，4 つのフィールドがすべて満たされるか，セミコロンに出会うと，以降の応答ファイル中のフィールドやコマンドラインの指定を無視します。

例

```
RLU8 MAIN @MYOBJ.RES , MYLIB.LIB
```

上の例では，オブジェクトファイル MAIN.OBJ の後に応答ファイル MYOBJ.RES を指定しています。MYOBJ.RES の内容は，“SUB , OUT , MAP”と記述されているものとします。

次の例は，上の例と同じ入力を，プロンプトを使って指定したものです。

```
C>RLU8 MAIN +
INPUT FILES [.OBJ]: @MYOBJ.RES ,
LIBRARIES [.LIB]: MYLIB.LIB
```

応答ファイルの内容

入力フィールドは，同じようにコンマでフィールドを区切ります。改行は単なる空白として扱います。オプションは，セミコロンが指定される前であれば，どのフィールドのどの位置にでも入力することができます。

応答ファイル内の任意の位置にコメントを記述することができます。コメントは RLU8 の処理に対して何の影響も与えません。コメントは，2 つの連続したスラッシュ記号（//），またはシャープ記号（#）に続けて記述します。 RLU8 は，“//”または“#”が現れると，そこから改行文字までに現れるすべての文字をコメントとして扱います。

以下に応答ファイルの例を示します。

```
1: // TM MODEL X1
2: // RELEASE 2.3.1
3: TMX1 GETDATA CALC
4: COMP DISPLAY    #new module
5: TABLE           #original data
6: /S
7: ,,TMX1LIST,
8: // library
9: MATH.LIB
```

左端の行番号は説明のためのもので，実際の応答ファイルには記述されていません。1，2 および 8 行目は，コメントだけが記述されているので無視されます。3 行目の TMX1 から 6 行目の /S までが *object_files* フィールドに対応します。*absolute_file* フィールドは指定なしで，*map_file* フィールドに TMX1LIST が，*libraries* フィールドに MATH.LIB が指定されたことになります。

そして，この応答ファイル名を TMX1.RES とし，次のように RLU8 を起動します。

```
RLU8 @TMX1.RES
```

RLU8 は，応答ファイルにしたがって次のように動作します。

(1) TMX1.OBJ，GETDATA.OBJ，CALC.OBJ，COMP.OBJ，DISPLAY.OBJ，TABLE.OBJ という６つのオブジェクトファイルをリンクし，TMX1.ABS という名前のアブソリュートオブジェクトファイルを作成します。

(2) ライブラリファイル MATH.LIB から必要なモジュールを取り出し，リンクします。

(3) TMX1LIST.MAP という名前のマップファイルを作成します。

(4) /S オプションが指定されているので，マップファイルにパブリックシンボルと共有シンボルのテーブルを出力します。

7.3 処理状態を示すメッセージ

RLU8 は，現在どのような処理を行っているのかを示すメッセージを表示します。

RLU8 に対して必要な項目が入力されると，RLU8 は起動メッセージを表示して，次のメッセージを順番に表示していきます。

```
Loading segments and symbols...
Allocating segments...
Writing fixed data...
```

“Loading segments and symbols...”は，入力されたオブジェクトモジュールから，セグメントとシンボルを読み込んでいることを示します。

“Allocating segments...”は，セグメントをメモリ空間に割り付けていることを示します。

“Writing fixed data...”は，未解決オペランドをフィックスアップしていることを示します。

すべての処理が正常に終了すると，次のメッセージが表示されます。

```
Absfile: absolute_file
Mapfile: map_file
Ablfile: abl_file

Linkage completed.
```

absolute_file，*map_file* は，リンク処理の結果作られたアブソリュートファイルとマップファイルの名前です。*abl_file* は ABL ファイル名です。ABL ファイル名については，/A オプションが指定された場合にのみ表示されます。

エラーが発生した場合，RLU8 はそのときの処理過程に応じて次のいずれかのメッセージを表示して，処理を中止します。

```
Discontinue!  Loading error detected.
Discontinue!  Allocation error detected.
Discontinue!  Fix up error detected.
```

“Discontinue! Loading error detected.”は，セグメントとシンボルを読み込んでいるときにエラーが発生したことを示します。

“Discontinue! Allocation error detected.”は，セグメントをメモリ空間に割り付けているときにエラーが発生したことを示します。

“Discontinue! Fix up error detected.”は，未解決オペランドをフィックスアップしているときにエラーが発生したことを示します。

7.4 終了コード

RLU8 は，動作終了時に，以下に示す終了コードのうちのいずれかを返します。終了コードは MAKE ファイル，またはバッチファイルの中で使用することができます。

終了コード	意味
0	エラーはない。
1	ワーニングがあった。
2	エラーがあった。
3	フェイタルエラーがあった。
4	コマンドラインエラーがあった。
5	ユーザによって Ctrl+C が入力された。

終了コードが 2，3，または 4 の場合，RLU8 はアブソリュートオブジェクトファイルを作成しません。

7.5 オプション

ここでは，RLU8 の動作の制御や出力の変更を行うオプションの使い方を説明します。各オプションの解説に加えて，オプションの指定方法についても紹介します。

7.5.1 オプションの指定方法

はじめに，オプションを使用するときの規則を説明します。

7.5.1.1 構文

オプションの構文は次のとおりです。

/*option_name* [(*argument_list*)]

すべてのオプションは，スラッシュ記号（/）またはハイフン記号（-）で始まります。このオプション開始記号に続けてオプション名 *option_name* を指定します。いくつかのオプションは，引数 *argument_list* を必要とします。引数は，オプション名の後に，丸カッコで囲んで指定します。オプション開始記号，オプション名，および左丸カッコの間には，スペースを入れてもかまいません。

RLU8 は，オプション名の大文字，小文字を区別しません。たとえば，/CODE オプションを/Code または/code と指定することもできます。

7.5.1.2 指定位置

オプションは，コマンドライン，プロンプトへの応答，または応答ファイル内のフィールドの，任意の位置に指定できます。オプションを複数の場所で指定しても，1 箇所にまとめて指定してもかまいません。

7.5.1.3 名前の引数

オプションには，名前を引数としてとるものがあります。RLU8 は，引数として与えられる名前の大文字，小文字を常に区別します。たとえば，次の/CODE オプションの引数 MOUSE，mouse，Mouse は，それぞれ別の名前として扱われます。

```
/CODE( MOUSE mouse Mouse )
```

注意

RASU8 はデフォルトでシンボルの大文字，小文字を区別しますが，シンボルの大文字，小文字を区別しないためのオプション/NCD が用意されています。/NCD オプションが指定された場合，RASU8 はプログラマが定義したシンボルをすべて大文字に変換し，オブジェクトファイルに出力します。したがって，/NCD オプションを付けてアセンブルされたモジュール中で定義されているシンボルを RLU8 のオプションの引数にする場合は，必ず大文字で指定する必要があります。

7.5.1.4 アドレスの引数

オプションには，メモリ空間のアドレスを引数としてとるものがあります。アドレス指定は，物理セグメントアドレスとオフセットアドレスをコロン（:）で区切って次のように指定します。

[*physical_seg* :]*offset*

physical_seg には，0 から 0FFH までの物理セグメントアドレス，*offset* には 0 から 0FFFFH までのアドレスを，10 進数か 16 進数のいずれかの表記法で指定します。物理セグメントアドレスとコロンを省略すると，物理セグメントアドレス 0 と解釈します。

physical_seg とコロン（:）の間，またはコロンと *offset* の間に空白が入ってもかまいません。

10 進数は，0 から 9 までの 10 進数字を使って表します。16 進数は，0 から 9 までの数字と A から F（または a から f）までの英字を使って表します。その場合，数値の最後には，“H”または“h”を付けます。たとえば，16 進数 1234 は，1234H または 1234h と表現します。数値の先頭の文字が，A から F（または a から f）までの英字となるときには，その文字の直前に数字の 0 を付けて指定します。たとえば，16 進数 C800H は，先頭の文字が英字の C なので，その直前に 0 を付けて 0C800H と表現します。

7.5.2 オプション一覧

RLU8 が用意するオプションを以下に示します。アスタリスク（*）は，そのオプションの機能がデフォルトで指定されることを表します。

オプション		機能
/D		デバッグ情報を出力する。
/ND	*	デバッグ情報を出力しない。
/S		パブリックシンボルリストを出力する。
/NS	*	パブリックシンボルリストを出力しない。
/CODE		CODE セグメントの割り付けを制御する。
/DATA		DATA セグメントの割り付けを制御する。
/BIT		BIT セグメントの割り付けを制御する。
/NVDATA		NVDATA セグメントの割り付けを制御する。
/NVBIT		NVBIT セグメントの割り付けを制御する。
/TABLE		TABLE セグメントの割り付けを制御する。
/ORDER		同じ優先度を持つセグメントの割り付け処理順を制御する。
/ROM		外部 ROM の実装範囲を設定する。
/RAM		外部 RAM の実装範囲を設定する。
/NVRAM		外部 NVRAM の実装範囲を設定する。

オプション		機能
/NVRAMP		物理セグメント#0 のプログラムメモリ空間に割り当てる外部 NVRAM の実装範囲を設定する。
/CC		エミュレーションライブラリの自動サーチを指定する。
/SD		C ソースレベルデバッグ情報を出力する。
/NSD	*	C ソースレベルデバッグ情報を出力しない。
/STACK		スタックセグメントのサイズを変更する。
/A		ABL ファイルを作成する。
/NA	*	ABL ファイルを作成しない。
/COMB		CODE セグメントまたは TABLE セグメントを結合する。
/EXC		未使用のイクスターナルシンボルを未解決エラーにしない。
/ROMWIN		ROMWINDOW 領域を指定する。
/PDIF		モジュール間でのメモリ情報の違いを許す。
/OVERLAY		オーバーレイを指定する。
/LA		セグメント参照の有無に関わらずすべてのセグメントをリンクする。
/CP		TABLE テーブルセグメントの割り付け優先度を変更する。

7.5.3 各オプションの機能

7.5.3.1 /D, /ND

構文

/D

/ND

説明

/D オプションは，デバッガが使用するアセンブリレベルデバッグ情報をアブソリュートオブジェクトファイルに出力することを RLU8 に指示します。このデバッグ情報には，ローカルシンボル，パブリックシンボル，共有シンボル，およびセグメント名が含まれます。入力オブジェクトファイルにアセンブリレベルデバッグ情報が含まれていなければ，/D オプションを指定しても効果はありません。

/ND オプションは，/D オプションの効果を抑制するためのオプションです。コマンドライン上で/D と/ND が同時に指定された場合は，より後に指定された方が有効となります。

また，/D，/ND は何度指定してもかまいませんが，より後に指定されたほうが有効となります

デフォルトは，/ND です。

7.5.3.2 /S, /NS

構文

/S

/NS

説明

/S オプションは，オブジェクトファイルで定義されているすべてのパブリックシンボルおよび共有シンボルのリストをマップファイルに追加することを RLU8 に指示します。シンボルはアルファベット順に並べられて出力されます。

map_file フィールドに NUL が指定されていると，マップファイルは作成されないので，このオプションを指定しても効果はありません。

/NS オプションは，/S オプションの効果を抑制するためのオプションです。コマンドライン上で/S と/NS が同時に指定された場合は，より後に指定された方が有効となります。

また，/S，/NS は何度指定してもかまいませんが，より後に指定された方が有効となります。

デフォルトは，/NS です。

7.5.3.3 /CODE, /TABLE, /DATA, /BIT, /NVDATA, /NVBIT

構文

/CODE ([*address*] *segment_name* [-] [*address*] ...)

/TABLE ([*address*] *segment_name* [-] [*address*] ...)

/DATA ([*address*] *segment_name* [-] [*address*] ...)

/BIT ([*address*] *segment_name* [-] [*address*] ...)

/NVDATA ([*address*] *segment_name* [-] [*address*] ...)

/NVBIT ([*address*] *segment_name* [-] [*address*] ...)

説明

これらのオプションは，リロケータブルセグメントの割り付け処理を制御するのに使用します。これらのオプションをまとめて，セグメント割り付け制御オプションと呼びます。

*segment_name*には，セグメント名を指定します。*address*には，「7.5.1.4 アドレスの引数」で説明した表記法を使って指定します。たとえば，アドレス 1234Hは，1234Hと指定します。このアドレスは，セグメントタイプに応じた割り付け可能なアドレス空間の範囲内でなければなりません。

それぞれのセグメント割り付け制御オプションは，オプションの種類によって指定できる論

理セグメントの種類が限定されます。

セグメント割り付け制御オプションの種類と，そのオプションに指定できる論理セグメントを以下に示します。

セグメント割り付け制御オプション	指定できる論理セグメントのセグメントタイプ
/CODE	CODE セグメント
/TABLE	TABLE セグメント
/DATA	DATA セグメント
/BIT	BIT セグメント
/NVDATA	NVDATA セグメント
/NVBIT	NVBIT セグメント

論理セグメントは，通常，「7.6.6.3 割り付けの優先度」に示される優先度にしたがって，プログラムメモリ空間，またはデータメモリ空間上に実装されたROM，RAM，またはNVRAMの範囲内に割り付けられます。

セグメント割り付け制御オプションで指定されたリロケータブルセグメントは，他のリロケータブルセグメントより高い優先度を与えられ，先に処理されることになります。

セグメント割り付け制御オプションを使用して，リロケータブルセグメントに対して，次のような指定ができます。

(1) セグメントを特定のアドレスよりも上位のメモリに割り付ける。

(2) セグメントを特定のアドレスに割り付ける。

以下に各指定方法について順に説明し，最後に，これらを組み合わせて指定した場合を紹介します。

(1)セグメントを特定のアドレスよりも上位のメモリに割り付ける

セグメント名と基準となるアドレスを指定します。

RLU8 は，対象の論理セグメントをリロケータブルセグメントとして扱います。

基準アドレスの初期値は 0 とします。そして，カッコ内にアドレスが現れると，その値に更新されます。またこの基準アドレスは，/CODE オプションが現れるごとに初期値に戻ります。次にこの指定方法の例を示します。

例

```
/CODE(SEG1 SEG2 100H SEG3 SEG4) /CODE(SEG5)
```

基準アドレスの初期値は 0 ですから，セグメント SEG1 と SEG2 はアドレス 0 より上位アドレスに割り付けられます。そして，アドレス 100H が現れたところで基準アドレスがそのアドレスに更新されます。続いて指定されているセグメント SEG3 と SEG4 は，アドレス 100H より上位

アドレスに割り付けられます。セグメント SEG5 は，新たな CODE オプションによって指定されているため，基準値の初期値にしたがって，アドレス 0 より上位アドレスに割り付けられることになります。以上のことをまとめて，各セグメントが割り付けられる範囲を次の表に示します。

セグメント	割り付けられる範囲
SEG1	0000H ～ FFFFH
SEG2	0000H ～ FFFFH
SEG3	0100H ～ FFFFH
SEG4	0100H ～ FFFFH
SEG5	0000H ～ FFFFH

(2)セグメントを特定のアドレスに割り付ける

これは，特定のアドレスにセグメントを割り付ける方法です。セグメント名の後にハイフン（-）を置き，続けてアドレスを指定します。RLU8 は，指定されたアドレスに，そのセグメントの先頭を置きます。

RLU8 は，対象のセグメントをアブソリュートセグメントと同等に扱います。

次にこの指定方法の例を示します。

```
/CODE(SEG1-3800H SEG2-8000H SEG3-1:2000H)
```

この例では，最初に，SEG1 を 3800H に割り付けます。次に，SEG2 を 8000H に割り付けます。最後に，SEG3 を 1:2000H に割り付けます。これらは，次のように分けて指定してもかまいません。

```
/CODE(SEG1-3800H) /CODE(SEG2-8000H) /CODE(SEG3-1:2000H)
```

(3)すべての指定方法を組み合わせて指定する

以上，説明した指定方法は，次のように組み合わせて使用することもできます。

```
/CODE(0F0H SEG1 SEG2-100H SEG3 200H SEG4 SEG5-1:300H SEG6)
```

これらのセグメントは次に示す範囲に割り付けられます。

セグメント	割り付けられる範囲
SEG1	00F0H ～ FFFFH
SEG2	0100H
SEG3	0100H ～ FFFFH
SEG4	0200H ～ FFFFH

セグメント	割り付けられる範囲
SEG5	1:0300H
SEG6	1:0300H ～ 1:FFFFH

セグメント割り付け制御オプションで指定されたアドレスが，ソースプログラムでそのセグメントに指定されている境界値属性と矛盾していてもかまいません。この場合，RLU8 は，その属性を無視し，指定されたアドレスにセグメントを割り付けます。このとき，RLU8 はワーニングメッセージを表示します。

補足

/BIT オプション，または/NVBIT オプションを使用して *address* を指定する場合，*address* はビットアドレスとなります。

ビットアドレスは，次のいずれかの方法を使って指定できます。

(1) ビットアドレスを，「7.5.1.4 アドレスの引数」で説明した表記法を使って直接指定する。

(2) 次に示すように，バイトアドレスとビット位置をピリオド（.）で区切って，アドレスを指定する。

data_address.bit_position

*data_address*にはデータアドレスを指定します。*bit_position*にはビット位置を表わす 0 から 7 までの数値を指定します。これらには，「7.5.1.4 アドレスの引数」で説明した表記法を使用します。

たとえば，データアドレス 1234H のビット 5 は，91A5H，または 1234H.5 と指定することができます。

アドレス引数として指定できる値の最大は 0FFFFH です。したがって，/BIT，/NVBIT オプションの引数にビットアドレスを直接指定する方法の場合，ビットアドレス 0FFFFH までしか指定できません。これを超えるビットアドレスを指定する場合は，バイトアドレスとビットアドレスを組み合わせて指定する方法を使います。たとえば，ビットアドレス 7FFF3H を指定したいときは，0FFFEH.3 のように指定します。

7.5.3.4 /ORDER

構文

/ORDER(*segment_name* ...)

説明

/ORDER オプションは，同じ優先度を持つセグメントの割り付け処理の順序を制御するのに使用します。通常，セグメントは，それ自身の持つ優先度にしたがって順番に割り付け処理されていきますが，同じ優先度を持つセグメントどうしの処理順は任意です。

segment_name には，セグメント名を指定します。指定されたセグメントは，その並びの順に

割り付け処理されることになります。

/ORDER オプションが複数指定された場合，1 つの/ORDER オプションは，他の/ORDER オプションに何の影響も与えません。たとえば，同じ優先度を持つセグメント SEG1，SEG2，SEG3，そして SEG4 があるとき，これらに対して次のように指定したとします。

```
/ORDER(SEG1 SEG2) /ORDER(SEG3 SEG4)
```

この場合，最初の/ORDER オプションによって SEG1，SEG2 の順に処理されることが指定され，2 番目の/ORDER オプションによって SEG3，SEG4 の順に処理されることが指定されます。しかし，この 2 組の処理順は指定されません。

また，1 つの/ORDER オプションで指定されているセグメントは，同じ優先度を持つ必要はありません。たとえば，同じ優先度を持つセグメント SEG1，SEG2，および，それより高い，同じ優先度を持つセグメント SEG3，SEG4 があるとき，次のように指定したとします。

```
/ORDER( SEG1 SEG2 SEG3 SEG4 )
```

この場合の処理順は，セグメントが本来持っている優先度と，/ORDER オプションの指定によって，SEG3，SEG4，SEG1，SEG2 になります。

7.5.3.5 /ROM, /RAM, /NVRAM, /NVRAMP

構文

/ROM (*start_address*, *end_address*)

/RAM (*start_address*, *end_address*)

/NVRAM (*start_address*, *end_address*)

/NVRAMP (*start_address*, *end_address*)

説明

これらのオプションは，外部メモリの実装範囲を指定します。

/ROM オプションは，外部 ROM の実装範囲を指定します。/RAM オプションは，外部 RAM の実装範囲を指定します。/NVRAM オプションは，物理セグメント#0 のデータメモリ空間，および物理セグメント#1 以上のメモリ空間に実装される外部不揮発性メモリの範囲を指定します。/NVRAMP オプションは，物理セグメント#0 のプログラムメモリ空間に実装される外部不揮発性メモリの範囲を指定します。

start_address には外部メモリの開始アドレスを，*end_address*には外部メモリの終了アドレスをそれぞれ指定します。アドレスの記述に関しては，「7.5.1.4 アドレスの引数」で説明した表記法を参照してください。

RLU8 は，これらのオプションで指定された領域を，外部メモリが実装されている領域として認識し，その領域へ各論理セグメントと共有シンボルを割り付けます。

これらのオプションは複数の指定が可能です。オプションで指定した領域が重なっていた場合には，メモリの種類が同一の場合には連結されますが，メモリの種類が異なる場合には，重なった領域を除いた部分が有効となります。

これらのオプションが指定されていない場合，外部メモリは実装されていないものと判断し，オブジェクトモジュール中のメモリ情報にしたがって，セグメントおよび共有シンボルを割り付けていきます。

外部 ROM の実装範囲を 1:0000H から 3:FFFFH，外部 RAM の実装範囲を 4:0000H から 7:FFFFH とし，物理セグメント#0 のプログラムメモリ空間に実装する外部不揮発性メモリの範囲を 0:A000H から 0:BFFFH，それ以外の外部不揮発性メモリの実装範囲を 8:0000H から 9:FFFFH とする場合，次のように指定します。

```
/ROM( 1:0000H, 3:0FFFFH )
/RAM( 4:0000H, 7:0FFFFH )
/NVRAMP( 0:0A000H, 0:0BFFFH )
/NVRAM( 8:0000H, 9:0FFFFH )
```

/ROM オプションで指定した範囲には，CODE セグメント，TABLE セグメントが割り付けられます。/RAM オプションで指定した範囲には，DATA セグメント，BIT セグメントが割り付けられます。/NVRAM オプションで指定した範囲には，NVDATA セグメント，NVBIT セグメントが割り付けられます。

7.5.3.6 /CC

構文

/CC

説明

/CCオプションが指定されると，RLU8 はC言語プログラムのために用意されたエミュレーションライブラリを自動的にサーチし，必要なモジュールを取り出しリンクします。詳細については，「7.2.1.4 *libraries*フィールド」を参照してください。

また，本オプションが指定されると，以下のオプションも自動的に指定されたことになります。

```
/COMB($$init_info $$init_info_end)
/COMB($$content_of_init $$end_of_init)
```

このオプションで指定されているセグメントは，C コンパイラ CCU8 が出力するものです。これらのセグメントには，おもに C 言語プログラムを実行するのに必要な初期化情報が含まれます。

7.5.3.7 /SD, /NSD

構文

/SD

/NSD

説明

/SD オプションは，C ソースレベルデバッガが使用する C ソースレベルデバッグ情報をアブソリュートオブジェクトファイルに出力することを RLU8 に指示します。このデバッグ情報には，行番号と変数に関する情報が含まれています。入力オブジェクトファイルに C ソースレベルデバッグ情報が含まれていなければ，/SD オプションを指定しても効果はありません。

/NSD オプションは，/SD オプションの効果を抑制するためのオプションです。コマンドライン上で，/SD と/NSD が同時に指定された場合は，より後に指定された方が有効になります。

また，/SD，/NSD は何度指定してもかまいませんが，より後に指定されたほうが有効となります。デフォルトは，/NSD です。

7.5.3.8 /STACK

構文

/STACK(*size*)

説明

/STACK オプションは，スタックセグメントのサイズを変更するのに使用します。スタックセグメントはアセンブラ擬似命令 STACKSEG によって定義されるものです。

size には，変更したいスタックセグメントのサイズを指定します。スタックセグメントは物理セグメント#0 に割り付けられますので，その範囲内に収まるサイズでなければなりません。

スタックセグメントは必ず偶数サイズで確保されなければなりませんので，*size* には偶数値を指定してください。*size* に奇数値が指定された場合，RLU8 は指定された値に 1 を加え，偶数値に調整します。

/STACK オプションが指定されたとき，入力モジュール中にスタックセグメントが定義されていなければエラーになります。

7.5.3.9 /A, /NA

構文

/A [(*abl_file*)]

/NA

説明

/A オプションは，ABL ファイルを作成することを RLU8 に指示します。ABL ファイルとは，RASU8 がアブソリュートプリントファイルを作成するときに必要となるファイルのことです。

アブソリュートプリントファイルについての詳細は，「11 アブソリュートリスティング機能」を参照してください。

abl_file には作成する ABL ファイルの名前を指定します。これを省略した場合，ABL ファイル名はアブソリュートオブジェクトファイル名の拡張子を“.ABL”にしたものになります。

/NA オプションは，/A オプションの効果を抑制するためのオプションです。コマンドライン上で/A と/NA が同時に指定された場合は，より後に指定された方が有効になります。

また，/A，/NA は何度指定してもかまいませんが，より後に指定されたほうが有効となります。デフォルトは，/NA です。

7.5.3.10 /COMB

構文

/COMB（*segment_name1 segment_name2 ...*）

説明

セグメントの *segment_name1* と *segment_name2* を，この順に結合してメモリ空間に割り付けます。それぞれのセグメントの指定は空白で区切ります。セグメントは 3 つ以上指定することもできます。セグメントは，CODE タイプまたは TABLE タイプのリロケータブルセグメントでなければなりません。

また，結合するセグメントのタイプ，特殊領域属性，物理セグメント属性，物理セグメントアドレスはすべて一致していなければなりません。*segment_name2* の属性が *segment_name1* の属性と一致しない場合，ワーニングを表示し，*segment_name2* を結合の対象から外します。ただし，バウンダリ値が異なる場合は，ワーニングを表示し，大きい方のバウンダリ値を有効とします。

segment_name1，*segment_name2* のどちらか一方が存在しない場合，何の影響も及ぼしません。

本オプションで指定されたセグメントのうち，2 番目以降に指定されたセグメント（以降，サブセグメント）に対し，/CODE オプションまたは/TABLE オプションが指定されている場合には，リンカはワーニングを表示して，サブセグメントに対する/CODE オプションまたは/TABLE オプションの指定を無視します。

例） `/COMB(SegA SegB) /CODE(100h SegA SegB)`

上の例の場合、/COMB オプションで指定された SegA に対する/CODE オプションが有効となります。リンカは，SegB に対する/CODE オプションを無視します。

7.5.3.11 /EXC

構文

/EXC

説明

未解決で残ったイクスターナルシンボルがあっても，それが未使用であれば未解決エラーを出力しないようにします。

7.5.3.12 /ROMWIN

構文

/ROMWIN（*start_address* , *end_address*）

説明

ROM ウィンドウ領域を指定します。*start_address*，*end_address* には，ROM ウィンドウ領域の開始アドレス，終了アドレスを指定します。ROM ウィンドウ領域が決定していた入力モジュールがあった場合，その領域と/ROMWIN オプションで指定した ROM ウィンドウ領域が一致していなければエラーとなります。*start_address*，*end_address* の指定値は，DCL ファイル中の ROM ウィンドウ領域の情報と一致していなければなりません。

7.5.3.13 /PDIF

構文

/PDIF

説明

モジュール間にメモリ情報の違いがあっても，リンクを可能としたい場合にこのオプションを指定します。このオプションは，RASU8 の/B オプションによって外部メモリが指定されたときに，同じマイクロコントローラのモジュールでも，メモリ情報が異なる場合があることを想定して用意しています。

このオプションが指定された場合，モジュール間でメモリ情報の違いがあった場合にはワーニングを表示し，メモリ情報の論理 OR をとります。ただし，/PDIF オプションが指定されている場合でも，メモリ情報に重なりがあり，その領域のメモリの種類が異なっている場合にはフェイタルエラーで終了します。

7.5.3.14 /OVERLAY

構文

/OVERLAY(*area_name*, *start_address*, *end_address*){[*overlay_unit*[*overlay_unit* ...]]}

説明

/OVERLAY オプションは，オーバーレイ機能を利用する場合に使用します。/OVERLAY オプションには，オーバーレイ領域と，そのオーバーレイ領域に割り付けるオーバーレイユニットを構成するセグメントを指定します。

オーバーレイ機能の詳細については，「10 オーバーレイ機能」を参照してください

area_name にはオーバーレイの領域名を，*start_address*, *end_address* にはオーバーレイ領域の開始アドレス，終了アドレスをそれぞれ指定します。{} で括られる部分にはオーバーレイユニット *overlay_unit* を指定します。 *overlay_unit* は，次のフォーマットで指定します。

overlay_unit の構文

UNIT(*segment* [*segment* ...])

