USENIX Association

Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

Sushi – an extensible human interface for NetBSD

by

Tim Rightnour
The NetBSD Project
garbled@netbsd.org

Abstract

This document describes Sushi, a menuing system designed for the administration and configuration of the NetBSD operating system. Included are detailed explanations of the reasons behind the creation and design of Sushi. In addition, this document includes descriptions of the file formats and examples of the menus provided by Sushi.

1. An introduction to Sushi

Sushi stands for Simple to Use System-Human Interface. Sushi was designed to provide an easy to use environment, allowing a new administrator to set up or maintain a NetBSD system. Sushi can be used for a variety of tasks, such as the initial configuration of a system or to facilitate maintenance. Custom menus can be created and installed at any time, allowing administrators to create their own menus to perform frequently used tasks.

Sushi is a text-based user interface to the system. It has been written in C and carries a BSD license. Tasks that have already been written for Sushi include configuring network interfaces, configuring startup (rc) scripts, user management, and installation of third party packages using the NetBSD package system.

Currently, Sushi is only available in the development branch of NetBSD and has not yet been made part of an official release. Administrators wishing to try out Sushi can do so by installing one of the NetBSD-current snapshots or by compiling it from source. Installation on an older system is unlikely to work, as the provided menus are tuned for the development branch and Sushi itself requires many curses features and related libraries not available in the 1.5 release of NetBSD, such as *libform* and *CDK*.

When starting Sushi the administrator is presented with a menu of task categories. The administrator can select one of these categories and will then be provided with additional options, either more categories or actual tasks will be presented. Once a task has been selected the administrator will then be presented with a form.

A form usually comprises a series of questions, with blanks next to each one for providing answers. A typical form might have configuration details, such as the IPv4 address of a network interface to be setup or possibly a series of yes/no questions to provide options. Once the administrator has edited this form it can then be submitted to be processed by Sushi.

Processing the form generally consists of interpreting the various options and fields the administrator has filled out and executing the proper actions. For example, had the administrator configured a network interface, Sushi would apply those changes to the interface. As the form is processed, a window with the output of the commands Sushi is executing will be displayed. If the commands fail, the administrator is informed and can then edit the form again. Sushi makes no attempt to interpret the output of the command that has been executed, the analyzation of the failure is left to the administrator. Once the task has been completed, the administrator can go on to other menus and perform additional tasks or exit Sushi.

2. Design Goals

Sushi was designed with a number of specific goals in mind. First, it needed to provide an intuitive interface to the system. Second, it needed to be easily extensible so virtually any task could be programmed into Sushi, without recompiling or having to learn another programming language. Third, it needed to provide support for a variety of native languages. Finally, it needed to provide for compatibility with manual changes made to the system.

2.1. Simple to Use

In order to be useful to most new administrators of the NetBSD operating system, Sushi needed to have a very instinctive interface. If the tool is just as or more complex than the actual system is to use, then there is no point in using it. The instinctive interface was accomplished by setting up a simple hierarchical menu structure. This allows tasks to be easily categorized by their general function or area of influence. For example, under a menu called "User and Group Management", one might place functions such as creating and deleting users, assigning users to groups, and modifying users or groups.

The interface also allows a number of input field types that define the types of data the administrator can enter. One such field is the basic text entry, where freeform text can be entered. Information such as host names or data files names can be entered by the administrator here. Other field types include multiple choice selection fields, where the administrator can pick from a predetermined set of choices or restricted fields, where the administrator can only enter things such as numbers or IPv4 addresses.

Sushi also attempts to provide help for all menus or tasks. A simple status bar at the bottom of the screen shows some of the basic navigation commands available to the administrator. In most menus or tasks, the help key (F1) will bring up a section of help text written specifically for the current menu or task. This help text is designed to explain to the administrator what some of the various questions or options mean and assist in choosing the appropriate selection. It often points to manual pages, which can assist the administrator further in determining how to proceed.

