VM LABS

VM Labs Makefile Fragments

Version 0.06

Tuesday, May 8, 2001
17:17:33

VM Labs, Inc.
520 San Antonio Rd.
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Revision	***Date***	***Who***	***What***
0.01	September 20, 2000	Christopher Heiny	Created document.
0.02	October 5, 2000	Christopher Heiny	Added `OBJDIR`, `MAKE_VERBOSITY`.
0.03	January 15, 2001	Christopher Heiny	Corrected typos. Updated `clean`/`clobber` target info. Added description of `vml_make_clean.mk`. Updated `OBJDIR` description.
0.04	January 16, 2001	Lisa Reeber	Added description of new definitions in `vml_make.mk`
0.05	January 19, 2001	Christopher Heiny	Added descriptions for `SDK_VERSION`, `SDK_RELEASE_INFO`, `no-target` and `sdkversion`.
	January 22, 2001	Christopher Heiny	Added sample code for using `SDK_RELEASE_INFO`.
0.06	February 16, 2001	Christopher Heiny	Added `vml_make_os.mk`

Table of Contents

1- Introduction

The VM Labs Makefile fragments provide templates for uniform setting of make macros and targets across the various software projects within VM Labs. They are also used in the SDK code samples, in order to reduce the problems that can occur when transporting makefiles into strange environments.

At this time, the VM Labs Makefile fragments are under development. Although the targets, macros and other behaviors they define are reasonably stable as released, they will change in the future. Every effort will be made to ensure compatibility with previous versions, but this may not always be possible.

1.1- Background

The reader of this document is assumed to be familiar with software development concepts in general, including a reasonable familiarity with software development (in particular the writing and use of makefiles) under either the Windows or Linux environment.

Two useful references that you may wish to have are:

- *Managing Projects with make*, Andrew Oram and Steve Talbot, O'Reilly & Associates, Inc. 1993 edition.
- *Gnu Make, A Program For Directing Recompilation*, Richard M. Stallman and Roland McGrath, Free Software Foundation, Inc. 1995 edition 0.43 for gmake 3.73 beta.

The second of these documents is included in the VM Labs SDK in `doc/gmake.pdf`.

1.2- Notation Conventions

The following font conventions are used in this document:

Font	***Used for***
Times New Roman	Main body text.
`Courier New`	Filenames, URLs, e-mail addresses, and so on.
`Lucida`	C, C++, Llama and shell source code examples.
`Courier New`; **`Courier New Bold`**	prompts and printouts; **user input**
italics *italics* ***italics***	Placeholders for user supplied values.

1.3- Terminology

In the most recent editions of the gmake documentation, the traditional terminology of "dependency" has been replaced by "prerequisite". We have attempted to update this document to conform with the new nomenclature, but may not have caught all occurrences.

On the other hand, traditional VM Labs practice refers the files containing a description of the prerequisites for a given file as "dependency" files, and assigns the extension `.d` to them. The nomenclature is retained.

1.4- URLS

When referencing a website, the URL name is given in the text as a hyperlink. Since it's difficult to hyperlink from printed text, though, each URL is expanded as a footnote.

1.5- Support

To report problems with the makefile fragments or this document, or to suggest changes, enhancements, or improvements, please contact the VM Labs SDK support team at: `sdk-release@vmlabs.com`.

2- Using the Makefile Fragments

There are two parts to the VM Labs Makefile fragments.

- `vml_make.mk` – this file should go at the start of your `Makefile`, preferably before any other macros or includes
- `vml_make_targets.mk` – this file should go at the end of your `Makefile`, preferably after all other macros, includes and targets.

Both of these files are found in the `$(VMLABS)/util/` directory.[1] They are intended for use with Gnu make version 3.79.1 and higher. This version of `gmake` is included in the current SDK release.

An additional fragment, `vml_make_os.mk`, is included by `vml_make.mk`. Another additional fragment, `vml_make_clean.mk`, is included automatically by `vml_make_targets.mk`. It is provided separately because the operations they implement are also useful outside the context of their parent files. See the section on the related parent makefile fragment for more details on each of these.

2.1- Platforms supported

The VM Labs Makefile fragments are officially supported on the following OS'es:

- Linux
- Windows 98 with the MKS toolkit
- Windows NT 4.0, service pack 4 or higher
- Windows 2000

All Windows installations require the presence of the VM Labs SDK. Windows ME is not supported. Windows 98 is not supported without the MKS toolkit.

