Extending XEmacs using C and C ++

Version 1.0, September 1998

J. Kean Johnston

Copyright © 1998 J. Kean Johnston.

Version 1.0
September, 1998.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU General Public License" is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled "GNU General Public License" may be included in a translation approved by the Free Software Foundation instead of in the original English.

1 Introduction

XEmacs is a powerful, extensible editor. The traditional way of extending the functionality of XEmacs is to use its built-in Lisp language (called Emacs Lisp, or Elisp for short). However, while Elisp is a full programming language and capable of extending XEmacs in more ways than you can imagine, it does have its short-comings.

Firstly, Elisp is an interpreted language, and this has serious speed implications. Like all other interpreted languages (like Java), Elisp is often suitable only for certain types of application or extension. So although Elisp is a general purpose language, and very high level, there are times when it is desirable to descend to a lower level compiled language for speed purposes.

Secondly, Elisp (or Lisp in general) is not a very common language any more, except for certain circles in the computer industry. C is a far more commonly known language, and because it is compiled, more suited to a wider range of applications, especially those that require low level access to a system or need to be as quick as possible.

This manual describes a new way of extending XEmacs, by using dynamically loadable modules (also known as dynamically loadable libraries (DLLs), dynamic shared objects (DSOs) or just simply shared objects), which can be written in C or C++ and loaded into XEmacs at any time. I sometimes refer to this technology as CEmacs, which is short for C Extensible Emacs

XEmacs modules are configured into and installed with XEmacs by default on all systems that support loading of shared objects. From a users perspective, the internals of XEmacs modules are irrelevant. All a user will ever need to know about shared objects is the name of the shared object when they want to load a given module. From a developers perspective though, a lot more is provided.

- Of primary interest is the `ellcc` program. This program is created during compile time, and is intended to abstract compiler specific characteristics from the developer. This program is called to compile and link all objects that will make up the final shared object, and accepts all common C compiler flags. `ellcc` also sets up the correct environment for compiling modules by enabling any special compiler modes (such as PIC mode), setting the correct include paths for the location of XEmacs internal header files etc. The program will also invoke the linker correctly to created the final shared object which is loaded into XEmacs.
- CEmacs also makes all of the relevant XEmacs internal header files available for module authors to use. This is often required to get data structure definitions and external variable declarations. The header files installed include the module specific header file '`emodules.h`'. Due to the nature of dynamic modules, most of the internals of XEmacs are exposed. See section "Top" in XEmacs Internals Manual, for a more complete discussion on how to extend and understand XEmacs. All of the rules for C modules are discussed there.
- Part of the XEmacs distribution is a set of sample modules. These are not installed when XEmacs is, but remain in the XEmacs source tree. These modules live in the directory '`modules`', which is a sub-directory of the main XEmacs source code directory. Please look at the samples carefully, and maybe even use them as a basis for making your own modules. Most of the concepts required for writing extension modules are covered in the samples.

- Last, but not least is this manual. This can be viewed from within XEmacs, and it can be printed out as well. It is the intention of this document that it will describe everything you need to know about extending XEmacs in C. If you do not find this to be the case, please contact the author(s).

The rest of this document will discuss the actual mechanics of XEmacs modules and work through several of the samples. Please be sure that you have read the XEmacs Internals Manual and understand everything in it. The concepts there apply to all modules. This document may have some overlap, but it is the internals manual which should be considered the final authority. It will also help a great deal to look at the actual XEmacs source code to see how things are done.

2 Anatomy of a Module

Each dynamically loadable XEmacs extension (hereafter referred to as a module) has a certain compulsory format, and must contain several pieces of information and several mandatory functions. This chapter describes the basic layout of a module, and provides a very simple sample. The source for this sample can be found in the file '`modules/simple/sample.c`' in the main XEmacs source code tree.

2.1 Required Header File

Every module must include the file '`<emodules.h>`'. This will include several other XEmacs internal header files, and will set up certain vital macros. One of the most important files included by '`emodules.h`' is the generated '`config.h`' file, which contains all of the required system abstraction macros and definitions. Most modules will probably require some pre-processor conditionals based on constants defined in '`config.h`'. Please read that file to familiarize yourself with the macros defined there.