Sushi attempts to provide this easy to use interface without becoming overly cumbersome. The features which Sushi can provide are useful to administrators of all skill levels, not because the tasks can't be accomplished by hand, but because the menu interface is easier than editing the files by hand. Sushi attempts to provide the menu and form to the administrator, without forcing them to answer too many questions or fill in unnecessary fields.

2.2. Easily Extensible

While a menuing system written entirely in a compiled language might provide an easy to use interface for the administrator, it does so at the price of maintenance headaches. Sushi needed to be easily modifiable and provide a way for administrators to quickly add menus of their own to the system.

One of the reasons Sushi is able to provide an easily extensible menu interface is because it is an interpreter, rather than a precompiled set of menus. All of the menus and tasks that are provided by Sushi are actually taken from a set of command files and directories living on the machine.

At start up, Sushi looks in a number of base directories for its index files. An index file contains a description of a sub-menu or task and the name of a subdirectory where that sub-menu or task lives. It also contains a "quicklink" keyword which can provide a fast method to jump to the menu from the command line. Subdirectories can contain more index files or they can terminate, in what is called an "endpoint".

An endpoint is a task that is to be completed. It can be anything from a command that is executed when the administrator selects the menu item, to a complex set of forms that need to be filled out by the administrator. They can even point to functions internal to Sushi, allowing very complex tasks to be performed or the modification of internal Sushi variables, such as turning on logging.

Forms are specified by a form file. This file describes the form to Sushi, indicating field types, field descriptions, and data specific to the field, such as predetermined choices for a multiple choice field.

Besides administrator input, fields can also gather data from multiple sources. For example, if we were writing a form to modify the machine's network interface, we would want to provide the administrator with the current settings for that interface when the form is displayed. This can be accomplished by using scripts. Scripts are basically any executable program that outputs the desired information. Arguments can be passed to these scripts, allowing the menu to tell the script which network interface is being configured. By passing these arguments, Sushi can call the script that can look up the IPv4 address of the interface and read the data back to fill in the field. When the form is displayed to the administrator the current IPv4 address will be present in the field, either to edit or leave as is.

In the case of a form, a script which is executed upon completion of the form also exists. The script can be any type of executable program, giving freedom to the designer to use the language most suitable for the task. This script is executed and given the contents of each field, in order, as its arguments. The script is expected to verify the choices made by the administrator and execute the appropriate task. For example, with the form that configures a network interface, the script would analyze the arguments given from the form and construct an ifconfig command and execute it. The exit code of the script determines the success or failure of the action to be displayed to the administrator watching the output screen.

The base directories that Sushi searches for index files can also be modified through a configuration file by simply adding more directories to the list. In addition, one of the directories searched by Sushi is the administrator's home directory. This allows administrators to create menus that are only available to the creator and store them locally. For example, an administrator might wish to create a menu to automate a personal task, such as customization of a personal rc file. Using this mechanism, third party packages can be extended to install Sushi menus for their own configuration. When Sushi scans for its index files, it creates a single hierarchical tree out of all the menus it finds.

2.3. Internationalization Support

Sushi also provides support for multiple written languages. All of the text displayed by the Sushi engine itself was written using the *catgets(3)* interface and is therefore capable of supporting a limited set of native languages.

In addition, most of the data files used by Sushi (such as the help text, forms, and index files) can all be written in an alternate language and stored using different file suffixes. When Sushi is started in an environment where the preferred user interface language variables have been set, it will load the proper files whenever they are available and failing that display the English defaults.

2.4. Compatibility with manual changes

When making changes to system configuration files, it is important to do so in a way that is compatible with manual changes. Many automated configuration systems control their files by placing special markers in the file, causing an administrator to have to work around these markers. Other programs might require that the files only be modified by the configuration program and never edited by hand. This is unacceptable for a configuration tool like Sushi for a number of reasons.