オーバーレイユニットの指定は UNIT で始まり，()で括られる部分にオーバーレイユニットを構成するセグメントの名前を列挙します。各セグメントは空白で区切ります。なお，RLU8 は，セグメントを指定された順にオーバーレイ領域へ割り付けていきます。

オーバーレイ領域は，重なって定義してもかまいませんが，オーバーレイ領域が重なる場合には，RLU8 は注意を促すためワーニングを表示します。

オーバーレイ領域の名前は，重複して指定することはできません。重複して指定した場合には，RLU8 はエラーを表示します。

1 つのオーバーレイ領域は，複数の物理セグメントにまたがってはいけません。複数の物理セグメントにまたがって指定した場合には，RLU8 はエラーを表示し強制終了します。

オーバレイユニットを構成するセグメントは，オーバレイオプション全体で重複して指定することはできません。重複して指定した場合には，RLU8 はエラーを表示し強制終了します。

7.5.3.15 /LA

構文

/LA

説明

すべての関数・テーブルをリンクする場合に，このオプションを指定します。

このオプションが指定された場合は，セグメントの参照関係のチェックを行わずに、すべての関数・テーブルをメモリへの割り付け対象として扱います。

7.5.3.16 /CP

構文

/CP

説明

物理セグメント指定なしの TABLE セグメントの割り付け優先度を 16（V1.50 以前と同じ優先度）に変更する場合に，このオプションを指定します。

セグメントの割り付け優先度については，「7.6.6.3 割り付けの優先度」を参照してください。

7.6 リンク処理

RLU8 は，*object_files* フィールドで指定されたファイルからオブジェクトモジュールを読み込んだあと，アブソリュートオブジェクトファイルを作るために，以下の手順で処理をします。

(1) リンクしようとしているモジュールが，お互いにリンク可能なものかどうかを検証(モジュールのマッチングチェック)します。

(2) グローバルシンボルを対応付けます。必要であれば，イクスターナルシンボルを解決するために与えられたライブラリをサーチします。

(3) 同じ名前のセグメントを結合します。

(4) 同じ名前の共有シンボルを結合します。

(5) CODE/TABLE セグメントの参照関係をチェックし，参照されない CODE/TABLE セグメントをメモリへの割り付け対象から除外します。

(6) セグメント，共有シンボル，および擬似セグメントをメモリに割り付けます。

(7) 未解決なオペランドをフィックスアップし，アブソリュートオブジェクトファイルに出力します。

以下に，リンク処理を理解するために必要な項目について説明します。

7.6.1 モジュールのマッチングチェック

RLU8 は，リンクしようとしているモジュールがお互いにリンク可能なものかどうかを検証します。

RLU8 が行うモジュール間のマッチングチェックは次のとおりです。

(1) CPU コア

(2) マイクロコントローラ名

(3) メモリモデル

(4) メモリ情報

(5) ROMWINDOW 属性

7.6.1.1 CPU コアのマッチングチェック

CPU コアはお互いに一致しなければなりません。CPU コアが一致しない場合は，フェイタルエラーで終了します。

7.6.1.2 マイクロコントローラ名のマッチングチェック

マイクロコントローラ名は，お互いに一致しなければなりません。ただし，固有のマイクロコントローラ名を持たない汎用モジュールは，どのようなマイクロコントローラのモジュールともリンク可能です。以下に結合規則を示します。

モジュール 1	モジュール 2	継承の結果
汎用	汎用	汎用（最終的に汎用になった場合は，エラーになります）
汎用	ML610001	ML610001
ML610001	ML610001	ML610001
ML610002	ML610001	CPU 名が違うのでエラーとなります

7.6.1.3 メモリモデルのマッチングチェック

メモリモデルは，一致しなければなりません。メモリモデルが一致しない場合は，フェイタルエラーで終了します。

7.6.1.4 メモリ情報のマッチングチェック

RLU8 は，お互いのモジュール間のメモリ情報に違いがないかどうかをチェックします。ただし，有効となるメモリ情報は，マイクロコントローラ名の決定しているものだけに限られます。汎用モジュールのメモリ情報は無視されます。

メモリ情報に違いがあった場合の動作は，/PDIF オプションの有無により以下のように異なります。

（1）/PDIF オプションが指定されていなければ，フェイタルエラーで終了します。

(2) /PDIF オプションが指定されている場合には，ワーニングを表示し，メモリ情報の論理 OR をとります。

7.6.1.5 ROMWINDOW 属性のマッチングチェック

RLU8 は，お互いのモジュール間の ROMWINDOW 属性がリンク可能なものかどうかをチェックします。

以下に，モジュールの組み合わせとリンクの可否を示します。以下の表において，デフォルトは，アセンブリソースで ROMWINDOW 擬似命令も NOROMWIN 擬似命令も使用していないものを示します。ROMWINDOW は，アセンブリソースで ROMWINDOW 擬似命令を指定して ROM ウィンドウ領域が確定しているものを示します。NOROMWIN はアセンブリソースで NOROMWIN 擬似命令を指定しているものを示します。

モジュール 1	モジュール 2	リンクの可否	備考
デフォルト	デフォルト	リンク可	/ROMWIN オプションをしない場合，ROM ウィンドウ領域が未確定になるため，エラーになります。

モジュール 1	モジュール 2	リンクの可否	備考
デフォルト	ROMWINDOW	リンク可	最終的な ROM ウィンドウ領域は，モジュール 2 で指定されている領域になります。
ROMWINDOW	ROMWINDOW	リンク可	お互いの ROM ウィンドウ領域が一致しない場合は，エラーになります。
NOROMWIN	デフォルト	リンク不可	エラーになります。
NOROMWIN	ROMWIN	リンク不可	エラーになります。
NOROMWIN	NOROMWIN	リンク可	ROM ウィンドウ領域はないものとして扱われます。

7.6.2 グローバルシンボルの対応付け

RLU8 は，*object_files* フィールドで指定されたファイルからオブジェクトモジュールを指定された順に読み込み，イクスターナルシンボルを同じ名前のパブリックシンボルまたは共有シンボルによって解決します。すべてのモジュールを読み込んでも，なお解決されないイクスターナルシンボルが残っている場合，RLU8 は *libraries* フィールドに指定されているライブラリファイルをサーチします。ライブラリファイルが見つかると，RLU8 は，そのライブラリ中に，未解決なイクスターナルシンボルと同じ名前のパブリックシンボルが定義されているモジュールがあるかどうか調べます。もしそのようなモジュールがあれば，ライブラリから取り出してリンクすべきモジュールに加えます。この処理はすべてのイクスターナルシンボルが解決されるまで繰り返されます。

イクスターナルシンボルを他のモジュールのパブリックシンボル，共有シンボルによって解決できるかどうかは，それらが持つユーセージタイプと物理セグメント属性によって判定されます。それらのユーセージタイプが同じで，かつ物理セグメント属性が同じであれば，イクスターナルシンボルは解決されます。

同じ名前の共有シンボルとパブリックシンボルが現れると，その名前は共有シンボルではなく，パブリックシンボルとして扱われます。それらが同じセグメントタイプを持っていることが対応付けの条件になります。パブリックシンボルと共有シンボルが対応付けられた場合，共有シンボルの情報はなくなりますので，それがメモリ空間に割り付けられることはありません。

7.6.3 セグメントの結合

複数のモジュールをリンクするときに，同じ名前を持つ複数のセグメント（それぞれを“パーシャルセグメント”と呼びます）が現れてもかまいません。RLU8 は，パーシャルセグメントを 1 つに結合しようとします。セグメントの結合は，2 つのセグメントの結合の繰り返しによって行われます。RLU8 は，パーシャルセグメントを結合するとき，それらの持ついくつかの属性を比較し，結合可能かどうかを判断します。

セグメントが結合されるには，次の条件が満たされなければなりません。

(1) 同じセグメントタイプを持つ。

(2) 特殊領域属性 DYNAMIC を持つものは，すべてのパーシャルセグメントで持たねばならない。

(3) 2 つのセグメントサイズの合計が，割り付け対象領域のメモリサイズを超えてはならない。

結合の結果，作られるセグメントの属性は次のようになります。

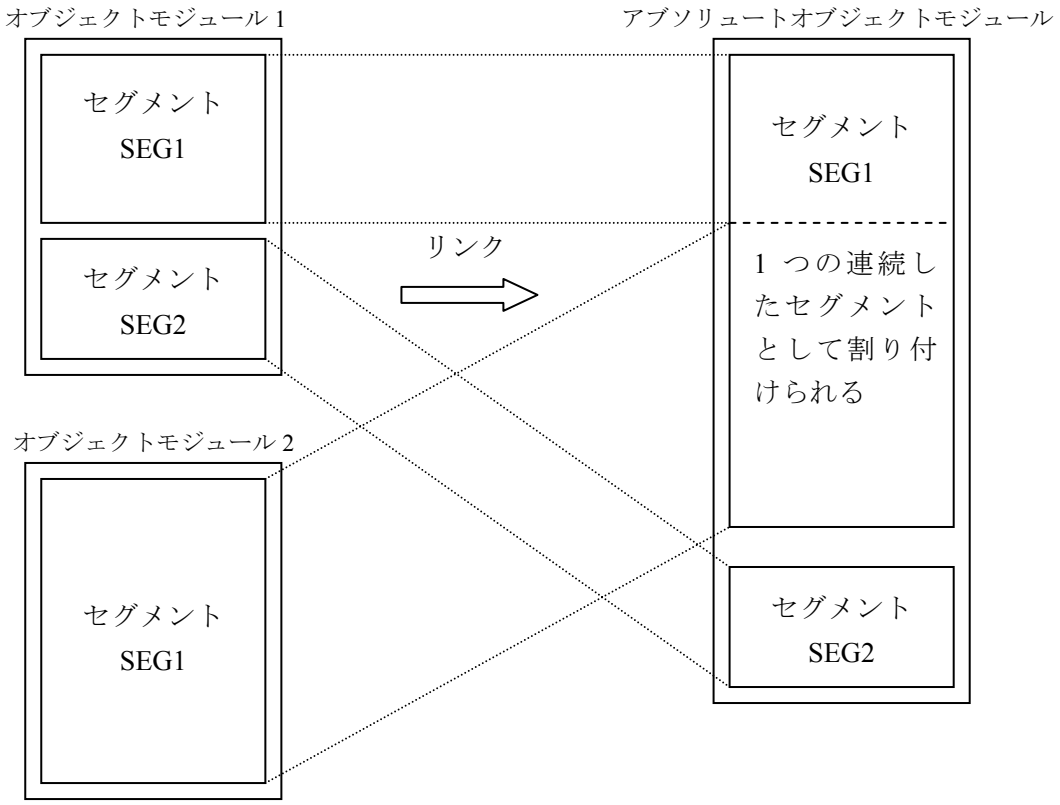
(1) セグメントタイプは，結合前のタイプを継承する。

(2) 境界値属性は，大きい方を継承する。

(3) サイズは，２つのセグメントの合計サイズになる。

パーシャルセグメントは，モジュール中に現われた順に後ろに連続して置かれます。すなわち，1 つのセグメントに結合された各パーシャルセグメントは連続してメモリに置かれることになります。このため，各パーシャルセグメントは，ソースプログラムで指定された境界値に置かれなくなる場合もあります。たとえば，境界値 2 が指定されている 2 つのパーシャルセグメントを結合した場合を考えてみます。結合後のセグメントは境界値 2 を持つので，セグメントの先頭は偶数アドレスに割り付けられます。このとき，最初のパーシャルセグメントのサイズが偶数バイトであれば，2 つのセグメントは偶数アドレスに割り付けられます。しかし，最初のパーシャルセグメントのサイズが奇数バイトであれば，後のセグメントは奇数アドレスから開始することになります。このセグメント内のアドレスに対して，ワード単位のアドレッシングをしている場合には注意が必要となります。

セグメントの結合のイメージは，次のようになります。

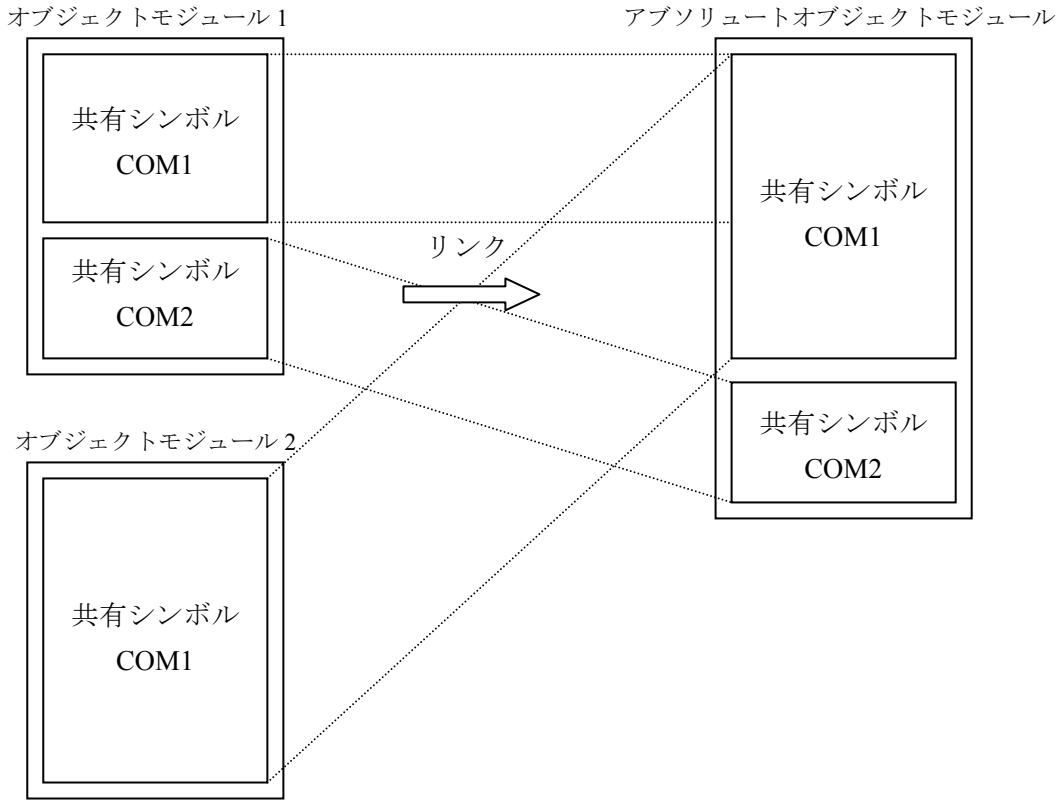


7.6.4 共有シンボルの結合

セグメントと同様，同じ名前を持つ共有シンボル同士も 1 つに結合されます。結合の過程はセグメントの場合と同じです。ただし，セグメントの結合ではパーシャルセグメントが順に後ろに連続して置かれるのに対し，共有シンボルは，それらの開始アドレスでオーバーラップさせて結合します。その結果作られた共有シンボルは，結合されたシンボルのうち最も大きなものと同じサイズを持つことになります。

共有シンボルの結合の条件は，セグメントの結合の場合と同じです。

共有シンボルの結合のイメージは次のようになります。



7.6.5 セグメントの参照関係のチェック

/LA オプションが指定されていない場合，RLU8 は CODE セグメントおよび TABLE セグメントの参照関係をチェックし，参照されない CODE セグメントまたは TABLE セグメントを割り付け対象から除外します。

7.6.6 セグメントの割り付け

セグメント，共有シンボル，それぞれの結合が終了すると，RLU8 は，メモリ空間にセグメント，共有シンボルおよび擬似セグメントを割り付けます。セグメントと共有シンボルが割り付けられる領域については，このあとの「7.6.6.1 割り付け空間と領域」で説明します。擬似セグメントについては，このあとの「7.6.6.2 擬似セグメント」で説明します。

割り付け可能なメモリの範囲は，DCLファイル，または/ROM，/RAM，/NVRAM，/ROMWINオプションによって指定されます。/ROM，/RAM，/NVRAM，/ROMWINオプションの詳細については，それぞれの「7.5.3 各オプションの機能」を参照してください。

RLU8 は，優先度の高いセグメントから順にメモリに割り付けていきます。同じ優先度のセグメントは任意に処理されます。また，同じ優先度を持つセグメントと共有シンボルでは，セグメントの方を先に割り付けます。優先度については，このあとの「7.6.6.3 割り付けの優先度」で説明します。

RLU8 は，セグメントと共有シンボルを割り付けるための空き領域を，アドレス 0 から上位メモリに向かってサーチします。ただし，DATA と BIT タイプのリロケータブルセグメントについては，物理セグメント#0 の場合のみ，アドレス 0FFFFH から下位メモリに向かってサーチします。

そして，このサーチ処理の中で最初に見つかった空き領域に割り付けます。割り付けられる空間が見つからなかった場合，エラーメッセージを表示しリンク処理を中止します。

セグメントは，プログラムで指定された境界値に割り付けられます。共有シンボルの場合，境界値はセグメントタイプとサイズに依存します。DATA，NVDATA，TABLE タイプの共有シンボルは，そのサイズが 1 バイトであればバイト境界に割り付けられ，2 バイト以上であればワード境界に割り付けられます。BIT, NVBIT タイプの共有シンボルは，サイズに関係なくビット境界に割り付けられます。境界値属性を除いて，セグメントと共有シンボルの扱いは同じです。

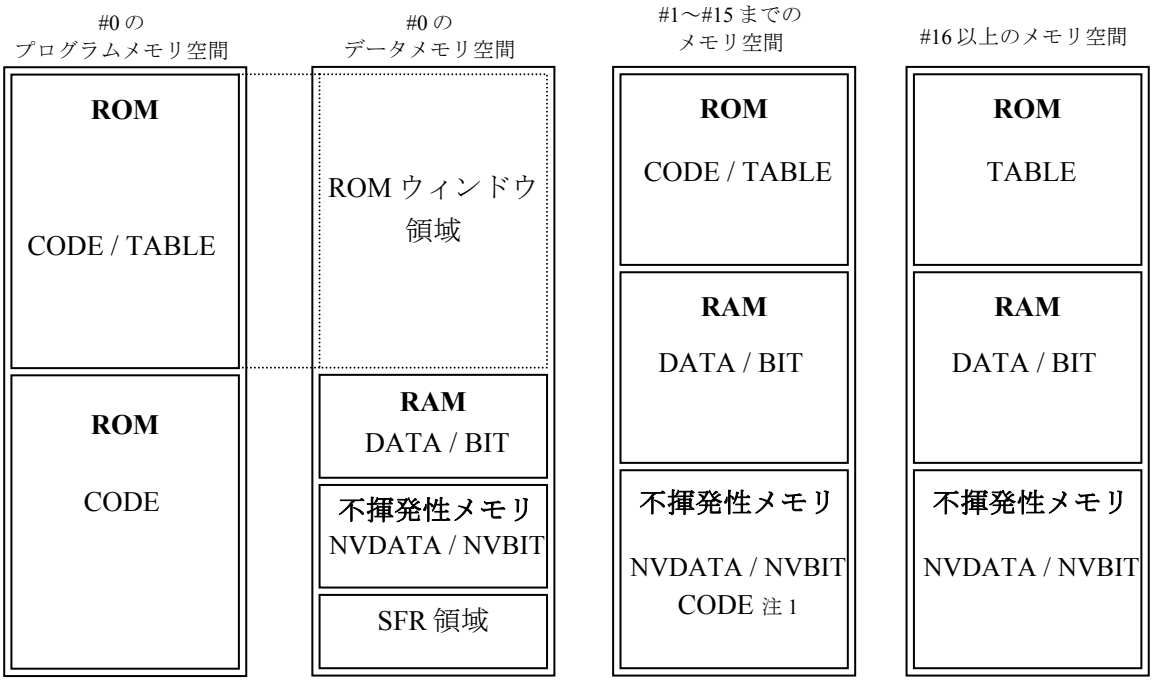
セグメントの割り付けにおいて，物理セグメントが複数存在する場合，各セグメントは物理セグメントの境界をまたがないものとします。

7.6.6.1 割り付け空間と領域

セグメントと共有シンボルが割り付けられる空間と領域は，セグメントタイプ，実装されるメモリの種類，および特殊領域属性によって決まります。次にその対応を説明します。

セグメントタイプと実装されるメモリの種類による割り付け空間

セグメントタイプと実装されるメモリの種類によって，割り付けられる空間が決まります。セグメントタイプと割り付け空間の対応は，次のとおりです。



注 1 リロケータブル CODE セグメントの場合，不揮発性メモリ上に割り付けられるのは，セグメントシンボル定義時に特殊領域属性として NVRAM を指定したものに限られます。

7.6.6.2 擬似セグメント

メモリ空間上には，プログラマによって定義されたセグメントが割り付けられてはならない特別な領域がいくつかあります。RLU8 は，自らセグメントを作り出し，あらかじめその領域に割り付けることで，プログラマによって定義されたセグメントと共有シンボルがそこに割り付けられないようにします。この，RLU8 が作り出すセグメントを“擬似セグメント”と呼びます。

擬似セグメントは，プログラマが定義するものではありませんのでご注意ください。

次に，擬似セグメントが割り付けられる領域を示します。

(1) データメモリ空間の SFR 領域

(2) データメモリ空間の ROM ウィンドウ領域

(3) オーバーレイ領域（オーバーレイに関しては「10 オーバーレイ機能」を参照してください）

7.6.6.3 割り付けの優先度

RLU8 は，アブソリュートアドレスを持つセグメントに最も高い優先度を与えます。続いて，オプションの指定のあるもの，より条件の厳しいものから順に高い優先度を与えます。以下にセグメントの優先度を示します。

優先度	セグメントの条件
1	CSEG, DSEG, BSEG, NVSEG, NVBSEG, TSEG 擬似命令で定義された，すべてのアブソリュートセグメント，および RLU8 が生成する擬似セグメント
2	/CODE，/DATA，/BIT，/NVDATA，/NVBIT，/TABLE オプションによって，アブソリュートアドレスを指定しているすべてのセグメント。例えば，次のような指定がされたものです。 `/CODE( CodeSegment-1:2000H )`
3	/TABLE オプションでアブソリュートアドレスを指定していないもので，物理セグメント#0 に割り付けることが指定されている，リロケータブル TABLE セグメント。例えば，次のような指定がされたものです。 `/TABLE( 0:1000H TableSegment0 )`
4	物理セグメント#0 に割り付けることが指定されている，リロケータブル TABLE セグメント または，/CP オプションが指定されておらず，ROM が物理セグメント#0 のみの場合の，物理セグメント指定なしのリロケータブル TABLE セグメント
5	/NVDATA または/NVBIT オプションで，アブソリュートアドレスを指定していないリロケータブル NVDATA, NVBIT セグメント
6	/CODE オプションで，アブソリュートアドレスを指定していないリロケータブル CODE セグメント
7	/TABLE オプションでアブソリュートアドレスを指定していないもので，物理セグメント#0 以外に割り付けることが指定されている，リロケータブル TABLE セグメント。例えば，次のような指定がされたものです。 例) `/TABLE( 1:1000H TableSegment1 )`
8	/DATA, /BIT オプションで，アブソリュートアドレスを指定していないリロケータブル DATA, BIT セグメント
9	物理セグメント指定付きのリロケータブル NVDATA, NVBIT セグメント
10	物理セグメント指定付きのリロケータブル CODE セグメント
11	物理セグメント指定付き(#0 以外)のリロケータブル TABLE セグメント
12	スタックセグメント，ダイナミックセグメントを除く，物理セグメント指定付きの DATA, BIT セグメント

優先度	セグメントの条件
13	スタックセグメント
14	上記以外の物理セグメント指定なしの NVDATA, BIT セグメント
15	上記以外の物理セグメント指定なしの CODE セグメント
16	/CP オプションが指定されていない場合： ROM が物理セグメント#1 以上に存在する場合の，物理セグメント指定なしのリロケータブル TABLE セグメント /CP オプションが指定されている場合： すべての物理セグメント指定なしのリロケータブル TABLE セグメント
17	スタックセグメント，ダイナミックセグメントを除く，物理セグメント指定なしの DATA, BIT セグメント
18	ダイナミックセグメント

7.6.7 フィックスアップ処理

すべてのシンボルにアブソリュート値が与えられると，次に RLU8 はアセンブル時に未解決だったオペランドを解決（フィックスアップ）します。

通常，プログラムは，アセンブル時にすべての部分が絶対値として与えられるわけではありません。リロケータブルシンボルを含むオペランドは，アセンブル時に解決できず未解決のままとなります。RASU8 は，一時的にその部分に値 0 を割り当てます。そして，そのオペランドをフィックスアップするための情報をオブジェクトファイルに出力します。RLU8 は，この情報に基づいてそのオペランドをフィックスアップします。

フィックスアップは，次の手順で行われます。

(1) フィックスアップ情報から絶対値を算出します。

(2) 必要に応じて，その値のチェックを行います。

(3) すべてのチェックが終了すると，RLU8 は，RASU8 によって一時的に 0 が置かれた部分を新しい値で書き換えます。

以上の手順は 1 つのフィックスアップに対するものです。RLU8 はすべてのフィックスアップを終了するまで，これらの手順を繰り返し行います。

7.7 マップファイル

ここでは，RLU8 が作成するマップファイルの書式と読み方を説明します。

次に示すのは，マップファイルの最初に表示される部分です。RLU8 の起動時に与えられたオプションの内容などが表示されます。

```
(1)  RLU8 Object Linker, Ver.1.51.2 Linkage Information
     [Mon Aug 27 16:21:15 2012]


                          ------------------
                           Control Synopsis
                          ------------------

(2)  I/O controls:    D NSD  S NA

(3)  Locating controls:

          Type     Address             Name
          ----     -------             ----
          CODE     At    01:1000       CSEG02
          DATA     After 07:8000       DSEG02

(4)  Other controls: STACK( 0100H(256) )
```

(1) マップファイルの先頭にはヘッダが付きます。RLU8 のバージョン，マップファイルの作成日時が出力されます。

(2) /D，/ND，/SD，/NSD，/S，/NS，/A，および/NA オプションのうち，RLU8 の起動時に与えられたものが表示されます。

(3) RLU8 の起動時に与えられた/CODE，/TABLE ，/DATA ，/BIT，/NVDATA，および/NVBIT オプションの内容が表示されます。Address フィールドに表示される“After *address*”は，*address* で示されるアドレスより上位に割り付けるように指定されたことを表します。“At *address*”は，*address* で示されるアブソリュートアドレスが与えられたことを表します。アドレスは，16 進数で表示されます。

(4) RLU8 の起動時に与えられた(2)，(3)以外のオプションの内容が表示されます。

続いて，処理したモジュールの情報が次のように表示されます。

```
                         ------------------------
                          Object Module Synopsis
                         ------------------------
                                                                Creator
(5)  Module Name           File Name                            CCU8     RASU8
     -----------           ---------                            -----    -----
     sample1               sample1.obj                          3.30     1.61
     sample2               sample2.obj                          -.--     1.61

(6)
     Number of Modules: 2

(7)  Number of Symbols:
      +-------------------------------------------------------------------------+
      |          | CODE | DATA | BIT  |NVDATA|NVBIT |TABLE |NUMBER| TBIT || total |
      |----------+------+------+------+------+------+------+------+------++-------|
      | SEGMENT  |     3|     3|     0|     2|     0|     2|      |      ||     10|
      |----------+------+------+------+------+------+------+------+------++-------|
      | COMMUNAL |     0|     1|     0|     0|     0|     0|      |      ||      1|
      |----------+------+------+------+------+------+------+------+------++-------|
      | PUBLIC   |     1|     1|     0|     1|     0|     1|     0|     0||      4|
      +-------------------------------------------------------------------------+
(8)
(9)  Target:  ML610001
     Model:   LARGE

(10) Memory Map - Program memory space #0:
           Type     Start      Stop
          ----------------------------
           ROM      00:0000    00:7FFF
           NVRAM    00:8000    00:FFFF

     Memory Map - Data memory space #0:
           Type     Start      Stop
          ----------------------------
           RAM      00:0000    00:7FFF
           NVRAM    00:8000    00:9FFF
           RAM      00:A000    00:FFFF

     Memory Map - Memory space above #1:
           Type     Start      Stop
          ----------------------------
           ROM      01:0000    06:FFFF
           RAM      07:0000    09:FFFF
           NVRAM    0A:0000    0C:FFFF
           ROM      0D:0000    0F:7FFF
```