Depending on exactly what your module will be doing, you will probably need to include one or more of the XEmacs internal header files. When you `#include <emodules.h>`, you will get a few of the most important XEmacs header files included automatically for you. The files included are:

'`lisp.h`' This file contains most of the macros required for declaring Lisp object types, macros for accessing Lisp objects, and global variable declarations.

'`sysdep.h`'
All system dependent declarations and abstraction macros live here. You should never call low level system functions directly. Rather, you should use the abstraction macros provided in this header file.

'`window.h`'
This header file defines the window structures and Lisp types, and provides functions and macros for manipulating multiple XEmacs windows.

'`buffer.h`'
All macros and function declarations for manipulating internal and user visible buffers appear in this file.

'`insdel.h`'
This header provides the information required for performing text insertion and deletion.

'`frame.h`' Provides the required structure, macro and function definitions for manipulating XEmacs frames.

2.2 Required Functions

Every module requires several initialization functions. It is the responsibility of these functions to load in any dependent modules, and to declare all variables and functions which are to be made visible to the XEmacs Lisp reader. Each of these functions performs a very specific task, and they are executed in the correct order by XEmacs. All of these

functions are `void` functions which take no arguments. Here, briefly, are the required module functions. Note that the actual function names do not end with the string `_module`, but rather they end with the abbreviated module name by which the module is known. More on the module name and its importance later. Just bear in mind that the text `_module` in the functions below is simply a place-holder, not an actual function name.

`syms_of_module`
: This required function is responsible for introducing to the Lisp reader all functions that you have defined in your module using `DEFUN()`. Note that *only* functions are declared here, using the `DEFSUBR()` macro. No variables are declared.

`vars_of_module`
: This required function contains calls to macros such as `DEFVAR_LISP()`, `DEFVAR_BOOL()` etc, and its purpose is to declare and initialize all and any variables that your module defines. They syntax for declaring variables is identical to the syntax used for all internal XEmacs source code. If the module is intended to be usable statically linked into XEmacs, the actions of this function are severely restricted. See section "General Coding Rules" in XEmacs Internals Manual. Also see the comments in '`src/emacs.c`' (`main_1`). Modules which perform initializations not permitted by these rules will probably work, but dual-use (dynamic loading and static linking) modules will require very careful, and possibly fragile, coding.

`modules_of_module`
: This optional function should be used to load in any modules which your module depends on. The XEmacs module loading code makes sure that the same module is not loaded twice, so several modules can safely call the module load function for the same module. Only one copy of each module (at a given version) will ever be loaded.

`docs_of_module`
: This is a required function, but not one which you need ever write. This function is created automatically by `ellcc` when the module initialization code is produced. It is required to document all functions and variables declared in your module.

2.3 Required Variables

Not only does a module need to declare the initialization functions mentioned above, it is also required to provide certain variables which the module loading code searches for in order to determine the viability of a module. You are *not* required to provide these variables in your source files. They are automatically set up in the module initialization file by the `ellcc` compiler. These variables are discussed here simply for the sake of completeness.

`emodules_compiler`
: This is a variable of type `long`, and is used to indicate the version of the XEmacs loading technology that was used to produce the module being loaded. This version number is completely unrelated to the XEmacs version number, as a

given module may quite well work regardless of the version of XEmacs that was installed at the time the module was created.

The XEmacs modules version is used to differentiate between major changes in the module loading technology, not versions of XEmacs.

`emodules_name`
: This is a short (typically 10 characters or less) name for the module, and it is used as a suffix for all of the required functions. This is also the name by which the module is recognized when loading dependent modules. The name does not necessarily have to be the same as the physical file name, although keeping the two names in sync is a pretty good idea. The name must not be empty, and it must be a valid part of a C function name. The value of this variable is appended to the function names `syms_of_`, `vars_of_`, `modules_of_` and `docs_of_` to form the actual function names that the module loading code looks for when loading a module.

  This variable is set by the `--mod-name` argument to `ellcc`.

`emodules_version`
: This string variable is used to load specific versions of a module. Rarely will two or more versions of a module be left lying around, but just in case this does happen, this variable can be used to control exactly which module should be loaded. See the Lisp function `load-module` for more details. This variable is set by the `--mod-version` argument to `ellcc`.

`emodules_title`
: This is a string which describes the module, and can contain spaces or other special characters. It is used solely for descriptive purposes, and does not affect the loading of the module. The value is set by the `--mod-title` argument to `ellcc`.