First and foremost, Sushi is designed to assist a new NetBSD administrator in getting his or her system up and running quickly. Once the new administrator has overcome the learning curve of using NetBSD and is competent in it, they should not be penalized for having used Sushi or be forced to continue to use it.

Secondly, a machine might be administered by multiple people or change hands. Sushi should not leave the system in an unmaintainable state, nor should it be unable to cope with changes made manually, if the new administrator wishes to use Sushi to edit previously handmade configuration files.

Sushi retains compatibility with manual changes by having a basic understanding of the various configuration file formats. Some file formats are more complex than others and in some cases certain options that are available are impossible to implement in a script. However, Sushi makes all possible efforts to interpret files properly and write them back in a readable format. Sushi also does not use any special markers or create uneditable entries in the files. This is not a feature of the engine, but rather a feature of the menus that ship with Sushi as part of NetBSD. It is also considered a golden rule when creating new menus for Sushi.

3. Designing menus for Sushi

The programming interface for Sushi has been designed to be very simple to pick up and begin writing menus with. Generally speaking, if an administrator can write a script to accomplish a specific task, they can turn that work into a Sushi menu with a minimal amount of work.

3.1. Search order

Upon entering a directory, Sushi looks for a number of different files specifying the action it should take. Sushi looks for these files in a specific order and will begin processing the first file that it recognizes, ignoring the rest of the files in that directory. It causes an error when an endpoint has no files in it and Sushi will exit if it encounters one. The search order of these files is as follows:

- The index file “index”.
- The preform file “preform”.
- The form file “form”.
- The script file “script”.
- The execute file “exec”.
- The function file “func”.
- The help file “help”.

For each file, Sushi will first attempt to load a file ending in a locale suffix, such as “.de” for German. Should it fail to find the appropriate translated item, it will then search for the file without a suffix.

3.2. The index file

The index file comprises multiple lines containing three whitespace separated columns where each line of this file represents a single menu item. These are the name of the subdirectory containing the menu item, a quicklink, and finally a description of the menu item.

The subdirectory argument is used to specify which subdirectory contains the next sub-menu or endpoint. Any subdirectories specified in the index file must exist for Sushi to process the tree properly. The subdirectory can be replaced with the keyword “BLANK” to place a blank line on the menu. This can be used to group certain types of actions together in the menu.

The second argument to a line in the index file is the quicklink. This is a single word that can be used to jump to this menu or task from the command line. It should be something that is easy to remember and obvious to the administrator, such as “users” to point to the “User and Group Management” sub-menu. The administrator can then jump directly into this menu by starting Sushi with “users” as the only argument. Again, the “BLANK” keyword can be used to specify a blank line in the menu.

The description of the menu item is meant to give a brief title to the sub-menu or task located in the subdirectory below. It should consist of a brief title, such as “User and Group Management”. Should the entry point to a sub-menu, the description text will be used as the main heading for that sub-menu. This title is limited to approximately 70 characters to allow it to fit on a standard 80 column wide screen. Again, the description can be replaced with the “BLANK” keyword to create a blank line.

It is important to note that when creating blank lines in the menu that all three arguments must contain the “BLANK” keyword. Sushi also ignores any lines beginning with a comment symbol (#). For an example of an index file see figure 1.

3.3. The form file

A form file is a whitespace delimited list of fields consisting of a field type and followed by a description. Fields may be of many different types and each one has a set of different arguments it expects. Arguments are separated from the field type keyword with a “:” and separated by commas.

There is also a “preform” file, which can be used to provide a series of forms to the user. The preform allows the programmer to gather data from the user, which can later be used in the form to populate certain fields. An example of where this might be useful would be setting up a network interface. The preform would ask the administrator which interface they wanted to setup and pass that information to the form so it could query the proper interface for its current settings. Any data entered into a preform is made available to the form via the special argument key “@@@1@@@”, which specifies the data from the first field of the preform. When Sushi interprets the form file, the special argument key will be replaced with the data the administrator entered whenever it is found in the form file. Sushi is limited to a single preform and associated form pair. Multiple preforms are not possible.

