

gasp

COLLABORATORS

	<i>TITLE :</i> gasp		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		June 8, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	gasp	1
1.1	gasp.guide	1
1.2	gasp.guide/Overview	1
1.3	gasp.guide/Invoking GASP	2
1.4	gasp.guide/Commands	4
1.5	gasp.guide/Conditionals	4
1.6	gasp.guide/Loops	6
1.7	gasp.guide/Variables	6
1.8	gasp.guide/Macros	7
1.9	gasp.guide/Data	9
1.10	gasp.guide/Initialized	10
1.11	gasp.guide/Uninitialized	11
1.12	gasp.guide/Listings	11
1.13	gasp.guide/Other Commands	12
1.14	gasp.guide/Syntax Details	13
1.15	gasp.guide/Markers	14
1.16	gasp.guide/Constants	15
1.17	gasp.guide/Symbols	15
1.18	gasp.guide/Expressions	15
1.19	gasp.guide/String Builtins	16
1.20	gasp.guide/Alternate	16
1.21	gasp.guide/Index	17

Chapter 1

gasp

1.1 gasp.guide

GASP

GASP is a preprocessor for assembly programs.

This file describes version 1 of GASP.

Steve Chamberlain wrote GASP; Roland Pesch wrote this manual.

Overview	What is GASP?
Invoking GASP	Command line options.
Commands	Preprocessor commands.
Index	Index.

1.2 gasp.guide/Overview

What is GASP?

The primary purpose of the GNU assembler is to assemble the output of other programs--notably compilers. When you have to hand-code specialized routines in assembly, that means the GNU assembler is an unfriendly processor: it has no directives for macros, conditionals, or many other conveniences that you might expect.

In some cases you can simply use the C preprocessor, or a generalized preprocessor like M4; but this can be awkward, since none of these things are designed with assembly in mind.

GASP fills this need. It is expressly designed to provide the facilities you need with hand-coded assembly code. Implementing it as a preprocessor, rather than part of the assembler, allows the maximum flexibility: you can use it with hand-coded assembly, without paying a

penalty of added complexity in the assembler you use for compiler output.

Here is a small example to give the flavor of GASP. This input to GASP

```

        .MACRO  saveregs from=8 to=14
count   .ASSIGNA \from
        ! save r\from..r\to
        .AWHILE \&count LE \to
        mov     r\&count,@-sp
count   .ASSIGNA \&count + 1
        .AENDW
        .ENDM

        saveregs from=12

bar:    mov     #H'dead+10,r0
foo     .SDATAC "hello"<10>
        .END

```

generates this assembly program:

```

        ! save r12..r14
        mov     r12,@-sp
        mov     r13,@-sp
        mov     r14,@-sp

bar:    mov     #57005+10,r0
foo:    .byte   6,104,101,108,108,111,10

```

1.3 gasp.guide/Invoking GASP

Command Line Options

The simplest way to use GASP is to run it as a filter and assemble its output. In Unix and its ilk, you can do this, for example:

```
$ gasp prog.asm | as -o prog.o
```

Naturally, there are also a few command-line options to allow you to request variations on this basic theme. Here is the full set of possibilities for the GASP command line.

```

gasp [ -a | --alternate ]
     [ -c CHAR | --commentchar CHAR ]
     [ -d | --debug ] [ -h | --help ]
     [ -o OUTFILE | --output OUTFILE ]
     [ -p | --print ] [ -s | --copysource ]
     [ -u | --unreasonable ] [ -v | --version ]
     INFILE ...

```

`INFILE ...'

The input file names. You must specify at least one input file; if you specify more, GASP preprocesses them all, concatenating the output in the order you list the INFILE arguments.

Mark the end of each input file with the preprocessor command ``.END'``. See Miscellaneous commands.

``-a'`

``--alternate'`

Use alternative macro syntax. See Alternate macro syntax, for a discussion of how this syntax differs from the default GASP syntax.

``-c 'CHAR''``

``--commentchar 'CHAR''``

Use CHAR as the comment character. The default comment character is `'!'``. For example, to use a semicolon as the comment character, specify ``-c ';'`` on the GASP command line. Since assembler command characters often have special significance to command shells, it is a good idea to quote or escape CHAR when you specify a comment character.

