TOPPERS新世代カーネル用コンフィギュレータ内蔵

マクロプロセッサ仕様

作成： 2007年 4月 6日
改定14：2012年10月24日
株式会社きじねこ 高木信尚

目次

改定履歴

改定1	2007年 8月21日	組込み関数追記、誤記修正など
改定2	2007年10月25日	式を伴う$ERRORおよび$WARNINGの解説を追記 TEXT_LINE変数に関する記述を追記 書式調整、誤記修正、細部の表現修正
改定3	2007年12月10日	仕様書のタイトルを「TOPPERS新世代カーネル用コンフィギュレータ内蔵マクロプロセッサ仕様」に変更 5.3.3.1 単純記法に文字列定数の要素を追加 5.7.4 SORT関数の安定性に関する記述を追加 6.1.2 クラスおよびドメインを追加 書式調整、誤記修正
改定4	2008年 4月 1日	5.7.13. RANGE 関数を追加 5.7.14. SYMBOL 関数 5.7.15. PEEK 関数 6.1.1. 静的API名を用いた行番号を追記
改定5	2008年 4月18日	制御構造の*any-content* は省略できないことを追記 5.7.16. DUMP 関数 5.7.17. TRACE 関数 5.7.18. NOOP 関数
改定6	2008年 7月29日	本仕様書のフォントを「TOPPERS新世代カーネル用コンフィギュレータ」と同じになるように変更 5.3.3 順序付きリスト定数において、0個以上の”整数定数の並び”の記述を”0個以上の式の並び”に仕様変更 それに伴いBNFも変更
改定7	2008年12月16日	誤記修正 整数定数が文字列属性を持つことを明記 6.1.5.を追加 5.7.19.～5.7.21を追加

改定8	2009年 1月26日	2.1.3 文字列を追加 5.2 変数において、値属性に文字列を持つことができるように修正 5.4.4.2～5.4.4.3において、演算子記号を明確化 5.4.4.4 値・文字列変換演算子を追加 ユーザー定義関数導入に伴い、5.7 組込み関数の記述を変更 5.7.22 CALL関数を追加 5.7.23 LSORT関数を追加 5.8.4 ARGC変数を追加 5.8.5 ARGV変数を追加 5.8.6 RESULT変数を追加 6. TOPPERSカーネル固有の仕様の章番号を7.に変更 6. ユーザー定義関数を追加
改定9	2010年 2月23日	7.1.1.静的APIのパラメータ変数に ... 付きパラメータに関する記述を追加 7.1.6.ソフトウェア部品対応変数および7.1.7.ソフトウェア部品対応組込み関数を追加
改定10	2010年 7月23日	Ticket #42および#43に関する修正
改定11	2011年 3月 8日	5.6.1. 反復制御に$WHILE$および$JOINWHILE$命令を追加 5.6.2.1. IF命令に$ELIF$を追加 5.7.12. FIND関数に文字列と整数値の優先順位を追記 5.7.23. LSORT関数の記述を修正 5.7.24. ISFUNCTION関数を追加 7.1.1. 静的APIのパラメータ変数が、起動オプション--with-software-componentを指定した場合に設定されない旨の記述を追記 7.1.6.ソフトウェア部品対応変数の見出しを「静的API列挙変数」に変更
改定12	2011年 3月17日	2.2. 変数で、スコープの概念がないことを明記 5.7.25～5.7.27を追加 6.1. ユーザー定義関数の定義で、関数の呼び出し元に戻る時点で変数RESULTがクリアされることを追記

	7.1.1. 静的APIのパラメータ変数で、.ID_LISTが昇順に並んでいると変更
	7.1.6. 静的API列挙変数で、連想配列であることを明記
改定13　2012年 2月15日	5.8.7. CFG_PASS変数を追加
	7.1.1. 静的APIテーブルで末尾に … を付加したパラメータに関する記述を追記
改定14　2012年10月24日	5.7.8. APPEND関数が3つ以上の要素に対応したことに伴い、記述を修正

1. マクロプロセッサの概要

TOPPERS新世代カーネル向けコンフィギュレータでは、出力処理の大部分とエラー検出処理の一部を「マクロプロセッサ」を用いて実現している。新コンフィギュレータに内蔵されているマクロプロセッサは、本来ターゲット依存部の割り込みベクタ等の生成用に設計されたが、ごく単純な文法しかサポートしないにも関わらず、出力処理やエラー検出処理に必要なほとんどの機能を備えているため、ターゲット非依存部にも使用している。

マクロプロセッサは、「テンプレートファイル」と呼ばれるテキストファイルを読み込み、その内部に記述された2つの‘$’に挟まれたマクロ命令を処理する。また、マクロ命令の中では、算術演算、論理演算、シフト演算、比較演算等を行うことができるため、きめ細かい計算や条件判断が可能になっている。

2. 用語の定義

2.1 値

マクロプロセッサが扱うことができる値には、大きく分けて３種類がある。整数値、も順序付きリスト、および文字列である。

2.1.1. 整数値

マクロプロセッサでは、整数値はすべて64ビットの符号付き整数として扱い、その表現範囲は、-9,223,372,036,854,775,808〜+9,223,372,036,854,775,807である※1。また、負の整数値の内部表現は2の補数として扱う。演算の結果が整数値の表現範囲を超える場合はオーバーフロー・エラーが発生する。

※1 コンフィギュレータをビルドするための処理系は、64ビット整数型をサポートし、かつ負の整数値の内部表現が2の補数でなければならない。また、トラップ表現があってはならない。

2.1.2. 順序付きリスト

マクロプロセッサでは、順序付きリストと呼ばれる整数値の並びを扱うことができる。順序付きリストは必ずしも連続である必要はなく、また、値の大きさの順に（昇順であれ、降順であれ）整列している必要もない。順序付きリストの最大長に関する規定は行わない。

2.1.3. 文字列

マクロプロセッサでは、文字列を値として扱うことができる。表現可能な文字列長は、メモリ上の制約をのぞき、特に制限は設けない。また、文字コードに対する配慮は何ら行

わない。

2.2 変数

マクロプロセッサでは「変数」を使用することができる。マクロプロセッサが扱う変数には、単一の識別子によって表される「単純変数」と、識別子に添え字を伴う「連想配列」がある。

マクロプロセッサには変数のスコープ（有効範囲）の概念はない。

2.3 演算子

マクロプロセッサは、算術演算子、論理演算子、シフト演算子、等価演算子、関係演算子、代入演算子などをサポートする。それらの意味および動作は、ほぼC言語の演算子と同等である。ただし、C言語において動作が未定義となるような演算および処理系定義の結果となるような演算の一部はエラーとなる。

2.4 式

マクロ命令において、「変数」、「定数」、および「演算子」の組み合わせによって記述される処理単位。制御構造が要求する場合を除き、式の評価結果がそのまま出力される。

2.5 組込み関数

マクロ命令の式中で呼び出すことができる関数。現状のマクロプロセッサではユーザー定義の関数は存在しないため、単に「関数」といえば組込み関数のことを指す。

2.6 組込み変数

あらかじめ定義された変数。通常の変数と同じように扱うことができるが、マクロプロセッサが、処理の開始時に値および文字列属性を自動的に設定する。

2.7 制御構造

マクロプロセッサの制御構造には、大別して、選択制御、反復制御、および出力制御の3種類がある。これらの多くは、制御構造の開始位置から‘END’までのブロック構造になっている。

2.8 前処理

マクロプロセッサは、マクロ命令の処理に先立って前処理（プリプロセス）を行う。前処理では、他のファイルの取り込み、コメントの除去、行頭空白類文字の除去を行う。

2.9 エラー

マクロプロセッサは、入力ファイルの処理および内部状態の変化に応じて、エラーを報告することがある。エラーを報告した場合でも、継続が可能な限り、ただちに処理を停止することはない。ただし、標準エラー出力に対する出力を除き、処理結果をファイルに書き込むことはしない。

2.10 警告

マクロプロセッサは、入力ファイルの処理および内部状態の変化に応じて、警告を報告することがある。警告はエラーとは異なり、ユーザーに対して参考情報を提示するだけであり、以後の処理は通常通り行われる。

3. 処理過程

マクロプロセッサは、次の過程を経て入力ファイルを処理する。

3.1 コメントおよび行頭空白類の除去

入力ファイル中のコメントは単なる改行文字に置換される。また、行頭にある水平空白類文字は削除される。

3.2 他のファイルの取り込み

入力ファイル中にマクロ命令‘$INCLUDE$’が記述されている場合、そのマクロ命令を除去し、‘$INCLUDE$’命令で指定した取り込むべきファイルが存在する場合、‘$INCLUDE$’命令が存在していた箇所に、そのファイルに対して3.1および3.2の処理を再帰的に施した上で挿入する。取り込むべきファイルが存在しない場合、エラーを報告し、マクロプロセッサは処理を中断する。