(5) RLU8 が処理したモジュールの名前，そのモジュールを格納しているファイル名，そのモジュールを作成した CCU8 と RASU8 のバージョン番号が表示されます。モジュールに CCU8 のバージョン情報が含まれていない場合は，CCU8 のフィールドにはハイフン”-.--”が表示されます。

(6) 処理したモジュールの数です。

(7) 処理した全モジュール中に含まれるセグメント，共有シンボル，およびパブリックシンボルの数がセグメントタイプ別に表示されます。結合されたセグメント，共有シンボルは 1 つとしてカウントされます。

(8) プログラムの対象となるマイクロコントローラの名前が表示されます。

(9) プログラムで選択されたメモリモデルが表示されます。“SMALL”，“LARGE”のいずれかが表示されます。

(10) 物理セグメント#0 のプログラムメモリ空間，データメモリ空間，および物理セグメント#1 以上のメモリ空間に実装されたメモリの種類と範囲が表示されます。Start，Stop フィールドのアドレスは 16 進数で表示されます。

続いて，セグメント，共有シンボルの割り付け状態がメモリ空間ごとに表示されます。

```
                          ------------------
                           Segment Synopsis
                          ------------------

(11)  Link Map - Program memory space #0 ( ROMWINDOW: 0000 - 3FFF )

        Type     Start      Stop       Size              Name
      -------------------------------------------------------
        S CODE   00:0000    00:0003    0004(4)           (absolute)
        S TABLE  00:0004    00:0041    003E(62)          TSEG01
        S CODE   00:0042    00:0083    0042(66)          CSEG01


(12)  Link Map - Data memory space #0

        Type     Start      Stop       Size              Name
      -------------------------------------------------------
        Q ROMWIN 00:0000    00:3FFF    4000(16384)       (ROMWIN)
>GAP<           00:4000.0  00:7FFF.7  4000.0(16384.0)   (RAM)
        S NVDATA 00:8000    00:803F    0040(64)          NVDSEG01
>GAP<           00:8040.0  00:9FFF.7  1FC0.0(8128.0)    (NVRAM)
>GAP<           00:A000.0  00:FDBF.7  5DC0.0(24000.0)   (RAM)
        S DATA   00:FDC0    00:FEBF    0100(256)         $STACK
        S DATA   00:FEC0    00:FEFF    0040(64)          DSEG01
        Q SFR    00:FF00    00:FFFF    0100(256)         (SFR)


(13)  Link Map - Memory space above #1

        Type     Start      Stop       Size              Name
      -------------------------------------------------------
        S TABLE  01:0000    01:004F    0050(80)          TSEG02
>GAP<           01:0050.0  01:0FFF.7  0FB0.0(4016.0)    (ROM)
        S CODE   01:1000    01:103F    0040(64)          CSEG02
 ---
        C DATA   07:0000    07:0003    0004(4)           DCOM01
>GAP<           07:0004.0  07:7FFF.7  7FFC.0(32764.0)   (RAM)
        S DATA   07:8000    07:803F    0040(64)          DSEG02
 ---
        S NVDATA 0A:0000    0A:003F    0040(64)          NVDSEG02
```

(14)

```
Total size (CODE  ) = 00086   (134)
Total size (DATA  ) = 00184   (388)
Total size (BIT   ) = 00000.0 (0.0)
Total size (NVDATA) = 00080   (128)
Total size (NVBIT ) = 00000.0 (0.0)
Total size (TABLE ) = 0008E   (142)
```

(11) 物理セグメント#0 のプログラムメモリ空間への割り付け状態が表示されます。プログラムメモリ空間に ROM ウィンドウ領域が定義されている場合，最初に ROM ウィンドウ領域の範囲が次のように表示されます。

```
( ROMWINDOW: 0000 - 3FFF )
```

ROM ウィンドウ領域が定義されていない場合は，次のように表示されます。

```
( ROMWINDOW: Not exist )
```

Type フィールドには，まず“S”，“C”，または“Q”のいずれか一文字が表示されます。それぞれの文字は順に，セグメント，共有シンボル，疑似セグメントを意味します。続いて，Type フィールドには，セグメントタイプ，または，以下の表に示される疑似セグメントのタイプが表示されます。

擬似セグメントの場合の Type，Name フィールドの表示

疑似セグメント	Type フィールドの表示	Name フィールドの表示
SFR	SFR	(SFR)
ROM WINDOW	ROMWIN	(ROMWIN)
オーバーレイ領域	OVRLAY	(*area_name*) *area_name* には，/OVERLAY オプションで指定したオーバーレイ領域の名前が表示されます。

Start, Stop フィールドには，そのセグメントが占有する領域の開始アドレスと終了アドレスが 16 進数で表示されます。

Size フィールドには，そのセグメントのサイズが 16 進数と 10 進数で表示されます。

Name フィールドには名前が表示されます。アブソリュートセグメントの場合は，“(absolute)”と表示されます。オーバーレイオプションで指定されたセグメントの場合，名前の右側に“*”が付加されます。疑似セグメントの場合は，上の表に示される名前が表示されます。

Type フィールドの左側には次のメッセージが表示される場合があります。

表示	説明
>GAP<	メモリ空間に空き領域があることを表します。 Name フィールドには空き領域のメモリの種類を示す“(ROM)”，“(RAM)”，“(NVRAM)”のいずれかが表示されます。NVRAM は不揮発性メモリを意味します。
OVL	メモリ上でセグメントがオーバーラップしていることを表します。
OUT	/DATA，/CODE オプションなどによりアドレスが指定されたセグメントや，アブソリュートセグメントが，割り付け可能領域外に割り付けられてしまっていることを表します。
---	そこで物理セグメントが変わったことを表します。

セグメントと共有シンボルの合計サイズが 16 進数，10 進数で表示されます。疑似セグメントのサイズは含まれません。

(12) 物理セグメント#0 のデータメモリ空間への割り付け状態が表示されます。表示形式は，プログラムメモリ空間の場合と同じです。

(13) 物理セグメント#1 以上のメモリ空間への割り付け状態が表示されます。表示形式は，物理セグメント#0 のメモリ空間の場合と同じです。

(14) それぞれのセグメントタイプごとに，セグメントと共有シンボルの合計サイズが 16 進数，10 進数で表示されます。擬似セグメントのサイズは含まれていません。

何らかのエラーによって，どこにも割り付けられなかったセグメント，または共有シンボルがあった場合，それらは次のように表示されます。

```
Ignored 2 segments or communal symbols:
      S CODE                        3393(13203)       TEST_C_SEG
      S DATA                        2415(9237)        TEST_D_SEG
```

参照されずメモリへ割り付けられなかった関数・テーブルのセグメントは，次のようにモジュール別に表示されます。

```
Not Linked Segments:
      Module Name         Type       Size              Segment Name
     ----------------------------------------------------------------
      file1
                          CODE       0100(256)         $$func1$file1
                          CODE       0120(288)         $$func3$file1
     ----------------------------------------------------------------
      file2
                          CODE       0200(512)         $$func10$file2
     ----------------------------------------------------------------
```

上の例で示しているセグメント名は，コンパイラがデフォルトで出力するセグメントです。ユーザが定義したセグメントの場合は，その名前が出力されます。コンパイラがデフォルトで出力するセグメントの場合，$$と$の間に挟まれた名前には関数名・テーブル名が含まれています。この情報により，どの関数・テーブルがリンクされなかったかを知ることができます。

以上は，標準的に出力される項目です。RLU8 は，これらの他に，オプションによってさらにいくつかの情報をマップファイルに追加出力します。この追加される情報について以下に説明します。

/D，/SD オプションが指定されると，アブソリュートオブジェクトファイルに出力したデバッグ情報からシンボル情報を取り出し，次のように出力します。シンボルは，モジュール単位にまとめられます。

```
       -----------------------
        Symbol Table Synopsis
       -----------------------

Module         Value     Type         Symbol
------         -----     ----------   ------
sample1
              0000001C   Loc NUMBER   _$$WINVAL
               00:3FFF   Loc TABLE    _$$ROMWINEND
               00:0000   Loc TABLE    _$$ROMWINSTART
               00:8000   Pub NVDATA   NVDATA1
               00:FEC0   Pub DATA     BUF1
               00:0004   Pub TABLE    STRING
               00:0042   Pub CODE     _START

Module         Value     Type         Symbol
------         -----     ----------   ------
sample2
              0000001C   Loc NUMBER   _$$WINVAL
               00:3FFF   Loc TABLE    _$$ROMWINEND
               00:0000   Loc TABLE    _$$ROMWINSTART
```

Type フィールドの Pub, Loc はそれぞれパブリックシンボル，ローカルシンボルを表しています。

/S オプションが指定されると，プログラム中で使用されているパブリックシンボルおよびの共有シンボルの情報を次のように，アルファベット順に出力します。

```
Public Symbols Reference

Symbol      Value       Type     Module
------      -----       ----     ------
BUF1        00:FEC0     DATA     sample1
NVDATA1     00:8000     NVDATA   sample1
STRING      00:0004     TABLE    sample1
_$$SP       00:FEC0     DATA     sample1
_START      00:0042     CODE     sample1
```

ワーニング以外のエラーがなかった場合，マップファイルの最後に次の一行が出力されます。

```
End of mapfile.
```

7.8 エラーメッセージ

RLU8 は，処理中にエラーを検出するとメッセージを標準出力に出力します。RLU8 が出力するエラーメッセージには，コマンドラインエラー，フェイタルエラー，エラー，ワーニングがあります。

コマンドラインエラー

コマンドラインエラーは，RLU8 のコマンドラインに無効な入力がなされ，処理の続行が不可能な場合に表示されます。このエラーが起きると，RLU8 は直ちに処理を中止します。

フェイタルエラー

フェイタルエラーは，明らかな間違いが見つかって，RLU8 の処理の続行が不可能な場合に表示されます。このエラーが表示されると，RLU8 は直ちに処理を中止します。

エラー

エラーは，明らかな間違いが見つかって，アブソリュートオブジェクトファイルの作成が不可能な場合に表示されます。

エラーが検出された場合，エラーメッセージを表示した後，RLU8 は処理を中止します。

ワーニング

ワーニングは，プログラム中に致命的ではないが疑わしいものがあったときに表示されます。この場合，RLU8 は処理を継続し，アブソリュートオブジェクトファイルを作成します。

7.8.1 エラーメッセージの形式

エラーメッセージは，次の形式で表示されます。

構文

error_type *error_code* : *message*

error_type にはエラーの種類を示す，次のいずれかが表示されます。

error_type	エラーの種類
Command line error	コマンドラインエラーであることを示します。
Fatal error	フェイタルエラーであることを示します。
Error	エラーであることを示します。
Warning	ワーニングであることを示します。

*error_code*にはエラー番号が，*message*にはエラーメッセージが表示されます。エラー番号とエラーメッセージの一覧は，「7.9 エラーメッセージ一覧」を参照してください。

RLU8 は，エラーメッセージの他に，エラーの原因を想定するのに必要なファイル名，シンボ

ル名などの情報をできる限り出力します。特に，未解決データのフィックスアップ時にエラーを検出した場合は，メッセージに続きエラーの発生した位置を次のように表示します。

offset / *segment_name* / *module_name*

これは，オブジェクトモジュール *module_name* に定義されているセグメント *segment_name* 内のオフセットアドレス *offset* でエラーが起こったことを意味します。

このオフセットアドレスは，RASU8 が作成するプリントファイルの，アセンブルリストのロケーションフィールドの値に対応しています。

次に示すようなプリントファイル TEST.PRN があったとします。このプリントファイルは，RASU8 によって作成されたものです。

```
## Loc. Object                    Line   Source Statements

                                     1       TYPE (M610001)
                                     2
                                     3       MODEL   LARGE, FAR
                                     4       ROMWINDOW   0, 3FFFH
                                     5
                                     6   EXTRN   NONE:ODD_ADDR EVEN_ADDR
                                     7   PROG    SEGMENT CODE
                                     8
  --------------------------         9       RSEG    PROG
??:0000 00 E0                       10       MOV     ER0,    #0
??:0002 00'E3 13-90 00-00'          11       ST      ER0,    ODD_ADDR
??:0008 00'E3 13-90 00-00'          12       ST      ER0,    EVEN_ADDR
```

このプログラムのオブジェクトモジュールをリンクしたとき，次のメッセージが表示されたとします。

```
Warning W006: Cannot access high byte, 0006/PROG/TEST
```

これは，モジュール TEST に定義されているセグメント PROGC のオフセットアドレス 0006 の未解決データを，フィックスアップしようとしたときにワーニングが検出されたことを示しています。プログラマは，このメッセージから，11 行目の命令のオペランドに記述されているシンボル ODD_ADDR に問題があることを知ることができます。

7.9 エラーメッセージ一覧

RLU8 が表示するエラーメッセージの一覧を以下に示します。エラーメッセージの左には，エラーメッセージの番号が記載されています。エラーメッセージの下には，その内容が説明されています。

7.9.1 コマンドラインエラーメッセージ

C001 Command line syntax error

コマンドライン指定が不正です。

C002 Duplicate segment name '*segment_name*'

/COMB，/OVERLAY オプションにおいて，セグメントを二重に指定しています。

/COMB オプション全体でセグメントを二重に指定することはできません。

/OVERLAY オプション全体でセグメントを二重に指定することはできません。

C003 Duplicate overlay area name

/OVERLAY オプションにおいて，オーバーレイ領域の名前が重複しています。オーバーレイ領域の名前が重複しないように変更してください。

C004 Invalid overlay area

/OVERLAY オプションにおいて，オーバーレイ領域の範囲指定が不正です。

C005 Invalid numerical argument

指定された値が，指定可能な範囲(1～65536)を超えています。

C006 Bad constant

オプションの引数として与えられた数値が正しくありません。これは，/CODE(SEG1-0F00)のように，16 進定数の最後に“H”が付いていない場合に表示されます。

C007 Unrecognized option name

指定されたオプションは RLU8 ではサポートしていません。

C008 Missing object file name

オブジェクトファイル名が 1 つも指定されませんでした。少なくとも 1 つの入力オブジェクトファイルを指定して，RLU8 を呼び出してください。

C009 Invalid specified area

指定された領域の範囲指定が不正です。

C010 Invalid ROM window

ROM ウィンドウ領域の範囲指定が不正です。

C011 Command line buffer overflow

入力された文字数がコマンド解析用バッファの許容範囲を超えました。

7.9.2 フェイタルエラーメッセージ

F001 Insufficient memory

RLU8 を実行するためのメモリが不足しています。

F002 Cannot open file

ファイルをオープンすることができません。

F003 Cannot close file

ファイルを閉じることができません。

F005 Versions not compatible

RASU8 のバージョン番号が不正です。

F006 Bad module

オブジェクトモジュールが壊れている可能性があります。再度，そのモジュールをアセンブルしてからリンクしてください。

F007 Record length too long

オブジェクトモジュールを構成するレコードの長さが許容範囲を超えています。オブジェクトモジュールが壊れている可能性があります。再度，そのモジュールをアセンブルしてからリンクしてください。

F008 Checksum error

チェックサムエラーが発見されました。

オブジェクトモジュールが壊れている可能性があります。そのモジュールをアセンブルしてからリンクしてください。

F009 Invalid Core ID

異なる CPU コアのオブジェクトモジュールをリンクしようとしています。すべてのアセンブリソースの中で TYPE 擬似命令を用いて同じ DCL ファイルを指定し，アセンブルしなおしてからリンクしてください。

F010 Inconsistent machine name

リンクしようとしたオブジェクトモジュールのマイクロコントローラ名が一致していません。すべてのアセンブリソースの中で TYPE 擬似命令を用いて同じ DCL ファイルを指

定し，アセンブルしなおしてからリンクしてください。

F011　Inconsistent memory model

リンクしようとしたオブジェクトモジュールのメモリモデルが一致していません。プログラムで同じメモリモデルを指定し，再度アセンブルしてからリンクしてください。

F014　Invalid family ID

オブジェクトモジュールが nX-U8 用のものではありません。

F015　Inconsistent memory values

リンクしようとしたオブジェクトモジュールのメモリ情報に一致しない部分があります。再度アセンブルしてから，リンクしてください。

RASU8 の/B オプションでメモリを追加した場合に，リンクしようとしているモジュールの間にメモリ情報の不一致があるとこのエラーが発生します。

そのときは，/PDIF オプションを指定すると，このエラーを回避できる場合があります。

F016　Illegal translator ID

オブジェクトモジュールが，RASU8 によって作成されたものではありません。RLU8 は，RASU8 によって作成されたオブジェクトモジュールだけを入力として扱います。

F017　Illegal object module format

オブジェクトモジュールのフォーマットが nX-U8 用のものではありません。

F018　Illegal library type

指定したライブラリファイルのフォーマットが nX-U8 用のものではありません。

F019　File seek error

ファイルをシークするときにエラーが発生しました。

F020　Cannot write, disk full!?

書き込みエラーが発生しました。ディスクの容量不足が考えられます。

F021　Not a library file

指定されたファイルはライブラリファイルではありません。

F022　Specified module not found

object_files フィールドで，“library(*module_name* …)”として指定されたモジュールがライブラリファイル中に見つかりませんでした。正しいモジュール名を確認してから，再度リンクしてください。

F023　All machine names are STANDARD

リンクしたオブジェクトモジュールはすべて汎用モジュールでした。

少なくとも１つは専用モジュールを指定しなければなりません。専用モジュールとは，特定のマイクロコントローラの名前が指定されている DCL ファイルを用いて作られたオブジェクトモジュールのことです。

F024 File used in conflicting contexts

入力ファイルと出力ファイルに同じ名前が指定されています。出力ファイルに別の名前を指定してください。

F025 No ROM window specification

ROM ウィンドウ領域が指定されていません。

いずれかのモジュールで ROM ウィンドウ領域を明示的に指定するか，/ROMWIN オプションを使用して ROM ウィンドウ領域を指定してください。

F026 ROM window mismatch

ROM ウィンドウ領域がモジュール間で一致していません。すべてのモジュール間で ROM ウィンドウ領域が一致するようにプログラムを変更して，アセンブルしてからリンクしてください。

F027 Inconsistent /ROMWIN values

/ROMWIN オプションの指定範囲が不正です。

F028 NOROMWIN is specified

NOROMWIN が指定されているモジュールに，/ROMWIN オプションを指定して ROM ウィンドウ領域を指定しようとしています。

F029 ROM WINDOW attribute mismatch

リンクしようとしているモジュールの ROMWINDOW 属性が一致していません。

F030 Inconsistent memory values (VECTOR)

ベクタ領域の情報がモジュール間で一致していません。異なるマイクロコントローラ用のモジュールをリンクしようとしている可能性があります。同じ DCL ファイルを TYPE 擬似命令で指定して，アセンブルしなおしてからリンクしてください。

F031 Inconsistent memory values (ROMWIN)

ROM ウィンドウ領域の情報がモジュール間で一致していません。異なるマイクロコントローラ用のモジュールをリンクしようとしている可能性があります。同じ DCL ファイルを TYPE 擬似命令で指定して，アセンブルしなおしてからリンクしてください。

F032 Inconsistent memory values (SFR)

SFR 領域の情報がモジュール間で一致していません。異なるマイクロコントローラ用のモジュールをリンクしようとしている可能性があります。同じ DCL ファイルを TYPE 擬似命令で指定して，アセンブルしなおしてからリンクしてください。

F033 Inconsistent memory values (INTERNAL RAM)

内部 RAM 領域の情報がモジュール間で一致していません。異なるマイクロコントローラ用のモジュールをリンクしようとしている可能性があります。同じ DCL ファイルを TYPE 擬似命令で指定して，アセンブルしなおしてからリンクしてください。

F034 Inconsistent max physical segment for Data memory '*module_name*'

リンクしようとしているモジュールのデータメモリ空間のセグメント上限値が異なっていた場合に表示されます。

このエラーの原因として、コンパイル時に/nofar オプションを指定したオブジェクトファイルと、/nofar オプションに対応していないスタートアップファイルをリンクしようとしていることが想定されます。コンパイル時に/nofar オプションを指定したファイルとリンクをする場合には、/nofar オプション対応のスタートアップファイル(NOFAR 擬似命令が記述されているファイル)を指定してください。

F035 Inconsistent max physical segment for Program memory ' *module_name* '

リンクしようとしているモジュールのプログラムメモリ空間のセグメント上限値が異なっていた場合に表示されます。

このエラーの原因として、コンパイル時に/nofar オプションを指定したオブジェクトファイルと、/nofar オプションに対応していないスタートアップファイルをリンクしようとしていることが想定されます。コンパイル時に/nofar オプションを指定したファイルとリンクをする場合には、/nofar オプション対応のスタートアップファイル(NOFAR 擬似命令が記述されているファイル)を指定してください。

7.9.3 エラーメッセージ

E001 Segment type mismatch

パーシャルセグメントの結合を行おうとしましたが，セグメントタイプが一致していません。

E002 Physical segment attribute mismatch

パーシャルセグメントの結合を行おうとしましたが，物理セグメント属性が一致していません。

E003 Physical segment address mismatch

パーシャルセグメントの結合を行おうとしましたが，物理セグメントアドレスが一致していません。

E004 Special region attribute mismatch

パーシャルセグメントの結合を行おうとしましたが，特殊領域属性が一致していません。

E005　Segment size out of range

セグメントのサイズが 64K バイトを超えました。RLU8 が扱えるセグメントのサイズは最大 64K バイトです。

E006　Out of overlay area

オーバーレイユニットを構成するセグメントがオーバーレイ領域の範囲を超えました。オーバーレイ領域の範囲を大きくするなどの処置を行ってください。

E007　Physical segment address out of range

物理セグメントアドレスが割り付け可能な範囲を超えています。

E008　Offset out of range

物理セグメント内のオフセットアドレスが 0H～0FFFFH の範囲を超えています。

E009　Duplicate public symbol

同一名のパブリックシンボルが存在します。リンクするときに同一名のパブリックシンボルが複数存在することは許されません。

E010　Unresolved external symbol

未解決なイクスターナルシンボルが残っています。

外部宣言のみで実際に参照していなければ，/EXC オプションを指定すれば，このエラーを回避することができます。

E011　Cannot find segment

オプションで指定したセグメントが見つかりません。

E012　Control type mismatch

/CODE ，/DATA，/BIT，/NVDATA，/NVBIT，または/TABLE オプションで指定されたセグメントのタイプが異なっています。例えば，セグメントタイプ CODE のセグメント名を/DATA オプションで指定した場合に表示されます。

E013　Cannot change physical segment address

物理セグメントアドレスが決定しているセグメントに対し，/CODE ，/DATA，/BIT，/NVDATA，/NVBIT，または/TABLE オプションを使って変更しようとしました。アセンブル時に決定している物理セグメントアドレスを変更することはできません。

E015　Segment not allocated

空き領域不足，または何らかの理由により，セグメントを割り付けることができませんでした。

E018　Out of range: physical segment address

フィックスアップ処理で決定した物理セグメントアドレスの値が，許容範囲を超えてい

ます。

E019 Out of range: from *min* to *max*

フィックスアップ処理で決定した値が，許容範囲(*min*～*max*)を超えています。

E020 Out of relative jump range

相対ジャンプ命令に関して，フィックスアップ処理で決定した値が，許容範囲を超えています。

E021 Out of SWI address range

ベクタアドレスの割り付け領域が SWI 領域の範囲を超えています。

E022 Out of SWI number

SWI の番号が許容範囲(0～63)を超えています。

E023 Vector address must be #0

ベクタアドレスは物理セグメントアドレス#0 に割り付けられなければなりません。

E024 Out of area: memory range not available

アクセスしようとしている領域には，メモリが存在していません。

E025 Usage type mismatch

イクスターナルシンボル，共有シンボル，またはパブリックシンボルのマッチング処理において，ユーセージタイプが一致しませんでした。両方のシンボルが同じユーセージタイプを持たなければなりません。

E026 Physical segment attribute mismatch

イクスターナルシンボル，共有シンボル，またはパブリックシンボルのマッチング処理において，物理セグメント属性が一致しませんでした。両方のシンボルが同じ物理セグメント属性を持たなければなりません。

E027 Physical segment address mismatch

イクスターナルシンボル，共有シンボル，またはパブリックシンボルのマッチング処理において，物理セグメントアドレスが一致しませんでした。両方のシンボルが同じ物理セグメント属性を持たなければなりません。

E028 CDB information might be incorrect

C コンパイラの出力する C デバッグ情報が正しくない可能性があります。このエラーが発生した場合には，弊社までお知らせください。

7.9.4 ワーニングメッセージ

W001 /NVRAMP is valid for only #0

/NVRAMP オプションで指定できる範囲は，物理セグメント#0 の範囲のみ有効です。

#1 以上の不揮発性メモリを追加したい場合は，/NVRAM オプションを指定してください。

W002 No stack segment, so setting _$$SP to 0

スタックセグメントが見つかりませんでした。スタックシンボル_$$SP の値を 0 にします。

W003 No stack segment, so ignoring /STACK option

/STACK オプションが指定されましたが，スタックセグメントが見つからなかったので，このオプションを無視します。

W004 Stack size must be even

スタックセグメントのサイズは偶数値でなければなりません。このワーニングが出力された場合，スタックサイズは 1 バイト分加算され，偶数サイズに補正されます。

W005 Memory information different

/PDIF オプションが指定されているときに表示されるときがあります。

オブジェクトモジュールに含まれるメモリ情報に違いがありますが，矛盾はありませんのでメモリの情報を OR します。

W006 Cannot access high byte

ワード単位のアクセスをする命令において，対象のアドレスが奇数アドレスになっています。対象のアドレスは偶数アドレスでなければなりません。

W007 Branch address must be even

ジャンプ先アドレスは偶数アドレスでなければなりません。

W008 Branch to different segment

カレントの物理セグメントアドレスから，別の物理セグメントアドレスにジャンプしようとしている命令があります。

W009 Physical segment address must be #0

物理セグメントアドレスは#0 でなければなりません。

W010 Cannot write to ROM

書き込もうとしたアドレスが ROM になっています。ROM にデータを書き込むことはできません。

W011 CODE/TABLE segments overlap

CODE または TABLE タイプのセグメントが，すでに割り付け済みのセグメント（疑似セグメントも含む）とオーバーラップしています。

W012 DATA/BIT segments overlap

DATA または BIT タイプのセグメントが，すでに割り付け済みのセグメント（疑似セグメントも含む）とオーバーラップしています。

W013 NVDATA/NVBIT segments overlap

NVDATA または NVBIT タイプのセグメントが，すでに割り付け済みのセグメント（疑似セグメントも含む）とオーバーラップしています。

W014 Overlay area overlap

オーバーレイ領域が，すでに割り付け済みのオーバーレイ領域，またはセグメントとオーバーラップしています。

W015 Memory type mismatch

ユーセージタイプによって，割り付け先のメモリの種類が決まっていますが，実際にアクセスするメモリの種類が期待するメモリとは異なっています。

W016 Usage type mismatch

アセンブラの記述において，オペランドとして指定できるシンボルのユーセージタイプは決まっていますが，そのユーセージタイプが期待するタイプと異なっています。

W017 Vector address must be even

ベクタアドレスは偶数アドレスでなければなりません。

W018 Specified stack size is too big, so adjusting to *size16*(*size10*) bytes

プログラム，または/STACK オプションによって指定されたサイズのスタックセグメントを割り付けるのに十分な空間がありません。RLU8 は，スタックセグメントのサイズを小さくして割り付けます。*size16*，*size10* は変更後のサイズを 16 進数，10 進数表現したものです。

W019 Out of allocatable memory area

割り付け可能なメモリ領域の範囲を超えています。

/CODE，/DATA，/BIT，/NVDATA，/NVBIT，または/TABLE オプションによって，アブソリュートアドレスが指定された場合，RLU8 は指定されたアドレスにセグメントを割り付けます。このときに，対象のアドレスにメモリが実装されていなかったり，セグメントタイプとメモリの種類の整合がとれていなかった場合（例えば，ROM が実装されている領域に DATA セグメントが割り付けられた場合など）に，このメッセージが表示されます。