For the sake of simplicity, all examples in this manual use the default comment character `'!'``.

``-d'`

``--debug'`

Show debugging statistics. In this version of GASP, this option produces statistics about the string buffers that GASP allocates internally. For each defined buffersize S, GASP shows the number of strings N that it allocated, with a line like this:

```
strings size S : N
```

GASP displays these statistics on the standard error stream, when done preprocessing.

``-h'`

``--help'`

Display a summary of the GASP command line options.

``-o OUTFILE'`

``--output OUTFILE'`

Write the output in a file called OUTFILE. If you do not use the ``-o'` option, GASP writes its output on the standard output stream.

``-p'`

``--print'`

Print line numbers. GASP obeys this option *only* if you also specify ``-s'` to copy source lines to its output. With ``-s -p'`, GASP displays the line number of each source line copied (immediately after the comment character at the beginning of the line).

``-s'`

``--copysource'`

Copy the source lines to the output file. Use this option to see the effect of each preprocessor line on the GASP output. GASP

places a comment character ('!' by default) at the beginning of each source line it copies, so that you can use this option and still assemble the result.

'-u'

'--unreasonable'

Bypass "unreasonable expansion" limit. Since you can define GASP macros inside other macro definitions, the preprocessor normally includes a sanity check. If your program requires more than 1,000 nested expansions, GASP normally exits with an error message. Use this option to turn off this check, allowing unlimited nested expansions.

'-v'

'--version'

Display the GASP version number.

1.4 gasp.guide/Commands

Preprocessor Commands

GASP commands have a straightforward syntax that fits in well with assembly conventions. In general, a command extends for a line, and may have up to three fields: an optional label, the command itself, and optional arguments to the command. You can write commands in upper or lower case, though this manual shows them in upper case. See Details of the GASP syntax, for more information.

Conditionals

Loops

Variables

Macros

Data

Listings

Other Commands

Syntax Details

Alternate

1.5 gasp.guide/Conditionals

Conditional assembly

=====

The conditional-assembly directives allow you to include or exclude portions of an assembly depending on how a pair of expressions, or a pair of strings, compare.

The overall structure of conditionals is familiar from many other

contexts. ``.AIF`` marks the start of a conditional, and precedes assembly for the case when the condition is true. An optional ``.AELSE`` precedes assembly for the converse case, and an ``.AENDI`` marks the end of the condition.

You may nest conditionals up to a depth of 100; GASP rejects nesting beyond that, because it may indicate a bug in your macro structure.

Conditionals are primarily useful inside macro definitions, where you often need different effects depending on argument values. See *Defining your own directives*, for details about defining macros.

``.AIF EXPRA CMP EXPRB``

``.AIF "STRA" CMP "STRB"```

The governing condition goes on the same line as the ``.AIF`` preprocessor command. You may compare either two strings, or two expressions.

When you compare strings, only two conditional CMP comparison operators are available: ``.EQ`` (true if STRA and STRB are identical), and ``.NE`` (the converse).

When you compare two expressions, *both expressions must be absolute* (see *Arithmetic expressions in GASP*). You can use these CMP comparison operators with expressions:

``.EQ``

Are EXPRA and EXPRB equal? (For strings, are STRA and STRB identical?)

``.NE``

Are EXPRA and EXPRB different? (For strings, are STRA and STRB different?)

``.LT``

Is EXPRA less than EXPRB? (Not allowed for strings.)

``.LE``

Is EXPRA less than or equal to EXPRB? (Not allowed for strings.)

``.GT``

Is EXPRA greater than EXPRB? (Not allowed for strings.)

``.GE``

Is EXPRA greater than or equal to EXPRB? (Not allowed for strings.)

``.AELSE``

Marks the start of assembly code to be included if the condition fails. Optional, and only allowed within a conditional (between ``.AIF`` and ``.AENDI``).

``.AENDI``

Marks the end of a conditional assembly.

