User's Manual

DSPASM

FAA/GREEN_DSP 構造化アセンブラ

ユーザーズマニュアル

本資料に記載の全ての情報は本資料発行時点のものであり、ルネサス エレクトロニクスは、
予告なしに、本資料に記載した製品または仕様を変更することがあります。
ルネサス エレクトロニクスのホームページなどにより公開される最新情報をご確認ください。

ルネサス エレクトロニクス
www.renesas.com

Rev.1.06　2024.03

目次

1. 概要

このドキュメントは、FAA/GREEN DSP用構造化記述対応アセンブラ(DSPASM)の機能仕様をまとめたものです。

2.DSPASM 本体について

DSPASMは、構造化記述を用いて記述されたプログラムコードをアセンブルし、アセンブルした結果をオブジェクトファイルとして出力するプログラムです。

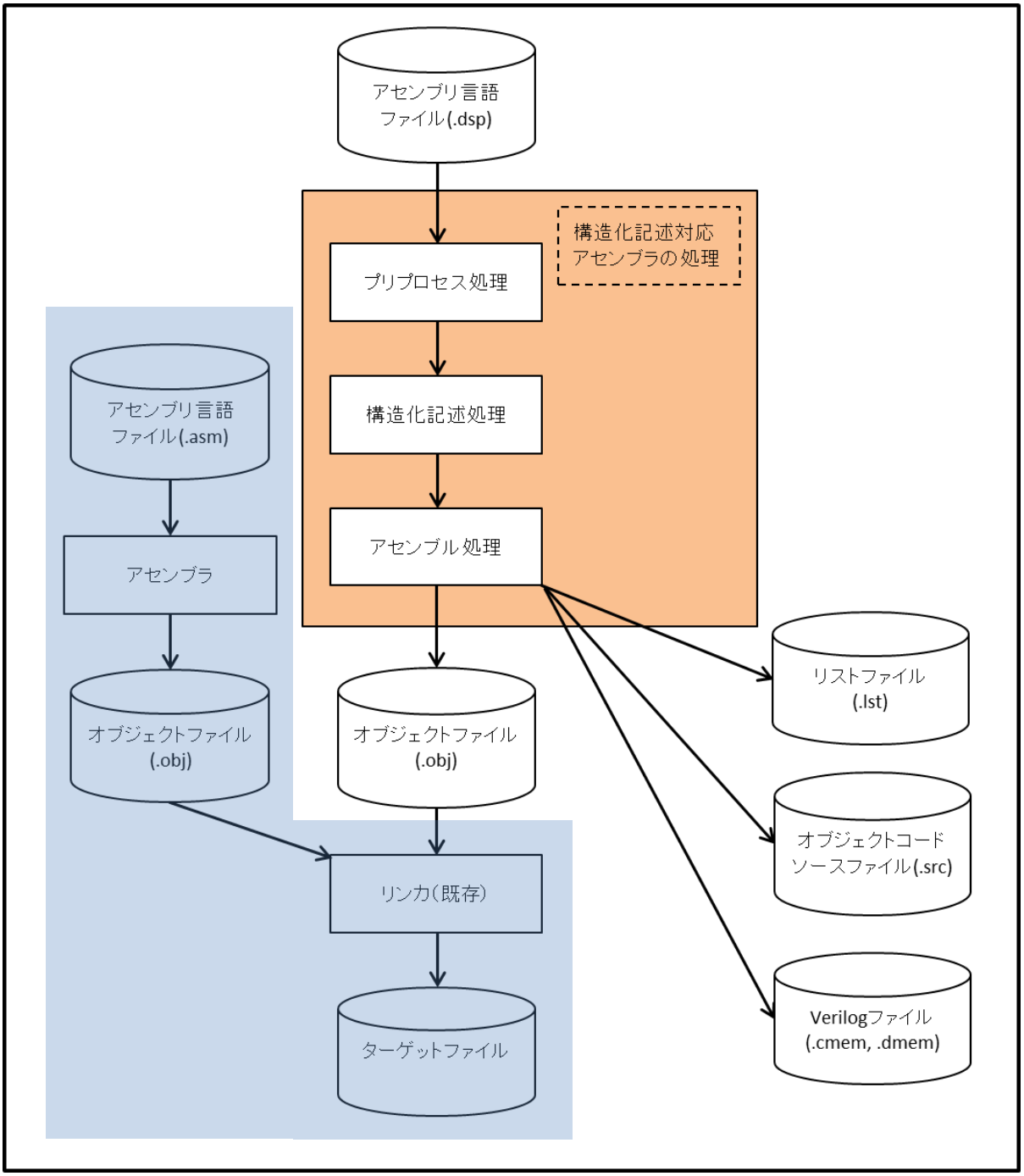

図 2.1 DSPASM 処理の流れ

2.1.動作環境

DSPASMは以下の環境で動作します。

表 2.1 DSPASMの動作環境

Microsoft Windows	Version 8.1 (32ビット/64ビット、日本語版/英語版)
	Version 10 (32ビット/64ビット、日本語版/英語版)
	Version 11 (64ビット、日本語版/英語版)

2.2.DSPASM への入力

DSPASMの入力は「アセンブリ言語ファイル」です。

「アセンブリ言語ファイル」にはアセンブル対象となるソースコードを記述します。ソースコードの記述には通常のアセンブラ記述に加えて、構造化記述を用いることができます。
また、ソースコード内にはプリプロセス命令を記述することもできます。

「アセンブリ言語ファイル」の概要は以下のとおりです。

表 2.2 アセンブリ言語ファイルの概要

アセンブリ言語ファイル: 拡張子(*.dsp)
書式
[コメント行] [プリプロセス命令] SECTION CODE [NAME コードセクション名] [LOCATE コードRAM アドレス] プログラムコードまたはコメント行 SECTION DATA [NAME データセクション名] [LOCATE データRAM アドレス] データまたはコメント行
サンプルコード

```
; Sample code
;

#include <header.h>

SECTION CODE LOCATE H'd000
      MOV #S1_ST, DP0
      MOV #S1_OUT, RP0
      ;
      MOV (DP0+), A0                ; DataB->A0
      MOV A0, R0                    ; DataB->R0
      MOV (DP0+), A0                ; DataA->A0
      ADD                           ; DataA + DataB
      MOV A0, (RP0+)
      STOP
      ;
SECTION DATA LOCATE H'c000
S1_ST:   DATA H'00000050      ; DataB
         DATA H'00000100      ; DataA
```

```
        DATA H'00000003      ; Param_M0
;
S1_OUT: DATA H'00000000
```

2.3.DSPASM の出力

DSPASMでは以下の三種類から、いずれか一つのファイルを出力することができます。

1.オブジェクトコードソースファイル
2.Verilog ファイル
3.オブジェクトファイル

出力するファイルの種類はコマンドラインオプション(-format)で指定します。
コマンドラインオプションの詳細は 2.4「DSPASMのコマンドラインオプション」 をご覧ください。

また、変換結果を確認するためにリストファイルを出力することができます。リストファイルを生成する場合はコマンドラインオプション(-list)を指定します。

各ファイルの概要は以下のとおりです。

(1) オブジェクトコードソースファイル:

表 2.3 オブジェクトコードソースファイル概要

アセンブル結果をプログラムとデータに分けて、二種類のテキストファイルとして出力します。 プログラム部を出力するファイルはファイル名の末尾に"_dspcode.src"が付加されます。 データ部を出力するファイルはファイル名の末尾に"_dspdata.src"が付加されます。
オブジェクトコードソースファイルの出力例(プログラム： *_dspcode.src)
.SECTION SAREA_DSPCODE, DATA, LOCATE=H'd000 .ORG H'd000 .DATA.B H'80 .DATA.B H'00 .DATA.B H'88 .DATA.B H'03 .DATA.B H'07 .DATA.B H'03 .DATA.B H'07 .DATA.B H'0F .DATA.B H'0D .DATA.B H'20 .END
オブジェクトコードソースファイルの出力例(データ： *_dspdata.src)
.SECTION SAREA_DSPDATA, DATA, LOCATE=H'c000 .ORG H'c000 S1_ST: .DATA.B H'00 .DATA.B H'00 .DATA.B H'00 .DATA.B H'50 .DATA.B H'00 .DATA.B H'00

```
.DATA.B  H'01
.DATA.B  H'00
.DATA.B  H'00
.DATA.B  H'00
.DATA.B  H'00
.DATA.B  H'03
S1_OUT:
.DATA.B  H'00
.DATA.B  H'00
.DATA.B  H'00
.DATA.B  H'00
.END
```

(2) Verilogファイル:

表 2.4 Verilogファイル概要

アセンブル結果をプログラムとデータに分けて、二種類のテキストファイルとして出力します。 プログラム部を出力するファイルはファイル名の拡張子が".cmem"となります。 データ部を出力するファイルはファイル名の拡張子が".dmem"となります。
Verilogファイルの出力例(プログラム： *.cmem)
@00000000 80008803 @00000001 0703070f @00000002 0d200000
Verilogファイルの出力例(データ： *.dmem)
@00000000 00000050 @00000001 00000100 @00000002 00000003 @00000003 00000000

(3) オブジェクトファイル:

表 2.5 オブジェクトファイル概要

アセンブル結果をバイナリ形式(ELF/DWARF2)で出力します。 ファイル名の拡張子は".obj"となります。

(4) リストファイル:

表 2.6 リストファイル概要

アセンブル前のソースコードと、アセンブル結果を同一のテキストファイルに出力します。 ファイル名の拡張子は".lst"となります。
リストファイル(*.lst)

```
1: .line "C:¥dspasm¥sample_code.dsp" , 1
2:
3                   : SECTION CODE
4 0000 8000         :       MOV #S1_ST, DP0
```

```
 5 0002 8803      :        MOV #S1_OUT, RP0
 6                : ;
 7                : SECTION DATA
 8 0000 00000050 : S1_ST:DATA  H'00000050
 9 0004 00000100 :           DATA  H'00000100
 10 0008 00000003 :            DATA  H'00000003
11                : ;
12 000c 00000000 : S1_OUT:DATA  H'00000000
```

2.4.DSPASM のコマンドラインオプション

DSPASMはWindowsのコマンドラインより、以下の形式で実行します（"△"は空白を意味します）。

```
dspasm△[コマンドラインオプション]△入力ソースファイル名
```

図 2.2 DSPASMコマンドラインの入力形式

入力ソースファイル名の拡張子を省略することはできません。また、入力ソースファイル名として指定できるのは一ファイルに限られます。複数の入力ソースファイルを指定した場合は、アセンブルエラーとなります。

DSPASMで指定できるコマンドラインオプションは以下のとおりです。なお、オプション指定は大文字/小文字を区別しません。

表 2.7 コマンドラインオプション一覧

コマンドラインオプション	機能
-format△出力フォーマット (出力フォーマット : ASM / VERILOG / OBJ)	アセンブラの出力フォーマットを指定します。
-list	リストファイルを出力します。
-output△フォルダ名	ファイルを出力するフォルダを指定します。
-text_macro△文字	define, ifdef などのテキストマクロを指定する際に使用する、先頭の一文字を指定します。
-define△名前#値	テキストマクロを指定します。
-allow_text_macro_redefine	-defineコマンドラインオプションや、ソースコード中のプリプロセス命令でシンボルを再定義することを許可します。
-inc_dir△フォルダ名	includeするファイルの起点となるフォルダを指定します。
-dsp△DSP種別 (DSP種別：RX_DSP / RL78_DSP / RL78_101_DSP / RL78_111_DSP / RL78_IAR_DSP / RL78_LLVM_DSP / RL78_GCC_DSP / ARM_DSP / ARM_EABI5_DSP)	コード生成の対象となるDSPを指定します。
-core_version△バージョン (バージョン：2 / 3)	コード生成の対象となるDSPコアバージョンを指定します。
-E	入力ファイルにプリプロセスを行い、その結果をファイルに出力します。
-cpuLittleEndian -cpuBigEndian	CPUのエンディアンを指定します。
-littleEndianData	アセンブル結果として出力するデータ値をリトルエンディアン形式で生成します。
-code_section_start -data_section_start	コードセクション、データセクションの割り付け開始アドレスを指定します。
-no_debug_info	オブジェクトファイル内にデバッグ情報を出力しません。

-debug_aranges-no-padding	コンバータ(renesas_cc_converter)用入力ファイルを作成する時に指定してください。
-label△属性 (属性：GLOBAL / LOCAL)	シンボルの属性を指定します。
-macro_identify△識別方法 (識別方法：FORWARD / EXACT)	テキストマクロの識別方法を指定します。
-dwarf_spec△出力設定 (出力設定 : INITIAL / GENERIC / RENESAS)	オブジェクトファイルに出力するDWARF情報の出力内容を指定します。
-code_execinstr	オブジェクトファイルの命令コードセクションに対してSHF_EXECINSTR フラグを設定します。
-code_lebal_type△コードラベル属性 (コードラベル属性：NOTYPE / FUNC) -data_label_type△データラベル属性 (データラベル属性：NOTYPE / OBJECT)	コードセクション、データセクションのラベルに付与する属性を指定します。

(1) ファイル出力に関するオプション

表 2.8 -formatコマンドラインオプション

<table>
<tr><td colspan="2">出力ファイル指定： -format</td></tr>
<tr><td>書式</td><td>-format△出力フォーマット

出力フォーマットは以下のいずれか：
<table><tr><td>ASM</td><td>オブジェクトコードソースファイルを出力します</td></tr><tr><td>VERILOG</td><td>Verilogファイルを出力します。</td></tr><tr><td>OBJ</td><td>オブジェクトファイルを出力します。</td></tr></table>
出力フォーマットの指定が無い場合、アセンブルエラーとなります。</td></tr>
<tr><td>備考</td><td>アセンブラの出力フォーマットを指定します。

このオプションを指定しなかった場合、アセンブルエラーとなります。
このオプションが複数回指定された場合は、最後に指定した内容が有効になります。</td></tr>
<tr><td>例</td><td>dspasm -format OBJ a.dsp</td></tr>
</table>

表 2.9 -listコマンドラインオプション

リストファイル出力指定： -list	
書式	-list
備考	リストファイルを出力します。

表 2.10 -outputコマンドラインオプション

出力フォルダ指定： -output	
書式	-output△フォルダ名 フォルダ名の指定が無い場合、アセンブルエラーとなります。
備考	ファイルを出力するフォルダを指定します。 このオプションを指定しなかった場合、コマンドラインからdspasm.exeを起動したフォルダにファイルを出力します。 このオプションが複数回指定された場合は、最後に指定した内容が有効になります。
例	dspasm -format OBJ -output .¥tmpdir a.dsp

(2) プリプロセス処理に関するオプション

表 2.11 -text_macro コマンドラインオプション

テキストマクロの先頭文字指定： -text_macro	
書式	-text_macro△文字 使用可能な文字は以下の5つのうちのいずれか： # ' ` @ _ 文字の指定が無い場合、アセンブルエラーとなります。
備考	define, ifdef などのテキストマクロを指定する際に使用する、先頭の一文字を指定します。 使用可能な文字以外の文字を指定した場合は、アセンブルエラーとなります。 このオプションを指定しなかった場合、'#' が指定されたものとみなします。 このオプションが複数回指定された場合は、最後に指定した内容が有効になります。
例	dspasm -format OBJ -text_macro @ a.dsp

表 2.12 -defineコマンドラインオプション

テキストマクロ指定： -define	
書式	-define△名前#値 「名前#値」の指定が無い場合、アセンブルエラーとなります。
備考	テキストマクロを指定します。 置き換え対象となる文字列を "#" の前に記述し、置き換え後の文字列を "#" の後に記述します。 複数のテキストマクロを指定する場合は、このオプションを必要なだけ繰返します。 置き換え対象となる文字列と置き換えた後の文字列は、強制的に大文字として処理されます（大文字小文字は区別されません）。 "#" 以降の値は省略可能です。この場合、置き換え後の文字列は空白となります。 オプションに "#" を二つ以上記述した場合は、最も左に記述した "#" を区切り文字とみなします。
例	dspasm -format OBJ -define AAA#5 a.dsp

表 2.13 -allow_text_macro_redefineコマンドラインオプション

テキストマクロ再定義許可： -allow_text_macro_redefine	
書式	-allow_text_macro_redefine

備考	-defineコマンドラインオプションや、ソースコード中のプリプロセス命令でシンボルを再定義することを許可します。
	このオプションを指定しなかった場合、シンボルの再定義を許可しません(アセンブルエラーが発生します)。
	`#define REG_R R0`
	`#define REG_R R1 ; -allow_text_macro_redefineオプション指定時は後に出現した`
	`; #define REG_R R1 の定義が有効になります。`
	`; 指定されていない場合は再定義が許可されないため、アセンブルエラーとなります。`
	`MOV A0, #REG_R ; -allow_text_macro_redefineオプション指定時は MOV A0, R1 に置き換わります。`

表 2.14 -inc_dirコマンドラインオプション

includeフォルダ指定： -inc_dir	
書式	-inc_dir△フォルダ名 フォルダ名の指定が無い場合、アセンブルエラーとなります。
備考	includeするファイルの起点となるフォルダを指定します。 このオプションを指定しなかった場合、アセンブリ言語ファイルが格納されているフォルダを相対パスの起点とみなします。 このオプションは複数回指定することができます。この場合、オプションで指定された順にincludeするファイルを検索します。
例	dspasm -format OBJ -inc_dir .¥abc a.dsp

(3) コード生成に関するオプション

表 2.15 -dspコマンドラインオプション

<table>
<tr><td colspan="2">DSP指定： -dsp</td></tr>
<tr><td>書式</td><td>-dsp△DSP種別

DSP種別は以下のいずれか：
<table><tr><td>オプションで指定する種別</td><td>対象となるチップ</td></tr><tr><td>RX_DSP</td><td>RX</td></tr><tr><td>RL78_DSP</td><td>RL78(CCRL V1.01未満対象)</td></tr><tr><td>RL78_101_DSP</td><td>RL78(CCRL V1.01以降対象)</td></tr><tr><td>RL78_111_DSP</td><td>RL78(CCRL V1.11以降対象)</td></tr><tr><td>RL78_IAR_DSP</td><td>RL78(IARコンパイラ対象)</td></tr><tr><td>RL78_LLVM_DSP</td><td>RL78(LLVM for Renesas RL78対象)</td></tr><tr><td>RL78_GCC_DSP</td><td>RL78(GCC for Renesas RL78対象)</td></tr><tr><td>ARM_DSP</td><td>ARM</td></tr><tr><td>ARM_EABI5_DSP</td><td>ARM(EABI 準拠)</td></tr></table>
DSP種別の指定が無い場合、アセンブルエラーとなります。</td></tr>
<tr><td>備考</td><td>コード生成の対象となるDSPを指定します。
このオプションを指定しなかった場合、"RX_DSP" が指定されたものとみなします。
このオプションが複数回指定された場合は、最後に指定した内容が有効になります。

"ARM_EABI5_DSP" は、IAR/GNU ARM用ツールでリンク可能なオブジェクトファイルを生成する場合に指定します。
オブジェクトファイルをCCRL V1.01未満で利用する場合は、RL78_DSPを指定してください。
CCRL V1.01-V1.10で利用する場合は、RL78_101_DSPを指定してください。コードとデータは定数(CONSTF)として扱われます。
CCRL V1.11以降で利用する場合は、RL78_111_DSPを指定してください。コードとデータはRL78メモリ空間のDSP用コードとDSP用データ領域に配置するものとして扱われます。

RL78用IARツールで利用する場合は、RL78_IAR_DSPを指定してください。
LLVM for Renesas RL78で利用する場合は、RL78_LLVM_DSPを指定してください。
GCC for Renesas RL78で利用する場合は、RL78_GCC_DSPを指定してください。</td></tr>
<tr><td>例</td><td>dspasm -format OBJ -dsp RL78_DSP a.dsp</td></tr>
</table>

表 2.16 -core_versionコマンドラインオプション

<table>
<tr><td colspan="2">DSPコアバージョン指定： -core_version</td></tr>
<tr><td>書式</td><td>-core_version△バージョン

