NAME

rgbasm — language documentation

DESCRIPTION

This is the full description of the assembly language used by *rgbasm*(1). For the full description of instructions in the machine language supported by the Game Boy CPU, see *gbz80*(7).

It is advisable to have some familiarity with the Game Boy hardware before reading this document. RGBDS is specifically targeted at the Game Boy, and thus a lot of its features tie directly to its concepts. This document is not intended to be a Game Boy hardware reference.

Generally, "the linker" will refer to*r gblink*(1), but any program that processes RGBDS object files (described in *rgbds*(5)) can be used in its place.

SYNTAX

The syntax is line-based, just as in any other assembler. Each line may have components in this order:

[*directive*] [; *comment*]
[*label*:] [*instruction* [:: *instruction ...*]] [; *comment*]

Directives are commands to the assembler itself, such as **PRINTLN**, **SECTION**, or **OPT**.

Labels tie a name to a specific location within a section (see "Labels" below).

Instructions are assembled into Game Boy opcodes. Multiple instructions on one line can be separated by double colons '`::`'.

The available instructions are documented in *gbz80*(7).

Note that where an instruction requires an 8-bit register *r8*, **rgbasm** can interpret **HIGH**(*r16*) as the top 8-bit register of the given *r16*, for example, **HIGH**(**HL**) for **H**; and **LOW**(*r16*) as the bottom one, for example, **LOW**(**HL**) for **L** (except for **LOW**(**AF**), since **F** is not a valid register).

Note also that where an instruction requires a condition code *cc*, **rgbasm** can interpret !*cc* as the opposite condition code; for example, **!nz** for **z**.

All reserved keywords (directives, register names, etc.) are case-insensitive; all identifiers (labels and other symbol names) are case-sensitive.

Comments are used to give humans information about the code, such as explanations. The assembler *always* ignores comments and their contents.

There are two kinds of comments, inline and block. Inline comments are anything that follows a semicolon '`;`' not inside a string, until the end of the line. Block comments, beginning with '`/*`' and ending with '`*/`', can be split across multiple lines, or occur in the middle of an expression.

An example demonstrating these syntax features:

```
SECTION "My Code", ROM0  ; a directive
MyFunction:              ; a label
    push hl              ; an instruction
    /* ...and multiple instructions,
       with mixed case */
    ld a, [hli] :: LD H, [HL] :: Ld l, a
    pop /*wait for it*/ hl
    ret
```

Sometimes lines can be too long and it may be necessary to split them. To do so, put a backslash at the end of the line:

```
    DB 1, 2, 3, \
       4, 5, 6, \ ; Put it before any comments
       7, 8, 9
    DB "Hello, \  ; Space before the \ is included
world!"           ; Any leading space is included
```

Symbol interpolation

A funky feature is writing a symbol between `{braces}`, called "symbol interpolation". This will paste the symbol's contents as if they were part of the source file. If it is a string symbol, its characters are simply inserted as-is. If it is a numeric symbol, its value is converted to hexadecimal notation with a dollar sign '$' prepended.

Symbol interpolations can be nested, too!

```
DEF topic EQUS "life, the universe, and \"everything\""
DEF meaning EQUS "answer"
; Defines answer = 42
DEF {meaning} = 42
; Prints "The answer to life, the universe, and "everything" is $2A"
PRINTLN "The {meaning} to {topic} is {{meaning}}"
PURGE topic, meaning, {meaning}
```

Symbols can be *interpolated* even in the contexts that disable automatic *expansion* of string constants: `name` will be expanded in all of `DEF({name})`, `DEF {name} EQU/=/EQUS/etc ...`, `PURGE {name}`, and `MACRO {name}`, but, for example, won't be in `DEF(name)`.

It's possible to change the way symbols are printed by specifying a print format like so: `{fmt:symbol}`. The `fmt` specifier consists of these parts: `<sign><exact><align><pad><width><frac><prec><type>`. These parts are:

Part	Meaning
`<sign>`	May be '`+`' or '` `'. If specified, prints this character in front of non-negative numbers.
`<exact>`	May be '`#`'. If specified, prints the value in an "exact" format: with a base prefix for non-decimal integer types ('`$`', '`&`', or '`%`'); with a '`q`' precision suffix for fixed-point numbers; or with '`\`' escape characters for strings.
`<align>`	May be '`-`'. If specified, aligns left instead of right.
`<pad>`	May be '`0`'. If specified, pads right-aligned numbers with zeros instead of spaces.
`<width>`	May be one or more '`0`' – '`9`'. If specified, pads the value to this width, right-aligned with spaces by default.
`<frac>`	May be '`.`' followed by one or more '`0`' – '`9`'. If specified, prints this many fractional digits of a fixed-point number. Defaults to 5 digits, maximum 255 digits.
`<prec>`	May be '`q`' followed by one or more '`0`' – '`9`'. If specified, prints a fixed-point number at this precision. Defaults to the current `-Q` option.
`<type>`	Specifies the type of value.

All the format specifier parts are optional except the `<type>`. Valid print types are:

Type	Format	Example
'`d`	Signed decimal	-42'
'`u`	Unsigned decimal	42'
'`x`	Lowercase hexadecimal	2a'
'`X`	Uppercase hexadecimal	2A'
'`b`	Binary	101010'
'`o`	Octal	52'
'`f`	Fixed-point	1234.56789'
'`s`	String	string contents'

Examples:

```
SECTION "Test", ROM0[2]
X:                  ; This works with labels**whose address is known**
DEF Y = 3           ; This also works with variables
DEF SUM EQU X + Y   ; And likewise with numeric constants
; Prints "%0010 + $3 == 5"
PRINTLN "{#05b:X} + {#x:Y} == {d:SUM}"
```

```
rsset 32
DEF PERCENT rb 1   ; Same with offset constants
DEF VALUE = 20
DEF RESULT = MUL(20.0, 0.32)
; Prints "32% of 20 = 6.40"
PRINTLN "{d:PERCENT}% of {d:VALUE} = {f:RESULT}"

DEF WHO EQUS STRLWR("WORLD")
; Prints "Hello world!"
PRINTLN "Hello {s:WHO}!"
```

Although, for these examples, **`STRFMT`** would be more appropriate; see “String expressions” below.

EXPRESSIONS

An expression can be composed of many things. Numeric expressions are always evaluated using signed 32-bit math. Zero is considered to be the only "false" number, all non-zero numbers (including negative) are "true".

An expression is said to be "constant" if **`rgbasm`** knows its value. This is generally always the case, unless a label is involved, as explained in the “SYMBOLS” section. However, some operators can be constant even with non-constant operands, as explained in “Operators” below.

The instructions in the macro-language generally require constant expressions.

Numeric formats

There are a number of numeric formats.

Format type	Possible prefixes	Accepted characters
Decimal	none	0123456789
Hexadecimal	`$`, `0x`, `0X`	0123456789ABCDEF
Octal	`&`, `0o`, `0O`	01234567
Binary	`%`, `0b`, `0B`	01
Fixed-point	none	01234.56789
Precise fixed-point	none	12.34q8
Character constant	none	’ABYZ’
Game Boy graphics	`` ` ``	0123

Underscores are also accepted in numbers, except at the beginning of one. This can be useful for grouping digits, like `123_456` or `%1100_1001`.

The "character constant" form yields the value the character maps to in the current charmap. For example, by default (refer to *ascii*(7)) ‘’A’’ yields 65. A character constant must represent a single value, so it cannot include multiple characters, or characters which map to multiple values. See “Character maps” for information on charmaps, and “String expressions” for information on escape characters allowed in character constants.

The last one, Game Boy graphics, is quite interesting and useful. After the backtick, 8 digits between 0 and 3 are expected, corresponding to pixel values. The resulting value is the two bytes of tile data that would produce that row of pixels. For example, ‘`01012323’ is equivalent to ‘$0F55’.

You can also use symbols, which are implicitly replaced with their value.