1.6 gasp.guide/Loops

Repetitive sections of assembly

=====

Two preprocessor directives allow you to repeatedly issue copies of the same block of assembly code.

```
\.AREPEAT AEXP'
\.AENDR'
```

If you simply need to repeat the same block of assembly over and over a fixed number of times, sandwich one instance of the repeated block between `\.AREPEAT` and `\.AENDR`. Specify the number of copies as `AEXP` (which must be an absolute expression). For example, this repeats two assembly statements three times in succession:

```
.AREPEAT      3
rotcl   r2
divl    r0,r1
.AENDR
```

```
\.AWHILE EXPRA CMP EXPRB'
\.AENDW'
\.AWHILE STRA CMP STRB'
\.AENDW'
```

To repeat a block of assembly depending on a conditional test, rather than repeating it for a specific number of times, use `\.AWHILE`. `\.AENDW` marks the end of the repeated block. The conditional comparison works exactly the same way as for `\.AIF`, with the same comparison operators (see Conditional assembly).

Since the terms of the comparison must be absolute expression, `\.AWHILE` is primarily useful within macros. See Defining your own directives.

You can use the `\.EXITM` preprocessor directive to break out of loops early (as well as to break out of macros). See Defining your own directives.

1.7 gasp.guide/Variables

Preprocessor variables

=====

You can use variables in GASP to represent strings, registers, or the results of expressions.

You must distinguish two kinds of variables:

1. Variables defined with `\.EQU` or `\.ASSIGN`. To evaluate this kind of variable in your assembly output, simply mention its name. For example, these two lines define and use a variable `'eg'`:

```

eg      .EQU   FLIP-64
      ...
      mov.l  eg,r0

```

Do not use this kind of variable in conditional expressions or while loops; GASP only evaluates these variables when writing assembly output.

2. Variables for use during preprocessing. You can define these with ``.ASSIGNC'` or ``.ASSIGNA'`. To evaluate this kind of variable, write ``\&'` before the variable name; for example,

```

opcit  .ASSIGNA  47
      ...
      .AWHILE  \&opcit GT 0
      ...
      .AENDW

```

GASP treats macro arguments almost the same way, but to evaluate them you use the prefix ``\'` rather than ``\&'`. See Defining your own directives.

``PVAR .EQU EXPR'`

Assign preprocessor variable PVAR the value of the expression EXPR. There are no restrictions on redefinition; use ``.EQU'` with the same PVAR as often as you find it convenient.

``PVAR .ASSIGN EXPR'`

Almost the same as ``.EQU'`, save that you may not redefine PVAR using ``.ASSIGN'` once it has a value.

``PVAR .ASSIGNA AEXPR'`

Define a variable with a numeric value, for use during preprocessing. AEXPR must be an absolute expression. You can redefine variables with ``.ASSIGNA'` at any time.

``PVAR .ASSIGNC "STR"'`

Define a variable with a string value, for use during preprocessing. You can redefine variables with ``.ASSIGNC'` at any time.

``PVAR .REG (REGISTER)'`

Use ``.REG'` to define a variable that represents a register. In particular, REGISTER is *not evaluated* as an expression. You may use ``.REG'` at will to redefine register variables.

All these directives accept the variable name in the "label" position, that is at the left margin. You may specify a colon after the variable name if you wish; the first example above could have started ``eg:'` with the same effect.

1.8 gasp.guide/Macros

Defining your own directives

=====

The commands ``.MACRO`` and ``.ENDM`` allow you to define macros that generate assembly output. You can use these macros with a syntax similar to built-in GASP or assembler directives. For example, this definition specifies a macro ``SUM`` that adds together a range of consecutive registers:

```

        .MACRO  SUM FROM=0, TO=9
        ! \FROM \TO
COUNT  mov     r\FROM,r10
        .ASSIGNA \FROM+1
        .AWHILE \&COUNT LE \TO
COUNT  add     r\FROM,r10
        .ASSIGNA \&COUNT+1
        .AENDW
        .ENDM

```

With that definition, ``SUM 0,5`` generates this assembly output:

```

! 0 5
mov     r0,r10
add     r1,r10
add     r2,r10
add     r3,r10
add     r4,r10
add     r5,r10

```

``.MACRO MACNAME``

``.MACRO MACNAME MACARGS ...``

Begin the definition of a macro called `MACNAME`. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with ``=DEFLT``. For example, these are all valid ``.MACRO`` statements:

``.MACRO COMM``

Begin the definition of a macro called ``.COMM``, which takes no arguments.

