Dynamic Macro

A program for creating structured text in GNU Emacs

Wayne Mesard

November 1991, Version 2.0

Administrivia

Questions, problems, suggestions should be sent to ‘`WMesard@Oracle.com`’. Enhancement requests are always welcome. Be sure to mention the version of Dmacro that you’re using.

The author would like to thank Jan-Erik Strömquist and Dean Norris for their numerous suggestions and bug reports on the beta versions of this release; and Aamod Sane for converting this manual to Texinfo format.

Overview

Dynamic Macro is a program for creating structured text in Emacs. It can significantly reduce typing time and increase the formatting consistency of source code and other structured text. It allows users to easily construct and use dynamic macros for complex text such as time stamps, comments and common program blocks.

There are three levels of interaction with Dmacro:

- DM users – people who use existing Dmacro files when writing source code or some other document.
- DM builders – people who create new Dmacro files or modify old ones. Requires minimal knowledge of Elisp (e.g., what a symbol is; how to customize a `.emacs` file).
- DM function builders – people who create new functions. Requires proficiency in Elisp (e.g., how to invoke and define Elisp functions).

This document covers each of these levels more or less in sequence. So if you find yourself getting bored and/or confused, you've probably read far enough.

1 Dmacros for C — A tutorial

This section is a step-by-step illustration of how a user interacts with Dmacro. Most of Dmacro's basic features will be touched on. Familiarity with Emacs is expected. The example uses a collection of macros built for writing C programs. Familiarity with C is not required for this tutorial, but it will probably help.

If Dmacro has already been properly installed at your site (see Chapter 3 [Installation], page 10), load it by typing: *`M-x load-library`* RET *`dm-c`* RET If this generates an error you must do it the hard way, using the `load-file` command to load the file `dmacro.el` and `dm-c.el`. Type:

```
M-x load-file RET mydirectory/dmacro.el RET
M-x load-file RET mydirectory/dm-c.el RET
```

where *mydirectory* is the name of the directory where you unpacked Dmacro. If either of these produces an error message, contact your system administrator or local Emacs guru for help.

Now let's begin, the demo. Since the macros we're working with are for writing C programs, we need an empty C-mode buffer to work with. Type:

```
C-x C-f ~/dmacro-demo.c RET
```

But wait a minute! Even though it's a new file, it contains a comment block like this:

```
/* Copyright (c) 1991 by A BIG Corporation.  All Rights Reserved */

/***
   NAME
     dmacro-demo
   PURPOSE

   NOTES

   HISTORY
     wmesard - Dec 24, 1991: Created.
***/
```

What happened was that the `masthead` macro automatically got inserted when you created the file. As you can see, `masthead` contains standard information that might go at the top of any C file. Notice that it contains the current year and date as well as your user id. It also contains the name of the current file, `dmacro-demo`. The cursor is positioned on the line below the '`PURPOSE`' header so that you may immediately begin typing to fill in the rest of the comment block.

Automatic macro insertion is just the beginning; you can also insert macros explicitly. But first move the cursor below the comment block, to the very end of the buffer:

```
M->
```

Now type:

```
C-c d
```

This invokes the command, `insert-dmacro`. Notice that you are being prompted for the name of a dynamic macro in the minibuffer. Type:

```
m
```

followed by a question mark:

?

In the '`*Completions*`' buffer, Dmacro displays a list of all macros that have names beginning with '`m`':

```
Possible completions are:
main    mal
masthead
```

Type question mark a second time:

?

Now the list in the '`*Completions*`' buffer has expanded so that each macro name is accompanied by a brief description:

```
Possible completions are:

main:     an empty main() function with args
mal:      call to malloc (prompts for var type)
masthead: comment block for the top of a .c file
```

This two-level help feature can be used whenever you are being prompted for a macro name. (Feel free to use it as we progress through this tutorial, even though we won't mention it again until the very end. In this case we want `main`, **so finish typing the word and hit `RET`.**

The `main` macro gets inserted into the buffer. The cursor is now positioned in the [empty] body of the function, ready for you to begin typing:

```
main(argv, argc)
char **argv;
int argc;
{

}
```

Let's insert a printf statement. For this we use the `p` macro. Type:

C-c d p `RET`

The following is inserted with the cursor at the start of the string:

```
(void) printf("\n");
```

Finish the command by typing some text like '`Hello world`'.

Now, let's make the program a little more interesting. **Position the cursor after the left curly brace ('`{`').** Add a variable declaration before the printf command:

`Linefeed` *int mycounter;*

Now let's try a `for` loop. **Move the cursor so that it's on the right curly brace ('`}`').** and type:

C-c d ifor `RET.`

The skeleton of a for statement is inserted:

```
for (<var> = 0; <var> < ; ++<var>)
{

}
```

and in the minibuffer, Dmacro is prompting you for the name of the increment variable to use. Enter the name of the variable you just declared:

```
mycounter RET
```

Notice that the placeholders are replaced by the variable name and the entire statement is indented properly:

```
  for (mycounter = 0; mycounter < ; ++mycounter)
  {

  }
```

And, as usual, the cursor is positioned at the point where you're mostly likely to type next. **Type a** *5* to complete the upper bounds condition of the loop.