Operators

You can use these operators in numeric expressions (listed from highest to lowest precedence):

Operator	Meaning
`( )`	Grouping
`FUNC()`	Built-in function call
**	Exponentiation

`+ - ~ !`	Unary plus, minus (ne gation), complement (bitwise negation), and Boolean negation
`* / %`	Multiplication, division, and modulo (remainder)
`<< >> >>>`	Bit shifts (left, sign-e xtended right, zero-extended right)
`& | ^`	Bitwise AND/OR/XOR
`+ -`	Addition and subtraction
`== != < > <= >=`	Comparisons
`&&`	Boolean AND
`||`	Boolean OR

‘**’ raises a number to a non-negative power. It is the only *right-associative* operator, meaning that `p ** q ** r` is equal to `p ** (q ** r)`, not `(p ** q) ** r`. All other binary operators are left-associative.

‘~’ complements a value by inverting all 32 of its bits.

‘%’ is used to get the remainder of the corresponding division, so that `x / y * y + x % y == x` is always true. The result has the same sign as the divisor. This makes `x % y` equal to `(x + y) % y` or `(x - y) % y`.

Shifting works by shifting all bits in the left operand either left (‘<<’) or right (‘>>’) by the right operand’s amount. When shifting left, all newly-inserted bits are reset; when shifting right, they are copies of the original most significant bit instead. This makes ‘a << b’ and ‘a >> b’ equivalent to multiplying and dividing by 2 to the power of b, respectively.

Comparison operators return 0 if the comparison is false, and 1 otherwise.

Unlike in many other languages, and for technical reasons, **`rgbasm`** still evaluates both operands of ‘&&’ and ‘||’.

The operators ‘&&’ and ‘&’ with a zero constant as either operand will be constant 0, and ‘||’ with a non-zero constant as either operand will be constant 1, even if the other operand is non-constant.

‘!’ returns 1 if the operand was 0, and 0 otherwise. Even a non-constant operand with any non-zero bits will return 0.

Integer functions

Besides operators, there are also some functions which have more specialized uses.

Name	Operation
`HIGH`(*n*)	Equivalent to ‘(*n* & $FF00) >> 8’.
`LOW`(*n*)	Equivalent to *n* & $FF. delim $$
`BITWIDTH`(*n*)	Returns the number of bits necessary to represent *n*. Some useful formulas: **`BITWIDTH`**(*n*) − 1 equals $⌊ log sub 2 (n) ⌋$, **`BITWIDTH`**(*n* − 1) equals $⌈ log sub 2 (n) ⌉$, and 32 − **`BITWIDTH`**(*n*) equals $roman clz (n)$.
`TZCOUNT`(*n*)	Returns $roman ctz (n)$, the count of trailing zero bits at the end of the binary representation of *n*.

delim off

Fixed-point expressions

Fixed-point numbers are technically just integers, but conceptually they have a decimal point at a fixed location (hence the name). This gives them increased precision, at the cost of a smaller range, while remaining far cheaper to manipulate than floating-point numbers (which **`rgbasm`** does not support).

The default precision of all fixed-point numbers is 16 bits, meaning the lower 16 bits are used for the fractional part; so they count in 65536ths of 1.0. This precision can be changed with the `-Q` command-line option, and/or by **`OPT Q`** (see “Changing options while assembling”). An individual fixed-point literal can specify its own precision, overriding the current default, by appending a “q” followed by the number of fractional bits: for example, `1234.5q8` is equal to $0004d2_80 delim $$ ($= 1234.5 * 2 sup 8$).

Since fixed-point values are still just integers, you can use them in normal integer expressions. You can easily truncate a fixed-point number into an integer by shifting it right by the number of fractional bits. It follows that you can convert an integer to a fixed-point number by shifting it left that same amount.

Note that the current number of fractional bits can be computed as **TZCOUNT**(1.0).

The following functions are designed to operate with fixed-point numbers:

Name	Operation
DIV(*x*, *y*)	Fixed-point division
MUL(*x*, *y*)	Fixed-point multiplication
FMOD(*x*, *y*)	Fixed-point modulo
POW(*x*, *y*)	$x sup y$
LOG(*x*, *y*)	Logarithm of x to the base y
ROUND(*x*)	Round x to the nearest integer
CEIL(*x*)	Round x up to the nearest integer
FLOOR(*x*)	Round x down to the nearest integer
SIN(*x*)	Sine of x
COS(*x*)	Cosine of x
TAN(*x*)	Tangent of x
ASIN(*x*)	Inverse sine of x
ACOS(*x*)	Inverse cosine of x
ATAN(*x*)	Inverse tangent of x
ATAN2(*y*, *x*)	Angle between (x, y) and $(1, 0)$

delim off

There are no functions for fixed-point addition and subtraction, because the ‘+’ and ‘-’ operators can add and subtract pairs of fixed-point operands.

> Note that some operators or functions are meaningful when combining integers and fixed-point values. For example, `2.0 * 3` is equivalent to `MUL(2.0, 3.0)`, and `6.0 / 2` is equivalent to `DIV(6.0, 2.0)`. Be careful and think about what the operations mean when doing this sort of thing.

All of these fixed-point functions can take an optional final argument, which is the precision to use for that one operation. For example, `MUL(6.0q8, 7.0q8, 8)` will evaluate to `42.0q8` no matter what value is set as the current `Q` option. **rgbasm** *does not check precisions for consistency*, so nonsensical input like `MUL(4.2q8, 6.9q12, 16)`will produce a nonsensical (b ut technically correct) result: “garbage in, garbage out”.

The **FMOD** function is used to get the remainder of the corresponding fixed-point division, so that `MUL(DIV(x, y), y) + FMOD(x, y) == x`is al ways true. The result has the same sign as the *dividend*; this is the opposite of how the integer modulo operator ‘%’ works!

The trigonometry functions (**SIN**, **COS**, **TAN**, etc) are defined in terms of a circle divided into 1.0 “turns” delim $$ (equal to $2 pi$ radians, or 360 degrees). delim off

These functions are useful for automatic generation of various tables. For example:

```
; Generate a table of 128 sine values
; from sin(0.0) to sin(0.5) excluded,
; with amplitude scaled from [-1.0, 1.0] to [0.0, 128.0].
FOR angle, 0.0, 0.5, 0.5 / 128
    db MUL(SIN(angle) + 1.0, 128.0 / 2) >> 16
ENDR
```

String expressions

The most basic string expression is any number of characters contained in double quotes (`"for instance"`). The backslash character ‘\’ is special in that it causes the character following it to be “escaped”, meaning that it is treated differently from normal. There are a number of escape sequences you can use within a string:

Sequence	Meaning
‘\\	Backslash (escapes the escape character itself)’
‘\"	Double quote (does not terminate a string)’
‘\’	Single quote (does not terminate a character literal)’
‘\{	Open curly brace (does not start interpolation)’
‘\}	Close curly brace (does not end interpolation)’
‘\n	Newline (ASCII $0A)’
‘\r	Carriage return (ASCII $0D)’
‘\t	Tab (ASCII $09)’
‘\0	Null (ASCII $00)’

Multi-line strings are contained in triple quotes (`"""for instance"""`). Escape sequences work the same way in multi-line strings; however, literal newline characters will be included as-is, without needing to escape them with ‘`\r`’ or ‘`\n`’.

Raw strings are prefixed by a hash ‘#’. Inside them, backslashes and braces are treated like regular characters, so they will not be expanded as macro arguments, interpolated symbols, or escape sequences. For example, the raw string `#"\t\1{s}\"` is equivalent to the regular string `"\\t\\1\{s}\\"`. (Note that this prevents raw strings from including the double quote character.) Raw strings also may be contained in triple quotes for them to be multi-line, so they can include literal newline or quote characters (although still not three quotes in a row).

You can use the ‘++’ operator to concatenate two strings. `"str" ++ "ing"` is equivalent to `"string"`, or to `STRCAT("str", "ing")`.

The following functions operate on string expressions, and return strings themselves.