2.4 Loading other Modules

During the loading of a module, it is the responsibility of the function `modules_of_module` to load in any modules which the current module depends on. If the module is stand-alone, and does not depend on other modules, then this function can be left empty or even undeclared. However, if it does have dependencies, it must call `emodules_load`:

```
int emodules_load (const char *module,
                   const char *modname,
                   const char *modver)
```

The first argument module is the name of the actual shared object or DLL. You can omit the '`.so`', '`.ell`' or '`.dll`' extension of you wish. If you do not specify an absolute path name, then the same rules as apply to loading Lisp modules are applied when searching for the module. If the module cannot be found in any of the standard places, and an absolute path name was not specified, `emodules_load` will signal an error and loading of the module will stop.

The second argument (modname) is the module name to load, and must match the contents of the variable emodule_name in the module to be loaded. A mis-match will cause the module load to fail. If this parameter is `NULL` or empty, then no checks are performed against the target module's emodule_name variable.

The last argument, modver, is the desired version of the module to load, and is compared to the target module's emodule_version value. If this parameter is not `NULL` or empty, and the match fails, then the load of the module will fail.

`emodules_load` can be called recursively. If, at any point during the loading of modules a failure is encountered, then all modules that were loaded since the top level call to `emodules_load` will be unloaded. This means that if any child modules fail to load, then their parents will also fail to load. This does not include previous successful calls to `emodules_load` at the top level.

Warning: Modules are *not* loaded with the `RTLD_GLOBAL` flag. The practical upshot is that individual modules do not have access to each other's C symbols. One module cannot make a C function call to a function defined in another module, nor can it read or set a C variable in another module. All interaction between modules must, therefore, take place at the Lisp level. This is by design. Other projects have attempted to use `RTLD_GLOBAL`, only to find that spurious symbol name clashes were the result. Helper functions often have simple names, increasing the probability of such a clash. If you really need to share symbols between modules, create a shared library containing those symbols, and link your modules with that library. Otherwise, interactions between modules must take place via Lisp function calls and Lisp variables accesses.

3 Using ellcc

Before discussing the anatomy of a module in greater detail, you should be aware of the steps required in order to correctly compile and link a module for use within XEmacs. There is little difference between compiling normal C code and compiling a module. In fact, all that changes is the command used to compile the module, and a few extra arguments to the compiler.

XEmacs now ships with a new user utility, called `ellcc`. This is the Emacs Loadable Library C Compiler . This is a wrapper program that will invoke the real C compiler with the correct arguments to compile and link your module. With the exception of a few command line options, this program can be considered a replacement for your C compiler. It accepts all of the same flags and arguments that your C compiler does, so in many cases you can simply set the `make` variable `CC` to `ellcc` and your code will be compiled as an Emacs module rather than a static C object.

`ellcc` has three distinct modes of operation. It can be run in compile, link or initialization mode. These modes are discussed in more detail below. If you want `ellcc` to show the commands it is executing, you can specify the option `--mode=verbose` to `ellcc`. Specifying this option twice will enable certain extra debugging messages to be displayed on the standard output.

3.1 Compile Mode

By default, `ellcc` is in compile mode. This means that it assumes that all of the command line arguments are C compiler arguments, and that you want to compile the specified source file or files. You can force compile mode by specifying the `--mode=compile` argument to `ellcc`.

In this mode, `ellcc` is simply a front-end to the same C compiler that was used to create the XEmacs binary itself. All `ellcc` does in this mode is insert a few extra command line arguments before the arguments you specify to `ellcc` itself. `ellcc` will then invoke the C compiler to compile your module, and will return the same exit codes and messages that your C compiler does.

By far the easiest way to compile modules is to construct a '`Makefile`' as you would for a normal program, and simply insert, at some appropriate place something similar to:

```
CC=ellcc --mode=compile

.c.o:
    $(CC) $(CFLAGS) -c $<
```

After this, all you need to do is provide simple `make` rules for compiling your module source files. Since modules are most useful when they are small and self-contained, most modules will have a single source file, aside from the module specific initialization file (see below for details).

3.2 Initialization Mode

XEmacs uses a rather bizarre way of documenting variables and functions. Rather than have the documentation for compiled functions and variables passed as static strings in the source code, the documentation is included as a C comment. A special program, called '`make-docfile`', is used to scan the source code files and extract the documentation from these comments, producing the XEmacs '`DOC`' file, which the internal help engine scans when the documentation for a function or variable is requested.

