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 GameBoy CPU is in gbz80(7).

GENERAL

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 pseudoâops, mnemonics and registers (reserved keywords) are caseâinsensitive and all labels are caseâsensitive.

There are two syntaxes for comments. In both cases, a comment ends at the end of the line. The most common one is: anything that follows a semicolon ";" (that isn't inside a string) is a comment. There is another format: anything that follows a "∗" that is placed right at the start of a line is a comment. The assembler removes all comments from the code before doing anything else.

Sometimes lines can be too long and it may be necessary to split them. The syntax to do so is the following one:

```
DB 1, 2, 3, 4 \
   5, 6, 7, 8
```

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!")
```

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 "CoolStuff",ROMX
```

This switches to the section called "CoolStuff" (or creates it if it doesn't already exist) and it defines it as a code section. All sections assembled at the same time that have the same name, type, etc, are considered to be the same one, and their code is put together in the object file generated by the assembler. All other sections must have a unique name, even in different source files, or the linker will treat it as an error.

Possible section types are as follows:

ROM0 A ROM section. Mapped to memory at $0000â$3FFF (or $0000-$7FFF if tiny ROM mode is enabled in rgblink(1)).

ROMX
A banked ROM section. Mapped to memory at $4000â$7FFF. Valid banks range from 1 to 511. Not available if tiny ROM mode is enabled in rgblink(1).

VRAM
A banked video RAM section. Mapped to memory at $8000â$9FFF. Can only allocate memory, not fill it. Valid banks are 0 and 1 but bank 1 isn't available if DMG mode is enabled in `rgblink`(1).

SRAM A banked external (save) RAM section. Mapped to memory at $A000â$BFFF. Can only allocate memory, not fill it. Valid banks range from 0 to 15.

WRAM0
A general-purpose RAM section. Mapped to memory at $C000â$CFFF, or $C000-$DFFF if DMG mode is enabled in `rgblink`(1). Can only allocate memory, not fill it.

WRAMX
A banked general-purpose RAM section. Mapped to memory at $D000â$DFFF. Can only allocate memory, not fill it. Valid banks range from 1 to 7. Not available if DMG mode is enabled in `rgblink`(1).

OAM An object attributes RAM section. Mapped to memory at $FE00-$FE9F. Can only allocate memory, not fill it.

HRAM
A high RAM section. Mapped to memory at $FF80â$FFFE. Can only allocate memory, not fill it.

NOTE: If you use this method of allocating HRAM the assembler will NOT choose the short addressing mode in the LD instructions **LD [$FF00+n8],A** and **LD A,[$FF00+n8]** because the actual address calculation is done by the linker. If you find this undesirable you can use**RSSET** / **RB** / **RW** instead or use the **LDH [$FF00+n8],A** and **LDH A,[$FF00+n8]** syntax instead. This forces the assembler to emit the correct instruction and the linker to check if the value is in the correct range. This optimization can be disabled by passing the **-L** flag to **rgbasm** as explained in `rgbasm`(1).

A section is usually defined as a floating one, but the code can restrict where the linker can place it.

If a section is defined with no indications, it is a floating section. The linker will decide where to place it in the final binary and it has no obligation to follow any specific rules. The following example defines a section that can be placed anywhere in any ROMX bank:

```
SECTION "CoolStuff",ROMX
```

If it is needed, the following syntax can be used to fix the base address of the section:

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

It won't, however, fix the bank number, which is left to the linker. If you also want to specify the bank you can do:

```
SECTION "CoolStuff",ROMX[$4567],BANK[3]
```

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

```
SECTION "CoolStuff",ROMX,BANK[7]
```

In addition, you can specify byte alignment for a section. This ensures that the section starts at a memory address where the given number of least-significant bits are 0. This can be used along with **BANK**, if desired. However, if an alignment is specified, the base address must be left unassigned. This can be useful when using DMA to copy data or when it is needed to align the start of an array to 256 bytes to optimize the code that accesses it.

```
SECTION "OAM Data",WRAM0,ALIGN[8] ; align to 256 bytes
```

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

HINT: If you think this is a lot of typing for doing a simple **ORG** type thing you can quite easily write an intelligent macro (called **ORG** for example) that uses **@** for the section name and determines correct section type etc as arguments for **SECTION**.

POPS and **PUSHS** provide the interface to the section stack. **PUSHS** will push the current section conte xt 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 destroy the section context for the program that included your file. The number of entries in the stack is limited only by the amount of memory in your machine.

Sections can also be placed by using a linkerscript 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.

SYMBOLS

Symbols

RGBDS supports several types of symbols:

Label Used to assign a memory location with a name

EQUate Give a constant a name.

