CLIP SPECIFICATION

David-John Burrowes
Joseph E. Kowalski III

Version 1.0 Approved 22 Jan 2003

PSARC 1999/645

1. Introduction

CLIP (Command Line Interface Paradigm) is a set of guidelines that specify the human interface for command line utilities. The goal of the CLIP specification is to provide a uniform definition of command line utility syntax and behavior while seeking to provide the best overall experience for the end user. It starts with familiar practice and adds guidelines for certain informal and *de facto* details.

2. Previous Work

For over 10 years, Sun has embraced a set of utility syntax guidelines commonly referred to as getopt. The formal approval of this syntax is provided by PSARC/1991/031, G*etopts Usage*. Fourteen command syntax guidelines are provided. Guidelines 1 through 13 are adopted directly from the *X/Open CAE Specification, System Interface Definitions, Issue 4 (XBD).* Guideline 14 is a Sun Microsystems local extension modeled after common practice, including BSD.

These guidelines were adopted without modification with the exceptions of the addition of guideline 14 and the deletion of text from the *XBD* which described syntax used elsewhere in the *XBD* document to characterize exceptions.

It is worth noting that the 13 *XBD* guidelines are based on definitions provided with *SVID Issue 1*, which in turn were based upon external customer polls. The only clear conclusion from those polls was that consistency was highly desirable. The exact syntax was far less important. This bias toward consistency should be retained.

This CLIP specification includes the 14 guidelines from *Getopts Usage* and as such represents a superset of the getopt convention.

3. Applicability

The getopt guidelines as described in PSARC/1991/031, *Getopts Usage*, continue to represent the basic command line syntax standard of Sun Microsystems. The CLIP specification represents an optional extension to those basic utility syntax guidelines.

The CLIP extensions to the getopts guidelines are not to be applied haphazardly to system utilities. The inconsistency that would result would far outweigh any gain in usability. Rather the CLIP extensions are only to be applied to logical collections of utilities. Note that this is an extension of the rationale provided in the opinion on *Getopts Usage*:

> "As the main benefit from using these guidelines is increased system regularity, it is important that they not be used irrationally to decrease system regularity. New additions to utilities or utility sets should conform to the style of the utility, even if conflicting with these guidelines. An example would be the "X" utilities. An addition to this set should use the same multi-character option names as is common with the "X" utilities. The same applies to Java utilities even though there was no significant reason for the initial non-conformance."

It is difficult to absolutely characterize the properties that define a logical collection of utilities. Most layered products are of a size such that the entire product should be converted to CLIP (or not) as a unit. The difficulty lies in characterizing what subcomponents of extremely large products comprise a logical collection. Perhaps this difficulty only applies to Sun Solaris utilities, with all other Sun products forming a logical collection.

There may be some consistency in first dividing Solaris utilities along consolidation boundaries. However, historical curiosities in the consolidation structure guarantee there will be exceptions, and the resulting decomposition still leaves the massive OS/Networking consolidation.

Occasionally, the Solaris directory structure may provides some clues. Certainly, nothing in the `/usr/xpg4`, `/usr/ucb`, `/usr/5bin`, or `/usr/sfw` hierarchies should be converted to CLIP because the intent of these components is compatibility with an external reference. `/usr/openwin` and `/usr/sadm` (most) are probably inappropriate for conversion because these components are near the end of their useful deployment. Conversely, the collection of utilities in `/usr/ccs/bin` could represent a logical collection appropriate to consider converting to CLIP, as these are fairly constantly program development tools. However, the collection of utilities in `/usr/ccs/bin` tend to have existing command line syntax which could not be brought into line with the CLIP syntax. This underlines that being a logical collection is only necessary, but not sufficient, for justifying conversion to CLIP.

That still leaves the large number of utilities in `/usr/bin` and `/usr/sbin` as a unit. Judgment must be applied as to which collections of these utilities would comprise a logical grouping. The concept is to place yourself in the mind of a typical user and ask the question, "Which utilities would I expect to behave similarly?". Uniform opinions are unrealistic in this area, but over time experience and consensus will be obtained on this topic. Project teams considering using CLIP are advised to solicit early review of collections of utilities they propose to modify.

Also, due to the expense relative to the gain of enhancing existing utilities, it is in the purview of each steering committee (or PIC) to determine if such projects provide sufficient utility to be funded.

4. Definitions and Conventions

This section describes the argument syntax of the standard utilities and introduces terminology used for describing the arguments processed by utilities. It is adapted from IEEE Std 1003.1-2001 (SUSv3). The changes are to allow for CLIP extensions and the deletion of irrelevant text. This section is normative with the actual specification in subsequent sections.