バージョンは以下のいずれか：
<table><tr><td>2</td><td>V2コアを対象としてアセンブルを行います。</td></tr><tr><td>3</td><td>V3コアを対象としてアセンブルを行います。</td></tr></table>
バージョンの指定が無い場合、アセンブルエラーとなります。</td></tr>
<tr><td>備考</td><td>コード生成の対象となるDSPコアバージョンを指定します。
このオプションを指定しなかった場合、"3"が指定されたものとみなします。
このオプションが複数回指定された場合は、最後に指定した内容が有効になります。</td></tr>
<tr><td>例</td><td>dspasm -format OBJ -dsp RL78_DSP -core_version 2 a.dsp</td></tr>
</table>

表 2.17 -Eコマンドラインオプション

プリプロセッサ処理結果ファイル出力： -E	
書式	-E
備考	入力ファイルにプリプロセスを行い、その結果をファイルに出力します。 出力するファイル名は、入力ファイルの拡張子を ".i" に変更したものです。 このオプションを指定した場合、DSPASMはプリプロセッサ処理だけを実行します。 構造化記述処理、ならびにアセンブラの実行は行いません。 このオプションと-listを併用した場合、リストファイルは出力されません。

表 2.18 -cpuLittleEndian, -cpuBigEndianコマンドラインオプション

CPUのエンディアン指定	
書式	-cpuLittleEndian -cpuBigEndian
備考	-cpuLittleEndianを指定した場合、CPUのエンディアンをリトルエンディアンに設定します。 -cpuBigEndianを指定した場合、またはどちらのオプションも指定しなかった場合、CPUのエンディアンをビッグエンディアンに設定します。 -cpuLittleEndianと-cpuBigEndianを併用した場合、後に指定したオプションが有効になります。 CPUのエンディアンをビッグエンディアンに設定する場合、プログラムのデータ値を4バイト単位で反転する必要があるため、-littleEndianDataも合わせて指定してください。

表 2.19 -littleEndianDataコマンドラインオプション

生成データ値リトルエンディアン指定： -littleEndianData	
書式	-littleEndianData
備考	アセンブル結果として出力するデータ値をリトルエンディアン形式で生成します。 このオプションを指定しなかった場合、データ値はビッグエンディアン形式で生成します。

表 2.20 -code_section_startコマンドラインオプション

コードセクション割り付け開始アドレス指定： -code_section_start	
書式	-code_section_start△割り付け開始アドレス 割り付け開始アドレスが無い場合、アセンブルエラーとなります。
備考	コードセクションの割り付けアドレスを10進数または16進数で指定します。 16進数表記については 表 4.6 16進数表記パターン をご参照ください。 指定できる割り付け開始アドレスの範囲は以下の通りです。範囲外の値を指定すると、アセンブルエラーとなります。 -core_version 2 指定あり：0 ～ FFFh -core_version 3 指定あり、または -core_version指定なし：0 ～ 3FFFh このオプションを指定しなかった場合、コードセクションは0番地から割り付けられます。
例	dspasm -format OBJ -code_section_start 500h a.dsp

表 2.21 -data_section_startコマンドラインオプション

<table>
<tr><td colspan="2">データセクション割り付け開始アドレス指定： -data_section_start</td></tr>
<tr><td>書式</td><td>-data_section_start△割り付け開始アドレス

割り付け開始アドレスが無い場合、アセンブルエラーとなります。</td></tr>
<tr><td>備考</td><td>データセクションの割り付けアドレスを10進数または16進数で指定します。
16進数表記については 表 4.6 16進数表記パターン をご参照ください。

指定できる割り付け開始アドレスの範囲は以下の通りです。範囲外の値を指定すると、アセンブルエラーとなります。
<table>
<tr><td>-core_version 2 指定あり</td><td>0 ～ FFF</td></tr>
<tr><td>-core_version 3 指定あり、または
-core_version指定なし</td><td>0 ～ 1FFFh</td></tr>
</table>

このオプションを指定しなかった場合、データセクションは0番地から割り付けられます。</td></tr>
<tr><td>例</td><td>dspasm -format OBJ -data_section_start 300h a.dsp</td></tr>
</table>

表 2.22 -no_debug_infoコマンドラインオプション

<table>
<tr><td colspan="2">デバッグ情報出力無効化指定： -no_debug_info</td></tr>
<tr><td>書式</td><td>-no_debug_info</td></tr>
<tr><td>備考</td><td>オブジェクトファイル内にデバッグ情報を出力しません。

このオプションを指定しなかった場合はオブジェクトファイル内にデバッグ情報を出力します。
ただし、 -dsp ARM_DSP オプションが指定されている場合は、 -no_debug_info オプションの指定の有無にかかわらず、デバッグ情報は出力されません。</td></tr>
</table>

表 2.23 -labelコマンドラインオプション

<table>
<tr><td colspan="2">ラベルのシンボル属性指定： -label</td></tr>
<tr><td>書式</td><td>-label△属性</td></tr>
<tr><td>備考</td><td>.public指定のないシンボルの属性を指定します。

属性は以下のいずれか：
<table>
<tr><td>オプションで指定する属性</td><td>シンボル設定</td></tr>
<tr><td>GLOBAL</td><td>グローバルシンボル(外部から参照可)</td></tr>
<tr><td>LOCAL</td><td>ローカルシンボル(外部から参照不可)</td></tr>
</table>

属性の指定が無い場合、アセンブルエラーとなります。
このオプションを指定しなかった場合、ローカルシンボルで出力します。
このオプションが複数回指定された場合は、最後に指定した内容が有効になります。</td></tr>
</table>

表 2.24 - macro_identifyコマンドラインオプション

<table>
<tr><td colspan="2">テキストマクロの識別方法指定： -macro_identify</td></tr>
<tr><td>書式</td><td>-macro_identify△識別方法</td></tr>
<tr><td>備考</td><td>テキストマクロの置換の際の、マクロの識別方法を指定します。

識別方法は以下のいずれか：
<table>
<tr><td>オプションで指定する識別方法</td><td>識別方法</td></tr>
<tr><td>FORWARD</td><td>'#' からマクロ名文字列をマクロ定義で置き換えます。置き換え対象となる識別子が、他の識別子の一部に含まれている場合でも置き換えを行います。</td></tr>
<tr><td>EXACT</td><td>'#' から始まるトークンがマクロ名文字列と一致する時のみマクロ定義で置き換えます。その他の動作はFORWARDと同じです。</td></tr>
</table>
このオプションを指定しなかった場合、'#' からマクロ名文字列をマクロ定義で置き換えます。置き換え対象となる識別子が、他の識別子の一部に含まれている場合でも置き換えを行います（FORWARDの動作）。
このオプションが複数回指定された場合は、最後に指定した内容が有効になります。</td></tr>
</table>

表 2.25 - dwarf_specコマンドラインオプション

<table>
<tr><td colspan="2">オブジェクトファイルのDWARF情報の出力仕様の指定： -dwarf_spec</td></tr>
<tr><td>書式</td><td>-dwarf_spec△出力設定</td></tr>
<tr><td>備考</td><td>オブジェクトファイルに出力するDWARF情報の仕様を指定します。

出力設定は以下のいずれか：
<table>
<tr><td>オプションで指定する出力設定</td><td>内容</td></tr>
<tr><td>INITIAL</td><td>DSPASMのDWARF仕様で出力します</td></tr>
<tr><td>GENERIC</td><td>GDBのDWARF仕様に準じた出力を行います</td></tr>
<tr><td>RENESAS</td><td>CCRL/CCRXのDWARF仕様に準じた出力を行います</td></tr>
</table>
このオプションを指定しなかった場合、INITIALの設定で出力します。

このオプションが複数回指定された場合、または -debug_areanges-no-paddingオプションと同時に指定した場合は、後に指定したオプションの内容が優先されます。</td></tr>
</table>

表 2.26 - code_execinstrコマンドラインオプション

コードセクションへのSHF_EXECINSTRフラグ設定有効化： -code_execinstr	
書式	-code_execinstr
備考	オブジェクトファイルのコードセクションに、SHF_EXECINSTRフラグを設定します。 このオプションを指定せず、かつ、-dsp オプションで ARM_DSP/ARM_EABI5_DSP/RL78_IAR_DSP/RL78_101_DSP/RL78_111_DSP が指定されている場合は、SHF_EXECINSTRフラグを設定しません。 このオプションを指定せず、かつ、-dsp オプションで ARM_DSP/ARM_EABI5_DSP/RL78_IAR_DSP/RL78_101_DSP/RL78_111_DSP 以外が指定されている場合は、SHF_EXECINSTRフラグを設定します。

表 2.27 -code_label_typeコマンドラインオプション

コードセクションのラベルに付与する属性を指定： -code_label_type	
書式	-code_label_type△ラベル属性
備考	コードセクションのラベルに付与する属性を指定します。 指定できる属性は以下の通りです。 オプションで指定する属性 / 内容 NOTYPE / STT_NOTYPE属性を付与します。 FUNC / STT_FUNC属性を付与します。 属性の指定が無い場合、アセンブルエラーとなります。 このオプションを指定せず、かつ、-dsp オプションで RL78_IAR_DSP を指定している場合は、コードセクションのラベルにSTT_FUNC属性を付与します。 このオプションを指定せず、かつ、-dsp オプションで RL78_IAR_DSP を指定していない場合は、コードセクションのラベルにSTT_NOTYPE属性を付与します。 このオプションが複数回指定された場合は、最後に指定した内容が有効になります。

表 2.28 -data_label_typeコマンドラインオプション

データセクションのラベルに付与する属性を指定： -data_label_type	
書式	-data_label_type△ラベル属性
備考	データセクションのラベルに付与する属性を指定します。 指定できる属性は以下の通りです。 オプションで指定する属性 / 内容 NOTYPE / STT_NOTYPE属性を付与します。 OBJECT / STT_OBJECT属性を付与します。 属性の指定が無い場合、アセンブルエラーとなります。 このオプションを指定せず、かつ、-dsp オプションで RL78_IAR_DSP を指定している場合は、データセクションのラベルにSTT_OBJECT属性を付与します。 このオプションを指定せず、かつ、-dsp オプションで RL78_IAR_DSP を指定していない場合は、データセクションのラベルにSTT_NOTYPE属性を付与します。 このオプションが複数回指定された場合は、最後に指定した内容が有効になります。

3.プリプロセス処理の概要

DSPASMでは、入力となるアセンブリ言語ファイルに以下のプリプロセス処理を行います。

- ●マクロ置き換え(ファイルに記述された文字列の置き換え)
- ●条件付き取り込み(アセンブル対象とするコードブロックの選択)
- ●ファイル取り込み(外部ファイルの取り込み)

この章では、それぞれのプリプロセス処理の概要を記述します。
プリプロセス処理の詳細につきましては、6「プリプロセス処理の詳細」をご覧ください。

3.1.プリプロセス処理の対象となる識別子の判別

DSPASMでは、アセンブリ言語ファイルに記述された内容がプリプロセス命令であるか、あるいはマクロ置き換えの対象となる文字列かを判断するため、ある特定の文字（テキストマクロ文字）を使用します。

テキストマクロ文字は、"-text_macro" コマンドラインオプションで指定することが可能であり、デフォルトでは '#' が用いられます。使用可能な文字については、表 2.11 -text_macro コマンドラインオプション をご参照ください。

この章ではテキストマクロ文字を'{TC}' という記述で表します。

3.2.マクロ置き換え

DSPASMでは、アセンブリ言語ファイルに記述された特定の文字列を、別の文字列に置き換えることができます。この置き換えのことをマクロ置き換えと呼びます。

マクロ置き換えは{TC}で始まる文字列が対象となり、どのような置き換えを行うかは"{TC} define"プリプロセス命令、もしくは "-define" コマンドラインオプションで指定します。

表 3.1 defineプリプロセス命令

マクロ置き換えプリプロセス命令：{TC}define
書式: 1){TC}define△識別子△置換要素並び 2){TC}define△識別子([識別子並び]) △置換要素並び 3){TC}define△識別子(...) △置換要素並び 4){TC}define△識別子(識別子並び , ...) △置換要素並び いずれも行頭の一文字は{TC} から始まります。
サンプルコード(このサンプルでは {TC} として '#' を使用します)。

```
; Sample code
;
#define MAGIC_NO     #H'123
MOV #MAGIC_NO, DP0                    ; "#MAGIC_NO" を "H'123" に置き換えます。

#define CLEAR_A0()   A0=0             ; 関数形式のマクロ置き換え。
#CLEAR_A0()                           ; "A0=0" に置き換えられます。

#define SET_A0(VAL)   A0 = #VAL   ; 引数付きの関数形式のマクロ置き換え。
#SET_A0(1)                            ; "A0 = 1" に置き換えられます。
```

```
#define SET_REG(…)   MOV   #__VA_ARGS__      ; 可変個引数を用いた関数形式のマクロ置き換え。
                                             ; 引数の内容をプリデファインドマクロ __VA_ARGS__に反映します。
#SET_REG(#H'123, DP0)                        ; "MOV H'123, DP0" に置き換えられます。
                                             ; 複数個の引数を指定することができます。

#define SET_REG2(R, …)   MOV   #__VA_ARGS__, #R   ; 可変個引数を用いた関数形式のマクロ置き換え。
                                                  ; 第一引数をRに反映し、残りの引数の内容を
                                                  ; プリデファインドマクロ__VA_ARGS__ に反映します。
#SET_REG2(DP0, #H'ABC)                            ; "MOV H'ABC, DP0" に置き換えられます。
```

備考:

a)マクロ置き換え対象となるシンボルは大文字/小文字の区別を行いません。

b)可変個引数で指定した引数は、プリデファインドマクロ "__VA_ARGS__" に展開されます。

c)マクロ置き換えは、置き換え対象となる識別子が、他の識別子の一部に含まれている場合でも置き換えを行います。トークン単位で置き換えを行う場合はコマンドラインオプション "-macro_identify EXACT" を指定してください。
また、識別子の置き換えは、先に定義されたものから優先して置き換えを行います。

```
#define   VALUE    8
MOV #H'7#VALUE9, DP0              ; "H'789" に置き換えられます

#define  AA    #H'1
#define  AAA    #H'9
MOV #AA23, DP0                    ; "#H'123" に置き換えられます。
```

d)マクロ置き換えは、同一行で複数回の置き換えを行います(置き換えをネストすることができます)。

```
#define   VAL_TYPE    #ZERO
#define  ZERO   H'00000000
A0 = #VAL_TYPE                    ; A0 = H'00000000" に置き換えられます
```

e)同一行で複数回の置き換えを行う場合の置換結果は、テキストマクロを定義した順序に依存します。

```
#define  ZERO   H'00000000
#define   VAL_TYPE    #ZERO
A0 = #VAL_TYPE                    ; A0 = #ZERO に置き換えられます
```

f)マクロ置き換えは、置き換え対象となる識別子が DSPASM の予約語であったとしても置き換えを行います。

g)マクロ置き換えを再定義することはできません。再定義を許可する場合は、コマンドラインオプション "-allow_text_macro_redefine" を指定してください。また、"-allow_text_macro_redefine" を指定した場合は、コマンドラインオプション "-define" で指定したマクロ置き換え定義を、ソースコード中の {TC}define 命令で上書きすることができます

h)マクロ置き換えプリプロセス命令を複数行にわたって記述することはできません。

```
#define   MANY_ARGS(arg1, arg2, arg3,¥    ; 複数行に渡る定義はアセンブルエラーとします。
arg4, arg5, arg6)   VAL
```

文字列長、識別子数の最大値:

文字列長、識別子数の最大値については、10.1「プリプロセス処理の翻訳限界」をご参照ください。

エラーとなる記述:

✧マクロ置き換えプリプロセス命令では、置換要素並びを省略することができます。
✧可変引数でない関数マクロ定義において、置き換え対象となる記述とマクロ定義の引数の個数は一致していなければなりません。

```
#define   CLEAR_A0()    A0 = 0
#CLEAR_A0(1)                ; 引数の数が一致していないためアセンブルエラーとなります。

#define   SET_A0(VAL)    A0 = #VAL
#SET_A0()                   ; 引数の数が一致していないためアセンブルエラーとなります。
```

✧関数形式のマクロ定義で使用した識別子が、"()" を伴わない形式で記述された場合はアセンブルエラーとなります。

```
#define   CLEAR_A0()   A0 = 0
#CLEAR_A0                    ;"()"がないためアセンブルエラーとなります。
```

✧関数形式のマクロ定義の引数名に使用できるのは英文字、数字、'_'(アンダーバー) のみです。

```
#define   CLEAR_A0(ARG+1)   A0 = #ARG+1     ; 引数名に記号があるためアセンブルエラーとなります。
#define   CLEAR2_A0(ARG@1)   A0 = #ARG@1   ; 引数名に記号があるためアセンブルエラーとなります。
```

3.3.条件付き取り込み

DSPASMでは、特定の条件に応じて、アセンブリ言語ファイルに記述されたソースコードの一部分をアセンブル対象にするか否かを切り分けることができます。この機能のことを条件付き取り込みと呼びます。

条件付き取り込みでは、以下の二種類の条件を判定します。
1)特定のシンボルが定義されているか？
2)定数式が成立するか？

シンボル定義の有無を判断する場合は、"{TC}ifdef", "{TC}ifndef" プリプロセス命令を使用します。また、定数式の結果を判断する場合は、"{TC}if" プリプロセス命令を使用します。

表 3.2 ifプリプロセス命令

定数式の結果による条件付き取り込みプリプロセス命令：{TC}if
書式: 1){TC}if△定数式 *取り込み対象となる次ソースコード記述ブロック* {TC}elif△定数式 *取り込み対象となる次ソースコード記述ブロック* {TC}else *取り込み対象となる次ソースコード記述ブロック* {TC}endif いずれも行頭の一文字は{TC} から始まります。 {TC}elif, および {TC}else は省略することができます。 また、ひとつの{TC}if ～ {TC}endif の間に、複数の{TC}elif を記述することができます。
サンプルコード(このサンプルでは {TC} として '#' を使用します)。
; Sample code ; SECTION CODE

```
#define NUM_UNITS     2
#if       #NUM_UNITS >= 2
    ;
    ; NUM_UNITS が 2 以上だった場合の処理
    ;
#elif   #NUM_UNITS == 1
    ;
    ; NUM_UNITS が 1だった場合の処理
    ;
#else
    ;
    ; NUM_UNITS が 2以上でも、1でもなかった場合の処理
    ;
#endif
```

備考:

a)定数式で使用可能な演算子につきましては、6「プリプロセス処理の詳細」をご覧ください。
b)定数式を省略することはできません。
c)定数式では代入演算子、単項加算演算子、および単項減算演算子は利用できません。
d)定数式は "()" によって演算の優先順位を変更することができます。
e)定数式では 10 進数表記、および 16 進数表記をサポートします。

定数式の長さ、ネスト回数の最大値:

定数式の長さ、ネスト回数の最大値については、10.1「プリプロセス処理の翻訳限界」をご参照ください。

エラーとなる記述:

✧{TC}if に対応する{TC}endif は、同一のファイルに記述しなければなりません。
✧定数式において、二項演算子の右辺や左辺が記述されていない、もしくは括弧の対応が取れていないなど、定数式として不正な形式であった場合はアセンブルエラーとなります。
✧定数式を処理する過程で、0 による除算、もしくは剰余算が発生した場合はアセンブルエラーとなります。

表 3.3 ifdef/ifndefプリプロセス命令

シンボル定義の有無による条件付き取り込みプリプロセス命令：{TC}ifdef / {TC}ifndef

書式:

1){TC}ifdef△識別子
　　取り込み対象となる次ソースコード記述ブロック
　　{TC}else
　　取り込み対象となる次ソースコード記述ブロック
　　{TC}endif

2){TC}ifndef△識別子
　　取り込み対象となる次ソースコード記述ブロック
　　{TC}else
　　取り込み対象となる次ソースコード記述ブロック
　　{TC}endif

いずれも行頭の一文字は{TC} から始まります。
{TC}else は省略することができます。
{TC}if～{TC}endif の間に、{TC}elif を記述することができます。