The first type of field is the basic freeform entry field, which is denoted by the keyword “entry”. The only argument for this keyword is the length of the maximum entry in characters. When the entry field is specified, a

```
# $NetBSD: index,v 1.5 2001/04/26 02:26:16 garbled Exp $
install         install Software Installation and Maintenance
system          system  System Maintenance
users           users   Security and Users
procs           procs   Processes and Daemons
network         network Network related configuration
BLANK           BLANK   BLANK
info            info    Using sushi (information only)
util            util    Sushi utilities (logging/scripting)
```

Figure 1. Example of an index file

blank underlined input field will be placed on the display into which any type of data may be entered. You may also prefix the entry keyword with ‘‘req−’’ to specify that the field must be filled in by the administrator before form processing can take place. Required fields are prefixed with an asterisk on the display.

An ‘‘escript’’ field type is an entry field whose initial value is filled in by running an associated script. The arguments to an escript field type are the maximum field length, the name of the script to be executed, and any optional arguments you wish to pass to the script. The script that is executed can be of any executable format and is expected to return a single line of text. It is important that the script always return something to the Sushi engine to avoid possible errors at run time. The escript keyword can be prefixed with ‘‘req−’’ to make it a required field or be specified as ‘‘nescript’’ to create an uneditable field. Uneditable fields may be used to display data to the administrator without allowing them to modify it. Data in these fields will still be passed to the task script upon completion of the form.

A list field type is specified by the keyword ‘‘list’’. This field type will present the administrator with a multiple choice field. An administrator can only select one of the predetermined choices from the list and may not modify any of them. The administrator can toggle the values of the list by using the TAB key or bring up a selection list box containing all of them with the F4 key. The arguments consist of a comma separated list of possible choices. This is especially useful for generating yes or no questions for the administrator. The list keyword can be prefixed with ‘‘req−’’ to make it a required field.

A multilist, specified by the ‘‘multilist’’ keyword, is a list where the administrator can select more than one of the possible choices from the list. This is accessed by the administrator via the F4 key and selections are toggled with the space bar. The format for the multilist is the same as a list.

The ‘‘blank’’ keyword can be used to create an item with no corresponding entry field. This can be used to provide additional lines of description for the previous field. No data is passed to the task script when a blank field is specified and a blank field does not count as an argument to a task script.

The ‘‘noedit’’ keyword can be used to create an uneditable field that will still be passed to the task script. This is similar in operation to the ‘‘nescript’’ field type and is used primarily in displaying data to the administrator, or passing special arguments to the task script. The only argument is the string to be displayed in the field.

The ‘‘invis’’ field type allows the programmer to create fields which will not show up on the form. The description for the field will still be visible however. The contents of the field will be passed to the task script upon completion of the form. The only argument is the string to be placed in the invisible field.

A function field, specified by the “func” keyword, allows the programmer to create special list field types whose values are populated by calling a function internal to the Sushi engine. In order to utilize the function field type, the appropriate function must first be programmed into the Sushi engine and the engine must be recompiled. The function field has two arguments, the name of the function to be called and a single text argument to be passed to the function. Functions are expected to return a list of values and have the following prototype:

*(char **)function(char *argument);*

Functions must be added to the “functions.c” source file and added to the “func_map” array at the top of that file, as well as to the “functions.h” header file. The function keyword may be prefixed with “req–” to make it a required field. Multilist functions are possible by specifying the keyword “multifunc” and are programmed identically to function fields. It is strongly encouraged that functions be avoided whenever possible in Sushi. Functions require recompiling of the engine in order to be made available and are therefore more difficult to maintain. Whenever possible other field types should be used.

