Title: mk.doc

Document: awfulhak.demon.co.uk:~ftp/pub/Personal/mk/mk.doc

Version: 2.12

Author: Brian Somers <brian@awfulhak.demon.co.uk>

Date: 11 October, 1996

Disclaimer: This document is provided as is, and is in no way guaranteed by the author to fulfil any intended purpose.

Updates: Updates may be obtained from Brian Somers through any of the following methods:

Telephone: Home: 0181 932 4255, Mobile: 0973 908 483
Fax: 0181 932 4259
E-Mail: brian@awfulhak.demon.co.uk

The author will attempt to supply anyone with updates to this document at their request, but no guarantee is made to do so.

Table Of Contents:

1. Document Synopsis

This document describes the **MK** program, detailing where to obtain the sources, executables and documentation, how to build the sources, and how to use the program.

1.1. Intended Reader

The reader should be familiar with normal **C/C++** development and build procedures on a micro or mini computer using a command line compiler and a build tool such as **MAKE**. This normally involves writing or maintaining the source of a large program spanning multiple directories, maintaining the build files and releasing the executable and support files.

Readers not directly involved with the building and maintenance of large projects should refer to section 2 for details of the advantages and disadvantages of using **MK** during project development.

2. Program Features

MK is a multi-user, multi-threading build tool. It is designed as a tool capable of rebuilding large source code hierarchies, guaranteeing consistency and utilising all available **CPU** power.

2.1. Invoking the program

Usually, it is invoked by one or more users by issuing the command

```
mk Module
```

where *Module* represents the **Base File Name** of your system. On invocation, all system dependencies are evaluated, and the specified compiler tools are used to re-build any system components that have been updated since the last build. Many command line switches are available for overriding the default way in which **MK** parses and rebuilds the system. They are discussed under ***Program Synopsis***.

2.2. Module Definition

A *Module* is defined as one or more files sharing the same **Base File Name**. Some such files are specifically recognised by **MK**, others may be used to contain either source or generated information

2.2.1. Recognised File Types

The following file types are understood by **MK**:

1. *The dependency file*. This file has a defaulted **.mk** file name extension. It contains a list of module dependencies (see below), and may contain directives that override the rules specified in the start-up file.
2. *The interface file*. This file has a defaulted **.h** file name extension. It specifies the external module interface.
3. *The source file*. This file has a defaulted **.cpp** file name extension. It contains the source code that will be compiled in order to create the *object* file.
4. *The object file*. This file has a defaulted **.obj** file name extension. It is the target of the compilers build of the *source* file.
5. *The include file*. This file has a defaulted **.inc** file name extension, and is used as a temporary file name during a *source* file compilation. It contains a list of all dependency interface files.
6. *The response file*. This file has a defaulted **.rsp** file name extension, and is used as a temporary file name during both *target* file and *library* file builds. It contains a list of all dependency object and library files.
7. *The library file*. This file has no defaulted file name extension, but is usually specified as **.lib** under **MS-DOS** and **.a** under **UNIX**. It is similar to a *target* file except that it is included in any dependant *target*s response file at link time.
8. *The target file*. This file has a defaulted **.exe** file name extension. It is the target of the linkers build of all dependency object and *library* files. It is never included as a file name inside a *response* file.

If a module contains a library file, it cannot contain either a *target* or a *source* file, and its *interface* file is generated internally by **MK** as a concatenation of all interface dependencies (see the ***Program Description*** section).

Modules are related through the specification of each modules dependencies in its respective **.MK** file.

2.2.2. Dependencies

Module dependencies are specified one per line in the modules **.MK** file. There are two types of module dependency, interface dependencies and code dependencies, relating almost directly to **#include** directives in a modules *header* and *code* files respectively. Each of these dependencies are treated differently, and control how **MK** performs subsequent parsing and how it generates *include* and *response* files when compiling and linking.

