NAME

rgbasm — language documentation

DESCRIPTION

This is the full description of the language used by *rgbasm*(1). The description of the instructions supported by the Game Boy CPU is in *gbz80*(7).

It is strongly recommended 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, meaning that you do one instruction or directive per line:

```
[label][instruction][; comment]
```

Example:

```
John: ld a,87 ;Weee
```

All reserved keywords (directives, mnemonics, registers, etc.) are case-insensitive; all identifiers (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 syntaxes for comments. The most common is that anything that follows a semicolon '`;`' not inside a string, is a comment until the end of the line. The second is a block comment, beginning with '`/*`' and ending with '`*/`'. It can be split across multiple lines, or occur in the middle of an expression:

```
X = /* the value of x
       should be 3 */ 3
```

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 `{symbol}` within a string, called "symbol interpolation". This will paste the contents of `symbol` 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><prefix><align><pad><width><frac><type>`. These parts are:

Part	Meaning
`<sign>`	May be '`+`' or '` `'. If specified, prints this character in front of non-negative numbers.
`<prefix>`	May be '`#`'. If specified, prints the appropriate prefix for numbers, '`$`', '`&`', or '`%`'.
`<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 digits of a fixed-point fraction. Defaults to 5 digits, maximum 255 digits.
`<type>`	Specifies the type of value.

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

Print 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	"example"'

Examples:

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

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

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

Although, for these examples, **`STRFMT`** would be more approriate; see "String expressions" further 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.

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

Numeric formats

There are a number of numeric formats.

Format type	Prefix	Accepted characters
Hexadecimal	$	0123456789ABCDEF
Decimal	none	0123456789
Octal	&	01234567
Binary	%	01
Fixed point (Q16.16)	none	01234.56789
Character constant	none	"ABYZ"
Gameboy 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. See “Character maps” for information on charmaps.

The last one, Gameboy 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

A great number of operators you can use in expressions are available (listed from highest to lowest precedence):

Operator	Meaning
`( )`	Precedence o verride
`FUNC()`	Built-in function call
`**`	Exponent
`~ + -`	Unary complement/plus/minus
`* / %`	Multiply/divide/modulo
`<<`	Shift left
`>>`	Signed shift right (sign-extension)
`>>>`	Unsigned shift right (zero-extension)
`& | ^`	Binary and/or/xor
`+ -`	Add/subtract
`!= == <= >= < >`	Comparison
`&& ||`	Boolean and/or
`!`	Unary not

`~` complements a value by inverting all its bits.

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

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 a lot of languages, and for technical reasons, **rgbasm** still evaluates both operands of ‘&&’ and ‘||’.

! returns 1 if the operand was 0, and 0 otherwise.

Fixed-point expressions

Fixed-point numbers are basically normal (32-bit) integers, which count 65536ths instead of entire units, offering better precision than integers but limiting the range of values. The upper 16 bits are used for the integer part and the lower 16 bits are used for the fraction (65536ths). Since they are still akin to integers, you can use them in normal integer expressions, and some integer operators like '+' and '-' don't care whether the operands are integers or fixed-point. You can easily truncate a fixed-point number into an integer by shifting it right by 16 bits. It follows that you can convert an integer to a fixed-point number by shifting it left.

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

Name	Operation
DIV(*x*, *y*)	$x ÷ y$
MUL(*x*, *y*)	$x × y$
POW(*x*, *y*)	x to the y power
LOG(*x*, *y*)	Logarithm of x to the base y
ROUND(*x*)	Round x to the nearest integer
CEIL(*x*)	Round x up to an integer
FLOOR(*x*)	Round x down to an 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(*x*, *y*)	Angle between (x, y) and $(1, 0)$

delim off

The trigonometry functions (**SIN**, **COS**, **TAN**, etc) are defined in terms of a circle divided into 65535.0 degrees.