The fragments have been tested on the following distributions of Linux:

- RedHat 6.2
- RedHat 7.0
- Suse 7.0

2.2- `vml_make.mk`

`vml_make.mk` is intended to be included at the start of your Makefile. It defines tools, switches, and utilities that may be used in all supported environments, as described in the preceding section.

2.2.1- General macro definitions

`vml_make.mk` defines the following general macro definitions. These include:

1 A previous release of the SDK, 0.81Beta, included copies of these files in the `$(VMLABS)/include` directory as well. These copies have been removed – if you are upgrading from SDK 0.81Beta, you will probably want to check your existing references and correct them as needed.

Macro	Definition	
BUILDHOST	Specifies the platform on which make/gmake has been invoked. Possible values are:	
	LINUX	All flavors of Linux.
	WIN98	Windows 98
	WINNT	Windows NT 4.0, Windows 2000

Note that if you want just the `BUILDHOST` definition, you may include the file `vml_make_os.mk`.

2.2.2- Useful tools and shell commands

The following definitions are provided by `vml_make.mk`

definition	purpose
MKDIR	shell command to create a directory
RMDIR	shell command to remove a directory
RM	shell command to remove files
RMEX	shell command to remove a directory
SEP	filename separator
ISEP	filename separator
CLS	shell command to clear the terminal screen
CP	shell command to copy files and directories
MV	shell command to move files and directories
ECHO	shell command that writes arguments to the standard output.
AWK	programming language used to manipulate text files
SED	stream editor used to manipulate text files
WAITKEY	waits for the user to press return before continuing

Additional shell commands will be added with future revisions.

2.2.3- Macros for standard make tools

The following definitions are provided for standard make tools.

Definition	Defined as	Purpose
AS	llama	Program for doing assembly.
CC	mgcc	Program for compiling C programs.
AR	vmar	Archive-maintaining program.
LD	vmld	Linker program (when not using `$(CC)`).

Additional tools (for example, for C++ compilation) will be added in future revisions.

2.2.4- Other macros

Two additional macros are defined to provide information about the VM Labs SDK that you are using.

definition	***purpose***
`SDK_VERSION`	The current SDK version number. For example:`0.85`, or `1.12.02`.
`SDK_RELEASE_INFO`	A short descriptive string for the current SDK release, including other information in addition to the version number. For example: `VMLabs SDK 0.86 (Game Developer)`

The exact format of the `SDK_RELEASE_INFO` string may change in the future. However, it is likely to have spaces, parentheses, and other problematic characters in it. To pass this into a C program and have it appear as a string (such as `"VMLabs SDK 0.86"`) to the compiler, you will need to do some rather strange quotes in your make file. For example,

```
-DFOO= "\"$(SDK_RELEASE_INFO)\""
```

For your convenience, there is pair of predefined macros, `QUOTED_VERSION` and `QUOTED_RELEASE_INFO` that you can use. Adding these to the `LOCAL_CFLAGS` definition will cause `SDK_VERSION` and `SDK_RELEASE_INFO`, respectively, to be defined as quoted strings to the C preprocessor.

2.3- `vml_make_targets.mk`

`vml_make_targets.mk` is intended to go at the end of your Makefile. It defines compilation rules for a variety of file types, default compilation flags for the standard tools, and several commonly used targets.

Rules are defined for the following actions:

- make a `.o` file from a `.c` or `.cc` file, using the C compiler
- make a `.o` file from a `.cpp` file, using the C++ compiler
- make a `.o` file from a `.s` file, using the assembler
- make a `.d` file from a `.c` or `.s` file

These rules support dependency file and object file redirection using the `OBJDIR` macro. See below for more details.

Additionally, procedures are defined for making `.cof` and `.a` files (see below).

Finally, the following targets are provided:

- `clean` – for removing output files;
- `clobber` – for stomping your directory into oblivion.

These targets are described in greater detail below.

2.3.1- Compilation Flags and Options

`vml_make_targets.mk` recognizes three kinds of flags for its build rules:

- Local Flags
- VM Labs Flags
- Default Flags

The flags appear on the command line in this order: Local Flags, VM Labs Flags, and finally Default Flags.

Additionally, `vml_make_targets.mk` recognizes two include paths:

- Local include path
- VM Labs include path

These definitions appear on the command line in this order: Local include path first, followed by VM Labs include path. This is important in processing include files and library archives.