Name	Operation
STRCAT(*strs...*)	Concatenates *strs*.
STRUPR(*str*)	Returns *str* with all ASCII letters (`a-z`) in uppercase.
STRLWR(*str*)	Returns *str* with all ASCII letters (`A-Z`) in lowercase.
STRSLICE(*str*, *start*, *stop*)	Returns a substring of *str* starting at *start* and ending at *stop* (exclusive). If *stop* is not specified, the substring continues to the end of *str*.
STRRPL(*str*, *old*, *new*)	Returns *str* with each non-overlapping occurrence of the substring *old* replaced with *new*.
STRFMT(*fmt*, *args...*)	Returns the string *fmt* with each `%spec` pattern replaced by interpolating the format *spec* (using the same syntax as “Symbol interpolation”) with its corresponding argument in *args* (‘%%’ is replaced by the ‘%’ character).
STRCHAR(*str*, *idx*)	Returns the substring of *str* for the charmap entry at *idx* with the current charmap. (*idx* counts charmap entries, not characters.)
REVCHAR(*vals...*)	Returns the string that is mapped to *vals* with the current charmap. If there is no unique charmap entry for *vals*, an error occurs.
READFILE(*name*, *max*)	Returns the contents of the file *name* as a string. Reads up to *max* bytes, or the entire contents if *max* is not specified. If the file isn’t found in the current directory, the include-path list passed to *rgbasm*(1)’s `-I` option on the command line will be searched.

The following functions operate on string expressions, but return integers.

Name	Operation
STRLEN(*str*)	Returns the number of characters in *str*.
STRCMP(*str1*, *str2*)	Compares *str1* and *str2* according to ASCII ordering of their characters. Returns -1 if *str1* is lower than *str2*, 1 if *str1* is greater than *str2*, or 0 if they match.
STRFIND(*str*, *sub*)	Returns the first index of *sub* in *str*, or -1 if it’s not present.

STRRFIND(*str*, *sub*) Returns the last index of *sub* in *str*, or -1 if it's not present.

BYTELEN(*str*) Returns the number of bytes in *str*. (Non-ASCII characters can be multiple bytes.)

STRBYTE(*str*, *idx*) Returns the byte value at *idx* in *str*.

INCHARMAP(*str*) Returns 1 if *str* has an entry in the current charmap, or 0 otherwise.

CHARLEN(*str*) Returns the number of charmap entries in *str* with the current charmap.

CHARCMP(*str1*, *str2*) Compares *str1* and *str2* according to their charmap entry values with the current charmap. Returns -1 if *str1* is lower than *str2*, 1 if *str1* is greater than *str2*, or 0 if they match.

CHARSIZE(*char*) Returns how many values are in the charmap entry for *char* with the current charmap.

CHARVAL(*char*, *idx*) Returns the value at *idx* of the charmap entry for *char*. If *idx* is not specified, *char* must have a single value, which is returned.

Note that indexes count starting from 0 at the beginning, or from -1 at the end. The characters of a string are counted by `STRLEN`; the charmap entries of a string are counted by `CHARLEN`; and the values of a charmap entry are counted by `CHARSIZE`.

Character maps

When writing text strings that are meant to be displayed on the Game Boy, the character encoding in the ROM may need to be different than the source file encoding. For example, the tiles used for uppercase letters may be placed starting at tile index 128, which differs from ASCII starting at 65.

Character maps allow mapping strings to arbitrary sequences of numbers:

```
CHARMAP "A", 42
CHARMAP ":)", 39
CHARMAP "<br>", 13, 10
CHARMAP "&euro;", $20ac
```

This would result in `db "Amen :)<br>"` being equivalent to `db 42, 109, 101, 110, 32, 39, 13, 10`, and `dw "25&euro;"` being equivalent to `dw 50, 53, $20ac`.

Any characters in a string without defined mappings will be copied directly, using the source file's encoding of characters to bytes.

It is possible to create multiple character maps and then switch between them as desired. This can be used to encode debug information in ASCII and use a different encoding for other purposes, for example. Initially, there is one character map called 'main' and it is automatically selected as the current character map from the beginning. There is also a character map stack that can be used to save and restore which character map is currently active.

Command	Meaning
NEWCHARMAP *name*	Creates a new, empty character map called *name* and switches to it.
NEWCHARMAP *name*, *basename*	Creates a new character map called *name*, copied from character map *basename*, and switches to it.
SETCHARMAP *name*	Switch to character map *name*.
PUSHC	Push the current character map onto the stack.
PUSHC *name*	Push the current character map onto the stack and switch to character map *name*.
POPC	Pop a character map off the stack and switch to it.

Note: Modifications to a character map take effect immediately from that point onward.

Other functions

There are a few other functions that do things beyond numeric or string operations:

Name	Operation

DEF(*symbol*) Returns 1 if *symbol* has been defined, 0 otherwise. String constants are not expanded within the parentheses.

ISCONST(*arg*) Returns 1 if *arg*'s value is known by RGBASM (e.g. if it can be an argument to **IF**), or 0 if only RGBLINK can compute its value.

BANK(*arg*) Returns a bank number. If *arg* is the symbol `@`, this function returns the bank of the current section. If *arg* is a string, it returns the bank of the section that has that name. If *arg* is a label, it returns the bank number the label is in. The result may be constant if **rgbasm** is able to compute it.

SECTION(*symbol*) Returns the name of the section that *symbol* is in. *symbol* must have been defined already.

SIZEOF(*arg*) If *arg* is a string, this function returns the size of the section named *arg*. If *arg* is a section type keyword, it returns the size of that section type. The result is not constant, since only RGBLINK can compute its value. If *arg* is an 8-bit or 16-bit re gister, it returns the size of that register.

STARTOF(*arg*) If *arg* is a string, this function returns the starting address of the section named *arg*. If *arg* is a section type keyword, it returns the starting address of that section type. The result is not constant, since only RGBLINK can compute its value.

SECTIONS

Before you can start writing code, you must define a section. This tells the assembler what kind of information follows and, if it is code, where to put it.

```
SECTION name, type
SECTION name, type, options
SECTION name, type[addr]
SECTION name, type[addr], options
```

name is a string enclosed in double quotes, and can be a new name or the name of an existing section. If the type doesn't match, an error occurs. All other sections must have a unique name, even in different source files, or the linker will treat it as an error.

Possible section *type*s are as follows:

ROM0 A ROM section. *addr* can range from *$0000* to *$3FFF*, or *$0000* to *$7FFF* if tin y ROM mode is enabled in the linker.

ROMX A banked ROM section. *addr* can range from *$4000* to *$7FFF*. *bank* can range from 1 to 511. Becomes an alias for **ROM0** if tiny ROM mode is enabled in the linker.

VRAM A banked video RAM section. *addr* can range from *$8000* to *$9FFF*. *bank* can be 0 or 1, b ut bank 1 is unavailable if DMG mode is enabled in the linker.

SRAM A banked external (save) RAM section. *addr* can range from *$A000* to *$BFFF*. *bank* can range from 0 to 15.

WRAM0 A general-purpose RAM section. *addr* can range from *$C000* to *$CFFF*, or *$C000* to *$DFFF* if WRAM0 mode is enabled in the linker.

WRAMX A banked general-purpose RAM section. *addr* can range from *$D000* to *$DFFF*. *bank* can range from 1 to 7. Becomes an alias for **WRAM0** if WRAM0 mode is enabled in the linker.

OAM An object attribute RAM section. *addr* can range from *$FE00* to *$FE9F*.

HRAM A high RAM section. *addr* can range from *$FF80* to *$FFFE*.

Since RGBDS produces ROMs, code and data can only be placed in **ROM0** and **ROMX** sections. To put some in RAM, have it stored in ROM, and copy it to RAM.

*option*s are comma-separated and may include:

BANK[*bank*]
: Specify which *bank* for the linker to place the section in. See above for possible values for *bank*, depending on *type*.

ALIGN[*align*, *offset*]
: Place the section at an address whose *align* least-significant bits are equal to *offset*. Note that **ALIGN**[*align*] is a shorthand for **ALIGN**[*align*, 0]. This option can be used with [*addr*], as long as they don't contradict each other. It's also possible to request alignment in the middle of a section; see "Requesting alignment" below.