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

```
; Generate a 256-byte sine table with values in the range [0, 128]
; (shifted and scaled from the range [-1.0, 1.0])
ANGLE = 0.0
    REPT 256
        db (MUL(64.0, SIN(ANGLE)) + 64.0) >> 16
ANGLE = ANGLE + 256.0 ; 256.0 = 65536 degrees / 256 entries
    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:

String	Meaning
'`\\`	Produces a backslash'
'`\"`	Produces a double quote without terminating'
'`\{`	Curly bracket left'
'`\}`	Curly bracket right'
'`\n`	Newline ($0A)'
'`\r`	Carriage return ($0D)'

'`\t`	Tab ($09)'
"\1" – "\9"	Macro argument (Only in the body of a macro; see "Invoking macros")
'`\#`	All `_NARG` macro arguments, separated by commas (Only in the body of a macro)'
'`\@`	Label name suffix (Only in the body of a macro or a **`REPT`** block)'

(Note that some of those can be used outside of strings, when noted further in this document.)

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`'.

The following functions operate on string expressions. Most of them return a string, however some of these functions actually return an integer and can be used as part of an integer expression!

Name	**Operation**
STRLEN(*str*)	Returns the number of characters in *str*.
STRCAT(*strs...*)	Concatenates *strs*.
STRCMP(*str1*, *str2*)	Returns -1 if *str1* is alphabetically lower than *str2*, zero if they match, 1 if *str1* is greater than *str2*.
STRIN(*str1*, *str2*)	Returns the first position of *str2* in *str1* or zero if it's not present (first character is position 1).
STRRIN(*str1*, *str2*)	Returns the last position of *str2* in *str1* or zero if it's not present (first character is position 1).
STRSUB(*str*, *pos*, *len*)	Returns a substring from *str* starting at *pos* (first character is position 1, last is position -1) and *len* characters long. If *len* is not specified the substring continues to the end of *str*.
STRUPR(*str*)	Returns *str* with all letters in uppercase.
STRLWR(*str*)	Returns *str* with all letters in lowercase.
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).
CHARLEN(*str*)	Returns the number of charmap entries in *str* with the current charmap.
CHARSUB(*str*, *pos*)	Returns the substring for the charmap entry at *pos* in *str* (first character is position 1, last is position -1) with the current charmap.

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 8-bit values:

```
CHARMAP "<LF>", 10
CHARMAP "&iacute", 20
CHARMAP "A", 128
```

This would result in `db "Amen<LF>"` being equivalent to `db 128, 109, 101, 110, 10`.

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.
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 various useful things:

Name	Operation
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.
SIZEOF(*arg*)	Returns the size of the section named *arg*. The result is not constant, since only RGBLINK can compute its value.
STARTOF(*arg*)	Returns the starting address of the section named *arg*. The result is not constant, since only RGBLINK can compute its value.
DEF(*symbol*)	Returns TRUE (1) if *symbol* has been defined, FALSE (0) otherwise. String constants are not expanded within the parentheses.
HIGH(*arg*)	Returns the top 8 bits of the operand if *arg* is a label or constant, or the top 8-bit register if it is a 16-bit register.
LOW(*arg*)	Returns the bottom 8 bits of the operand if *arg* is a label or constant, or the bottom 8-bit register if it is a 16-bit register (`AF` isn't a valid register for this function).
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.

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*.

Note: While **rgbasm** will automatically optimize **ld** instructions to the smaller and faster **ldh** (see *gbz80*(7)) whenever possible, it is generally unable to do so when a label is involved. Using the **ldh** instruction directly is recommended. This forces the assembler to emit a **ldh** instruction and the linker to check if the value is in the correct range.

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 eachother. 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
```

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.

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, RAMLocation.end - RAMLocation
.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
.end
  ENDL
```

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 the current section within them.

LOAD blocks can use the **UNION** or **FRAGMENT** modifiers, as described below.

Unionized sections

When you're tight on RAM, you may want to define overlapping static memory allocations, as explained in the "Unions" 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 eachother, just like "Unions". 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 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.
- 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.

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 '#' 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.

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" further 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
PlayerTiles:
    ds 6 * 16
```

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
COUNT = 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
```

Numeric constants

EQU is used to define immutable numeric 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
```

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*.
RB *constexpr*	Sets the preceding symbol to **_RS** and adds *constexpr* to **_RS**.
RW *constexpr*	Sets the preceding symbol to **_RS** and adds *constexpr* * 2 to **_RS**.
RL *constexpr*	Sets the preceding symbol to **_RS** and adds *constexpr* * 4 to **_RS**.