``.MACRO PLUS1 P, P1``

``.MACRO PLUS1 P P1``

Either statement begins the definition of a macro called ``.PLUS1``, which takes two arguments; within the macro definition, write ``.P`` or ``.P1`` to evaluate the arguments.

``.MACRO RESERVE_STR P1=0 P2``

Begin the definition of a macro called ``.RESERVE_STR``, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as ``.RESERVE_STR A,B`` (with ``.P1`` evaluating to A and ``.P2`` evaluating to B), or as ``.RESERVE_STR ,B`` (with ``.P1`` evaluating as the default, in this case ``.0``, and ``.P2`` evaluating to B).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, 'SUM 9,17' is equivalent to 'SUM TO=17, FROM=9'. Macro arguments are preprocessor variables similar to the variables you define with '.ASSIGNA' or '.ASSIGNC'; in particular, you can use them in conditionals or for loop control. (The only difference is the prefix you write to evaluate the variable: for a macro argument, write '\ARGNAME', but for a preprocessor variable, write '\&VARIABLE'.)

'NAME .MACRO'

'NAME .MACRO (MACARGS ...)'

An alternative form of introducing a macro definition: specify the macro name in the label position, and the arguments (if any) between parentheses after the name. Defaulting rules and usage work the same way as for the other macro definition syntax.

'.ENDM'

Mark the end of a macro definition.

'.EXITM'

Exit early from the current macro definition, '.AREPEAT' loop, or '.AWHILE' loop.

'@'

GASP maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with '@', but *only within a macro definition*.

'LOCAL NAME [, ...]'

Warning: 'LOCAL' is only available if you select "alternate macro syntax" with '-a' or '--alternate'. See Alternate macro syntax.

Generate a string replacement for each of the NAME arguments, and replace any instances of NAME in each macro expansion. The replacement string is unique in the assembly, and different for each separate macro expansion. 'LOCAL' allows you to write macros that define symbols, without fear of conflict between separate macro expansions.

1.9 gasp.guide/Data

Data output

=====

In assembly code, you often need to specify working areas of memory; depending on the application, you may want to initialize such memory or not. GASP provides preprocessor directives to help you avoid repetitive coding for both purposes.

You can use labels as usual to mark the data areas.

Initialized

Uninitialized

1.10 `gasp.guide/Initialized`

Initialized data

These are the GASP directives for initialized data, and the standard GNU assembler directives they expand to:

```
\.DATA  EXPR,  EXPR,  ...'
\.DATA.B EXPR,  EXPR,  ...'
\.DATA.W EXPR,  EXPR,  ...'
\.DATA.L EXPR,  EXPR,  ...'
```

Evaluate arithmetic expressions `EXPR`, and emit the corresponding `'as'` directive (labelled with `LAB`). The unqualified `'\.DATA'` emits `'\.long'`; `'\.DATA.B'` emits `'\.byte'`; `'\.DATA.W'` emits `'\.short'`; and `'\.DATA.L'` emits `'\.long'`.

For example, `'foo .DATA 1,2,3'` emits `'foo: .long 1,2,3'`.

```
\.DATAB REPEAT,  EXPR'
\.DATAB.B REPEAT,  EXPR'
\.DATAB.W REPEAT,  EXPR'
\.DATAB.L REPEAT,  EXPR'
```

Make `'as'` emit `REPEAT` copies of the value of the expression `EXPR` (using the `'as'` directive `'\.fill'`). `'\.DATAB.B'` repeats one-byte values; `'\.DATAB.W'` repeats two-byte values; and `'\.DATAB.L'` repeats four-byte values. `'\.DATAB'` without a suffix repeats four-byte values, just like `'\.DATAB.L'`.

