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 RGB object files (described in *rgbds*(5)) can be used in its place.

SYNTAX

The syntax is lineâbased, just as in any other assembler, meaning that you do one instruction or pseudoâop per line:

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

Example:

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

All reserved keywords (pseudoâops, mnemonics, registers etc.) are caseâinsensitive, 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 other is that lines beginning with a '`*`' (not even spaces before it) are ignored. This second syntax is deprecated (will be removed in a future version) and should be replaced with the first one.

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
```

This works anywhere in the code except inside of strings. To split strings it is needed to use **`STRCAT`**() like this:

```
db STRCAT("Hello ", \
          "world!")
```

EXPRESSIONS

An expression can be composed of many things. Numerical 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 (16.16)	none	01234.56789
Character constant	none	"ABYZ"
Gameboy graphics	`	0123

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
`~ + -`	Unary complement/plus/minus
`* / %`	Multiply/divide/modulo
`<< >>`	Shift left/right
`& | ^`	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. '5 % 2' is 1.

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 1 otherwise.

Fixedâpoint Expressions

Fixed-point numbers are basically normal (32-bit) integers, which count 65536th's 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*) $x × y$
SIN(*x*) $sin (x)$
COS(*x*) $cos (x)$
TAN(*x*) $tan (x)$
ASIN(*x*) $asin (x)$
ACOS(*x*) $acos (x)$
ATAN(*x*) $atan (x)$
ATAN2(*x*, *y*) Angle between (x, y) and $(1, 0)$
delim off

These functions are useful for automatic generation of various tables. Example: assuming a circle has 65536.0 degrees, and sine values are in range [-1.0 ; 1.0]:

```
; --
; -- Generate a 256-byte sine table with values between 0 and 128
; --
ANGLE = 0.0
      REPT 256
      db MUL(64.0, SIN(ANGLE) + 1.0) >> 16
ANGLE = ANGLE + 256.0 ; 256 = 65536 / table_len, with table_len = 256
      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:

StringMeaning
'`\\`Produces a backslash'
'`\"`Produces a double quote without terminating'
'`\,`Comma'
'`\{`Curly bracket left'
'`\}`Curly bracket right'
'`\n`Newline ($0A)'
'`\r`Carriage return ($0D)'
'`\t`Tab ($09)'
"\1" – "\9"Macro argument (Only the body of a macro, see "Invoking macros")
'\@Label name suffix (Only in the body of macros and REPTs)'

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

A funky feature is `{symbol}` within a string, called "symbol interpolation". This will paste *`symbol`*'s contents as a string. If it's a string symbol, the string is simply inserted. If it's a numeric symbol, its value is converted to hexadecimal notation with a dollar sign '$' prepended.

```
TOPIC equs "life, the universe, and everything"
ANSWER = 42
; Prints "The answer to life, the universe, and everything is $2A"
PRINTT "The answer to {TOPIC} is {ANSWER}\n"
```

Symbol interpolations can be nested, too!

It's possible to change the way numeric symbols are converted by specifying a print type like so: `{d:symbol}`. Valid print types are:

Print type Format Example

‘`d`	Decimal	42’
‘`x`	Lowercase hexadecimal	2a’
‘`X`	Uppercase hexadecimal	2A’
‘`b`	Binary	101010’

Note that print types should only be used with numeric values, not strings.

HINT: The **`{symbol}`** construct can also be used outside strings. The symbol's value is again inserted directly.

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(*string*)	Returns the number of characters in *string*.
STRCAT(*str1*, *str2*)	Appends *str2* to *str1*.
STRCMP(*str1*, *str2*)	Returns negative if *str1* is alphabetically lower than *str2*, zero if they match, positive if *str1* is greater than *str2*.
STRIN(*str1*, *str2*)	Returns the 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) and *len* characters long.
STRUPR(*str*)	Converts all characters in *str* to capitals and returns the new string.
STRLWR(*str*)	Converts all characters in *str* to lower case and returns the new string.

Character maps

When writing text that is meant to be displayed in the Game Boy, the characters used in the source code may have a different encoding than the default of ASCII. For example, the tiles used for uppercase letters may be placed starting at tile index 128, which makes it difficult to add text strings to the ROM.

Character maps allow mapping strings up to 16 characters long to an abitrary 8-bit value:

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

By default, a character map contains ASCII encoding.

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*.
NEWCHARMAP *name*, *basename*	Creates a new character map called *name*, copied from character map *basename*.
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: Character maps affect all strings in the file from the point in which they are defined, until switching to a different character map. This means that any string that the code may want to print as debug information will also be affected by it.