If [*addr*] is not specified, the section is considered "floating"; the linker will automatically calculate an appropriate address for the section. Similarly, if **BANK**[*bank*] is not specified, the linker will automatically find a bank with enough space.

Sections can also be placed by using a linker script file. The format is described in *rgblink*(5). They allow the user to place floating sections in the desired bank in the order specified in the script. This is useful if the sections can't be placed at an address manually because the size may change, but they have to be together.

Section examples:

```
SECTION "Cool Stuff", ROMX
```

This switches to the section called "CoolStuff", creating it if it doesn't already exist. It can end up in any ROM bank. Code and data may follow.

If it is needed, the the base address of the section can be specified:

```
SECTION "Cool Stuff", ROMX[$4567]
```

An example with a fixed bank:

```
SECTION "Cool Stuff", ROMX[$4567], BANK[3]
```

And if you want to force only the section's bank, and not its position within the bank, that's also possible:

```
SECTION "Cool Stuff", ROMX, BANK[7]
```

Alignment examples: The first one could be useful for defining an OAM buffer to be DMA'd, since it must be aligned to 256 bytes. The second could also be appropriate for GBC HDMA, or for an optimized copy code that requires alignment.

```
SECTION "OAM Data", WRAM0, ALIGN[8] ; align to 256 bytes
SECTION "VRAM Data", ROMX, BANK[2], ALIGN[4] ; align to 16 bytes
```

The current section can be ended without starting a new section by using **ENDSECTION**. This directive will clear the section context, so you can no longer write code until you start another section. It can be useful to avoid accidentally defining code or data in the wrong section.

Section stack

POPS and **PUSHS** provide the interface to the section stack. The number of entries in the stack is limited only by the amount of memory in your machine.

PUSHS will push the current section context on the section stack. **POPS** can then later be used to restore it. Useful for defining sections in included files when you don't want to override the section context at the point the file was included.

PUSHS can also take the same arguments as **SECTION**, in order to push the current section context and define a new section at the same time:

```
SECTION "Code", ROM0
Function:
    ld a, 42
    PUSHS "Variables", WRAM0
```

```
        wAnswer: db
    POPS
    ld [wAnswer], a
```

RAM code

Sometimes you want to have some code in RAM. But then you can't simply put it in a RAM section, you have to store it in ROM and copy it to RAM at some point.

This means the code (or data) will not be stored in the place it gets executed. Luckily, **LOAD** blocks are the perfect solution to that. Here's an example of how to use them:

```
SECTION "LOAD example", ROMX
CopyCode:
    ld de, RAMCode
    ld hl, RAMLocation
    ld c, RAMCode.end - RAMCode
.loop
    ld a, [de]
    inc de
    ld [hli], a
    dec c
    jr nz, .loop
    ret

RAMCode:
  LOAD "RAM code", WRAM0
RAMLocation:
    ld hl, .string
    ld de, $9864
.copy
    ld a, [hli]
    ld [de], a
    inc de
    and a
    jr nz, .copy
    ret

.string
    db "Hello World!\0"
  ENDL
.end
```

A **LOAD** block feels similar to a **SECTION** declaration because it creates a new one. All data and code generated within such a block is placed in the current section like usual, but all labels are created as if they were placed in this newly-created section.

In the example above, all of the code and data will end up in the "LOAD example" section. You will notice the 'RAMCode' and 'RAMLocation' labels. The former is situated in ROM, where the code is stored, the latter in RAM, where the code will be loaded.

You cannot nest **LOAD** blocks, nor can you change or stop the current section within them.

The current **LOAD** block can be ended by using **ENDL**. This directive is only necessary if you want to resume writing code in its containing ROM section. Any of **LOAD**, **SECTION**, **ENDSECTION**, or **POPS** will end the current **LOAD** block before performing its own function.

LOAD blocks can use the **UNION** or **FRAGMENT** modifiers as described in "Unionized sections" below.

Unionized sections

When you're tight on RAM, you may want to define overlapping static memory allocations, as explained in the "Allocating overlapping spaces in RAM" section. However, a **UNION** only works within a single file, so it can't be used e.g. to define temporary variables across several files, all of which use the same statically allocated memory. Unionized sections solve this problem. To declare an unionized section, add a **UNION** keyword after the **SECTION** one; the declaration is otherwise not different. Unionized sections follow some different rules from normal sections:

- The same unionized section (i.e. having the same name) can be declared several times per **rgbasm** invocation, and across several invocations. Different declarations are treated and merged identically whether within the same invocation, or different ones.
- If one section has been declared as unionized, all sections with the same name must be declared unionized as well.
- All declarations must have the same type. For example, even if *rgblink*(1)'s `-w` flag is used, **WRAM0** and **WRAMX** types are still considered different.
- Different constraints (alignment, bank, etc.) can be specified for each unionized section declaration, but they must all be compatible. For example, alignment must be compatible with any fixed address, all specified banks must be the same, etc.
- Unionized sections cannot have type **ROM0** or **ROMX**.

Different declarations of the same unionized section are not appended, but instead overlaid on top of each other, just like "Allocating overlapping spaces in RAM". Similarly, the size of an unionized section is the largest of all its declarations.

Section fragments

Section fragments are sections with a small twist: when several of the same name are encountered, they are concatenated instead of producing an error. This works within the same file (paralleling the behavior "plain" sections has in previous versions), but also across object files. To declare an section fragment, add a **FRAGMENT** keyword after the **SECTION** one; the declaration is otherwise not different. However, similarly to "Unionized sections", some rules must be followed:

- If one section has been declared as fragment, all sections with the same name must be declared fragments as well.
- All declarations must have the same type. For example, even if *rgblink*(1)'s `-w` flag is used, **WRAM0** and **WRAMX** types are still considered different.
- Different constraints (alignment, bank, etc.) can be specified for each section fragment declaration, but they must all be compatible. For example, alignment must be compatible with any fixed address, all specified banks must be the same, etc.
- A section fragment may not be unionized; after all, that wouldn't make much sense.

When RGBASM merges two fragments, the one encountered later is appended to the one encountered earlier.

When RGBLINK merges two fragments, the one whose file was specified last is appended to the one whose file was specified first. For example, assuming `bar.o`, `baz.o`, and `foo.o` all contain a fragment with the same name, the command

```
rgblink -o rom.gb baz.o foo.o bar.o
```

would produce the fragment from `baz.o` first, followed by the one from `foo.o`, and the one from `bar.o` last.

Fragment literals

Fragment literals are useful for short blocks of code or data that are only referenced once. They are section fragments created by surrounding instructions or directives with '`[[`' double brackets '`]]`', without a separate **SECTION FRAGMENT** declaration.

The content of a fragment literal becomes a **SECTION FRAGMENT**, sharing the same name and bank as its parent ROM section, but without any other constraints. The parent section also becomes a **FRAGMENT** if it was not one already, so that it can be merged with its fragment literals. RGBLINK merges the fragments in no particular order.

A fragment literal can take the place of any 16-bit integer constant `n16` from the *gbz80*(7) documentation, as well as a **DW** item. The fragment literal then evaluates to its starting address. For example, you can **CALL** or **JP** to a fragment literal.

This code using named labels:

```
DataTable:
    dw First
    dw Second
    dw Third
First:  db 1
Second: db 4
Third:  db 9
Routine:
    push hl
    ld hl, Left
    jr z, .got_it
    ld hl, Right
.got_it
    call .print
    pop hl
    ret
.print:
    ld de, $1003
    ld bc, STARTOF(VRAM)
    jp Print
Left:  db "left\0"
Right: db "right\0"
```

is equivalent to this code using fragment literals:

```
DataTable:
    dw [[ db 1 ]]
    dw [[ db 4 ]]
    dw [[ db 9 ]]
Routine:
    push hl
    ld hl, [[ db "left\0" ]]
    jr z, .got_it
    ld hl, [[ db "right\0" ]]
.got_it
    call [[
        ld de, $1003
        ld bc, STARTOF(VRAM)
        jp Print
    ]]
    pop hl
    ret
```

The difference is that the example using fragment literals does not declare a particular order for its pieces.