3.3 構文解析

入力ファイル中のマクロ命令を構文解析し、構文木を生成する。構文に誤りが検出された場合、エラーを報告し、マクロプロセッサは処理を中断する。

3.4 マクロ命令の逐次実行

3.3で生成した構文木を順にたどり、各マクロ命令を逐次実行する。また、マクロ命令以外の部分はそのまま出力する。マクロ命令の実行中にエラーを検出した場合はその内容を報告するが、処理を継続可能な限り中断は行わない。ただし、処理の結果がファイルとして出力されることはない。

4. 前処理

マクロプロセッサの処理過程のうち、3.1および3.2の過程を「前処理」（プリプロセス）と呼ぶ。おおむねC言語の前処理に相当するが、（C言語における）マクロ定義や三文字表記などに相当する機能は存在しない。

4.1 コメント

マクロプロセッサは、入力ファイル中の、‘$’で始まり、水平空白類文字が続く行をコメントとして扱う。コメントは、処理過程の3.1で単なる改行文字に置換される。ブロックコメントまたは行途中から始まるコメントはサポートしない。

4.2 行頭空白類文字の除去

マクロプロセッサは、入力ファイルの各行に存在する水平空白類文字（空白文字と水平タブ）を除去する。行頭に水平空白類文字が複数存在する場合、それらすべてを除去する。

これにより、マクロプロセッサへの入力ファイルは、可読性向上のために適切な字下げを施すことができる。

4.3 他のファイルの取り込み

マクロプロセッサは、入力ファイル中にマクロ命令‘$INCLUDE$’を検出した場合、そのマクロ命令を除去し、‘$INCLUDE$’命令で指定した取り込むべきファイルが存在する場合、‘$INCLUDE$’命令が存在していた箇所に、そのファイルに対して3.1および3.2の処理を再帰的に施した上で挿入する。取り込むべきファイルが存在しない場合、エラーを報告し、マクロプロセッサは処理を中断する。

マクロ命令‘$INCLUDE$’の構文を下記に示す。

```
include-directive ::= '$' 'INCLUDE' '"' path-name '"' '$'
path-name ::= [^"]+
```

‘$INCLUDE$’命令で指定するパス名（ファイル名）では、ディレクトリの区切り子として、実行環境固有の文字を使用する必要がある。ただし、Windows環境の場合でも‘/’をディレクトリの区切り子として使用することができるため、極力‘/’を区切り子として使用することを推奨する。

‘$INCLUDE$’命令で指定したパス名の探索は次の手順で行われる。

1. カレントディレクトリを探索
2. `--include-path` オプション（または `-I` オプション）で指定したディレク

トリを指定した順に探索

C言語の `#include` 指令では、指令を記述したファイルと同じディレクトリが探索対象になる場合が多い（標準規格ではあくまでも処理系定義）が、マクロプロセッサではそうはならないため注意が必要である。

5. マクロ命令

マクロプロセッサは、‘$’で始まり‘$’で終わるマクロ命令を処理する。マクロ命令には制御構造や式などが存在する。マクロ命令は、ファイル中のあらゆる箇所で同じように解釈される。すなわち、引用符類や括弧類に囲まれた箇所であっても、マクロ命令は何ら変わりなく扱われる。なお、マクロ命令が入れ子になることはない。ただし、‘END’を伴うブロック構造の場合、ブロック内に他のマクロ命令が入れ子になることはある（マクロ命令そのものが入れ子になるわけではないため）。以下にマクロ命令の構文を示す。

```
macro-statement ::= doller-letter
                  | expression
                  | assignment-statement
                  | control-statement
```

5.1 ＄ 文字

マクロプロセッサへの入力ファイル中では‘$’は特別な文字として扱われる。そのため、マクロ命令の開始ではない単なる‘$’という文字を記述するにはエスケープする必要がある。単なる‘$’文字を記述するには、‘$$’のように2つの‘$’を連続して記述する。マクロプロセッサは‘$$’を内部的には単なる‘$’として処理する。以下に‘$’文字の構文を示す。

```
dollor-letter ::= '$$'
```

5.2 変数

マクロプロセッサが扱う変数は、入力ファイル全体で一意の名前を持っており、C言語等の一般的なプログラム言語に存在するような有効範囲（スコープ）、生存期間、および静的な型は持たない。

マクロプロセッサが扱う変数は、単一の識別子によって表される「単純変数」と、識別子に添え字を伴う「連想配列」に大別される。連想配列の添え字には変数を指定することもできるが、その場合、添え字となるのは変数の持つ値であって文字列ではない。また、

多次元の連想配列を作ることはできない。

変数の構文を以下に示す。

```
variable ::= identifier
           | identifier '[' expression ']'
identifier ::= non-digit [non-digit | digit | '.']*
non-digit ::= ['A'-'Z' | 'a'-'z' | '_']
digit ::= [0-9]
```

マクロプロセッサの変数は 0 個以上の要素を持つ。通常、変数は単一要素のみを持つが、複数要素を扱う場合は順序付きリストと呼ぶ。単一要素のみを持つ変数と、1 要素の順序付きリストは等価である。要素を代入されておらず、かつ組み込み変数ではない変数は、要素をひとつも持たない。

各要素は2種類の属性を持つ。ひとつは値（整数値または文字列）であり、もうひとつは文字列である。文字列は、例えばシステムコンフィギュレーションファイル中の静的APIのパラメータとして与えられた ”字面” を意味する。例えば、静的APIのパラメータが TA_HLNG|TA_ACT であった場合、そのパラメータを格納する変数は、”TA_HLNG|TA_ACT” という文字列と、2 という値の両方を持つことになる。

5.3 定数

マクロプロセッサが扱うことができる定数には整数定数、文字列定数、および順序付きリスト定数がある。C言語のような文字定数や浮動小数点定数は存在しない。また、整数定数には ‘L’ や ‘U’ などの（型を表すための）添え字を付けることもできない。以下に定数の構文を示す。

```
constant ::= integer-constant
           | string-constant
           | ordered-list-constant
```

5.3.1. 整数定数

整数定数はC言語と同様、10進数、16進数、および8進数で表記することができる。表記の方法もC言語と同様である。以下に整数定数の構文を示す。

```
integer-constant ::= dicimal-constant
                   | hexadecimal-constant
                   | octal-constant
```

```
decimal-constant ::= [‘1’-‘9’] [‘0’-‘9’]*
hexadecimal-constant ::= [‘0x’ | ‘0X’] [‘0’-‘9’ | ‘A’-‘F’ | ‘a’-‘f’]+
octal-constant ::= ‘0’ [‘0’-‘7’]*
```

整数定数は、ソース上の表記を文字列属性として、整数値を符号付き64ビット整数の値属性を持つ一種の変数として扱われる。ただし、整数定数に値を代入するなど、更新することはできない。整数定数が整数値の表現範囲を超える場合、マクロプロセッサはエラーを報告する。

5.3.2. 文字列定数

文字列定数は原則としてC言語と同様であるが、拡張ソース文字集合は考慮しない。また、三文字表記、二文字表記および国際文字名の変換も行わない。以下に文字列定数の構文を示す。

```
string-constant ::= ‘”’ [[^¥”] | escape-sequence]* ‘”’
escape-sequence ::= ‘¥a’ | ‘¥b’ | ‘¥f’ | ‘¥n’ | ‘¥r’ | ‘¥t’ | ‘¥v’
                  | ‘¥x’ | ‘¥¥’ | ‘¥”’ | ‘¥’’ | ‘¥?’
```

文字列定数がマクロプロセッサに評価されると、文字列定数に含まれるエスケープシーケンスはすべて展開される。

5.3.3. 順序付きリスト定数

順序付きリスト定数は、‘{’で始まり‘}’で終わる 0 個以上の式の並びである。順序付きリスト定数の要素には、単純記法と等差数列記法の2つの記法があり、順序付きリスト定数はその一方または両方を混在して記述することができる。順序付きリスト定数の途中で記法を返る場合は‘;’で区切る。以下に順序付きリストの構文を示す。

```
ordered-list-constant ::= ‘{’ ordered-list? ‘}’
orderd-list ::= [ordered-item-list | ordered-sequence]
                [‘;’ ordered-item-list | ordered-sequence]*
```