The overall ordering of flags on the command line is:

1. Local flags
2. Local include path
3. VM Labs flags
4. VM Labs include path
5. Default Flags

2.3.1.1- Default Flags

Compilation rules are provided for C, C++, and Llama assembler. Standard flags for compilation are defined by the following macros:

Macro	***Use***	***default***
`ARFLAGS`	Flags for building archives.	`crs`
`ASFLAGS`	Flags for the Llama assembler.	`-nologo -fcoff -c`
`CFLAGS`	Flags for the C compiler.	`-Wall`
`CPPFLAGS`	Flags for the C preprocessor, used by both C and C++ compilation.	*none*
`CXXFLAGS`	Flags for the C++ compiler.	`-Wall`
`LDFLAGS`	Flags for linking, used by `$(LD)` and `$(CC)`.	`-mrom -mpe3`

You can override these flags by defining your own settings for them in your Makefile. If `vml_make_targets.mk` detects that you have done this, it will not attempt to define these flags. If you wish to specify flags in addition to the defaults, instead of replacing them, you are encouraged to use the `LOCAL_`*`whatever`*`FLAGS`, as described below.

2.3.1.2- Local Flags

You can specify additional flags by using the `LOCAL_`*`whatever`*`FLAGS` macro for that particular tool. `vml_make_targets` understands the following local flags.

Local flags	***Use***
`LOCAL_ARFLAGS`	Local flags for building archives.
`LOCAL_ASFLAGS`	Local flags for the Llama assembler.
`LOCAL_CFLAGS`	Local flags for the C compiler.
`LOCAL_CPPFLAGS`	Local flags for the C preprocessor, used by both C and C++ compilation.
`LOCAL_CXXFLAGS`	Local flags for the C++ compiler.
`LOCAL_LDFLAGS`	Local flags for linking.

2.3.1.3- VM Labs Flags

VM Labs Flags are not implemented at this time.

2.3.1.4- Include Paths

As mentioned above, `vml_make_targets.mk` recognizes both a local and VM Labs include path.

VM Labs include path macro is `VML_INCLUDE_PATH`, and is defined as

```
-I$(VMLABS)/include
```

If the `VMLABS_LOCAL` macro is defined in your makefile or environment, the VM Labs include path have that prepended to it:

```
-I$(VMLABS_LOCAL)/include -I$(VMLABS)/include
```

This is extremely useful if you are preparing local versions of SDK components. If you install your version of the component in `$(VMLABS_LOCAL)/include`, it will be used in preference to the version in the SDK.

It is highly recommended that you structure your build environment to take advantage of this.

The local include path macro, `LOCAL_INCLUDE_PATH`, is available for you to specify include directories other than those specified by the VM Labs include path. It is highly recommended that you use this macro rather than redefining `VML_INCLUDE_PATH`. Since `LOCAL_INCLUDE_PATH` precedes `VML_INCLUDE_PATH` on the command lines, any directories it specifies will be searched before the VM Labs include directories.

2.3.2- Object File Redirection with `OBJDIR`

If you define the `OBJDIR` macro, the `.o` files will be placed in the directory specified by `OBJDIR`. If this macro is not defined, the `.o` files will be placed in the working directory. Note that `OBJDIR` must NOT contain a trailing directory separator. For example:

Valid	***Invalid***
`OBJDIR = C:/proj/foo/obj`	`OBJDIR = C:/proj/foo/obj/`

Valid	***Invalid***
`OBJDIR = ./product-19/obj`	`OBJDIR = ./product-19/obj/`
`OBJDIR = ~/my_objs`	`OBJDIR = ~/my_objs/`

Dependency files (`.d` files) will be redirected in the same manner as the `.o` files.

If `OBJDIR` is present, it will be checked for object and dependency files when evaluating dependencies.

The clean target will look in OBJDIR for the following file extensions:

- `.a`
- `.cof`
- `.d`
- `.dll`
- `.exe`
- `.lib`
- `.o`
- `.obj`

when trying to determine which files to clean from a directory.

If `OBJDIR` is not defined, it defaults to your current working directory (that is, `./`).

2.3.3- Libraries

Libraries used in linking have a set of macro definitions parallel to those used in compilation. These macros are used by the `makecof` procedure (see below)

2.3.3.1- Specifying libraries

`vml_make_targets.mk` recognizes both local and default library macros for linking. These macros are `LOCAL_LIBS` and `LDLIBS`, respectively.