In addition to setting the point in the right place, the `ifor` macro sets a mark in its body. (In fact, this is true of many of the macros defined in `dm-c`.) So instead of cursoring down to fill in the body, you can simply type:

```
C-x C-x
```

to run the Emacs command `exchange-point-and-mark`. ('`Point`' is Emacs terminology for the cursor. See Section "The Mark and the Region" in *GNU Emacs Manual*, for more information about marks.)

Using the `p` macro as before, create the following line:

```
(void) printf("value is %d\n", mycounter);
```

Suppose we decide that we want to print something else when the variable is equal to three. That means we want to wrap the printf command in a conditional statement. Type:

```
C-u C-c l
```

(Note that that last character is an '`L`' not the number '`1`'). At the familiar '`Dmacro:`' prompt, type:

```
ife RET
```

The for loop now looks like:

```
if ()
  (void) printf("value is %d\n", mycounter);
else
```

Okay, a lot just happened. Let's look at it again in slow motion. We'll get to that *C-u* in a minute. First let's look at the *C-c l*. This key sequence invokes the command `dmacro-wrap-line`. This is similar to `insert-dmacro` (*C-c d*), but instead of merely inserting the macro, it *wraps* the current line inside of the macro. If you look at the buffer, that's exactly what happened: the old printf statement got stuck in the middle of the if/else macro.

You may be wondering how Dmacro decides where to stick the text. As we've seen, many macros leave the cursor strategically positioned after they're inserted. Normally, `dmacro-wrap-line` sticks the old text at this point. However, in the current example, the `ife` macro leaves the cursor in the parenthesis immediately after the word `if`.

So how did the `printf` wind up on the next line? As we saw with `ifor`, many macros also set marks. `ife` does just that, it sets a mark where the then and else clauses go. And that's where the mysterious C-u comes in. It tells `dmacro-wrap-line` to swap the point and the first mark. See Chapter 2 [Using DM], page 8, for more information about this command, and it's sibling, `dmacro-wrap-region`.

Now back to our tutorial. Type:

```
C-x C-x
```

The cursor jumps back to the mark (which would normally be the point). Enter the condition for the if statement:

```
mycounter != 3
```

Now move the cursor down to the line below the word '`else`' and enter the else clause (using the `p` macro if you like):

```
(void) printf("and now for something completely different\n");
```

Finally, lets add a comment. **Move the cursor up to the beginning of the main() body and create a blank line.** Now put a comment on that line by typing:

```
M-; Tab
```

An empty comment gets created. (This is an Emacs thing, not a Dmacro thing, although it would be easy to create a macro to do this.) Fill it in so that it looks something like:

```
/* Print the value iff it isn't 3 */
```

Since you're a conscientious programmer, you want to initial and date this comment, so others will know who to blame. Type:

```
C-c t
```

Notice that your initials and today's date get inserted so that the line looks something like:

```
/* Print the value iff it isn't 3 -wsm12/24/91.*/
```

Time for some more slow motion playback. Any macro can be bound to a key sequence. This makes it easy to invoke frequently used macros. In this case, `dm-c` has a macro named `dstamp` that it binds to *C-c t*. (The *t* stands for "timestamp".) So another way to do what you just did would be to type:

```
C-c d dstamp RET
```

Let's undo what we just did and do it again the long way. This will also give us a chance to show off another feature of the '`Dmacro:`' prompt. Type:

```
C-x u
```

(or which ever key the `undo` command is bound to on your system) to undo the last action. Then type:

```
C-c d d
```

but don't hit `RET` yet. Type a question mark:

```
?
```

Notice that several macro names start with the letter '`d`' but only `dstamp` starts with the letters `ds` Wouldn't it be nice if you only had to type enough letters of a macro name to identify it uniquely, rather than typing the entire name? Well in fact, you can. Type:

```
s RET
```

The `dstamp` macro gets inserted (again).

Now let’s take a look at your final product:

```
/* Copyright (c) 1991 by A BIG Corporation.  All Rights Reserved */

/***
   NAME
     dmacro-demo
   PURPOSE
     To learn about Dmacro.
   NOTES

   HISTORY
     wmesard - Dec 24, 1991: Created.
***/

main(argc, argv)
int argc;
char **argv;
{
  int mycounter;

  (void) printf("Hello world\n");
  for (mycounter = 0; mycounter < 5; ++mycounter)
  {
    /* Print the value iff it isn't 3 -wmesard12/24/91. */
    if (mycounter != 3)
      (void) printf("value is %d\n", mycounter);
    else
      (void) printf("and now for something completely different\n");
  }
}
```

You’ve created a runnable (albeit, not very useful) C program with far less wear and tear on your fingertips. Once you get used to Dmacro, you’ll find that it will significantly reduce the amount of time you spend typing. It can also reduce the effort needed to get programs to compile since it reduces the risk of syntax errors.