5.3.3.1. 単純記法

単純記法では、整数定数または文字列定数を‘,’で区切って記述する。記述した整数定数は、その順序で順序付きリスト定数の要素となる。以下に単純記法の構文を示す。

```
ordered-item-list ::= ordered-item [‘,’ ordered-item]*
```

ordered-item ::= *expression*

5.3.3.2. 等差数列記法

等差数列記法では、第1項、第2項、および最終項を記述する。第2項と第1項の差を公差とする等差数列として、第3項から最終項のひとつ手前の項までを補完する。第2項に公差の整数倍を加えた結果が最終項に等しくならない場合、マクロプロセッサはエラーを報告する。以下に等差数列記法の構文を示す。

ordered-sequence ::= *integer-constant* ‘,’ *integer-constant* ‘,’ ‘...’
‘,’ *integer-constant*

例)

$ABC = { 2,5,…,17 }$

という記述は、

$ABC = { 2,5,8,11,14,17 }$

と同義である。

5.4 式

式は、マクロ命令中において、定数、変数、および演算子を組み合わせることで、値の表現または計算を行う。以下に式の構文を示す。

expression ::= *logical-or-expression*

マクロプロセッサで用いる式は、C言語等の式とは異なり、副作用を生じることがない。式は常に評価結果を返す。

5.4.1. 式の評価結果

式の評価結果は無名の一時変数に格納されるものとする。したがって、式の評価結果は変数と同様、値と文字列の2つの属性を持つ。一部の例外を除き（→ 5.7.6 VALUE 関数）、何らかの演算を伴う式の評価結果では、文字列属性が失われ、値のみを保持することになる。

制御構造が要求する式（‘IF’ 命令の条件式など）の場合、その評価結果は制御構造ごとに定められた目的に使用される。

式がマクロ命令全体を構成する場合、その評価結果は、その時点で出力先として設定されているファイルに出力される※1。その際、式が文字列を保持しているならば、その文字列

が出力される。その変数が文字列を保持していない場合、値が文字列に変換された内容が出力される。値が文字列に変換される場合、整数定数は常に10進数で表現される。順序付きリストは、各要素は10進数として表現され、要素の区切りとして ‘,’ が出力される。

※1 これは式の副作用ではない。便宜的にここで解説しているが、式の評価結果が出力されるのはマクロ命令としての作用である。

5.4.2. 基本式

基本式は、より複雑な式の基本となる式であり、変数、定数、または括弧で囲まれた式のことである。以下に基本式の構文を示す。

```
primary-expression ::= variable
                     | constant
                     | `(' expression `)'
```

基本式の評価結果が演算子のオペランドとなる場合、結果が値を保持しない場合はエラーが報告される。ただし、関数の実引数であって、関数がその実引数の値を要求しない場合を除く。

5.4.3. 後置式

マクロプロセッサにおける後置式は、関数の呼び出しを指す。以下に後置式の構文を示す。

```
postfix-expression ::= primary-expression
                       | identifier `(' expression [`,' expression]* `)'
```

関数の呼び出しでは、第1実引数から順に評価が行われ、すべての実引数の評価が終わったのちに関数が呼び出される。関数の返却値が式の評価結果となる。

5.4.4. 単項式

単項式では、「算術符号演算子」、「補数演算子」、および「論理否定演算子」が解決される。以下に単項式の構文を示す。

```
unary-expression ::= postfix-expression
                   | unary-operator postfix-expression
unary-operator ::= `+' | `-' | `~' | `!' | `@'
```

5.4.4.1. 算術符号演算子

`+` 演算子はオペランドの値を評価し、その結果を式の評価結果とする。すなわち、オペランドが持つ文字列属性は破棄され、値属性を持たない場合はエラーを報告する。

`-` 演算子はオペランドの値を評価し、その符号を反転した結果を式の評価結果とする。結果が表現できない場合、すなわちオペランドの値が `-9,223,372,036,854,775,808` であった場合はエラーを報告する。

5.4.4.2. 補数演算子

補数演算子 `~` はオペランドの値を評価し、その1の補数、すなわち全ビットを反転した結果を式の評価結果とする。

5.4.4.3. 論理否定演算子

論理否定演算子 `!` はオペランドの値を評価し、その結果が0の場合は1を、0以外の場合は0を式の評価結果とする。

5.4.4.4. 値・文字列変換演算子

論理否定演算子 `@` はオペランドの値を文字列属性として返す。その際の値属性は空とする。

5.4.5. 乗除式

乗除式では、「乗算演算子」、「除算演算子」、および「剰余算演算子」が解決される。以下に乗除式の構文を示す。

```
multiplicative-expression ::= unary-expression
                        | multiplicative-expression '*' unary-expression
                        | multiplicative-expression '/' unary-expression
                        | multiplicative-expression '%' unary-expression
```

5.4.5.1. 乗算演算子

乗算演算子はかけ算を行う。オペランドの値を左辺、右辺の順に評価し、それぞれを乗じた結果を式の評価結果とする。結果が表現できない場合はエラーを報告する。

5.4.5.2. 除算演算子

除算演算子は割り算を行う。オペランドの値を左辺、右辺の順に評価し、左辺を右辺で割ったときの商を式の評価結果とする。割り切れない場合には、代数上の商から小数部分

を切り捨てた結果とする。右辺の評価結果が0の場合はエラーを報告する。

5.4.5.3. 剰余算演算子

除算演算子は割り算の余りを求める。オペランドの値を左辺、右辺の順に評価し、左辺を右辺で割ったときの剰余を式の評価結果とする。このとき、式 `( a / b )*b + a % b` は `a` に等しくなる。右辺の評価結果が 0 の場合はエラーを報告する。

5.4.6. 加減式

加減式では、「加算演算子」および「減算演算式」を解決する。以下に加減式の構文を示す。

```
additional-expression ::= multiplicative-expression
                | additional-expression '+' multiplicative-expression
                | additional-expression '-' multiplicative-expression
```

5.4.6.1. 加算演算子

加算演算子は足し算を行う。オペランドの値を左辺、右辺の順に評価し、両辺の和を式の評価結果とする。結果が表現できない場合はエラーを報告する。

5.4.6.2. 減算演算子

加算演算子は足し算を行う。オペランドの値を左辺、右辺の順に評価し、両辺の差を式の評価結果とする。結果が表現できない場合はエラーを報告する。

5.4.7. シフト式

シフト式では、「左シフト式」および「右シフト式」を解決する。以下にシフト式の構文を示す。

```
shift-expression ::= additional-expression
                  | shift-expression '<<' additional-expression
                  | shift-expression '>>' additional-expression
```

5.4.7.1. 左シフト演算子

左シフト演算子は、オペランドの値を左辺、右辺の順に評価し、左辺の評価結果を右辺で指定したビット数だけ左にシフトする。空いたビットには0を詰める。このとき、以下のいずれかの条件に合致する場合にはエラーを報告する。

- 左辺または右辺が負
- 右辺が64以上
- 結果が 左辺×$2^{右辺}$ で表現できない。

5.4.7.2. 右シフト演算子

右シフト演算子は、オペランドの値を左辺、右辺の順に評価し、左辺の評価結果を右辺で指定したビット数だけ右にシフトする。空いたビットには元の値の最上位ビットと同じ値を詰める。このとき、以下のいずれかの条件に合致する場合にはエラーを報告する。

- 右辺が負
- 右辺が64以上

5.4.8. 関係式

関係式は、4種類の「不等号演算子」を解決する。以下に関係式の構文を示す。

```
relational-expression ::= shift-expression
                        | relational-expression '<' shift-expression
                        | relational-expression '>' shift-expression
                        | relational-expression '<=' shift-expression
                        | relational-expression '>=' shift-expression
```

< （小さい）、> （大きい）、<=（以下）、>=（以上）の各演算子は、オペランドの値を左辺、右辺の順に評価し、両辺にその関係が成立する場合には1を、そうでない場合は0を式の評価結果とする。

5.4.9. 等価式

等価式は、「== 演算子」および「!= 演算子」を解決する。以下に等価式の構文を示す。

```
equality-expression ::= relational-expression
                      | equality-expression '==' relational-expression
                      | equality-expression '!=' relational-expression
```

==（等しい）、!=（等しくない）の各演算子は、オペランドの値を左辺、右辺の順に評価し、両辺にその関係が成立する場合には1を、そうでない場合は0を式の評価結果とする。

5.4.10. ビット単位の AND 式

ビット単位のAND式は、「& 演算子」を解決する。以下にビット単位のAND式の構文を示す。

```
and-expression ::= equality-expression
                 | and-expression '&' equality-expression
```

& 演算子はオペランドの値を左辺、右辺の順に評価し、ビット単位の論理積を行った結果を式の評価結果とする。

5.4.11. ビット単位の排他 OR 式

ビット単位の排他OR式は、「^ 演算子」を解決する。以下にビット単位の排他OR式の構文を示す。

```
xor-expression ::= and-expression
                 | xor-expression '^' and-expression
```