Fragment literals can be arbitrarily nested, so extreme use cases are *technically* possible. This code using named labels:

```
dw FortyTwo
FortyTwo:
    call Sub1
    jr Sub2
Sub1:
    ld a, [Twenty]
    ret
Twenty: db 20
Sub2:
    jp Sub3
Sub3:
    call Sub1
    inc a
    add a
    ret
```

is equivalent to this code using fragment literals:

```
dw [[
    call [[
        Sub1: ld a, [ [[db 20]] ] :: ret
    ]]
    jr [[
        jp [[ call Sub1 :: inc a :: add a :: ret ]]
    ]]
]]
```

SYMBOLS

RGBDS supports several types of symbols:

Label Numeric symbol designating a memory location. May or may not have a value known at assembly time.

Constant Numeric symbol whose value has to be known at assembly time.

Macro A block of **`rgbasm`** code that can be invoked later.

String A text string that can be expanded later, similarly to a macro.

Symbol names can contain ASCII letters, numbers, underscores '_', hashes '#', dollar signs '$', and at signs '@'. However, they must begin with either a letter or an underscore. Additionally, label names can contain up to a single dot '`.`', which may not be the first character.

A symbol cannot have the same name as a reserved keyword, unless its name is a "raw identifier" prefixed by a hash '#'. For example, `#load` denotes a symbol named `load`, and `#LOAD` denotes a different symbol named `LOAD`; in both cases the '#' prevents them from being treated as the keyword **`LOAD`**.

Labels

One of the assembler's main tasks is to keep track of addresses for you, so you can work with meaningful names instead of "magic" numbers. Labels enable just that: a label ties a name to a specific location within a section. A label resolves to a bank and address, determined at the same time as its parent section's (see further in this section).

A label is defined by writing its name at the beginning of a line, followed by one or two colons, without any whitespace between the label name and the colon(s). Declaring a label (global or local) with two colons '`::`' will define and**`EXPORT`** it at the same time. (See "Exporting and importing symbols" belo w). When defining a local label, the colon can be omitted, and **`rgbasm`** will act as if there was only one.

A label is said to be *local* if its name contains a dot '`.`'; otherwise, it is said to be *global* (not to be mistaken with "exported", explained in "Exporting and importing symbols" below). More than one dot in label names is not allowed.

For convenience, local labels can use a shorthand syntax: when a symbol name starting with a dot is found (for example, inside an expression, or when declaring a label), then the current "label scope" is implicitly prepended.

Defining a global label sets it as the current "label scope", until the next global label definition, or the end of the current section.

Here are some examples of label definitions:

```
GlobalLabel:
AnotherGlobal:
.locallabel ; This defines "AnotherGlobal.locallabel"
.another_local:
AnotherGlobal.with_another_local:
ThisWillBeExported:: ; Note the two colons
ThisWillBeExported.too::
```

In a numeric expression, a label evaluates to its address in memory. (To obtain its bank, use the `BANK()` function described in "Other functions"). For example, given the following, `ld de, vPlayerTiles` would be equivalent to `ld de, $80C0` assuming the section ends up at *$80C0*:

```
SECTION "Player tiles", VRAM
vPlayerTiles:
    ds 6 * 16
.end
```

A label's location (and thus value) is usually not determined until the linking stage, so labels usually cannot be used as constants. However, if the section in which the label is defined has a fixed base address, its value is known at assembly time.

Also, while **`rgbasm`** obviously can compute the difference between two labels if both are constant, it is also able to compute the difference between two non-constant labels if they both belong to the same section, such as `PlayerTiles` and `PlayerTiles.end` above.

Anonymous labels

Anonymous labels are useful for short blocks of code. They are defined like normal labels, but without a name before the colon. Anonymous labels are independent of label scoping, so defining one does not change the scoped label, and referencing one is not affected by the current scoped label.

Anonymous labels are referenced using a colon '`:`' followed by pluses '`+`' or minuses '`-`'. Thus **`:+`** references the next one after the expression, **`:++`** the one after that; **`:-`** references the one before the expression; and so on.

```
    ld hl, :++
:   ld a, [hli] ; referenced by "jr nz"
    ldh [c], a
    dec c
    jr nz, :-
    ret

:   ; referenced by "ld hl"
    dw $7FFF, $1061, $03E0, $58A5
```

Variables

An equal sign '=' is used to define mutable numeric symbols. Unlike the other symbols described below, variables can be redefined. This is useful for internal symbols in macros, for counters, etc.

```
DEF ARRAY_SIZE EQU 4
DEF COUNT = 2
DEF COUNT = 3
DEF COUNT = ARRAY_SIZE + COUNT
```

```
DEF COUNT *= 2
; COUNT now has the value 14
```

Note that colons ‘:’ following the name are not allowed.

Variables can be conveniently redefined by compound assignment operators like in C:

Operator	Meaning
+= -=	Compound plus/minus
*= /= %=	Compound multiply/divide/modulo
<<= >>=	Compound shift left/right
&= |= ^=	Compound and/or/xor

Examples:

```
DEF x = 10
DEF x += 1    ; x == 11
DEF y = x - 1 ; y == 10
DEF y *= 2    ; y == 20
DEF y >>= 1   ; y == 10
DEF x ^= y    ; x == 1
```

Declaring a variable with **EXPORT DEF** or **EXPORT REDEF** will define and **EXPORT** it at the same time. (See “Exporting and importing symbols” below).

Numeric constants

EQU is used to define numeric constant symbols. Unlike ‘=’ above, constants defined this way cannot be redefined. These constants can be used for unchanging values such as properties of the hardware.

```
def SCREEN_WIDTH  equ 160 ; In pixels
def SCREEN_HEIGHT equ 144
```

Note that colons ‘:’ following the name are not allowed.

If you *really* need to, the **REDEF** keyword will define or redefine a numeric constant symbol. (It can also be used for variables, although it’s not necessary since they are mutable.) This can be used, for example, to update a constant using a macro, without making it mutable in general.

```
    def NUM_ITEMS equ 0
MACRO add_item
    redef NUM_ITEMS equ NUM_ITEMS + 1
    def ITEM_{02x:NUM_ITEMS} equ \1
ENDM
    add_item 1
    add_item 4
    add_item 9
    add_item 16
    assert NUM_ITEMS == 4
    assert ITEM_04 == 16
```

Declaring a numeric constant with **EXPORT DEF** or **EXPORT REDEF** will define and **EXPORT** it at the same time. (See “Exporting and importing symbols” below).

Offset constants

The RS group of commands is a handy way of defining structure offsets:

```
                RSRESET
DEF str_pStuff RW   1
DEF str_tData  RB   256
DEF str_bCount RB   1
DEF str_SIZEOF RB   0
```

The example defines four constants as if by:

```
DEF str_pStuff EQU 0
DEF str_tData  EQU 2
DEF str_bCount EQU 258
DEF str_SIZEOF EQU 259
```

There are five commands in the RS group of commands:

Command	Meaning
RSRESET	Equivalent to `RSSET 0`.
RSSET *constexpr*	Sets the **_RS** counter to *constexpr*.
DEF *name* **RB** *constexpr*	Sets *name* to **_RS** and then adds *constexpr* to **_RS**.
DEF *name* **RW** *constexpr*	Sets *name* to **_RS** and then adds *constexpr* * 2 to **_RS**.
DEF *name* **RL** *constexpr*	Sets *name* to **_RS** and then adds *constexpr* * 4 to **_RS**.

If the *constexpr* argument to **RB**, **RW**, or **RL** is omitted, it's assumed to be 1.

Note that colons ':' following the name are not allowed.

Declaring an offset constant with **EXPORT DEF** will define and **EXPORT** it at the same time. (See "Exporting and importing symbols" below).