Due to the internal construction of Lisp objects, subrs and other such things, adding documentation for a compiled function or variable in a compiled module, at any time after XEmacs has been dumped is somewhat problematic. Fortunately, as a module writer you are insulated from the difficulties thanks to your friend `ellcc` and some internal trickery in the module loading code. This is all done using the initialization mode of `ellcc`.

The result of running `ellcc` in initialization mode is a C source file which you compile with (you guessed it) `ellcc` in compile mode. Initialization mode is where you set the module name, version, title and gather together all of the documentation strings for the functions and variables in your module. There are several options that you are required to pass `ellcc` in initialization mode, the first of which is the mode switch itself, `--mode=init`.

Next, you need to specify the name of the C source code file that `ellcc` will produce, and you specify this using the `--mod-output=FILENAME` argument. FILENAME is the name of the C source code file that will contain the module variables and `docs_of_module` function.

As discussed previously, each module requires a short handle or module name. This is specified with the `--mod-name=NAME` option, where NAME is the abbreviated module name. This NAME must consist only of characters that are valid in C function and variable names.

The module version is specified using `--mod-version=VERSION` argument, with VERSION being any arbitrary version string. This version can be passed as an optional second argument to the Lisp function `load-module`, and as the third argument to the internal module loading command `emodules_load`. This version string is used to distinguish between different versions of the same module, and to ensure that the module is loaded at a specific version.

Last, but not least, is the module title. Specified using the `--mod-title=TITLE` option, the specified TITLE is used when the list of loaded modules is displayed. The module title serves no purpose other than to inform the user of the function of the module. This string should be brief, as it has to be formatted to fit the screen.

Following all of these parameters, you need to provide the list of all source code modules that make up your module. These are the files which are scanned by '`make-docfile`', and provide the information required to populate the `docs_of_module` function. Below is a sample '`Makefile`' fragment which indicates how all of this is used.

```
CC=ellcc --mode=compile
LD=ellcc --mode=link
MODINIT=ellcc --mode=init
CFLAGS=-O2 -DSOME_STUFF

.c.o:
    $(CC) $(CFLAGS) -c $<

MODNAME=sample
MODVER=1.0.0
MODTITLE="Small sample module"

SRCS=modfile1.c modfile2.c modfile3.c
OBJS=$(SRCS:.c=.o)

all: sample.ell
clean:
    rm -f $(OBJS) sample_init.o sample.ell

install: all
    mkdir `ellcc --mod-location`/mymods > /dev/null
    cp sample.ell `ellcc --mod-location`/mymods/sample.ell

sample.ell: $(OBJS) sample_init.o
    $(LD) --mod-output=$ $(OBJS) sample_init.o

sample_init.o: sample_init.c
sample_init.c: $(SRCS)
    $(MODINIT) --mod-name=$(MODNAME) --mod-version=$(MODVER) \
    --mod-title=$(MODTITLE) --mod-output=$ $(SRCS)
```

The above '`Makefile`' is, in fact, complete, and would compile the sample module, and optionally install it. The `--mod-location` argument to `ellcc` will produce, on the standard output, the base location of the XEmacs module directory. Each sub-directory of that directory is automatically searched for modules when they are loaded with `load-module`. An alternative location would be '`/usr/local/lib/xemacs/site-modules`'. That path can change depending on the options the person who compiled XEmacs chose, so you can always determine the correct site location using the `--mod-site-location` option. This directory is treated the same way as the main module directory. Each sub-directory within it is searched for a given module when the user attempts to load it. The valid extensions that the loader attempts to use are '`.so`', '`.ell`' and '`.dll`'. You can use any of these extensions, although '`.ell`' is the preferred extension.

3.3 Link Mode

Once all of your source code files have been compiled (including the generated init file) you need to link them all together to create the loadable module. To do this, you invoke `ellcc` in link mode, by passing the `--mode=link` option. You need to specify the final output file using the `--mod-output=NAME` option, but other than that all other arguments

are passed on directly to the system compiler or linker, along with any other required arguments to create the loadable module.