The “script” field type, allows the programmer to create a list-type field, whose contents are created by running an executable program. The arguments to the script field type are the name of the script to be executed and any number of arguments the programmer wishes to pass to that script. The script is expected to produce a list of values, one per line, on standard output. These values are then read and used to create the list field type. The script keyword may be prefixed with “req–” to make it a required field. Multilist scripts can be made by specifying the keyword “multiscript”. The named script is expected to be located in the same directory that the form file exists in.

To restrict field input to an integer type the “integer” keyword can be used. This field type has three required arguments and an optional fourth argument. The first three arguments are, in order, the maximum length of the field in characters, the minimum integer value allowed for this field, and the maximum integer value allowed for this field. The fourth optional argument is an integer default value for this field. Administrator entries in fields of these types will be checked by the Sushi engine to make sure they are between the minimum and maximum values for the field type. The integer keyword may be prefixed with “req–” to make it a required field.

Integer fields may be prefilled with data by using the “iscript” field type. This field type is similar to the escript field type, in that it executes a script which provides a value back to Sushi to fill in the field. The arguments for the iscript field type are the maximum length of the field in characters, the minimum integer value allowed for this field, the maximum integer value allowed for this field and the name of the script to execute, as well as any number of additional arguments the programmer wishes to pass to the script. The iscript field type can be made into a required field by prefixing it with “req–”.

Fields may also be restricted to IPv4 and IPv6 addresses through the field types “ipv4” and “ipv6”. Each of these may be prefixed with “req–” to make it a required field. The IPv4 field type allows addresses to be entered in dotted quad format or in hex. Both of these field types have one argument, which is the optional prefilled value for the field.

In addition, there are script forms of the IPv4 and IPv6 field types, called “ipv4script” and “ipv6script.” Both of these may be made into required fields by prefixing them with “req–”. The arguments to these fields are the name of the script which is to be executed and any optional arguments the programmer wishes to pass that script. These fields both behave similarly to the escript field type.

When programming a field that has optional arguments, such as the ipv4 field type, it is important to still use the “:” symbol to separate the field keyword from the arguments. In order to produce an ipv4 field which has no data prefilled into the field the programmer would create a line such as:

```
ipv4:     IPV4 address
```

By using combinations of the above field types, it is possible to create nearly any type of form that the programmer may wish to build. Going back to the previous example of creating a Sushi menu to configure network interfaces, figures 2 and 3 are the preform and form files to accomplish this, respectively.

In the preform file (figure 2), a script is being run which collects the names of different interfaces on the machine. This script could be something like:

```
ifconfig -l | xargs -n 1 echo
```

Once the administrator has selected an interface from the list, that interface name will then be passed to the form.

In the form file (figure 3) each instance of "@@@1@@@" will be replaced with the network interface the administrator had selected from the preform, say for example "fxp0". The fourth line of the form asks the administrator for the IPv4 address of the interface. The ipv4script field type will run the script "script2" giving it the arguments "4" and "fxp0". In this case, the script2 script, runs "ifconfig fxp0" and pulls the address out of that, prefilling the field for the administrator with the IPv4 address of the fxp0 interface.

3.4. The script file

The script file is a script or executable program of some type, that is executued by Sushi when encountered. The script can be encountered in one of two ways.

First, if there are no other files found in the search order, the script will be run when the

```
# $NetBSD: preform,v 1.1 2001/04/25 03:43:33 garbled Exp $
script:script1                Select an interface to operate on:
```