W020 Overlay area must RAM or NVRAM

オーバーレイ領域が ROM に割り付けられています。オーバーレイ領域は RAM，または不揮発性メモリに割り付けられなければなりません。

W021 Overlay segment type must be CODE

オーバーレイユニットを構成するセグメントは CODE タイプのセグメントでなければなりません。

W022 Outside ROM window

物理セグメント#0 に配置される TABLE タイプのセグメントが，ROM ウィンドウ領域の範囲を超えて割り付けられています。

W023 Ignoring boundary specification

バウンダリ指定がされていますが，無視します。

W024 0 size segment detected

サイズ 0 のセグメントが見つかりました。RLU8 は，このようなセグメントを，対応するメモリ空間の最下位アドレスに割り付けます。

W025 Cannot find segment

セグメントが見つかりませんでした。

W026 /COMB requires CODE/TABLE segments

/COMB オプションで指定できるセグメントのタイプは CODE または TABLE のみです。

W027 Ignoring /COMB subsegment

/COMB オプションで指定されたメインセグメントが存在しないため，サブセグメントも無視します。サブセグメントは，通常のリロケータブルセグメントとして扱われます。

W028 /COMB segment type mismatch

/COMB オプションで指定されたセグメントのセグメントタイプが一致していません。

W029 /COMB physical segment attribute mismatch

/COMB オプションで指定されているセグメントの物理セグメント属性が一致していません。対象のセグメントは結合されません。

W030 /COMB physical segment address mismatch

/COMB オプションで指定されているセグメントの物理セグメントアドレスが一致していません。対象のセグメントは結合されません。

W031 /COMB special region attribute mismatch

/COMB オプションで指定されているセグメントの特殊領域属性が一致していません。対

象のセグメントは結合されません。

W032 Overlay segments reference each other

オーバーレイユニット間で，互いに参照してはならないもの同士で参照を行っています。

例えば，次のようにオプションで指定した場合，

```
/OVERLAY(OVL1, 1:8000H, 1:8FFFH){UNIT(SEG1) UNIT(SEG2)}
/OVERLAY(OVL2, 1:8000H, 1:9FFFH){UNIT(SEG3)}
```

SEG1 と SEG2 は同じオーバーレイ領域に割り付けられます。このとき，SEG1 内のプログラムから SEG2 内のプログラムを呼び出したりすると，このワーニングメッセージが表示されます。その逆の場合も同様です。

また，オーバーレイ領域 OVL1 とオーバーレイ領域 OVL2 はオーバーラップしています。このとき，SEG3 内のプログラムから SEG1 内のプログラムや SEG2 内のプログラムを呼び出したりする場合にも，このワーニングメッセージが表示されます。その逆の場合の同様です。

W033 Reset vector table not allocated

アブソリュートセグメントをすべて割り付けた時点で，リセットベクタ領域の 0 番地から 3 番地に，ベクタテーブルが割り付けられていませんでした。

リセットベクタ領域の 0 番地からの 2 バイト領域は，スタックポインタの初期値を設定しておく領域です。リセットベクタ領域の 2 番地からの 2 バイト領域は，リセット入力時のプログラム開始アドレスを設定しておく領域です。

これらの設定が行われていない場合，プログラムの正常な動作は保証されません。

```
CSEG AT 0:0000H
DW      _$$SP    ;スタックポインタの初期値
DW      START    ;リセット入力時のプログラム開始アドレス
```

リセットベクタ領域へのベクタテーブルの記述例です。CSEG 擬似命令でアブソリュート CODE セグメントを定義し，ワード単位でスタックポインタの初期値と，リセット入力時のプログラム開始アドレスを定義しています。

W034 Ignoring /CODE or /TABLE option for '*segment_name*'

/COMB オプションで指定した 2 番目以降のセグメント（以降，サブセグメント）に対し，/CODE オプションまたは/TABLE オプションを指定している場合にこのワーニングを表示します。このときリンカは，そのサブセグメントに対する/CODE オプションまたは/TABLE オプションの指定を無視します。

7.9.5 内部処理エラーメッセージ

内部処理エラーメッセージは，RLU8 の内部処理に不具合があった場合に表示されます。形式は次のとおりです。

RLU8 internal error (*position*)

position は，内部処理エラーの発生位置を示す文字列です。通常このエラーが発生することはありませんが，このエラーが発生した場合は，RLU8 のバージョン，エラーが発生したときの状況，および *position* の内容を弊社までお知らせください。

8 LIBU8

8.1 概要

LIBU8 はライブラリファイルを管理するためのソフトウェアです。

ライブラリファイルは，RASU8 で作成した複数のオブジェクトファイルを 1 つのファイルにまとめたもので，LIBU8 によって作成，更新されます。ライブラリファイルに追加したオブジェクトファイルを，オブジェクトモジュールと呼びます。

作成されたライブラリファイルは，RLU8 によって使用されます。

8.1.1 LIBU8 の機能

LIBU8 の機能は次のとおりです。

(1)新しいライブラリファイルを作成します。

(2)オブジェクトモジュールをライブラリファイルに追加します。

(3)ライブラリファイルを別のライブラリファイルに追加します。

(4)ライブラリファイルからモジュールを削除します。

(5)ライブラリファイル中のモジュールを新しいモジュールと置き換えます。

(6)ライブラリファイル中のモジュールをオブジェクトファイルにコピーします。

(7)ライブラリファイル中のモジュールをオブジェクトファイルに抽出します。

(8)リストファイルを作成します。

8.1.2 LIBU8 を使う利点

プログラムを複数のモジュールに分けて作成した場合に，あるモジュールは汎用性があり，それを他のプログラムにも使用する場合が出てくると思います。このような汎用性のあるモジュールが少数のうちは問題ないのですが，数が増えてくるにつれて，それらのモジュールをユーザが管理することは次第に困難になってきます。

このような時に，それらのモジュールをライブラリに登録しておけば，前に述べたような問題を解決することができます。RLU8 によってリンクをするときにそのライブラリファイルを指定すれば，RLU8 が必要なオブジェクトモジュールをライブラリファイルから探し出してくれるため，ユーザはモジュール管理の苦労から開放されます。

8.1.3 ファイル名とモジュール名の違い

LIBU8 はファイル名，モジュール名を次のように定義しています。

ファイル名にはドライブ名，ディレクトリ名，拡張子の指定ができます。

モジュール名は，ライブラリファイルの中のモジュールを表す名前です。この名前は RASU8 によって決定されます。RASU8 はコマンドラインで指定されたソースファイル名からドライブ名，ディレクトリ名，拡張子を取り去り，残ったベース名をモジュール名とします。そしてこ

の情報をオブジェクトファイル中に出力します。

モジュール名の大文字と小文字は区別されます。このため，同じファイルから違うモジュール名を持つオブジェクトファイルを作成してしまうことがあります。例えばソースファイル MODULE.ASM をアセンブルするとき

```
RASU8 MODULE
```

と大文字で記述すると，モジュール名は“MODULE”となり，

```
RASU8 module
```

と小文字で記述すると，モジュール名は“module”となります。

ライブラリ中に 2 つの別々のモジュールとして登録可能ですが，使用する際に意図しない動作をする恐れがありますので，登録時のモジュール名の指定は大文字と小文字の区別ではなくモジュール名での区別をお勧めします。

ライブラリ中のモジュールのモジュール名を知りたい場合は，リストファイルを作成してください。

例

```
LIBU8  MYLIB;
```

MYLIB.LIB のリストファイル MYLIB.LST が作成されます。

LIBU8 の操作では，モジュールを追加するときにオブジェクトファイル名を指定し，ライブラリファイルからの削除やコピーのときなどはモジュール名を指定します。

8.2 LIBU8 の実行

LIBU8 を実行する方法は，次の種類があります。

(1)コマンドラインによるライブラリ操作

(2)プロンプトを使っての実行

(3)コマンドラインとプロンプトを組み合わせた使い方

(4)応答ファイルを利用した使い方

8.2.1 コマンドラインによるライブラリ操作

DOS のプロンプト上で LIBU8 への指定をすべて行う方法です。コマンドラインの書式は，次のとおりです。

LIBU8 *library_file* [*operations*][,[*list_file*][,[*output_library_file*]]][;]

library_file と *operations* の間はスペースで区切ります。*operations* 以降の各フィールドは，かならずコンマ（,）で区切る必要があります。

フィールド	指定する内容
library_file	入力ライブラリファイル名 （作成，または更新したいライブラリファイルの名前）
operations	*library_file* で指定したライブラリへの操作
list_file	リストファイル名
output_library_file	出力ライブラリファイル名 （操作によって作成されたライブラリファイルの名前）

各フィールドに指定するファイル名は，パス，拡張子を含めて最大 255 文字までを指定することができます。

library_file を省略することはできません。その後のフィールドは，コンマを指定することによって省略できます。セミコロン（;）を指定すると，それ以降のフィールドをすべて省略できます。つまりセミコロンは入力の終わりを示します。LIBU8 は，セミコロンより後の記述をすべて無視します。コンマやセミコロンで省略したフィールドには，デフォルトの値が使用されます。

フィールド	デフォルトの値
operations	何の操作も行いません。
list_file	リストファイルを作成します。ファイル名は出力ライブラリファイルの拡張子を“.lst”に変えたものとします。
output_library_file	*library_file* で指定した入力ライブラリファイル名と同じになります。

ファイルの種類と，ファイル指定の各フィールドのデフォルトを以下に示します。

ファイル指定	ディレクトリ	ベース名	拡張子
入力ライブラリ	カレント	省略不可	.lib
オペレーションのファイル	カレント	省略不可	.obj
出力ライブラリ	入力ライブラリのディレクトリ*	入力ライブラリのベース名	.lib
リストファイル	出力ライブラリのディレクトリ*	出力ライブラリのベース名	.lst

*ファイル名が指定されており，ディレクトリの指定が省略されていた場合は，カレントディレクトリになります。

例

```
LIBU8  MYLIB;
```

operations が省略されているので何の操作も行わず，リストファイルだけを作成します。*list_file* の指定も省略されているので，リストファイル名をデフォルトの MYLIB.LST にします。

8.2.1.1 *library_file* フィールド

library_file には，作成または操作するライブラリのファイル名を指定します。このフィールドに対する入力は省略できません。コマンドラインでこのフィールドを省略すると，LIBU8 はプロンプトを表示し，入力を促します。ここでもファイル名を指定しなければ，LIBU8 はユーセージを表示して終了します。

ライブラリファイルのデフォルトの拡張子は“.LIB”とします。たとえば，MYLIB と指定すると，LIBU8 の解釈は MYLIB.LIB とみなします。デフォルト以外の拡張子をつけることもできます。

その場合は拡張子まで指定しなければなりません，拡張子を付けない場合は MYLIB. のようにピリオド（.）をファイル名の最後につけます。ドライブ名，ディレクトリ名の指定がなければカレントドライブ，カレントディレクトリが指定されたものとみなします。

8.2.1.2 *operations* フィールド

このフィールドには，*library_file* で指定したライブラリに対する操作を記述します。このフィールドの指定を省略すると，ライブラリファイルに対して何の操作も行いません。ただし，ライブラリファイルの内容が正常であるかを確認するチェックは行い，リストファイルを作成します。

操作の記述は，操作を表すオペレーションシンボル（+， -， %， *， &）と，操作の対象になるファイル名またはモジュール名で構成します。複数の操作を指定するときは，

```
LIBU8 MYLIB +ADCON +CALC +DISPLAY;
```

のように必ず各操作の間をスペースであける必要があります。オペレーションシンボルの後に

スペースはあってもなくてもかまいません。オペレーションシンボルの意味は，次のとおりです。

機能	オペレーションシンボル	説明
追加	+	オブジェクトファイルまたはライブラリファイル中のモジュールを，ライブラリに追加します。
削除	-	ライブラリ中からモジュールを削除します。
置換	%	ライブラリ中のモジュールを新しいモジュールと置き換えます。
コピー	＊	ライブラリ中のモジュールをオブジェクトファイルにコピーします。ライブラリ中のモジュールは残ります。
抽出	&	ライブラリ中のモジュールをオブジェクトファイルに抽出します。ライブラリ中のモジュールは残りません。

ファイル名の拡張子を省略すると，デフォルトの拡張子“.OBJ”とみなします。モジュール名にドライブ名，ディレクトリ名，および拡張子を指定した場合は，それらの指定は無視します。

例

```
LIBU8 MYLIB +GET-KEY;
LIBU8 MYLIB +GET -KEY;
```

上の例では，MYLIB.LIB に GET-KEY.OBJ を追加します。下の例では，MYLIB.LIB 中のモジュール KEY を削除した後に，GET.OBJ を追加します。

複数の操作を記述したとき，それらの実行順序は，操作のもつ優先度によって決定します。詳細については，「8.5.8 操作の優先度」を参照してください。

8.2.1.3 *list_file* フィールド

このフィールドには，リストファイルのファイル名を指定します。リストファイルは，テキストファイルで，ライブラリ中のモジュールの情報とそのモジュールに含まれているパブリックシンボルをアルファベット順に表示しています。リストファイルの詳細については「8.6 リストファイルの形式」を参照してください。

リストファイルは，特に指定をしなくてもデフォルトで作成します。この時のファイル名は，出力ライブラリファイル名の拡張子を“.LST”に変更したものとします。デフォルトのファイル名を使いたくない時には，このフィールドでファイル名を指定します。

リストファイルを作成しない場合は，このフィールドに NUL を指定します。

例

```
LIBU8 MYLIB +CALC, C:¥WORK¥LIBLIST.;
```

この例では，リストファイル名を C:¥WORK¥LIBLIST にします。

この例では，リストファイル名を C:¥WORK¥LIBLIST にします。

例

```
LIBU8 MYLIB %CALC, NUL;
```

NUL が指定されているので，リストファイルは作成しません。

8.2.1.4 *output_library_file* フィールド

このフィールドには，出力ライブラリファイル名を指定します。指定を省略すると，LIBU8 は出力ライブラリファイル名を入力ライブラリファイルと同じ名前にします。デフォルトのファイル名を変更したい時に，このフィールドでファイル名を指定します。

LIBU8 は，出力ライブラリファイルと同じ名前のファイルが存在すると，そのファイルの拡張子を“.LBK”に変更して，バックアップファイルにします。

例

```
LIBU8 MYLIB -DISPLAY, , NEWLIB
```

ライブラリファイル MYLIB.LIB からモジュール DISPLAY を削除し，その結果を NEWLIB.LIB へ出力します。もしすでに NEWLIB.LIB が存在していた場合，そのファイルのファイル名を NEWLIB.LBK に変更し，バックアップファイルとします。MYLIB.LIB は書き変わらりません。リストファイル NEWLIB.LST を作成します。

このフィールドは，ライブラリファイルの内容を変更した時だけ有効です。

以下の場合は，このフィールドを指定しても意味がないためワーニングを表示します。

1. 操作を指定していない時
2. 操作がコピーだけだった時
3. 新規ライブラリの作成時

8.2.2 プロンプトを使っての実行

LIBU8 が出力するプロンプトに対して指定を行う方法です。DOS のプロンプトへ LIBU8 とタイプすると，LIBU8 は以下のプロンプトを 1 行ずつ表示し，ユーザの応答を待ちます。それぞれのプロンプトに応答するまで，LIBU8 は次のプロンプトを表示しません。入力があれば次のプロンプトを表示します。

```
Library file   :
Operations     :
List file      :
Output library :
```

各プロンプトは，コマンドラインの各フィールドに対応しています。

プロンプト	コマンドラインフィールド
Library file :	*library_file* フィールド
Operations :	*operations* フィールド
List file :	*list_file* フィールド
Output library :	*output_library_file* フィールド

例

```
Library file   : ABC
Operations     : +A +B +C ;
```

ライブラリファイル ABC にオブジェクトファイル A.OBJ， B.OBJ， C.OBJ を追加します。リストファイル ABC.LST も作成します。

例

```
Library file   : ABC
Operations     : %C:XYZ
List file      : NUL
Output library : C:DEF
```

ライブラリファイル ABC.LIB のモジュール XYZ を，オブジェクトファイル C:XYZ.OBJ と置き換えます。そして，その結果を C:DEF.LIB へ出力します。もとの ABC.LIB は書き変えません。NUL が指定してあるのでリストファイルは作成しません。

例

```
Library file   : MYLIB ;
```

または，

```
Library file   : MYLIB
Operations     : ,
```

両方とも同じ動作をします。ライブラリファイル MYLIB.LIB のリストファイル MYLIB.LST を作成します。ライブラリファイルは変更しません。

例

```
Library file   : MYLIB
Operations     : -DISPLAY
List file      : ,
Output library : NEWLIB
```

ライブラリファイル MYLIB.LIB からモジュール DISPLAY を削除し，その結果を NEWLIB.LIB へ出力します。もし NEWLIB.LIB がすでに存在していた場合，そのファイルのファイル名を NEWLIB.LBK に変更して，バックアップファイルとします。MYLIB.LIB は変更しません。リストファイル NEWLIB.LST を作成します。

8.2.3 コマンドラインとプロンプトを組み合わせた使い方

コマンドラインで，LIBU8 への指定が足りない場合，LIBU8 はプロンプトを表示して入力を促します。

例

```
LIBU8 MYLIB +CALC
```

と入力すると，次のように *operations* よりも後のプロンプトを表示します。

このプロンプトへの指定がすむと LIBU8 が実行します。

```
List file      :
Output library :
```

このプロンプトを表示させたくない場合は，コマンドラインの最後にセミコロンをつけます。LIBU8 はセミコロンを入力の終わりと認識し，それ以降のプロンプトを表示しません。

```
LIBU8 MYLIB +CALC;
```

これでプロンプトは表示されずに，即座に実行を開始します。

8.2.4 応答ファイルを利用した使い方

LIBU8 の起動に，応答ファイルを使うことができます。コマンドラインで 1 度に入らないような長いコマンドや，おなじ操作を何回も繰り返す場合に使用します。応答ファイルの指定は，コマンドラインでのみ有効です。プロンプトを利用した入力形式で応答ファイルを指定しても，それは応答ファイルとはみなしません。

まず，エディタを使って応答ファイルを作成します。このファイルには，コマンドラインの入力形式で必要な操作を書いておきます。一行に記述できない場合は，複数の行に分けて記述してもかまいません。LIBU8 は，改行を空白として扱います。

すべての入力フィールドを複数の行に分けて記述し，応答ファイル MYLIB.RES を作成します。応答ファイルには，コメントを記述することもできます。コメントは“#”または“//”で始ま

り，行末までとします。

```
##############################################
#   応答ファイル記述例
#   '#'または'//'から行末まではコメントとみなします
##############################################
MYLIB                          // 入力ライブラリファイル
+A +B +C                       // オペレーション指定
+D +E +F +G +H +I +K +L ,      // オペレーション指定
LISTFILE,                      // リストファイル
OUTLIB                         // 出力ライブラリファイル
```

この場合は，次のようにタイプして LIBU8 を起動します。

```
LIBU8 @MYLIB.RES
```

また，次のように入力フィールドの一部だけを応答ファイルに記述することもできます。

例

オペレーションフィールドについてのみ記述し，応答ファイル OPERATE.RES を作成します。

```
+A +B +C +D +E +F +G +H +I +K +L
```

この場合は，次のようにタイプして LIBU8 を起動します。

```
LIBU8 MYLIB @OPERATE.RES , LISTFILE , OUTLIB
```

8.3 出力されるメッセージのリダイレクト

LIBU8 が画面に表示するメッセージは，すべて標準出力に出力します。したがって，DOS のリダイレクトの機能を使って，メッセージをファイルに出力することができます。画面にメッセージを表示したくないときは，NUL を指定します。

例

```
LIBU8 MYLIB +ERR ; > ERRMES
```

画面に表示されるメッセージをファイル ERRMES にリダイレクトします。

例

```
LIBU8 MYLIB %A %B %C ; > NUL
```

メッセージを画面に表示しません。

8.4 終了コード

LIBU8 は，終了時に次のような終了コードを DOS へ返します。

終了コード	終了時の状態	意味
0	正常終了	エラーの発生がありませんでした。
1	ワーニング	問題の起きた操作も実行されました。
2	エラー	問題の起きた操作が無視されました（それ以外の操作は実行されました）。
3	フェイタルエラー	実行中に問題が発生し操作を続行できませんでした。

8.5 LIBU8 の操作

LIBU8 は次の機能を持ちます。

(1) 新規ライブラリの作成

(2) モジュールの追加

(3) ライブラリファイルの追加

(4) モジュールの削除

(5) モジュールの置換

(6) モジュールのコピー

(7) モジュールの抽出

(8) リストファイルの作成

この節では，(1)〜(7)について詳しく説明していきます。

(8)のリストファイルはデフォルトで作成します。リストファイル名の指定方法については「コマンドラインによるライブラリ操作　8.2.1.3 *list_file* フィールド」を参照してください。リストファイルの形式については「8.6 リストファイルの形式」を参照してください。

8.5.1 新規ライブラリの作成

新しくライブラリを作成する場合は，コマンドラインの *library_file* フィールドか，“Library file”プロンプトに対して，そのファイル名を指定します。

指定したファイル名に拡張子がない場合は，LIBU8 は“.LIB”を自動的に付加します。別の拡張子を指定することも可能ですが，LIBU8 と RLU8 では，ライブラリファイルの拡張子のデフォルトは“.LIB”とするので，この拡張子を使うことを推奨します。

ライブラリファイル名にドライブ名，ディレクトリ名の指定をすることもできます。指定されない場合は，カレントドライブ，カレントディレクトリにライブラリファイルを作成します。

起動時にライブラリファイル名以外の指定がなければ，次のプロンプトを表示します。

```
filename.lib - File does not exist. Create ? [Y/N]
```

ここで n または N をいれると，LIBU8 はファイルを作成せずに終了します。作成するときは y または Y と入力します。また，Enter キーを押すだけでもかまいません。

他のフィールドの指定があるとライブラリファイルを作成するとみなします。その場合，このプロンプトは表示しません。セミコロンやコンマで他のフィールドを省略したときも同様です。

例

```
LIBU8 NEWLIB
```

この場合はプロンプトを表示して，作成の有無を問います。

例

```
LIBU8 NEWLIB;
```

プロンプトを表示せず，NEWLIB.LIB と NEWLIB.LST を作成します。NEWLIB.LIB には 1 つのモジュールも入りません。

例

```
LIBU8 NEWLIB +A ;
```

プロンプトを表示せずに，モジュール A を持つ NEWLIB.LIB と NEWLIB.LST を作成します。

新規ライブラリ作成時には，追加以外の操作はエラーとなります。削除，コピー，置換，および抽出はライブラリファイル中にモジュールが無いので無効となります。

新規ライブラリ作成時は，出力ライブラリファイルの指定はできません。新規ライブラリ作成時に出力ライブラリファイルの指定をした場合はワーニングを表示し，出力ライブラリファイルの指定を無視します。

8.5.2 モジュールの追加

構文

+*object_file*

説明

+は，指定したオブジェクトファイルをライブラリに追加します。*object_file* には，ライブラリに追加するオブジェクトファイルの名前を指定します。*object_file* には，DOS のファイル指定が可能です。オブジェクトファイル名の長さはパス，拡張子も含めて，最大 255 文字までを指定可能です。拡張子を省略した場合，デフォルトの拡張子“.OBJ”を付加します。

LIBU8 は，指定されたファイルからモジュール名を取り出します。LIBU8 は，ライブラリ中の各モジュールを，モジュール名のアルファベット順に並べ替えます。

LIBU8 はモジュールをライブラリに追加するときに，すでに追加されているモジュールかどうかチェックを行います。チェックの内容は次のとおりです。

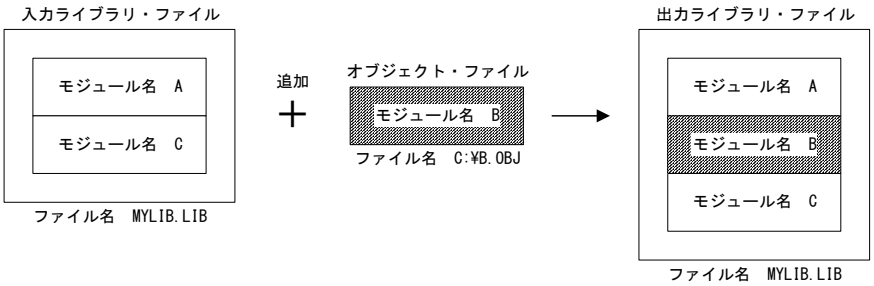
1. ライブラリ中に，追加するモジュールと同一名のモジュールが存在しないこと。

2. 追加するモジュールで宣言されているパブリックシンボルと，同じパブリックシンボルが，ライブラリ中の他のモジュールで宣言されていないこと。（パブリックシンボルとは，ソースプログラム中で，RASU8 の PUBLIC 擬似命令を使って宣言したシンボルです。このシンボルは他のモジュールから参照できます。）

このチェックでエラーが発生すると，LIBU8 はそのモジュールを追加しません。

例

```
LIBU8 MYLIB +C:¥B.OBJ ;
```



8.5.3 ライブラリファイルの追加

構文

+library_file

説明

+ の後にライブラリファイルを指定すると，その中のすべてのモジュールを入力ライブラリファイルに追加します。*library_file* には DOS のファイル指定が可能です。ライブラリファイル名の長さはパス，拡張子も含めて，最大 255 文字までを指定可能とします。また，拡張子は省略できません。これは *operations* のフィールドで，ファイル名の拡張子を省略すると“.OBJ”とみなすためです。ライブラリファイルの拡張子は“.LIB”以外のものでもかまいません。

LIBU8 は，ライブラリ中のモジュールを追加するときに，それらのモジュールが追加できるかどうかをチェックします。チェックの内容は次のとおりです。

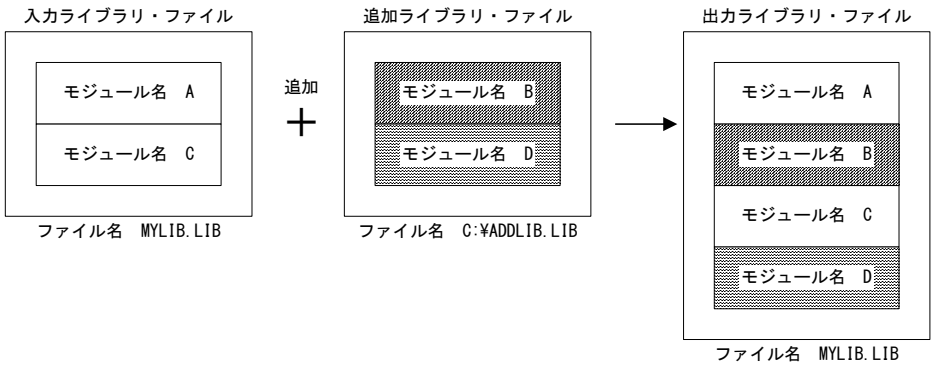
1. 編集対象のライブラリ中に，追加するモジュールと同一名のモジュールが存在しないこと。
2. 追加するモジュールで宣言されているパブリックシンボルと，同じパブリックシンボルが，編集対象のライブラリ中において他のモジュールで宣言されていないこと。

このチェックでエラーが発生すると，LIBU8 はそのモジュールを追加しません。エラーの発生しなかったモジュールの追加は行います。

LIBU8 は，ライブラリ中の各モジュールを，モジュール名のアルファベット順に並べ替えます。

例

```
LIBU8 MYLIB +C:¥ADDLIB.LIB ;
```



8.5.4 モジュールの削除

構文

-module_name

説明

ライブラリファイルからモジュールを削除するときは - を使います。*module_name* には，削除したいモジュールのモジュール名を指定します。モジュール名の長さは最大 255 文字まで指定可能です。次の記述例では，ライブラリ MYLIB.LIB から モジュール B を削除しています。

```
LIBU8 MYLIB -B ;
```

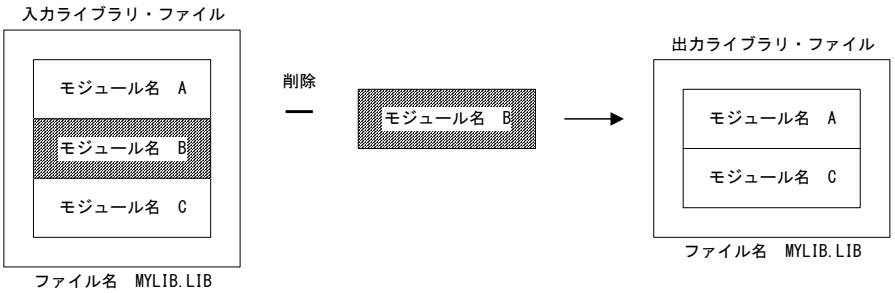
もし *module_name* に，ドライブ名，ディレクトリ名および拡張子が指定されていた場合，LIBU8 はそれらを無視します。たとえば，次のように指定しても，LIBU8 はカレントディレクトリ上の MYLIB.LIB からモジュール B を削除します。

```
LIBU8 MYLIB -C:¥WORK¥B ;
```