サンプルコード(このサンプルでは {TC} として '#' を使用します)。

```
; Sample code
;
SECTION CODE

#define TEST_VERSION
#ifdef    TEST_VERSION
    ;
    ; TEST_VERSIONが定義されている場合の処理
    ;
#else
    ;
    ; TEST_VERSIONが定義されていない場合の処理
    ;
#endif

#define    CUSTOM_ONLY
#ifndef    CUSTOM_ONLY
    ;
    ; CUSTOM_ONLYが定義されていない場合の処理
    ;
#else
    ;
    ; CUSTOM_ONLYが定義されている場合の処理
    ;
#endif

#define VERSION_NUM     10
#ifdef    VERSION_NUM
    ;
    ; VERSION_NUMがどのように置換されるかに関わらず、
    ;  シンボルが定義されていれば、このブロックはアセンブル対象となります。
    ;
#endif
```

備考:

a)プリプロセス命令：{TC}ifdef / {TC}ifndef は、シンボルがどのような値に置き換えられるかは問わず、シンボル定義の有無だけを判別します。シンボルの値によって条件判定を行う場合は、プリプロセス命令：{TC}if を使用してください。
b)識別子は大文字/小文字の区別を行いません。
c)識別子を省略することはできません。

ネスト回数の最大値:

ネスト回数の最大値については、10.1「プリプロセス処理の翻訳限界」をご参照ください。

エラーとなる記述:

✧{TC}ifdef / {TC}ifndef に対応する{TC}endif は、同一のファイルに記述しなければなりません。

3.4.ファイル取り込み

DSPASMでは、アセンブル対象となるアセンブリ言語ファイルに、他のファイルの内容を取り込むことができます。この機能のことをファイル取り込みと呼びます。
ファイル取り込みを行うには "{TC}include" プリプロセス命令を使用します。

表 3.4 includeプリプロセス命令

ファイル取り込みプリプロセス命令：{TC}include
書式: 1){TC}include△<ファイル名> いずれも行頭の一文字は{TC} から始まります。
サンプルコード(このサンプルでは {TC} として '#' を使用します)。 ; Sample code ; SECTION_CODE #include <value_no.def> ; value_no.def ファイルを取り込みます。 #include <..¥def¥const.def> ； 相対パスで指定したファイルを取り込みます。 #define STARTUP_FILE startup.asm; ファイル名をシンボル定義。 #include <#STARTUP_FILE> ; "startup.asm" ファイルを取り込みます。
備考: a)ファイル名にはパス名を付加することができます。パス名の形式は絶対パス、相対パスの両方をサポートします。 b)パス名の区切り記号は、"¥"、"/" の両方をサポートします。 c)相対パスの起点となるフォルダは、コマンドラインオプション "-inc_dir" で指定することができます。"-inc_dir" オプションを指定しなかった場合、アセンブリ言語ファイルが格納されているフォルダを相対パスの起点とみなします。 d)ファイル名を囲む記号は '<'、'>' のみをサポートしています。 e){TC}include で取り込んだファイルに、マクロ置き換えプリプロセス命令：{TC}define を記述した場合、{TC}define が記述された行からシンボルの定義が有効になります。
ファイル取り込みの最大ネスト回数: ファイル取り込みの最大ネスト回数については、10.1「プリプロセス処理の翻訳限界」をご参照ください。
エラーとなる記述: ✧ファイルを読み込むことができない場合はアセンブルエラーとなります。

3.5.プリデファインドマクロ(定義済みマクロ)

DSPASMでは、以下のシンボルをプリデファインドマクロとして使用します。
これらのシンボルは予約語であり、ユーザが定義内容を変更することはできません。

- __VA_ARGS__
- __RENESAS__
- __RENESAS_VERSION__
- __DSPASM__

4.構造化記述処理の概要

DSPASMでは、ソースコードを記述する際に通常のアセンブラ表記に加えて、DSPASM独自の構造化記述を用いることができます。

DSPASMの構造化記述はユーザプログラムの作成を支援するためのものであり、以下の特徴を備えています。

- ●分岐、多分岐、繰返しなどのプログラム構造を C 言語に近い形式で記述することができます。
- ●メモリやレジスタに対する数値演算、代入、ビット演算、比較操作を C 言語に近い形式で記述することができます。
- ●通常のアセンブラ記述と比べて、レジスタのビット操作、ビット比較操作を容易に記述することができます。

この章では、構造化記述処理の概要を記述します。

構造化記述処理の詳細につきましては、7「構造化記述処理の詳細」をご覧ください。

4.1.構造化記述で使用できる変数名

構造化記述では、以下の変数名を使用することができます。

- ●レジスタ変数
- ●フラグ変数
- ●A0 レジスタビット変数
- ●ポインタ変数

4.1.1.レジスタ変数

構造化記述では以下のレジスタ変数を使用することができます。

表 4.1 レジスタ変数

変数名	レジスタ種別
A0	アキュームレータレジスタ
M0	乗数レジスタ
M1	シフト数レジスタ
L0	リミット上限値レジスタ
L1	リミット下限値レジスタ
R0	加算値レジスタ
R1	加算値レジスタ1
DP0	アキュームレータ用アドレスポインタ
DP1	演算パラメータ用アドレスポインタ
RP0	演算結果格納用アドレスポインタ
F0	フラグレジスタ
PS0	プログラムセグメントレジスタ
DS0	データセグメントレジスタ
SS0	スタックセグメントレジスタ

※R1、F0、PS0、DS0、SS0 レジスタは、V3 コア以降でのみサポートされます。

また、BR0レジスタ、およびPG0につきましては、レジスタ操作を行うアセンブラ命令が存在しないため、構造化記述ではサポートしておりません。SP0レジスタにつきましては、構造化記述で生成されるアセンブラコードでスタックを利用しており、その結果SP0が書き換わるため、構造化記述ではサポートしておりません。これらのレジスタを操作する場合は、ユーザが通常のアセンブラコードを記述する必要があります。

4.1.2.フラグ変数

構造化記述では以下のフラグ変数を使用することができます。

表 4.2 フラグ変数

変数名	レジスタ種別
I	割り込みフラグ
Z	ZEROフラグ
U	UNDERフラグ
O	OVERフラグ

4.1.3.A0 レジスタビット変数

構造化記述では、レジスタビット変数を用いることで、A0レジスタのそれぞれのビットを直接操作することができます。

指定するビットは0～31の範囲に納まる必要があります。範囲外のビットを指定した場合はアセンブルエラーとなります。

表 4.3 A0レジスタビット変数

変数名	レジスタ種別
A0_0	A0レジスタの最下位ビット
A0_1	A0レジスタの下位側から数えて2ビット目のビット
A0_2	A0レジスタの下位側から数えて3ビット目のビット
省略	
A0_29	A0レジスタの上位側から数えて3ビット目のビット
A0_30	A0レジスタの上位側から数えて2ビット目のビット
A0_31	A0レジスタの最上位ビット

A0レジスタビット変数のイメージ

表 4.4 A0レジスタビットイメージ

上位側← A0 レジスタ → 下位側						
31	30	29	省略	2	1	0

4.1.4.ポインタ変数

表 4.5 ポインタ変数

変数名	レジスタ種別

(DP0)	アキュームレータ用アドレスポインタ
(DP1)	演算パラメータ用アドレスポインタ
(*XX*)	データ用メモリ
(*NN*, DP1)	演算パラメータ用アドレスポインタ
(*NN*, DP0)	アキュームレータ用アドレスポインタ
(*NN*, RP0)	演算結果格納用アドレスポインタ
(DP0+R0)	アキュームレータ用アドレスポインタ + 加算値レジスタ

※表の '*XX*' には即値、もしくはラベル名を記述することができます。
※表の '*NN*' には -128 ～ 127 の範囲の即値を記述することができます。
※制御文(詳細は 4.4「構造化記述で使用できる制御文」をご参照ください)中の式では、"()"は
ポインタ変数ではなく、演算の優先順位を変えるという意味になるため、ご注意ください。(詳細は 7.6「"( )" の使用箇所による意味の違い」をご参照ください)
※ポインタ変数(DP0+R0)は、V3 コア以降でのみサポートされます。

4.2.構造化記述で使用できる定数

構造化記述では、定数表記として10進数表記と16進数表記をサポートします。
10進数表記、16進数表記共に記述できる桁数に制限はありませんが、値が4バイトを超える場合は
アセンブルエラーとなります。
また、16進定数の表記には、以下のいずれかの書き方を用いることができます。

表 4.6 16進数表記パターン

表記パターン	備考
H'xxxxxxxx	16進定数の先頭に "H'"プレフィックスを付加したパターンです。
h'xxxxxxxx	プレフィックスの 'H' は大文字小文字のどちらでも構いません。
xxxxxxxxh	16進定数の末尾に "H'"サフィックスを付加したパターンです。
xxxxxxxxH	サフィックスの 'H' は大文字小文字のどちらでも構いません。

※表の '*x*'は 16 進数の一桁を表します(0-9 に加えて、A-F/a-f の文字を使用することができます)。

4.3.構造化記述で使用できる演算子

表 4.7 演算子

種別	演算子	内容
単項演算子	+	正数を示します
	-	負数を示します
	~	ビット反転演算を行います (NOT)
	++	加算を行います (符号なし)
	++.s	加算を行います (符号あり、飽和演算)
	--	減算を行います (符号なし)
	--.s	減算を行います (符号あり、飽和演算)
乗除演算子	*	乗算を行います (符号なし)
	*.s	乗算を行います (符号あり、飽和演算)
	/	除算を行います (符号なし) この命令は、V2 コアでは使用できません。

		使用した場合はアセンブルエラーとなります。
	%	剰余算を行います (符号なし) この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
加減演算子	+	加算を行います (符号なし)
	+.s	加算を行います (符号あり、飽和演算)
	-	減算を行います (符号なし)
	-.s	減算を行います (符号あり、飽和演算)
ビット単位のシフト演算子	<<	指定したビット分を論理左シフトします
	<<.s	指定したビット分を算術左シフトします
	>>	指定したビット分を論理右シフトします
	>>.s	指定したビット分を算術右シフトします
関係演算子	<	左辺が右辺より小さい場合に演算結果が真 (符号なし) この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	<.s	左辺が右辺より小さい場合に演算結果が真 (符号あり)
	>	左辺が右辺より大きい場合に演算結果が真 (符号なし) この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	>.s	左辺が右辺より大きい場合に演算結果が真 (符号あり)
	<=	左辺が右辺より小さいか等しい場合に演算結果が真 (符号なし) この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	<=.s	左辺が右辺より小さいか等しい場合に演算結果が真 (符号あり)
	>=	左辺が右辺より大きいか等しい場合に演算結果が真 (符号なし) この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	>=.s	左辺が右辺より大きいか等しい場合に演算結果が真 (符号あり)
等価演算子	==	左辺と右辺が等しい場合に演算結果が真
	!=	左辺と右辺が等しくない場合に演算結果が真
ビット単位のAND/OR/排他OR演算子	&	ビット毎の論理積演算を行います (AND)
	|	ビット毎の論理和演算を行います (OR)
	^	ビット毎の排他的論理和演算を行います (XOR)
論理AND/OR演算子	&&	左辺と右辺の値を0 と比較してともに等しくない場合に真 短絡評価する(左辺が偽なら右辺を評価しません)
	||	左辺と右辺の値を0 と比較していずれか一方でも等しくない場合に真 短絡評価する(左辺が真なら右辺を評価しません)
代入演算子	=	右辺を左辺に代入 var ← exp
	*=	左辺と右辺の乗算結果を左辺に代入 (符号なし) var ← var * exp
	*=.s	左辺と右辺の乗算結果を左辺に代入 (符号あり、飽和演算)

		var ← var * exp
	/=	左辺と右辺の除算結果を左辺に代入 (符号なし) var ← var / exp この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	%=	左辺と右辺の剰余算結果を左辺に代入 (符号なし) var ← var % exp この命令は、V2 コアでは使用できません。 使用した場合はアセンブルエラーとなります。
	+=	左辺と右辺の加算結果を左辺に代入 (符号なし) var ← var + exp
	+=.s	左辺と右辺の加算結果を左辺に代入 (符号あり、飽和演算) var ← var + exp
	-=	左辺と右辺の減算結果を左辺に代入 (符号なし) var ← var - exp
	-=.s	左辺と右辺の減算結果を左辺に代入 (符号あり、飽和演算) var ← var - exp
	<<=	左辺を右辺のビット分論理左シフトした結果を左辺に代入 var ← var << exp
	<<=.s	左辺を右辺のビット分算術左シフトした結果を左辺に代入 var ← var << exp
	>>=	左辺を右辺のビット分論理右シフトした結果を左辺に代入 var ← var >> exp
	>>=.s	左辺を右辺のビット分算術右シフトした結果を左辺に代入 var ← var >> exp
	&=	左辺と右辺の論理積演算結果を左辺に代入 var ← var and exp
	|=	左辺と右辺の論理和演算結果を左辺に代入 var ← var or exp
	^=	左辺と右辺の排他的論理和演算結果を左辺に代入 var ← var xor exp
	複合代入(=..=)	最右辺を左辺に代入 var ← var2 ← … ← exp

※表の *'var', 'var2'* は変数を意味し、*'exp'* は任意の式を意味します(式の構成要素は変数、定数、演算子を組み合わせたものです)。

※単項演算子 '++', '—' は前置のみをサポートします。'A0—', 'A0++' のように後置形式で記述した場合はアセンブルエラーとなります。

※演算子が扱えない変数を使用した場合はアセンブルエラーとなります。(詳細は 7.2.4「演算子が扱えない変数」をご参照ください)

4.3.1.演算子の優先順位

構造化記述で使用する演算子の優先順位は以下のとおりです。

表 4.8 演算子の優先順位

高 ↑ ↓ 低	(1)	+, - , ~, ++, ++.s, --, --.s (いずれも単項演算子)
	(2)	*, *.s, /, %, <<, <<.s, >>, >>.s
	(3)	+. +.s, -, -.s
	(4)	&, | , ^
	(5)	==, !=, <, <.s, >, >.s, <=, <=.s, >=, >=.s
	(6)	&&, ||
	(7)	=, *=, *=.s, /=, %=, +=, +=.s, -=, -=.s, <<=, <<=.s, >>=, >>=.s, &=, |=, ^=

制御文(詳細は4.4「構造化記述で使用できる制御文」をご参照ください)中の式では、式を"( )"で囲むことで、演算の実行順序を変更することができます。"( )" で囲まれた式は、他の式よりも優先して演算されます。
なお、制御文中の式以外では、"( )"はポインタ変数を意味するため、演算の実行順位を変更するために利用できません。

4.4.構造化記述で使用できる制御文

構造化記述では、以下の制御文を使用することができます。

- ●if … elif … else … endif
- ●switch … case … default … endsw
- ●while … endwh
- ●do … during
- ●for … to … step … endfor
- ●goto
- ●continue
- ●break

表 4.9 if 制御文

条件分岐：if … elif … else … endif

書式:
if△[式1]
　　文1
elif△[式2]
　　文2
else
　　文3
endif
 	
※一つの if 文の中に複数の'elif'を記述することができます。

機能： if文とは式を評価した結果の真偽値によって、プログラムの制御を分岐させるものです。 1) 'if', 'elif' で記述した式の評価結果が真であった場合、対応する文に制御が移ります。 2) 'if', 'elif' で記述した式の評価結果が全て偽であった場合、'else'が記述された文に制御が移ります。
仕様: a)式には変数、定数、演算子を記述することができます。 b)式で使用できる演算子は以下のとおりです、

●関係演算子(<, >, <=, >=, <.s, >.s, <=.s, >=.s)
●等価演算子(==, !=)
●論理 AND/OR 演算子(&&, ||)
●ビット単位の AND/OR/排他 OR 演算子(&, |, ^)
c)'elif', 'else' の記述は省略することができます。
d)一つの if 文の中に複数の'elif'を記述することができます。
e)'endif'を省略することはできません。

構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)

if [A0 < R0]
R0 = A0
else
A0 = R0
endif
	;--- レジスタ退避 ---
	PUSH A0
	PUSH RP0
	;--- オペランド2取得 ---
	PUSH RP0
	PUSH A0
	PUSH R0
	POP A0
	MOV.L #__V_00000002,RP0
	MOV A0,(RP0+)
	POP A0
	POP RP0
	;--- オペランド1取得 ---
	PUSH A0
	POP A0
	;--- 比較実行 ---
	PUSH A0
	PUSH L0
	PUSH L1
	PUSH DP1
	MOV A0,L0
	MOV.L #__V_00000002,DP1
	MOV (DP1+),L1
	CLAMP UNSIGNED
	POP DP1
	POP L1
	POP L0
	POP A0
	;--- 比較結果保存 ---
	MOV.L (#__C_00000001),A0
	MOV.L #__V_00000001,RP0
	MOV A0,(RP0+)
	; --- レジスタ復帰 ---

```
        POP RP0
        POP A0
        ; --- 条件分岐実施 ---
        JMP UNDER, #__L_00000004
        ;--- 比較結果保存 ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; --- 条件分岐実施
        JMP #__L_00000005
__L_00000004:

        PUSH  A0
        POP  R0
        PUSH  A0
        PUSH  RP0
        PUSH A0
        POP  A0
        MOV.L  #__V_00000001,RP0
        MOV  A0,(RP0+)
        POP  RP0
        POP  A0
        PUSH  A0
        PUSH  RP0
        MOV.L  (#__V_00000001),A0
        MOV.L#__V_00000002,RP0
        MOVA0,(RP0+)
        MOV.L#__V_00000003,RP0
        MOV  A0,(RP0+)
        POP  RP0
        POP  A0
        JMP #__L_00000003
__L_00000005:
        PUSH  R0
        POP  A0
        PUSH  A0
        PUSH  RP0
        PUSH  R0
        POP  A0
```

```
        MOV.L  #__V_00000001,RP0
        MOV  A0,(RP0+)
        POP  RP0
        POP  A0
        PUSH  A0
        PUSH  RP0
        MOV.L  (#__V_00000001),A0
        MOV.L  #__V_00000002,RP0
        MOV  A0,(RP0+)
        MOV.L  #__V_00000003,RP0
        MOV  A0,(RP0+)
        POP  RP0
        POP  A0
        JMP #__L_00000003
__L_00000003:

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
__V_00000003:       DATA H'00000000
```

表 4.10 switch … case 制御文

多分岐：switch … case … default … endsw

書式:

```
switch△[式]
case△値:
   文1
   break
case△値:
   文2
   break
default:
   文3
   break
endsw
```

機能：

switch文とは式の値によって、プログラムの制御を分岐させるものです。

1) 式の値とcase ラベルの記述が一致する文に制御が移ります。
2) 式の値とcase ラベルの記述が一致しなかった場合、default文に制御が移ります。
3) 式の値とcase ラベルの記述が一致せず、default節も記述されていなかった場合は、いずれの文も実行されません。

仕様:

a)式には変数、定数、および単項演算子のみを記述することができます。
b)case ラベルの値として記述できるのは定数のみです。
c)同じ値の case ラベルを複数記述することはできません。case ラベルの値が重複している場合、アセンブルエラーとなります。
d)'break' の記述は省略することができます。省略した場合、次の'case'ラベル、もしくは 'default'ラベルの直後に記述された処理を実行します。
e)'default' の記述は省略することができます。
f)'endsw'を省略することはできません。