There are many more macros defined in `dm-c` than the few we covered here. To list them all type:

`C-c d ?`

This collection of macros can be used by any C programmer, but it was based on the coding standards of one particular software house. You are encouraged to extend and customize it to suit your needs.

2 Using DM

This section describes the commands and keys users invoke when inserting macros. See Chapter 5 [dont-bind-my-keys], page 19, for information about changing the default key bindings.

2.1 insert-dmacro

This is the main interface to Dmacro. It is normally bound to `C-c d`. It prompts for a macro name. Question mark (`?`) and `TAB` work as expected during prompting. For example, if you type `ds` followed by question mark at the prompt, a buffer will pop up containing a list of macro names that start with the letters '`ds`'. If you type `?` again it will also display the documentation for those macros, or the actual macro text if the macro is undocumented.

After entering the name, the macro is expanded and inserted in the current buffer.

2.2 dmacro-wrap-line

This is similar to `insert-dmacro` except that it has the effect of sticking the current line in the middle of the to-be-inserted macro. It is normally bound to `C-c l`. For example, consider a macro named `ife` that expands to:

```
if (p)
  m
else
  m
```

where the *p* represents where the point winds up and each *m* indicates a mark. If you type '`abc`' on a line by itself and then invoke `dmacro-wrap-line`, the result will look like:

```
if (abc)

else
```

With a prefix argument the line will be wrapped at a mark instead of point. So in the current example, typing `C-u C-c l` would produce:

```
if ()
  abc
else
```

If a macro has more than one mark, you select which one to wrap at with a prefix argument. So typing `C-u 2 C-c l` would use the second mark, producing:

```
if ()

else
  abc
```

(The observant Emacs user will note that the default value of a prefix argument is 4. So why did `C-u C-c C-l` use the first mark? Because DM knows to use the first mark if the user specifies one that doesn't exist. (So it looked for a fourth mark, didn't find it, and used the first one instead.) Since most macros have, at most, two or three marks, this is a useful shortcut.)

What's actually happening is that the original line is being deleted, then the macro is inserted, then the original text is put back in. This mechanism allows the same macros to be used for both wrapping and insertion. Many macros (such as `ife`, `b`, `iifd`, etc. from `dm-c`) are useful for both operations.

2.3 dmacro-wrap-region

This command, normally bound to *C-c r* is just like `dmacro-wrap-line` except that it operates on all the text between point and mark. For example, to wrap several lines of C code in curly braces using the `b` macro defined in `dm-c.el`; or to apply a macro containing a TEX command to a word or sentence.

Mark-setting commands such as `mark-word` (ESC *@*) and `mark-paragraph` (ESC *h*) are useful for specifying the to-be-wrapped region (see Section "The Mark and the Region" in *GNU Emacs Manual*).

3 Installing Dynamic Macro

This section describes how to make Dmacro a permanent part of your Emacs environment.

1. Put the file `dmacro.el`, and optionally, `dmacro-bld.el` and `dm-c.el` in your Emacs `load-path` if they're not already there. Byte-compile these files after you've copied them over. (See Section "Compiling Libraries" in *GNU Emacs Manual*, for details.)
2. In your `.emacs` file, add a command to load each Dmacro file that you plan to use. For example:

```
(load "dm-c")
```

4 Creating Macros

This section describes two ways of constructing new dynamic macros. The old-fashioned way is to create a Dmacro file by hand. However, this release introduces Dmacro Builder, which allows you to create macros interactively. This section also describes all the built-in Dmacro functions and modifiers. These are the building blocks from which macros are constructed.

Dmacros are stored in Emacs Abbrev tables. An Abbrev table is associated with one or more major modes. For example, macros for Lisp programs would go in the `lisp-mode-abbrev-table`, macros for text would go in the `text-mode-abbrev-table`. If you want a macro to be available all the time, it should be stored in the `global-abbrev-table`. See Section "Abbrevs" in *GNU Emacs Manual*, for more information about the Abbrev facility.

A Dmacro file is a file containing macro definitions (e.g., the `dm-c.el` file which came with the Dmacro distribution). See Chapter 3 [Installation], page 10, for information about making a collection of macros a permanent part of your Emacs environment. To load Dmacro files on a per-session basis use the `load-file` or `load-library` commands.