The module has complete access to all symbols that were present in the dumped XEmacs, so you do not need to link against libraries that were linked in with the main executable. If your library uses some other extra libraries, you will need to link with those. There is nothing particularly complicated about link mode. All you need to do is make sure you invoke it correctly in the '`Makefile`'. See the sample '`Makefile`' above for an example of a well constructed '`Makefile`' that invoked the linker correctly.

3.4 Other ellcc options

Aside from the three main `ellcc` modes described above, `ellcc` can accept several other options. These are typically used in a '`Makefile`' to determine installation paths. `ellcc` also allows you to over-ride several of its built-in compiler and linker options using environment variables. Here is the complete list of options that `ellcc` accepts.

`--mode=compile`
: Enables compilation mode. Use this to compile source modules.

`--mode=link`
: Enabled link edit mode. Use this to create the final module.

`--mode=init`
: Used to create the documentation function and to initialize other required variables. Produces a C source file that must be compiled with `ellcc` in compile mode before linking the final module.

`--mode=verbose`
: Enables verbose mode. This will show you the commands that are being executed, as well as the version number of `ellcc`. If you specify this option twice, then some extra debugging information is displayed.

`--mod-name=NAME`
: Sets the short internal module NAME to the string specified, which must consist only of valid C identifiers. Required during initialization mode.

`--mod-version=VERSION`
: Sets the internal module VERSION to the specified string. Required during initialization mode.

`--mod-title=TITLE`
: Sets the module descriptive TITLE to the string specified. This string can contain any printable characters, but should not be too long. It is required during initialization mode.

`--mod-output=FILENAME`
: Used to control the output file name. This is used during initialization mode to set the name of the C source file that will be created to FILENAME . During link mode, it sets the name of the final loadable module to FILENAME .

`--mod-location`
: This will print the name of the standard module installation path on the standard output and immediately exit `ellcc`. Use this option to determine the directory prefix of where you should install your modules.

`--mod-site-location`
: This will print the name of the site specific module location and exit.

`--mod-archdir`
: Prints the name of the root of the architecture-dependent directory that XEmacs searches for architecture-dependent files.

`--mod-config`
: Prints the name of the configuration for which XEmacs and `ellcc` were compiled.

3.5 Environment Variables

During its normal operation, `ellcc` uses the compiler and linker flags that were determined at the time XEmacs was configured. In certain rare circumstances you may wish to over-ride the flags passed to the compiler or linker, and you can do so using environment variables. The table below lists all of the environment variables that `ellcc` recognizes.

`ELLCC` This is used to over-ride the name of the C compiler that is invoked by `ellcc`.

`ELLLD` Sets the name of the link editor to use to created the final module.

`ELLCFLAGS`
: Sets the compiler flags passed on when compiling source modules. This only sets the basic C compiler flags. There are certain hard-coded flags that will always be passed.

`ELLLDFLAGS`
: Sets the flags passed on to the linker. This does not include the flags for enabling PIC mode. This just sets basic linker flags.

`ELLDLLFLAGS`
: Sets the flags passed to the linker that are required to created shared and loadable objects.

`ELLPICFLAGS`
: Sets the C compiler option required to produce an object file that is suitable for including in a shared library. This option should turn on PIC mode, or the moral equivalent thereof on the target system.

`ELLMAKEDOC`
: Sets the name of the '`make-docfile`' program to use. Usually `ellcc` will use the version that was compiled and installed with XEmacs, but this option allows you to specify an alternative path. Used during the compile phase of XEmacs itself.

4 De ning Functions

One of the main reasons you would ever write a module is to provide one or more functions for the user or the editor to use. The term function is a bit overloaded here, as it refers to both a C function and the way it appears to Lisp, which is a subroutine, or simply a subr. A Lisp subr is also known as a Lisp primitive, but that term applies less to dynamic modules. See section "Writing Lisp Primitives" in XEmacs Internals Manual, for details on how to declare functions. You should familiarize yourself with the instructions there. The format of the function declaration is identical in modules.

Normal Lisp primitives document the functions they defining by including the documentation as a C comment. During the build process, a program called '`make-docfile`' is run, which will extract all of these comments, build up a single large documentation file, and will store pointers to the start of each documentation entry in the dumped XEmacs. This, of course, will not work for dynamic modules, as they are loaded long after XEmacs has been dumped. For this reason, we require a special means for adding documentation for new subrs. This is what the macro `CDOCSUBR` is used for, and this is used extensively during `ellcc` initialization mode.