構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
switch [A0] case 1: ... break case 2: ... break case 3: ... break default: ... break endsw	__L_00000001: PUSH A0 PUSH RP0 PUSH RP0 PUSH A0 PUSH A0 POP A0 MOV.L #__V_00000002,RP0 MOV A0,(RP0+) POP A0 POP RP0 MOV.L (#__C_00000000),A0 PUSH DP0 MOV.L #__V_00000002,DP0 CMP (DP0+) POP DP0 MOV.L (#__C_00000001),A0 MOV.L #__V_00000001,RP0 MOV A0,(RP0+) POP RP0 POP A0 JMP NOT_ZERO, #__L_00000006 PUSH A0 PUSH RP0 MOV.L (#__C_00000000),A0 MOV.L #__V_00000001,RP0 MOV A0,(RP0+) POP RP0 POP A0 JMP #__L_00000003 __L_00000006: __L_00000003:

```
        ; case 1:
        ; ---  レジスタ退避  ---
        PUSH A0
        PUSH RP0
        ; ---  オペランド2取得  ---
        PUSH RP0
        PUSH A0
        MOV.L #__V_00000004,RP0
        MOV.L (#__C_00000001),A0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ; ---  オペランド1取得  ---
        PUSH A0
        POP A0
        ; ---  比較実行  ---
        PUSH DP0
        MOV.L #__V_00000004,DP0
        CMP (DP0+)
        POP DP0
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施
        JMP ZERO, #__L_00000009
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_0000000a
__L_00000009:
        PUSH A0
        PUSH RP0
        MOV.L (#__V_00000001),A0
        MOV.L #__V_00000002,RP0
        MOV A0,(RP0+)
        MOV.L #__V_00000003,RP0
```

```
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_00000005
__L_0000000a:
        ; case 2:
        ; ---  レジスタ退避  ---
        PUSH A0
        PUSH RP0
        ; ---  オペランド2取得  ---
        PUSH RP0
        PUSH A0
        MOV.L #__V_00000004,RP0
        MOV.L (#__C_00000002),A0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ; ---  オペランド1取得  ---
        PUSH A0
        POP A0
        ; ---  比較実行  ---
        PUSH DP0
        MOV.L #__V_00000004,DP0
        CMP (DP0+)
        POP DP0
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施
        JMP ZERO, #__L_0000000d
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_0000000e
__L_0000000d:
        PUSH A0
```

```
        PUSH RP0
        MOV.L (#__V_00000001),A0
        MOV.L #__V_00000002,RP0
        MOV A0,(RP0+)
        MOV.L #__V_00000003,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_00000005
__L_0000000e:
        ; case 3:
        ; ---  レジスタ退避  ---
        PUSH A0
        PUSH RP0
        ; ---  オペランド2取得  ---
        PUSH RP0
        PUSH A0
        MOV.L #__V_00000004,RP0
        MOV.L (#__C_00000003),A0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ; ---  オペランド1取得  ---
        PUSH A0
        POP A0
        ; ---  比較実行  ---
        PUSH DP0
        MOV.L #__V_00000004,DP0
        CMP (DP0+)
        POP DP0
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施
        JMP ZERO, #__L_00000011
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
```

```
        POP RP0
        POP A0
        JMP #__L_00000012
__L_00000011:
        PUSH A0
        PUSH RP0
        MOV.L (#__V_00000001),A0
        MOV.L #__V_00000002,RP0
        MOV A0,(RP0+)
        MOV.L #__V_00000003,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_00000005
__L_00000012:
        JMP #__L_00000005
        JMP #__L_00000005
__L_00000014:
__L_00000005:

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__C_00000002:       DATA H'00000002
__C_00000003:       DATA H'00000003
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
__V_00000003:       DATA H'00000000
__V_00000004:       DATA H'00000000
```

表 4.11 while 制御文

条件付き繰返し：while … endwh
書式:
while△[式] 　文 endwh
機能：
while文とは式の結果が真である間、プログラムの制御を繰返し実行するためのものです。 1) 式の評価結果が真である間、文の記述内容を繰返し実行します。
仕様:
a)式には変数、定数、演算子を記述することができます。 b)式で使用できる演算子は以下のとおりです、

●関係演算子(<, >, <=, >=, <.s, >.s, <=.s, >=.s)
●等価演算子(==, !=)
●論理 AND/OR 演算子(&&, ||)
●ビット単位の AND/OR/排他 OR 演算子(&, |, ^)
c)式に'forever' と記述した場合、無限ループとなるコードを生成します。
d)'endwh'を省略することはできません。

構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)

while [A0 < 9]
++A0
endwh
	__L_00000001:
	; --- レジスタ退避 ---
	PUSH A0
	PUSH RP0
	; --- オペランド2取得 ---
	PUSH RP0
	PUSH A0
	MOV.L #__V_00000002,RP0
	MOV.L (#__C_00000009),A0
	MOV A0,(RP0+)
	POP A0
	POP RP0
	; --- オペランド1取得 ---
	PUSH A0
	POP A0
	; --- 比較実行 ---
	PUSH A0
	PUSH L0
	PUSH L1
	PUSH DP1
	MOV A0,L0
	MOV.L #__V_00000002,DP1
	MOV (DP1+),L1
	CLAMP UNSIGNED
	POP DP1
	POP L1
	POP L0
	POP A0
	; --- 比較結果保存 ---
	MOV.L (#__C_00000001),A0
	MOV.L #__V_00000001,RP0
	MOV A0,(RP0+)
	; --- レジスタ復帰 ---
	POP RP0

```
        POP A0
        ; --- 条件分岐実施 ---
        JMP UNDER, #__L_00000005
        ;--- 比較結果保存 ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; --- 条件分岐実施 ---
        JMP #__L_00000006
__L_00000005:
        ; --- レジスタ退避 ---
        PUSH R0
        PUSH RP0
        ; --- インクリメント ---
        MOV.L (#__C_00000001),R0
        ADD_R
        ; --- 結果保存 ---
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; --- 結果更新 ---
        PUSH DP0
        MOV.L #__V_00000001,DP0
        MOV (DP0+),A0
        PUSH A0
        POP A0
        POP DP0
        ; --- レジスタ復帰 ---
        POP RP0
        POP R0
        PUSH    A0
        PUSH    RP0
        MOV.L     (#__V_00000001),A0
        MOV.L    #__V_00000002,RP0
        MOV         A0,(RP0+)
        MOV.L    #__V_00000003,RP0
        MOV      A0,(RP0+)
        POP      RP0
        POP      A0
        JMP #__L_00000001
```

	__L_00000006: __L_00000004: SECTION DATA NAME SAREA_DSPDATA __C_00000000: DATA H'00000000 __C_00000001: DATA H'00000001 __C_00000009: DATA H'00000009 __V_00000001: DATA H'00000000 __V_00000002: DATA H'00000000 __V_00000003: DATA H'00000000

表 4.12 do 制御文

条件付き繰返し(後判定)：do … during	
書式: do 　文 during△[式]	
機能： do文とは式が真である間、プログラムの制御を繰返し実行するためのものです。 while文では、文を実行する前に繰返しを継続するか否かの判定を行いますが、do文では文を実行した後に繰返しを継続するか否かの判定を行います。 1) 記述内容を実行した後で式の評価を行い、その結果が真である間、文の記述内容を繰返し実行します。	
仕様: a)式には変数、定数、演算子を記述することができます。 b)式で使用できる演算子は以下のとおりです、 ●関係演算子(<, >, <=, >=, <.s, >.s, <=.s, >=.s) ●等価演算子(==, !=) ●論理 AND/OR 演算子(&&, ||) ●ビット単位の AND/OR/排他 OR 演算子(&, |, ^) c)式に'forever' と記述した場合、無限ループとなるコードを生成します。	
構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
do ++A0 during [A0 < 9]	__L_00000001: ; --- レジスタ退避 --- PUSH R0 PUSH RP0 ; --- インクリメント --- MOV.L (#__C_00000001),R0 ADD_R ; --- 結果保存 ---

```
MOV.L #__V_00000001,RP0
MOV A0,(RP0+)
; --- 結果更新 ---
PUSH DP0
MOV.L #__V_00000001,DP0
MOV (DP0+),A0
PUSH A0
POP A0
POP DP0
; --- レジスタ復帰 ---
POP RP0
POP R0
PUSH A0
PUSH RP0
MOV.L (#__V_00000001),A0
MOV.L #__V_00000002,RP0
MOV A0,(RP0+)
MOV.L #__V_00000003,RP0
MOV A0,(RP0+)
POP RP0
POP A0
; --- レジスタ退避 ---
PUSH A0
PUSH RP0
; --- オペランド2取得 ---
PUSH RP0
PUSH A0
MOV.L #__V_00000002,RP0
MOV.L (#__C_00000009),A0
MOV A0,(RP0+)
POP A0
POP RP0
; --- オペランド1取得 ---
PUSH A0
POP A0
; --- 比較実行 ---
PUSH A0
PUSH L0
PUSH L1
PUSH DP1
MOV A0,L0
MOV.L #__V_00000002,DP1
MOV (DP1+),L1
```

```
        CLAMP UNSIGNED
        POP DP1
        POP L1
        POP L0
        POP A0
        ; ---  比較結果保存  ---
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP UNDER, #__L_00000005
        ;---  比較結果保存  ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP #__L_00000006
__L_00000005:
        JMP #__L_00000001
__L_00000006:
__L_00000004:

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__C_00000009:       DATA H'00000009
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
__V_00000003:       DATA H'00000000
```

表 4.13 for 制御文

条件付き繰返し(初期化あり)：for … to … step … endfor
書式:
for△[式1]△to△[式2]△step△[式3] 　文

endfor

機能：

for文とは式2の評価結果が真である間、プログラムの制御を繰返し実行するためのものです。

for文では式2の継続条件に加えて、式1で初期値、式3で繰返し後の処理を指定することができます。そのため、プログラムの実行を一定回数繰り返す場合によく用いられます。

1) 式2の評価結果が真である間、文の記述内容を繰返し実行します。
2) 式1で初期値、式3で繰返し後の処理を指定することができます

仕様:

a)式 1 には for 文を制御する変数への初期値を代入する式を記述します。
b)式 1 の記述は省略することができます。その場合、'[]' と記述します。
c)式 2 には for 文の継続条件を記述します。
d)式 2 の記述を省略することもできます。その場合、'[]' と記述します。式 2 を省略した場合、式 1 は実行されますが、継続条件が常に偽となるため、式 3 および for 文内の文が実行されることはありません。
e)式 2 には変数、定数、演算子を記述することができます。
f)式 2 で使用できる演算子は以下のとおりです、
- ●関係演算子(<, >, <=, >=, <.s, >.s, <=.s, >=.s)
- ●等価演算子(==, !=)
- ●論理 AND/OR 演算子(&&, ||)
- ●ビット単位の AND/OR/排他 OR 演算子(&, |, ^)

g)式 3 には for 文を制御する変数への代入式を記述します。
h)式 3 の記述を省略することもできます。その場合、'[]' と記述します。
i)式 1 と式 3 にラベルは使用できません。
j)式 2 にラベルを記述した場合、ラベルのアドレスとみなします。
k)'endfor'を省略することはできません。

構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
for [A0 = 0] to [A0 < 9] step [++A0] ... endfor	; 式１ [A0 = 0] MOV.L (#__C_00000000),A0 PUSH RP0 MOV.L #__V_00000001,RP0 MOV A0,(RP0+) POP RP0 PUSH A0 PUSH RP0 MOV.L (#__V_00000001),A0 MOV.L #__V_00000002,RP0 MOV A0,(RP0+) MOV.L #__V_00000003,RP0 MOV A0,(RP0+) POP RP0 POP A0 JMP #__L_00000002 __L_00000001: ; 式３ [++A0]

```
        ; --- レジスタ退避 ---
        PUSH R0
        PUSH RP0
        ; --- インクリメント ---
        MOV.L (#__C_00000001),R0
        ADD_R
        ; --- 結果保存 ---
        MOV.L #__V_00000004,RP0
        MOV A0,(RP0+)
        ; --- 結果更新 ---
        PUSH DP0
        MOV.L #__V_00000004,DP0
        MOV (DP0+),A0
        PUSH A0
        POP A0
        POP DP0
        ; --- レジスタ復帰 ---
        POP RP0
        POP R0
        PUSH    A0
        PUSH    RP0
        MOV.L   (#__V_00000004),A0
        MOV.L   #__V_00000005,RP0
        MOV     A0,(RP0+)
        MOV.L   #__V_00000006,RP0
        MOV     A0,(RP0+)
        POP     RP0
        POP     A0
        JMP #__L_00000002
__L_00000002:   ; 式２ [A0＜9]
        ; --- レジスタ退避 ---
        PUSH A0
        PUSH RP0
        ; --- オペランド2取得 ---
        PUSH RP0
        PUSH A0
        MOV.L #__V_00000008,RP0
        MOV.L (#__C_00000009),A0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ; --- オペランド１取得 ---
        PUSH A0
```

```
        POP A0
        ; ---  比較実行  ---
        PUSH A0
        PUSH L0
        PUSH L1
        PUSH DP1
        MOV A0,L0
        MOV.L #__V_00000008,DP1
        MOV (DP1+),L1
        CLAMP UNSIGNED
        POP DP1
        POP L1
        POP L0
        POP A0
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000007,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP UNDER, #__L_00000006
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000007,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_00000007
__L_00000006:
        JMP #__L_00000001
__L_00000007:
__L_00000005:

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__C_00000009:       DATA H'00000009
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
__V_00000003:       DATA H'00000000
__V_00000004:       DATA H'00000000
```

	__V_00000005: DATA H'00000000 __V_00000006: DATA H'00000000 __V_00000007: DATA H'00000000 __V_00000008: DATA H'00000000

表 4.14 goto 制御文

無条件分岐：goto	
書式: goto△ラベル	
機能： goto文とは指定したラベルに制御を移すためのものです。 1) 指定したラベルにプログラムの制御を移します。	
仕様: a)goto の飛び先として指定したラベルが存在しなかった場合は、アセンブルエラーとなります。	
構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
goto _L1 ... _L1:	JMP #_L1 ... _L1:

表 4.15 continue 制御文

繰返し処理の継続：continue	
書式: continue	
機能： continue文とは、continueを記述している行を含む、最も内側の繰返し(while, do … during, for)から制御を再開させるためのものです。 1) continueを記述している行を含む、最も内側の繰返しから制御を再開します。	
仕様: a)繰返しが入れ子になっている場合でも、continue を記述している行を含む、最も内側の繰返しから制御を再開します。 b)continue が繰返しの外側で記述された場合、アセンブルエラーとなります。	
構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
while [forever] ...(1)...	__L_00000001:; while [forever] PUSH A0

if [A0 < 9] continue endif ...(2)... endwh	PUSH RP0 PUSH RP0 PUSH A0 MOV.L #__V_00000002,RP0 MOV.L (#__C_00000001),A0 MOV A0,(RP0+) POP A0 POP RP0 MOV.L (#__C_00000000),A0 PUSH DP0 MOV.L #__V_00000002,DP0 CMP (DP0+) POP DP0 MOV.L (#__C_00000001),A0 MOV.L #__V_00000001,RP0 MOV A0,(RP0+) POP RP0 POP A0 JMP NOT_ZERO, #__L_00000005 PUSH A0 PUSH RP0 MOV.L (#__C_00000000),A0 MOV.L #__V_00000001,RP0 MOV A0,(RP0+) POP RP0 POP A0 JMP #__L_00000003 __L_00000005: ...(1)... ; --- レジスタ退避 --- PUSH A0 PUSH RP0 ; --- オペランド2取得 --- PUSH RP0 PUSH A0 MOV.L #__V_00000002,RP0 MOV.L (#__C_00000009),A0 MOV A0,(RP0+) POP A0 POP RP0 ; --- オペランド 1 取得 --- PUSH A0 POP A0

```
        ; ---  比較実行  ---
        PUSH A0
        PUSH L0
        PUSH L1
        PUSH DP1
        MOV A0,L0
        MOV.L #__V_00000002,DP1
        MOV (DP1+),L1
        CLAMP UNSIGNED
        POP DP1
        POP L1
        POP L0
        POP A0
        ; ---  比較結果保存  ---
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施
        JMP UNDER, #__L_00000009
        ; ---  比較結果保存  ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP #__L_0000000a
__L_00000009:
        JMP #__L_00000001
        JMP #__L_00000008
__L_0000000a:
__L_00000008:
        ...(2)...
        JMP #__L_00000001
__L_00000003:
__L_00000004:       ; endwh

SECTION DATA NAME SAREA_DSPDATA
```

	__C_00000000: DATA H'00000000 __C_00000001: DATA H'00000001 __C_00000009: DATA H'00000009 __V_00000001: DATA H'00000000 __V_00000002: DATA H'00000000

表 4.16 break 制御文

繰返し処理、switch文の終了：break
書式:
break
機能：
break文には、以下の二つの役割があります。 1) breakを記述している行を含む、最も内側の繰返し(while, do … during, for)を中断します。 2) switch … case 文を終了し、'endsw' の次の行に制御を移します。
仕様:
a)繰返しが入れ子になっている場合でも、break を記述している行を含む最も内側の繰返しを中断します。 b)break が繰返しや switch 文の外側で記述された場合、アセンブルエラーとなります。 c)繰返しの中に switch 文を記述した場合でも、switch 文の中に記述した break は switch 文を終了させるものとして機能します。 d)switch 文中に繰返しを記述した場合でも、繰返しの中に記述した break は繰返しを終了させるものとして機能します。
構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
while [forever] ...(1)... if [A0 < 9] break endif ...(2)... endwh	; while __L_00000001: PUSH A0 PUSH RP0 PUSH RP0 PUSH A0 ; [forever] MOV.L #__V_00000002,RP0 MOV.L (#__C_00000001),A0 MOV A0,(RP0+) POP A0 POP RP0 MOV.L (#__C_00000000),A0 PUSH DP0 MOV.L #__V_00000002,DP0 CMP (DP0+) POP DP0 MOV.L (#__C_00000001),A0

```
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP NOT_ZERO, #__L_00000005
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_00000003
__L_00000005:
        ...(1)...
        ; if [ A0 < 9 ]
        ; ---  レジスタ退避  ---
        PUSH A0
        PUSH RP0
        ; ---  オペランド2取得  ---
        PUSH RP0
        PUSH A0
        MOV.L #__V_00000002,RP0
        MOV.L (#__C_00000009),A0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ; ---  オペランド 1 取得  ---
        PUSH A0
        POP A0
        ;---  比較実行  ---
        PUSH A0
        PUSH L0
        PUSH L1
        PUSH DP1
        MOV A0,L0
        MOV.L #__V_00000002,DP1
        MOV (DP1+),L1
        CLAMP UNSIGNED
        POP DP1
        POP L1
        POP L0
        POP A0
```

```
        MOV.L (#__C_00000001),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        JMP UNDER, #__L_00000009
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        JMP #__L_0000000a
__L_00000009:
        JMP #__L_00000004                    ; break
        JMP #__L_00000008
__L_0000000a:
__L_00000008:
        ...(2)...
        JMP #__L_00000001
__L_00000003:
__L_00000004:                    ; endwh

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__C_00000009:       DATA H'00000009
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
```

4.5.ビット操作命令

構造化記述では、以下のビット操作命令を使用することができます。

- ●ビットセット：bset
- ●ビットクリア：bclr
- ●ビットテスト：btst

表 4.17 bset命令

ビットセット：bset
書式:
1)bset△bpos, dest 2)bset△A0_*n*

機能：
bset 命令は引数で指定したレジスタ、変数の特定ビットを'1' に設定するためのものです。 1) bpos にはビット位置を示す即値、ビット位置を格納したレジスタ、変数のいずれかを指定します。 2) dest にはビット設定の対象となるレジスタ、および変数を指定します。 3) A0_*n* には任意のA0レジスタビット変数を指定します(A0_0～A0_31)。
仕様:
a)bpos で指定する値は 0～31 の範囲に納まる必要があります。bpos の値が範囲外となった場合、dest のビットは変化しません。 b)bpos が即値の場合で値が範囲外となった場合、アセンブルエラーとなります。 c)A0_n で n に指定する値は 0～31 の範囲に納まる必要があります。n の値が範囲外となった場合、アセンブルエラーとなります。