4.1 Dmacro Builder

There are an awful lot of syntax rules and other nastiness to be aware of when you are building macros by hand (see Section 4.2 [Dmacro Files], page 12). Dmacro Builder is designed to take care of all that for you. See Section 5.1.1 [dont-bind-my-keys], page 19, for information about changing the key bindings described here.

4.1.1 build-dmacro

Invoke this command by typing:

```
M-x build-dmacro
```

to begin defining a new mode-specific macro.

To define a global macro, preface the command with an argument:

```
C-u M-x build-dmacro
```

You will be prompted for the name of the new macro and its documentation string. If the macro already exists, you will be asked if you really want to redefine it. You will then be advised:

```
Build macro. Type C-c C-d to insert directive. ESC C-c when done.
```

Begin typing the text of the new macro. When you're done, position the cursor at the end of the text and type *`M-C-c`*. All the text between the cursor's starting location and its current location will be used as the new macro definition. If any line of the text was indented, the new macro will automatically indent, as well. If you change your mind, you can abort the macro definition by typing *`C-]`*.

4.1.2 dmacro-build-directive

When typing the text of the new macro, you may find that you need something more than plain old hard-coded text. That's where `dmacro-build-directive` comes in. Normally, this is bound to *`C-c C-d`*.

You will be prompted for a function name and, where appropriate, function arguments. After you have supplied the required information, the text that would result from the directive is inserted in the buffer.

For example, if you invoke the `~month` function, the text '`December`' might be inserted. See Section 4.3.1 [Functions], page 14, for a complete list of DM functions.

4.1.3 dmacro-build-modifier

When the cursor is positioned on or immediately after the text from a directive, you may apply one or more modifiers to it. Normally, this command is bound to *C-c C-m*. When you invoke it, you will be prompted as follows:

```
Modifiers: (U)pper (L)ower (C)aps (P)ad (S)ubstring (E)xpression. Or Return
```

Enter one or more of the indicated letters. When you're finished hit `RET`. The prompt will disappear and the modifiers will be applied to the text. See Section 4.3.2 [Modifiers], page 16, for a complete description of the modifiers.

4.1.4 write-dmacro-file

After you have defined your macros, you can create a Dmacro file to store them permanently. Type *M-x write-dmacros* `RET`. You will be prompted for the name of a file.

4.2 Dmacro Files

A Dmacro file is nothing more than an Elisp file containing one or more Dmacro table definitions. Each table definition is a call to the Elisp function `add-dmacros`. This function takes two arguments: an Abbrev table name and a list of macro definitions. Each macro definition consists of a list containing the macro name, the macro text, an optional expansion qualifier and an optional documentation string.

The following expression defines two macros for C-mode:

```
(add-dmacros 'c-mode-abbrev-table
  '(("d"         "#define ")
    ("masthead"  "/* File: ~(file). Copyright (c) ~(year) BIG Corp. */\n"
     nil "Banner for the top of a source code file")
    ))
```

4.2.1 Dmacro Name

This should be an Elisp symbol in double quotes. This is what the user will eventually enter at `insert-dmacro`'s prompt to invoke the macro.

4.2.2 Dmacro Text

The text is a string of characters interspersed with Dmacro directives. Each directive is prefixed by the tilde character ('`~`'). So this macro definition:

```
("test"  "I am ~(user-name), and the time is ~(hour):~(min):~(sec).")
```

might product something like this:

```
I am Wayne Mesard, and the time is 11:33:20.
```

Several functions take arguments. For example, the `insert-file` function takes a file name as an argument. So a macro definition using this function might look like:

```
("test2"  "On ~(day) the file contained:\n ~(insert-file \"myfile.txt\").")
```

Notice that the file name must appear in quotes (because it is a string) and the quotes must be preceded by backslashes since the directive itself appears in a string. (In Lisp syntax, string constants begin and end with double-quotes. '`\"`' stands for a double-quote as part of the string. '`\\`' for a backslash as part of the regexp, '`\t`' for a tab and '`\n`' for a newline.)

4.2.3 Expansion Qualifier

The expansion qualifier may be omitted (as it was in the two examples above). If specified, it's value should be `nil` or `expand` for ordinary macro expansion, or `indent` which means that in addition to expansion, each line of the expanded text will be indented in whatever way is appropriate for the current buffer's mode.

4.2.4 Documentation

The fourth item in a macro definition list is an optional documentation string. See Section 2.1 [insert-dmacro], page 8, for a description of how the documentation is accessed by the user. So a more complete specification of the first example above would be:

```
("test"  "I am ~(user-name), and the time is ~(hour):~(min):~(sec)."
 expand  "User's name and the current time.")
```

4.2.5 Shortcuts

If a directive doesn't have any arguments or modifiers (described below) The parenthesis may not be needed. In this case, the word immediately after the tilde is used as the directive. So the previous example could be rewritten as:

```
("test"  "I am ~(user-name), and the time is ~hour:~min:~sec."
 expand  "User's name and the current time.")
```

Notice that the parenthesis could not be removed from the `~(user-name)` directive, since the function name is two words long.