`REPEAT` must be an absolute expression with a positive value.

```
\.SDATA "STR" ...'
```

String data. Emits a concatenation of bytes, precisely as you specify them (in particular, **nothing is added to mark the end** of the string). See *String and numeric constants*, for details about how to write strings. `'\.SDATA'` concatenates multiple arguments, making it easy to switch between string representations. You can use commas to separate the individual arguments for clarity, if you choose.

```
\.SDATAB REPEAT, "STR" ...'
```

Repeated string data. The first argument specifies how many copies of the string to emit; the remaining arguments specify the string, in the same way as the arguments to `'\.SDATA'`.

```
\.SDATAZ "STR" ...'
```

Zero-terminated string data. Just like `'\.SDATA'`, except that `'\.SDATAZ'` writes a zero byte at the end of the string.

```
\.SDATAC "STR" ...'
```

Count-prefixed string data. Just like `'\.SDATA'`, except that GASP precedes the string with a leading one-byte count. For example, `'\.SDATAC "HI"'` generates `'\.byte 2,72,73'`. Since the count field

is only one byte, you can only use ``.SDATAC'` for strings less than 256 bytes in length.

1.11 gasp.guide/Uninitialized

Uninitialized data

Use the ``.RES'`, ``.SRES'`, ``.SRESC'`, and ``.SRESZ'` directives to reserve memory and leave it uninitialized. GASP resolves these directives to appropriate calls of the GNU ``as'` ``.space'` directive.

```
`.RES COUNT'
`.RES.B COUNT'
`.RES.W COUNT'
`.RES.L COUNT'
```

Reserve room for COUNT uninitialized elements of data. The suffix specifies the size of each element: ``.RES.B'` reserves COUNT bytes, ``.RES.W'` reserves COUNT pairs of bytes, and ``.RES.L'` reserves COUNT quartets. ``.RES'` without a suffix is equivalent to ``.RES.L'`.

```
`.SRES COUNT'
`.SRES.B COUNT'
`.SRES.W COUNT'
`.SRES.L COUNT'
```

``.SRES'` is a synonym for ``.RES'`.

```
`.SRESC COUNT'
`.SRESC.B COUNT'
`.SRESC.W COUNT'
`.SRESC.L COUNT'
```

Like ``.SRES'`, but reserves space for ``.COUNT+1'` elements.

```
`.SRESZ COUNT'
`.SRESZ.B COUNT'
`.SRESZ.W COUNT'
`.SRESZ.L COUNT'
```

Like ``.SRES'`, but reserves space for ``.COUNT+1'` elements.

1.12 gasp.guide/Listings

Assembly listing control

=====

The GASP listing-control directives map straightforwardly to related GNU ``as'` directives.

```
`.PRINT LIST'
`.PRINT NOLIST'
```

Print control. This directive emits the GNU ``as'` directive

`\.list'` or `\.nolist'`, according to its argument. See `\.list'`, for details on how these directives interact.

`\.FORM LIN=LN'`

`\.FORM COL=COLS'`

`\.FORM LIN=LN COL=COLS'`

Specify the page size for assembly listings: LN represents the number of lines, and COLS the number of columns. You may specify either page dimension independently, or both together. If you do not specify the number of lines, GASP assumes 60 lines; if you do not specify the number of columns, GASP assumes 132 columns. (Any values you may have specified in previous instances of `\.FORM'` do *not* carry over as defaults.) Emits the `\.psize'` assembler directive.

`\.HEADING STRING'`

Specify STRING as the title of your assembly listings. Emits `\.title "STRING"'`.

`\.PAGE'`

Force a new page in assembly listings. Emits `\.eject'`.

1.13 `gasp.guide/Other Commands`

Miscellaneous commands

=====

`\.ALTERNATE'`

Use the alternate macro syntax henceforth in the assembly. See Alternate macro syntax.

`\.ORG'`

This command is recognized, but not yet implemented. GASP generates an error message for programs that use `\.ORG'`.

`\.RADIX S'`

GASP understands numbers in any of base two, eight, ten, or sixteen. You can encode the base explicitly in any numeric constant (see String and numeric constants). If you write numbers without an explicit indication of the base, the most recent `\.RADIX S'` command determines how they are interpreted. S is a single letter, one of the following:

`\.RADIX B'`

Base 2.