指定したモジュール名が，ライブラリ中にみつからない場合はエラーとなります。

例

```
LIBU8 MYLIB -B.OBJ ;
```



8.5.5 モジュールの置換

構文

%object_file

説明

%は，ライブラリ中のモジュールを，指定したオブジェクトファイルに置き換えます。*object_file* には，置き換えるオブジェクトファイル名を指定します。*object_file* には，DOS のファイル指定が可能です。オブジェクトファイル名の長さはパス，拡張子も含めて，最大 255 文字まで指定可能です。拡張子を省略すると，デフォルトの拡張子“.OBJ”を付加します。LIBU8 は *object_file* で指定したファイル名からベース名を取り出し，モジュール名とします。このモジュール名でライブラリ中のモジュールをサーチします。

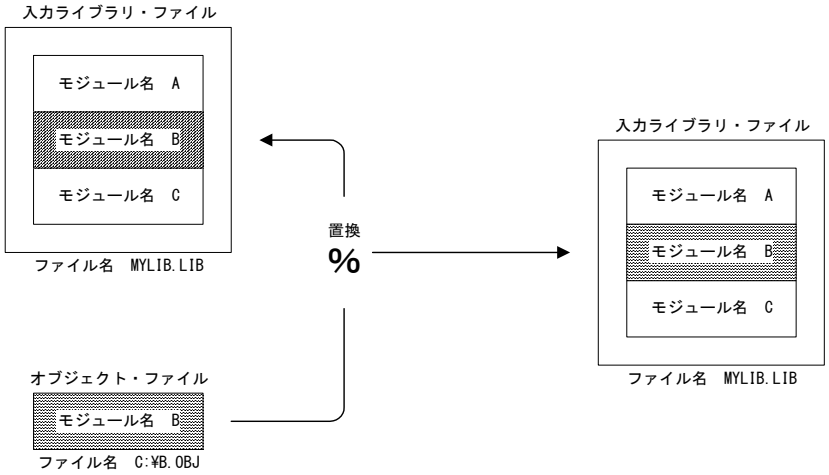
最初に LIBU8 は次のチェックを行います。このチェックでエラーが発生すると，モジュールの置換は行われません。

1. 指定したモジュール名がライブラリ中にあること。
2. ライブラリ中のモジュールと，置き換えるモジュールのモジュール名が一致すること。
3. 置き換えるモジュールのパブリックシンボルが，ライブラリ中のほかのモジュールで定義されていないこと（置き換えられるモジュールのパブリックシンボルは考慮されていません）。

上のチェックでエラーが発生しなければ，LIBU8 は指定したモジュールをライブラリから削除します。削除が正常に行われるとオブジェクトファイルを追加します。指定したオブジェクトファイルが存在しなかった場合や，エラーが発生したときは，置き換えられるモジュールは削除されず，そのままライブラリに残ります。

例

```
LIBU8 MYLIB %C:¥B.OBJ;
```



8.5.6 モジュールのコピー

構文

**object_file*

説明

*は，ライブラリ中のモジュールを取り出して，オブジェクトファイルにコピーします。コピーされたモジュールはライブラリ中に残ります。

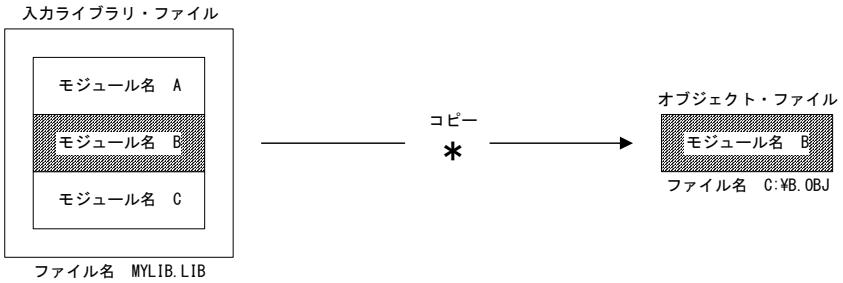
object_file には，DOS のファイル指定が可能です。オブジェクトファイル名の長さはパス，拡張子も含めて，最大 255 文字まで指定可能です。拡張子が省略されていると，デフォルトの拡張子“.OBJ”を付加します。

LIBU8 は，*object_file* で指定したファイル名から，ベース名を取り出しモジュール名を作成します。このモジュール名を使ってライブラリ中からモジュールをサーチします。モジュールが見つからなかった場合はエラーを表示し，この作業を無視します。モジュールが見つかると，*object_file* で指定した名前でオブジェクトファイルを作成し，モジュールをコピーします。同じ名前のファイルが存在していた場合は，LIBU8 はワーニングメッセージを表示してモジュールを上書きします。

この操作はライブラリの内容を変更しません。したがって，コピーの操作だけを指定した時は，出力ライブラリファイルを作成しません。このとき起動時に出力ライブラリファイルの指定をするとワーニングを表示します。またバックアップファイルも作成しません。

例

```
LIBU8 MYLIB *C:¥B.OBJ;
```



8.5.7 モジュールの抽出

構文

&object_file

説明

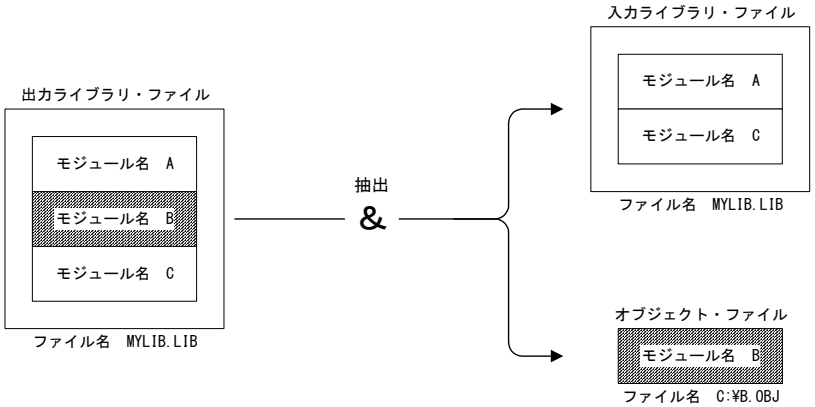
&は，ライブラリ中のモジュールを取り出して，オブジェクトファイルにコピーします。コピーが終了するとそのモジュールはライブラリから削除します。この操作は，モジュールをコピー（*）した後，削除（-）するのと同じことになります。

object_file には，DOS のファイル指定が可能です。オブジェクトファイル名の長さはパス，拡張子も含めて，最大 255 文字まで指定可能です。拡張子が省略されていると，デフォルトの拡張子“.OBJ”を付加します。

LIBU8 は，*object_file* で指定したファイル名から，ベース名を取り出しモジュール名を作成します。このモジュール名を使ってライブラリ中からモジュールを探します。モジュールが見つからなかった場合はエラーを表示し，この作業を無視します。モジュールが見つかると，*object_file* で指定した名前でオブジェクトファイルを作成し，モジュールをコピーします。同じ名前のファイルが存在していた場合は，LIBU8 はワーニングメッセージを表示してモジュールを上書きします。コピーが終わるとライブラリの中のモジュールを削除します。

例

```
LIBU8 MYLIB &C:¥B.OBJ;
```



8.5.8 操作の優先度

操作は優先度を持っています。LIBU8 は操作の指定された順序に関係なく，優先度の高いものから実行します。下に優先度の表を示します。1 がもっとも高い優先度を持っています。同じ優先度をもつ場合は，起動時に指定した順番（左から右）に実行します。

優先度	操作
1	削除（-）　コピー（*）　抽出（&）
2	置換（%）
3	追加（+）

例

```
LIBU8 MYLIB +ADCON %RAMCHK -DISPLAY *CALC &EXTMEM;
```

まず，ライブラリ MYLIB 中のモジュール DISPLAY の削除，CALC のコピー，EXTMEM の抽出を実行します。次にモジュール RAMCHK の置換を行い，最後に ADCON.OBJ をライブラリに追加します。

8.5.9 テンポラリファイル

LIBU8 はテンポラリファイルを，出力ライブラリファイルと同じディレクトリ上に作成します。LIBU8 はこのファイルを使って作業を行います。作業が正常に終わると，テンポラリファイルの名前を出力ライブラリファイル名に変更します。したがって，テンポラリファイルはディレクトリに残りません。

ただし，実行時にフェイタルエラーが起きた場合や，ユーザが Ctrl+C で LIBU8 の実行を中断させた場合などは，テンポラリファイルがそのまま残る場合があります。テンポラリファイルが残っていても，LIBU8 の動作に支障はありません。

テンポラリファイル名は，“$LIBU8$.$$$”とし，常に上書きします。したがって，ユーザは同一名のファイルを作成しないように注意する必要があります。

また，ライブラリの内容が変更されない場合，LIBU8 はテンポラリファイルを作成しません。これは次のような場合です。

1. オペレーションの指定がない場合。
2. オペレーションでコピー（*）だけが指定された場合。

8.6 リストファイルの形式

リストファイルは，ライブラリの内容を表すテキストファイルです。ライブラリ自身に関する情報とライブラリ中に含まれるすべてのモジュールの情報と，そのモジュール中で定義されているパブリックシンボルを表示します。

デフォルトでは，常にリストファイルを作成します。リストファイルを作らないときは，*list_file* フィールドに“NUL”を指定します。

リストファイルの例を下に示します。(1)～(10)は，説明のために付けている番号で，実際のリストには表示されません。

```
LIBU8 Object Librarian, Ver.1.00 Library Information
[Tue Jun 01  12:00:00  1999] ← (1)

LIBRARY FILE : test.lib ← (2)
MODULE COUNT : 1 ← (3)


MODULE NAME  : test ← (4)
DATE         : 06-01-1999  11:55:32 ← (5)
BYTE SIZE    : 0000034Eh(846) ← (6)
MEMORY MODEL : SMALL ← (7)
TRANSLATOR   : RASU8 (Ver.1.00) ← (8)
CORE ID      : 1  ← (9)
TARGET       : ML610001 ← (10)

==== PUBLIC SYMBOLS ==== ← (11)

_PubSym1    _PubSym     _PubSym3
```

上のリストファイルの例に付いている番号の順に説明します。(1)～(3)まではライブラリファイル自身の情報です。(4)～(10)はライブラリファイル中の各モジュールの情報です。

(1)LIBU8 実行時の日付。［曜　月　日　時：分：秒　年］の形式で表示します。

(2)ライブラリファイル名。

(3)ライブラリ中に含まれるモジュールの数。

(4)モジュール名。モジュール名の文字数は最大 255 文字までです。

(5)ライブラリに登録された日付，および時間。月-日-年 時:分:秒 の形式で表示します。

(6)モジュールのバイトサイズ。16 進（10 進）の形式で表示します。

(7)モジュールのメモリモデル。“SMALL”または“LARGE”のいずれかを表示します。

(8)モジュールを作成したアセンブラの名前。（）内はそのアセンブラのバージョンです。

(9) CPU コアの種類を示す番号。

(10) モジュールのターゲット名。

(11)モジュール中で宣言されているパブリックシンボル。アルファベット順に表示します。パブリックシンボルが無い場合は，“- None -”を表示します。

8.7 エラーメッセージ

LIBU8 はフェイタルエラー，エラー，ワーニングの 3 種類のエラーメッセージを表示します。

フェイタルエラー

フェイタルエラーが発生すると，LIBU8 はエラーメッセージを表示して，ただちに作業を終了します。既存のファイルは変更しません。フェイタルエラーが発生したときは，テンポラリファイルがディスク上に残る場合があります。

エラー

エラーが発生すると，エラーの起きた操作を無視します。他の操作は行います。作成したファイルは使用できます。LIBU8 は，終了時にエラーの数を画面に表示します。

ワーニング

ワーニングが発生してもその操作を実行します。操作によって作成されたファイルは使用できます。 LIBU8 は，終了時にワーニングの数を画面に表示します。

8.7.1 エラーメッセージの書式

LIBU8 は，次の書式でエラーメッセージを画面に表示します。

error_level *error_code*: *error_message*

またエラーによっては，エラーの情報を伝えるために，もう 1 行表示する場合があります。これは次に示した行のうち，いずれかのフォーマットで表示します。

Library file : *library_file*　Module name : *module_name*

Library file : *library_file*　Module : *module_name*　Offset : *XXXX*H

Library file : *library_file*　Offset : *XXXX*H

Module name : *module_name*　Offset : *XXXX*H

上記のフォーマットで使用している表記の説明を以下に示します。

マニュアルでの表記	画面での表示
error_level	Fatal error, Error, Warning のどれか。
error_code	エラーコード。
error_message	エラーの内容を示すメッセージ。
library_file	エラーの起きたライブラリファイル名。
module_name	エラーの起きたモジュールのモジュール名。
XXXX	エラーの起こった位置（ファイル先頭からのオフセット）。16 進数で表示します。

以下，エラーメッセージをエラーの種類別に分けて説明します。エラーメッセージはメッセージコード順に並べてあります。

8.7.2 フェイタルエラーメッセージ

001 Bad input format

起動行の記述に間違いがあります。

002 Bad object filename specification

オペレーションシンボルの後にオブジェクトファイル名が与えられていません。またはオブジェクトファイル名に無効な文字が記述されていました。

004 Cannot open temporary file

テンポラリファイル ”$LIBU8$.$$$” が読み出し専用ファイルになっています。

DOS の ATTRIB コマンド等を使ってファイルの読み取り専用属性を解除してください。

005 Cannot rename old library

入力したライブラリファイルをバックアップファイルに変更できません。入力ライブラリファイルに指定したファイルの拡張子が “.LBK” だと，このメッセージが表示されます。

LIBU8 は入力ライブラリファイルの拡張子をチェックしていません。ライブラリが書き換わると無条件で入力ライブラリファイルの拡張子を “.LBK” に変更します。ところが，入力ライブラリファイル名の拡張子が “.LBK” の時はバックアップファイル名に変更することができません。

バックアップファイルを入力ライブラリファイルとして扱いたい場合は，DOS の REN コマンド等でファイル名を変更してから指定してください。

006 Checksum error

現在，処理しているレコードのチェックサムが間違っています。ファイルが壊れている可能性があります。ファイルを作り直してから LIBU8 を実行してください。

008 Disk full error

ディスクの容量が足りません。必要のないファイルを削除または移動してディスクに空き領域を作ってください。

009 EOF expected after module index records

ライブラリファイルの最後に EOF がありません。ライブラリファイルが壊れている可能性があります。

010 011 File name too long

起動時に指定されたファイル名，またはモジュール名の長さが 255 文字を超えました。

012 File seek error

シークエラーが発生しました。

013 Premature EOF

LIBU8 のプロンプトに EOF が入力されました。リダイレクトを使って起動しているとき，LIBU8 への指定が終わる前に EOF が現れました。たとえば，次のような場合です。

```
MYLIB %ADCON %CALC %DISPLAY  , , <EOF>
```

次のように，行の最後にはかならず改行コードを入れるか，

```
MYLIB %ADCON %CALC %DISPLAY  , , <改行コード>
```

最後にセミコロンを指定してください。

```
MYLIB %ADCON %CALC %DISPLAY  , ;
```

014 Insufficient memory

LIBU8 の実行に必要なメモリが足りません。常駐プログラムをはずしたり，デバイスドライバを減らしたりして，使用可能なメモリを増やしてください。

015 Invalid library

指定されたライブラリファイルに，異常なレコードや情報が含まれていました。

018 I/O read error

ファイル読み込み時にエラーが発生しました。

019 I/O write error

ファイル書き込み時にエラーが発生しました。

020 ‘filename’ is write-protected

ファイル属性がで読み取り専用属性になっています，読み取り専用属性を解除してください。

021 Library file might be corrupted

ライブラリファイルが間違った情報を持っています。

031 Too many public symbols

パブリックシンボルが多すぎてライブラリに登録できません。別のライブラリファイルを　　作成するか，パブリックシンボルにする必要のない（ほかのモジュールから参照されない）シンボルのパブリック宣言を削除してください。

ライブラリファイルに登録できるパブリックシンボルの数は，１つのモジュールにつき，すべてのシンボルの長さが 32 文字の場合，1983 個になります。

032 Unable to create new library

出力ライブラリファイルを作成できません。

034 Unable to open library

入力ライブラリファイルがオープンできません。

037 Unable to open response file

応答ファイルをオープンすることができません。

038 Unexpected end of file

存在すると予想されるデータを読み込むことができませんでした。ファイルが途中で終わっているときなどに，このエラーが出ます。指定したファイルが壊れている可能性があります。

8.7.3 エラーメッセージ

103 Ignoring modules past 65535

ライブラリファイルに登録できるモジュールの数は 65535 個までです。それ以上のモジュールは登録できません。新しく別のライブラリファイルを作成してください。

115 Invalid library

指定したライブラリファイルが正しくありません。ライブラリファイルにモジュールを追加するときは LIBU8 で作成したオブジェクトファイルを指定してください。

116 Ignoring invalid object module

指定したオブジェクトモジュールが正しくありません。ライブラリファイルにモジュールを追加するときは RASU8 で作成したオブジェクトファイルを指定してください。

124 Ignoring module already in library

指定したモジュールはすでにライブラリに登録されています。このモジュールは登録されていません。

125 Ignoring module with different name

このエラーはモジュールの置換(%)のときだけ発生します。オブジェクトファイルの中のモジュール名と，ライブラリ中のモジュール名が違います。このモジュールは登録されません。

126 Ignoring module not in library

コピー(*)，抽出(&)，削除(-)で指定したモジュールがライブラリに登録されていません。

この操作は無視されます。

127 No room in library, so ignoring

ライブラリファイルが 4 G バイトを超えたためモジュールを登録することができません。不要なモジュールを削除するか別に新しくライブラリファイルを作り，それに登録して

ください。

133 Unable to open file, so ignoring

コピー(*)，抽出(&)の操作で指定されたファイル名では，ファイルを作成することができません。この操作は無視されます。

134 Unable to open library

ライブラリファイルをオープンできません。

135 Unable to open list file

リストファイルをオープンできません。

136 Unable to open object file, so ignoring

指定されたオブジェクトファイルをオープンできません。このファイルは無視されます。

140 Ignoring public symbol *public_symbol* (*file_name*) redefinition in *module_name* (*library_file*)

追加しようとした *file_name* で示されるオブジェクトファイルまたはライブラリファイルに含まれる *public-symbol* は *library_file* の *module_name* ですでに定義済みです。追加するためには *public-symbol* のシンボル名を変更する必要があります。

8.7.4 ワーニングメッセージ

210 Overwriting existing file in directory

コピー(*)，抽出(&)の操作の時，指定したファイルがすでにライブラリ中に存在します。

この時 LIBU8 はファイルを上書きし，前のファイルは消えてしまいます。

226 Ignoring module not in library

このエラーはモジュールの置換(%)のときだけ発生します。指定したモジュールがライブラリ中に存在しませんでした。

229 Ignoring output library file specification

入力ライブラリファイルが書き換わらないのに，出力ライブラリファイルの指定をしました。出力ライブラリファイルの指定は無視されます。次のような場合にこのメッセージが表示されます。

1. 入力ライブラリファイルが存在しなかった時。（新規ライブラリ作成時）
2. 操作を指定しなかった時。
3. 操作がコピー(*)だけだった時。

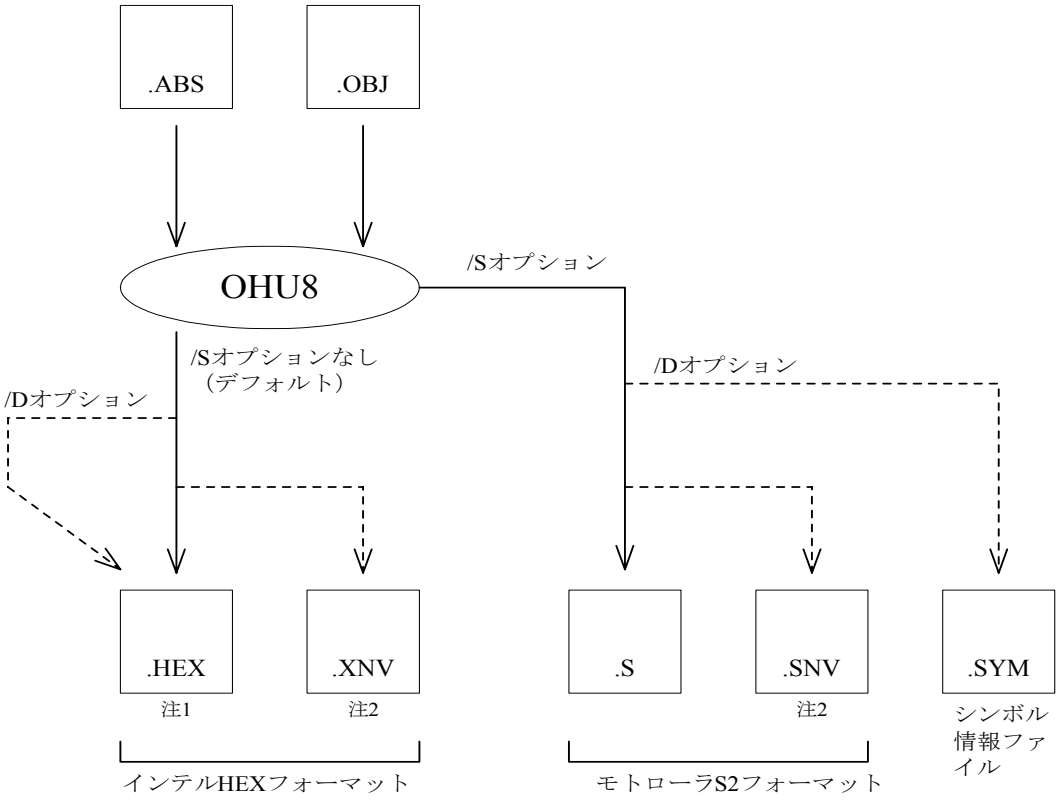
9 OHU8

9.1 概要

オブジェクトコンバータ OHU8 は，アブソリュートオブジェクトファイルをインテル HEX フォーマットまたはモトローラ S2 フォーマットに変換するソフトウェアです。OHU8 で変換可能なアブソリュートオブジェクトファイルは，リロケータブルな情報を持たないオブジェクトファイルです。リロケータブルな情報を持たないオブジェクトファイルは，通常リンカ RLU8 によって作成された.ABS ファイルを示します。ただし，アセンブラ RASU8 によって作成された.OBJ ファイルも，リロケータブルな情報を持たない.OBJ ファイルであれば，OHU8 を使って変換することが可能です。

OHU8 が作成する HEX ファイルのフォーマットは，インテル HEX フォーマット，またはモトローラ S2 フォーマットのいずれかです。この章では，インテル HEX フォーマットのファイルをインテル HEX ファイル，モトローラ S2 フォーマットのファイルを S2 フォーマットファイルと呼ぶことがあります。

作成された HEX ファイルはエミュレータや PROM プログラマなどで使うことができます。エミュレータでシンボリックデバッグを行う場合は，/D オプションを指定して，デバッグ情報を出力します。



注1： インテルHEXフォーマットへのコンバートでは，/Dオプション指定時のシンボル情報は，HEXファイルの先頭部に出力されます。

注2： 物理セグメント#0のデータメモリ空間に不揮発性メモリが存在し，そこにオブジェクトコードが存在した場合に生成されます。

図 9-1 OHU8 の入出力概要図

OHU8 は，オブジェクトファイルが持つさまざまな情報の中から，オブジェクトコードのみを取り出しファイルに出力します。

物理セグメント#0 のデータメモリ空間に不揮発性メモリ領域が存在する場合，物理セグメント#0 のプログラムメモリ空間とアドレスが重なってしまうため，OHU8 は物理セグメント#0 のデータメモリ空間上の不揮発性メモリに置かれたオブジェクトコードだけを別ファイルとして出力します。これより以降では EEPROM や FLASH メモリ等の不揮発性メモリのことを NVRAM と表現します。以下に OHU8 が変換するオブジェクトコードのメモリ構成図を示します。

（下図の 部分： 以下ではプログラムコードと呼びます）

物理セグメント#0 のプログラムメモリ空間に置かれたオブジェクトコードと，物理セグメント#1 以上の ROM および NVRAM に置かれたオブジェクトコードを変換します。

インテル HEX フォーマット .HEX，モトローラ S2 フォーマット .S に変換されます。

（下図の 部分： 以下では NVRAM コードと呼びます）

物理セグメント#0 のデータ空間の NVRAM に置かれたオブジェクトコードを変換します。

インテル HEX フォーマット .XNV，モトローラ S2 フォーマット .SNV に変換されます。

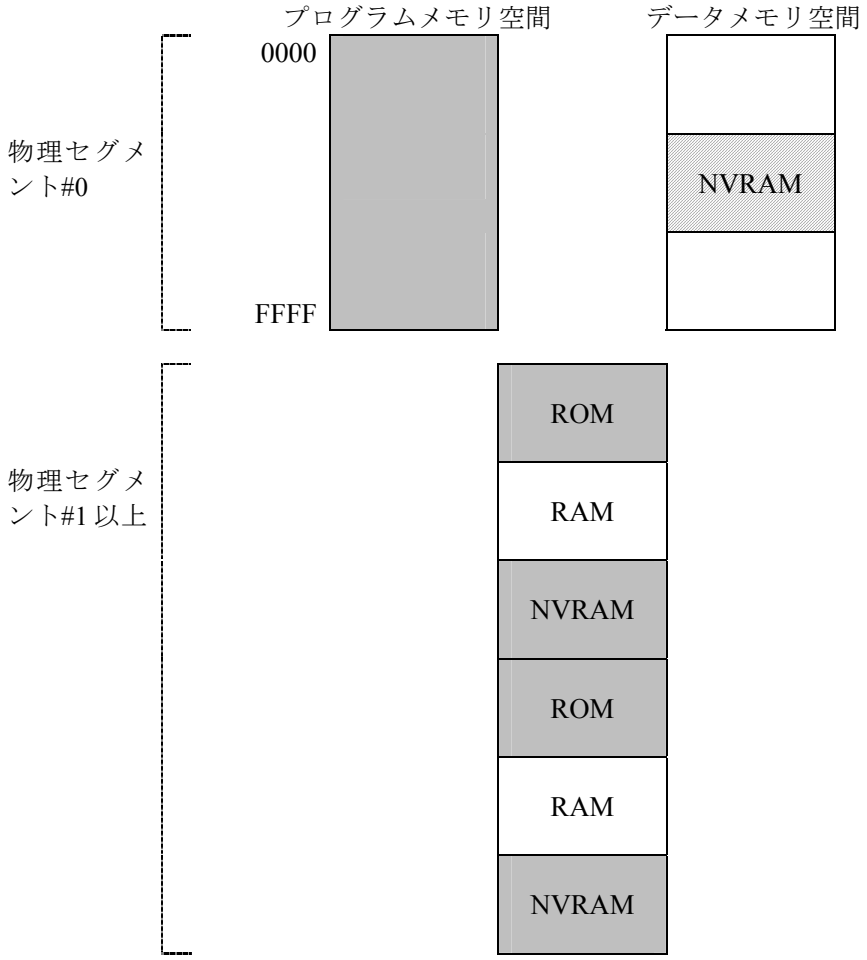


図 9-2　OHU8 が変換するオブジェクトコードのメモリ構成図

デフォルトでは，OHU8 は，これらのコードをインテル HEX フォーマットに変換します。/S オプションを指定するとモトローラ S2 フォーマットに変換します。

さらに，/D オプションを使ってデバッグ情報を出力することもできます。

これら 2 種類のオブジェクトコードとデバッグ情報の出力先は，変換フォーマットによって，次のように決定されます。

(1) オブジェクトファイルを HEX フォーマットに変換する場合

プログラムコード

入力ファイルのベース名に拡張子“.HEX”を付けたファイル（.HEX ファイル）に出力されます。

NVRAM コード

NVRAM コードが存在すれば，入力ファイルのベース名に拡張子“.XNV”を付けたファイル（.XNV ファイル）に出力されます。存在しなければ，このファイルは作成されません。