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

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

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. If you are familiar with C, you can think of it as similar to **`#define .`**

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.

```
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 can't be exported or imported.

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}0
```

Important note: When a string constant is expanded, its expansion may contain another string constant, which will be expanded as well. 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 invokes another macro, recursively.

The examples above for `EQU`, '`=`', '`RB`', '`RW`', '`RL`', and `EQUS` all start with `DEF`. (A variable definition may start with `REDEF` instead, since they are redefinable.) You may use the older syntax without `DEF`, but then the name being defined *must not* have any whitespace before it; otherwise **rgbasm** will treat it as a macro invocation. Furthermore, without the`DEF` k eyword, string constants may be expanded for the name. This can lead to surprising results:

```
X EQUS "Y"
; this defines Y, not X!
X EQU 42
; prints "Y $2A"
PRINTLN "{X} {Y}"
```

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 MyMacro
        ld a, 80
        call MyFunc
ENDM
```

The example above defines `MyMacro` as a new macro. String constants are not expanded within the name of the macro. You may use the older syntax `MyMacro: MACRO` instead of `MACRO MyMacro`, with a single colon '`:`' following the macro's name. With the older syntax, string constants may be expanded for the name.

Macros can't be exported or imported.

Plainly nesting macro definitions is not allowed, but this can be worked around using **EQUS**. So this won't work:

```
MACRO outer
    MACRO inner
        PRINTLN "Hello!"
    ENDM
ENDM
```

But this will:

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

Macro arguments support all the escape sequences of strings, as well as '`\,`' to escape commas, as well as '`\(`' and '`\)`' to escape parentheses, since those otherwise separate and enclose arguments, respectively.

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 existed. *USE WITH EXTREME CAUTION!!!* I can't stress this enough, **you seriously need to know what you are doing**. DON'T purge a symbol that you use in expressions the linker needs to calculate. When not sure, it's probably not safe to purge anything other than variables, numeric or string constants, or macros.

```
DEF Kamikaze EQUS  "I don't want to live anymore"
DEF AOLer    EQUS  "Me too"
         PURGE Kamikaze, 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)
`_RS`	**=**	_RS Counter
`_NARG`	**EQU**	Number of arguments passed to macro, updated by **SHIFT**
`__LINE__`	**EQU**	The current line number
`__FILE__`	**EQUS**	The current filename
`__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 https://reproducible-builds.org/docs/source-date-epoch/.

DEFINING DATA

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, e xcept when using overlays with `-O`.

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 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) (see the `-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.

Unions

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
Name: ds 8
    ; PC = $C0E6
Nickname: ds 8
    ; PC = $C0EE
    NEXTU
    ; PC is back to $C0DE
Health: dw
    ; PC = $C0E0
Something: ds 6
    ; And so on
Lives: db
    NEXTU
VideoBuffer: ds 19
    ENDU
```

In the example above, 'Name, Health, VideoBuffer' all have the same value, as do 'Nickname' and 'Lives'. Thus, keep in mind that **ld [Health], a** is identical to **ld [Name], a**.

The size of this union is 19 bytes, as this is the size of the largest block (the last one, containing 'VideoBuffer'). Nesting unions is possible, with each inner union's size being considered as described above.

Unions may be used in any section, but inside them may only be **DS -** like commands (see "Statically allocating space in RAM").

THE MACRO LANGUAGE

Invoking macros

You execute the macro by inserting its name.