^ 演算子はオペランドの値を左辺、右辺の順に評価し、ビット単位の排他的論理和を行った結果を式の評価結果とする。

5.4.12. ビット単位の OR 式

ビット単位のOR式は、「| 演算子」を解決する。以下にビット単位のOR式の構文を示す。

```
or-expression ::= xor-expression
                | or-expression '|' xor-expression
```

| 演算子はオペランドの値を左辺、右辺の順に評価し、ビット単位の論理和を行った結果を式の評価結果とする。

5.4.13. 論理 AND 式

論理AND式は、「&& 演算子」を解決する。以下にビット単位の論理AND式の構文を示す。

```
logical-and-expression ::= or-expression
                         | logical-and-expression '&&' or-expression
```

&& 演算子は、まず左辺の値を評価し、その結果が0の場合には式の評価結果を0とし、右

辺の評価は行わない。左辺が0でない場合、右辺を評価してその結果が0の場合は式の評価結果を0に、そうでなければ式の評価結果を1にする。

5.4.14. 論理 OR 式

論理OR式は、「|| 演算子」を解決する。以下にビット単位の論理OR式の構文を示す。

```
logical-or-expression ::= logical-and-expression
                        | logical-or-expression '||' logical-and-expression
```

|| 演算子は、まず左辺の値を評価し、その結果が0以外の場合には式の評価結果を1とし、右辺の評価は行わない。左辺が0の場合、右辺を評価してその結果が0の場合は式の評価結果を0に、そうでなければ式の評価結果を1にする。

5.5 変数への代入

変数への代入では、変数に値または文字列のいずれかの属性を代入する。一度も代入を行っておらず、あらかじめ属性が設定される予約済み変数でもない変数を式中で用いた場合、その時点で変数が生成される。そのような変数は、値属性も文字列属性も持たない。以下に変数への代入の構文を示す。

```
assignment-statement ::= variable '=' expression
```

変数への代入では、まず右辺の式が評価され、その評価結果が左辺の変数に格納される。このとき、値属性および文字列属性の両方がコピーされる。

マクロプロセッサでは、変数への代入は式ではない。したがって、変数への代入を他の式の一部とすることはできない。また、変数への代入命令は評価結果をファイルに出力することもない。

5.6 制御構造

マクロ命令ではいくつかの制御構造を使用することができる。制御構造は、大別して、「反復制御」、「選択制御」、および「出力制御」の3種類がある。以下に制御構造の構文を示す。

```
control-statement ::= iteration-statement
                    | selection-statement
                    | output-statement
```

5.6.1. 反復制御

反復制御は繰り返しの処理を行う。反復制御には、‘$FOREACH$’命令、‘$JOINEACH$’命令、‘$WHILE$’命令および‘$JOINWHILE$’命令の4種類がある。以下に反復制御の構文を示す。

```
iteration-statement ::= foreach-statement
                      | joineach-statement
                      | while-statement
                      | joinwhile-statement
```

5.6.1.1. $FOREACH$ 命令

‘$FOREACH$’命令は、順序付きリストの各要素について反復処理を行う。以下に‘$FOREACH$’命令の構文を示す。

```
foreach-statement ::= foreach-directive any-content '$END$'
foreach-directive ::= '$FOREACH' variable expression '$'
any-content ::= macro-statement | plain-text
plain-text ::= [^'$']+
```

‘$FOREACH$’命令は、それ自身と‘END’命令の間にブロックを形成する。*any-content* を省略することはできない。‘$FOREACH$’命令では変数を1つ指定する必要がある。この変数をループ変数と呼ぶ。ループ変数に続いて指定した式を順序付きリストとみなし、その最初の要素をループ変数に格納する。そして、ブロックの内容を実行する。ブロックの内容からはループ変数を参照することができる。ブロックの内容の実行を終えると、‘$FOREACH$’命令はループ変数に順序付きリストの次の値を格納し、再びブロックの内容を実行する。これを順序付きリストの最後の要素まで繰り返す。ループ変数の後に指定した式が、（順序付きリストではなく）単なる整数値しか格納していない場合、要素が1つだけの順序付きリストであるとみなす。また、式が値を持たない場合、要素を持たない順序付きリストであるとみなす。‘$FOREACH$’命令では、ループを中断する手段は提供されない。

以下に‘$FOREACH$’命令の具体例を挙げる。

```
$result = 0$
$FOREACH i { 1,2,3,4 }$
$result = result + i$
$END$
```

上の例では、ループ変数‘i’に順序付きリスト 1,2,3,4 の値を順に格納し、$result = result + i$ を実行している。結果として、変数‘result’には、1から4の合計（= 10）が格納される。

5.6.1.2. $JOINEACH$ 命令

‘$JOINEACH$’命令は、’$FOREACH$’命令と同様、順序付きリストの各要素について反復処理を行う。以下に‘$JOINEACH$’命令の構文を示す。

joineach-statement ::= *joineach-directive any-content* ‘END’
joineach-directive ::= ‘$JOINEACH’ *variable expression delimiter* ‘$’
delimiter ::= *string-constant*

‘$JOINEACH$’命令では、「区切り子」と呼ばれる文字列定数を順序付きリストを表す式の後に指定する。ブロックの内容を実行するごとに、区切り子の内容を出力する。ただし、ループの最後では区切り子の内容は出力されない。その他の点においては、‘$FOREACH$’命令と同じである。

以下に‘$JOINEACH$’命令の具体例を挙げる。

```
$JOINEACH i { 3,7,1,3,0 } ”, ”$
(base + $i$)
$END$
```

上の例では、文字列定数 ”, ” が区切り子として扱われる。結果として、

```
(base + 3), (base + 7), (base + 1), (base + 3), (base + 0)
```

が出力される。

5.6.1.3. $WHILE$ 命令

‘$WHILE$’命令は、条件式が真に評価される間、反復処理を行う。以下に‘$WHILE$’命令の構文を示す。

while-statement ::= *while-directive any-content* ‘END’
while-directive ::= ‘$WHILE’ *expression* ‘$’
any-content ::= *macro-statement* | *plain-text*

plain-text ::= [^'$']+

'$WHILE$'命令は、それ自身と'END'命令の間にブロックを形成する。*any-content* を省略することはできない。'$WHILE$'命令では、反復ごとに条件式 *expression* を評価する。条件式が0でなければブロックの内容を実行する。'$WHILE$'命令では、ループを中断する手段は提供されない。

以下に'$WHILE$'命令の具体例を挙げる。

```
$n = 10$
$WHILE (n > 0)$
$n = n - 1$
$END$
```

上の例では、まず変数'n'に10を設定し、反復するごとにデクリメントしている。反復は変数'n'が0になるまで繰り返されるため、10回反復されることになる。

5.6.1.4. $JOINWHILE$ 命令

'$JOINWHILE$'命令は、'$WHILE$'命令と同様、条件式が0以外に評価される間、反復処理を行う。以下に'$JOINWHILE$'命令の構文を示す。

joinwhile-statement ::= *joinwhile-directive any-content* 'END'
joinwhile-directive ::= '$JOINWHILE' *expression delimiter* '$'
delimiter ::= *string-constant*

'$JOINWHILE$'命令では、'$JOINEACH$'命令と同様、「区切り子」を条件式の後に指定する。ブロックの内容を実行するごとに、区切り子の内容を出力する。ただし、ループの最後では区切り子の内容は出力されない。その他の点においては、'$WHILE'命令と同じである。

以下に'$JOINWHILE$'命令の具体例を挙げる。

```
$i = 0$
$JOINWHILE (i < 5) ", "$
(base + $i$)
$END$
```

上の例では、文字列定数 ", " が区切り子として扱われる。結果として、

```
(base + 0), (base + 1), (base + 2), (base + 3), (base + 4)
```

が出力される。

5.6.2. 選択制御

選択制御は条件判断を行う。選択制御には‘IF’命令がある。以下に選択制御の構文を示す。

```
selection-statement ::= if-statement
```

5.6.2.1. IF 命令

‘IF’命令は、条件式の真偽に応じて実行する内容を選択する。以下に‘IF’命令の構文を示す。

```
 if-statement ::= if-group '$END$'
                | if-group else-group '$END$'
                | if-group elif-group '$END$'
                | if-group elif-group else-group '$END$'
if-group ::= '$IF' expression '$' any-content
elif-group ::= '$ELIF' expression '$' any-content
             | elif-group '$ELIF' expression '$' any-content
else-group ::= '$ELSE$' any-content
```

‘$FOREACH$’命令同様、’IF’命令も‘END’命令との間にブロックを形成する。また、‘IF’命令と‘END’命令の間に‘$ELIF$’命令および‘$ELSE$’命令を記述することもできる。‘$ELIF$’命令は複数指定することができる。‘$ELSE$’命令は必ず最後に指定しなければならない。