デバッグ情報

/D オプションが指定されると，“.HEX”を付けたファイルの先頭に出力されます。デバッグ情報として，パブリックシンボルを出力します。

(2) オブジェクトファイルを S2 フォーマットに変換する場合

プログラムコード

入力ファイルのベース名に拡張子“.S”を付けたファイル（.S ファイル）に出力されます。

NVRAM コード

もし存在すれば，入力ファイルのベース名に拡張子“.SNV”を付けたファイル（.SNV ファイル）に出力されます。存在しなければ，このファイルは作成されません。

デバッグ情報

/D オプションが指定されると，入力ファイルのベース名に拡張子“.SYM”を付けたファイル（.SYM ファイル）に出力されます。OHU8 は，デバッグ情報として，パブリックシンボルを出力します。

9.2 OHU8 の操作方法

OHU8 を操作する方法は次の 3 通りがあります。

(1) コマンドラインを使用した操作方法

(2) OHU8 が表示するプロンプトを使用した操作方法

(3) 応答ファイルを利用した操作方法

この節では，これらの操作方法を順番に説明します。

9.2.1 コマンドラインを使用した起動方法

OHU8 のコマンドラインの書式は，次のとおりです。

OHU8 *inputfile* [*outputfile*][;]

inputfile フィールドには変換するオブジェクトファイルの名前を指定します。*outputfile* フィールドには，出力ファイルのベース名を指定します。なお，オプションをコマンドライン上の任意の位置に指定することができます。

セミコロン（;）は入力の終わりを示します。セミコロン以降の文字はすべて無視されます。

セミコロンによって *outputfile* の指定を省略することができます。

例

```
OHU8 TEST;
```

TEST.ABS を変換して TEST.HEX を作成します。

```
OHU8 /S TEST.OBJ  ;
```

TEST.OBJ を変換して TEST.S を作成します。

各フィールドの説明は次のとおりです。

inputfile

変換するアブソリュートオブジェクトファイルの名前を指定します。この指定は省略できません。コマンドラインで *inputfile* を省略した場合は，OHU8 からプロンプトが出力されます。このプロンプトに対しても *inputfile* を指定しなければ，OHU8 はユーセージを表示して終了します。

ファイル名はパスを含めて最大 255 文字のロングファイル名を指定できます。また，拡張子は最後のドット(.)以降を拡張子とみなします。

ファイル名の拡張子が“.ABS”ならば，拡張子の指定を省略することができます。これ以外の拡張子が付いている場合，および“.ABS”の前にドット(.)が含まれている場合は拡張子の省略はできません。

```
TEST.ABS
```

“.ABS”の拡張子は省略可能です。

```
TEST.LONGNAME.ABS
```

拡張子の省略はできません。“.ABS”を省略すると，“.LONGNAME”が拡張子とみなされます。

outputfile

OHU8 によって作成されるファイルの名前です。このフィールドは省略できます。省略するときは，*outputfile* の代わりにセミコロンを指定します。たとえば次のとおりに指定します。

```
OHU8 TEST.LONGNAME.ABS ;
```

この場合，出力ファイルにはデフォルトのファイル名“TEST.LONGNAME.HEX”が与えられます。

出力ファイル名の指定にドット(.)が一つもなければ，デフォルトの拡張子である“.HEX”または“.S”が付加されます。ドットが一つ以上あれば指定されたファイル名で出力ファイルが生成されます。但し，このときも NVRAM データの出力ファイル名の拡張子は，“.XNV”または“.SNV”で固定となります。

例

```
OHU8 TEST D:
```

カレントディレクトリの TEST.ABS を変換して D:TEST.HEX を作成します。

```
OHU8 TEST     ¥DATA¥
```

TEST.ABS を変換して ¥DATA¥TEST.HEX を作成します。パスだけを指定するときは，ディレクトリ名の最後に“¥”を付けてください。

```
OHU8 TEST SAMPLE /S
```

TEST.ABS を変換して SAMPLE.S を作成します。

```
OHU8 TEST SAMPLE.HHH
```

TEST.ABS を変換して SAMPLE.HHH を作成します。NVRAM コードが存在すればデータの出力ファイル名は SAMPLE.XNV になります。

```
OHU8 TEST.ABC.DEF SAMPLE.GHI.XYZ /S
```

TEST.ABC.DEF を変換して SAMPLE.GHI.XYZ を作成します。

NVRAM コードが存在すればデータの出力ファイル名は SAMPLE.GHI.SNV になります。

9.2.2 プロンプトを使用した起動方法

OHU8 から出力されるプロンプトに対して指定を行う方法です。

OHU8 は，実行するための情報が足りないと，プロンプトを表示してユーザーの入力を促します。次にどのような場合にプロンプトが表示されるかを説明します。

DOS のプロンプトに対して OHU8 とタイプします。これだけでは入力ファイル名がわからないので，OHU8 は次のプロンプトを表示します。

```
INPUT FILE [.ABS] :
```

[.ABS]は入力ファイルの拡張子を省略すると，“.ABS”になるということを表しています。

このプロンプトに対して入力ファイル名を指定します。ファイル名の指定をせずにリターンを入力すると，OHU8 の使い方が表示されて終了します。

入力ファイル名を指定すると，次のプロンプトが表示されます。

```
OUTPUT FILE(S) [input.ext] :
```

このプロンプトにはデフォルトの出力ファイル名が［］の中に表示されています。*input* は入力ファイル名のベース名　*.ext* は出力ファイルのフォーマットに対応する拡張子です。

表示されているファイル名でよければリターンを押します。ファイル名を変更するときは改めてファイル名を指定します。出力ファイルのフォーマットを変更したい場合は，オプションを指定します。

出力ファイル名の指定が終わると，OHU8 は変換作業を開始します。

コマンドラインで入力ファイル名だけが指定されているときは，出力ファイル名のプロンプトが表示されます。

9.2.3 応答ファイルを利用した操作方法

OHU8 の起動に、応答ファイルを使うことができます。応答ファイルはロングファイル名を指定する場合などで入力タイプ数が多い場合に使用します。応答ファイルの指定は、コマンドラインでのみ有効です。プロンプトを利用した入力形式で応答ファイルを指定しても、それは応答ファイルとはみなされません。

まず、エディタを使って応答ファイルを作成します。このファイルには、コマンドラインの入力形式で必要な操作を書いておきます。一行に記述できない場合は、複数の行に分けて記述してもかまいません。OHU8 は、改行を空白として扱います。

OHU8 の応答ファイルを利用するための書式は，次のとおりです。

OHU8 [*inputfile*] *@response_file* [*outputfile*][;]

inputfile フィールドには変換するオブジェクトファイルの名前を指定します。*@response_file* には応答ファイル名を指定します。*outputfile* フィールドには，出力ファイルのベース名を指定

します。なお，オプションをコマンドライン上の任意の位置に指定することができます。

セミコロン（;）は入力の終わりを示します。セミコロン以降の文字はすべて無視されます。

セミコロンによって *outputfile* の指定を省略することができます。

例

すべての入力フィールドを複数の行に分けて記述し、応答ファイル MYOH.RES を作成します。応答ファイルには、コメントを記述することもできます。コメントは“#”または“//”で始まり、行末までとなります。

```
#####################################################
#   応答ファイル記述例
#   '#'または'//'から行末まではコメントとみなす
#####################################################
input_file          // 入力アブソリュートファイル
output_file         // 出力 HEX ファイル
/D                  // デバッグ情報出力
```

この場合は、次のようにタイプして OHU8 を起動します。

```
OHU8 @MYOH.RES
```

また、次のように入力フィールドの一部だけを応答ファイルに記述することもできます。

例

出力 HEX ファイルとデバッグ情報出力オプションについて記述し、応答ファイル MYHEX.RES を作成します。

```
output_file /D
```

この場合は、次のようにタイプして OHU8 を起動します。

```
OHU8 INPUT_FILE @MYHEX.RES
```

このコマンドは，上記の OHU8 @MYOH と同じ動作をします。

9.3 OHU8 が出力するメッセージ

ここでは，OHU8 によって出力されるメッセージについて説明します。メッセージには，OHU8 の起動または終了を表すメッセージと，処理の最中に何らかの異常があったことを表すエラーメッセージがあります。

OHU8 が出力するメッセージはすべて，標準出力デバイスに送られます。デフォルトでは，このデバイスは画面ですが，メッセージをファイルやプリンタなどのデバイスにリダイレクトすることもできます。

9.3.1 起動メッセージ

OHU8 は起動直後に，次のメッセージを画面に表示します。

```
OHU8 Object Converter, Ver.1.20
Copyright (C) 2008-2011 LAPIS Semiconductor Co., Ltd.
```

入力されたオブジェクトファイルがどのユーティリティによって作成されたかが判別できれば，OHU8 は次のメッセージを表示します。

```
Object was created by translator
```

translator　　：オブジェクトファイルを生成したユーティリティの名前

RASU8，RLU8 のいずれかが表示されます。

もし，入力オブジェクトファイルが何によって作成されたかが判別できなかった場合，OHU8 は次のメッセージを表示します。

```
Undefined translation id
```

9.3.2 終了メッセージ

/R オプションが指定された場合は，実際に変換した開始アドレスと終了アドレスを表示します。

```
Converted range = a:bbbb - c:dddd
```

a:bbbb　　：実際に変換した開始アドレス

c:dddd　　：実際に変換した終了アドレス

正常終了時には次のメッセージを表示します。

```
Convert End.
```

エラー発生時には，次のメッセージを表示します。

```
Error : error message
File Offset : offset
```

error message　　：エラーの内容を表すメッセージ
offset　　：エラーの発生したレコードの位置を示すオフセット

9.3.3 終了コード

OHU8 は終了コードを返します。正常に終了した場合は 0，フェイタルエラーにより終了した場合には 3 を返します。それ以外の値は返しません。

終了コード	終了時の状態	意味
0	正常終了	エラーの発生がありませんでした。
3	フェイタルエラー	実行中に問題が発生し操作を続行できませんでした。

9.4 オプション

OHU8 はオプションを指定することにより，インテル HEX フォーマットとモトローラ S2 フォーマットの 2 種類の内，どちらかの出力フォーマットを選択することができます。

またエミュレータでシンボリックデバッグを行うためのデバッグ情報出力の選択，実際に変換したいアドレス範囲指定などを行うことができます。

オプションは，セミコロンの前であれば，どの位置でも指定することができます。OHU8 はオプションの大文字，小文字を区別しません，たとえば /D オプションを /d と指定することもできます。

複数のオプションを指定する場合，各オプションの間をスペースで区切る必要はありません。

OHU8 で使用するオプションを次に示します。

/H

出力ファイルのフォーマットをインテル HEX フォーマットにします。

本オプションを省略した場合でもデフォルトでインテル HEX フォーマットになります。

インテル HEX フォーマットの場合はコードが 0H:0H～0FH:0FFFFH までの範囲でなければなりません。この範囲を超えている場合は，OHU8 はフェイタルエラーにより強制終了します。

/S または/HS

出力ファイルのフォーマットをモトローラ S2 フォーマットにします。

/D

デバッグ情報を作成します。デバッグ情報は出力ファイルのフォーマットによって以下のように出力されます。

インテル HEX フォーマット …… .HEX ファイルの先頭にデバッグ情報が出力されます。

モトローラ S2 フォーマット …… .SYM ファイルにデバッグ情報が出力されます。

入力ファイルがデバッグ情報を持っていない場合，出力ファイルにはデバッグ情報終了レコードだけが出力されます。

/R(*start_addr*, *end_addr*)

プログラムコードの出力範囲を指定します。

start_addr と *end_addr* で指定した範囲のプログラムコードのみを，インテル HEX フォーマットまたはモトローラ S2 フォーマットのファイルに出力します。

start_addr と *end_addr* の値は 4K バイト単位のアドレスで指定します。物理セグメントアドレスはオフセットアドレスの前にコロンを付加して指定します。省略時の物理セグメント

は#0 となります。

例　`/R(8000h, 1:7fffh)`

0:8000h から 1:7FFFh までの 64K の範囲のプログラムコードを出力します

start_addr と *end_addr* の値が 4K バイト単位のアドレスでなかった場合，OHU8 は 4K バイト単位のアドレスに補正します。

例　`/R(2:2222h, 3:3333h)`

2:2222h は 2:2000h に，3:3333h は 3:3FFFh に補正します。

start_addr および *end_addr*（補正があったら補正されたアドレス）がコンテントレコードのオブジェクトデータならびの途中であった場合，*start_addr* から *end_addr* までを出力します。

/R オプションは２回以上指定することはできますが，その場合は最初の指定だけが有効となり，２回目以降の指定は無視されます。また，本オプションはプログラムコードに対して範囲を指定するものとし，NVRAM コードについては，存在すれば/R で指定された範囲に関係なくすべて別ファイルに出力されます。

9.5 OHU8 で使うファイル

OHU8 の入出力するファイルの種類とその内容について説明します。

9.5.1 入力ファイル

OHU8 へは次の 2 種類のファイルが入力できます。

1. RASU8 によって作成されたリロケータブルな情報の含まれていないオブジェクトファイル。
2. RLU8 によって作成されたアブソリュートオブジェクトファイル。

入力ファイルにはリロケータブルな情報があってはいけません。もしあればエラーになります。OHU8 は変換作業中に，入力ファイルがどのソフトウェアによって作成されたか画面に表示します。

9.5.2 出力ファイル

OHU8 が作成する HEX ファイルは，インテル HEX フォーマットと S2 フォーマットの 2 種類です。

インテル HEX ファイルはインテル HEX フォーマットのファイルです。S2 フォーマットファイルはモトローラ S2 フォーマットのファイルです。

NVRAM コードファイルは，物理セグメント#0 のデータメモリ空間の不揮発性メモリ領域に置かれるオブジェクトコードがオブジェクトファイル中に含まれていた場合だけ作成されます。プログラムコードファイルは必ず作成されます。

エミュレータでシンボリックデバッグを行うためには，OHU8 を起動するときに/D オプションを指定してデバッグ情報を出力します。デバッグ情報は，インテル HEX ファイルでも S2 フォーマットファイルでも同じフォーマットですが，出力ファイルのフォーマットによってそれを書き込む場所が異なります。インテル HEX フォーマットでは拡張子を“.HEX”とするプログラムコードファイルの先頭に，モトローラ S2 フォーマットでは拡張子を“.SYM”とする独立したファイルに，デバッグ情報を書き込みます。

各 HEX ファイルおよびデバッグ情報のフォーマットを以下に示します。最初にファイルの構成を示し，次に各レコードの説明を行います。レコードの説明では，まず出力例を使って各フィールドがどのように構成されているかを表し，その次に各フィールドの説明を行います。

9.5.3 インテル HEX ファイル

インテル HEX フォーマットは，図 9-3 に示すように，コードセグメントレコード，データレコード，ファイル終了レコードの 3 種類のレコードから構成されています。

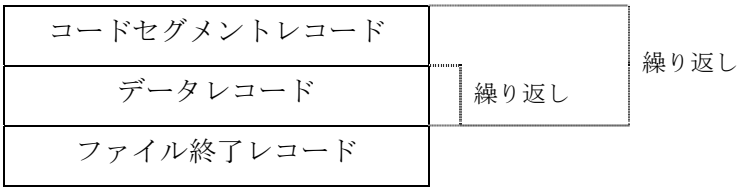


図 9-3　インテル HEX フォーマットの構成

注意

通常のインテル HEX フォーマットでは物理セグメントアドレスを表現できない（0FFFFH 以上のアドレスを表すことができない）ため，拡張インテル HEX フォーマットを採用します。

ただし，拡張インテル HEX フォーマットのコードセグメントレコードで表現できる物理セグメントアドレスの範囲は 0H～0FH までのため，0FH:0FFFFH を超える空間を持つファイルは変換できません。

コードセグメントレコードは，物理セグメントアドレスを持つレコードで，物理セグメントアドレスが変化するごとに出力されます。このレコードは，後に続くデータレコードを，どの物理セグメントに配置するかを指示するために出力されます。

コードセグメントレコードが出現しなければ，物理セグメントアドレスは #0 として扱われます。

データレコードは，ROM または不揮発性メモリ領域に配置されるオブジェクトコードを持つレコードです。

ファイル終了レコードはファイルの終了を示すためのもので，ファイルの最後に 1 度だけ出力されます。

各レコードのフォーマットは，次のとおりです。

9.5.3.1 コードセグメントレコード

```
: 02 0000 02 F000 0C
```

REC_MARK　REC_LEN　LOAD_ADR　REC_TYP　DATA　CHK_SUM

フィールド	説明
REC_MARK	文字“:”。
REC_LEN	“02”固定。
LOAD_ADR	“0000”固定。
REC_TYP	“02”固定。コードセグメントレコードを表します。
DATA	物理セグメントアドレスが最上位 4 ビットに入ります。下位 12 ビットは“000”固定となります。“0000”～“F000”の範囲を取ります。

フィールド	説明
CHK_SUM	チェックサム。 REC_LEN, LOAD_ADR, REC_TYP, DATA フィールドの文字列を 1 バイト（2 文字 1 組）ごとに区切り，全バイトの総和を計算します。その結果の 2 の補数の最下位バイトがチェックサムとなります。

9.5.3.2 データレコード

```
: 10 0000 00 000102030405060708090A0B0C0D0E0F 78
```

REC_MARK REC_LEN LOAD_ADR REC_TYP DATA CHK_SUM

フィールド	説明
REC_MARK	文字“:”。
REC_LEN	DATA フィールドに格納されるオブジェクトコードのバイト数を示します。
LOAD_ADR	DATA フィールドの先頭に格納されているオブジェクトコードがロードされるアドレス。
REC_TYP	“00”固定。データレコードを表します。
DATA	オブジェクトコードが入るフィールド。
CHK_SUM	チェックサム。

9.5.3.3 ファイル終了レコード

```
: 00 0000 01 FF
```

REC_MARK REC_LEN LOAD_ADR REC_TYP CHK_SUM

フィールド	説明
REC_MARK	文字“:”。
REC_LEN	“00”固定。
LOAD_ADR	“0000”固定。
REC_TYP	“01”固定。ファイル終了レコードを表します。
CHK_SUM	“FF”固定。

9.5.4 モトローラ S2 フォーマット

モトローラ S2 フォーマットは，図 9-4 に示すように，S0，S2，S8 レコードの 3 種類のレコードから構成されています。

図 9-4　モトローラ S2 フォーマットの構成

S0 レコードは ファイルの先頭に 1 度だけ出力されます。 S2 レコードは，ROM 空間または NVRAM 領域に配置されるオブジェクトコードを持つレコードです。 S8 レコードは，ファイルの最後に 1 度だけ出力されます。

9.5.4.1 S0 レコード

```
S0 0E 0000 5538204445425547474552 FF
```

REC_TYP REC_LEN LOAD_ADR DATA CHK_SUM

フィールド	説明
REC_TYP	“S0”固定。
REC_LEN	REC_LEN フィールドの次のフィールドから，CHK_SUM フィールドまでを，1 バイト単位（2 文字 1 組）で数えた値。S0 レコードでは，この後の LOAD_ADR，DATA，CHK_SUM フィールドが固定なので，“0E”固定になります。
LOAD_ADR	“0000”固定。
DATA	定数 “5538204445425547474552”。この DATA は U8 開発環境ツール間で使用するものでユーザプログラムとは一切関係ありません。
CHK_SUM	REC_LEN フィールドから CHK_SUM フィールドの直前までを，1 バイト（2 文字 1 組）ごとに区切り，すべて加算します。その値の 1 の補数の最下位バイト値。すべての値が固定されているため“FF”固定になります。

9.5.4.2 S2 レコード

```
S2 14 000000 000102030405060708090A0B0C0D0E0F 73
```

REC_TYP REC_LEN LOAD_ADR DATA CHK_SUM

フィールド	説明
REC_TYP	“S2”固定。
REC_LEN	算出方法は S0 レコードと同じ。
LOAD_ADR	DATA フィールドの先頭に格納されているオブジェクトコードがロードされるアドレス。
DATA	オブジェクトコードが入ります。
CHK_SUM	算出方法は S0 レコードと同じ。

9.5.4.3 S8 レコード

```
S8 04 000000 FB
```

REC_TYP REC_LEN LOAD_ADR CHK_SUM

フィールド	説明
REC_TYP	“S8”固定。
REC_LEN	算出方法は S0 レコードと同じ。ただし S8 レコードは，LOAD_ADR，CHK_SUM のフィールドが固定されているため，REC_LEN は“04”固定になります。
LOAD_ADR	“000000”固定。
CHK_SUM	算出方法は S0 レコードと同じ。ただし OHU8 が作成する S8 レコードは，REC_LEN， LOAD_ADR フィールドの値が固定されているため，CHK_SUM は“FB”固定になります。

9.6 デバッグ情報

シンボリックデバッグをおこなうために必要な情報も一定のフォーマットで表されます。これをデバッグ情報といいます。

デバッグ情報は，図 9-5 に示すように，デバッグシンボルレコード，デバッグ情報終了レコードから構成されます。

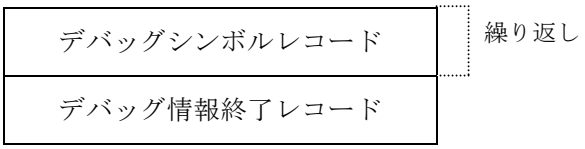


図 9-5　デバッグ情報の構成

デバッグシンボルレコードは，変換するモジュールに含まれるパブリックシンボルの情報を持ちます。

デバッグ情報終了レコードは，デバッグ情報の終了を示します。

デバッグ情報のフォーマットは，コード情報のフォーマットに依存しません。

注意

ここで説明するデバッグ情報のフォーマットは，アセンブリ言語で記述されたプログラムをデバッグするためのものです。したがってこのフォーマットでは，C ソースレベルのデバッグ情報を表すことはできません。

9.6.1 デバッグシンボルレコード

```
0 DEBUGSYM 80H 0 C
```

REC_MARK SYMBOL VALUE SEG USAGE

フィールド	説明
REC_MARK	文字“0”　デバッグシンボルレコードを表します。
SYMBOL	モジュール内のパブリックシンボルです。
VALUE	シンボルの持つ値。16 進数で表されます。 NUMBER 以外のアドレスシンボルの場合は，物理セグメント内のオフセットアドレスを表します（ビットアドレスの場合は最大 7FFFFH）。
SEG	物理セグメントアドレス。0～FFH までの 16 進数で表されます。NUMBER シンボルの場合，この値は 0 になります。
USAGE	シンボルの持つユーセージタイプを表します。 C　: CODE D　: DATA B　: BIT ND : NVDATA NB : NVBIT T　: TABLE TB : TBIT N　: NUMBER NO : NONE

※各フィールドの間には 1 つのスペース（20H）が入ります。

9.6.2 デバッグ情報終了レコード

__ $

デバッグ情報の終わりを示すレコードです。このレコードはスペース（20H）と“$”（24H）で構成されます。

9.7 入出力ファイル例

ここではソースファイルが実際どのように変換されるのか例を使って説明します。始めにソースファイル TEST.ASM を/D オプションを指定して次のようにアセンブルし，オブジェクトファイル TEST.OBJ を作成します。

```
RASU8 TEST /D
```

ソースファイル：TEST.ASM

```
TYPE    (M610001)
        MODEL   LARGE
        ROMWINDOW   0,3FFFH
NUM     EQU     100H
    TSEG
CODE_SYM1:
        DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
        CSEG    AT      1:0000H
CODE_SYM2:
        DW      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
TBIT_SYM1       TBIT    3000H.0
TBIT_SYM2       TBIT    3000H.1
        DSEG AT 0E000H
DATA_SYM1:
        DS      2
        DSEG    AT      1:9000H
DATA_SYM2:
        DS      1
        BSEG AT 0E100H.0
BIT_SYM1:
        DBIT    1
        BSEG    AT      1:0A000H.0
BIT_SYM2:
        DBIT    1
        NVSEG
EDATA_SYM1:
        DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
EDATA_SYM2:
        DW      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
        NVBSEG
EBIT_SYM1:
        DBIT    1
EBIT_SYM2:
        DBIT    1
        PUBLIC  NUM CODE_SYM1 TBIT_SYM1 TBIT_SYM2 DATA_SYM1 DATA_SYM2
        PUBLIC  EDATA_SYM1 EDATA_SYM2 EBIT_SYM1 EBIT_SYM2 BIT_SYM1
        PUBLIC  BIT_SYM1 BIT_SYM2
     END
```

作成された TEST.OBJ を，OHU8 で 2 種類の HEX ファイルに変換する例を示します。

```
OHU8 TEST.OBJ   /D;
```

この手順で，次のデバッグ情報付きのインテル HEX ファイル TEST.HEX と TEST.XNV が作成されます。

デバッグ情報付きインテル HEX ファイル（プログラムコードファイル）：TEST.HEX

```
0 TBIT_SYM1 18000H 0 TB
0 TBIT_SYM2 18001H 0 TB
0 EDATA_SYM1 8000H 0 ND
0 EDATA_SYM2 8010H 0 ND
0 CODE_SYM1 0H 0 T
0 EBIT_SYM1 40000H 0 NB
0 EBIT_SYM2 40001H 0 NB
0 NUM 100H 0 N
0 BIT_SYM1 70800H 0 B
0 BIT_SYM2 50000H 1 B
0 DATA_SYM1 E000H 0 D
0 DATA_SYM2 9000H 1 D
 $
:100000000001020304050607080 90A0B0C0D0E0F78
:020000021000EC
:10000000000001000200030004000500060007 00D4
:10001000080009000A000B000C000D000E000F0084
:00000001FF
```

インテル HEX ファイル（NVRAM コードファイル）：TEST.XNV

```
:10800000000102030405060708090A0B0C0D0E0FF8
:10801000000001000200030004000500060007 0044
:10802000080009000A000B000C000D000E000F00F4
:00000001FF
```

```
OHU8  TEST.OBJ  /S  /D;
```

S2 フォーマットファイル TEST.S と TEST.SNV，そしてパブリックシンボルだけのデバッグ情報ファイル TEST.SYM が作成されます。

S2 フォーマットファイル（プログラムコードファイル）：TEST.S

```
S00F00004F4B492C30312C30302C55383B
S214000000000102030405060708090A0B0C0D0E0F73
S214010000000001000200030004000500060007 00CE
S214010010080009000A000B000C000D000E000F007E
S804000000FB
```

S2 フォーマットファイル（NVRAM コードファイル）：TEST.SNV

```
S00F00004F4B492C30312C30302C55383B
S214008000000102030405060708090A0B0C0D0E0FF3
S214008010000001000200030004000500060007003F
S214008020080009000A000B000C000D000E000F00EF
S804000000FB
```

デバッグ情報ファイル（パブリックシンボルのみ）：TEST.SYM

```
0 TBIT_SYM1 18000H 0 TB
0 TBIT_SYM2 18001H 0 TB
0 EDATA_SYM1 8000H 0 ND
0 EDATA_SYM2 8010H 0 ND
0 CODE_SYM1 0H 0 T
0 EBIT_SYM1 40000H 0 NB
0 EBIT_SYM2 40001H 0 NB
0 NUM 100H 0 N
0 BIT_SYM1 70800H 0 B
0 BIT_SYM2 50000H 1 B
0 DATA_SYM1 E000H 0 D
0 DATA_SYM2 9000H 1 D
 $
```

9.8 テンポラリファイル

OHU8 は変換作業のために，最大 3 つのテンポラリファイル“$_$”，“$_ _$”，“$_ _ _$”を使用します。使用するテンポラリファイルの数は出力ファイルのフォーマットによって変わります。

OHU8 は入力ファイルからデータを読み込みます。そして変換したデータをテンポラリファイルに出力します。変換が正常に終了した後，テンポラリファイルを出力ファイルに書き出します。終了時にテンポラリファイルは削除されます。