Use the `LOCAL_LIBS` macro to define any libraries you want to be included in your coff file.

Currently, no default libraries are are defined, since many existing makefiles use the `LDLIBS` macro to define libraries for linking. You are encouraged to convert existing makefiles and to author new makefiles to use `LOCAL_LIBS`.

2.3.3.2- Library search paths

`vml_make_targets.mk` recognizes both a local and VM Labs library path.

VM Labs library path macro is `VML_LIB_PATH`, and is defined as

```
-L$(VMLABS)/lib
```

If the `VMLABS_LOCAL` macro is defined in your makefile or environment, the VM Labs library path have that prepended to it:

```
-L$(VMLABS_LOCAL)/lib -L$(VMLABS)/lib
```

This is extremely useful if you are preparing local versions of SDK components. If you install your version of the component in `$(VMLABS_LOCAL)/lib`, it will be used in preference to the version in the SDK. It is highly recommended that you structure your build environment to take advantage of this.

The local library path macro, `LOCAL_LIB_PATH`, is available for you to specify include library directories other than those specified by the VM Labs library path. It is highly recommended that you use this macro rather than redfining `VML_LIB_PATH` or using the `LDLIBS` macro. Since `LOCAL_LIB_PATH` precedes `VML_LIB_PATH` and `LDLIBS` on the command lines, any directories it specifies will be searched before the VM Labs include directories.

2.3.4- Build commands

The following commands are provided for building `.cof` files and `.a` files.

They are provided as commands, because they don't neatly fit into the gmake's idea of pattern based rules. To use these commands, insert one of the following rules in your makefile.

2.3.4.1- Building a `.cof`

The `makecof` procedure is used for building `.cof` files. It's invoked like this:

```
coffname: file file file file...
      $(makecof)
```

For example

```
room.cof: $(OBJS)
      $(makecof)
```

The exact action performed by `makecof` is to build the specified coff file from the list of files it depends on, using the following command:

```
$(CC) $(LOCAL_LDFLAGS) $(LOCAL_LIB_PATH) $(VML_LIB_PATH) $(LDFLAGS) $^ $(LOCAL_LIBS) $(LDLIBS) -o $@
```

2.3.4.2- Building a `.a`

The `makelib` procedure is used for building `.a` files. It's invoked like this:

```
libname: file file file file...
      $(makelib)
```

For example

```
libcut.a: snip.o hack.o chop.o scissors.o chainsaw.o nailtrimmer.o
      $(makelib)
```

The exact action performed by `makelib` is to make an archive file from the list of files it depends on, using the following command

```
$(AR) $(LOCAL_ARFLAGS) $(ARFLAGS) $@ $^
```

2.3.5- Targets

`vml_make_targets.mk` provides the following targets:

- `clean`
- `clobber`
- `no-target`
- `sdkversion`

Despite the fire hazard[2], `vml_make_targets` will do a recursive make for the `clean` and `clobber` targets by invoking

2 See Peter Miller's paper *Recursive Make Considered Harmful*, available at http://www.canb.auug.org.au/~millerp/rmch/recu-make-cons-harm.html

```
        gmake clean
```

or

```
        gmake clobber
```

for each subdirectory defined in the `COMPONENTS` macro. If you don't want to use recursive make in this fashion, do not define the `COMPONENTS` macro.

If you want to just include these two targets themselves, include the `vml_make_clean.mk` makefile fragment. Note that it is **NOT** necessary to do this if you have already include `vml_make_targets.mk`.

The no-target target exists to catch make file configuration errors. In particular, it prints a warning if you have failed to define any targets in your make file.

sdkversion prints information about the version of the VM Labs SDK that you're using.

2.3.5.1- clean

Clean will remove all the `.o`, `.a`, `.cof`, `.exe`, and `.dll` files from the current directory. It will also invoke **`gmake clean`** on all the subdirectories listed in `COMPONENTS` (see above).

You can override `clean`'s actions for a given filetype by setting one or more of the following macros

Macro	*Overrides cleaning for*
`CLEAN_LIBS`	`.a` and `.lib` (library) files
`CLEAN_COFS`	`.cof` files
`CLEAN_EXES`	Windows executables
`CLEAN_OBJS`	`.o` and `.obj` files
`CLEAN_DLLS`	Windows `.dll` files.
`CLEAN_DEPS`	`.d` dependency files.