SET Almost the same as EQUate, but you can change the value of a SET during assembling.

Structure (**the RS group**) Define a structure easily.

String equate (**EQUS**) Give a frequently used string a name. Can also be used as a mini-macro, like #define in C.

MACRO A block of code or pseudo instructions that you invoke like any other mnemonic. You can give them arguments too.

A symbol cannot have the same name as a reserved keyword.

Label

One of the assembler's main tasks is to keep track of addresses for you so you don't have to remember obscure numbers but can make do with a meaningful name, a label.

This can be done in a number of ways:

```
GlobalLabel
AnotherGlobal:
.locallabel
.yet_a_local:
AnotherGlobal.with_another_local:
ThisWillBeExported:: ;note the two colons
ThisWillBeExported.too::
```

In the line where a label is defined there musn't be any whitespace before it. Local labels are only accessible within the scope they are defined. A scope starts after a global label and ends at the next global label. Declaring a label (global or local) with :: does an EXPORT at the same time. Local labels can be declared as scope.local or simply as as .local. If the former notation is used, the scope must be the actual current scope.

Labels will normally change their value during the link process and are thus not constant. The exception is the case in which the base address of a section is fixed, so the address of the label is known at assembly time.

The subtraction of two labels is only constant (known at assembly time) if they are two local labels that belong to the same scope, or they are two global labels that belong to sections with fixed base addresses.

EQU

EQUates are constant symbols. They can, for example, be used for things such as bit-definitions of hardware registers.

```
EXIT_OK      EQU $00
EXIT_FAILURE EQU $01
```

Note that a colon (:) following the label-name is not allowed. EQUates cannot be exported and imported. They don't change their value during the link process.

SET

SETs are similar to EQUates. They are also constant symbols in the sense that their values are defined during the assembly process. These symbols are normally used in macros.

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

Note that a colon (:) following the label-name is not allowed. SETs cannot be exported and imported. Alternatively you can use = as a synonym for SET.

```
COUNT = 2
```

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 equated symbols:

```
str_pStuff = 0
str_tData  = 2
str_bCount = 258
str_SIZEOF = 259
```

There are four commands in the RS group of commands:

Command	**Meaning**
RSRESET	Resets the _RS counter to zero.
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.**

Note that a colon (:) following the symbol-name is not allowed. **RS** symbols cannot be exported and imported. They don't change their value during the link process.

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 the same as #define.

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

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

Note that : following the label-name is not allowed, and that strings must be quoted to be useful.

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 a colon (:) following the label-name is 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. This means that, if you aren't careful, you may trap the assembler into an infinite loop if there's a circular dependency in the expansions. Also, a MACRO can have inside an EQUS which references the same MACRO, which has the same problem.

MACRO

One of the best features of an assembler is the ability to write macros for it. Macros also provide a method of passing arguments to them and they can then react to the input using IF-constructs.

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

Note that a colon (:) following the macro-name is required. Macros can't be exported or imported. It's valid to call a macro from a macro (yes, even the same one).

The above example is a very simple macro. You execute the macro by typing its name.

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

When the assembler meets MyMacro it will insert the macrodefinition (the text 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. That is, if you only use the macro once per scope. To get around this problem there is a special label string equate called **\@** that you can append to your labels and it will then expand to a unique string.

\@ also works in REPT-blocks should you have any loops there.

```
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 problems of circular dependency. They trap the assembler in an infinite loop, so you have to be careful when using recursion with MACROs. Also, a MACRO can have inside an EQUS which references the same MACRO, which has the same problem.

Macro Arguments

I'd like LoopyMacro a lot better if I didn't have to pre-load the registers with values and then call it. What I'd like is the ability to pass it arguments and it then loaded the registers itself.

And I can do that. In macros you can get the arguments by using the special macro string equates **\1** through **\9**, **\1** being the first argument specified on the calling of the macro.

```
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 bytecount. The macro will then reset all bytes in this range.

```
LoopyMacro MyVars,54
```

Arguments are passed as string equates. There's no need to enclose them in quotes. An expression will not be evaluated first but passed directly. 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, if you pass 1 + 2 as the first argument and then do **PRINTV \1** $*$ 2 you will get the value 5 on screen and not 6 as you might have expected.

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 keyword **SHIFT**.