テンポラリファイルはカレントディレクトリに作成されます。このため，同名のファイルをカレントディレクトリに置かないようにしてください。

9.9 エラーメッセージ

OHU8 によって生成されるエラーはすべてフェイタルエラーです。変換作業は中止され，出力ファイルは作成されません。

9.9.1 エラーメッセージの書式

エラーメッセージのフォーマットは 2 種類に分けることができます。変換中のエラーと，それ以外のエラーです。

１．変換中のエラーのフォーマット

Error *error_code*:*error_message*

File Offset *hhhhhhhhH(ddddddd)*

２．それ以外のエラーのフォーマット

Error : *error_code*: *error_message*

上記のフォーマットで使用している表記の説明を以下に示します。

マニュアルでの表記	画面での表示
hhhhhhhhH	エラーが起きた位置を示すファイルのオフセット（16 進）。
(*ddddddd*)	その 10 進表示。
error_code	エラーコード。
error_message	エラーの状態を示すメッセージ。

以下にエラーコード，エラーメッセージとその説明を示します。エラーメッセージはエラーコード順に並べてあります。

001 Bad syntax on command line

コマンドラインの指定が間違っています。

002 Unable to open OHU8 temporary file

テンポラリファイルがオープンできません。

003 Checksum failure

入力ファイルのチェックサムファイルが異常です。

004 Command option duplicated

コマンドラインで同じオプションが 2 回指定されました。

006 File not absolute

入力ファイルにリロケータブルな情報があります。RASU8 で作成した.OBJ.ファイルを指定していた場合は，RLU8 を使って.ABS ファイルを作成し，そのファイルを指定するようにしてください。

007 File offset error

入力ファイルから読み込んだレコードのファイルオフセットを得ることができませんでした。

008 File read error

入力ファイルを読み込んでいるときにエラーが発生しました。

009 File remove error

テンポラリファイルの削除時にエラーが発生しました。

010 Illegal command option

コマンドラインで指定したオプションが間違っています。

011 Input file not specified

入力ファイル名の指定が間違っています。入力ファイル名を指定するフィールドにファイル名以外のものが指定されました。

012 Insufficient disk space

出力ファイルをディスクに書き込む時にディスクの容量が足りないため書き込みが完了できませんでした。

013 Insufficient memory

作業用のメモリが足りません。

014 Invalid family id

入力ファイルは OHU8 用に作られたものではありません。OHU8 で変換できるファイルは RASU8 または RLU8 が作成したアブソリュートオブジェクトファイルです。

015 Invalid object module

入力ファイルが間違っています。

016 Invalid record type

入力ファイルに OHU8 が認識できないレコードタイプがあります。入力ファイルを作成した RASU8，RLU8 は OHU8 と同時に提供されたプログラムか確認してください。同じパッ

ケージで提供されたプログラムをお使いください。

017 Invalid segment type

入力ファイルに入っているデバッグ情報のシンボルに，サポートされていないセグメントタイプがありました。

018 Invalid target

入力ファイルは汎用モジュール用として作成されたものです。

019 Invalid version number

オブジェクトファイルを作成したソフトウェアのバージョンが異常な値を示しています。

020 I/O error

ファイルのクローズ中にエラーが発生しました。

021 Unable to use Intel HEX format, so specify /S

入力ファイルに 0H:0H～0FH:0FFFFH の範囲を超えるオブジェクトコードが含まれています。このファイルをインテル HEX フォーマットに変換することはできません。この場合は，/S オプションを指定してモトローラ S2 フォーマットファイルに変換してください。

023 Bad constant, illegal character

/R オプション指定時に使用できない文字が使用されています。

024 Bad constant, out of range

/R オプション指定時に 0H:0H～0FH:0FFFFH の範囲を超えて指定されています。

025 Invalid conversion range

/R オプション指定時の範囲指定が不正です。

10 オーバーレイ機能

10.1 概要

オーバーレイとは，プログラムの一部をいくつかのプログラム単位（これをオーバーレイユニットといいます）に分割して，それをデータメモリ空間上の記憶領域に置いておき，オーバーレイユニットのうちのひとつを，必要なときにプログラムメモリ空間上に確保された書き換え可能な特定の領域（これをオーバーレイ領域といいます）にロードして実行することをいいます。

オーバーレイの概念図を以下に示します。

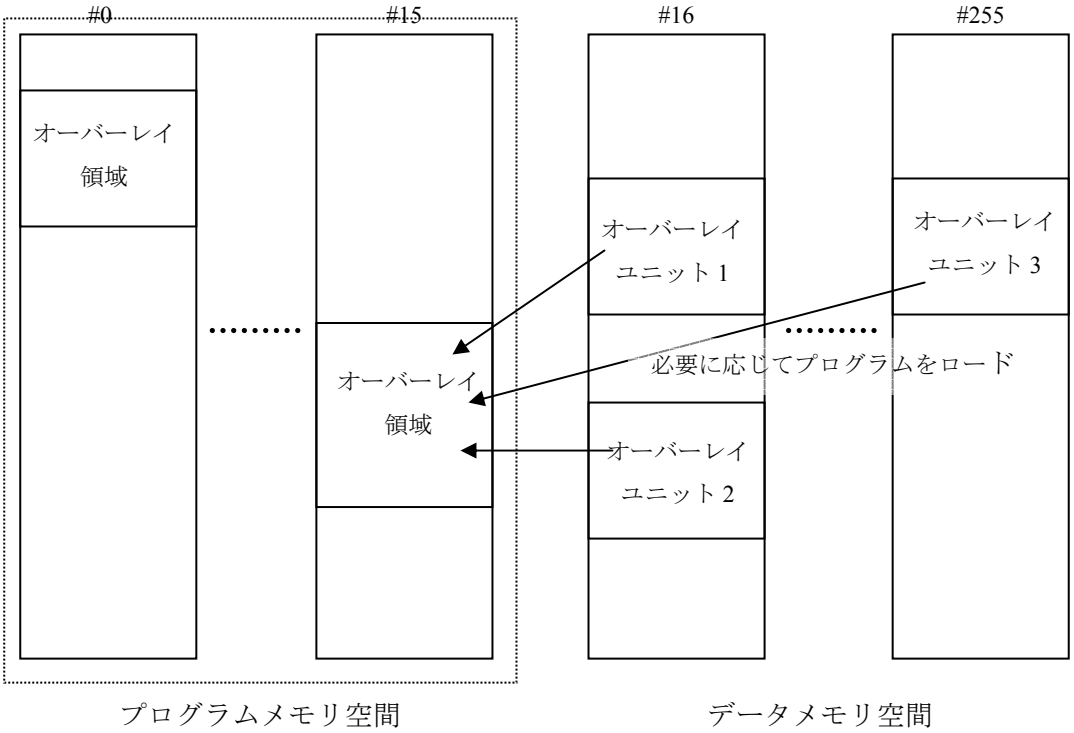


図 10-1 オーバーレイの概念図

オーバーレイ領域は，物理セグメント#0 から#15 までのいずれかのプログラムメモリ空間に配置します。オーバーレイ領域は、書き換え可能な領域でなければならないため，RAM か不揮発性メモリ上に配置されなければなりません。

オーバーレイユニットは、1 つ以上の CODE セグメントから構成されます。そして、各オーバーレイユニットをデータメモリ空間上に配置します。上の概念図では，オーバーレイユニットを物理セグメント#16 以上に配置していますが，物理セグメント#1 以上のデータメモリ空間上であれば，オーバーレイユニットはどこにでも配置できます。ただし，オーバーレイユニットを配置することができるメモリの種類は，ROM または不揮発性メモリに限定されます。

実際にオーバーレイユニットを構成するプログラムを実行する場合には、あらかじめオーバーレイユニットをオーバーレイ領域にロードしておかなければなりません。オーバーレイユニットをロードするプログラム（ローダ）は，ユーザが作成する必要があります。

10.1.1 実配置アドレスと実行時アドレス

先に述べたように，オーバーレイユニットを構成するプログラムはデータメモリ空間上に配置されます。そして，プログラム実行時にプログラムメモリ空間上に配置されたオーバーレイ領域にロードされ，プログラムが実行されます。

プログラムが実際のメモリ上に配置されるアドレスのことを，実配置アドレスといいます。プログラム実行時のアドレスのことを実行時アドレスといいます。

通常のプログラムは，実配置アドレスと実行時アドレスは一致するものですが，オーバーレイユニットを構成するプログラムの場合，実配置アドレスと実行時アドレスは異なることになります。

プログラムの呼び出しなどに参照されるラベルなどは，プログラム実行時に参照されるものであるため，シンボルのアドレス値は実行時アドレスとして算出されます。ただし，実際のオブジェクトファイルに出力されるオブジェクトコードの割り付けアドレスは，実配置アドレスとして算出されます。

```
    CSEG AT 1:8000H OVL 16:1000H
Overlay_func:
    MOV     ER0,    #00H
    .
    .
    .
    RT
```

上の例ではオーバーレイ用のアブソリュート CODE セグメントを定義しています。この場合，ラベルとして定義されている Overlay_func には，実行時アドレスとして 1:8000H が割り当てられます。そして，実際のオブジェクトコードは物理セグメント#16 のオフセットアドレス 1000H 番地を開始アドレスとする領域に配置されます。

10.1.2 オーバーレイ領域の制限

オーバーレイ領域は，プログラムの実行，およびプログラムコードの読み書きが可能な領域でなければなりません。

オーバーレイ領域として使用できる領域は，次の領域に限られます。

メモリモデル	オーバーレイ領域として使用できる領域
スモールモデル	物理セグメント#0 のプログラムメモリ空間上の RAM または不揮発性メモリが実装される領域
ラージモデル	物理セグメント#0 から#15 までのプログラムメモリ空間上の RAM または不揮発性メモリが実装される領域

10.1.3 オーバーレイユニットの配置可能領域

オーバーレイユニットは，1 つ以上の CODE セグメントから構成されます。通常の CODE セグメントは，プログラムメモリ空間上に配置されなければなりませんが，オーバーレイ用の CODE セグメントの場合，プログラムが実行されるときに対象のプログラムがオーバーレイ領域にロードされていればよいので，必ずしもプログラムメモリ空間上に配置される必要はありません。

したがって，通常の CODE セグメントと，オーバーレイ用の CODE セグメントの配置可能領域には，次のような違いがあります。

CODE セグメントの種類	CODE セグメントの配置可能領域
通常の CODE セグメント	スモールモデルの場合は，物理セグメント#0 のプログラムメモリ空間上の ROM または不揮発性メモリが実装される領域。 ラージモデルの場合は，物理セグメント#0～#15 のプログラムメモリ空間上の ROM または不揮発性メモリが実装される領域。
オーバーレイ用の CODE セグメント	物理セグメント#0 のプログラムメモリ空間上の ROM または不揮発性メモリが実装される領域。または，物理セグメント#1～#255 のデータメモリ空間上の ROM または不揮発性メモリが実装される領域。 リロケータブル CODE セグメントにおいては，/CODE オプションで割り付けアドレスを指定しない限り，物理セグメント#1～#255 のデータメモリ空間上の ROM または不揮発性メモリが実装される領域に割り付けられます。

リロケータブル CODE セグメントが不揮発性メモリの実装領域に割り付けられるのは，セグメントシンボル定義時に特殊領域属性（*relocation_attr*）として NVRAM を指定したものに限られます。ただし，/CODE オプションを使用してアブソリュートアドレスを指定した場合には，この限りではありません。

10.2 オーバーレイユニットの作成方法

オーバーレイユニットを作成する場合，アブソリュートセグメントで構成する方法と，リロケータブルセグメントで構成する方法があります。

10.2.1 アブソリュートセグメントで構成する方法

オーバーレイユニットをアブソリュートセグメントで構成する場合，次のようにオーバーレイ用のアブソリュートセグメントを記述します。

構文

CSEG [#*pseg_addr*] AT *overlay_address* OVL *allocation_address*

説明

オーバーレイ用のアブソリュートセグメントを定義する場合には，OVL 記述子を指定します。OVL 記述子を指定できるのは，CSEG 擬似命令を使用してアブソリュート CODE セグメントを定義する場合だけです。*overlay_address* には実行時のアドレスを指定します。*allocation_address* には実配置アドレスを指定します。#*pseg_addr* は実行時の物理セグメントアドレスを指定します。

OVL *allocation_address* を指定する場合，#*pseg_addr* または AT *overlay_address* のどちらかで必ず物理セグメントアドレスを指定する必要があります。

オーバーレイ領域については，DSEG 擬似命令，または NVSEG 擬似命令を用いて，オーバーレイ用のプログラムをロードするのに十分な領域を確保する必要があります。

```
;    オーバーレイ用 CODE セグメントの開始
     CSEG AT 1:8000H OVL 10H:1000H
Overlay_func:
     MOV     ER0,    #00H
     .
     .
     .
     RT

;    オーバーレイ領域の確保
     DSEG AT 1:8000H
Overlay_area:
     DS      1000H
```

上の例では，オーバーレイ用のプログラムを 1:8000H から実行することを宣言しています。プログラム自体は，10H:1000H を開始アドレスとする領域に配置されます。オーバーレイ領域は DSEG 擬似命令を用いて確保しています。

対象のプログラムは，プログラムを実行するまでにオーバーレイ領域に読み込んでおく必要があります。

10.2.2 リロケータブルセグメントで構成する方法

オーバーレイユニットをリロケータブルセグメントで構成する場合，RLU8 のオプションを使用して定義します。

RLU8 のオーバーレイ指定オプションは，次のとおりです。

構文

/OVERLAY(*area_name*, *start_address*, *end_address*){[*overlay_unit* [*overlay_unit* …]]}

説明

/OVERLAY オプションは，オーバーレイ領域の割り付けられる範囲，およびオーバーレイ領域に割り付けられるオーバーレイユニット，そしてオーバーレイユニットを構成するリロケータブルセグメントを指定します。

area_name には，オーバーレイ領域の名前を指定します。オーバーレイの領域名はどのような名前でもかまいません。*start_address* にはオーバーレイ領域の開始アドレスを，*end_address* にはオーバーレイ領域の終了アドレスを指定します。*overlay_unit* には，オーバーレイユニットを構成するリロケータブルセグメントを指定します。*overlay_unit* は次のフォーマットで指定します。

overlay_unit の構文

UNIT(*segment* [*segment* …])

オーバーレイユニットの指定は UNIT で始まり，( )で括られる部分にオーバーレイユニットを構成するリロケータブルセグメントの名前を列挙します。各リロケータブルセグメントの指定は空白で区切ります。なお，リロケータブルセグメントは，列挙した順にオーバーレイ領域へ割り付けられます。

例

```
#オーバーレイ指定用応答ファイル(LINK.RES)
/OVERLAY(OVL1, 1:8000H, 1:9FFFH)
{
    UNIT(segA segB)       // オーバーレイユニット 1
    UNIT(segC segD)       // オーバーレイユニット 2
    UNIT(segE segF segG) // オーバーレイユニット 3
}
/CODE(10H:0H segA segB segC segD segE segF segG)
# EOF
```

この例では，/OVERLAY オプションを RLU8 の応答ファイル LINK.RES に記述しています。このオプションの指定で，オーバーレイ領域 OVL1 の範囲は 1:8000H～1:9FFFH として定義されます。そして，3 つのオーバーレイユニットが定義されます。また，/OVERLAY オプションで指定したオーバーレイ用のリロケータブル CODE セグメントに対し，/CODE オプションを使用して実配置アドレスを指定しています。

この応答ファイルを RLU8 起動時に指定します。

```
RLU8 OVLSAMPLE @LINK.RES ;
```

オーバーレイ領域，およびオーバーレイユニットのプログラム実行時の割り付けイメージは，次のようになります。

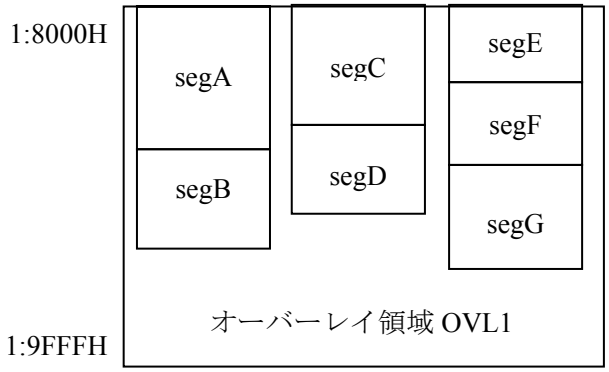


図 10-2 オーバーレイ領域へのオーバーレイユニットの割り付けイメージ

そして，オーバーレイ領域 OVL1 に対する各セグメントの割り付け情報は， RLU8 の出力するマップファイルに，次のように出力されます。

```
Link Map - Overlay Area "OVL1" (01:8000 - 01:9FFF)

      Type     Start      Stop       Size              Name
     -------------------------------------------------------
      S CODE   01:8000    01:89FF    0A00(2560)        segA
      S CODE   01:8A00    01:9133    0734(1844)        segB
 ---
      S CODE   01:8000    01:8965    0966(2406)        segC
      S CODE   01:8966    01:8F9D    0638(1592)        segD
 ---
      S CODE   01:8000    01:8595    0596(1430)        segE
      S CODE   01:8596    01:8D2B    0796(1942)        segF
      S CODE   01:8D2C    01:9583    0858(2136)        segG
```

オーバーレイ領域へのセグメントの割り付け情報は，それぞれのオーバーレイユニットごとに“---”で区切られてマップファイルに出力されます。

各セグメントの開始アドレス，および終了アドレスは，実行時アドレスを示します。

実配置アドレスに対するセグメントの割り付け情報は，次のようにマップファイルに出力されます。

```
      Type     Start      Stop       Size              Name
     -------------------------------------------------------
>GAP<          01:0000.0  01:7FFF.7  8000.0(32768.0)    (RAM)
      Q OVRLAY 01:8000    01:9FFF    2000(8192)         (OVL1)
 ---
      S CODE   10:0000    10:09FF    0A00(2560)         segA *
      S CODE   10:0A00    10:1133    0734(1844)         segB *
      S CODE   10:1134    10:1A99    0966(2406)         segC *
      S CODE   10:1A9A    10:20D1    0638(1592)         segD *
      S CODE   10:20D2    10:2667    0596(1430)         segE *
      S CODE   10:2668    10:2DFD    0796(1942)         segF *
      S CODE   10:2DFE    10:3655    0858(2136)         segG *
```

この例は，物理セグメント#1 以上のメモリ空間に対するセグメント割り付け情報の抜粋です。

/OVERLAY オプションで指定されたオーバーレイ領域“OVL1”は，擬似セグメントとしてマップファイルに出力されます。/CODE オプションで指定されたリロケータブルセグメント segA，segB，segC，segD，segE，segF，segG は，物理セグメント#16 のオフセット 0 番地以降のメモリ空間に割り付けられています。それぞれのセグメントの右側にアスタリスク（*）が付いていますが，これは，それらのセグメントがオーバーレイ用のセグメントであることを示しています。

10.3 オーバーレイローダ

オーバーレイローダは，ユーザが作成する必要があります。もっとも単純なローダは，実配置アドレスに割り付けられたオブジェクトコードを，オーバーレイ領域にコピーするというものです。

実配置アドレスに割り付けられたオブジェクトコードを，オーバーレイ領域にコピーするためには，コピー元のアドレスとコピー先のアドレスを取得しなくてはなりません。

通常，シンボルを参照する場合，そのシンボルの持つアドレスは実行時アドレスを示します。そこで，実配置アドレスを取得するための演算子として，次の演算子を用意しています。

演算子	意味
OVL_ADDRESS *symbol*	セグメント *symbol* の実配置アドレスをアドレス型で取り出します。
OVL_SEG *symbol*	セグメント *symbol* の実配置アドレスの物理セグメントアドレスを数値型で取り出します。
OVL_OFFSET *symbol*	セグメント *symbol* の実配置アドレスのオフセットアドレスを数値型で取り出します。

ここでは，オーバーレイローダに汎用性を持たせるため，オーバーレイ用のプログラムのコピー元，およびコピー先のアドレスをテーブルとして用意し，そのテーブルを参照してオーバーレイ用のプログラムをロードする例を示します。

まず，オーバーレイ用の CODE セグメント Unit1Seg があったとします。このとき，オーバーレイローダ用のテーブルを，次のように定義します。

```
;オーバーレイローダ用テーブル
UnitAdrTable    segment table word any
        rseg    UnitAdrTable
Unit1p:
        dw      offset      Unit1Seg    ;実行時オフセットアドレス
        dw      ovl_offset  Unit1Seg    ;実配置オフセットアドレス
        dw      size        Unit1Seg    ;セグメントサイズ
        db      seg         Unit1Seg    ;実行時物理セグメントアドレス
        db      ovl_seg     Unit1Seg    ;実配置物理セグメントアドレス
```

この例では，オーバーレイ用の演算子 OVL_OFFSET と OVL_SEG を使用して，オーバーレイ用 CODE セグメント Unit1Seg の実配置アドレスを取得しています。

このオーバーレイローダ用のテーブルから，コピー元のアドレスとコピー先のアドレスを取り出して，コピー処理を行うローダプログラムの例を以下に示します。

```
;オーバーレイローダ サンプルプログラム
; OvlsegLoad のパラメータ
; er0 : オーバーレイローダ用テーブルのオフセットアドレス
; r2  : オーバーレイローダ用テーブルの物理セグメントアドレス
;
OverlayLoader   segment code  any
        rseg    OverlayLoader
OvlsegLoad:
        push    qr0
        push    qr8
        l       er4,    r2:[er0]   ;実行時オフセットアドレス
        l       er6,    r2:2[er0]  ;実配置オフセットアドレス
        l       er8,    r2:4[er0]  ;オーバーレイセグメントサイズ
        l       er10,   r2:6[er0]  ;実行時と実配置の物理セグメントアドレス
copy_loop:
        cmp     r8,     #0
        cmpc    r9,     #0
        beq     copy_end
        l       er12,   r11:[er6]
        st      er12,   r10:[er4]
        add     er4,    #2
        add     er6,    #2
        add     er8,    #-2
        bal     copy_loop
copy_end:
        pop     qr8
        pop     qr0
        rt
public  OvlsegLoad
```

このローダプログラムを利用して，実際にオーバーレイ用のプログラムをロードする場合は，次のように記述します。

```
;オーバーレイ用プログラム Unit1Seg のロード
        mov     r0,     #byte1 offset Unit1p
        mov     r1,     #byte2 offset Unit1p
        mov     r2,     #seg Unit1p
        bl      OvlsegLoad
```

ここまでで，オーバーレイ用のプログラムのロードが行われます。オーバーレイ用のプログラムのロードが完了すれば，あとは通常のプログラムと扱いは同じです。ただし，これはオーバーレイユニットが 1 つのセグメントから構成される場合です。オーバーレイユニットが複数の CODE セグメントで構成される場合は，同じようなロード処理を行う必要があります。

以上のことを考慮すると，オーバーレイユニットを構成する CODE セグメントは，できるだ

け 1 つにまとめるのがよいと思われます。オーバーレイユニットを構成する CODE セグメントが複数のファイルに分かれるときには，セグメントシンボルを同じにしてパーシャルセグメントとして定義するのがよいでしょう。

11 アブソリュート
リスティング機能

11.1 概要

アブソリュートリスティング機能とは，不確定なマシンコードやアドレス情報をまったく含まないプリントファイル，すなわちすべての情報が確定しているプリントファイル（これをアブソリュートプリントファイルと呼びます）を生成する機能のことをいいます。

プログラムを複数のモジュールごとに分割した場合や，リロケータブルセグメントを使用した場合，通常のプリントファイルには不確定なマシンコード情報やアドレス情報が含まれることになります。したがって，プリントファイルを使ってデバッグ作業を行う場合に，RLU8 の出力するマップファイル中に含まれるシンボル情報などと見合わせながら作業を進めなければならないため，大変面倒です。

MACU8 アセンブラパッケージでは，この問題の解決策として一度アセンブルされたプログラムのリンク情報をもとに再アセンブルすることによって，不確定な情報を含まないアブソリュートプリントファイルを作成する機能をサポートしています。

11.2 アブソリュートプリントファイルの作成手順

ここでは，例として FOO1.ASM，FOO2.ASM，および FOO3.ASM の 3 つのファイルで構成されるプログラムのアブソリュートプリントファイルを作る手順を説明します。

最初に通常のアセンブルを実行します。

```
RASU8 FOO1
RASU8 FOO2
RASU8 FOO3
```

3 つのオブジェクトファイルをリンクします。このとき/A オプションを指定します。

```
RLU8 FOO1 FOO2 FOO3 /A;
```

このとき RLU8 は，FOO1.ABL という名前のファイルを作成します。このファイルには，シンボルの絶対アドレス情報や，確定マシンコード情報が含まれています。このファイルを“ABL ファイル”と呼びます。

ここで再びアセンブルします。このとき，アセンブラのオプションとして“/AFOO1”を指定します。/A の後には，RLU8 が作成した ABL ファイル名を指定します。拡張子.ABL は省略してもかまいません。したがって，この例の場合は，/AFOO1 を指定します。

```
RASU8 FOO1 /AFOO1
RASU8 FOO2 /AFOO1
RASU8 FOO3 /AFOO1
```

再アセンブルの結果，FOO1.APR，FOO2.APR，および FOO3.APR という拡張子.APR のファイルが作成されます。このファイルがアブソリュートプリントファイルです。

この処理過程を，図で表すと次のようになります。

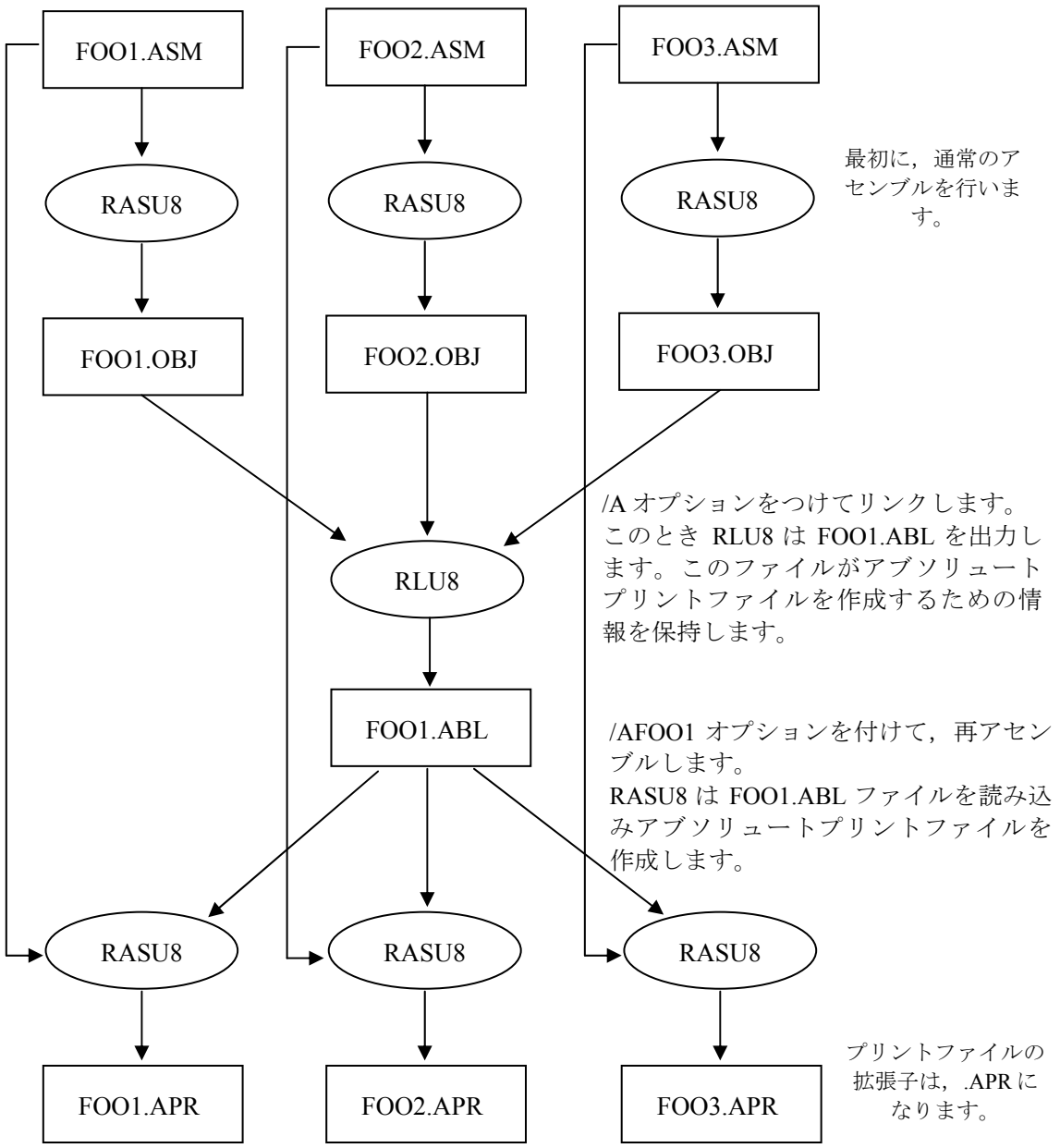


図 11-1 アブソリュートプリントファイルの生成処理の流れ

11.3 リンク時に指定するオプション

アブソリュートプリントファイルを作成する場合は，リンク時に/A オプションを指定します。/A オプションを指定してリンクを行った場合，RLU8 は ABL ファイルと呼ばれる，シンボルの絶対アドレスや確定マシンコードの情報を持つファイルを作成します。