Figure 2. Example of a preform file

```
# $NetBSD: form,v 1.1 2001/04/25 03:43:33 garbled Exp $
noedit:@@@1@@@                   Changing interface:
list:both,now,boot               Modify interface at boot-time, now, or both?
req-ipv4script:script2,4,@@@1@@@        Interface IPV4 Address
ipv4script:script2,n,@@@1@@@            Interface IPV4 Netmask
ipv4script:script2,b,@@@1@@@            Interface IPV4 Broadcast Address
script:script2,m,@@@1@@@                Media Type
script:script2,o,@@@1@@@                Media Options
ipv6script:script2,6,@@@1@@@            Interface IPV6 Address
iscript:3,0,128,script2,pre,@@@1@@@     Interface IPV6 Prefix Length(netmask)
escript:32,script2,i,@@@1@@@            Interface Network-ID
multilist:link0,link1,link2             Interface link options
iscript:5,1,99999,script2,mtu,@@@1@@@   Interface MTU
iscript:2,0,99,script2,met,@@@1@@@      Interface Metric
```

Figure 3. Example of a form file

menu item is selected. No arguments will be passed to the script when run in this manner. Output of the script will be displayed to the administrator and success or failure status will be noted.

The second way a script file can be executed is in response to completing a form. When a form has been filled out and accepted by the administrator, Sushi will execute the script file giving it the data filled in the form as arguments. Empty fields will be passed to the script as empty string arguments, ensuring that field position will always remain the same when translated to arguments.

The second form of the script file is generally where the actual actions take place in Sushi. In the example of modifying a network interface, the script would interpret the data from the form and make the actual changes to the network device or startup files. The script should make all attempts to fail cleanly with an error code of 1. Success should be noted with an exit code of 0.

3.5. The execute file

The execute file ‘‘exec’’ can be used to execute a simple program located anywhere on the system. It is generally used to provide simple access to programs that provide information about the system. The program will be executed as it appears in the execute file, with no arguments being passed or interpreted.

An example of using the execute file, would be to program a simple menu which displayed a list of packages installed on the administrator’s system, in which case the execute file would contain: ‘‘pkg_info’’.

3.6. The function file

The function file ‘‘func’’ can be used to provide access to some of Sushi’s internal functions that are made available to the menu programmer. The format of this file is the name of a function, followed by a comma and an optional single argument which will be passed to the function as a string.

This can be used to activate simple features in Sushi or program more complex ones. One example of using this, is to turn on the internal logging functionality of Sushi, which writes all actions out to a logfile. This is accomplished by having a function file containing: ‘‘log_do,on’’.

3.7. The help file

The help file is a simple text document that will be displayed to the administrator when the help key is pressed (usually F1). The help file can be used to give the administrator more information about what the various menus available do or what the individual fields mean in a form. A help file can be located anywhere in the Sushi tree.

The help file should give a brief description of the menu items or form it is describing. It should not be used to completely explain a subject matter to the administrator, rather it should point him to documents or manual pages which provide that information. It is a good idea to provide help files for every menu in the tree, to allow administrators to understand what each item does and warn them of potential pitfalls that may lie ahead.

4. The future of Sushi

Sushi still has a number of goals left to accomplish before it can be considered complete. Additional menus and tasks still need to be written. Eventually, most administration tasks that need to be performed on a NetBSD system will be automated in some way by Sushi. Certain areas of Sushi still need enhancement, such as displaying the command that will be executed to the administrator before executing it (to aid a new administrator in learning NetBSD commands).

In addition, due to the design of Sushi, it would not be difficult to write other processing engines with a different interface. For example, a web-based interface or X11 interface could easily be written for Sushi, reusing most of the parsing code.

5. Conclusion

Sushi was written to provide a intuitive and easy to use interface to the NetBSD operating system. It is my hope that as Sushi evolves and encompasses more tasks that administrators do on a daily basis or while setting up a machine for the first time, more users will find the learning curve of NetBSD less daunting. Sushi will one day allow a novice administrator to completely administrate a machine, making NetBSD a more user-friendly operating system in the process.

6. About the Author

Tim Rightnour has been a user of NetBSD since 1994 and has been a developer for NetBSD for approximately 3 years. He has worked on projects ranging from device drivers to the NetBSD Package System. He is also the author of ClusterIt, a collection of programs used to automate and administer large groups of machines.