‘IF’命令で指定した式の評価結果が0でなければ、直後の *any-content* の内容を実行する。

直前の‘IF’命令または‘$ELIF$’命令で指定した式の評価結果が0であり、かつ‘$ELIF$’命令がある場合、‘$ELIF$’命令で指定した式を評価する。式の評価結果が0でなければ、‘$ELIF$’命令の直後の *any-content* の内容を実行する。‘$ELIF$’命令が続く限り、この手順を反復する。

先行するすべての‘IF’命令および‘$ELIF$’命令で指定した式の評価結果が0であり、かつ‘$ELSE$’命令がある場合、‘$ELSE$’命令と‘END’命令の間の

any-content の内容を実行する。

5.6.3. 出力制御

出力制御は、ファイルまたは標準入出力への出力を制御する。出力制御には、‘$FILE$’命令、‘$ERROR$’命令および‘$WARNING$’命令がある。以下に出力制御の構文を示す。

```
output-statement ::= file-statement
                   | error-statement
                   | warning-statement
```

5.6.3.1. $FILE$ 命令

‘$FILE$’命令は、実行結果の出力先を制御する。以下に‘$FILE$’命令の構文を示す。

```
file-statement ::= '$FILE' string-constant '$'
```

‘$FILE$’命令で指定した文字列は出力先として設定するファイル名である。このファイル名は、基準となるディレクトリからの相対パスで指定する※1。ファイル名として ”stdout”が指定された場合、通常のファイルではなく標準出力が出力先となる。同様に、”stderr”が指定された場合、標準エラー出力が出力先となる。‘$FILE$’命令が記述されていない場合、出力先は標準出力に設定される。

以下に‘$FILE$’命令の具体例を挙げる。

```
$FILE "kernel_id.h"$
(A)
$FILE "kernel_cfg.c"$
(B)
```

上の例では、(A)の実行結果は“kernel_id.h”に、(B)の実行結果は“kernel_cfg.c”にそれぞれ出力される。

※1 TOPPERS/ASPカーネルの場合、--output-directory オプションで指定したディレクトリが基準となる。--output-directory オプションがない場合、カレントディレクトリが基準となる。

5.6.3.2. $ERROR$ 命令

‘$ERROR$’命令は、明示的にエラーを発生させる。以下に‘$ERROR$’命令の構文を示す。

```
error-statement ::= '$ERROR$' any-content '$END$'
                  |'$ERROR' expression '$' any-content '$END$'
```

'$ERROR$'命令は'END'命令との間にブロックを形成する。ブロックの内容は常に実行されるが、'$FILE$'命令による設定内容に関わらず、その出力先は標準エラー出力である。なお、*any-content* を省略することはできない。'$ERROR$'命令が実行されると、マクロプロセッサ自体がエラーを発生させた場合と同様に扱う。

'$ERROR$'命令に式を指定した場合、その式の評価結果の文字列属性をファイル名、値属性を行番号として、エラーメッセージに埋め込む。

5.6.3.3. $WARNING$ 命令

'$WARNING$'命令は、明示的に警告を発生させる。以下に'$WARNING$'命令の構文を示す。

```
warning-statement ::= '$WARNING$' any-content '$END$'
                    |'$WARNING' expression '$' any-content '$END$'
```

'$WARNING$'命令は、エラーではなく警告を発生させることを除いて'$ERROR$'命令と同等である。

5.7 組込み関数

マクロプロセッサは、あらかじめ定義された、いくつかの「組込み関数」を搭載している。組込み関数は、関数名に続けて、括弧で囲まれ、コンマで区切られた0個以上の実引数を伴って呼び出される（→ 5.4.3 後置式）。

組込み関数の評価結果は、式の評価結果と同様に扱うことができる。すなわち、値と文字列の2つの属性を持つ一時変数を生成する。組込み関数の評価結果のことを「返却値」と呼ぶ。

5.7.1. LENGTH 関数

'LENGTH'関数は、実引数として指定した順序付きリストの要素数を返す。実引数が順序付きリストではなく単なる整数値の場合は1を返す。実引数が無効な変数の場合は0を返す。実引数が文字列属性だけを持ち、値属性を持たない場合でも1を返す。

```
$LENGTH({ 1,2,3 })$
```
結果： 3

$**LENGTH**(123)$

結果： 1

$**LENGTH**(”abc”)$

結果： 1

$**LENGTH**(invalid)$

結果： 0

5.7.2. EQ 関数

‘EQ’関数は、2つの実引数の文字列属性を比較し、異なる場合は0を、そうでなければ1を返す。

$**EQ**(”ABC”, ”ABC”)$

結果： 1

$**EQ**(”ABC”, ”DEF”)$

結果： 0

5.7.3. ALT 関数

‘ALT’関数は、2つの実引数を受け取り、第1実引数が無効な関数の場合は第2実引数を、そうでなければ第1実引数を返却値として返す。

$valid = 1$

$**ALT**(valid, 2)$

結果： 1

$**ALT**(invalid,2)$

結果： 2

5.7.4. SORT 関数

‘SORT’関数は、2つの実引数を受け取り、第1実引数で指定した順序付きリストを、第2実引数で指定した文字列を名前とする連想配列の添え字とする。各連想配列の値を評価し、昇順になるよう第1実引数の順序付きリストを整列する。等価な要素の順序は保存される。

$A[0] = 3$

$A[1] = 5$

$A[2] = 1$

$**SORT**({ 0,1,2 }, ”A”)$

結果： 2,0,1

5.7.5. ENVIRON 関数

‘ENVIRON’関数は、実引数で指定した名前の環境変数を返す。環境変数の値は文字列属性として設定する。環境変数の値が整数定数の形式である場合、その文字列を整数値に変換した結果を値属性として設定する。ただし、整数値として扱うことができる範囲は、-2,147,483,647〜+2,147,483,647 であり、この範囲に収まらない値に関しては結果が保証されない（文字列属性は正しく設定される）。

5.7.6. VALUE 関数

‘VALUE’関数は、文字列属性と値属性を設定した一時変数を返す。組込み変数がマクロプロセッサによって定義される場合を除き、通常、変数には文字列または値のどちらかの属性しか設定することができない。‘VALUE’関数は、この原則に対する唯一の例外である。‘VALUE’関数では、第1実引数を文字列属性、第2実引数を値属性として扱う。実引数が無効な変数であった場合、その属性は設定されない。

```
$VALUE(”abc”, 123)$
結果: abc
$+VALUE(”abc”, 123)$
結果: 123
```

5.7.7. CONCAT 関数

‘CONCAT’関数は、2つの文字列を連結する。実引数が値属性しか持たない場合、値属性の整数値を10進数で表記したものを文字列として扱う。実引数が無効な変数の場合、空文字列として扱う。

```
$CONCAT(”abc”, ”def”)$
結果: abcdef
$CONCAT(”abc”, 123)$
結果: abc123
```

5.7.8. APPEND 関数

‘APPEND’関数は、順序付きリストを連結する。実引数が単なる整数値の場合、要素数1の順序付きリストとして扱う。実引数が無効な変数の場合、要素数0の順序付きリストとして扱う。実引数が文字列属性しか持たない場合、文字列属性のみを持つ要素数1の順序付きリストとして扱う。‘APPEND’関数を使うことにより、通常は生成することができない文字列の順序付きリストを生成することもできる。

$**APPEND**({ 1,2,3 }, { 4,5,6})$
結果： 1,2,3,4,5,6

5.7.9. AT 関数

‘AT’関数は、第1実引数で指定した順序付きリストの中の、第2実引数で指定した位置の要素を返す。指定した位置が不正な場合、無効な一時変数を返す。位置の指定では、先頭の要素を0として扱う。

$**AT**({ 1,2,3 }, 2)$
結果： 3

5.7.10. _ 関数

‘_’関数は、実引数で指定した文字列を、環境変数 TOPPERS_CFG_LANG で指定した言語に翻訳する。翻訳にあたっては、ファイル $(TOPPERS_CFG_LANG).po が参照される。ファイル $(TOPPERS_CFG_LANG).po は、通常マクロプロセッサの実行ファイルと同じディレクトリに、BOMを伴わないUTF-8で保存しなければならない。

5.7.11. FORMAT 関数

‘FORMAT’関数は、第 1 実引数で指定した書式文字列に従い、第 2 実引数以降を書式化する。書式文字列の記法は、おおむねC言語のprintf系関数と同じであるが、%n はサポートしていない。また、いくつかの拡張書式をサポートしている。

‘FORMAT’関数は、内部的にはboost::format クラスにより実現している。書式仕様の詳細については、http://www.boost.org/libs/format/doc/format.html#syntax を参照のこと。なお、書式の如何にかかわらず、第 2 実引数以降の実引数が文字列属性を持っていれば文字列として、値属性のみを持つ場合は整数値として扱う。