RLU8 の/A オプションの構文は次のとおりです。

構文

/A[(*abl_file*)]

ここで，*abl_file* には，作成する ABL ファイルの名前を指定します。abl_file を省略した場合，ABL ファイル名はアブソリュートオブジェクト（ABS）ファイル名の拡張子を“.ABL”にしたものになります。また拡張子を省略した場合は，拡張子は“.ABL”となります。

例

```
RLU8 FILE1 FILE2 FILE3 /A;
```

abl_file を省略した場合，ABL ファイル名は，FILE1.ABL となります。

例

```
RLU8 FILE1 FILE2 FILE3 /A(PRNDATA);
```

abl_file の拡張子を省略した場合，ABL ファイル名は，PRNDATA.ABL となります。

例

```
RLU8 FILE1 FILE2 FILE3 /A(PRNDATA.DAT);
```

abl_file をフルネームで指定した場合，ABL ファイル名は，PRNDATA.DAT となります。

11.4 再アセンブル時に指定するオプション

アブソリュートプリントファイルを作成するために再アセンブルする場合は，RASU8 のオプションとして，/A オプションを追加指定してください。

RASU8 の/A オプションの構文は次のとおりです。

構文

/A[*abl_file*]

ここで *abl_file* には，RLU8 が生成した ABL ファイルの名前を指定します。/A と *abl_file* の間に空白を入れてはいけません。*abl_file* を省略した場合，ABL ファイル名はソースファイル名の拡張子を.ABL にしたものになります。また拡張子を省略した場合は，拡張子は.ABL となります。

/A オプションを指定して RASU8 を起動した場合，RASU8 は ABL ファイルを読み込み，アブソリュートプリントファイルを作成します。

アブソリュートプリントファイル名は，ソースファイル名の拡張子を.APR にしたものになります。アブソリュートプリントファイルの名前は，/PR オプションにより変更することができます。

再アセンブル処理は，通常のアセンブル処理と比べて次のような動作上の違いがあります。

(1) オブジェクトファイルは作成しません。/O オプションおよび OBJ 擬似命令は無効となります。

(2) C ソースレベルデバッグ情報に関する処理は行いません。/SD オプションは無効になります。

(3) EXTRN 宣言ファイルは作成しません。/X オプションは無効になります。

(4) /NPR オプションまたは NOPRN 擬似命令が指定されていたとしても，アブソリュートプリントファイルは作成されます。

無効となるオプションや擬似命令は，指定しても無視されるだけで，指定してはいけないというわけではありません。

通常アセンブル時と再アセンブル時の起動オプションの指定のうち，必ず同じ指定をする必要のあるものを次に示します。

オプション	機能
/MS, /ML	メモリモデル指定。 メモリモデルの指定はなるべく MODEL 擬似命令で行うことをお薦めします。
/DN, /DF	データモデル指定。 データモデルの指定はなるべく MODEL 擬似命令で行うことをお薦めします。
/CD, /NCD	シンボルの大文字と小文字の区別。

オプション	機能
/I*include_path*	インクルードファイルのパス指定。

以上のことから，アブソリュートプリントファイルを作成する場合は，最初のアセンブルのときの起動オプションをそのまま残して，/A オプションを追加することをお薦めします。

例えば，最初のアセンブル時のオプション指定が次のような場合，

```
RASU8 FOO1 /D /R /PW120 /NPR /T4 /IHEADER
```

再アセンブル時のオプション指定は，次のように/A オプションを追加します。

```
RASU8 FOO1 /D /R /PW120 /NPR /T4 /IHEADER /AFOO1
```

11.5 再アセンブル時のエラー

通常のアセンブル処理時にはエラーが発生しなかったプログラムでも，/A オプションを指定して再アセンブルした場合に，エラーやワーニングが発生することがあります。これは，通常のアセンブル処理では，アドレスの確定していないオペランドに対してはエラーチェックを行わないのに対して，再アセンブル処理では，すべてのオペランドに対してエラーチェックを行うことが可能になるためです。

次のような例を考えてみます。

```
EXTRN DATA:DATA_TBL
    L       ER0,    DATA_TBL
```

通常のアセンブル処理では，上の命令文はエラーになりません。ただし，これはエラーが無いのではなく，DATA_TBL のアドレスが不明であるために通常のアセンブル時にはエラーチェックができなかったにすぎません。

ここで，仮に DATA_TBL のアドレスが 8001H のように奇数アドレスであった場合，このアクセスはワードバウンダリエラーです。RLU8 でリンクしてみると，次のようなメッセージが表示されます。ここで，ソースファイル名は foo1.asm，ワーニングの対象となる命令のアドレスが 200H 番地であると仮定します。

```
Warning W006: Cannot access high byte, 0200/(absolute)/foo1
```

エラーが存在することは確認できました。しかし，実際のソースプログラム上におけるエラーの箇所は，プログラマが RLU8 のエラーメッセージに含まれるアドレスを元に捜さなければなりません。これは，プログラムが大規模になるにつれて大変面倒な作業となります。

アブソリュートリスティング機能は，このような場合にも利用することができます。上記のソースプログラムを，/A オプションを付けて再アセンブルします。すると今度は，

```
foo1.asm(206):206: Warning 28 : cannot access high byte
```

と RLU8 が表示した同じ内容のエラーメッセージを表示します。これにより，プログラマはエラーの正確な位置（行番号）を知ることができます。

以上のように，アブソリュートリスティング機能は，リンク時に発生したアドレッシング関連のワーニングの，ソースプログラム上の正確な位置を知るという目的でも使用することができます。

ただし，どのようなエラーに対しても再アセンブルが有効なわけではありません。再アセンブルしてもかまわないリンクエラーは，ワーニングに限られます。ワーニング以外のエラーが発生したときに再アセンブルを行った場合，正しいアブソリュートプリントファイルを得ることはできません。この場合，再アセンブルの途中で，フェイタルエラーを起こす場合もあります。

11.6 アブソリュートプリントファイルの出力例

アブソリュートプリントファイルのフォーマットは，アセンブル時に出力されるプリントファイルのフォーマットと同じです。プリントファイルのフォーマットについては、「6.7.1 1 アセンブリリストの読み方」を参照してください。

なお，参照されなかった関数・テーブルは，リンカによりリンク対象から除外されます。そのような関数・テーブルに対して，ロケーションフィールドはアスタリスク（**:****）で表示されます。

以下にアブソリュートプリントファイルの出力例を示します。説明の便宜上アブソリュートプリントファイルの左側に番号を付けています。

```
    ## Loc. Object                      Line   Source Statements
                                          :
                                          :
      --------------------------          14       rseg $$ref_func$tapr
                                          15
 ┌─ 00:00A8                             16   _ref_func   :
 │                                        17
 │                                        18   ;;{
 │                                        19
 │                                        20   ;;    var += 10;
(1) 00:00A8 12-90 FE-E7                   21       l   er0,    NEAR _var
 │  00:00AC 8A E0                        22       add er0,    #10
 │  00:00AE 13-90 FE-E7                  23       st  er0,    NEAR _var
 │                                        24
 │                                        25   ;;}
 └─ 00:00B2 1F-FE                        26       rt
                                          27
                                          28
      --------------------------          29       rseg $$noref_func$tapr
                                          30
 ┌─ **:****                             31   _noref_func :
 │                                        32
 │                                        33   ;;{
 │                                        34
 │                                        35   ;;    var -= 10;
(2) **:**** 12-90 FE-E7                   36       l   er0,    NEAR _var
 │  **:**** F6 E0                        37       add er0,    #-10
 │  **:**** 13-90 FE-E7                  38       st  er0,    NEAR _var
 │                                        39
 │                                        40   ;;}
 └─ **:**** 1F-FE                        41       rt
                                          42
                                          :
                                          :
```

上の例では，(1)がリンクされた関数に対する出力例を示しており、(2)がリンクされなかった関数に対する出力例を示しています。

11.7 Fatal Error 11 が発生した場合には

再アセンブルを行ったときに，次のフェイタルエラーメッセージが表示される場合があります。

```
MACRO.asm 29 : Fatal Error 11: illegal reading binary file :
ABL file : error_message
```

このエラーの原因は ABL ファイルの内容に問題がある場合がほとんどです。このエラーが発生した場合，まずは次のことを確認してください。

(1) 最初のアセンブルでエラー（ワーニングは除く）は発生していませんか。

(2) 最初のアセンブルと再アセンブルで

メモリモデル指定（/MS，/ML，/DN，/DF オプションまたは MODEL 擬似命令）

シンボルの大文字小文字の区別の指定（/CD，/NCD オプション）

インクルードパス指定（/I*include_path* オプション）

は完全に一致していますか。

(3) リンク時にワーニング以外のエラーは発生していませんか。

(4) リンクするときに/A オプションを指定していますか。

これらのことを確認した上で，なおエラーが発生する場合は，弊社までご連絡ください。

付録

付録 A　擬似命令一覧

以下に擬似命令の一覧を示します。

擬似命令	構文 機能
TYPE	TYPE (*dcl_name*) ターゲットデバイスを指定する。
ROMWINDOW	ROMWINDOW *base_address*, *end_address* ROM WINDOW 領域範囲を指定する。
NOROMWIN	NOROMWIN ROM WINDOW 領域を使用しないことを指定する。
MODEL	MODEL *memory_model* [, *data_model*]　又は MODEL *data_model* [, *memory_model*] メモリモデルを指定する。
END	END プログラムの終了を宣言する。
EQU SET	*symbol* EQU *simple_expression* *symbol* SET *simple_expression* 汎用的なローカルシンボルを定義する。
CODE	*symbol* CODE *simple_expression* CODE 型のローカルシンボルを定義する。
TABLE	*symbol* TABLE *simple_expression* TABLE 型のローカルシンボルを定義する。
TBIT	*symbol* TBIT *simple_expression* TBIT 型のローカルシンボルを定義する。
DATA	*symbol* DATA *simple_expression* DATA 型のローカルシンボルを定義する。
BIT	*symbol* BIT *simple_expression* BIT 型のローカルシンボルを定義する。
NVDATA	*symbol* NVDATA *simple_expression* NVDATA 型のローカルシンボルを定義する。

擬似命令	構文 機能
NVBIT	*symbol* NVBIT *simple_expression* NVBIT 型のローカルシンボルを定義する。
SEGMENT	*segment_symbol* SEGMENT *segment_type* [*boundary_attr*] [*seg_attr*] [*relocation_attr*] リロケータブルセグメントを定義する。
STACKSEG	STACKSEG *stack_size* スタックセグメントを定義する。
CSEG	CSEG [#*pseg_addr*][AT *start_address*] アブソリュート CODE セグメントに切替える。
TSEG	TSEG [#*pseg_addr*][AT *start_address*] アブソリュート TABLE セグメントに切替える。
DSEG	DSEG [#*pseg_addr*][AT *start_address*] アブソリュート DATA セグメントに切替える。
BSEG	BSEG [#*pseg_addr*][AT *start_address*] アブソリュート BIT セグメントに切替える。
NVSEG	NVSEG [#*pseg_addr*][AT *start_address*] アブソリュート NVDATA セグメントに切替える。
NVBSEG	NVBSEG [#*pseg_addr*][AT *start_address*] アブソリュート NVBIT セグメントに切替える。
RSEG	RSEG *segment_symbol* リロケータブルセグメントに切替える。
ORG	ORG *address* 開始アドレスを指定する。
DS	[*label* :] DS *size* バイト単位で領域を確保する。
DBIT	[*label* :] DBIT *size* ビット単位で領域を確保する。
ALIGN	ALIGN カレントロケーションを偶数アドレスにする。

擬似命令	構文 機能
DB	[*label* :] DB { *expression* | *string_constant* | *duplicate_expression* } [, { *expression* | *string_constant* | *duplicate_expression* }] … バイト単位でコード初期化を行う。
DW	[*label* :] DW { *expression* | *duplicate_expression* } [, { *expression* | *duplicate_expression* }] … ワード単位でコード初期化を行う。
CHKDBDW	CHKDBDW DB/DW 擬似命令のチェックを行う。
NOCHKDBDW	NOCHKDBDW DB/DW 擬似命令のチェックを行わない。
GJMP	[*label* :] GJMP *symbol* 最適なジャンプ命令に変換される。
GB*cond*	[*label* :] GB*cond symbol* 最適な条件分岐命令に変換される。
EXTRN	EXTRN *usage_type* [*attribute*] : *symbol* [*symbol* …] [*usage_type* [*attribute*] : *symbol* [*symbol* …]] … 外部参照シンボルを定義する。
PUBLIC	PUBLIC *symbol* [*symbol* …] シンボルのパブリック宣言を行う。
COMM	*communal_symbol* COMM *segment_type size* [*relocation_attr*] コミュナルシンボルを定義する。
INCLUDE	INCLUDE (*include_file*) ファイルを読込む。
DEFINE	DEFINE *symbol* ”*macro_body*” マクロシンボルを定義する。

擬似命令	構文 機能

IF
IFDEF
IFNDEF
ELSE
ENDIF
	IF*xxx conditional_operand*　　(IFxxx は IF，IFDEF，IFNDEF のいずれか)
	　　true_conditional_body
	[ELSE
	　　false_conditional_body]
	ENDIF
	条件アセンブルを行う。

CFILE	CFILE *file_id total_line "filename"* C 言語のファイル情報を与える。
CFUNCTION	CFUNCTION *fn_id* C 言語の関数開始位置を示す。
CFUNCTIONEND	CFUNCTIONEND *fn_id* C 言語の関数終了位置を示す。
CARGUMENT	CARGUMENT *attrib size offset "variable_name" hierarchy* C 言語の関数引数定義情報を与える。
CBLOCK	CBLOCK *fn_id block_id c_source_line* C 言語のブロック開始位置を示す。
CBLOCKEND	CBLOCKEND *fn_id block_id c_source_line* C 言語のブロック終了位置を示す。
CLABEL	CLABEL *label_no "label_name"* C 言語のラベル定義情報を与える。
CLINE	CLINE *line_attr line_no start_column end_column* C 言語のライン番号を与える。
CLINEA	CLINEA *file_id line_attr line_no start_column end_column* C 言語のライン番号を与える。
CGLOBAL	CGLOBAL *usg_typ attrib size "variable_name" hierarchy* C 言語のグローバル変数定義情報を与える。
CSGLOBAL	CSGLOBAL *usg_typ attrib size "variable_name" hierarchy* C 言語の静的グローバル変数定義情報を与える。
CLOCAL	CLOCAL *attrib size offset block_id "variable_name" hierarchy* C 言語のローカル変数定義情報を与える。

擬似命令	構文 機能
CSLOCAL	CSLOCAL *attrib size alias_no block_id ”variable_name” hierarchy* C 言語の静的ローカル変数定義情報を与える。
CSTRUCTTAG	CSTRUCTTAG *fn_id block_id st_id total_mem total_size ”tag_name”* C 言語の構造体タグ定義情報を与える。
CSTRUCTMEM	CSTRUCTMEM *attrib size offset ”member_name” hierarchy* C 言語の構造体メンバ定義情報を与える。
CUNIONTAG	CUNIONTAG *fn_id block_id un_id total_mem total_size ”tag_name”* C 言語の共用体タグ定義情報を与える。
CUNIONMEM	CUNIONMEM *attrib size ”member_name” hierarchy* C 言語の共用対メンバ定義情報を与える。
CENUMTAG	CENUMTAG *fn_id block_id emu_id total_mem ”tag_name”* C 言語の列挙型変数の定義情報を与える。
CENUMMEM	CENUMMEM *value ”member_name”* C 言語の列挙型変数のメンバ定義情報を与える。
CTYPEDEF	CTYPEDEF *fn_id block_id attrib ”type_name” hierarchy* C 言語の typedef の定義情報を与える。
FASTFLOAT	FASTFLOAT OBJ ファイルに高速演算ライブラリをリンクする指定情報を追加する。
OBJ	OBJ [(*object_file*)] オブジェクトファイルを指定ファイルに出力する。
NOOBJ	NOOBJ オブジェクトファイルを生成しない。
PRN	PRN [(*print_file*)] プリントファイルを指定ファイルに出力する。
NOPRN	NOPRN プリントファイルを生成しない。
ERR	ERR [(*error_file*)] エラーファイルを指定ファイルに出力する。
NOERR	NOERR エラーファイルを生成しない。

擬似命令	構文 機能
DEBUG	DEBUG デバッグ情報をオブジェクトファイルに出力する。
NODEBUG	NODEBUG デバッグ情報をオブジェクトファイルに出力しない。
LIST	LIST 以降の文をアセンブルリストに出力する。
NOLIST	NOLIST 以降の文をアセンブルリストに出力しない。
SYM	SYM シンボルリストを作成する。
NOSYM	NOSYM シンボルリストを作成しない。
REF	REF 以降の文に現れるシンボルの行番号情報をクロスリファレンスリストに出力する。
NOREF	NOREF 以降の文に現れるシンボルの行番号情報をクロスリファレンスリストに出力しない。
PAGE	PAGE PAGE [*page_length*][, *page_width*] 1. プリントファイルの改ページを行う。 2. プリントファイルの 1 行の文字数と 1 ページの行数を設定する。
NOPAGE	NOPAGE プリントファイルの 1 行の文字数と 1 ページの行数の制限を解除する。
DATE	DATE ”*charcter_string*” プリントファイルの日付フィールドに入る文字列を指定する。
TITLE	TITLE ”*character_string*” プリントファイルのタイトルフィールドに入る文字列を指定する。
TAB	TAB [*tab_width*] プリントファイルのタブ幅を設定する。

擬似命令	構文 機能
NOFAR	NOFAR データメモリ空間を物理セグメント#0 のみに限定する。

付録 B　予約語一覧

以下に予約語一覧をアルファベット順に示します。ここには、それぞれの予約語の用途を記述しています。予約語の用途が 2 つ以上ある場合には、用途別に／で区切っています。

	予約語	用途	備考
A	ADD	基本命令	
	ADDC	基本命令	
	AL	BC 命令分岐条件	
	ALIGN	擬似命令	
	AND	基本命令	
B	B	基本命令	
	BAL	基本命令	
	BC	基本命令	
	BCY	基本命令	
	BEQ	基本命令	
	BGE	基本命令	
	BGES	基本命令	
	BGT	基本命令	
	BGTS	基本命令	
	BIT	擬似命令／擬似命令オペランド	
	BL	基本命令	
	BLE	基本命令	
	BLES	基本命令	
	BLT	基本命令	
	BLTS	基本命令	
	BNC	基本命令	
	BNE	基本命令	
	BNS	基本命令	
	BNV	基本命令	
	BNZ	基本命令	
	BOV	基本命令	

	予約語	用途	備考
	BP	レジスタ名	
	BPOS	演算子	
	BPS	基本命令	
	BRK	基本命令	
	BSEG	擬似命令	
	BYTE1	演算子	
	BYTE2	演算子	
	BYTE3	演算子	
	BYTE4	演算子	
	BZ	基本命令	
C	CARGUMENT	擬似命令	
	CBLOCK	擬似命令	
	CBLOCKEND	擬似命令	
	CENUMMEM	擬似命令	
	CENUMTAG	擬似命令	
	CER0	レジスタ名	
	CER2	レジスタ名	
	CER4	レジスタ名	
	CER6	レジスタ名	
	CER8	レジスタ名	
	CER10	レジスタ名	
	CER12	レジスタ名	
	CER14	レジスタ名	
	CFILE	擬似命令	
	CFUNCTION	擬似命令	
	CFUNCTIONEND	擬似命令	
	CGLOBAL	擬似命令	
	CHKDBDW	擬似命令	
	CLABEL	擬似命令	

	予約語	用途	備考
	CLINE	擬似命令	
	CLINEA	擬似命令	
	CLOCAL	擬似命令	
	CMP	基本命令	
	CMPC	基本命令	
	CODE	擬似命令／擬似命令オペランド	
	COMM	擬似命令	
	CPL	基本命令	使用不可
	CPLC	基本命令	
	CQR0	レジスタ名	
	CQR8	レジスタ名	
	CR0	レジスタ名	
	CR1	レジスタ名	
	CR2	レジスタ名	
	CR3	レジスタ名	
	CR4	レジスタ名	
	CR5	レジスタ名	
	CR6	レジスタ名	
	CR7	レジスタ名	
	CR8	レジスタ名	
	CR9	レジスタ名	
	CR10	レジスタ名	
	CR11	レジスタ名	
	CR12	レジスタ名	
	CR13	レジスタ名	
	CR14	レジスタ名	
	CR15	レジスタ名	
	CSEG	擬似命令	
	CSGLOBAL	擬似命令	

	予約語	用途	備考
	CSLOCAL	擬似命令	
	CSTRUCTMEM	擬似命令	
	CSTRUCTTAG	擬似命令	
	CTYPEDEF	擬似命令	
	CUNIONMEM	擬似命令	
	CUNIONTAG	擬似命令	
	CXR0	レジスタ名	
	CXR4	レジスタ名	
	CXR8	レジスタ名	
	CXR12	レジスタ名	
	CY	BC 命令分岐条件	
D	DAA	基本命令	
	DAS	基本命令	
	DATA	擬似命令／擬似命令オペランド	
	DATE	擬似命令	
	DB	擬似命令	
	DBIT	擬似命令	
	DEBUG	擬似命令	
	DEC	基本命令	
	DEFINE	擬似命令	
	DI	基本命令	
	DIV	基本命令	
	DS	擬似命令	
	DSEG	擬似命令	
	DSR	レジスタ名／SFR シンボル	
	DUP	擬似命令オペランド	
	DW	擬似命令	
	DYNAMIC	擬似命令オペランド	
E	EA	レジスタ名	

	予約語	用途	備考
	ECSR	レジスタ名	
	EDSR	基本命令	使用不可
	EI	基本命令	
	ELR	レジスタ名	
	ELSE	擬似命令	
	END	擬似命令	
	ENDIF	擬似命令	
	EPSW	レジスタ名	
	EQ	BC 命令分岐条件	
	EQU	擬似命令	
	ER0	レジスタ名	
	ER2	レジスタ名	
	ER4	レジスタ名	
	ER6	レジスタ名	
	ER8	レジスタ名	
	ER10	レジスタ名	
	ER12	レジスタ名	
	ER14	レジスタ名	
	ERR	擬似命令	
	EXTBW	基本命令	
	EXTRN	擬似命令	
F	FAR	アドレッシング指定子／擬似命令オペランド	
	FASTFLOAT	擬似命令	
	FP	レジスタ名	
G	GBCY	擬似命令	
	GBEQ	擬似命令	
	GBGE	擬似命令	
	GBGES	擬似命令	
	GBGT	擬似命令	

	予約語	用途	備考
	GBGTS	擬似命令	
	GBLE	擬似命令	
	GBLES	擬似命令	
	GBLT	擬似命令	
	GBLTS	擬似命令	
	GBNC	擬似命令	
	GBNE	擬似命令	
	GBNS	擬似命令	
	GBNV	擬似命令	
	GBNZ	擬似命令	
	GBOV	擬似命令	
	GBPS	擬似命令	
	GBZ	擬似命令	
	GE	BC 命令分岐条件	
	GES	BC 命令分岐条件	
	GJMP	擬似命令	
	GT	BC 命令分岐条件	
	GTS	BC 命令分岐条件	
H			
I	ICESWI	基本命令	
	IF	擬似命令	
	IFDEF	擬似命令	
	IFNDEF	擬似命令	
	INC	基本命令	
	INCLUDE	擬似命令	
J			
K			
L	L	基本命令	
	LARGE	擬似命令オペランド	

	予約語	用途	備考
	LE	BC 命令分岐条件	
	LEA	基本命令	
	LES	BC 命令分岐条件	
	LIST	擬似命令	
	LR	レジスタ名	
	LT	BC 命令分岐条件	
	LTS	BC 命令分岐条件	
M	MODEL	擬似命令	
	MOV	基本命令	
	MUL	基本命令	
N	NC	BC 命令分岐条件	
	NE	BC 命令分岐条件	
	NEAR	アドレッシング指定子／擬似命令オペランド	
	NEG	基本命令	
	NOCHKDBDW	擬似命令	
	NODEBUG	擬似命令	
	NOERR	擬似命令	
	NOFAR	擬似命令	
	NOLIST	擬似命令	
	NOOBJ	擬似命令	
	NOP	基本命令	
	NOPAGE	擬似命令	
	NOPRN	擬似命令	
	NOREF	擬似命令	
	NOROMWIN	擬似命令	
	NOSYM	擬似命令	
	NS	BC 命令分岐条件	
	NV	BC 命令分岐条件	
	NVBIT	擬似命令／擬似命令オペランド	

	予約語	用途	備考
	NVBSEG	擬似命令	
	NVDATA	擬似命令／擬似命令オペランド	
	NVRAM	擬似命令オペランド	
	NVSEG	擬似命令	
	NZ	BC 命令分岐条件	
O	OBJ	擬似命令	
	OFFSET	演算子	
	OR	基本命令	
	ORG	擬似命令	
	OV	BC 命令分岐条件	
	OVL_ADDRESS	演算子	
	OVL_OFFSET	演算子	
	OVL_SEG	演算子	
P	PAGE	擬似命令	
	PC	レジスタ名	
	POP	基本命令	
	PRN	擬似命令	
	PS	BC 命令分岐条件	
	PSW	レジスタ名	
	PUBLIC	擬似命令	
	PUSH	基本命令	
Q	QR0	レジスタ名	
	QR8	レジスタ名	
R	R0	レジスタ名	
	R1	レジスタ名	
	R2	レジスタ名	
	R3	レジスタ名	
	R4	レジスタ名	
	R5	レジスタ名	

	予約語	用途	備考
	R6	レジスタ名	
	R7	レジスタ名	
	R8	レジスタ名	
	R9	レジスタ名	
	R10	レジスタ名	
	R11	レジスタ名	
	R12	レジスタ名	
	R13	レジスタ名	
	R14	レジスタ名	
	R15	レジスタ名	
	RB	基本命令	
	RC	基本命令	
	REF	擬似命令	
	ROMWINDOW	擬似命令	
	RSEG	擬似命令	
	RT	基本命令	
	RTI	基本命令	
	RTICE	基本命令	
	RTICEPSW	基本命令	使用不可
S	SB	基本命令	
	SC	基本命令	
	SEG	演算子	
	SEGMENT	擬似命令	
	SET	擬似命令	
	SIZE	演算子	
	SLL	基本命令	
	SLLC	基本命令	
	SMALL	擬似命令オペランド	
	SP	レジスタ名	

	予約語	用途	備考
	SRA	基本命令	
	SRL	基本命令	
	SRLC	基本命令	
	ST	基本命令	
	STACKSEG	擬似命令	
	SUB	基本命令	
	SUBC	基本命令	
	SWI	基本命令	
	SYM	擬似命令	
T	TAB	擬似命令	
	TABLE	擬似命令／擬似命令オペランド	
	TB	基本命令	
	TBIT	擬似命令／擬似命令オペランド	
	TITLE	擬似命令	
	TSEG	擬似命令	
	TYPE	擬似命令	
U	UNIT	擬似命令オペランド	
V			
W	WORD	擬似命令オペランド	
	WORD1	演算子	
	WORD2	演算子	
X	XOR	基本命令	
	XR0	レジスタ名	
	XR4	レジスタ名	
	XR8	レジスタ名	
	XR12	レジスタ名	
Y			
Z	ZF	BC 命令分岐条件	

MACU8 アセンブラパッケージ
ユーザーズマニュアル
SQ003099E001

2012 年 8 月　　第 7 版発行

©2008 - 2012 LAPIS Semiconductor Co., Ltd.