表 4.18 bclr命令

ビットクリア：bclr
書式:
1)bclr△bpos, dest 2)bclr△A0_*n*
機能：
bclr 命令は引数で指定したレジスタ、変数の特定ビットを'0' に設定するためのものです。 1) bpos にはビット位置を示す即値、ビット位置を格納したレジスタ、変数のいずれかを指定します。 2) dest にはビット設定の対象となるレジスタ、および変数を指定します。 3) A0_*n* には任意のA0レジスタビット変数を指定します(A0_0～A0_31)。
仕様:
a)bpos で指定する値は 0～31 の範囲に納まる必要があります。bpos の値が範囲外となった場合、dest のビットは変化しません。 b)bpos が即値の場合で値が範囲外となった場合、アセンブルエラーとなります。 c)A0_n で n に指定する値は 0～31 の範囲に納まる必要があります。n の値が範囲外となった場合、アセンブルエラーとなります。

表 4.19 btst命令

ビットテスト：btst
書式:
1)btst△bpos, dest 2)btst△A0_n
機能：
btst 命令は引数で指定したレジスタ、変数の特定ビットの値をチェックし、その値に応じてZフラグ、Oフラグを更新するためのものです。 1) bpos にはビット位置を示す即値、ビット位置を格納したレジスタ、変数のいずれかを指定します。 2) dest にはビット判定の対象となるレジスタ、および変数を指定します。 3) A0_*n* には任意のA0レジスタビット変数を指定します(A0_0～A0_31)。 4) 引数で指定したビットの値に応じて、Zフラグ、Oフラグは以下のように変化します Zフラグ：指定ビットが"0"のとき"1"、それ以外のとき"0"になります。 Oフラグ：指定ビットが"1"のとき"1"、それ以外のとき"0"になります。

仕様:

a)bpos で指定する値は 0～31 の範囲に納まる必要があります。
bpos の値が範囲外となった場合、以下のようになります。
bpos の値が即値：アセンブルエラーとなります。
bpos の値がレジスタまたは変数：dest の値は保証しません。
b)A0_n で n に指定する値は 0～31 の範囲に納まる必要があります。n の値が範囲外となった場合、アセンブルエラーとなります。

4.6.論理演算子を用いた、式の連結

構造化記述では、複数の式を論理演算子('&&', '||') を用いて連結することができます。

表 4.20 式の連結

構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)

if [A0 < M0 && A0 < R0]
...
endif
	; --- レジスタ退避 ---
	PUSH A0
	PUSH RP0
	;--- オペランド2取得 ---
	PUSH RP0
	PUSH A0
	PUSH M0
	POP A0
	MOV.L #__V_00000004,RP0
	MOV A0,(RP0+)
	POP A0
	POP RP0
	;--- オペランド1取得 ---
	PUSH A0
	POP A0
	; --- 比較実行 ---
	PUSH A0
	PUSH L0
	PUSH L1
	PUSH DP1
	MOV A0,L0
	MOV.L #__V_00000004,DP1
	MOV (DP1+),L1
	CLAMP UNSIGNED
	POP DP1
	POP L1
	POP L0
	POP A0
	; --- 比較結果保存 ---
	MOV.L (#__C_00000001),A0
	MOV.L #__V_00000001,RP0
	MOV A0,(RP0+)
	; --- レジスタ復帰 ---
	POP RP0
	POP A0

```
        ; ---  条件分岐実施  ---
        JMP UNDER, #__L_00000004        ; true side
        ; ---  比較結果保存  ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000001,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP #__L_00000007               ; false side
__L_00000004:
        ; ---  レジスタ退避  ---
        PUSH A0
        PUSH RP0
        ;---  オペランド2取得  ---
        PUSH RP0
        PUSH A0
        PUSH R0
        POP A0
        MOV.L #__V_00000005,RP0
        MOV A0,(RP0+)
        POP A0
        POP RP0
        ;---  オペランド1取得  ---
        PUSH A0
        POP A0
        ; ---  比較実行  ---
        PUSH A0
        PUSH L0
        PUSH L1
        PUSH DP1
        MOV A0,L0
        MOV.L #__V_00000005,DP1
        MOV (DP1+),L1
        CLAMP UNSIGNED
        POP DP1
        POP L1
        POP L0
        POP A0
        ; ---  比較結果保存  ---
        MOV.L (#__C_00000001),A0
```

```
        MOV.L #__V_00000002,RP0
        MOV A0,(RP0+)
        ; ---  レジスタ復帰  ---
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP UNDER, #__L_00000006        ; true side
        ; ---  比較結果保存  ---
        PUSH A0
        PUSH RP0
        MOV.L (#__C_00000000),A0
        MOV.L #__V_00000002,RP0
        MOV A0,(RP0+)
        POP RP0
        POP A0
        ; ---  条件分岐実施  ---
        JMP #__L_00000007               ; false side
__L_00000006:
        ; ...
        JMP #__L_00000003
__L_00000007:
__L_00000003:

SECTION DATA NAME SAREA_DSPDATA
__C_00000000:       DATA H'00000000
__C_00000001:       DATA H'00000001
__V_00000001:       DATA H'00000000
__V_00000002:       DATA H'00000000
__V_00000003:       DATA H'00000000
__V_00000004:       DATA H'00000000
__V_00000005:       DATA H'00000000
```

4.7.データおよびラベルの自動生成

構造化記述では、アセンブラコードを生成する際に以下の三種類のデータおよびラベルを生成します。

- ●定数データおよび定数データ参照用ラベル
- ●ワーク変数およびワーク変数参照用ラベル
- ●飛び先参照用ラベル

定数データおよびワーク変数は "SAREA_DSPDATA" という名前のデータセクションに追加されます。
SAREA_DSPDATAセクションが存在しない場合は、SAREA_DSPDATAセクションを自動生成します。
ただし、ソースコード上でリロケータブル配置のデータセクションが定義されている場合は、そのセクションの末尾に追加されます。
セクションのリロケータブル配置については 5.4「セクションについて」をご参照ください。

表 4.21 定数データおよび参照用ラベル

定数データおよび参照用ラベル:	
用途: if文における整数値との比較、for文における変数の初期化など、構造化記述において定数を使用した場合、その値を参照するために定数値データと参照用ラベルが生成されます。	
仕様: a)ラベル名の命名規則は以下のとおりです。 "__C_" + 定数値を８桁の１６進数で表現した文字列。 b)構造化記述で定数が 10 進数で表記された場合でも、定数データ参照用ラベルは 16 進数表記でラベル名を生成します。	
構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
A0 = 10	MOV.L(#__C_0000000a),A0 ;A0 = 10 ;;; 中略 ;;; SECTION DATA NAME SAREA_DSPDATA __C_0000000a:DATA H'0000000a ;10 の定数ラベル ;;; 後略 ;;;

表 4.22 ワーク変数および参照用ラベル

ワーク変数参照ラベル:
用途: 構造化記述においてワーク領域が必要になった場合、その値を変数として確保するためにワーク変数データと参照用ラベルが生成されます。
仕様: a)ラベル名の命名規則は以下のとおりです。 "__V_" + 内部 ID 値を８桁の１６進数で表現した文字列。
構造化記述展開例:

構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)

if [A0 < R0]
R0 = A0
else
A0 = R0
endif
	; if [A0 < R0]
	;--- レジスタ退避 ---
	PUSH A0
	PUSH RP0
	;--- オペランド2取得 ---
	PUSH RP0
	PUSH A0
	PUSH R0
	POP A0
	MOV.L #__V_00000002,RP0 ;オペランド2の値を保存するワーク変数
	MOV A0,(RP0+)
	POP A0
	POP RP0
	;--- オペランド1取得 ---
	PUSH A0
	POP A0
	;--- 比較実行 ---
	PUSH A0
	PUSH L0
	PUSH L1
	PUSH DP1
	MOV A0,L0
	MOV.L #__V_00000002,DP1
	MOV (DP1+),L1
	CLAMP UNSIGNED
	POP DP1
	POP L1
	POP L0
	POP A0
	;;; 中略 ;;;
	
	SECTION DATA NAME SAREA_DSPDATA
	__C_00000000: DATA H'00000000
	__C_00000001: DATA H'00000001
	__V_00000001: DATA H'00000000 ;ワーク変数のラベル
	__V_00000002: DATA H'00000000 ;ワーク変数のラベル
	__V_00000003: DATA H'00000000 ;ワーク変数のラベル
	;;; 後略 ;;;

表 4.23 飛び先参照用ラベル

飛び先参照用ラベル:	
用途:	
条件分岐や繰返しなどの制御構造を実現するため、構造化記述ではjmp命令の飛び先を示すラベルが自動的に生成されます。	
仕様:	
a)ラベル名の命名規則は以下のとおりです。 "__L_" + 内部 ID 値を８桁の１６進数で表現した文字列。	
構造化記述展開例:	
構造化記述	アセンブラ展開例 (アセンブラ展開例は現段階のサンプルであり、 今後変更になる場合があります。)
while [A0 < 9] ++A0 endwh	__L_00000001: ;while [A0 < 9] の先頭ラベル ;;; 中略 ;;; JMP #__L_00000001 ;endwh __L_00000006: __L_00000004: ;while [A0 < 9] の終端ラベル SECTION DATA NAME SAREA_DSPDATA ;;; 後略 ;;;

4.8.構造化記述で使用するスタック領域

構造化記述を使用する場合、ユーザが適切なスタック領域を確保し、スタック領域のアドレスをSP0レジスタに設定する必要があります。

構造化記述で必要なスタックサイズは以下のとおりです。ユーザアプリケーションでスタックを使用する場合、このサイズを加算したサイズをユーザにて確保してください。

●GREEN DSP Ver.2 向けのコード生成を行う場合：
(レジスタ個数 9 × 4 バイト) × 2 セット ＝72 バイト。

●GREEN DSP Ver.3 向けのコード生成を行う場合：
(レジスタ個数 10 × 4 バイト) × 2 セット ＝80 バイト。

表 4.24 スタック設定記述例

アセンブラ記述例:
SECTION CODE LOCATE H'mmm MOV #STACK_TOP,SP0; スタックポインタの初期化 … ; スタック領域確保(80バイト確保, GREEN DSP Ver.3向け) SECTION DATA LOCATE H'nnn STACK_END:DATA H'00000000

```
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
                DATA    H'00000000
STACK_TOP :DATA H'00000000
```

4.9.構造化記述のリストファイル出力

リストファイルには、構造化記述の記述内容と構造化記述によって生成されたアセンブラコードの両方を出力します。この際、構造化記述の内容を三重のセミコロンを付けて出力し、その次の行から生成されたアセンブラコードを出力します。

```
;
; ソースコードの記述
MOV    #val, DP0
A0 = (DP0)
MOV (DP0+),A0
```

```
;
; リストファイルの出力
MOV    #val, DP0

;;; A0 = (DP0)
PUSH DP0
MOV (DP0+),A0
POP DP0
PUSH RP0
MOV.L #__V_00000001,RP0
MOV A0,(RP0+)
POP RP0
PUSH A0
PUSH RP0
```

```
MOV.L (#__V_00000001),A0
MOV.L #__V_00000002,RP0
MOV A0,(RP0+)
MOV.L #__V_00000003,RP0
MOV A0,(RP0+)
POP RP0
POP A0
MOV (DP0+),A0
```

図 4.1 構造化記述のリストファイル出力例

5.アセンブル処理の概要

DSPASMのアセンブル処理では、ユーザが記述したアセンブラソースコード、ならびに構造化記述によって生成されたアセンブラコードをGREEN DSPの命令コードに変換し、変換した結果をファイルに出力します。

この章では、アセンブラ処理の概要を記述します。
アセンブラ処理の詳細につきましては、8「アセンブル処理の詳細」をご覧ください。

5.1.アセンブラコードの変換仕様

DSPASMはアセンブラソースコードに記述された命令をGREEN DSPの命令コードに変換します。
また、DSPコアバージョンによって、使用できるアセンブラコード、ならびに変換結果となる命令コードの値が異なります。
DSPコアバージョンと使用できるアセンブラコードの対応は以下のとおりです。

表 5.1 GREEN-DSP V2 コア指定時のアセンブラコード、および命令コード一覧

GREEN-DSP V2 コア: コマンドラインオプション "-core_version" の引数に "2" を指定した場合。				
分類	命令	オペランド1	オペランド2	命令コード(16進数)
転送	MOV	A0	M0	1
	MOV	A0	M1	2
	MOV	A0	R0	3
	MOV	A0	L0	5
	MOV	A0	L1	6
	MOV	#XX	DP0	80+XX
	MOV	#XX	DP1	84+XX
	MOV	#XX	RP0	88+XX
	MOV	(DP0+)	A0	7
	MOV	(DP0-)	A0	21
	MOV	(DP1+)	M0	8
	MOV	(DP1+)	M1	9
	MOV	(DP1+)	R0	0a
	MOV	(DP1+)	L0	0b
	MOV	(DP1+)	L1	0c
	MOV	A0	(RP0+)	0d
	MOV	A0	(RP0-)	22
算術演算	MUL			0e
	ADD			0f
	SUB			11
	MUL_ADD			12
	MUL_SUB			14
	LIMIT			1c
比較	CMP	(DP0+)		1d

分岐	JMP	OVER	#XX	cXXX
	JMP	UNDER	#XX	dXXX
	JMP	ZERO	#XX	eXXX
	JMP	NOT_ZERO	#XX	fXXX
	JMP	#XX		bXXX
IO	OUT	A0	(H'XX)	1eXX
	IN	(H'XX)	A0	1fXX
制御	NOP			0
	STOP			20
スタック操作	MOV	#XX	SP0	90+XX
	PUSH	A0		2c
	PUSH	M0		2d
	PUSH	M1		2e
	PUSH	R0		2f
	PUSH	L0		30
	PUSH	L1		31
	PUSH	DP0		32
	PUSH	DP1		33
	PUSH	RP0		34
	POP	A0		35
	POP	M0		36
	POP	M1		37
	POP	R0		38
	POP	L0		39
	POP	L1		3a
	POP	DP0		3b
	POP	DP1		3c
	POP	RP0		3d
サブルーチン	JSR	#XX		aXXX
	RET			3e
論理演算	OR	A0	R0	23
	AND	A0	R0	24
	XOR	A0	R0	25
	NOT	A0		26
	ABS	A0		27
	ABS_S	A0		4f
	SFT_RL			28
	SFT_RA			29
	SFT_LL			2a
	SFT_LA			2b
拡張転送	MOV	(XX, DP0)	A0	41XX
	MOV	(XX, DP1)	M0	42XX

	MOV	(XX, DP1)	M1	43XX
	MOV	(XX, DP1)	R0	44XX
	MOV	(XX, DP1)	L0	45XX
	MOV	(XX, DP1)	L1	46XX
	MOV	A0	(XX, RP0)	47XX
	MOV	(#XX)	A0	8C+XX
	MOV	(#XX)	M0	94+XX
	MOV	(#XX)	M1	98+XX
	MOV	(#XX)	R0	9C+XX
	MOV	SP0	RP0	48
	MOV	RP0	SP0	49
非飽和算術演算	MUL_R			4a
	ADD_R			4b
	SUB_R			4c
	MUL_ADD_R			4d
	MUL_SUB_R			4e
割り込み関連	CLI			51
	STI			52
	RETI			54
その他	CODE	H'XX		XX（オペランドの値）
	JIV			53

【注】表中の "定数+XX" は定数値とXXを加算した結果が命令コードとなります。また、"定数値XX"は定数値とXXを組み合わせて複数バイトの命令コードを生成することを意味します。

表 5.2 GREEN-DSP V3 コア指定時のアセンブラコード、および命令コード一覧

GREEN-DSP V3 コア: コマンドラインオプション "-core_version" の引数に "3" を指定した場合。				
分類	命令	オペランド1	オペランド2	命令コード(16進数)
転送	MOV	A0	M0	1
	MOV	A0	M1	2
	MOV	A0	R0	3
	MOV	A0	R1	4
	MOV	A0	L0	5
	MOV	A0	L1	6
	MOV	#XX	DP0	80+XX
	MOV	#XX	DP1	84+XX
	MOV	#XX	RP0	88+XX
	MOV.S	#XX	DP0	80+XX
	MOV.S	#XX	DP1	84+XX
	MOV.S	#XX	RP0	88+XX
	MOV.L	#XX	DP0	[68-6b]80+XX
	MOV.L	#XX	DP1	[68-6b]84+XX
	MOV.L	#XX	RP0	[68-6b]88+XX
	MOV	(DP0+)	A0	7
	MOV	(DP0-)	A0	21
	MOV	(DP1+)	M0	8
	MOV	(DP1+)	M1	9
	MOV	(DP1+)	R0	0a
	MOV	(DP1+)	L0	0b
	MOV	(DP1+)	L1	0c
	MOV	A0	(RP0+)	0d
	MOV	A0	(RP0-)	22
算術演算	MUL			0e
	ADD			0f
	SUB			11
	MUL_ADD			12
	MUL_ADD3			13
	MUL_SUB			14
	MUL_LIMIT			15
	LIMIT_ADD			16
	LIMIT_ADD3			17
	LIMIT_SUB			18
	MUL_LIMIT_ADD			19
	MUL_LIMIT_ADD3			1a
	MUL_LIMIT_SUB			1b

	MUX			0e
	ADD3			10
	MUX_ADD			12
	MUX_ADD3			13
	MUX_SUB			14
	MUX_CLAMP			15
	CLAMP_ADD			16
	CLAMP_ADD3			17
	CLAMP_SUB			18
	MUX_CLAMP_ADD			19
	MUX_CLAMP_ADD3			1a
	MUX_CLAMP_SUB			1b
	CLAMP			1c
	CLAMP	UNSIGNED		78
	LIMIT			1c
	LIMIT	UNSIGNED		78
比較	CMP	(DP0+)		1d
	CMP	UNSIGNED	(DP0+)	79
分岐	JMP	OVER	#XX	[64～67]cXXX
	JMP	UNDER	#XX	[64～67]dXXX
	JMP	ZERO	#XX	[64～67]eXXX
	JMP	NOT_ZERO	#XX	[64～67]fXXX
	JMP	#XX		[64～67]bXXX
IO	OUT	A0	(H'XX)	1eXX
	IN	(H'XX)	A0	1fXX
制御	NOP			0
	STOP			20
スタック操作	MOV	#XX	SP0	90+XX
	MOV.S	#XX	SP0	90+XX
	MOV.L	#XX	SP0	[6c-6f]90+XX
	PUSH	A0		2c
	PUSH	M0		2d
	PUSH	M1		2e
	PUSH	R0		2f
	PUSH	R1		3f
	PUSH	L0		30
	PUSH	L1		31
	PUSH	DP0		32
	PUSH	DP1		33
	PUSH	RP0		34
	POP	A0		35
	POP	M0		36

	POP	M1		37
	POP	R0		38
	POP	R1		40
	POP	L0		39
	POP	L1		3a
	POP	DP0		3b
	POP	DP1		3c
	POP	RP0		3d
サブルーチン	JSR	#XX		[64～67]aXXX
	RET			3e
論理演算	OR	A0	R0	23
	AND	A0	R0	24
	XOR	A0	R0	25
	NOT	A0		26
	ABS	A0		27
	ABS_S	A0		4f
	SFT_RL			28
	SFT_RA			29
	SFT_LL			2a
	SFT_LA			2b
拡張比較	CMPX	(DP0+)		50
	CMPX	UNSIGNED	(DP0+)	7a
拡張転送	MOV	(XX, DP0)	A0	41XX
	MOV	(XX, DP1)	M0	42XX
	MOV	(XX, DP1)	M1	43XX
	MOV	(XX, DP1)	R0	44XX
	MOV	(XX, DP1)	L0	45XX
	MOV	(XX, DP1)	L1	46XX
	MOV	A0	(XX, RP0)	47XX
	MOV	(#XX)	A0	8C+XX
	MOV	(#XX)	M0	94+XX
	MOV	(#XX)	M1	98+XX
	MOV	(#XX)	R0	9C+XX
	MOV.S	(#XX)	A0	8C+XX
	MOV.S	(#XX)	M0	94+XX
	MOV.S	(#XX)	M1	98+XX
	MOV.S	(#XX)	R0	9C+XX
	MOV.L	(#XX)	A0	[68-6b]8C+XX
	MOV.L	(#XX)	M0	[68-6b]94+XX
	MOV.L	(#XX)	M1	[68-6b]98+XX
	MOV.L	(#XX)	R0	[68-6b]9C+XX
	MOV	SP0	RP0	48