Line continuations work as usual inside macros or lists of arguments of macros. Strings, however, are a bit trickier. The following example shows how to use strings as arguments for a macro:

```
PrintMacro : MACRO
    PRINTT \1
ENDM

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

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 v alue of **\2**, **\2** will get the value in **\3** and so forth.

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

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 the assembler doesn't know where a symbol is defined.

EXPORT *label* [, *label* , ...]

The assembler will make label accessible to other files during the link process.

GLOBAL *label* [, *label* , ...]

If label is defined during the assembly it will be exported, if not, it will be imported. Handy (very!) for include-files. Note that, since importing is done automatically, this keyword has the same effect as **EXPORT**.

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 symbol that you use in expressions the linker needs to calculate. In fact, it's probably not even safe to purge anything other than string symbols and macros.

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

Note that string symbols that are part of a **PURGE** command WILL NOT BE EXPANDED as the ONLY exception to this rule.

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

Defining constant data

DB defines a list of bytes that will be stored in the final image. Ideal for tables and text (which is not zero-terminated).

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

Alternatively, you can use **DW** to store a list of words (16-bits) or **DL** to store a list of doublewords/longs (32-bits). 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, **DB**, **DW** and **DL** can be used in a **WRAM0** / **WRAMX** / **HRAM** / **VRAM** / **SRAM** section.

Declaring variables in a RAM section

DS allocates a number of bytes. The content is undefined. This is the preferred method of allocationg space in a RAM section. You can, however, use **DB**, **DW** and **DL** without any arguments instead.

```
DS str_SIZEOF ;allocate str_SIZEOF bytes
```

Including binary files

You probably have some graphics 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 the linker on the command line will be searched.

```
INCBIN "titlepic.bin"
INCBIN "sprites/hero.bin" ; UNIX
INCBIN "sprites\\hero.bin" ; Windows
```

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 share the same space in memory, like unions in C. This allows you to easily reuse memory for different purposes, depending on the game's state.

You create unions using the **UNION**, **NEXTU** and **ENDU** keywords. **NEXTU** lets you create a new block of allocations, and you may use it as many times within a union as necessary.

```
UNION
Name: ds 8
Nickname: ds 8
NEXTU
Health: dw
Something: ds 3
Lives: db
NEXTU
Temporary: ds 19
ENDU
```

This union will use up 19 bytes, as this is the size of the largest block (the last one, containing 'Temporary'). Of course, as 'Name', 'Health', and 'Temporary' all point to the same memory locations, writes to any one of these will affect values read from the others.

Unions may be used in any section, but code and data may not be included.

THE MACRO LANGUAGE

Printing things during assembly

These three instructions type text and values to stdout. 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.

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 symbols value.

PRINTI prints out a signed integer value.

PRINTF prints out a fixed point value.

Automatically repeating blocks of code

Suppose you're feeling lazy and you want to unroll a time consuming loop. **REPT** is here for that purpose. Everything between **REPT** and **ENDR** will be repeated a number of times just as if you 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 SET   0.0
      REPT  256
      DB    (MUL(64.0,SIN(ANGLE))+64.0)>>16
ANGLE SET ANGLE+256.0
      ENDR
```

REPT is also very useful in recursive macros and, as in macros, you can also use the special label operator **\@**. 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.

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 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** are used to conditionally assemble parts of your file. 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** 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.

Integer and Boolean expressions

An expression can be composed of many things. Expressions are always evaluated using signed 32-bit math.

The most basic expression is just a single number.

Numeric Formats

There are a number of numeric formats.

- Hexadecimal: $0123456789ABCDEF. Case-insensitive
- Decimal: 0123456789
- Octal: &01234567
- Binary: %01
- Fixedpoint (16.16): 01234.56789
- Character constant: "ABYZ"
- Gameboy graphics: `0123

The last one, Gameboy graphics, is quite interesting and useful. The values are actually pixel values and it converts the "chunky" data to "planar" data as used in the Gameboy.

```
DW `01012323
```

Admittedly, an expression with just a single number is quite boring. To spice things up a bit there are a few operators you can use to perform calculations between numbers.

Operators

A great number of operators you can use in expressions are available (listed in order of precedence):

Operator	Meaning
()	Precedence override
FUNC()	Function call

`~ + -`	Unary not/plus/minus
`* / %`	Multiply/divide/modulo
`<< >>`	Shift left/right
`& | ^`	Binary and/or/xor
`+ -`	Add/subtract
`!= == <=`	Boolean comparison
`>= < >`	Boolean comparison (Same precedence as the others)
`&& ||`	Boolean and/or
`!`	Unary Boolean not

The result of the boolean operators is zero if when FALSE and non-zero when TRUE. It is legal to use an integer as the condition for IF blocks. You can use symbols instead of numbers in your expression if you wish.

An expression is said to be constant when it doesn't change its value during linking. This basically means that you can't use labels in those expressions. The instructions in the macro-language all require expressions that are constant. The only exception is the subtraction of labels in the same section or labels that belong to sections with a fixed base addresses, all of which must be defined in the same source file (the calculation cannot be passed to the object file generated by the assembler). In this case, the result is a constant that can be calculated at assembly time.