`\.RADIX Q'`

Base 8.

`\.RADIX D'`

Base 10. This is the original default radix.

`\.RADIX H'`

Base 16.

You may specify the argument *S* in lower case (any of 'bqdh') with the same effects.

``.EXPORT NAME'`

``.GLOBAL NAME'`

Declare *NAME* global (emits ``.global NAME'`). The two directives are synonymous.

``.PROGRAM'`

No effect: GASP accepts this directive, and silently ignores it.

``.END'`

Mark end of each preprocessor file. GASP issues a warning if it reaches end of file without seeing this command.

``.INCLUDE "STR"'`

Preprocess the file named by *STR*, as if its contents appeared where the ``.INCLUDE'` directive does. GASP imposes a maximum limit of 30 stacked include files, as a sanity check.

``.ALIGN SIZE'`

Evaluate the absolute expression *SIZE*, and emit the assembly instruction ``.align SIZE'` using the result.

1.14 `gasp.guide/Syntax Details`

Details of the GASP syntax

=====

Since GASP is meant to work with assembly code, its statement syntax has no surprises for the assembly programmer.

Whitespace (blanks or tabs; **not** newline) is partially significant, in that it delimits up to three fields in a line. The amount of whitespace does not matter; you may line up fields in separate lines if you wish, but GASP does not require that.

The **first field**, an optional "label", must be flush left in a line (with no leading whitespace) if it appears at all. You may use a colon after the label if you wish; GASP neither requires the colon nor objects to it (but will not include it as part of the label name).

The **second field**, which must appear after some whitespace, contains a GASP or assembly "directive".

Any **further fields** on a line are "arguments" to the directive; you can separate them from one another using either commas or whitespace.

Markers

Constants

Symbols

Expressions

String Builtins

1.15 gasp.guide/Markers

Special syntactic markers

GASP recognizes a few special markers: to delimit comments, to continue a statement on the next line, to separate symbols from other characters, and to copy text to the output literally. (One other special marker, '@', works only within macro definitions; see Defining your own directives.)

The trailing part of any GASP source line may be a "comment". A comment begins with the first unquoted comment character ('!' by default), or an escaped or doubled comment character ('\!' or '!!' by default), and extends to the end of a line. You can specify what comment character to use with the '-c' option (see Command Line Options). The two kinds of comment markers lead to slightly different treatment:

'!'

A single, un-escaped comment character generates an assembly comment in the GASP output. GASP evaluates any preprocessor variables (macro arguments, or variables defined with '.ASSIGNA' or '.ASSIGNC') present. For example, a macro that begins like this

```
.MACRO SUM FROM=0, TO=9
! \FROM \TO
```

issues as the first line of output a comment that records the values you used to call the macro.

'\!'

'!!'

Either an escaped comment character, or a double comment character, marks a GASP source comment. GASP does not copy such comments to the assembly output.

To *continue a statement* on the next line of the file, begin the second line with the character '+'.

Occasionally you may want to prevent GASP from preprocessing some particular bit of text. To *copy literally* from the GASP source to its output, place '\(' before the string to copy, and ')' at the end. For example, write '\(\!)' if you need the characters '\!' in your assembly output.

To *separate a preprocessor variable* from text to appear immediately after its value, write a single quote (''). For example, '.SDATA "\P'1"' writes a string built by concatenating the value of 'P' and the digit '1'. (You cannot achieve this by writing just '\P1', since 'P1' is itself a valid name for a preprocessor variable.)

1.16 gasp.guide/Constants

String and numeric constants

There are two ways of writing "string constants" in GASP: as literal text, and by numeric byte value. Specify a string literal between double quotes ("STR"). Specify an individual numeric byte value as an absolute expression between angle brackets (<EXPR>'. Directives that output strings allow you to specify any number of either kind of value, in whatever order is convenient, and concatenate the result. (Alternate syntax mode introduces a number of alternative string notations; see Alternate macro syntax.)

You can write "numeric constants" either in a specific base, or in whatever base is currently selected (either 10, or selected by the most recent '.RADIX').

