GenMake
v1.00

Reference guide.

CONTENTS

INTRODUCTION

GenMAKE is a powerful project management utility for MDOS on your Geneve 9640 Computer.

GenMAKE should be used with any MDOS program which has input files and resulting output files (This type of program is known as a "Translator", examples of which are: Assemblers, Compilers, Text Formatters, Linkers, etc.).

For people working on large programming projects, it can keep track of the relations between the input and output files for various stages of program development. For people working on large text projects, it can keep track of the relations between sections of the document and the formatted results. It is most useful when there are multiple output files which share common input files during their creation.

NOTE: Throughout this manual, the term "object file" refers tothe file output by a translator program. The term "source file" refers to the files read by the translator program. T "Source file" for a Small-C compiler is "C" source code. The "object file" for a Small-C compiler is usually a "source file" for an assembler.

OVERVIEW

GenMAKE uses four different files during the course of its execution, these will be described briefly now.

"MAKE" is the batch (Dis/Var 80) file normally invoked by a user to update the files in the current directory

"$MAKE" is the program file which reads a description of file dependencies provided by the user and creates a batch file of the commands needed to keep the output files up to date. This program is normally executed by the "MAKE" batch file

"!**MAKEFILE"** is a text (Dis/Var 80) file, in the current directory, provided by the user, which contains the description of file dependencies which need to be checked in order to keep the output files up to date.

"!DOIT" is a batch (Dis/Var 80) file, in the current directory, containing the commands which must be executed to keep the output files up to date. This batch file is created by the "$MAKE" program file, and is normally executed by the "MAKE" batch file

GenMAKE also determines the or in which actions must be executed to correctly update all of the output files.

USING GenMAKE

You must first prepare a directory for GenMAKE by creating thedescription file "!MAKEFILE" in the directory with a text editor.

Next, MDOS must be able to find the files "MAKE" and "$MAKE" somewhere in your current command path (set with the "PATH"command in MDOS).

Third, the directory must not be write protected, and it must have enough space to allow creation of the "!DOIT" batch file, as well as any files which result from the execution of “!DOIT” batch file.

Fourth, all source files referenced in the description file should have valid creation and update dates on their directory entries (make sure that your system clock is set correctly).

Lastly, you must "CHDIR" to the directory before invoking $MAKE".

Continuing with the assumption that the previous conditions have been met, GenMAKE is invoked from MDOS with the following command format:

MAKE [rule][,flags] (brackets indicate optional items)

Note that GenMAKE does not require any arguments at all, and is normally invoked simply by typing

MAKE

The optional rule is useful if the description file contains many rules, and you don't want GenMAKE to use the first rule defined in the description file. (This would be true if your project was very large, and you only wanted to update one small portion of the project without causing GenMAKE to waste time by checking files which you know are already up to date).

There are two flags recognized by GenMAKE, these are recognized as either uppercase or lowercase letters.

The "V" (View) flag causes GenMAKE to print, on the your screen, the list of actions which it has determined must be taken to update the output files. With the "V" flag present, GenMAKE will not create the "!DOIT" batch file, and no action will actually be performed. This flag is useful when you are debugging your description file, so that you can see that all of the required actions are present, and that there are no unexpected actions which you have directed GenMAKE to use.

The "F" (Force) flag causes GenMAKE to treat all output files as if they were outdated. GenMAKE writes all of the actions encountered into the "!DOIT" batch file, and all output files will be updated.

CREATING A DESCRIPTION FILE

The desciption file "!MAKEFILE" can be creating using a standard text editor program.

There are four types of lines which the file may contain:

1) comments

these may occur at a any point within the description file

2) macro definitions

macros must be defined on a line earlier than the first reference to the macro.

3) rules

rules consist of an object name followed by a list of source names.

4) actions

an action is text which is written into the "!DOIT" batch file for interpretation by the MDOS batch processor.

The most common form of a description file is:

macro definitions

rule 1

actions 1

rule 2

actions 2

In general, you would create one rule for each object file which is a part of your project.

COMMENTS

Comment lines must begin with an asterisk "*" or a hash mark"#", all remaining characters on the line are ignored.

MACROS

A macro definition is one line long, and has the following form:

macro_name=string

The macro name can contain from 1 to 10 printable ascii characters, and must be immediately followed by the equals signI"=" (no spaces are allowed before the "=" equals sign). If themacro name is longer than 10 characters, the name will be truncated to the first 10 characters, and a warning will be issued.

The macro definition consists of all characters following the"=" equals sign, up to the end of the line. (Note that most text editor automatically remove the trailing spaces from lines as the lines are saved to the file. For this reason, you can't define a macro with trailing spaces with such an editor).

A macro can be used on any subsequent line in the description file by use of the following sequence of characters:

$(macro_name)

The sequence of characters given above is replaced by the string defined for the macro, until such substitutions would cause the length of the current input line to exceed 255 characters.

You may redefine a macro simply by defining it again, the original definition is "forgotten".

You may use a macro:

In a rule.

In an action.

In another macro definition.

RULES

A rule can be several lines long, and has one of the following forms:

rule_name: **rule_names**

and

rule_name: **rule_names **

rule_names

The backslash "\" indicates that the list of source files is to be continued on the next line of input. Placing the backslash "\" as the last character of any line with a "rule_names" will cause the "rule_names" to be continued on the next non-comment line in the file (intervening comment lines are ignored).

The first rule defined in the description file is usually the default rule. The default rule can be changed by specifying the name of an alternate default rule on the command line used to invoke GenMAKE.