String constants

EQUS is used to define string constant symbols. Wherever the assembler reads a string constant, it gets *expanded*: the symbol's name is replaced with its contents, similarly to **#define** in the C programming language. This expansion is disabled in a few contexts: `DEF(name)`, `DEF name EQU/=/EQUS/etc ...`, `PURGE name`, and `MACRO name` will not expand string constants in their names. Expansion is also disabled if the string constant's name is a raw identifier prefixed by a hash '#'.

```
DEF COUNTREG EQUS "[hl+]"
    ld a, COUNTREG

DEF PLAYER_NAME EQUS "\"John\""
    db PLAYER_NAME
```

This will be interpreted as:

```
    ld a, [hl+]
    db "John"
```

String constants can also be used to define small one-line macros:

```
DEF pusha EQUS "push af\npush bc\npush de\npush hl\n"
```

Note that colons ':' following the name are not allowed.

String constants, like numeric constants, cannot be redefined. However, the **REDEF** keyword will define or redefine a string constant symbol. For example:

```
DEF s EQUS "Hello, "
REDEF s EQUS "{s}world!"
; prints "Hello, world!"
PRINTLN "{s}\n"
```

String constants can't be exported or imported.

Important note: When a string constant is expanded, its expansion may contain another string constant, which will be expanded as well, and may be recursive. If this creates an infinite loop, **rgbasm** will error out once a certain depth is reached (see the `-r` command-line option in *rgbasm*(1)). The same problem can occur if the expansion of a string constant invokes a macro, which itself expands.

Macros

One of the best features of an assembler is the ability to write macros for it. Macros can be called with arguments, and can react depending on input using **IF** constructs.

```
MACRO my_macro
    ld a, 80
    call MyFunc
ENDM
```

The example above defines `my_macro` as a new macro. String constants are not expanded within the name of the macro.

Macros can't be exported or imported.

Nesting macro definitions is not possible, so this won't work:

```
MACRO outer
    MACRO inner
        PRINTLN "Hello!"
    ENDM ; this actually ends the 'outer' macro...
ENDM     ; ...and then this is a syntax error!
```

But you can work around this limitation using **EQUS**, so this will work:

```
MACRO outer
    DEF definition EQUS "MACRO inner\nPRINTLN \"Hello!\"\nENDM"
    definition
    PURGE definition
ENDM
```

More about how to define and invoke macros is described in "THE MACRO LANGUAGE" below.

Exporting and importing symbols

Importing and exporting of symbols is a feature that is very useful when your project spans many source files and, for example, you need to jump to a routine defined in another file.

Exporting of symbols has to be done manually, importing is done automatically if **rgbasm** finds a symbol it does not know about.

The following will cause *symbol1*, *symbol2* and so on to be accessible to other files during the link process:

EXPORT *symbol1* [, *symbol2*, ...]

For example, if you have the following three files:

`a.asm`:

```
SECTION "a", WRAM0
LabelA:
```

`b.asm`:

```
SECTION "b", WRAM0
ExportedLabelB1::
ExportedLabelB2:
    EXPORT ExportedLabelB2
```

`c.asm`:

```
SECTION "C", ROM0[0]
    dw LabelA
    dw ExportedLabelB1
    dw ExportedLabelB2
```

Then `c.asm` can use `ExportedLabelB1` and `ExportedLabelB2`, but not `LabelA`, so linking them together will fail:

```
$ rgbasm -o a.o a.asm
$ rgbasm -o b.o b.asm
$ rgbasm -o c.o c.asm
$ rgblink a.o b.o c.o
error: c.asm(2): Unknown symbol "LabelA"
Linking failed with 1 error
```

Note also that only exported symbols will appear in symbol and map files produced by *rgblink*(1).

Purging symbols

PURGE allows you to completely remove a symbol from the symbol table, as if it had never been defined. Be *very* careful when purging symbols, especially labels, because it could result in unpredictable errors if something depends on the missing symbol (for example, expressions the linker needs to calculate).

```
DEF Kamikaze EQUS "I don't want to live anymore"
AOLer: DB "Me too lol"
PURGE Kamikaze, AOLer
ASSERT !DEF(Kamikaze) && !DEF(AOLer)
```

String constants are not expanded within the symbol names.

Predeclared symbols

The following symbols are defined by the assembler:

Name	Type	Contents
@	**EQU**	PC value (essentially, the current memory address)
.	**EQUS**	The current global label scope
..	**EQUS**	The current local label scope
_RS	**=**	_RS Counter
_NARG	**EQU**	Number of arguments passed to macro, updated by **SHIFT**
__DATE__	**EQUS**	Today's date
__TIME__	**EQUS**	The current time
__ISO_8601_LOCAL__	**EQUS**	ISO 8601 timestamp (local)
__ISO_8601_UTC__	**EQUS**	ISO 8601 timestamp (UTC)
__UTC_YEAR__	**EQU**	Today's year
__UTC_MONTH__	**EQU**	Today's month number, 1–12
__UTC_DAY__	**EQU**	Today's day of the month, 1–31
__UTC_HOUR__	**EQU**	Current hour, 0–23
__UTC_MINUTE__	**EQU**	Current minute, 0–59
__UTC_SECOND__	**EQU**	Current second, 0–59
__RGBDS_MAJOR__	**EQU**	Major version number of RGBDS
__RGBDS_MINOR__	**EQU**	Minor version number of RGBDS
__RGBDS_PATCH__	**EQU**	Patch version number of RGBDS
__RGBDS_RC__	**EQU**	Release candidate ID of RGBDS, not defined for final releases
__RGBDS_VERSION__	**EQUS**	Version of RGBDS, as printed by `rgbasm --version`

The current time values will be taken from the `SOURCE_DATE_EPOCH` environment variable if that is defined as a UNIX timestamp. Refer to the spec at *reproducible-builds.org*: https://reproducible-builds.org/docs/source-date-epoch/.

DEFINING DATA

Defining constant data in ROM

DB defines a list of bytes that will be stored in the final image. Ideal for tables and text.

```
DB 1,2,3,4,"This is a string"
```

Alternatively, you can use **DW** to store a list of words (16-bit) or **DL** to store a list of double-words/longs (32-bit). Both of these write their data in little-endian byte order; for example, `dw $CAFE` is equivalent to `db $FE, $CA`and not `db $CA, $FE`.

Strings are handled a little specially: they first undergo charmap conversion (see “Character maps”), then each resulting character is output individually. For example, under the default charmap, the following two lines are identical:

```
DW "Hello!"
DW "H", "e", "l", "l", "o", "!"
```

If you do not want this special handling, enclose the string in parentheses.

DS can also be used to fill a region of memory with some repeated values. For example:

```
; outputs 3 bytes: $AA, $AA, $AA
DS 3, $AA
; outputs 7 bytes: $BB, $CC, $BB, $CC, $BB, $CC, $BB
DS 7, $BB, $CC
```

You can also use **DB**, **DW** and **DL** without arguments. This works exactly like **DS 1**, **DS 2** and **DS 4** respectively. Consequently, no-argument **DB**, **DW** and **DL** can be used in a **WRAM0** / **WRAMX** / **HRAM** / **VRAM** / **SRAM** section.

Including binary data files

You probably have some graphics, level data, etc. you’d like to include. Use **INCBIN** to include a raw binary file as it is. If the file isn’t found in the current directory, the include-path list passed to *rgbasm*(1)’s `-I` option on the command line will be searched.

```
INCBIN "titlepic.bin"
INCBIN "sprites/hero.bin"
```

You can also include only part of a file with **INCBIN**. The example below includes 256 bytes from data.bin, starting from byte 78.

```
INCBIN "data.bin", 78, 256
```

The length argument is optional. If only the start position is specified, the bytes from the start position until the end of the file will be included.

Statically allocating space in RAM

DS statically allocates a number of empty bytes. This is the preferred method of allocating space in a RAM section. You can also use **DB**, **DW** and **DL** without any arguments instead (see “Defining constant data in ROM” below).

```
DS 42 ; Allocates 42 bytes
```

Empty space in RAM sections will not be initialized. In ROM sections, it will be filled with the value passed to the `-p` command-line option, except when using overlays with `-O`.

