

NAME

rgbds — object file format documentation

DESCRIPTION

This is the description of the object files used by `rgbasm(1)` and `rgblink(1)`. *Please note that the specification is not stable yet.* RGBDS is still in active development, and some new features require adding more information to the object file, or modifying some fields, both of which break compatibility with older versions.

FILE STRUCTURE

The following types are used:

LONG is a 32-bit integer stored in little-endian format. **BYTE** is an 8-bit integer. **STRING** is a 0-terminated string of **BYTE**. Brackets after a type (e.g. **LONG**[*n*]) indicate *n* consecutive elements (here, **LONG**s). All items are contiguous, with no padding anywhere—this also means that they may not be aligned in the file!

REPT *n* indicates that the fields between the **REPT** and corresponding **ENDR** are repeated *n* times.

All IDs refer to objects within the file; for example, symbol ID \$0001 refers to the second symbol defined in *this* object file's **Symbols** array. The only exception is the **Source file info** nodes, whose IDs are backwards, i.e. source node ID \$0000 refers to the *last* node in the array, not the first one. References to other object files are made by imports (symbols), by name (sections), etc.—but never by ID.

Header

BYTE *Magic*[4]
"RGB9"

LONG *RevisionNumber*

The format's revision number this file uses. (This is always in the same place in all revisions.)

LONG *NumberOfSymbols*

How many symbols are defined in this object file.

LONG *NumberOfSections*

How many sections are defined in this object file.

Source file info

LONG *NumberOfNodes*

The number of source context nodes contained in this file.

REPT *NumberOfNodes*

LONG *ParentID*

ID of the parent node, -1 meaning that this is the root node.

Important: the nodes are actually written in **reverse** order, meaning the node with ID 0 is the last one in the list!

LONG *ParentLineNo*

Line at which the parent node's context was exited; meaningless for the root node.

BYTE *Type*

Value	Meaning
0	REPT node
1	File node
2	Macro node

IF *Type* ≠ 0

If the node is not a REPT node...

```

        STRING Name
            The node's name: either a file name, or the macro's name prefixes by its defini-
            tion's file name (e.g. src/includes/defines.asm::error).
ELSE If the node is a REPT, it also contains the iteration counter of all parent REPTs.

        LONG Depth
        LONG Iter[Depth]
            The number of REPT iterations, by increasing depth.
ENDC
ENDR

```

Symbols

```

REPT NumberOfSymbols
    STRING Name
        This symbol's name. Local symbols are stored as their full name (Scope.symbol).
    BYTE Type
        Value  Meaning
        0      Local symbol only used in this file.
        1      Import of an exported symbol (by name) from another object file.
        2      Exported symbol visible from other object files.
    IF Type  $\neq$  1
        If the symbol is defined in this object file...
        LONG NodeID
            Context in which the symbol was defined.
        LONG LineNo
            Line number in the context at which the symbol was defined.
        LONG SectionID
            The ID of the section in which the symbol is defined. If the symbol doesn't belong
            to any specific section (i.e. it's a constant), this field contains -1.
        LONG Value
            The symbol's value. If the symbol belongs to a section, this is the offset within
            that symbol's section.
    ENDC
ENDR

```

Sections

```

REPT NumberOfSections
    STRING Name
        The section's name.
    LONG Size
        The section's size, in bytes.
    BYTE Type
        Bits 0–2 indicate the section's type:
        Value  Meaning
        0      WRAM0
        1      VRAM
        2      ROMX
        3      ROM0
        4      HRAM

```

5 WRAMX
 6 SRAM
 7 OAM

Bit 7 being set means that the section is a "union" (see "Unionized sections" in `rgbasm(5)`). Bit 6 being set means that the section is a "fragment" (see "Section fragments" in `rgbasm(5)`). These two bits are mutually exclusive.

LONG *Address*

Address this section must be placed at. This must either be valid for the section's *Type* (as affected by flags like `-t` or `-d` in `rgblink(1)`), or -1 to indicate that the linker should automatically decide (the section is "floating").

LONG *Bank*

ID of the bank this section must be placed in. This must either be valid for the section's *Type* (with the same caveats as for the *Address*), or -1 to indicate that the linker should automatically decide.

BYTE *Alignment*

How many bits of the section's address should be equal to *AlignOfs*, starting from the least-significant bit.

LONG *AlignOfs*

Alignment offset. Must be strictly less than $1 \ll \text{Alignment}$.

IF *Type* = 2 || *Type* = 3

If the section has ROM type, it contains data.

BYTE *Data*[*Size*]

The section's raw data. Bytes that will be patched over must be present, even though their contents will be overwritten.

LONG *NumberOfPatches*

How many patches must be applied to this section's *Data*.

REPT *NumberOfPatches*

LONG *NodeID*

Context in which the patch was defined.

LONG *LineNo*

Line number in the context at which the patch was defined.

LONG *Offset*

Offset within the section's *Data* at which the patch should be applied. Must not be greater than the section's *Size* minus the patch's size (see *Type* below).