For example, to specify that only `libglarp.a` should be `clean`ed, you would use the line

```
        CLEAN_LIBS = libglarp.a
```

in your makefile. Specifying an empty value for one of these macros means that `clean` will not attempt to delete that kind of file at all. For example, including the line

```
        CLEAN_DLLS =
```

in your makefile would preserve any `.dll` files in that directory.

Note that these definitions are **NOT** passed into the `COMPONENTS` subdirectories when they are `clean`ed.

2.3.5.1.1- Extending the `clean` target

You can specify additonal files for cleaning using the `LOCAL_CLEAN_FILES` macro. For example, to specify that clean should also delete `.log` files and `.ps` files, you would use the following line:

```
        LOCAL_CLEAN_FILES = *.log *.ps
```

in your makefile.

If you have additional actions that you wish to be performed when clean is invoked, you can define your own local clean target, which will be executed before the main body of the clean target is executed. You specify the name of this target using the `LOCAL_CLEAN` macro. For example, to delete a temporary picture data directory, you would use the following commands:

```
LOCAL_CLEAN = removetemp


removetemp:
	$(RMDIR)	$(MY_WORK)/jpegs/out $(MY_WORK)/gifs/out
```

2.3.5.1.2- Overriding the `clean` target entirely

You can prevent the `clean` target from being defined by `vml_make_targets.mk` by including the following definition in your makefile:

```
NO_DEFAULT_CLEAN = yes
```

In this case, you are expected to define your own `clean` target.

2.3.5.2- clobber

Invoking the `clobber` target will delete the entire contents of the directory from which clobber is invoked, including all subdirectories, and then it will delete the directory itself. If you don't realize that you should use caution when invoking **`gmake clobber`**, you probably should be in some safer occupation, like the turnip census.

2.3.5.3- no-target

The `no-target` target is a utility target that is invoked if you (for some reason) didn't define any targets in your make file. All it does is print the message

```
You have failed to define any targets in your makefile.
```

and exit with status -1.

2.3.5.4- sdkversion

The `sdkversion` target is a utility target that will tell you a little bit about the SDK you are currently using. Typing **`gmake sdkversion`** will return the message

```
info: VMLabs SDK 0.84
version: 0.84
```

and exit with status 0. Note that these are the values of the `SDK_RELEASE_INFO` and `SDK_VERSION` macros, respectively

2.3.6- Controlling Verbosity

By default, the rules and commands defined in `vml_make_targets.mk` produce a verbose output of the commands being executed during construction of the target. You can quiet or silence this output by setting the `MAKE_VERBOSITY` macro to an appropriate value in your makefile. The valid options for `MAKE_VERBOSITY` are quiet, which produces a simple action/filename summary for each command executed, and `silent`, which produces almost no output at all.

`MAKE_VERBOSITY`	*Level of detail*
undefined	**Default:** detailed output of commands executed.

MAKE_VERBOSITY	*Level of detail*
`quiet`	Summary output of commands executed.
`silent`	Almost no output at all.

The summary output of quiet consists of an action/filename pair of the form

```
        action - filename
```

Actions recorded are

Action	*What's going on*
`c`	compiling c module
`c++`	compiling C++ module
`d`	determining dependencies for c, C++, and assembler.
`lib`	building a library
`link`	linking a .cof file
`s`	assembling

Examples of the output for each setting are as follows. In each case, `gmake` was invoked on a simple project with a single source file.

MAKE_VERBOSITY	

undefined	`C:\proj\biosver>`**`gmake`**
	`mgcc -c   -IC:\SDK-Local/include -Wall  -o biosver.o biosver.c`
	`mgcc -LC:SDK-Local/lib -LC:SDK-Beta/lib -mrom -mpe3 biosver.o -o biosver.cof`
	
	`C:\proj\biosver>`

`quiet`	`C:\proj\biosver>`**`gmake`**
	`c - biosver.o`
	`link - biosver.cof`
	
	`C:\proj\biosver>`

`silent`	`C:\proj\biosver>`**`gmake`**
	`C:\proj\biosver>`

Examples shown were executed in the Windows98 environment – other environments will appear similar by not identical.

Note for this SDK release the value of `MAKE_VERBOSITY` is not passed to component makefiles during recursive makes. This behavior may change with the next SDK release.