4.2.6 define-dmacro-table and define-dmacro

There are two other functions which may be used for macro definition. Neither of them are recommended. But if you really want them, here they are. `define-dmacro-table` is just like `add-dmacros`, except that it clears out the table before defining the new macros. Use this function if you want a particular set of macros—and only those macros—defined.

`define-dmacro` is used for defining a single macro. It takes five arguments. The Abbrev table, the new macro name, the macro text, the expansion qualifier and the documentation string.

4.3 Dmacro Directives

At minimum, a Dmacro directive consists of a function name. It may also include arguments and modifiers. This section describes these parts in more detail.

4.3.1 Functions

Dynamic Macro functions are the predefined set of routines on which all Dynamic Macro directives are built. Each function returns a string or nil.

`@`
: Synonym for `~point`. See below.

`~`
: A single tilde. Usage: '`~~`' or '`~(~)`'.

`ampm`
: '`am`' if it's before noon, '`pm`' after noon.

`chron`
: The complete time stamp as a 24 character string. Example: '`Tue Dec 24 22:59:00 1991`'

`date`
: Day of the month as a two digit string (1-31).

`day`
: The three character abbreviation for the day of the week. Example: '`Tue`'.

`eval`
: Takes a single argument, a Lisp form to be evaluated. This may be any valid Elisp form. The result is converted to a string (if it isn't one already). Usage:

```
~(eval (system-name))
~(eval (yow))
~(eval (mapconcat 'identity
                  (directory-files "~") "\n"))
```

The Lisp form must leave the point where it was.

`file`
: File name without directory. Example: '`myfile.txt`'.

`file-dir`
: Directory without file name. Example: '`/home/bbush`'.

`file-ext`
: File name extension. Example: '`txt`'.

`file-long`
: The full name of the file being edited in the current buffer. Example: '`/home/bbush/myfile.txt`'.

`file-name`
: File name without directory or extension. Example: '`myfile`'.

`hour`
: The hour as a two digit string (1-12).

`hour24`
: The hour as a two digit string (0-23).

`if`
: Takes three args. *expression* is a directive. *then* and *else* can be strings (in double quotes) or directives. *else* is optional. If expression returns something other than an empty string or nil, *then* is evaluated and returned. Otherwise *else* is evaluated and returned. Usage:

```
~(if (prompt optional-arg) ",")
~(if (eval (getenv "HOME"))
    (eval (getenv "HOME")) "unknown!")
```

`insert-file`
: Takes a single argument, a string containing the name of a file. Returns the entire contents of that file. mark Tells Dmacro to leave a mark at this position. The user can jump between the point and the current mark via C-x C-x. If a macro contains multiple marks, the user can step through them via ***`C-u C-Space`***

(or *C-u C-@* on terminals which don't handle *C-Space*). This is useful for macros containing complex "fill in the blank" forms. The "mark ring" is one of the nifty unsung features of Emacs. You are urged to use this function freely (and make sure to tell your users about the mark manipulation commands).

`min`
: The minutes as an unpadded two digit string (00-59).

`mon`
: The three character abbreviation of the current month. Example: '`Dec`'.

`month`
: The current month (unabbreviated). Example: '`December`'.

`month-num`
: The two digit number for the current month (1-12). point Tells Dmacro to leave the cursor at this position after the macro is expanded. (Returns nil.)

`prompt`
: A user-specified string. Prompts for the string when the macro is expanded. Takes several arguments, all of which are optional. name is the name of this prompt; it can be any symbol; the default is "your-text". prompt-string is the string to display in the minibuffer at prompt time; the default is name followed by a colon. prompter is the Elisp function to prompt with; the default is read-string; other reasonable choices are functions like read-file-name or read-minibuffer. Any other arguments to ~prompt are passed on to the prompter. Usage:

```
~prompt
~(prompt datatype "Enter datatype: ")
~(prompt file-name "Header file name:
   read-file-name "/usr/include")
```

The prompting arguments are only meaningful the first time that a particular name appears in each macro. They are ignored thereafter (since a particular prompt can appear multiple times in a macro, but it is only prompted for once).

`sec`
: The seconds as an unpadded two digit string (00-59).