To write a number in a *specific base*, use the pattern 'S'DDD': a base specifier character S, followed by a single quote followed by digits DDD. The base specifier character matches those you can specify with '.RADIX': 'B' for base 2, 'Q' for base 8, 'D' for base 10, and 'H' for base 16. (You can write this character in lower case if you prefer.)

1.17 gasp.guide/Symbols

Symbols

GASP recognizes symbol names that start with any alphabetic character, '_', or '\$', and continue with any of the same characters or with digits. Label names follow the same rules.

1.18 gasp.guide/Expressions

Arithmetic expressions in GASP

There are two kinds of expressions, depending on their result: "absolute" expressions, which resolve to a constant (that is, they do not involve any values unknown to GASP), and "relocatable" expressions, which must reduce to the form

ADDSYM+CONST-SUBSYM

where ADDSYM and SUBSYM are assembly symbols of unknown value, and CONST is a constant.

Arithmetic for GASP expressions follows very similar rules to C. You can use parentheses to change precedence; otherwise, arithmetic primitives have decreasing precedence in the order of the following list.

1. Single-argument '+' (identity), '-' (arithmetic opposite), or '~' (bitwise negation). *The argument must be an absolute expression.*
2. '*' (multiplication) and '/' (division). *Both arguments must be absolute expressions.*
3. '+' (addition) and '-' (subtraction). *At least one argument must be absolute.*
4. '&' (bitwise and). *Both arguments must be absolute.*
5. '|' (bitwise or) and '^' (bitwise exclusive or; '^' in C). *Both arguments must be absolute.*

1.19 gasp.guide/String Builtins

String primitives

You can use these primitives to manipulate strings (in the argument field of GASP statements):

``.LEN("STR")'`
Calculate the length of string `"STR"`, as an absolute expression. For example, ``.RES.B .LEN("sample")'` reserves six bytes of memory.

``.INSTR("STRING", "SEG", IX)'`
Search for the first occurrence of SEG after position IX of STRING. For example, ``.INSTR("ABCDEFGH", "CDE", 0)'` evaluates to the absolute result `'2'`.

The result is `'-1'` if SEG does not occur in STRING after position IX.

``.SUBSTR("STRING", START, LEN)'`
The substring of STRING beginning at byte number START and extending for LEN bytes.

1.20 gasp.guide/Alternate

Alternate macro syntax
=====

If you specify `'-a'` or `'--alternate'` on the GASP command line, the preprocessor uses somewhat different syntax. This syntax is reminiscent of the syntax of Phar Lap macro assembler, but it is **not** meant to be a full emulation of Phar Lap or similar assemblers. In particular, GASP does not support directives such as `'DB'` and `'IRP'`, even in alternate syntax mode.

In particular, `'-a'` (or `'--alternate'`) elicits these differences:

Preprocessor directives

You can use GASP preprocessor directives without a leading `'.'` dot. For example, you can write `'SDATA'` with the same effect as `'SDATA.'`

LOCAL

One additional directive, `'LOCAL'`, is available. See *Defining your own directives*, for an explanation of how to use `'LOCAL'`.

String delimiters

You can write strings delimited in these other ways besides `"STRING"`:

`'STRING'`

You can delimit strings with single-quote characters.

`<STRING>`

You can delimit strings with matching angle brackets.

single-character string escape

To include any single character literally in a string (even if the character would otherwise have some special meaning), you can prefix the character with `'!'` (an exclamation mark). For example, you can write `<4.3 !> 5.4!!>` to get the literal text `'4.3 > 5.4!'`.

Expression results as strings

You can write `'%EXPR'` to evaluate the expression `EXPR` and use the result as a string.