MK begins by parsing the module(s) specified on the command line. If the module contains a component **.MK** file, it is parsed for dependencies, otherwise it is assumed that the module has no dependencies. For a given module, interface dependencies are examined before code dependencies. Examination involves parsing the module in the same manner as the module specified on the command line*. When all dependency modules have been parsed, the current module is appended to an internal list. The result is a single list, ordered with the least dependant module first, and the most dependant last. All file names written to *include* and *interface* files are written according to the module order defined in this internal list. A more detailed description of both include and response files can be found in the *Description* section.

It should be noted that the order in which file names appear in *include* and *response* files is entirely dependant on the above rule, and ***not*** on the order in which they are specified in any given **.MK** file. This is because a given module must not be aware if its dependencies' dependencies. It should only know the names of the modules that it is directly dependant on.

A system is a module that contains special build directives and depends on other top-level modules. It is usually specified on the command line when invoking **MK**.

2.2.3. The System Module

A system module is defined as a top level module that depends on each of the systems targets, and may contain an overriding compile or link directive that gathers all of the executable and support files into a single installation location. It is a convenient way for the development team to re-build an entire system in one pass, rather than having to build each part of the system individually.

Further information about modules and their dependencies may be found in the *Program Description* section.

2.3. Performance

If the computer system on which **MK** is invoked is capable of running more than one program at a time†, each such invocation will co-operate with each other invocation so that no work is repeated by each users. This is of no appreciable value on single processor systems, but on multi-processor systems it will allow an almost linear performance increase when re-building a system.

* A recursive algorithm is used by **MK** in order to parse each module.

† Systems such as UNIX™ and NOVELL™ allow many users have access to the same source code hierarchy at one time, thus allowing more that one invocation of MK to run on that system at once.

2.4. System Integrity

MK inhibits the potential inconsistencies in normal systems where the **MAKE** tool is used to maintain source code targets. Rather than requiring the programmer to write a **MAKEFILE**, system dependencies are specified in **.MK files**, and no #include syntax is allowed in the system source code. This limitation forces the programmer to be consistent in specifying their module dependencies, in fact, it guarantees the avoidance of some additional build inconsistencies that can be very difficult to identify:

⇒ Executable linking is done in the correct order, accordant with the modules dependencies..
⇒ Module interface files will be included in an order consistent with the modules dependencies.
⇒ Modules cannot be inter-dependant, even through a contrived route of dependencies, i.e. module A depending on module B depending on module C depending on module A.
⇒ Modules that are not given the correct dependencies will not build.

2.5. Source Code Mapping

MK allows each user on a system to make local changes to the system without affecting the release environment or any other users local environment. This capability is called ***Source Code Mapping***. Each user may take a copy of one or more modules into a local directory hierarchy that maps onto the live source code hierarchy. When building, all resulting intermediate and executable files that will change due to local source code changes are produced in the local directory hierarchy, and all such files that will not change are re-used from the live hierarchy. This allows many development team members to test system alterations prior to updating the actual system source code, without re-building a local copy of the entire system.

2.6. Additional Capabilities

Because of the way in which the system is built using **MK**, many additional features can be provided:

⇒ All source files, object files, executable files and temporary files can be easily listed in a report like style.
⇒ A build script can be produced rather than actually compiling the system.
⇒ All target files can be removed instead of being built.

3. Program Synopsis

As mentioned above, the user may specify many command line switches in order to customise how **MK** builds the system. Switches are prefixed with the - character. If the given switch does not require an argument, it may be followed directly by another switch. If it does require an argument (the -i -m and -p switches), it must be the last switch character of that argument. For example, the following invocations specify the same thing;

```
mk -flcimk.rc Target
mk -flc -i mk.rc Target
mk -fl -imk.rc -c Target
mk -i mk.rc -f -l -c Target
```

All switches must be specified before any targets, and apply to all specified targets.

3.1. Program Usage

The program is invoked as follows:

```
mk [ -i ifile ] [ -m mfile ] [ -p n ]
   [ -cfhklqrsvLMORST ] Module-List
```

If **MK** is invoked with invalid arguments, the above message is output to the Standard Error device[‡], and **MK** terminates. Each switch is described in detail below. If the reader is not familiar with the **MK** program, they should return to this section after reading the program description. Note, **MK** switches are case sensitive.