Instead of an exact number of bytes, you can specify **ALIGN**[*align*, *offset*] to allocate however many bytes are required to align the subsequent data. Thus, ‘**DS ALIGN**[*align*, *offset*], ...’ is equivalent to ‘**DS** *n*, ...’ followed by ‘**ALIGN**[*align*, *offset*]’, where *n* is the minimum value needed to satisfy the **ALIGN** constraint (see “Requesting alignment” below). Note that **ALIGN**[*align*] is a shorthand for **ALIGN**[*align*, 0].

Allocating overlapping spaces in RAM

Unions allow multiple static memory allocations to overlap, like unions in C. This does not increase the amount of memory available, but allows re-using the same memory region for different purposes.

A union starts with a **UNION** keyword, and ends at the corresponding **ENDU** keyword. **NEXTU** separates each block of allocations, and you may use it as many times within a union as necessary.

```
; Let’s say PC == $C0DE here
UNION
    ; Here, PC == $C0DE
    wName:: ds 10
    ; Now, PC == $C0E8
```

```
    wNickname:: ds 10
    ; PC == $C0F2
NEXTU
    ; PC is back to $C0DE
    wHealth:: dw
    ; PC == $C0E0
    wLives:: db
    ; PC == $C0E1
    ds 7
    ; PC == $C0E8
    wBonus:: db
    ; PC == $C0E9
NEXTU
    ; PC is back to $C0DE again
    wVideoBuffer: ds 16
    ; PC == $C0EE
ENDU
; Afterward, PC == $C0F2
```

In the example above, ‘wName, wHealth’, and ‘wVideoBuffer’ all have the same value; so do ‘wNickname’ and ‘wBonus’. Thus, keep in mind that `ld [wHealth], a` assembles to the exact same thing as `ld [wName], a`.

This whole union’s total size is 20 bytes, the size of the largest block (the first one, containing ‘wName’ and ‘wNickname’).

Unions may be nested, with each inner union’s size being determined as above, and affecting its outer union like any other allocation.

Unions may be used in any section, but they may only contain space-allocating directives like **DS** (see “Statically allocating space in RAM”).

Requesting alignment

While **ALIGN** as presented in “SECTIONS” is often useful as-is, sometimes you instead want a particular piece of data (or code) in the middle of the section to be aligned. This is made easier through the use of mid-section **ALIGN** *align*, *offset*. It will retroactively alter the section’s attributes to ensure that the location the **ALIGN** directive is at, has its *align* lower bits equal to *offset*.

If the constraint cannot be met (for example because the section is fixed at an incompatible address), an error is produced. Note that **ALIGN** *align* is a shorthand for **ALIGN** *align*, 0.

There may be times when you don’t just want to specify an alignment constraint at the current location, but also skip ahead until the constraint can be satisfied. In that case, you can use **DS ALIGN**[*align*, *offset*] to allocate however many bytes are required to align the subsequent data.

If the constraint cannot be met by skipping any amount of space, an error is produced. Note that **ALIGN**[*align*] is a shorthand for **ALIGN**[*align*, 0].

THE MACRO LANGUAGE

Invoking macros

A macro is invoked by using its name at the beginning of a line, like a directive, followed by any comma-separated arguments.

```
add a, b
ld sp, hl
my_macro      ; This will be expanded
sub a, 87
my_macro 42   ; So will this
ret c
my_macro 1, 2 ; And this
```

After **rgbasm** has read the macro invocation line, it will expand the body of the macro (the lines between **MACRO** and **ENDM**) in its place.

Important note: When a macro body is expanded, its expansion may contain another macro invocation, which will be expanded as well, and may be recursive. If this creates an infinite loop,**rgbasm** will error out once a certain depth is reached (see the `-r` command-line option in *rgbasm*(1)). The same problem can occur if the expansion of a macro then expands a string constant, which itself expands.

It's possible to pass arguments to macros as well!

```
MACRO lb
    ld \1, (\2) << 8 | (\3)
ENDM
    lb hl, 20, 18        ; Expands to "ld hl, ((20) << 8) | (18)"
    lb de, 3 + 1, NUM**2 ; Expands to "ld de, ((3 + 1) << 8) | (NUM**2)"
```

You expand the arguments inside the macro body by using the escape sequences **\1** through **\9**, **\1** being the first argument, **\2** being the second, and so on. Since there are only nine digits, you can only use the first nine macro arguments that way. To use the rest, you put the argument number in angle brackets, like **\<10>**.

This bracketed syntax supports decimal numbers and numeric symbols, where negative values count from the last argument. For example, `\<_NARG>` or `\<-1>` will get the last argument.

Other macro arguments and symbol interpolations will also be expanded inside the angle brackets. For example, if '`\1`' is '`13`', then `\<\1>` inside the macro body will expand to `\<13>`. Or if`DEF v10 = 42` and `DEF x = 10`, then `\<v{d:x}>` will expand to `\<42>`.

Macro arguments are passed as string constants, although there's no need to enclose them in quotes. Thus, arguments are not evaluated as expressions, but instead are expanded directly inside the macro body. This means that they support all the escape sequences of strings (see "String expressions" above), as well as some of their own:

Sequence	Meaning
'`\,`	Comma (does not terminate the argument)'
'`\(`	Open parenthesis (does not start enclosing argument contents)'
'`\)`	Close parenthesis (does not end enclosing argument contents)'

Line continuations work as usual inside macros or lists of macro arguments. However, some characters need to be escaped, as in the following example:

```
MACRO PrintMacro1
    PRINTLN STRCAT(\1)
ENDM
    PrintMacro1 "Hello "\, \
                "world"
MACRO PrintMacro2
    PRINT \1
ENDM
    PrintMacro2 STRCAT("Hello ", \
                       "world\n")
```

The comma in `PrintMacro1` needs to be escaped to prevent it from starting another macro argument. The comma in `PrintMacro2` does not need escaping because it is inside parentheses, similar to macro arguments in the C programming language. The backslash in '`\n`' also does not need escaping because quoted string literals work as usual inside macro arguments.

Since macro arguments are expanded directly, it's often a good idea to put parentheses around them if they're meant as part of a numeric expression. For instance, consider the following:

```
MACRO print_double
    PRINTLN \1 * 3
ENDM
    print_double 1 + 2
```

The body will expand to `PRINTLN 1 + 2 * 3`, which will print 7 and not 9 as you might have expected.

The **`SHIFT`** directive is only available inside macro bodies. It shifts the argument numbers by one to the left, so what was **`\2`** is now **`\1`**, what was **`\3`** is now **`\2`**, and so forth. (What was **`\1`** is no longer accessible, so `_NARG` is decreased by 1.)

`SHIFT` can also take an integer parameter to shift that many times instead of once. A negative parameter will shift the arguments to the right, which can regain access to previously shifted ones.

`SHIFT` is especially useful in **`REPT`** loops to iterate over different arguments, evaluating the same loop body each time.

There are some escape sequences which are only valid inside the body of a macro:

Sequence	Meaning
'\1' – '\9'	The 1st–9th macro argument
`\<...>`	Further macro arguments
'`\#`	All `_NARG` macro arguments, separated by commas'
'`\@`	Unique symbol name affix (see below)'

The **`\@`** escape sequence is often useful in macros which define symbols. Suppose your macro expands to a loop of assembly code:

```
MACRO loop_c_times
    xor a, a
.loop
    ld [hl+], a
    dec c
    jr nz, .loop
ENDM
```

If you use this macro more than once in the same label scope, it will define `.loop` twice, which is an error. To work around this problem, you can use **`\@`** as a label suffix:

```
MACRO loop_c_times_fixed
    xor a, a
.loop\@
    ld [hl+], a
    dec c
    jr nz, .loop\@
ENDM
```

This will expand to a different value in each invocation, similar to **`gensym`** in the Lisp programming language.

`\@` also works in **`REPT`** blocks, expanding to a different value in each iteration.