Note: The output value of a mapping can be 0. If this happens, the assembler will treat this as the end of the string and the rest of it will be trimmed.

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.
DEF(*label*)	Returns TRUE (1) if *label* has been defined, FALSE (0) otherwise. String symbols 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. All sections assembled at the same time that have the same name are considered to be the same section, and their code is put together in the object file generated by the assembler. 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*]
: Place the section at an address whose *align* leastâsignificant bits are zero. This option can be used with *addr*, as long as they don't contradict eachother.

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 "CoolStuff",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 "CoolStuff",ROMX[$4567]
```

An example with a fixed bank:

```
SECTION "CoolStuff",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 "CoolStuff",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.

Unionized Sections

When you’re tight on RAM, you may want to define overlapping blocks of variables, as explained in the “Unions” section. However, the **UNION** keyword only works within a single file, which prevents e.g. defining temporary variables on a single memory area across several files. 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 (= 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.
- A section cannot be declared both as unionized or non-unionized.
- 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.

SYMBOLS

RGBDS supports several types of symbols:

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

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

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

String equate String symbol that can be evaluated, similarly to a macro.

Symbol names can contain letters, numbers, underscores, hashes and ‘@’. However, they must begin with either a letter, a number, or an underscore. Periods are allowed exclusively for labels, as described below. A symbol cannot have the same name as a reserved keyword. *In the line where a symbol is defined there mustn’t be any whitespace before it*, otherwise **rgbasm** will treat it as a macro invocation.

Label declaration

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.

This can be done in a number of ways:

```
GlobalLabel ; This syntax is deprecated,
AnotherGlobal: ; please use this instead
.locallabel
.yet_a_local:
AnotherGlobal.with_another_local:
ThisWillBeExported:: ; Note the two colons
ThisWillBeExported.too::
```

Declaring a label (global or local) with ‘`::`’ does an**EXPORT** at the same time. (See “Exporting and importing symbols” below).

Any label whose name does not contain a period is a global label, others are locals. Declaring a global label sets it as the current label scope until the next one; any local label whose first character is a period will have the global label’s name implicitly prepended. Local labels can be declared as `scope.local:` or simply as as `.local:`. If the former notation is used, then `scope` must be the actual current scope.

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 declared has a fixed base address, its value is known at assembly time.

rgbasm is able to compute the subtraction of two labels either if both are constant as described above, or if both belong to the same section.

EQU **EQU** allows defining constant symbols. Unlike **SET** below, constants defined this way cannot be redefined. They can, for example, be used for things such as bit definitions of hardware registers.

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

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

SET **SET**, or its synonym **=**, defines constant symbols like **EQU**, but those constants can be re-defined. This is useful for variables in macros, for counters, etc.

```
ARRAY_SIZE EQU 4
COUNT      SET 2
COUNT      SET ARRAY_SIZE+COUNT
```

```
; COUNT now has the value 6
COUNT      = COUNT + 1
```

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

RSSET, **RSRESET**, **RB**, **RW**

The RS group of commands is a handy way of defining structures:

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

The example defines four constants as if by:

```
str_pStuff EQU 0
str_tData  EQU 2
str_bCount EQU 258
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**. (In practice, this one cannot be used due to a bug).

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

EQUS **EQUS** is used to define string symbols. Wherever the assembler meets a string symbol its name is replaced with its value. If you are familiar with C you can think of it as similar to **`#define`**.

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

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

This will be interpreted as:

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

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

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

Note that colons ‘`:`’ following the name are not allowed. String equates can’t be exported or imported.

Important note: An **EQUS** can be expanded to a string that contains another **EQUS** and it 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 *r gbasm*(1). Also, a macro can contain an**EQUS** which calls the same macro, which causes the same problem.

MACRO 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.

```
MyMacro: MACRO
         ld   a,80
         call MyFunc
         ENDM
```

Note that a single colon ‘`:`’ following the macro’s name is required. Macros can’t be exported or imported.

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`*, ...]

`GLOBAL` is a deprecated synonym for **`EXPORT`**, do not use it.

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 string symbols, macros, and constants.