`shell`
: Takes a single argument, *command*, a shell command to be run. Returns the result. Usage:

```
~(shell "/usr/games/fortune")
~(shell "ls *.c")
```

Many commands, add a final newline to their output. To suppress the final newline, specify substring modifiers of 0 and -1. For example:

```
~((shell "uptime") 0 -1)
```

`dmacro`
: Takes an argument name, which is a symbol corresponding to another macro. The named macro is expanded and inserted. A second, optional argument, pointP, if non-nil, will cause point to be left where the inner macro puts it. (By default, the outer macro—the one that the user invoked directly—has control of positioning point.) Usage:

```
~(dmacro malloc)
~(dmacro hifdef t)
```

If no macro name is currently defined, name itself is inserted. So the following macro text:

```
"I play ~(guitar)."
```

could produce different results for different people. Someone who plays electric guitar could define a new macro in his/her personal Dmacro file:

```
("guitar"    "electric guitar")
```

Look at the `mal` and `ifmal` macros in dm-c.el for a more practical example. (They use this technique with a macro named `malloc` to allow people to use different malloc functions without having to modify the Dmacro file.

`user-id` The current user's login id. Example: '`bbush`'.

`user-initials`
The current user's initials. Example: '`BB`'.

`user-name`
The current user's name. Example: '`Barbara Bush`'.

`year` The year as a four digit number.

4.3.2 Modifiers

Dmacro modifiers are transformations applied to a directive. This section lists the modifiers available and gives several examples of their usage. Modifiers are applied to a directive using the following format:

```
~((function args...) modifiers...)
```

4.3.2.1 Casification

The three modifiers, `:up`, `:down` and `:cap` will, respectively, convert the text to all upper case, all lower case, or capitalize each word. So to display the current month in all upper case, you would use the directive:

```
~((month) :up)
```

4.3.2.2 Padding

If the text contains any leading spaces, the `:pad` modifier can be used to specify how it should be handled. For example, the `~(hour)` directive always produces a two character string. Before 10 o'clock the first character is a space. To replace the space with a zero, you would say:

```
~((hour) :pad ?0)
```

Notice the question mark '`?`'. The `:pad` token must be followed by the character to use as a pad. The question mark is Emacs' way of saying it is "the character zero" as opposed to "the number zero." `:pad` may also be followed by `nil`. This means don't pad at all. So:

```
~((hour) :pad nil)
```

would return a one character string before 10 o'clock.

4.3.2.3 Substrings

You may only be interested in a portion of the string returned by a directive. To return a part of the string, specify the position of characters in which you are interested. If you specify a negative number, Dmacro counts from the end of the string. For example:

```
~((user-id) 0 2)       ==> wm
```

```
~((user-id) 2)        ==> esard
~((user-id) -5)       ==> esard
```

The behavior is the same as the Elisp function substring, except that the original string is returned if there's an error. For example:

```
~((user-id) 150 200)  ==> wmesard
```

4.3.2.4 Sub-expressions

If you specify the `:sexp` modifier, Dmacro will return sub-expressions instead of a substring of the original text. (An expression is a balanced Elisp expression, i.e., a string, token or list.) For example, to get the user's first name only, you would say:

```
~((user-name) :sexp 0 0)
```

To get the last name only, say:

```
~((user-name) :sexp -1)
```

4.4 Auto-Insertion of Dmacros

The Dmacro package can automatically insert a macro whenever you create a new file. This behavior is controlled by the variable `auto-dmacro-alist`. Its format is similar to Emacs' `auto-mode-alist` (see Section "Choosing Modes" in *The GNU Emacs Manual*). Each element in the list is a dotted pair containing a regular expression describing a filename and a macro name.

For example, if you defined a macro named `masthead` that you want to automatically insert whenever you create a new `.c` or `.h` file, and a macro named `manskeleton` that you want inserted whenever you create a new `.man` file, you would put the following in your `.emacs` file or directly in the file containing the macro definitions:

```
(setq auto-dmacro-alist (append '(("\\.[ch]$" . masthead)
                                   ("\\.man$" . manskeleton))
                                 auto-dmacro-alist))
```

4.5 Binding Dmacros to Keys

Some die-hard Emacs users like to have everything bound to keys. Dmacro supports these weirdos by making it possible to turn macros into Emacs commands using the Elisp function `dmacro-command`. It takes three arguments: *dmacro1*, *dmacro2* and *command-name*. *dmacro2* and *command-name* are optional. The first two are macro names.

`dmacro-command` builds an Emacs command which invokes the macro named by the first argument. If *dmacro2* is specified, it will be inserted when the command is given a prefix argument. If *command-name* is specified, the resulting Emacs command will be given that name (otherwise the command is anonymous). The following examples illustrate how this works.

These lines could be placed in a `.emacs` file, or in the Dmacro file where the particular macros are defined:

```
(define-key c-mode-map "\C-cm" (dmacro-command "mal"))
(global-set-key        "\C-ct" (dmacro-command "dstamp" "dtstamp"))
(define-key c-mode-map "\C-cf" (dmacro-command "for" nil 'c-insert-for))
```

The first command binds the `mal` macro to *`C-c m`* when editing C files. The second example binds the `dstamp` macro to *`C-c d`* and the `dtstamp` macro to *`C-u C-c d`*. The final example binds a macro named `for` to *`C-c f`* and creates a real live Emacs command called `c-insert-for`, suitable for use with *`C-h f`* and *`M-x`*.