```
add a,b
ld sp,hl
MyMacro ; This will be expanded
sub a,87
```

It's valid to call a macro from a macro (yes, even the same one).

When **rgbasm** sees **MyMacro** it will insert the macro definition (the code enclosed in **MACRO** / **ENDM**).

Suppose your macro contains a loop.

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

This is fine, but only if you use the macro no more than once per scope. To get around this problem, there is the escape sequence **\@** that expands to a unique string.

\@ also works in **REPT** blocks.

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

Important note: Since a macro can call itself (or a different macro that calls the first one), there can be circular dependency problems. If this creates an infinite loop, **rgbasm** will error out once a certain depth is reached. See the `-r` command-line option in *r gbasm*(1). Also, a macro can have inside an **EQUS** which references the same macro, which has the same problem.

It's possible to pass arguments to macros as well! You retrieve the arguments by using the escape sequences **\1** through **\9**, **\1** being the first argument specified on the macro invocation.

```
MACRO LoopyMacro
            ld   hl,\1
            ld   c,\2
            xor  a,a
.loop\@     ld   [hl+],a
            dec  c
            jr   nz,.loop\@
            ENDM
```

Now you can call the macro specifying two arguments, the first being the address and the second being a byte count. The generated code will then reset all bytes in this range.

```
LoopyMacro MyVars,54
```

Arguments are passed as string constants, although there's no need to enclose them in quotes. Thus, an expression will not be evaluated first but kind of copy-pasted. This means that it's probably a very good idea to use brackets around **\1** to **\9** if you perform further calculations on them. For instance, consider the following:

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

The **PRINTLN** statement will expand to `PRINTLN 1 + 2 * 2`, which will print 5 and not 6 as you might have expected.

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 C. The backslash in '`\n`' also does not need escaping because string literals work as usual inside macro arguments.

Since there are only nine digits, you can only access the first nine macro arguments like this. To use the rest, you need to put the multi-digit argument number in angle brackets, like `\<10>`. This bracketed syntax supports decimal numbers and numeric constant symbols. For example, `\<_NARG>` will get the last argument.

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

Another way to access more than nine macro arguments is the **`SHIFT`** command, a special command only available in macros. It will shift the arguments by one to the left, and decrease `_NARG` by 1. **`\1`** will get the value of **`\2`**, **`\2`** will get the value of **`\3`**, and so forth.

`SHIFT` can optionally be given an integer parameter, and will apply the above shifting that number of times. A negative parameter will shift the arguments in reverse.

`SHIFT` is useful in **`REPT`** blocks to repeat the same commands with multiple arguments.

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 "Line #", __LINE__
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 line feed ('`\n`').

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 256-byte sine table with values in the range [0, 128]
; (shifted and scaled from the range [-1.0, 1.0])
ANGLE = 0.0
    REPT 256
        db (MUL(64.0, SIN(ANGLE)) + 64.0) >> 16
ANGLE = ANGLE + 256.0 ; 256.0 = 65536 degrees / 256 entries
```

```
    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:

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

You can customize the range of **FOR** values, similarly to Python's `range` function:

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*, *step*	*V* goes from *start* to *stop* by *step*

The **FOR** value will be updated by *step* until it reaches or exceeds *stop*. For example:

```
FOR V, 4, 25, 5
      PRINT "{d:V} "
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
```

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.

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 (see the `-i` option in *rgbasm*(1)) will be searched. You may nest **INCLUDE** calls infinitely (or until you run out of memory, whichever comes first).

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

Conditional assembling

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.

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, L    ; acts like command-line -g.oOX -Wdiv -L
    DW '..ooOOXX          ; uses the graphics constant characters from OPT g
    PRINTLN $80000000/-1  ; prints a warning about division
    LD [$FF88], A         ; encoded as LD, not LDH
POPO
    DW '00112233          ; uses the default graphics constant characters
    PRINTLN $80000000/-1  ; no warning by default
    LD [$FF88], A         ; optimized to use LDH by default
```

The options that **OPT** can modify are currently: `b`, `g`, `p`, `r`, `h`, `L`, and `W`. The Boolean flag options`h` and `L` can be negated as `OPT !h` and `OPT !L` to act like omitting them from the command-line.

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.

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 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.

SEE ALSO

rgbasm(1), *rgblink*(1), *rgblink*(5), *rgbds*(5), *rgbds*(7), *gbz80*(7)

HISTORY

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