$X = VALUE(”abc”, 123)$
$**FORMAT**(”%d”, X)$
結果： abc
$X = VALUE(”abc”, 123)$
$**FORMAT**(”%d”, +X)$
結果： 123

上記の例では、書式文字列として %d を使用しているが、X が文字列属性を持つため、そのまま実引数とした場合には 123 ではなく abc が展開される。実引数を +X とした場

合、+ 演算子の評価結果は文字列属性を失い、値属性だけとなるため、123 に展開される。

```
$FORMAT("%2% is %1%", "abc", "def")$
```
結果：def is abc

‘FORMAT’関数の拡張書式により、上記のように、書式文字列中で展開すべき実引数を指定することができる。%1% は第 2 実引数を、%2% は第 3 実引数を表す。以下、同様に、%3%, %4%, … を用いることができる。これにより、‘_’関数によって日本語等に翻訳を行う場合、それぞれの言語の文法に基づき語順を制御することができる。

%1% 等の書式では、数値を 16 進数に変換するなどの指定ができないため、詳細な指定が必要な場合は、以下のようにする。

```
$FORMAT("%2$x, %1$o", +123, +456)$
```
結果：1c8, 173

上記のように、% の直後に 1$ 等の文字列を含むことで、展開すべき実引数を指定し、継続する文字列で 16 進変換や最小フィールド幅等の通常の書式指定を行うことができる。

5.7.12. FIND 関数

‘FIND’関数は、第1マクロ実引数で指定した順序付きリストに含まれる第2マクロ実引数で指定した値に等しい要素を、先頭から順に探索する。等しい要素が見つかればその要素へのインデックスを、そうでなければ空値を返す。

第2マクロ実引数で指定した値が、文字列と整数値の両方を持つ場合、整数値を指定したものとして扱われる。

```
$FIND({ 1,2,3,4,5 }, 3)$
```
結果：2

5.7.13. RANGE 関数

‘RANGE’関数は、第1マクロ実引数で最初の値を、第2マクロ実引数で最後の値を指定することで、

{ 最初の値, 最初の値 + 1, … 最後の値 }

となる順序付きリストを生成する。第1マクロ実引数の値は第2マクロ実引数の値以下（同じでもよい）でなければならない。マクロ実引数が正しくない場合は空値を返す。

$**RANGE**(3, 6)$
結果：{ 3, 4, 5, 6 }

5.7.14. SYMBOL 関数

‘SYMBOL’関数は、第1マクロ実引数で指定した文字列（シンボル）に対応するアドレスを整数値で返す。シンボルとアドレスの対応付けは、--symbol-table（または-s）オプションで指定したシンボルテーブル（nmコマンドの出力形式）によって行う。

$**SYMBOL**(”_kernel_tinib_table”)$
結果：_kernel_tinib_tableのアドレス

この関数は、シンボルテーブルから単純に対応するアドレスを取得する機能しか持たない。DATAセクションの転送等により、LMA→VMAの変換が必要になる場合は必要な変換処理を記述する必要がある。

5.7.15. PEEK 関数

‘PEEK’関数は、第1マクロ実引数で指定したアドレス（整数値）から始まる第2実引数で指定したバイト数（整数値）の値を整数値として読み込む。値の読み込みには--rom-image（または-r）オプションで指定したSレコードを使用する。バイトオーダーは自動的に判別する。なお、不正なアドレスを指定した場合の動作は未定義とする。

$**PEEK**(0xc000, 4)$
結果：0xc000番地から4バイトを読み込んだ値

5.7.16. DUMP 関数

‘DUMP’関数は、マクロプロセッサが管理している変数の内容をダンプする。ダンプ出力の形式は、

$変数名$ = { 文字列属性(値属性) }

となる。順序付きリストなど、変数に複数の要素が含まれる場合には、各要素をコンマ , で区切って出力する。

ダンプの出力先はマクロ実引数で指定した名前のファイルのとする。ただし、”stdout”を指定した場合は標準出力、”stderr”を指定した場合は標準エラーに出力する。なお、マクロ実引数を省略した場合は”stderr”を指定したものとして扱う。

‘DUMP’関数は ”” 文字列を返す。

5.7.17. TRACE 関数

‘TRACE’関数は、変数の内容をトレース出力する。

第1マクロ実引数ではトレース対象の変数を指定する。トレース出力の形式は、

```
{ 文字列属性(値属性) }
```

となる。順序付きリストなど、変数に複数の要素が含まれる場合には、各要素をコンマ , で区切って出力する。

第2マクロ実引数では出力先のファイル名を指定する。ただし、”stdout”を指定した場合は標準出力、”stderr”を指定した場合は標準エラーに出力する。なお、第2マクロ実引数を省略した場合は”stderr”を指定したものとして扱う。

‘TRACE’関数は ”” 文字列を返す。

5.7.18. NOOP 関数

‘NOOP’関数は何も行わない。マクロ実引数の個数や内容のチェックも行わない。

‘NOOP’関数は ”” 文字列を返す。

5.7.19. BCOPY 関数

指定したコピー元アドレスからコピー先アドレスへ、指定バイト数のメモリブロックを転送する。この関数は、LMAからVMAへのアドレス変換を目的として使用することを想定している。

5.7.20. ESCSTR 関数

指定した文字列を二重引用符で囲む。同時に、文字列中に含まれる二重引用符および制御文字をC言語における拡張表記に置換する。

5.7.21. UNESCSTR 関数

指定した二重引用符で囲まれた文字列を展開する。両端の二重引用符を除去し、文字列中に含まれるC言語の拡張表記を実際の文字に置換する。

5.7.22. CALL 関数

第一引数で指定した名前の可変関数を、第二引数以降で指定した引数を伴い呼び出す。

```
$CALL(”foo”, 1, 2, 3)$
```

は
`$foo(1, 2, 3)$`
と等価である。

‘CALL’関数は、ユーザー定義関数をコールバックさせるために使用することを想定している。

5.7.23. LSORT 関数

第一引数で指定した順序付きリストを、第二引数で指定したユーザー定義関数を用いて整列したものを返す。等価な要素の順序は保存される。

第二引数で指定するユーザー定義関数は、順序付きリスト内の２要素を比較するためのものであり、組み込み変数ARGV[1]とARGV[2]の比較結果（下記参照）を組み込み変数RESULTに代入する必要がある。

```
ARGV[1]  >  ARGV[2]  →  RESULT  >  0
ARGV[1]  =  ARGV[2]  →  RESULT  =  0
ARGV[1]  <  ARGV[2]  →  RESULT  <  0
```

```
$FUNCTION compare$
        $RESULT = ARGV[1] - ARGV[2]$
$END$
$a = { 4, 2, 1, 3 }$
$LSORT(a, ”compare”)$
```

結果：`{ 1, 2, 3, 4 }`

5.7.24. ISFUNCTION 関数

第一引数で指定した文字列が関数として定義されているかどうかを調べる。関数として定義されている場合には1を、そうでない場合は0を返す。

5.7.25. REGEX_REPLACE 関数

第1マクロ実引数で指定した文字列のうち、第2マクロ実引数で指定した正規表現にマッチする箇所を第3マクロ実引数の内容で置換した文字列を返す。

正規表現はECMAScript互換とする。

5.7.26. CLEAN 関数

第1マクロ実引数で指定した名前の配列を全削除する。

5.7.27. DIE 関数

現在の状態にかかわらず、マクロプロセッサを強制終了する。コンフィギュレータを終了するわけではないので、マクロプロセッサ終了後の処理は継続して行われる。

5.8 組込み変数

マクロプロセッサは、起動時にいくつかの組込み変数を自動的に設定する。組込み変数も、通常の変数と同様に扱うことができる。代入により、組込み変数に異なる属性を設定することも可能であるが、‘RESULT’をのぞき、そのような使い方は推奨しない。

5.8.1. SPC 変数

‘SPC’変数は、空白 1 文字からなる文字列属性を持つ。

5.8.2. TAB 変数

‘TAB’変数は、水平タブ 1 文字からなる文字列属性を持つ。

5.8.3. NL 変数

‘NL’変数は、改行 1 文字からなる文字列属性を持つ。

5.8.4. ARGC 変数

‘ARGC’変数は、ユーザー定義関数の内部において、渡された引数の個数（関数名を含む）を表す。

‘ARGC’変数をユーザー定義関数の外部で参照した場合の内容は不定である。また、ユーザー定義関数の内部から他のユーザー定義関数を呼び出した場合も‘ARGC’変数は破壊される。