IMPORTANT: GenMAKE only evaluates rules on which the default depends. Rules defined the description file are ignored if the default rule does not in some fashion depend on them. (For advanced users: the default rule is the root of a tree of rules, rules in the description file which aren't part of the tree are not evaluated).

The "rule_name" is any string which would be valid as an MDOS file name. As in MDOS, any characters in the name which are special to MDOS must be surrounded by double-quote marks. The "rule_name" must be followed immediately by the colon ":" character, with no intervening spaces.

A "rule_name" is the name of another rule in the description file, or the name of an MDOS file. A rule can only be defined once in the description file, and any attempt to redefine a rule results in a warning message from GenMAKE.

GenMake looks at the directory entries for each file specified in a rule to get the last date that data was written to the file. The rule can be "active" under three conditions. The first (and usual) condition is that MDOS file with the same name as one of the "rule_names" following the colon ":" has been written to more recently than the MDOS file with the same name as the "rule_name" before the colon ":" The second condition occurs when there is no file already in existence with the same name as the "rule_name" before the colon ":" The third condition which causes a rule to be "active" is specification of the "F" flag on the command line.

Note for advanced readers: the actions following a rule do not necessarily need to create/update a file of the same name as the rule. This flexibility allows you to define a complex rule by breaking it into several simpler rules, without creating files for the simpler rules. These simpler rules do not need to be followed by a list of actions.

ACTIONS

Any line in the description file beginning with at least one space is an "action" for the previous rule in the file. Actions can not be processed before the first rule in the file is defined (such an attempt will cause GenMAKE to issue a warning).

The "action" is all characters on a line starting with the first non-blank character. An "action" is any command which is recognized by the MDOS batch command interpreter.

When GenMAKE finds an "active" rule, all actions following the definition of the rule (preceeding the next rule definition, or the end of the file) are copied into the "!DOIT" batch file, for later execution by MDOS.

Only actions following "active" rules are placed into the "!DOIT" batch file. This is where the power of GenMAKE is: execute only the actions necessary to update the objects files which were affected by changes to source files.

PUTTING IT ALL TOGETHER

EXAMPLE #1, Control of an assembler

```
*    Makefile for Assembler project

*    define the object to source dependencies

PROJECTO:     COPY EQUATES SOURCE1 SOURCE2

   * update....a tagged object file in the project

ASM COPY,PROJECTO,!LIST,RCO

* end of example #1
```

When you invoke "MAKE", the assembler will be executed to create a new object file if any of the source files had been changed since the old object file had been changed.

EXAMPLE #2, control of a linker

```
*    Makefile for Linker

*   define the program image to tagged object file dependencies

PROGRAM:      LINKERC OBJ1 OBJ  LIB1 LIB2

*   update a program file in the project

        LINK LINKERC

* end of example #2
```

Note that the control for the linker has been included in the list of file dependencies. It is needed because the contents of the program image file usually depend on commands executed in the linker control file. Also note that the libraries referenced from the linker have been included in the dependencies. In general, ANY file read by a translator program should be included in the list of file dependencies.

EXAMPLE #3, control of a compiler

```
*   Makefile for Compiler project
*    define the object to source dependencies
PROJECTO:     SOURCE1  INCLUDE1  INCLUDE2
    * update a tagged object file in the project
    COMPILE SOURCE1,PROJECT,flags
* end of example #3
```

EXAMPLE #4, set of rules for assembling

```
*   Makefile for complete assembler project
*   The program file is the last file to be created, so define it's rule first
PROGRAM:     LINKERC  OBJ1  OBJ2  LIB1  LIB2
    *  Update a program file in the project
    LINK LINKERC
* Now for object files
OBJ1:COPY1 EQUATES SOURCE1
        ASM COPY1,OBJ1,!LIST1,RCO
OBJ2:COPY2 EQUATES SOURCE2
        ASM COPY2,OBJ2,!LIST2,RCO
* End of example #4
```

Note that all three rules are active if the file "EQUATES" has been altered. Both assembler source files will be assembled then linked. If only the file "SOURCE2" has been altered, only the actions for "OBJ2" and "PROGRAM" would be executed.

EXAMPLE #5, set of rules for compiling Small-C style

* Makefile for complete compiler project

* The program file is the last file to be created, so define it's rule first

```
PROGRAM:        LINKERC  OBJ1  LIB1  LIB2
```

* Update a program file in the project

```
        LINK LINKERC
```

* Now for object files

```
OBJ1:C_OUTPUT

        ASM C_OUTPUT,OBJ1,!LIST1,RCO
```

* Now for "C" source code

```
C_OUTPUT:       C_SOURCE INCLUDE1 INCLUDE2

        SMALLC   C_SOURCE,C_OUTPUT,flags
```

* End of Example #5

EXAMPLE #6, set of rules for compiling Small-C style

* Better Makefile for complete compiler project

* The program file is the last file to be created, so define it's rule first

```
PROGRAM:  LINKERC  C_SOURCE  INCLUDE1  INCLUDE2  LIB1  LIB2
```

* Update a program file in the project

```
   SMALLC    C_SOURCE,C_OUTPUT,flags

   ASM  C_OUTPUT,OBJ1,!LIST1,RCO

   LINK LINKERC

   DEL OBJ1

  DEL  C_OUTPUT
```

*End of example #6

Note that all of the actions are performed as the result of one rule, and that all temporary file are deleted.