5 Customizing DM

This section describes user-settable variables which customize Dmacro's behavior. They can be set using the `setq` function in your `.emacs` file or in a Dmacro file. Alternatively, you can use *`M-x set-variable`* or *`M-x edit-options`* to change their values interactively.

Use *`C-h v`* to find out the current setting of these options. See Section "Examining and Setting Variables" in *GNU Emacs Manual*, for information on these variable manipulation commands.

5.1 Interface Options

5.1.1 dont-bind-my-keys

Dmacro and Dmacro Builder automatically bind certain functions to keys. To prevent this, set `dont-bind-my-keys` to `t` before loading these programs. This is useful if you want to bind the functions to different keys or if you simply don't want Dmacro messing with your key mappings.

By default, this variable is unbound (which tells Dmacro to do the bindings).

5.1.2 dmacro-on-abbrev

As you may have realized by now, Dmacro is an overgrown hack built on top of Emacs' Abbrev Mode. Some people like to use both Abbrev Mode and Dmacro at the same time, but they don't want their macros auto-expanded. If `dmacro-on-abbrev` is `nil`, macros will only be expanded if they were accessed through: `insert-dmacro`, `dmacro-wrap-line` or `dmacro-wrap-region`.

The default value is nil.

5.1.3 dmacro-prompt

One of Dmacro's most important features is its ability get a string from the user at expansion time. There are three modes in for doing this: prompting mode, post-expansion mode and pre-expansion mode.

In prompting mode the user is prompted in the minibuffer for each string. This is the default (and recommended) mode. To select it set `dmacro-prompt` to `t`.

In post-expansion mode, no prompting is done, instead the macro is inserted with placeholders (surrounded by angle-brackets). For example:

```
for (i = <var>; <var> > 0; --<var>)
```

Then to complete the macro, the user types balanced expressions into the buffer (one for each unique placeholder) and invokes the command `dmacro-fill-in-blanks,` which is normally bound to *`C-c f`*. The expressions are deleted and then reinserted at each placeholder. This can be done any time before the next macro is inserted (in other words, Dmacro only remembers the most recent set of placeholders). To select post-expansion mode set dmacro-prompt to nil.

In pre-expansion mode, the user must type the balanced expressions before inserting the macro. The appropriate number of expressions will be deleted from the buffer and inserted into the macro. To select this mode, set `dmacro-prompt` to something other than `t` or `nil`.

Pre- and post-expansion mode are provided for people who hate typing in the minibuffer. Keep in mind, however, that if you forget to type the right number of balanced expressions, Dmacro will blindly use—and delete—whatever it finds in the buffer. Use at your own risk. Also, if the expression is a string, it will be inserted without the quotes. In other words, you must wrap quotes around multi-word entries.

5.2 Expansion Options

5.2.1 dmacro-month-names

This variable contains a list of the names of the 12 months. The `~month` function uses this list. Change these to suit your language or your tastes.

The default value is:

```
("January" "February" "March" "April" "May" "June" "July"
 "August" "September" "October" "November" "December")
```

5.2.2 dmacro-prefix-char

By default, Dmacro uses a tilde (‘`~`’) to mark the start of a directive within the macro text. If, for some reason you want to use another character, set the value of this variable at the top of your Dmacro files. It must be a string containing a single character. This applies to every active Dmacro file, so don’t change it unless you know what you’re doing and have a really good reason for doing it.

5.2.3 dmacro-rank-in-initials

When the `~(user-initials)` function sees a rank (that is, ‘`Jr`’, ‘`Sr`’, ‘`II`’, ‘`III`’, etc.) in a user’s name, it normally ignores it. If this variable is non-nil, it will include it as is. For example, Pope John Paul II would normally have the initials ‘`PJP`’. If he set this variable to non-nil, his initials would change to ‘`PJPII`’.

6 Defining Functions

This section describes how to create new DM functions. If the existing set of functions are not sufficient for your needs, you can build new ones. New function definitions should be placed at the top of the Dmacro file in which they are used.

Dmacro Builder will recognize newly-defined functions. There are actually two types of functions, proper functions and aliases. From the macro builder's point of view they both behave the same, so we tend to just say "function" rather than invent yet another term to describe "functions and aliases. However, the way they are defined is very different.

6.1 def-dmacro-alias

Aliases are synonyms for directives. There are several reasons for defining aliases:

- A few long, complex directives are making your Dmacro file hard to read. If you define aliases for these directives, you can then use the alias within the macro text.
- You use the same complex directive several times and you want to type it once so that it's easier to change later on.
- You simply hate the name of a built-in function. For example, it was mentioned above that the `~(user-name)` function always had to appear in parenthesis because of the dash ('`-`') in its name. You could define an alias called `~username` which would not have this restriction.