When using `DEFUN` in normal XEmacs C code, the sixth "parameter" is a C comment which documents the function. For a dynamic module, we of course need to convert the C comment to a usable string, and we need to set the documentation pointer of the subr to this string. As a module programmer, you don't actually need to do any work for this to happen. It is all taken care of in the `docs_of_module` function created by `ellcc`.

4.1 Using DEFUN

The full syntax of a function declaration is discussed in the XEmacs internals manual in greater depth. section "Writing Lisp Primitives" in XEmacs Internals Manual. What follows is a brief description of how to define and implement a new Lisp primitive in a module. This is done using the `DEFUN` macro. Here is a small example:

```
DEFUN ("my-function", Fmy_function, 1, 1, "FFile name: ", /*
Sample Emacs primitive function.

The specified FILE is frobnicated before it is fnozzled.
*/
    (file))
{
  char *filename;

  if (NILP(file))
    return Qnil;

  filename = (char *)XSTRING_DATA(file);
  frob(filename);
  return Qt;
}
```

The first argument is the name of the function as it will appear to the Lisp reader. This must be provided as a string. The second argument is the name of the actual C function that will be created. This is typically the Lisp function name with a preceding capital `F`, with hyphens converted to underscores. This must be a valid C function name. Next come the minimum and maximum number of arguments, respectively. This is used to ensure that the correct number of arguments are passed to the function. Next is the `interactive` definition. If this function is meant to be run by a user interactively, then you need to specify the argument types and prompts in this string. Please consult the XEmacs Lisp manual for more details. Next comes a C comment that is the documentation for this function. This comment must exist. Last comes the list of function argument names, if any.

4.2 Declaring Functions

Simply writing the code for a function is not enough to make it available to the Lisp reader. You have to, during module initialization, let the Lisp reader know about the new function. This is done by calling `DEFSUBR` with the name of the function. This is the sole purpose of the initialization function `syms_of_module`. See Section 2.2 [Required Functions], page 3, for more details.

Each call to `DEFSUBR` takes as its only argument the name of the function, which is the same as the second argument to the call to `DEFUN`. Using the example function above, you would insert the following code in the `syms_of_module` function:

```
DEFSUBR(Fmy_function);
```

This call will instruct XEmacs to make the function visible to the Lisp reader and will prepare for the insertion of the documentation into the right place. Once this is done, the user can call the Lisp function `my-function`, if it was defined as an interactive function (which in this case it was).

Thats all there is to defining and announcing new functions. The rules for what goes inside the functions, and how to write good modules, is beyond the scope of this document. Please consult the XEmacs internals manual for more details.

5 Defining Variables

Rarely will you write a module that only contains functions. It is common to also provide variables which can be used to control the behavior of the function, or store the results of the function being executed. The actual C variable types are the same for modules and internal XEmacs primitives, and the declaration of the variables is identical.

See section "Adding Global Lisp Variables" in XEmacs Internals Manual, for more information on variables and naming conventions.

Once your variables are defined, you need to initialize them and make the Lisp reader aware of them. This is done in the `vars_of_module` initialization function using special XEmacs macros such as `DEFVAR_LISP`, `DEFVAR_BOOL`, `DEFVAR_INT` etc. The best way to see how to use these macros is to look at existing source code, or read the internals manual.

One *very* important difference between XEmacs variables and module variables is how you use pure space. Simply put, you never use pure space in XEmacs modules. The pure space storage is of a limited size, and is initialized properly during the dumping of XEmacs. Because variables are being added dynamically to an already running XEmacs when you load a module, you cannot use pure space. Be warned: do not use pure space in modules. Repeat, do not use pure space in modules.Once again, to remove all doubts: DO NOT USE PURE SPACE IN MODULES!!!

Below is a small example which declares and initializes two variables. You will note that this code takes into account the fact that this module may very well be compiled into XEmacs itself. This is a prudent thing to do.

```
Lisp_Object Vsample_string;
int sample_boolean;

void
vars_of_module()
{
  DEFVAR_LISP ("sample-string", &Vsample_string /*
This is a sample string, declared in a module.

Nothing magical about it.
*/);

  DEFVAR_BOOL("sample-boolean", &sample_boolean /*
*Sample user-settable boolean.
*/);

  sample_boolean = 0;
  Vsample_string = build_string("My string");
}
```

Table of Contents

Index