3.2. Switch Definitions

-c	Compile only the specified modules' object file (if it is out of date). If the **-r** switch (see below) is also specified, all out of date library header files are also built.
-f	Force all targets to be considered as out of date. This option instructs **MK** to rebuild all of the specified target system, irrespective of the times found on any of the source files.
-h	Display a list of all available options, along with a brief description of each. When this switch is specified, all other switches are ignored.
-i ifile	Use **ifile** as the initialisation file. This option overrides all of the other mechanisms for locating a start-up file. Refer to the section 4.1, File Location for further details.
-k	Keep all generated include and response files. Normally, **MK** removes these files as soon as the compiler or loader is complete. This option is useful in conjunction with the **-s** switch.
-l	Load only the specified target module if it is out of date. If the **-r** switch is specified, any out of date dependant library files are also rebuilt. This switch compliments the **-c** switch.
-m mapping	If the given **mapping** option does not contain embedded spaces, it should refer to a file name that contains a list of mapping directory pairs. If it

[‡] The standard error device is used by programs to output program errors, and defaults to the terminal device on file descriptor two. It is possible to re-direct error messages under UNIX™, but not under MS-DOS™ (without a tailored program).

contains one or more embedded spaces, it should specify two directory names, specifying an individual mapping directory pair. For example;

```
c:/this/that/theother
```

refers to a file named **theother** in directory **c:/this/that**, whereas;

```
'f:/server/directory c:/localsrc'
```

refers to a single mapping directory pair of **f:/server/directory** and **c:/localsrc**. Refer to section 5.5, Module Mapping for further details.

-p n Execute up to **n** processes at one time. If possible, **MK** will determine the number of **CPU**s on the computer, and use a value equal to half of this number as its default. If this switch is specified, this detection phase is skipped and **n** is used.

-q Normally, **MK** outputs each command that it executes to **the standard output device**[§]. If this switch is specified, no such information is output.

-r Recurse when encountering target directives. On normal invocation, **MK** will not examine dependencies specified in **.MK** files where there is also a **#target** directive, unless the given module was specified as the command line target. The **-r** switch instructs **MK** to parse **all** dependency modules.

-s Show only. This switch instructs **MK** not to actually execute each command. It can be used with the **-k** switch to generate a script (redirecting standard output) that will build the system. It should not be used with the **-q** switch, as the resulting **MK** invocation would do nothing.

-v This option has two purposes. If it is given without a command line target, it instructs **MK** to output its program version number and exit. If given with a command line target, **MK** will output verbose program information as it executes. This information is *very* verbose, and may slow **Mk**s execution speed.

-L Link modules in reverse order. Normally, **MK** writes all object files into the response file in order of dependency, with the most dependant first and the least dependent last. If this option is specified, the ordering is reversed so that the least dependent module appears first. Refer to section 5.4, Object File Linking for further information.

-M This switch instructs **MK** to show a list of all module names encountered during the parse stage, then exit.

-O This switch instructs **MK** to show a list of all object file names that should be built so that the system can be linked. Modules that have no *source file* will not appear in this list.

-R Remove all out of date targets. This option instructs **MK** to remove out of date targets rather than to built them. It is normally used with the **-f** switch, forcing **MK** to remove all system object files (perhaps prior to a backup).

-S This switch instructs **MK** to show a list of all source file names that constitute the system. It may be used to obtain a list of all file names necessary to build the specified system.

-T This switch instructs **MK** to show a list of all temporary include and response files.

Module-List The list of modules to operate on. Although **MK** is normally invoked with only one module name as an argument, any number of module names may be given. For each module name given, **MK** will parse an build it according to any switches supplied.

[§] The standard output device is used by programs to output progress information, and defaults to the terminal device on file descriptor one. It is possible to re-direct standard output with the > syntax.

4. The Initialisation File

The **MK** program has a default set of rules for building a system. These rules however, will not be sufficient for building any given system, and therefore must be re-defined by the initialisation file.