	MOV	RP0	SP0	49
非飽和算術演算	MUX_R			4a
	MUL_R			4a
	ADD_R			4b
	SUB_R			4c
	MUX_ADD_R			4d
	MUL_ADD_R			4d
	MUX_SUB_R			4e
	MUL_SUB_R			4e
割り込み関連	CLI			51
	STI			52
	RETI			54
拡張演算	INC	A0		55
	DEC	A0		56
	DIV			58
テーブル参照	MOV	(DP0+R0)	A0	57
拡張ビット操作	SFT_RL	n		70
	SFT_LL	n		71
	SFT_RA	n		76
	SFT_LA	n		77
	BTEST	n		72
フラグ操作	MOV	A0	F0	59
	MOV	F0	A0	5a
	FLGTEST	n		73
	FLGSET	n		74
	FLGCLR	n		75
セグメント操作	MOV	#XX	PS0	60+XX
	MOVP	#XX	PS0	64+XX
	MOV	#XX	DS0	68+XX
	MOV	#XX	SS0	6c+XX
その他	CODE	H'XX		XX（オペランドの値）
	JIV			53

【注】表中の "[定数1-定数2]"は、命令の対照アドレスが他のセグメントであった場合、命令コードの前にセグメントを指定するためのコードが出力されることを意味します。

5.2.アセンブラコード中のコメント

DSPASMでは、以下の記述をコメントとして解釈します。
1)";" から行末まで。
2)"//" から行末まで。
3)"/*" で始まり、"*/" で終わる一行以上の記述。

```
; この行はコメントです
nop     ; セミコロンから行末まではコメントです。

// この行もコメントです。
nop     // 二重のスラッシュから行末まではコメントです。

 /* 複数行に
 *  渡る
 *  コメントです。
 */
```

図 5.1 コメント記述例

コメントには日本語(いわゆる全角文字列)を用いることができます。その場合、日本語の文字コードにはUTF-8を使用してください。

5.3.データセクションのデータ定義

データセクションにおけるデータ定義の記述方法を以下に示します。

書式
[ラベル名文字列:]△DATA△[10進数、16進数 または #ラベル名文字列]△[コメント] ラベルを参照する場合は、ラベル名文字列の先頭に「#」を付けてください。
仕様:
a)参照するラベル名は、コードセクション内およびデータセクション内に定義されているラベルを記述することができます。定義されていないラベル名を記述した場合、アセンブルエラーとなります。
サンプルコード

```
SECTION CODE LOCATE H'0
SUB_START:
        (中略)

SECTION DATA LOCATE H'1000
S1_ST:  DATA 50                 ; DataA
        DATA H'100              ; DataB
        DATA 00000200h          ; DataC
        DATA #SUB_START         ; Address of SUB_START
        DATA #S1_ST             ; Address of S1_ST
```

5.4.アセンブラコード中の擬似命令

DSPASMでは、以下の擬似命令を使用することができます。
1).public
2).line

表 5.3 .public 擬似命令

.public 擬似命令
書式:
1).public△シンボル
機能：
.public 擬似命令で指定したシンボルを、ほかのモジュールから参照できるよう宣言します。
仕様:
a)シンボルにはラベルを指定することができます。
サンプルコード

```
; Sample code
;
SECTION CODE
      MOV #S1_ST, DP0
      MOV #S1_OUT, RP0
      ;
      MOV (DP0+), A0                ; DataB->A0
      MOV A0, R0                    ; DataB->R0
      MOV (DP0+), A0                ; DataA->A0
      ADD                           ; DataA + DataB
      ;
      MOV A0, (RP0+)                ; A0->DataC
      STOP
      ;
SECTION DATA
S1_ST:
            .public _DataB
_DataB:
            DATA H'00000000       ; DataB
            .public _DataA
_DataA:
            DATA H'00000000       ; DataA
;
S1_OUT:
            .public _DataC
_DataC:
            DATA H'00000000       ; DataC
```

表 5.4 .line 擬似命令

.line 擬似命令
書式:
1).line△"ファイル名",行番号
機能：
.line 擬似命令を使用することで、DSPASMに対して現在処理中のファイル名、ならびに行番号を指定する

ことができます。

1) 行番号は 1 から始まります。

仕様:

a)DSPASM は、.line 擬似命令で指定したファイルが実際に存在することを確認しません。
b)行番号に記述できる値は 1～100,000（10 万）の範囲に収まる必要があります。範囲外の値を記述した場合はアセンブルエラーとなります。

サンプルコード

```
.LINE    "sample.dsp" , 1       ;次の行が  sample.dspの1行目
; Sample code
;
SECTION CODE
      MOV #S1_ST, DP0
      MOV #S1_OUT, RP0
      ;
.LINE    "includesample.dsp" , 1       ;次の行が  includesample.dspの1行目
      MOV (DP0+), A0                  ; DataB->A0
      MOV A0, R0                      ; DataB->R0
      MOV (DP0+), A0                  ; DataA->A0
      ADD                             ; DataA + DataB
      ;
.LINE    "sample.dsp" , 7       ;次の行が  sample.dspの7行目
      MOV A0, (RP0+)
      STOP
      ;
SECTION DATA
S1_ST:   DATA H'00000050       ; DataB
         DATA H'00000100       ; DataA
         DATA H'00000003       ; Param_M0
;
S1_OUT: DATA H'00000000
```

5.5.セクションについて

アセンブラソースはコードセクションとデータセクションの2種類のセクションで構成されています。

コードセクションを定義する場合はセクション種別としてCODEを、データセクションを定義する場合にはDATAをSECTION文の中に指定します。

CODEセクション記述
書式:
1)SECTION△CODE△[NAME△セクション名]△[LOCATE△開始アドレス]
機能：
プログラムコードを配置するセクションを定義します。 1) セクション名には英数字、及びアンダーバーを利用することができます。 2) 開始アドレスには10進定数、および16進定数を利用することができます。
仕様:
a)セクション名は省略することができます。 b)セクション名を省略した場合、直前のセクション名を割り当てます。 ただし、直前にセクションがない場合、セクション名は、SAREA_DSPCODE となります。 c)開始アドレスは省略することができます。 d)開始アドレスを指定した場合、そのセクションはアブソリュート配置になります。 e)開始アドレスを指定しなかった場合、そのセクションはリロケータブル配置になります。 f)同じ名前のセクションを複数記述した場合、連続したアドレスになるように割り当てられます。 g)同じ名前のセクションが既に定義されている場合、開始アドレスを指定することができません。指定した場合、アセンブルエラーとなります。 h)データセクションとコードセクションで、セクション名が重複した場合はアセンブルエラーとなります。 i)複数のセクションでアドレス領域が重複した場合は、アセンブルエラーとなります。

DATAセクション記述
書式:
1)SECTION△DATA△[NAME△セクション名]△[LOCATE△開始アドレス]
機能：
プログラムデータを配置するセクションを定義します。 1) セクション名には英数字、及びアンダーバーを利用することができます。 2) 開始アドレスには10進定数、および16進定数を利用することができます。
仕様:
a)セクション名は省略することができます。 b)セクション名を省略した場合、直前のセクション名を割り当てます。 ただし、直前にセクションがない場合、セクション名は、SAREA_DSPDATA となります。 c)開始アドレスは省略することができます。 d)開始アドレスを指定した場合、そのセクションはアブソリュート形式で配置されます。 e)開始アドレスを指定しなかった場合、そのセクションはリロケータブル形式で配置されます。 f)同じ名前のセクションを複数記述した場合、連続したアドレスになるように割り当てられます。 g)同じ名前のセクションが既に定義されている場合、開始アドレスを指定することができません。指定した場合、アセンブルエラーとなります。 h)データセクションとコードセクションで、セクション名が重複した場合はアセンブルエラーとなります。 i)複数のセクションでアドレス領域が重複した場合は、アセンブルエラーとなります。

5.5.1.セクションの配置方法とセクションの個数

セクションはアブソリュート配置、もしくはリロケータブル配置を指定することができます。

リロケータブル配置を指定したセクションを定義した場合、そのセクション以外にセクションを定義することはできません。

アブソリュート配置を指定したセクションは、複数定義することができます。

コードセクションとデータセクションのそれぞれの組み合わせは以下の通りになります。

	コードセクションに 定義可能なセクション数	データセクションに 定義可能なセクション数
リロケータブルなコードセクション リロケータブルなデータセクション	1セクションのみ	1セクションのみ
リロケータブルなコードセクション アブソリュートなデータセクション	1セクションのみ	複数セクション定義可能
アブソリュートなコードセクション リロケータブルなデータセクション	複数セクション定義可能	1セクションのみ
アブソリュートなコードセクション アブソリュートなデータセクション	複数セクション定義可能	複数セクション定義可能

```
; Sample code
SECTION CODE NAME DSPCODE1   ;このセクションはリロケータブル配置になります
      NOP
      STOP

SECTION CODE NAME DSPCODE1   ;既に定義済みのセクションのみ、利用することができます。
      NOP                    ;なお、LOCATE は指定できません。
      STOP
;
SECTION DATA NAME DSPDATA1 LOCATE H'00004000   ;このセクションはアブソリュート配置
S0_ST:   DATA H'00000050                       ;になります。
S0_OUT: DATA H'00000000

SECTION DATA NAME DSPDATA2 LOCATE H'00005000
S2_ST:   DATA H'00000050
S2_OUT: DATA H'00000000

SECTION DATA NAME DSPDATA1   ;既に定義済みのセクションを利用することができます。
S1_ST:   DATA H'00000050     ;なお、LOCATE は指定できません。
S1_OUT: DATA H'00000000
```

図 5.2 セクション記述例1(コードセクション：リロケータブル、データセクション：アブソリュート)

```
; Sample code
SECTION CODE NAME DSPCODE1 LOCATE H'00006000   ;このセクションはアブソリュート
      NOP                                      ;配置になります
      STOP

SECTION CODE NAME DSPCODE2 LOCATE H'00007000
      NOP
      STOP

SECTION CODE NAME DSPCODE1   ;既に定義済みのセクションを利用する場合は、
      NOP                    ;LOCATE を指定できません。
      STOP
;
SECTION DATA NAME DSPDATA1   ;このセクションはリロケータブル配置になります
S0_ST:   DATA H'00000050
S0_OUT: DATA H'00000000

SECTION DATA NAME DSPDATA1   ;既に定義済みのセクションのみ、利用することが
S1_ST:   DATA H'00000050     ;できます。なお、LOCATE は指定できません。
S1_OUT: DATA H'00000000
```

図 5.3 セクション記述例2(コードセクション：アブソリュート、データセクション：リロケータブル)

5.5.2. 複数セクション定義時の注意事項

コード、データそれぞれで複数のセクションを記述した場合は、CPUのROMからDSPのSRAMへ転送する際に、セクション間に隙間があってもそのままの配置で転送する必要があります。
（ベースアドレスからのオフセットを変えないでください。）

5.6.命令コードの直接記述

DSPASMでは命令コードを直接記述する、"CODE H'xx" という記述をサポートします。この命令を記述した場合、引数で指定した値をそのままアセンブルコードとして出力します。

6.プリプロセス処理の詳細

この章では、3「プリプロセス処理の概要」 で取り上げなかった、プリプロセス処理の詳細について記述します。

6.1.定数式の演算子

条件付き取り込みプリプロセス命令：{TC}if の定数式で利用できる演算子は以下の通りです。

表 6.1 プリプロセス命令の定数式で使用可能な演算子

種別	演算子	内容
単項演算子	+	正数を示します
	−	負数を示します
	~	ビット毎の否定演算を行います（NOT）
乗除演算子	*	乗算を行います（符号なし）
	*.s	乗算を行います（符号あり、飽和演算）
	/	除算を行います（符号なし）
	%	剰余算を行います（符号なし）
加減演算子	+	加算を行います（符号なし）
	+.s	加算を行います（符号あり、飽和演算）
	−	減算を行います（符号なし）
	−.s	減算を行います（符号あり、飽和演算）
ビット単位のシフト演算子	<<	指定したビット分を論理左シフトします
	<<.s	指定したビット分を算術左シフトします
	>>	指定したビット分を論理右シフトします
	>>.s	指定したビット分を算術右シフトします
関係演算子	<	左辺が右辺より小さい場合に演算結果が真（符号なし）
	<.s	左辺が右辺より小さい場合に演算結果が真（符号あり）
	>	左辺が右辺より大きい場合に演算結果が真（符号なし）
	>.s	左辺が右辺より大きい場合に演算結果が真（符号あり）
	<=	左辺が右辺より小さいか等しい場合に演算結果が真（符号なし）
	<=.s	左辺が右辺より小さいか等しい場合に演算結果が真（符号あり）
	>=	左辺が右辺より大きいか等しい場合に演算結果が真（符号なし）
	>=.s	左辺が右辺より大きいか等しい場合に演算結果が真（符号あり）
等価演算子	==	左辺と右辺が等しい場合に演算結果が真
	!=	左辺と右辺が等しくない場合に演算結果が真
ビット単位のAND/OR/排他OR 演算子	&	ビット毎の論理積演算を行います（AND）
	|	ビット毎の論理和演算を行います（OR）
	^	ビット毎の排他的論理和演算を行います（XOR）
論理AND/OR演算子	&&	左辺と右辺の値を0 と比較してともに等しくない場合に真 短絡評価する(左辺が偽なら右辺を評価しません)

	||	左辺と右辺の値を0 と比較していずれか一方でも等しくない場合に真 短絡評価する(左辺が真なら右辺を評価しません)

また、演算子の優先順位は 4.3.1「演算子の優先順位」の記載と同様です。

7.構造化記述処理の詳細

この章では、4「構造化記述処理の概要」で取り上げなかった、構造化記述処理の詳細について記述します。

7.1.アドレス値の記述方法

アドレスポインタレジスタにはアドレス値を即値で記述することができます。ただし、A0 や M0 等の演算パラメータレジスタには記述できません。

演算パラメータレジスタにアドレス値を指定する場合は、アドレス値を代入したレジスタや変数を用いてください。

アセンブラ記述例:

MOV #H'000, DP0 ;アドレス値を即値(16進)で記述できます
MOV #100,DP0 ;アドレス値を即値(10進)で記述できます
;
MOV #H'00000000, A0 ;A0レジスタに対しては、アドレス値を即値で記述できません

7.2.構造化記述の制限事項

7.2.1.複数行にわたる式の記述

DSPASMは構造化記述の解析を行単位で行うため、複数行に渡る式を記述することはできません。
同様に構造化記述と判断するために必要な要素 (for 文であれば、予約語 "for" から、増分を指定する [式3]まで) も一行で記述する必要があります。

7.2.2.制御文中の演算子

構造化記述では真偽値を返す演算子は、制御文中の式でのみ利用することができます。
制御文以外の箇所で記述した場合、アセンブルエラーとなります。

真偽値を返す演算子種別	演算子
関係演算子	<, <.s, >, >.s, <=, <=.s, >=, >=.s
等価演算子	==, !=
論理 AND/OR 演算子	&&, ||

また、演算結果が真偽値となる式に対して論理AND/OR演算子以外の演算子を使用することはできません。
例えば、次のように記述した場合、アセンブルエラーとなります。

真偽値となる演算結果に対して続けて演算を行う記述例:
if [(A0 < R0) == 0] ; 関係演算「A0 < R0」の結果に対し等価演算を行っているため、 アセンブルエラーとなります。 (中略) endif

7.2.3.ビット操作命令

ビット操作命令は、制御文中の式で利用することができません。
また、他の演算子と組み合わせて利用することができません。

ビット操作命令	bset, bclr, btst

7.2.4.演算子が扱えない変数

演算子が使用できない変数について示します。
使用できない演算子を記述した場合はアセンブルエラーとなります。

種別	演算子	使用できない変数
単項演算子	+	ありません
	-	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	~	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O ポインタ変数 (即値, RP0)
	++	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (DP0), (DP1), (即値, DP0), (即値, DP1), (即値, RP0), (DP0+R0) 定数
	++.s	
	--	
	--.s	
乗除演算子	*	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	*.s	
	/	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	%	
加減演算子	+	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O
	+.s	

	-	A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	-.s	
ビット単位のシフト演算子	<<	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	<<.s	
	>>	
	>>.s	
関係演算子	<	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (DP0), (DP1), (即値, DP0), (即値, DP1), (即値, RP0), (DP0+R0)
	<.s	
	>	
	>.s	
	<=	
	<=.s	
	>=	
	>=.s	
等価演算子	==	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I ポインタ変数 (即値, RP0) フラグ変数 Z, U, O および A0レジスタビット変数は、0または1との比較しかできません。
	!=	
ビット単位のAND/OR/排他OR演算子	&	以下の変数は右辺に使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 A0_0 ～ A0_31 ポインタ変数 (即値, RP0)
	|	
	^	
論理AND/OR演算子	&&	以下の変数は使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I ポインタ変数 (即値, RP0)
	||	
代入演算子	=	以下の変数は左辺に使用できません。 フラグ変数 Z, U, O ポインタ変数 (DP0), (DP1), (即値, DP0), (即値, DP1), (DP0+R0) 以下の変数は右辺に使用できません。 レジスタ変数 PS0, DS0, SS0 フラグ変数 I, Z, U, O A0レジスタビット変数 ポインタ変数 (即値, RP0)
	*=	
	*=.s	
	/=	
	%=	
	+=	
	+=.s	
	-=	
	-=.s	

	<<=	以下の変数を左辺に使用する場合、右辺には定数以外使用できません。 レジスタ変数 PS0, DS0, SS0 以下の変数を左辺に使用する場合、右辺には 0 または 1 以外使用できません。 フラグ変数 I A0 レジスタビット変数 A0_0 ～ A0_31
	<<=.s	
	>>=	
	>>=.s	
	&=	
	|=	
	^=	
	複合代入(=..=)	

上記以外の、構造化記述の制限事項については10.2「構造化記述の翻訳限界」をご参照ください。

7.3.構造化記述の入れ子の交差

構造化記述で入れ子を交差することはできません。入れ子を交差して記述した場合はアセンブルエラーとなります。

表 7.1 構造化記述の入れ子が交差する例

アセンブラ記述例:
while [A<B] 　　if [A==C] 　　　　break endwh　　;;; この行でアセンブルエラーとなります。 　　endif

7.4.DSP コアバージョンによるコード生成の違い

構造化記述では、- core_version オプションで指定するDSPコアのバージョンによって、生成するアセンブラコードに以下の違いがあります。

コアバージョン2 ：V2	コアバージョン3 ： V3
1)乗算には MUL/MUL_R を使用します 2)データ転送には MOV を使用します 3)関係演算には LIMIT を使用します	1)乗算には MUX/MUX_R を使用します 2)データ転送には MOV.L を使用します 3)関係演算には CLAMP を使用します 4)V3 のみ以下の命令を使用します。 　　DIV 　　BTEST

7.5.構造化記述で使用可能な文字セット

構造化記述では以下の文字を使用することができます。

表 7.2 構造化記述で利用可能な文字セット

利用可能文字セット	
項目	値
英大文字	A B C D E F G H I J K L M N O P Q R S V W X Y Z
英小文字	a b c d e f g h i j k l m n o p q r s t u v w x y z
数字	0 1 2 3 4 5 6 7 8 9
特殊文字	! % & ( ) * + , - . / ; < = > [ ] ^ | ~
空白文字	スペース タブ
改行文字	CR LF

なお、構造化記述では英文字の大文字/小文字を区別しません。

7.6."( )" の使用箇所による意味の違い

"( )" は使用箇所により、以下のように意味が異なります。

"( )" の使用箇所	内容
制御文	演算の優先順位を変更します。
制御文以外	ポインタ変数として扱います。 4.1.4「ポインタ変数」に記載のない変数名を利用した場合はアセンブルエラーとなります。