```
Kamikaze EQUS  "I don't want to live anymore"
AOLer    EQUS  "Me too"
         PURGE Kamikaze, AOLer
```

Note that, as an exception, string symbols in the argument list of a **`PURGE`** command *will not be expanded*.

Predeclared Symbols

The following symbols are defined by the assembler:

Type	Name	Contents
`EQU`	`@`	PC value
`EQU`	`_PI`	Fixed point π
`SET`	`_RS`	_RS Counter
`EQU`	`_NARG`	Number of arguments passed to macro
`EQU`	`__LINE__`	The current line number
`EQUS`	`__FILE__`	The current filename
`EQUS`	`__DATE__`	Today’s date
`EQUS`	`__TIME__`	The current time
`EQUS`	`__ISO_8601_LOCAL__`	ISO 8601 timestamp (local)
`EQUS`	`__ISO_8601_UTC__`	ISO 8601 timestamp (UTC)
`EQU`	`__UTC_YEAR__`	Today’s year
`EQU`	`__UTC_MONTH__`	Today’s month number, 1–12
`EQU`	`__UTC_DAY__`	Today’s day of the month, 1–31
`EQU`	`__UTC_HOUR__`	Current hour, 0–23
`EQU`	`__UTC_MINUTE__`	Current minute, 0–59
`EQU`	`__UTC_SECOND__`	Current second, 0–59
`EQU`	`__RGBDS_MAJOR__`	Major version number of RGBDS
`EQU`	`__RGBDS_MINOR__`	Minor version number of RGBDS
`EQU`	`__RGBDS_PATCH__`	Patch version number of RGBDS

DEFINING DATA

Declaring variables in a RAM section

DS 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" 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

DB defines a list of bytes that will be stored in the final image. Ideal for tables and text. Note that strings are not zero-terminated!

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

DS can also be used to fill a region of memory with some value. The following produces 42 times the byte $FF:

```
DS 42, $FF
```

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). Strings are not allowed as arguments to **DW** and **DL**.

You can also use **DB**, **DW** and **DL** without arguments, or leaving empty elements at any point in the list. 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
```

Unions

Unions allow multiple 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 "Declaring variables in a RAM section").

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.

```
LoopyMacro: MACRO
            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.

```
LoopyMacro: MACRO
            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.

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

Now I 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 equates, 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:

```
print_double: MACRO
    PRINTV \1 * 2
ENDM
    print_double 1 + 2
```

The **PRINTV** statement will expand to `PRINTV 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:

```
PrintMacro: MACRO
    PRINTT \1
ENDM

    PrintMacro STRCAT("Hello "\, \
                      "world\\n")
```

The comma needs to be escaped to avoid it being treated as separating the macro's arguments. The backslash '\' (from '\n') also needs to be escaped because of the way **rgbasm** processes macro arguments.

In reality, up to 256 arguments can be passed to a macro, but you can only use the first 9 like this. If you want to use the rest, you need to use the **SHIFT** command.

SHIFT is a special command only available in macros. Very useful in **REPT** blocks. It will shift the arguments by one to the left. **\1** will get the value of **\2**, **\2** will get the value of **\3**, and so forth.

This is the only way of accessing the value of arguments from 10 to 256.

SHIFT can optionally be given an integer parameter, and will apply the above shifting that number of times.

Printing things during assembly

The next four 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.

```
PRINTT "I'm the greatest programmer in the whole wide world\n"
PRINTI (2 + 3) / 5
PRINTV $FF00 + $F0
PRINTF MUL(3.14, 3987.0)
```

PRINTT prints out a string. Be careful to add a line feed ("\n") at the end, as it is not added automatically.

PRINTV prints out an integer value in hexadecimal or, as in the example, the result of a calculation. Unsurprisingly, you can also print out a constant symbol's value.

PRINTI prints out a signed integer value.

PRINTF prints out a fixed point value.

Be careful that none of those automatically print a line feed; if you need one, use **PRINTT \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 between 0 and 128
; --
ANGLE =   0.0
      REPT  256
      db    (MUL(64.0, SIN(ANGLE)) + 64.0) >> 16
ANGLE = ANGLE+256.0
      ENDR
```

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

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(Variable) == 0
      ld h, HIGH(Variable)
      ld l, a
      ld a, [hli]
      ; You can also indent this!
      ASSERT BANK(OtherFunction) == BANK(Function)
      call OtherFunction
; Lowercase also works
assert Variable + 1 == OtherVariable
      ld c, [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
  PRINTT "NUM < 0\n"
ELIF NUM == 0
  PRINTT "NUM == 0\n"
ELSE
  PRINTT "NUM > 0\n"
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.

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

```
PUSHO
OPT   g.oOX ;Set the GB graphics constants to use these characters
DW    '..ooOOXX
POPO
DW    '00112233
```

The options that OPT can modify are currently: `b`, `g` and `p`.

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.

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/rednex/rgbds.