4.1. File Location

The initialisation file may be specified in one of three ways. Each location method overrides the previous one, but only if **MK** is instructed to use that method.

4.1.1. The Default File

By default, **MK** will search for an initialisation file called **mk.rc**. The directory in which this file is found should be the same as the directory in which the **MK** executable resides. If **MK** was executed using a specific path name (passed to the program as it's first argument... *argv[0]*), that path will be used, otherwise the environment variable **PATH** will be searched, and the first directory in which a file called **mk.rc** is found will be used.

4.1.2. Using the Environment

If an environment variable called **MK_RC** is set in the environment, it should specify the name of the initialisation file, and will override the default initialisation file above.

4.1.3. Using the Command Line

If the **-i** command line switch is used (see section 3.2, Switch Definitions), this overrides both the MK_RC variable and the default initialisation file.

If none of the above mechanisms result in the name of a valid initialisation file, the default rules are assumed. These are sure to be insufficient for building your system.

Once, the initialisation file has been located, its contents are used to override the default rules compiled into **MK**. It is not possible to specify more than one initialisation file, although it is possible to include one initialisation file from another. Inside an initialisation file, blank lines and lines begininning with the ';' character are ignored.

4.2. File Sections

The Initialisation file contains up to three sections, *Definitions*, *Assignments* and *Values*. The section in which a given command is found specifies its meaning. The *Definitions* section allows the user to override the words understood by **MK** as directives in a **.MK** file. The *Values* section allows the user to assign default values to each of the **MK** directives. If a given **.MK** file does not contain any given directive, the value specified here will be used as a default. The *Assignments* section allows the user to assign useful definitions for use in **.MK** fiels.

4.2.1. Definitions

A *Definitions* section begins with the **#define** directive, and continues until another section is begun. Each of the lines in the *Definitions* section are of the following format;

```
SYMBOL = Value
```

where *SYMBOL* is one of the following words:

CODE_EXTENSION	**CODE_NEEDS**
COMPILE	**INCLUDE_LINE**
INCLUDE_LINE_CONT	**ITERFACE_EXTENSION**
INTERFACE_NEEDS	**LOAD**
LOCK_EXTENSION	**OBJECT_EXTENSION**
OBJECT_NEEDS	**RESPONSE_LINE**
RESPONSE_LINE_CONT	**TARGET**

Value must not contain any space characters, and overrides the default name for the given directive in all **.MK** files. When appearing in a **.MK** file, directives are always prefixed with the **#** character. The above *SYMBOL*s and their default names are described in section 4.3, Section Specific Commands.

4.2.2. Values

A *Values* section begins with the **#values** directive, and continues until another section is begun. *Value* definitions are of the format ;

```
SYMBOL = Value
```

where *SYMBOL* is one of the following words:

EXECUTABLE_EXTENSION	**CODE_EXTENSION**
COMPILE	**INCLUDE_EXTENSION**
INCLUDE_LINE	**INCLUDE_LINE_CONT**
INTERFACE_EXTENSION	**LOAD**
LOCK_EXTENSION	**MK_EXTENSION**
OBJECT_EXTENSION	**RESPONSE_EXTENSION**
RESPONSE_LINE	**RESPONSE_LINE_CONT**

Value may contain any characters including space characters, and specifies the default for each module if not overriden in that modules **.MK** file. The above *SYMBOL*s and their default values are described in section 4.3, Section Specific Commands.

4.2.3. Assignments

An *Assignments* section begins with the **#assign** directive, and continues until another section is begun. *Assignments* are of the form

```
DEFINE SYMBOL Name = Value
ECHO Text
SET MAPPING = mapping
SET LOAD = Direction
```

where *DEFINE*, *ECHO*, *SET*, *MAPPING* and *LOAD* are those literal words, *SYMBOL* is any of the symbols specified in section 4.2.2, *Name* may not contain any space characters, and *Value* may contain any characters including space characters.

The define line allows the user to make per-symbol definitions applicable to section 4.2.2, Values. The symbol called *Name* is replaced by *Value* each time it is located in the text of the same *SYMBOL* in that section. This command provides a convenient way to short-hand the Values section.