5.8.5. ARGV 変数

‘ARGV’変数は、ユーザー定義関数の内部において、渡された引数を取得するための連想配列である。‘ARGV[0]’には関数名が、‘ARGV[0]’から‘ARGV[ARGC-1]’には呼び出し時に渡された引数が格納される。‘ARGV[ARGC]’は空となる。‘ARGV[ARGC+1]’以降の内容は不定であり、参照してはならない。

‘ARGV’変数をユーザー定義関数の外部で参照した場合の内容は不定である。また、ユーザー定義関数の内部から他のユーザー定義関数を呼び出した場合も‘ARGC’変数は破壊される。

5.8.6. RESULT 変数

‘RESULT’変数は、ユーザー定義関数の内部において、返却値を格納するための変数である。‘RESULT’変数に格納した内容は、ユーザー定義関数の呼び出し側で、関数の評価結果として取得することができる。

‘RESULT’変数をユーザー定義関数の外部で参照した場合の内容は不定である。また、ユーザー定義関数の外部で‘RESULT’変数に代入した場合の動作は未定義である。

5.8.7. CFG_PASS 変数

‘CFG_PASS’変数は、コンフィギュレータ起動時に `--path` オプションで指定したパス番号を格納する。

6. ユーザー定義関数

マクロプロセッサでは、ユーザー定義関数を定義し呼び出すことができる。定義したユーザー定義関数は、組込み関数と同様に扱うことができる。なお、ユーザー定義関数は前方参照ができない。したがって、ユーザー定義関数を定義するより前に、その関数を呼び出すことはできない。

6.1 ユーザー定義関数の定義

ユーザー定義関数の定義は、‘$FUNCTION$’命令を用いて行う。以下に‘$FUNCTION$’命令の構文を示す。

```
function-definition ::= function-directive any-content '$END$'
function-directive ::= '$FUNCTION' identifier '$'
any-content ::= macro-statement | plain-text
plain-text ::= [^'$']+
```

‘$FUNCTION$’命令は、それ自身と‘END’命令の間にブロックを形成する。*any-content* を省略することはできない。‘$FUNCTION$’命令では識別子を1つ指定する必要がある。この識別子が関数名となる。

ユーザー定義関数は、受け取った引数を参照するために２種類の組込み変数を使用する。ひとつは‘ARGC’変数であり、もうひとつは‘ARGV’連想配列である。‘ARGC’変数は、呼び出された関数の関数名を含む引数の個数である。‘ARGV’連想配列には実際の引数の内容が格納される。‘ARGV[0]’には関数名が、‘ARGV[0]’から‘ARGV[ARGC-1]’には呼び出し時に渡された引数が格納される。‘ARGV[ARGC]’は空となる。‘ARGV[ARGC+1]’以降の

内容は不定であり、参照してはならない。これらは、C 言語における main 関数の引数‘argc’および‘argv’と同様に考えてよい。なお、ユーザー定義関数を定義する段階では、引数の個数を規定することができない。

ユーザー定義関数から呼び出し元に値を返すには、‘RESULT’変数に返却値を代入する。これにより、呼び出し元ではユーザー定義関数呼び出しの評価結果として、返却値を取得することができる。‘RESULT’変数は関数から呼び出し元に戻る時点でクリアされる。なお、関数の途中で脱出する手段はない。

以下にユーザー定義関数の具体例を挙げる。

```
$FUNCTION increment$
        $RESULT = ARGV[1] + 1$
$END$

$a = 1$
$increment(a)$
```

結果：2

上記のユーザー定義関数‘increment’は、受け取った引数に１を加えた結果を返す単純なものである。

6.2 可変関数

可変関数は、関数名を表す文字列をもとに、ユーザー定義関数を呼び出すための仕組みである。C言語における関数へのポインタと同等に考えてよい。用途としては、‘LSORT’関数における比較関数のような、他の関数からのコールバックが主である。

可変関数を呼び出すのは、‘CALL’関数を使用する。

7. TOPPERS カーネル固有の仕様

コンフィギュレータが対応するカーネルに応じて、マクロプロセッサにいくつかの仕様を追加する。

7.1 μITRON 系カーネル固有の仕様

μITRON系カーネル（ASPカーネルなど）では、いくつかの組込み変数を追加する。

7.1.1. 静的 API のパラメータ変数

システムコンフィギュレーションファイルに記述された静的APIの各パラメータは、次の

形式の名前を持つ連想配列として定義される。これらの変数は、cfgの起動オプションとして--with-software-componentを指定した場合は設定されない。

オブジェクト種別.パラメータ名

オブジェクト種別は、静的APIの右側3文字で表される。例えば、タスクであれば‘CRE_TSK’の右側3文字である‘TSK’がオブジェクト種別となる。ただし、タスク例外に関しては、タスクに付随する機能であることから、‘TEX’ではなく‘TSK’とする必要がある。パラメータ名は、静的APIの各パラメータの名前を大文字にしたものである。

以下に、‘CRE_TSK’を例として具体的な解説を行う。

```
CRE_TSK(tskid, { tskatr, exinf, task, itskpri, stksz, stk });
```

上記の静的APIの場合、以下の名前を持つ連想配列が定義される。

```
TSK.TSKID
TSK.TSKATR
TSK.EXINF
TSK.TASK
TSK.ITSKPRI
TSK.STKSZ
TSK.STK
```

システムコンフィギュレーションファイルに、

```
CRE_TSK(TASK1, { TA_HLNG|TA_ACT, 0, task1, 5, 0x1000, NULL });
```

が記述された場合、（TASK1には1が割付けられるため）

```
$TSK.TSKID[1] = VALUE(”TASK1”, 1)$
$TSK.TSKATR[1] = VALUE(”TA_HLNG|TA_ACT”, 0x02)$
$TSK.EXINF[1] = ”0”$
$TSK.TASK[1] = ”task1”$
$TSK.ITSKPRI[1] = VALUE(”5”, 5)$
$TSK.STKSZ[1] = VALUE(”0x1000”, 0x1000)$
$TSK.STK[1] = ”NULL”$
```

が実行されたかのように、連想配列が設定される。ここで、連想配列の添え字である‘1’は、‘TASK1’に割付けられたID番号の値である。また、‘EXINF’、‘TASK’、および‘STK’は文字列属性しか設定されていないが、これはプログラム全体のリンクが解決されるまで値が特定できないためである。

静的APIテーブルにおいて、末尾に ... を付加して記述したパラメータの場合、個々のパラメータ名は、... の代わりに、0，1，2，... の連番が付加される。また、パラメータが1個以上ある場合には ... も連番も付加しない変数が、各パラメータを格納する順序付きリストとして定義される。ただし、この順序付きリストの各要素は文字列属性しか持たない。将来的に、値属性を格納するように仕様変更する可能性があるため、現時点ではこの順序付きリストの値属性を利用してはならない。

タスクやセマフォなど、ID番号によって記述された静的APIが特定できる場合には、連想配列の添え字にはID番号を使用する。‘DEF_INH’の割り込みハンドラ番号のように、静的APIの第1パラメータの値によって記述された静的APIが特定できる場合には、その値を連想番号の添え字として使用する。‘ATT_INI’のように、記述された静的APIを特定するための値がパラメータとして存在しない場合、同種の静的APIが記述された順に、1，2，3，... の連番を与え、それを連想配列の添え字として使用する。

システムコンフィギュレーションファイルにどのような種類の静的APIがどれだけ記述されたかを知るため、ID番号または連番を並べた順序付きリストを持つ組込み変数が定義される。それらの変数は連想配列ではなく単純変数であり、以下のような名前となる。

オブジェクト種別.ID_LIST
オブジェクト種別.ORDER_LIST
オブジェクト種別.RORDER_LIST

ここで、‘ID_LIST’はID番号または（割り込みハンドラ番号のような）それに準ずる値の並びであり、昇順に整列されている。‘ORDER_LIST’は同種の静的APIが記述された順に与えられた連番の並びである。‘ORDER_LIST’は‘ID_LIST’が定義されるオブジェクト種別でも定義される。その場合、ID番号（またはそれに準ずる値）を出現順に並べたものである。‘RORDER_LIST’は、‘ORDER_LIST’を逆順に並べたものである。

また、ID番号（またはそれに準する値）から連番を求められるように、

オブジェクト種別.ORDER

という名前の連想配列も定義される。この連想配列の添え字にはID番号（またはそれに準する値）を使用する。

さらに、静的APIが記述されたシステムコンフィギュレーションファイル名、および行番号を保持する

オブジェクト種別.TEXT_LINE

という名前の連想配列が定義される。この連想配列の添え字にはID番号（またはそれに準する値）を使用する。 連想配列の各要素は、文字列属性としてファイル名を、値属性として行番号を保持する。