1.21 `gasp.guide/Index`

Index

+	Markers
<code>--alternate</code>	Invoking GASP
<code>--commentchar 'CHAR'</code>	Invoking GASP
<code>--copysource</code>	Invoking GASP
<code>--debug</code>	Invoking GASP
<code>--help</code>	Invoking GASP

--output OUTFILE	Invoking GASP
--print	Invoking GASP
--unreasonable	Invoking GASP
--version	Invoking GASP
-a	Invoking GASP
-c 'CHAR'	Invoking GASP
-d	Invoking GASP
-h	Invoking GASP
-o OUTFILE	Invoking GASP
-p	Invoking GASP
-s	Invoking GASP
-u	Invoking GASP
-v	Invoking GASP
.AELSE	Conditionals
.AENDI	Conditionals
.AENDR	Loops
.AENDW	Loops
.AENDW	Loops
.AIF	Conditionals
.AIF EXPRA CMP EXPRB	Conditionals
.ALIGN SIZE	Other Commands
.ALTERNATE	Other Commands
.AREPEAT AEXP	Loops
.AWHILE EXPRA CMP EXPRB	Loops
.AWHILE STRA CMP STRB	Loops
.DATA EXPR, EXPR, ...	Initialized
.DATA.B EXPR, EXPR, ...	Initialized
.DATA.L EXPR, EXPR, ...	Initialized
.DATA.W EXPR, EXPR, ...	Initialized
.DATAB REPEAT, EXPR	Initialized
.DATAB.B REPEAT, EXPR	Initialized
.DATAB.L REPEAT, EXPR	Initialized
.DATAB.W REPEAT, EXPR	Initialized
.END	Other Commands
.ENDM	Macros
.EXITM	Macros
.EXPORT NAME	Other Commands
.FORM COL=COLS	Listings
.FORM LIN=LN	Listings
.FORM LIN=LN COL=COLS	Listings
.GLOBAL NAME	Other Commands
.HEADING STRING	Listings
.INCLUDE	Other Commands
.INSTR(String Builtins
.LEN(String Builtins
.MACRO MACNAME	Macros
.MACRO MACNAME MACARGS ...	Macros
.ORG	Other Commands
.PAGE	Listings
.PRINT LIST	Listings
.PRINT NOLIST	Listings
.PROGRAM	Other Commands
.RADIX S	Other Commands
.RES COUNT	Uninitialized
.RES.B COUNT	Uninitialized
.RES.L COUNT	Uninitialized
.RES.W COUNT	Uninitialized

.SDATA	Initialized
.SDATAB REPEAT,	Initialized
.SDATAC	Initialized
.SDATAZ	Initialized
.SRES COUNT	Uninitialized
.SRES.B COUNT	Uninitialized
.SRES.L COUNT	Uninitialized
.SRES.W COUNT	Uninitialized
.SRESC COUNT	Uninitialized
.SRESC.B COUNT	Uninitialized
.SRESC.L COUNT	Uninitialized
.SRESC.W COUNT	Uninitialized
.SRESZ COUNT	Uninitialized
.SRESZ.B COUNT	Uninitialized
.SRESZ.L COUNT	Uninitialized
.SRESZ.W COUNT	Uninitialized
.SUBSTR(String Builtins
! default comment char	Invoking GASP
; as comment char	Invoking GASP
INFILE ...	Invoking GASP
NAME .MACRO	Macros
NAME .MACRO (MACARGS ...)	Macros
PVAR .ASSIGN EXPR	Variables
PVAR .ASSIGNA AEXPR	Variables
PVAR .ASSIGNC	Variables
PVAR .EQU EXPR	Variables
PVAR .REG (REGISTER)	Variables
absolute expressions	Expressions
argument fields	Syntax Details
avoiding preprocessing	Markers
bang, as comment	Invoking GASP
breaking out of loops	Loops
comment character, changing	Invoking GASP
comments	Markers
continuation character	Markers
copying literally to output	Markers
directive field	Syntax Details
EQ	Conditionals
exclamation mark, as comment	Invoking GASP
fields of GASP source line	Syntax Details
GE	Conditionals
GT	Conditionals
label field	Syntax Details
LE	Conditionals
literal copy to output	Markers
LOCAL NAME [, ...]	Macros
loops, breaking out of	Loops
LT	Conditionals
macros, count executed	Macros
NE	Conditionals
number of macros executed	Macros
preprocessing, avoiding	Markers
relocatable expressions	Expressions
semicolon, as comment	Invoking GASP
shriek, as comment	Invoking GASP
symbol separator	Markers
symbols, separating from text	Markers

text, separating from symbols
whitespace
\@

Markers
Syntax Details
Macros