The format for defining an alias is:

`(def-dmacro-alias` *`name dmacro-directive`*`)`

Examples:

```
;; The ~@ function is a synonym for ~point. It could have been defined like:
(def-dmacro-alias @ point)
;; The last 2 digits of the year.
(def-dmacro-alias year2  ((year) 2))
;; Prompt for header file name.
(def-dmacro-alias hfileprompt
  (prompt header-file "Header file name:" read-file-name "/usr/include"))
```

6.2 def-dmacro-function

The `~eval` function can be inconvenient if you're macros contain complex or frequently-used Elisp expression. In this case, you may want to create a new Dmacro function. You are responsible for ensuring that your functions always return a string or nil, and that they always leave the point where they found it.

`def-dmacro-function` comes in two flavors:

`(def-dmacro-function` *`macro-name Elisp-function`*`)`

For example, to define a function named `~env` which does the same thing as the Elisp function `getenv`, you would say:

```
(def-dmacro-function env getenv)
```

Since the Elisp function takes one string argument, the new function does, too. So this could be used in macro text as follows: "My terminal type is ~(env \"TERM\"), isn't that interesting?"

The second format is exactly like the Emacs Lisp's `defun`:

```
(def-dmacro-function macro-name (args...) body...)
```

For example, the ~ampm function could have been defined like this:

```
(def-dmacro-function ampm ()
  (if (<= 12 (string-to-int (substring (current-time-string) 11 13)))
      "pm"
    "am"))
```

7 Changes Since Template 1.5

If you're not a user of Template version 1.5, you may skip this section.

The predecessor to Dynamic Macro 2.0 is Template 1.5, released in April 1991. The name was changed because people were confusing it with Template Mode by Mark Ardis, a program which does many of the same things.

Backwards compatibility was maintained wherever it was possible to do so without significantly harming the performance of Dmacro version 2.0.

This section describes some additional procedures to make Dmacro 2.0 behave like Template 1.5.

7.1 Interface

`C-c x` is no longer bound to Emacs' `expand-abbrev` command. This version of DM does its best to hide the fact that it is built on top of the Abbrev facility. It will take a day or so to get used to typing the command and then the macro name instead of the other way 'round. If you really miss the old behavior, add the following two lines to your Emacs init file:

```
(global-set-key "\C-cx" 'expand-abbrev)
(setq dmacro-on-abbrev t)
```

See Chapter 5 [Customization], page 19, for information on `dmacro-on-abbrev`.

7.2 File Format

The DM directive syntax has been drastically changed. The good news is that it takes about five minutes to update a typical Dmacro file by hand. If even that is too much for you, the Dmacro 2.0 distribution contains a file called `dm-compat.el`. Loading this file should enable Dmacro 2.0 to handle Template 1.5 files with the following exceptions:

- The `~s#` directives are only partially supported. The user will be prompted, but with a default prompt string, not the ones supplied in those ridiculous lambda expressions. You are urged to upgrade to the new `~prompt` function.

  Old: `(foo "The user typed ~s0" (lambda () (dmacro-prompt "Type it: ")))`

  New: `(foo "The user typed ~(prompt type-it)")`
- The `~pC` directive is no longer supported. You must now use the `:pad` modifier.
- `~>>` and `~>@` are not supported. You must use the new `~(dmacro)` function.
- Some of the more obscure directives are not defined in dm-compat.el. For example, `~u#` is supported only if # is less than 4. `~fd` is supported, but `~Fd` is not. If your favorite directive is missing, you should be able to add it by using the others as a model.

Note that `~@` and `~~` will work in both versions even without `dm-compat.el`.

8 Glossary

alias
: A symbol corresponding to a shorthand notation for a directive.

directive
: A function or alias, arguments (if required) and optional modifiers. Examples:

```
ampm
(prompt data-type)
((user-id) :cap)
```

Dmacro file
: An Elisp file containing macro definitions.

Elisp
: The Emacs Lisp programming language. This is the language programmers use to customize and extend GNU Emacs. Your `.emacs` file contains Elisp code. Dmacro is written in Elisp.

expression
balanced expression
: An Elisp S-expression. This can be a symbol, list, string, etc. Examples:

```
foobar2
my-dog-has-fleas
underscores_count
(and lists (and lists of lists))
"and strings, of course"
```

function
: An actual piece of Emacs Lisp code. In the context of DM, this is something defined by `def-dmacro-function` and must return a string or nil.

modifier
: A transformation applied to the string returned by a directive. Modifiers are used to change the case of a string, affect the left-padding of a string or extract a portion of the string.

Key Index

Function and Variable Index

Concept Index

Table of Contents