

NAME

mrc – A resource compiler

SYNOPSIS

```
mrc [-o|--output outputfile]
      [--root arg]
      [--resource-prefix arg]
      [--elf-machine arg]
      [--elf-class arg]
      [--elf-data arg]
      [--elf-abi arg]
      [--elf-flags arg]
      file1 [file2...]
```

```
mrc [-h|--help]
```

```
mrc [-v|--version]
```

```
mrc [--header] > msrc.h
```

```
#include "msrc.h"
void foo()
{
  msrc::rsrc data("/alerts/text.txt");
  if (data) {
    msrc::istream is(data);
    ...
  }
}
```

DESCRIPTION

Many applications come with supplementary data. This data is usually stored on disk as regular files. The disadvantage of this procedure is that an application cannot simply be copied to another location or computer and expected to function properly.

Resources are a way to overcome this problem by including all data inside the executable file. The mrc resource compiler can create object files containing both the data and an index. This data can then be accessed from within an application using C++ classes.

OPTIONS

- [-o|--output] file**
Specify the output file, the resulting file will be an object file you can link together with the rest of your object files into an executable file.
- root** The resources are accessed using a path. You can specify the root part of the path using this parameter.
- resource-prefix name**
Use this option to specify another name for the global variables in the data section.
- elf-machine arg**
By default mrc assumes you want to create a resource file for the machine it runs on. But using this option you can create files for other architectures, useful when cross compiling.

The machine flag is used to specify the value of the *e_machine* field in the ELF header.
- elf-class number**
The ELF class to use, should be either *1* for 32-bit objects or *2* for 64-bit objects.

--elf-data number

The ELF data endianness to use, should be either *1* for little-endian (=LSB) objects or *2* for big-endian (=MSB) objects.

--elf-abi number

The ELF OS ABI flag to use, the exact value for this flag should be looked up in *elf.h*. Default is to use the value for the current architecture. (Value of *3* is for Linux, *9* is for FreeBSD).

--elf-flags number

A value to store in the *e_flags* field of the ELF header. This can contain the EABI version for ARM e.g.

--coff type

When this option is specified, a COFF file is created for use on Windows. The argument *type* should be one of **x64**, **x86** or **arm64**.

--header

This option will print a **mrsrc.h** file to **stdout** which you can write to disk and use to access resources. Use with the **--output** option to write to a file instead.

[-v|--verbose]

Print debug output, useful to track where all data ends up in the resource.

--version

Print version number and exit.

[-h|--help]

Print simple help summary and exit.

file [file...]

One or more files to include in the resource file. Directory names can be used as well. All regular files end up in the root of the resource tree, files found in directories end up in directories in the resource tree. The following rules apply:

Regular files are added in the root of the resource tree using their proper file name.

If the file name refers to a directory, the directory is traversed recursively and all files are added. If the file name ends with a forward slash (*/*) files are added to the root. If the file does not end with a slash, the name of the directory will be placed in the root and all contained files will be placed beneath this.

EXAMPLES

Here's a list of usage cases.

```
mrc -o x.o my-resource.txt my-image.png
```

Will create a resource file containing two resources accessible using the path `"/my-resource.txt"` and `"/my-image.png"` respectively.

```
mrc -o x.o img/my-image.png
```

Will create a resource file containing a single resource accessible using the path `"/my-image.png"`.

```
mrc -o x.o img/
```

Assuming there are two images in the directory `img` called `my-image-1.png` and `my-image-2.png`, the resource file will contain them both accessible under the name `"/my-image-1.png"` and `"/my-image-1.png"`.

```
mrc -o x.o img
```

Same as the previous, but note there's no trailing slash, the resource file will contain both images but they are now accessible under the name `"/img/my-image-1.png"` and `"/img/my-image-1.png"`.

Use the verbose flag (**--verbose**) to track what ends up where.

DETAILS

The way this works is that mrc first collects all data from the files specified, including the files found in specified directories. An simple index is created to allow hierarchical access to the data. The data is then flattened into three data structures and these are written to the **.data** section of the object file. The three data blobs are then made available as globals in your application with the names **gResourceIndex**, **gResourceName** and **gResourceData**. You can specify the prefix part of this variable with the `-fB--resource-prefix` option.

The index entries have the following format:

```
struct rsrc_imp
{
    unsigned int m_next; // index of the next sibling entry
    unsigned int m_child; // index of the first child entry
    unsigned int m_name; // offset of the name for this entry
    unsigned int m_size; // data size for this entry
    unsigned int m_data; // offset of the data for this entry
};
```

The classes in the **mrsrc.h** file are contained in the namespace **mrsrc**. The available classes are

mrsrc::rsrc

This is the basic class to access data. It has a constructor that takes a path to a resource. Data can be accessed using the **data** method and the size of the data is available via the **size** method. If the resource was not found, **data** will return **nullptr** and **size** will return zero. You can also use **operator bool** to check for valid data.

mrsrc::streambuf

This class is derived from **std::streambuf**. It can take both a **mrsrc::rsrc** or a path as constructor parameter.

mrsrc::istream

This class is derived from **std::istream**. It can take both a **mrsrc::rsrc** or a path as constructor parameter.

BUGS

This application can only generate ELF formatted object files on machines that have an **<elf.h>** header file installed.

Only a single resource entry can be generated and there's no way to merge or manipulate resource files yet.