The *ECHO* command instructs MK to output the given *Text* to the standard error device. No interpretation of the text is done.

The *SET MAPPING* command instructs **MK** to use *mapping* as a mapping directive in the same manner as the **-m** command line switch. Refer to section 5.5, Module Mapping for further information.

The *SET LOAD* command has a *Direction* value of either *FORWARD* or *BACKWARD*, and specifies the order in which object files should be linked. If the **-L** command line switch is used, and the *Direction* value is specified as *FORWARD*, a warning is given to inform the user that this directive overrides the **-L** switch. Refer to section 5.4, Object File Linking for further information.

4.3. Section Specific Commands

The initialisation file contains commands that are dependent on appearing in either the *Definitions* or *Values* section. These are of the form;

```
SYMBOL = Value
```

The following *SYMBOL*s are available, and have the given default directive names, values and meanings. All directives are available for use in a **.MK** file unless otherwise stated. Directives used in the **.MK** file are the same as **#value** section commands except that they only apply to the current module, rather than being used as global defaults. All such values may contain environment variables as described in section 4.5, Environment Variables.

4.3.1. BINARY_NAME

Default Name: binary_name
Default Value: $*.exe

This variable defines the default executable file name. It is the name assumed for a **#target** directive without an argument (see below).

4.3.2. CODE_NAME

Default Name: code_name
Default Value: $*.c

The code extension is the file name that contains the source code.

4.3.3. CODE_NEEDS

Default Name: code_needs
Default Value: None

This directive, when given in a **.MK** file allows the user to specify a module on which the current module code file depends. There can be no default value for this directive, and many such directives are allowed in one **.MK** file. Full information about code dependencies can be found in section 5.1, Code File Dependencies.

4.3.4. COMPILE

Default Name: compile
Default Value: $(CC) -c $#$<

This directive tells **MK** how to compile a source file module. It will be necessary to use the special variables described in section 4.5.1, Special Context Dependent Variables when defining the value for this directive. When this command is actually executed, the module include file will be available for examination. Refer to section 5.3, Source File Compilation for details of how this command is executed.

4.3.5. INCLUDE_LINE

Default Name: XXXXXXX
Default Value:

4.3.6. INCLUDE_LINE_CONT

Default Name:
Default Value:

4.3.7. INTERFACE_NAME

Default Name:
Default Value:

4.3.8. INTERFACE_NEEDS

Default Name:
Default Value:

4.3.9. LOAD

Default Name:
Default Value:

4.3.10. LOCK_EXTENSION

Default Name:
Default Value:

4.3.11. OBJECT_EXTENSION

Default Name:
Default Value:

4.3.12. OBJECT_NEEDS

Default Name:
Default Value:

4.3.13. RESPONSE_LINE

Default Name:
Default Value:

4.3.14. RESPONSE_LINE_CONT

Default Name:
Default Value:

4.3.15. TARGET

Default Name:
Default Value:

4.4. Including Other Initialisation Files

There is an additional command that may appear in any of the three sections, that allows the inclusion of a different file as if that file were a part of the initialisation file. Its format is;

```
#include FileName
```

where *FileName* is the name of the file to be included. If *FileName* is not an absolute path, its location is taken as being relative to the file in which the *#include* directive appears. Care should be taken when including other initialisation files, as the last section directive contained in the included file will be the current section for all lines after the *#include* directive.

4.5. Environment Variables

4.5.1. Special Context Dependent Variables

5. Program Description

5.1. Code File Dependencies

5.2. Interface File Dependencies

5.3. Source File Compilation

5.4. Object File Linking

5.4.1. Libraries

5.4.2. Binaries

5.5. Module Mapping

6. Supported Platforms

7. Support and Updates

request a given format

8. Examples

mapping examples

9. Scenarios

How to ...

10. Appendix A: Building the MK Sources

unpacking sources
building for a new machine
porting considerations

11. Appendix B: Glossary

MAKE
MAKEFILE
C
C++
CPU
Base File Name