LONG *PCSectionID*

ID of the section in which PC is located. (This is usually the same section within which the patch is applied, except for e.g. LOAD blocks, see "RAM code" in `rgbasm(5)`.)

LONG *PCOffset*

Offset of the PC symbol within the section designated by *PCSectionID*. It is expected that PC points to the instruction's first byte for instruction operands (i.e. `jmp @` must be an infinite loop), and to the patch's first byte otherwise (`db`, `'dw'`, `'dl'`).

BYTE *Type*

Value Meaning

0 Single-byte patch

1 Little-endian two-byte patch

```

        2      Little-endian four-byte patch
        3      Single-byte 'jr' patch; the patch's value will be subtracted to PC
              + 2 (i.e. jr @ must be the infinite loop 18 FE).
LONG RPNSize
        Size of the RPNExpr below.
BYTE RPNExpr[RPNSize]
        The patch's value, encoded as a RPN expression (see RPN
        EXPRESSIONS).
ENDR
ENDC

```

Assertions

```

LONG NumberOfAssertions
        How many assertions this object file contains.
REPT NumberOfAssertions
        Assertions are essentially patches with a message.

LONG NodeID
        Context in which the assertions was defined.
LONG LineNo
        Line number in the context at which the assertion was defined.
LONG Offset
        Unused leftover from the patch structure.
LONG PCSectionID
        ID of the section in which PC is located.
LONG PCOffset
        Offset of the PC symbol within the section designated by PCSectionID.
BYTE Type
        Describes what should happen if the expression evaluates to a non-zero value.
        Value Meaning
        0      Print a warning message, and continue linking normally.
        1      Print an error message, so linking will fail, but allow other assertions to be
              evaluated.
        2      Print a fatal error message, and abort immediately.
LONG RPNSize
        Size of the RPNExpr below.
BYTE RPNExpr[RPNSize]
        The patch's value, encoded as a RPN expression ( see RPN EXPRESSIONS ).
STRING Message
        The message displayed if the expression evaluates to a non-zero value. If empty, a generic
        message is displayed instead.

ENDR

```

RPN EXPRESSIONS

Expressions in the object file are stored as RPN, or “Reverse Polish Notation”, which is a notation that allows computing arbitrary expressions with just a simple stack. For example, the expression `2 5 -` will first push the value “2” to the stack, then “5”. The ‘-’ operator pops two arguments from the stack, subtracts them, and then pushes back the result (“3”) on the stack. A well-formed RPN expression never tries to pop from an empty stack, and leaves exactly one value in it at the end.

RGBDS encodes RPN expressions as an array of **BYTE**s. The first byte encodes either an operator, or a literal, which consumes more **BYTE**s after it:

Value	Meaning
\$00	Addition operator ('+')
\$01	Subtraction operator ('-')
\$02	Multiplication operator ('*')
\$03	Division operator ('/')
\$04	Modulo operator ('%')
\$05	Negation (unary '-')
\$06	Exponent operator ('**')
\$10	Bitwise OR operator (' ')
\$11	Bitwise AND operator ('&')
\$12	Bitwise XOR operator ('^')
\$13	Bitwise complement operator (unary '~')
\$21	Logical AND operator ('&&')
\$22	Logical OR operator (' ')
\$23	Logical complement operator (unary '!')
\$30	Equality operator ('==')
\$31	Non-equality operator ('!=')
\$32	Greater-than operator ('>')
\$33	Less-than operator ('<')
\$34	Greater-than-or-equal operator ('>=')
\$35	Less-than-or-equal operator ('<=')
\$40	Left shift operator ('<<')
\$41	Arithmetic/signed right shift operator ('>>')
\$42	Logical/unsigned right shift operator ('>>>')
\$50	BANK (<i>symbol</i>); followed by the <i>symbol</i> 's LONG ID.
\$51	BANK (<i>section</i>); followed by the <i>section</i> 's STRING name.
\$52	PC's BANK () (i.e. BANK (@)).
\$53	SIZEOF (<i>section</i>); followed by the <i>section</i> 's STRING name.
\$54	STARTOF (<i>section</i>); followed by the <i>section</i> 's STRING name.
\$55	SIZEOF (<i>sectiontype</i>); followed by the <i>sectiontype</i> 's BYTE value (see the <i>Type</i> v alues in Sections).
\$56	STARTOF (<i>sectiontype</i>); followed by the <i>sectiontype</i> 's BYTE value (see the <i>Type</i> v alues in Sections).
\$60	ldh check. Checks if the value is a valid ldh operand (see "Load Instructions" in gbz80(7)), i.e. that it is between either \$00 and \$FF, or \$FF00 and \$FFFF, both inclusive. The value is then ANDed with \$00FF (& \$FF).
\$61	rst check. Checks if the value is a valid rst (see "RST vec" in gbz80(7)) v ector, that is one of \$00, \$08, \$10, \$18, \$20, \$28, \$30, or \$38. The value is then ORed with \$C7 (\$C7).
\$80	Integer literal; followed by the LONG integer.
\$81	A symbol's value; followed by the symbol's LONG ID.

SEE ALSO

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

HISTORY

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