サンプルコード

```
SECTION CODE
    if [(DATA2) == H'100]     // (DATA2) はポインタ変数として扱われないため、
                              // DATA2 == H'100 と同じ意味となり、結果は真となります。

        (DATA2) = H'100       // (DATA2) はポインタ変数として扱われるため、
                              // DATA2 に H'100 が設定されます
    endif
    ;
    for [R0 = (DATA1) * 2] to [R0 < 16] step [R0 += 4] // (DATA1)はポインタ変数として扱われ
                                                       // ないため、R0にはH'0000が設定されます。
        (DATA1) = R0    // (DATA1)はポインタ変数として扱われるため、
                        // DATA1から始まる4バイトの領域にR0の値が設定されます
    endfor
    STOP
;
SECTION DATA NAME DATASEC1 LOCATE H'0000
DATA1:
    DATA H'00000004
;
SECTION DATA NAME DATASEC2 LOCATE H'0100
DATA2:
    DATA H'00000200
;
```

7.7.V3 コア利用時の構造化記述使用時の注意事項

構造化記述では、生成するアセンブラコードで MOV.L命令を利用するため、構造化記述使用箇所でDS0レジスタの値が変更されることがあります。このため、ユーザがデータセグメントを指定する必要がある場合は、GREEN-DSPのソフトウェアマニュアルでデータセグメントの仕様を確認した後、必要に応じて以下の手順に従ってアセンブラコードを修正してください。

(ア)ソースコードをアセンブルし、リストファイルを出力します。
(イ)リストファイルの構造化記述部から、MOV.Lを用いている箇所を見つけます。
(ウ)そのMOV.L命令によってデータセグメントが意図しない値に書き換わっていないことを確認します。
(エ)もし、データセグメントの値が意図しない値に書き換わっている場合は、構造化記述部を抜けた後のMOV.S命令、およびMOV命令が意図しないセグメントを参照することになります。
(オ)そのため、それ以降のMOV.S命令、およびMOV命令をMOV.L命令に変更してください。

7.8.構造化記述で副作用のないコードを記述した場合の注意事項

DSPASMの構造化記述では、代入がない演算式などの「副作用のないコード」を記述した場合、アセンブルエラーとならずにワーク領域に演算結果の値をセットするアセンブラコードを出力します。

【副作用のない構造化記述の例】

```
A0 + R0
10
1 + 2
```

表 7.3 構造化記述で"10"を記述した場合の出力例

アセンブラ出力例:

```
  1                  : .line "C:¥dspasm¥sample_code.dsp" , 1
  2                  : SECTION CODE
  3                  :
  4                  : ;;; 10
  5 0000 2c          :  PUSH A0
  6 0001 34          :  PUSH RP0
  7 0002 688c00      :  MOV.L (#__C_0000000a),A0
  8 0005 688801      :  MOV.L #__V_00000001,RP0
  9 0008 0d          :  MOV A0,(RP0+)
 10 0009 3d          :  POP RP0
 11 000a 35          :  POP A0
 12 000b 2c          :  PUSH A0
 13 000c 34          :  PUSH RP0
 14 000d 688c00      :  MOV.L (#__C_0000000a),A0
 15 0010 688802      :  MOV.L #__V_00000002,RP0
 16 0013 0d          :  MOV A0,(RP0+)
 17 0014 3d          :  POP RP0
 18 0015 35          :  POP A0
 19                  :
 20                  : SECTION DATA NAME SAREA_DSPDATA
 21 0000 0000000a : __C_0000000a:DATA H'0000000a
 22 0004 00000000 : __V_00000001:DATA H'00000000
 23 0008 00000000 : __V_00000002:DATA H'00000000
```

8.アセンブル処理の詳細

この章では、5「アセンブル処理の概要」で取り上げなかった、アセンブル処理の詳細について記述します。

8.1.アセンブル処理の制限事項

アセンブル処理の制限事項については、10.3「アセンブル処理の翻訳限界」をご参照ください。

8.2.アセンブラ記述で使用可能な文字セット

アセンブラ記述では以下の文字を使用することができます。また、アセンブラ記述では文字の大文字/小文字を区別しません。

表 8.1 アセンブラ記述で利用可能な文字セット

利用可能文字セット	
項目	値
英大文字	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
英小文字	a b c d e f g h i j k l m n o p q r s t u v w x y z
数字	0 1 2 3 4 5 6 7 8 9
特殊文字	! " # $ ' * ( ) + , - . / : ; ? _ `
空白文字	スペース タブ
改行文字	CR LF

ラベル名またはセクション名に使用できる文字および記号の一覧を以下に示します。
ラベル名またはセクション名として認識できない文字が使用されている場合はアセンブルエラーとなります。

表 8.2 ラベル名・セクション名に使用できる文字および記号

英大文字	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
英小文字	a b c d e f g h i j k l m n o p q r s t u v w x y z
数字	0 1 2 3 4 5 6 7 8 9
特殊文字	! $ ' . ? _ `

8.3.アセンブラコード生成に関する補足

●JMP/JSR 命令について

JMP/JSR 命令の飛び先アドレスは、12 ビット幅で指定できる値となります。そのため、12 ビット幅を超えるアドレスが指定された場合は、指定した値を 0x0FFF でマスクした値が飛び先アドレスとなります。

また、V3 コアを使用する場合に限り、上位 4 ビット(0xF000 でマスクした値)はセグメント番号とみなします。

●OUT/IN 命令について

OUT/IN 命令のポート指定は、8 ビット幅で指定できる値となります。そのため、8 ビット幅を超えるポート番号が指定された場合は、指定した値を 0xFF でマスクした値がポート番号となります。

●セグメント操作命令(PS0, DS0, SS0 レジスタへの転送)について

セグメント操作命令で設定できるセグメントの値は、0～3 の範囲内である必要があります。この範囲を超えた値を指定した場合は、アセンブルエラーとなります。

●CODE 命令について

CODE 命令の出力コード指定は、8 ビット幅で指定できる値となります。そのため、8 ビット幅を超える出力コードが指定された場合は、指定した値を 0xFF でマスクした値が出力コードとなります。

●ラベル重複について

DSPASM ではラベルが重複した場合にアセンブルエラーとなります。データラベルとコードラベル間の重複もアセンブルエラーの対象となります。

●グローバルシンボルについて

ラベルはローカルシンボルになります。CPU 側オブジェクトファイルから参照することが出来ません。
CPU 側オブジェクトファイルから参照できるグローバルシンボルとするには以下のように .PUBLIC を利用してください。
CPU 側 C プログラムで変数として参照するには、C 言語仕様に合わせたシンボル名を利用してください。

アセンブラ記述例:

```
SECTION DATA
    .PUBLIC _DATA1      ; ラベル(_DATA1)をグローバルシンボルと宣言します。
_DATA1:
    DATA H'00000004
```

9.予約語

DSPASMで使用する予約語は以下のとおりです。

表 9.1 プリデファインドマクロ予約語(1)

__VA_ARGS__	__RENESAS_VERSION__
__RENESAS__	__DSPASM__

表 9.2 プリプロセッサ予約語(2)

{TC}define	{TC}ifdef
{TC}if	{TC}ifndef
{TC}else	{TC}elif
{TC}endif	{TC}include

表 9.3 構造化記述予約語(3)

if	endsw	for	break
elif	default	to	forever
else	while	step	bset
endif	endwh	endfor	bclr
switch	do	goto	btst
case	during	continue	

表 9.4 アセンブラ記述予約語(4)

section	.line
code	.public
data	
locate	
name	

表 9.5 アセンブラ命令予約語(5)

ABS	FLGSET	MUL_ADD_R	MUX_SUB_R
ABS_S	FLGTEST	MUL_LIMIT	NOP
ADD	IN	MUL_LIMIT_ADD	NOT
ADD3	INC	MUL_LIMIT_ADD3	OR
ADD_R	JIV	MUL_LIMIT_SUB	OUT
AND	JMP	MUL_R	POP
BTEST	JSR	MUL_SUB	PUSH
CLAMP	LIMIT	MUL_SUB_R	RET

CLAMP_ADD	LIMIT_ADD	MUX	RETI
CLAMP_ADD3	LIMIT_ADD3	MUX_ADD	SFT_LA
CLAMP_SUB	LIMIT_SUB	MUX_ADD3	SFT_LL
CLI	MOV	MUX_ADD_R	SFT_RA
CMP	MOV.L	MUX_CLAMP	SFT_RL
CMPX	MOV.S	MUX_CLAMP_ADD	STI
CODE	MOVP	MUX_CLAMP_ADD3	STOP
DEC	MUL	MUX_CLAMP_SUB	SUB
DIV	MUL_ADD	MUX_R	SUB_R
FLGCLR	MUL_ADD3	MUX_SUB	XOR

表 9.6 アセンブラレジスタ予約語(6)

A0	R1	PS0	I
M0	DP0	DS0	Z
M1	DP1	SS0	U
L0	RP0	BR0	O
L1	F0	PG0	
R0	SP0		

表 9.7 A0ビット変数予約語(7)

A0_0	A0_8	A0_16	A0_24
A0_1	A0_9	A0_17	A0_25
A0_2	A0_10	A0_18	A0_26
A0_3	A0_11	A0_19	A0_27
A0_4	A0_12	A0_20	A0_28
A0_5	A0_13	A0_21	A0_29
A0_6	A0_14	A0_22	A0_30
A0_7	A0_15	A0_23	A0_31

●その他予約語

上記の予約語に加えて、二重の下線("__")を含む語も予約語となります。

10.翻訳限界

DSPASMには以下の翻訳限界があります。

10.1.プリプロセス処理の翻訳限界

プリプロセス処理には以下の翻訳限界があります。

表 10.1 プリプロセス処理の翻訳限界

マクロ置き換えプリプロセス命令の記述数	1024個まで ※V1.03.00以前は512個まで
マクロ置き換え対象となる識別子の文字列長	200文字まで
関数形式マクロの仮引数の数	31個まで
条件付き取り込み ({TC}if/{TC}ifdef/{TC}ifndef/{TC}elif) の入れ子のレベル数	入れ子レベル31まで ※条件が成立したソースコードのみカウント対象となります
ファイル取り込み({TC}include)のネスト数	64回まで
1行の長さ(改行コード含まず)	2,048文字まで

これらの限界を超えた場合は、アセンブルエラーとなります。

表 10.2 入れ子のレベル数のカウント方法

```
#define   NUM   1
#if   #NUM == 1          ; 一番外側のプリプロセス命令は入れ子にカウントしません。
  #if   #NUM > 0         ; 入れ子レベル1
    #if   #NUM <= 2      ; 入れ子レベル2
    #endif
  #endif
#endif
```

また、条件付き取り込みプリプロセス命令により取り込み対象とならないソースコードでは、入れ子のレベル数はカウントされず、その中で入れ子レベルが31を超えていてもアセンブルエラーにはなりません。

表 10.3 条件付き取り込みプリプロセス命令の入れ子が翻訳限界を超える記述例

```
#define   NUM    0

#if   #NUM == 0
  #if   #NUM < 1              ; 入れ子レベル1
    ～省略～
      #if   #NUM < 31         ; 入れ子レベル31
          #if   #NUM < 32     ; 入れ子レベルが31を超えたため、この行でアセンブルエラーが発生します。
```

```
            ; 入れ子レベル32での処理
          #endif
        #endif
      ～省略～
    #endif
#endif
```

```
#define   ABC    0

#ifdef   ABCD              ; ABCDはマクロ定義されていないため、次の行からの入れ子は取り込み対象となりません。
                           ; #else 以降が取り込み対象となります。
  #ifdef   ABC             ; 入れ子レベル1
    ～省略～
      #ifdef   ABC         ; 入れ子レベル31
          #ifdef   ABC     ; 入れ子レベルが31を超えていますが、取り込み対象でないため
                           ; アセンブルエラーにはなりません
            ; 入れ子レベル32での処理
          #endif
      #endif
    ～省略～
  #endif
#else
  #ifdef   ABC             ; 入れ子レベル1
    ～省略～
      #ifdef   ABC         ; 入れ子レベル31
          #ifdef   ABC     ; 入れ子レベルが31を超えたため、この行でアセンブルエラーが発生します。
            ; 入れ子レベル32での処理
          #endif
      #endif
    ～省略～
  #endif
#endif
```

```
#define   NUM    31

#if   #NUM > 0
  #if   #NUM > 1       ; 入れ子レベル1
    ～省略～
      #if   #NUM > 31      ; 入れ子レベル31。この定数式が偽となるため、次の #if は取り込み対象となりません
        #if   #NUM > 32    ; 入れ子レベルが31を超えていますが、取り込み対象でないため
                           ; アセンブルエラーにはなりません
          ; 入れ子レベル32での処理
```

```
        #endif
      #elif  #NUM == 31
        #ifdef  NUM        ; 入れ子レベルが31を超えたため、この行でアセンブルエラーが発生します。
            ; 入れ子レベル32での処理
        #endif
      #else
            ; 入れ子レベル31での処理。ここは取り込み対象となりません。
      #endif
    ～省略～
  #endif
#endif
```

```
#define  ABC   0

#ifndef  ABCD            ; ABCDはマクロ定義されていないため、次の行からの入れ子は取り込み対象となります。
                         ; #else 以降は取り込み対象となりません。
  #ifdef  ABC            ; 入れ子レベル1
    ～省略～
      #ifdef  ABC        ; 入れ子レベル31
          #ifdef  ABCD   ; 入れ子レベルが31を超えたため、この行でアセンブルエラーが発生します。
                         ; 条件が偽となる場合でも入れ子としてカウントされます。
            ; 入れ子レベル32での処理
          #endif
      #endif
    ～省略～
  #endif
#else
  #ifdef  ABC            ; 入れ子レベル1
    ～省略～
      #ifdef  ABC        ; 入れ子レベル31
          #ifdef  ABC    ; 入れ子レベルが31を超えていますが、取り込み対象でないため
                         ; アセンブルエラーにはなりません
            ; 入れ子レベル32での処理
          #endif
      #endif
    ～省略～
  #endif
#endif
```

10.2.構造化記述の翻訳限界

構造化記述には以下の翻訳限界があります。

表 10.4 構造化記述の翻訳限界

1行の長さ(改行コード含まず)	2,048文字まで
制御文の入れ子のレベル数	入れ子レベル31まで ※入れ子の深さは制御文の種類にかかわらず、最大31までとなります。
1つの制御文で使用できる演算子の数	31個まで

これらの翻訳限界を超えた場合は、アセンブルエラーとなります。

10.3.アセンブル処理の翻訳限界

アセンブル処理には以下の翻訳限界があります。

表 10.5 アセンブラ記述の制限

1行の長さ(改行コード含まず)	2,048文字まで

これらの翻訳限界を超えた場合は、アセンブルエラーとなります。

11.エラーメッセージ

この章ではDSPASMが出力するエラーメッセージについて記述します。

11.1.エラーメッセージ形式

アセンブラの出力メッセージの形式はCubeSuite+の規定に従い、以下のフォーマットで出力されます。

(1) ファイルパスと行番号を含む場合

ファイルパス(行番号):メッセージ種別 コンポーネント番号 メッセージ番号:メッセージ

(2) ファイルパスと行番号を含まない場合

メッセージ種別 コンポーネント番号 メッセージ番号:メッセージ

なお、メッセージ種別、コンポーネント番号、およびメッセージ番号は連続した文字列として出力されます。

メッセージ種別	1文字の英字(エラーの場合は"E"、ワーニングの場合は"W")
コンポーネント番号	2桁の数値("05"固定)
メッセージ番号	5桁の数値(先頭の桁は"5"固定)

11.2.エラーメッセージ一覧

DSPASMで発生するエラーは以下のとおりです。
エラーが発生した場合、処理を中断します。ワーニングが発生した場合はメッセージの表示後、処理を継続します。

E0553001： command line option. (-format)	
内容	無効な-formatオプション指定。
表示情報	なし。
対処方法	OBJ/ASM/VERILOG のいずれかを指定してください。

E0553002： command line option. (-dsp)	
内容	無効な-dspオプション指定。
表示情報	なし。
対処方法	以下のいずれかを指定してください。 RX_DSP/RL78_DSP/RL78_101_DSP/RL78_111_DSP/RL78_IAR_DSP/RL78_LLVM_DSP RL78_GCC_DSP/ARM_DSP/ARM_EABI5_DSP

E0553003： command line option. (-core_version)	
内容	無効な-core_versionオプション指定。
表示情報	なし。
対処方法	2または3のいずれかを指定してください。

E0553004： command line option. (-text_macro)

内容	-text_macroオプションで指定可能な文字以外の文字が指定された。
表示情報	なし。
対処方法	# ' ` @ _ のいずれかを指定してください。

E0553005： command line option. (*指定されたオプション名*)	
内容	存在しないオプションを指定した。
表示情報	指定されたオプション名。
対処方法	オプションの指定をご確認ください。

E0553006： illegal file name.	
内容	アセンブリ言語ファイル名に何らかの異常があった(拡張子が異なるなど)。
表示情報	入力として指定したアセンブリ言語ファイル名。
対処方法	アセンブリ言語ファイルのファイル名をご確認ください。

E0553007： illegal include file name.	
内容	include対象となるファイル名に何らかの異常があった ('>' が見つからなかった。ファイル名が記載されていないなど)。
表示情報	include元のファイル名、エラーが発生した行番号
対処方法	インクルード対象ファイルのファイル名をご確認ください。

E0553008： source file open error.	
内容	アセンブリ言語ファイルを読み込むことができなかった。
表示情報	入力として指定したアセンブリ言語ファイル名、エラーが発生した行番号。
対処方法	アセンブリ言語ファイルが読み込み可能であるかをご確認ください。

E0553009： include file open error.	
内容	include対象となるファイルを読み込むことができなかった。
表示情報	include元のファイル名、エラーが発生した行番号
対処方法	インクルード対象ファイルが読み込み可能であるかをご確認ください。

E0553010： include file nesting over.	
内容	includeファイルのネスト数上限(64)を超えた。
表示情報	include元のファイル名、エラーが発生した行番号。
対処方法	includeファイルのネストが循環していないか確認してください。

E0553011： file write error.