Automatically repeating blocks of code

Suppose you want to unroll a time-consuming loop without copy-pasting it. **`REPT`** is here for that purpose. Everything between **`REPT`** and the matching **`ENDR`** will be repeated a number of times just as if you had done a copy/paste operation yourself. The following example will assemble `add a, c` four times:

```
REPT 4
    add a, c
ENDR
```

You can also use **REPT** to generate tables on the fly:

```
; Generate a table of square values from 0**2 = 0 to 100**2 = 10000
DEF x = 0
REPT 101
    dw x * x
    DEF x += 1
ENDR
```

As in macros, you can also use the escape sequence **\@**. **REPT** blocks can be nested.

A common pattern is to repeat a block for each value in some range. **FOR** is simpler than **REPT** for that purpose. Everything between **FOR** and the matching **ENDR** will be repeated for each value of a given symbol. String constants are not expanded within the symbol name. For example, this code will produce a table of squared values from 0 to 255:

```
FOR N, 256
    dw N * N
ENDR
```

It acts just as if you had done:

```
DEF N = 0
    dw N * N
DEF N = 1
    dw N * N
DEF N = 2
    dw N * N
; ...
DEF N = 255
    dw N * N
DEF N = 256
```

You can customize the range of **FOR** values, similarly to the `range` function in the Python programming language:

Code	**Range**
FOR *V*, *stop*	*V* increments from 0 to *stop*
FOR *V*, *start*, *stop*	*V* increments from *start* to *stop*
FOR *V*, *start*, *stop*, *stepV*	goes from *start* to *stop* by *step*

The **FOR** value will be updated by *step* until it reaches or exceeds *stop*, i.e. it covers the half-open range from *start* (inclusive) to *stop* (exclusive). The variable *V* will be assigned this value at the beginning of each new iteration; any changes made to it within the **FOR** loop's body will be overwritten. So the symbol *V* need not be already defined before any iterations of the **FOR** loop, but it must be a variable ("Variables") if so. For example:

```
FOR V, 4, 25, 5
    PRINT "{d:V} "
    DEF V *= 2
ENDR
    PRINTLN "done {d:V}"
```

This will print:

```
4 9 14 19 24 done 29
```

Just like with **REPT** blocks, you can use the escape sequence **\@** inside of **FOR** blocks, and they can be nested.

You can stop a repeating block with the **BREAK** command. A **BREAK** inside of a **REPT** or **FOR** block will interrupt the current iteration and not repeat any more. It will continue running code after the block's **ENDR**. For example:

```
FOR V, 1, 100
    PRINT "{d:V}"
    IF V == 5
        PRINT " stop! "
        BREAK
    ENDC
    PRINT ", "
ENDR
    PRINTLN "done {d:V}"
```

This will print:

```
1, 2, 3, 4, 5 stop! done 5
```

Conditionally assembling blocks of code

The four commands **IF**, **ELIF**, **ELSE**, and **ENDC** let you have **rgbasm** skip over parts of your code depending on a condition. This is a powerful feature commonly used in macros.

```
IF NUM < 0
  PRINTLN "NUM < 0"
ELIF NUM == 0
  PRINTLN "NUM == 0"
ELSE
  PRINTLN "NUM > 0"
ENDC
```

The **ELIF** (standing for "else if") and **ELSE** blocks are optional. **IF** / **ELIF** / **ELSE** / **ENDC** blocks can be nested.

Note that if an **ELSE** block is found before an **ELIF** block, the **ELIF** block will be ignored. All **ELIF** blocks must go before the **ELSE** block. Also, if there is more than one **ELSE** block, all of them but the first one are ignored.

Including other source files

Use **INCLUDE** to process another assembler file and then return to the current file when done. If the file isn't found in the current directory, the include-path list passed to *rgbasm*(1)'s `-I` option on the command line will be searched. You may nest **INCLUDE** calls infinitely (or until you run out of memory, whichever comes first).

```
INCLUDE "irq.inc"
```

You may also implicitly **INCLUDE** a file before the source file with the `-P` option of *rgbasm*(1).

Printing things during assembly

The **PRINT** and **PRINTLN** commands print text and values to the standard output. Useful for debugging macros, or wherever you may feel the need to tell yourself some important information.

```
PRINT "Hello world!\n"
PRINTLN "Hello world!"
PRINT _NARG, " arguments\n"
PRINTLN "sum: ", 2+3, " product: ", 2*3
PRINTLN STRFMT("E = %f", 2.718)
```

PRINT prints out each of its comma-separated arguments. Numbers are printed as unsigned uppercase hexadecimal with a leading '$'. For different formats, use **STRFMT**.

PRINTLN prints out each of its comma-separated arguments, if any, followed by a newline (‘\n’).

Aborting the assembly process

FAIL and **WARN** can be used to print errors and warnings respectively during the assembly process. This is especially useful for macros that get an invalid argument. **FAIL** and **WARN** take a string as the only argument and they will print this string out as a normal error with a line number.

FAIL stops assembling immediately while **WARN** shows the message but continues afterwards.

If you need to ensure some assumption is correct when compiling, you can use **ASSERT** and **STATIC_ASSERT**. Syntax examples are given below:

```
Function:
    xor a
ASSERT LOW(MyByte) == 0
    ld h, HIGH(MyByte)
    ld l, a
    ld a, [hli]
; You can also indent this!
    ASSERT BANK(OtherFunction) == BANK(Function)
    call OtherFunction
; Lowercase also works
    ld hl, FirstByte
    ld a, [hli]
assert FirstByte + 1 == SecondByte
    ld b, [hl]
    ret
.end
    ; If you specify one, a message will be printed
    STATIC_ASSERT .end - Function < 256, "Function is too large!"
```

First, the difference between **ASSERT** and **STATIC_ASSERT** is that the former is evaluated by RGBASM if it can, otherwise by RGBLINK; but the latter is only ever evaluated by RGBASM. If RGBASM cannot compute the value of the argument to **STATIC_ASSERT**, it will produce an error.

Second, as shown above, a string can be optionally added at the end, to give insight into what the assertion is checking.

Finally, you can add one of **WARN**, **FAIL** or **FATAL** as the first optional argument to either **ASSERT** or **STATIC_ASSERT**. If the assertion fails, **WARN** will cause a simple warning (controlled by *rgbasm*(1) flag `-Wassert`) to be emitted; **FAIL** (the default) will cause a non-fatal error; and **FATAL** immediately aborts.

MISCELLANEOUS

Changing options while assembling

OPT can be used to change some of the options during assembling from within the source, instead of defining them on the command-line. (See *rgbasm*(1)).

OPT takes a comma-separated list of options as its argument:

```
PUSHO
    OPT g.oOX, Wdiv       ; acts like command-line -g.oOX -Wdiv
    DW ‘..ooOOXX          ; uses the graphics constant characters from OPT g
    PRINTLN $80000000/-1  ; prints a warning about division
POPO
    DW ‘00112233          ; uses the default graphics constant characters
    PRINTLN $80000000/-1  ; no warning by default
```

OPT can modify the options `b`, `g`, `p`, `Q`, `r`, and `W`.

POPO and **PUSHO** provide the interface to the option stack. **PUSHO** will push the current set of options on the option stack. **POPO** can then later be used to restore them. Useful if you w ant to change some options in an include file and you don't want to destroy the options set by the program that included your file. The stack's number of entries is limited only by the amount of memory in your machine.

PUSHO can also take a comma-separated list of options, to push the current set and apply the argument set at the same time:

```
PUSHO b.X, g.oOX
    DB %..XXXX..
    DW '..ooOOXX
POPO
```

SEE ALSO

rgbasm(1), *rgblink*(1), *rgblink*(5), *rgbfix*(1), *rgbgfx*(1), *gbz80*(7), *rgbasm-old*(5), *rgbds*(5), *rgbds*(7)

HISTORY

rgbasm(1) was originally written by Carsten Sørensen as part of the ASMotor package, and was later repackaged in RGBDS by Justin Lloyd. It is now maintained by a number of contributors at https://github.com/gbdev/rgbds.