オブジェクト種別の代わりに、静的API名に関連付けられた行番号の連想配列も定義される。

静的API名.TEXT_LINE

例えば、タスク例外を定義するためのDEF_TEXに指定したパラメータにエラーがあった場合、DEF_TEX.TEXT_LINE[tskid]のようにすることで静的API、DEF_TEXが記述された行番号を取得することができる。

7.1.2. クラスおよびドメイン

TOPPERS新世代カーネルに対応するため、システムコンフィギュレーションファイルに記述したクラスおよびドメインに関する情報が連想配列として定義される。

オブジェクト種別.CLASS
オブジェクト種別.DOMAIN

これらの連想配列は、カーネルオブジェクトの静的API（例えば‘CRE_TSK’）が属しているクラスまたはドメインの名称を保持する。値は保持しない。これらの連想配列は、静的APIの他のパラメータと同様に扱うことができる。

クラスおよびドメインの囲みに関する情報が下記の連想配列として定義される。

CLASS
DOMAIN

これらの連想配列はクラスまたはドメインの囲みの出現順を添え字とする。各要素は、クラスまたはドメインの名称を保持し、値は保持しない。なお、同名のクラスまたはドメインであっても、これらの連想配列では異なる要素として扱うものとする。

クラスまたはドメインの出現順序は、

CLASS.ORDER_LIST
CLASS.RORDER_LIST
DOMAIN.ORDER_LIST
DOMAIN.RORDER_LIST

という順序付きリストで定義される。なお、‘RORDER_LIST’は、‘ORDER_LIST’を逆順に並べたものである。また、以下の連想配列

CLASS.TEXT_LINE
DOMAIN.TEXT_LINE

は、クラスまたはドメインの囲みの先頭行番号（文字列属性としてファイル名を、値属性として行番号）を保持する。連想配列の添え字にはクラスまたはドメインの囲みの出現順を指定する。

7.1.3. 標準割り込みモデルのための組込み変数

標準割り込みモデルに対応したターゲット依存部のテンプレートファイル（マクロプロセッサへの入力ファイル）で使用するために、以下の組込み変数が定義される。

```
INTNO_ATTISR_REG
INTNO_CFGINT_REG
INHNO_DEFINH_REG
EXCNO_DEFEXC_REG
INTATR_CFGINT
INTPRI_CFGINT
INHATR_DEFINH
INTHDR_DEFINH
EXCATR_DEFEXC
EXCHDR_DEFEXC
```

これらの組込み変数は、おおむね下記のルールにしたがって命名されている。

パラメータ名_静的API名[_REG]

ただし、パラメータ名はすべて大文字であり、静的API名は下線（アンダースコア）を省

略している。これらの組込み変数は、その名前が示すパラメータの順序付きリストである。

名前の末尾に‘_REG’を伴う組込み変数は、名前の末尾に‘_VALID’を伴う変数名と区別するためのものである。‘_VALID’付きの変数は、その名前が示すパラメータの値として使用することができる有効な値の順序付きリストである。‘_VALID’付きの変数は組込み変数ではなく、カーネルのターゲット依存部が提供するテンプレートファイル内で定義すべきものである。それに対して、‘_REG’付きの組込み変数は、システムコンフィギュレーションファイルに実際に記述された静的APIの、その名前が示すパラメータの順序付きリストである。

7.1.4. テンプレートファイルで定義すべき変数

割り込み番号など、静的APIのパラメータとして指定することができる有効な値はターゲットに依存している。それらの有効な値を示す順序付きリストを格納した変数を、カーネルの各ターゲット依存部のテンプレートファイルで定義する必要がある。ターゲット依存部のテンプレートファイルで定義すべき変数を以下に示す。

```
INTNO_ATTISR_VALID
INTNO_DISINT_VALID
INTNO_CFGINT_VALID
INHNO_ATTISR_VALID
INHNO_DEFINH_VALID
EXCNO_DEFEXC_VALID
INTPRI_CHGIPM_VALID
INTPRI_CFGINT_VALID
```

いずれも、パラメータとして有効な値を並べた順序付きリストである。命名のルールに関しては 6.1.3 で解説している。このうち、静的 API が対象ではないもの、例えば‘INTNO_DISINT_VALID’などは、マクロプロセッサないしはコンフィギュレータでは直接扱わない。

7.1.5. コンフィギュレータのバージョンを表す組込み変数

コンフィギュレータのバージョン番号を表す組み込み変数として CFG_VERSION を定義する。文字列属性としてバージョン番号を、値属性としてコンフィギュレータをビルドした時刻を、年月日時分秒を表す値として保持する。

7.1.6. 静的 API 列挙変数

システムコンフィギュレーションファイルに出現した静的 API を列挙するために、次の連

想配列が設定される。

API.TEXT_LINE
API.NAME
API.TYPE
API.PARAMS
API.ARGS

API.TEXT_LINE には、その静的 API が記述されたファイル名および行番号を保持する。API.NAME には、その静的 API の名称（'CRE_TSK'など）を保持する。API.TYPE には、その静的 API の種別（'TSK'など）を保持する。API.PARAMS には、7.1.1.で示すオブジェクト種別.パラメータ名（'TSK.TSKID'など）がリスト形式で保持される。API.ARGS には、静的 API に実際に渡された実引数がリスト形式で保持される。

これらの連想配列の添え字には、システムコンフィギュレーションファイルにおける静的 API の出現順序が 1 から順に割付けられる。

7.1.7. ソフトウェア部品対応用組込み関数

以下の組込み関数は、cfg の起動オプションとして--with-software-component を指定した場合だけ使用することができる。

7.1.7.1. ASSIGNID 関数

第一引数で指定した種別のオブジェクトに対してID番号を割付け、個々のパラメータを表す変数、TEXT_LINE、ID_LIST、ORDER_LIST、およびRORDER_LISTを設定する。ただし、ID番号を持たないオブジェクトの場合、ID_LISTは設定しない。

以下に、カーネル本体におけるASSIGNIDの使用例を示す。

```
$type_list = { "tsk", "sem", "sem", "flg", "dtq", "pdq", "mbx", "mpf",
"cyc", "alm", "int", "isr", "inh", "ics", "exc", "ini", "ter" }$
$FOREACH type type_list$
        $ASSIGNID(type)$
$END$
```

7.1.7.2. ADDAPI 関数

API.で始まる連想配列群を、必要に応じて適切に変換を行った上で、別のプレフィックスで始まる連想配列群に追加する。想定する使用方法としては、特定のソフトウェア部品で

理解可能な静的APIに関しては、追加を行わないか、他の静的APIに置き換えて追加を行う。理解できない静的 API に関してはそのまま追加を行う。

この関数の第一引数には API.で始まる連想配列の連番を、第二引数には追加先に連想配列群のプレフィックスを、第三引数には対象とする静的 API 名を、第四引数には追加するパラメータシンボルの並びを、第五引数にはパラメータの並びを指定する。

以下に、使用例を示す。

```
$FOREACH i API.ORDER_LIST$
  $IF EQ(API.TYPE[i], “YYY”)$
    $PARAMS = {“SEM.SEMID”, “SEM.SEMATR”,
                        “SEM.ISEMCNT”, “SEM.MAXSEM” }$
    $ARGS = { “YYYSEM1”, VALUE(“TA_NULL”, 0), 1, 1 }$
    $ADDAPI(i, “TEMP_API”, “CRE_SEM”, PARAM, ARGS)$
  $ELSE$
    $ADDAPI(i, “TEMP_API”, API.NAME[i], API.PARAMS[i], API.ARGS[i])$
  $END$
$END$
```

上記の例では、あるソフトウェア部品の静的 API である XXX_YYY を CRE_SEM に置き換えている。その際、いったん TEMP_API.で始まる連想配列として XXX_YYY 処理後の静的 API を保存している。

7.1.7.3. SWAPPREFIX 関数

第一引数および第二引数で指定したプレフィックスを持つ変数群を入れ替える。

‘ADDAPI’関数で、別のプレフィックスを持つ連想配列を組み立てたあとは、この関数を用いることで、API.をプレフィックスに持つ連想配列と交換することができる。

7.1.6.2.の例のあとで、下記のように記述すれば、TEMP_API.をプレフィックスとする変数群と API.をプレフィックスとする変数群が入れ替わる。

```
$SWAPPREFIX("TEMP_API", "API")$
```

‘SWAPPREFIX’を呼出す前に、TEMP_API.TEXT_LINE であったものは API.TEXT_LINE に、API.TEXT_LINE であったものは TEMP_API.TEXT_LINE に置き換わる。他の変数についても同様に振舞う。

7.1.7.4. CLEANVARS 関数

第一引数で指定したプレフィックスで始まる変数群を削除する。'SWAPPREFIX'関数で交換したあと、不要になった変数群はこの関数で削除しておくことが望ましい。