内容	出力ファイルへの書き込みに失敗した。
表示情報	出力ファイル名。
対処方法	出力ファイルが書き込み可能であるかをご確認ください。

E0553012： define symbol not found.	
内容	{TC}defineの記述で識別子が指定されていない。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	識別子が定義されていることをご確認ください。

E0553013： text macro redefined.	
内容	テキストマクロを再定義した。 ※コマンドラインオプション "-allow_text_macro_redefine" を指定した場合、このエラーは発生しません。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	識別子の定義が重複していないことをご確認ください。

E0553014： illegal ifdef.	
内容	{TC}ifdefの記述で識別子が指定されていない。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	エラーが発生した行の記述内容をご確認ください。

E0553015： illegal ifndef.	
内容	{TC}ifndefの記述で識別子が指定されていない。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	識別子が定義されていることをご確認ください。

E0553016： illegal if.	
内容	{TC}ifの記述で定数式が指定されていない。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	定数式が記述されていることをご確認ください。

E0553017： illegal elif.	
内容	{TC}elifの記述で定数式が指定されていない。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	定数式が記述されていることをご確認ください。

E0553018： illegal else.	
内容	{TC}elseに対応するプリプロセス命令が見つからなかった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}if、ifdef、ifndefのいずれかを{TC}elseより前に記述してください。

E0553019： illegal endif.	
内容	{TC}endifに対応するプリプロセス命令が見つからなかった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}if、ifdef、ifndefのいずれかを{TC}endifより前に記述してください。

E0553020： if - endif not found.	
内容	{TC}ifに対応する〔TC〕endifが見つからなかった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}ifに対応する〔TC〕endifが記述されていることをご確認ください。

E0553021： ifdef - endif not found.	
内容	{TC}ifdefに対応する〔TC〕endifが見つからなかった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}ifdefに対応する〔TC〕endifが記述されていることをご確認ください。

E0553022： ifndef - endif not found.	
内容	{TC}ifndefに対応する〔TC〕endifが見つからなかった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}ifndefに対応する〔TC〕endifが記述されていることをご確認ください。

E0553023： can not allocate memory.	
内容	動的なメモリの確保に失敗した。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	Windows上で他のプログラムを実行している場合は、そのプログラムを終了してから、再度DSPASMを実行してください。 上記の対応に加えて、エラーが発生した行の記述内容もご確認ください。

E0553024： illegal define. (*識別子名*)	
内容	{TC}defineの記述に誤りがあった。
表示情報	ファイル名、エラーが発生した行番号、識別子名。
対処方法	エラーが発生した行の記述内容をご確認ください。

E0553025： illegal expression.	
内容	定数式の記述に誤りがあった。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	定数式の記述内容をご確認ください。

E0553026： constant value overflow.	
内容	定数の値が上限(4バイト)を超えている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	定数には0から4294967295までの値を指定してください。

E0553027： zero divide.	
内容	定数式の記述を処理する際に0除算が発生した。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	定数式の記述内容をご確認ください。

E0553028： unexpected EOF.	
内容	予期せぬ EOF を検出した。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	エラーが発生した行の記述内容について、以下の点をご確認ください。 ●複数行に渡るコメントが閉じられているか

E0553029： unknown section.	
内容	SECTION文のセクション種別に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	SECTION文にはCODEかDATAのどちらか一方のみを指定してください。

E0553030： unknown move operand 1.	
内容	転送命令の一つ目のオペランドに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	一つ目のオペランドの記述をご確認ください。

E0553031： unknown move operand 2.	
内容	転送命令の二つ目のオペランドに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	二つ目のオペランドの記述をご確認ください。

E0553032： unknown move operand 3.	
内容	転送命令の三つ目のオペランドに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	三つ目のオペランドの記述をご確認ください。

E0553033： unknown push operand.	
内容	PUSH命令のオペランドに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	オペランドの記述をご確認ください。

E0553034： unknown pop operand.	
内容	POP命令のオペランドに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	オペランドの記述をご確認ください。

E0553035： displacement error.	
内容	転送命令のセグメント番号または拡張転送命令のディスプレースメントの記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	セグメント番号は0から3まで、ディスプレースメントには-128から127までの値を指定してください。

E0553036： unknown port no.	
内容	IN命令およびOUT命令のポートアドレス指定に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ポートアドレスには0から255までの値を指定してください。

E0553037： code format error.	
内容	CODE命令の命令コードに誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	命令コードには0から255までの値を指定してください。

E0553038： unknown code. (*認識できないアセンブラ命令記述*)	
内容	認識できないアセンブラ命令記述がある。
表示情報	ファイル名、エラーが発生した行番号、認識できないアセンブラ命令記述。
対処方法	アセンブラ命令の記述をご確認ください。

E0553039： reserved symbol . (*シンボル名*)	
内容	シンボル名に予約語が含まれている。
表示情報	ファイル名、エラーが発生した行番号、シンボル名。
対処方法	予約語を含まない名前に変更してください。

E0553040： data format error.	
内容	データセクションに認識できない記述がある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	データセクションの記述をご確認ください。

E0553041： address(code) resolve error .	
内容	定義されていないコードラベルがプログラム領域で参照された。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ラベルの定義をご確認ください。

E0553042： address(code) format error.	
内容	プログラム領域を示すアドレスの記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	アドレスの記述をご確認ください。

E0553043： address(data) resolve error.	
内容	定義されていないデータラベルがプログラム領域で参照された。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ラベルの定義をご確認ください。

E0553044： address(data) format error.	
内容	データ領域を示すアドレスの記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	アドレスの記述をご確認ください。

E0553045： address resolve error.	
内容	定義されていないラベルがデータセクションで参照された。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ラベルの定義をご確認ください。

E0553046： data number error.	
内容	データセクションのDATAキーワードに続く数値の記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	数値の記述をご確認ください。

E0553047： section address format error.	
内容	SECTION文のLOCATEに続くアドレスの記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	アドレスの記述をご確認ください。

E0553048： code label defined.	
内容	ラベルの定義が重複している。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ラベルの定義をご確認ください。

E0553049： section name defined.	
内容	データセクション名とコードセクション名が重複している。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	セクション名は異なる名前を指定してください。

E0553050： code section address base error.	
内容	コードセクションのベースアドレスがプログラム領域外。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	コードセクションのアドレス指定をご確認ください。

E0553051： data label defined.	
内容	ラベルの定義が重複している。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ラベルの定義をご確認ください。

E0553052： section address overlapped.	
内容	セクションのアドレス範囲が重複している。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	各セクションのアドレス範囲をご確認ください。

E0553053： data section address base error.	
内容	データセクションのベースアドレスがデータ領域外。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	データセクションのアドレス指定をご確認ください。

E0553054： .LINE line number is out of range.	
内容	疑似命令 .LINE の行番号が範囲外。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	行番号には1から100,000までの値を指定してください。

E0553055： macro definition number over.	
内容	{TC}defineで定義されたマクロの数が上限(512)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	{TC}defineの記述数を512以内にしてください。

E0553056： identifier string size over.	
内容	{TC}defineで定義された識別子の文字列長が上限(200)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	識別子の文字列長をご確認ください。

E0553057： function macro arguments number over.	
内容	関数形式マクロの仮引数の数が上限(31)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	仮引数の数をご確認ください。

E0553058： condition directives nesting over.	
内容	条件付き取込み(if,ifdef,ifndef,elif,else)の入れ子の数が上限(31)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	条件付き取込み(if,ifdef,ifndef,elif,else)の入れ子の数を31以内にしてください。

E0553059： statement nesting over.	
内容	制御文(if,elif,else,switch,while,during,for)の入れ子の数が上限(31)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	制御文(if,elif,else,switch,while,during,for)の入れ子の数を31以内にしてください。

E0553060： operator number in a statement over.	
内容	1つの制御文で使用している演算子の数が上限(31)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	制御文で使用している演算子の数をご確認ください。

E0553061： line string size over.	
内容	1行の文字数が上限(2048)を超えた。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	1行の文字数を2048以内にしてください。

E0553062： command line option -define define symbol not found.	
内容	コマンドラインオプション -define の指定で識別子が指定されていない。
表示情報	なし。
対処方法	識別子が定義されていることをご確認ください。

E0553063： command line option -define text macro redefined. (*識別子名*)	
内容	コマンドラインオプション -define でテキストマクロを再定義した。 ※コマンドラインオプション "-allow_ text_macro _redefine" を指定した場合、 このエラーは発生しません。
表示情報	識別子名。
対処方法	識別子の定義が重複していないことをご確認ください。

E0553064： command line option -define illegal define. (*識別子名*)	
内容	コマンドラインオプション -define の指定に誤りがあった。
表示情報	識別子名。
対処方法	コマンドラインオプション -define の指定内容をご確認ください。

E0553065： command line option -define macro definition number over.	
内容	コマンドラインオプション -define で定義されたマクロの数が上限(512)を超えた。
表示情報	なし。
対処方法	コマンドラインオプション -define の指定数を512以内にしてください。

E0553066： command line option -define identifier string size over.	
内容	コマンドラインオプション -define で定義された識別子の文字列長が上限(200)を超えた。
表示情報	なし。
対処方法	識別子の文字列長をご確認ください。

E0553067： command line option -define function macro arguments number over. (*識別子名*)	
内容	コマンドラインオプション -define で、関数形式マクロの仮引数の数が上限(31)を超えた。
表示情報	識別子名。
対処方法	仮引数の数をご確認ください。

E0553068： command line option. (-output)	
内容	無効な-outputオプション指定 。
表示情報	なし。
対処方法	引数に文字列を指定してください。

E0553069： command line option. (-define)	
内容	無効な-defineオプション指定 。
表示情報	なし。
対処方法	引数に文字列を指定してください。

E0553070： command line option. (-inc_dir)	
内容	無効な-inc_dirオプション指定 。
表示情報	なし。
対処方法	引数に文字列を指定してください。

E0553071： out of section.	
内容	セクションが定義されていない箇所にコードが記述されている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	コードセクションまたはデータセクションの開始行より下に記述してください。

E0553074： not supported instruction.	
内容	コアバージョンV2でサポートされていない命令が記述されている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	サポートされていない命令を使わないようにするか、 -core_version オプションで3を指定してください。

E0553075： not supported register.	
内容	コアバージョンV2でサポートされていないレジスタが記述されている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	サポートされていないレジスタを使わないようにするか、 -core_versionオプションで3を指定してください。

E0553076： illegal operator. (*演算子*)	
内容	制御文の[ ]内に、使用できない演算子が記述されている。
表示情報	ファイル名、エラーが発生した行番号、演算子。
対処方法	制御文の式で使用可能な演算子のみ記述するようにしてください。

E0553077： section end address is out of range.	
内容	セクションの終了アドレスが、0xFFFFFFFFを超えている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	セクションの終了アドレスは0xFFFFFFFF以下になるようにしてください。

E0553078： end statement not found.	
内容	構造化記述の制御文に対するendがない。
表示情報	ファイル名。
対処方法	制御文に対応するend文を記述してください。

E0553079： invalid operand. (*演算子 オペランド*) ※演算子の右辺値が予期しないオペランドの場合 invalid operand. (*オペランド 演算子*) ※演算子の左辺値が予期しないオペランドの場合	
内容	構造化記述で演算子の左辺値または右辺値に予期しないオペランドが記述されている。
表示情報	ファイル名、エラーが発生した行番号、エラーが発生した演算子、 エラーとなったオペランド。
対処方法	演算子が扱えるオペランドを記述してください。

E0553080： section definition error.	
内容	アブソリュート配置のセクションとリロケータブル配置のセクションが混在している。 または、セクション名が異なるリロケータブル配置のセクションが2つ以上存在する。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	アブソリュート配置のセクションまたはリロケータブル配置のセクションのどちらか一方のみを定義してください。 リロケータブル配置のセクションはすべて同じ名前にしてください。

E0553081： invalid struct description.(*制御文*)	
内容	構造化記述の制御文が、期待する構造になっていない。
表示情報	ファイル名、エラーが発生した行番号、エラーとなった制御文。
対処方法	構造化記述で記載した制御構造が、「4.4 構造化記述で使用できる制御文」の通りであることを確認してください。

E0553082： DSP unsupported operation.(*オペランド、または演算子*)	
内容	オプション -core_versionで指定されるDSPバージョンに対して、使用できないオペランド、もしくは演算子を記述している。
表示情報	ファイル名、エラーが発生した行番号、エラーとなるオペランド、または演算子。
対処方法	サポートされていない演算子、またはオペランドを使わないようにするか、 -core_version オプションで3を指定してください。

E0553083： unknown pointer.	
内容	ポインタ変数の記述に誤りがある。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	ポインタ変数の記述をご確認ください。

W0553084： MOVP instruction was inserted before JMP/JSR instruction.	
内容	JMP/JSR命令の前にMOVP命令を追加しました。 ※このワーニングは、以下の二つの条件をともに満たす JMP/JSR命令 がある場合に発生します ●分岐先が同じセグメント内のアドレスである ●命令のアドレスの下位12ビットが 0xFFE または 0xFFF である
表示情報	ファイル名、ワーニングが発生した行番号。
対処方法	JMP/JSR命令の配置アドレスをご確認ください。

E0553085： command line option. (-code_section_start)	
内容	-code_section_startオプションで指定している割り付け開始アドレスの記述に誤りがある、または範囲外の値である。
表示情報	なし。
対処方法	割り付け開始アドレスの値をご確認ください。

E0553086： command line option. (-data_section_start)	
内容	-data_section_startオプションで指定している割り付け開始アドレスの記述に誤りがある、または範囲外の値である。
表示情報	なし。
対処方法	割り付け開始アドレスの値をご確認ください。

E0553087： An operator other than the logical AND/OR operator is used for the expression returning the boolean value.	
内容	真偽値を返す式に、論理AND/OR演算子以外の演算子が使われている。
表示情報	ファイル名、エラーが発生した行番号。
対処方法	演算結果が真偽値となる式を連結するには論理AND/OR演算子を使用してください。

E0553088： command line option. (-label)	
内容	無効な-labelオプション指定。
表示情報	なし。
対処方法	LOCAL/GLOBAL のいずれかを指定してください。

E0553090： command line option. (-macro_identify)	
内容	無効な-macro_identifyオプション指定。
表示情報	なし。
対処方法	FORWARD/EXACT のいずれかを指定してください。

E0553094： command line option. (-dwarf_spec)	
内容	無効な-dwarf_specオプション指定。
表示情報	なし。
対処方法	INITIAL/GENERIC/RENESAS のいずれかを指定してください。

E0553095： command line option. (-code_label_type)	
内容	無効な-code_label_typeオプション指定。
表示情報	なし。
対処方法	NOTYPE/FUNC のいずれかを指定してください。

E0553096： command line option. (-data_label_type)	
内容	無効な-data_label_typeオプション指定。
表示情報	なし。
対処方法	NOTYPE/OBJECT のいずれかを指定してください。

改訂記録	DSPASM ユーザーズマニュアル

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2016.09.09	ー	初版発行
1.01	2017.01.10	8	表 2.11 -text_macro コマンドラインオプション アセンブルエラーとなる記述についての説明を追記
		10	表 2.17 -E コマンドラインオプション -list コマンドラインオプションと併用した場合の説明を追記
		10	表 2.18 -cpuLittleEndian, -cpuBigEndian コマンドラインオプション -cpuLittleEndian コマンドラインオプションと-cpuBigEndian コマンドラインオプションを併用した場合の説明を追記
		18	4.1.3.A0 レジスタビット変数 アセンブルエラーとなる記述についての説明を追記
		19	表 4.6 ポインタ変数 ディスプレースメントを扱うポインタ変数についての説明を修正
		21	表 4.8 演算子 構造化記述で、演算子が扱えない変数を使用した場合の動作を追記
		36	表 4.14 for 制御文 ループ変数に関する説明を削除し、ラベルを使用した場合の説明を追記
		45	表 4.18 bset 命令 アセンブルエラーとなる記述についての説明を追記
		45	表 4.19 bclr 命令 アセンブルエラーとなる記述についての説明を追記
		45	表 4.20 btst 命令 A0 レジスタビット変数を指定する場合の書式を修正 アセンブルエラーとなる記述についての説明を追記
		50	4.7.データおよびラベルの自動生成 構造化記述で自動生成されるデータの追加先、および専用のデータセクションを生成する機能についての説明を追記
		69	7.2.2.制御文中の演算子 アセンブルエラーとなる記述についての説明を追記
		69	7.2.4.演算子が扱えない変数 構造化記述で、演算子の左辺値または右辺値に指定できない変数についての説明を追記
		72	7.4.DSP コアバージョンによるコード生成の違い コアバージョン 2 の不要な説明を削除 コアバージョン 3 でのみ生成される命令の誤りを修正 関係演算の生成コードの違いを追記
		78	表 10.2 入れ子のレベル数のカウント方法 表 10.3 条件付き取り込みプリプロセス命令の入れ子が翻訳限界を超える記述例 入れ子のカウント方法や記述例を追記
		82	11.2.エラーメッセージ一覧 -text_macro コマンドラインオプションで発生するエラー、「E0553004 : command line option. (-text_macro)」の内容と対処方法を修正

1.02	2017.09.01	7	2.4 DSPASM のコマンドラインオプション 表 2.7 コマンドラインオプション一覧 に以下のコマンドラインオプションを追加 ・-code_section_start ・-data_section_start ・-no_debug_info
		11	2.4 DSPASM のコマンドラインオプション 以下のコマンドラインオプションの説明を追加 ・-code_section_start ・-data_section_start ・-no_debug_info
		96	11.2.エラーメッセージ一覧 以下のエラーメッセージを追加 E0553085： command line option. (-code_section_start) E0553086： command line option. (-data_section_start)
1.03	2017.12.01	8～12	以下のコマンドラインオプションの書式説明に、パラメータ指定が無い場合アセンブルエラーとなる旨を追記 -format -output -text_macro -define -inc_dir -dsp -core_version -code_section_start -data_section_start
		27	表 4.10 switch … case 制御文 同じ値の case ラベルを複数記述した場合、アセンブルエラーとなる仕様について記載
		71	7.2.2. 制御文中の演算子 演算結果が真偽値となる式に演算子を続けて記述することが出来ない仕様について記載
		75	表 7.2 構造化記述で利用可能な文字セット アセンブラ記述でしか用いられない特殊文字を削除
		78	表 8.1 アセンブラ記述で利用可能な文字セット 構造化記述でしか用いられない特殊文字を削除
		78	表 8.2 ラベル名・セクション名に使用できる文字および記号 を追加
		99	11.2.エラーメッセージ一覧 以下のエラーメッセージを追加 E0553087：An operator other than the logical AND/OR operator is used for the expression returning the boolean value.
1.04	2019.02.01	9	表 2.7 コマンドラインオプション一覧 -dsp オプションの DSP 種別に ARM_EABI5_DSP を追加 -debug_aranges-no-padding オプションを追加
		12	表 2.15 -dsp コマンドラインオプション DSP 種別に ARM_EABI5_DSP を追加
		66	5.3 データセクションのデータ定義 データセクションでのデータ定義の記述方法について説明を追加
		88	11.2 エラーメッセージ一覧 エラーコード E0553002: command line option. (-dsp) の対処方法で指定する DSP 種別に ARM_EABI5_DSP を追加

1.05	2022.12.01	6	表 2.1 DSPASM の動作環境 Windows11 の環境を追加
		9-10	表 2.7 コマンドラインオプション一覧 -dsp オプションの DSP 種別に RL78_101_DSP, RL78_111_DSP, RL78_IAR_DSP, RL78_LLVM_DSP, RL78_GCC_DSP を追加 -label オプションを追加 -macro_identify オプションを追加 -dwarf_spec オプションを追加 -code_execinstr オプションを追加
		13	表 2.15 -dsp コマンドラインオプション DSP 種別に RL78_101_DSP, RL78_111_DSP, RL78_IAR_DSP, RL78_LLVM_DSP, RL78_GCC_DSP を追加
		16-17	表 2.23 -label コマンドラインオプションの説明を追加 表 2.24 - macro_identify コマンドラインオプションの説明を追加 表 2.25 - dwarf_spec コマンドラインオプションの説明を追加 表 2.26 - code_execinstr コマンドラインオプションの説明を追加
		19	表 3.2 備考(c) 説明文追加
		70	表 5.3 .public 擬似命令の追加
		85	8.3 アセンブラコード生成に関する補足 グローバルシンボルについての説明を追加
		86	表 9.4 アセンブラ記述予約語(4) .public を追加
		88	表 10.1 プリプロセス処理の翻訳限界 マクロ置き換えプリプロセス命令の記述数を 512 から 1024 に拡張
		92	11.1 エラーメッセージ形式 エラーメッセージ形式のファイル名をファイルパスに変更
		92,105	11.2 エラーメッセージ一覧 以下のエラーメッセージの説明を変更 E0553002： DSP 種別に RL78_101_DSP, RL78_111_DSP, RL78_IAR_DSP, RL78_LLVM_DSP, RL78_GCC_DSP を追加 以下のエラーメッセージを追加 E0553088： command line option. (-label) E0553090： command line option. (-macro_identify) E0553094： command line option. (-dwarf_spec)
1.06	2024.03.01	6	表 2.1 DSPASM の動作環境 サポート OS を Windows8.1/10/11 に変更
		10	表 2.7 コマンドラインオプション一覧 -code_label_type オプションを追加 -data_label_type オプションを追加
		18	表 2.27 -code_label_type オプションの説明を追加 表 2.28 -data_label_type オプションの説明を追加
		30,33	4.4.構造化記述で使用できる制御文 switch 制御文に default 句を追記
		106	11.2 エラーメッセージ一覧 以下のエラーメッセージを追加 E0553095： command line option. (-code_label_type) E0553096： command line option. (-data_label_type)

DSPASM
FAA/GREEN_DSP 構造化アセンブラ ユーザーズマニュアル

発行年月日　2024年03月01日　　Rev.1.06

発行　　ルネサス エレクトロニクス株式会社
　　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

DSPASM
FAA/GREEN_DSP 構造化アセンブラ
ユーザーズマニュアル

ルネサス エレクトロニクス株式会社

R20UT3911JJ0106