Fixedâpoint Expressions

Fixed point constants are basically normal 32-bit constants where the upper 16 bits are used for the integer part and the lower 16 bits are used for the fraction (65536ths). This means that you can use them in normal integer expression, and some integer operators like plus and minus don't care whether the operands are integer or fixed-point. You can easily convert a fixed-point number to an integer by shifting it right 16 bits. It follows that you can convert an integer to a fixed-point number by shifting it left.

Some things are different for fixed-point math, though, which is why you have the following functions to use:

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)`	arcsin(x)
`ACOS(x)`	arccos(x)
`ATAN(x)`	arctan(x)
`ATAN2(x,y)`	Angle between (x,y) and (1,0)

These functions are extremely useful for automatic generation of various tables. A circle has 65536.0 degrees. Sine values are between [-1.0; 1.0].

```
; --
; -- Generate a 256 byte sine table with values between 0 and 128
; --
ANGLE SET   0.0
      REPT  256
      DB    (MUL(64.0,SIN(ANGLE))+64.0)>>16
ANGLE SET ANGLE+256.0
      ENDR
```

String Expressions

The most basic string expression is any number of characters contained in double quotes ("for instance"). Like in C, the escape character is \, and there are a number of commands you can use within a string:

String	Meaning
`\\`	Backslash
`\"`	Double quote
`\,`	Comma
`\{`	Curly bracket left
`\}`	Curly bracket right
`\n`	Newline ($0A)
`\t`	Tab ($09)
`\1 - \9`	Macro argument (Only the body of a macros)
`\@`	Label name suffix (Only in the body of macros and repts)

A funky feature is **{symbol}** withing a string. This will examine the type of the symbol and insert its value accordingly. If symbol is a string symbol, the symbols value is simply copied. If it's a numeric symbol, the value is converted to hexadecimal notation and inserted as a string.

HINT: The **{symbol}** construct can also be used outside strings. The symbol's value is again inserted as a string. This is just a short way of doing "{symbol}".

Whenever the macro-language expects a string you can actually use a string expression. This consists of one or more of these function (yes, you can nest them). Note that 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 with len characters.
`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 ASCII characters used in the source code may not be the same ones used in the tileset used in the ROM. 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 the code to map strings up to 16 characters long to an abitrary 8-bit value:

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

Note: Character maps affect all strings in the file from the point in which they are defined. 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(@/str/lbl)`	Returns a bank number. If the argument is the symbol `@`, this function returns the bank of the current section. If the argument is a string, it returns the bank of the section that has that name. If the argument is a label, it returns the bank number the label is in. For labels, as the linker has to resolve this, it can't be used when the expression has to be constant.
`DEF(label)`	Returns TRUE if label has been defined.
`HIGH(r16/cnst/lbl)`	Returns the top 8 bits of the operand if it is a label or constant, or the top 8-bit register if it is a 16-bit register.
`LOW(r16/cnst/lbl)`	Returns the bottom 8 bits of the operand if it 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).

MISCELLANEOUS

Changing options while assembling

`OPT` can be used to change some of the options during assembling the source instead of defining them on the commandline.

`OPT` takes a comma-seperated 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**, **e** and **g**.

`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 stacks number of entries is limited only by the amount of memory in your machine.

ALPHABETICAL LIST OF KEYWORDS

@
__DATE__
__FILE__
__ISO_8601_LOCAL__
__ISO_8601_UTC__
__LINE__
__TIME__
__RGBDS_MAJOR__
__RGBDS_MINOR__
__RGBDS_PATCH__
__UTC_YEAR__
__UTC_MONTH__
__UTC_DAY__
__UTC_HOUR__
__UTC_MINUTE__
__UTC_SECOND__

_NARG
_PI
_RS
ACOS
ASIN
ATAN
ATAN2
BANK
CHARMAP
COS
DB
DEF
DIV
DL
DS
DW
ELIF
ELSE
ENDC
ENDM
ENDR
EQU
EQUS
EXPORT
FAIL
GLOBAL
HIGH
HRAM
IF
INCBIN
INCLUDE
LOW
MACRO
MUL
OPT
POPO
POPS
PRINTF
PRINTI
PRINTT
PRINTV
PURGE
PUSHO
PUSHS
REPT
RB
RL
ROM0
ROMX

RSRESET
RSSET
RW
SECTION
SET
SHIFT
SIN
SRAM
STRCAT
STRCMP
STRIN
STRLEN
STRLWR
STRSUB
STRUPR
TAN
VRAM
WRAM0
WRAMX
WARN

SEE ALSO

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

HISTORY

`rgbds` 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**