The following discussion use the following command invocation as an example:

```
utility_name -a --longopt1 -c option_argument \
    -foption_argument --longopt2=option_argument \
    --longopt3 option_argument operand
```

1. The utility in the example is named *utility_name*. It is followed by options, option-arguments, and operands, collectively referred to as arguments. The arguments that consist of a hyphen followed a single letter or digit, such as '`-a`', are known as “short-options”. The arguments that consist of two hyphens followed by a series of letters, digits and hyphens, such as '`--longopt1`', are known as “long-options”. Collectively, short-options and long-options are referred to as “options” (or historically, “flags”). Certain options are followed by an “option-argument”, as shown with “-c *`option_argument`*”. The arguments following the last options and option-arguments are named “operands”. Once the first operand is encountered, all subsequent arguments are interpreted to be operands.
2. Option-arguments are sometimes shown separated from their short-options by <blank>s, sometimes directly adjacent. This reflects the situation that in some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. This document requires that the option be a separate argument from its option-argument, but there are some exceptions to ensure continued operation of historical applications:

   a) If the SYNOPSIS of a utility shows a <space> between a short-option and option-argument (as with “`-c` *`option_argument`*” in the example), a conforming application shall use separate arguments for that option and its option-argument.

   b) If a <space> is not shown (as with “`-f`*`option_argument`*” in the example), a conforming application shall place an option and its option-argument directly adjacent in the same argument string, without intervening <blank>s.

   c) Notwithstanding the preceding requirements on conforming applications, a conforming implementation shall permit an application to specify short-options and option-arguments as a single argument or as separate arguments whether or not a <space> is shown on the synopsis line.

   d) Long-options with option-arguments are always documented as using an equals sign as the separator between the option name and the option-argument. If the OPTIONS section of a utility shows an equals sign ('=') between a long-option and its option-argument (as with

"`--longopt2=`*`option_argument`*" in the example), a conforming application shall also permit the use of separate arguments for that option and its option-argument (as with "`--longopt1` *`option_argument`*" in the example).

5. CLIP Guidelines

This section presents the 21 guidelines that make up the CLIP specification.

Guidelines 1 through 14 are imported from PSARC/1991/031, *Getopts Usage*. However, guidelines 1 through 13 are updated to the current wording in *Open Group Technical Standard Base Specifications, Issue 6* (SUSv3). Guideline 14 in enhanced by clarifying the character set associated subcommand names. A utility that conforms to guidelines 1 through 14 is said to be **getopt conformant**.

CLIP adds guidelines 15 through 21 to the getopt set of guidelines. A utility that conforms to all guidelines 1 through 21 is said to be **CLIP conformant**. All CLIP conformant utilities are also getopt conformant.

As the specification is expected to evolve compatibly over time, all claims of getopt conformance or CLIP conformance should be to a specific version of this document. Is is expected that description enhancements and additions to the conventional names enumerated in section 6 would constitute a minor revision to the document, while addition or deletion of a guideline would constitute a major revision.

#	Source	Guideline	Notes
1	SUS[1]	Utility names should be between two and nine characters, inclusive.	The nine character maximum is often waived for rarely used utilities.
2	SUS	Utility names should include lower-case letters (the **lower** character classification) and digits only from the portable character set.	Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. Operands and option-arguments can contain characters not specified in the portable character set. The characters '`-`' (hyphen) and '_' (underscore) are often present in utility names not expected to be subject to standardization. Other exceptions, such as mixed case, are rarely appropriate. The name of a utility cannot have colon as the last character. If a colon is used in this position some shells will misinterpret a command as a label.
3	SUS	Each option name should be a single alphanumeric character (the **alnum** character classification) from the portable character set. The **-W** (capital-W) option shall be reserved for vendor options.	This only applies to short-option names. The clause about the **-W** option is only of concern when modifying utilities defined by SUS or when creating/modifying a

1 Single Unix Specification ...

#	Source	Guideline	Notes
		Multi-digit options should not be allowed.	utility that may be standardized in the future .
4	SUS	All options should be preceded by the '`-`' delimiter character.	This only applies to short-options.
5	SUS	Options without option-arguments should be accepted when grouped behind one '`-`' delimiter.	This only applies to short-options.
6	SUS	Each option and option-argument should be a separate argument.	This only applies to short-options. Specifically, this doesn't apply when a long-option is separated from the option-argument with '`=`' (equals).
7	SUS	Option-arguments should not be optional.	
8	SUS	When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using commas within that argument or blanks within that argument to separate them.	This is interpreted to mean that if an option-argument contains multiple values, it may accept commas or blanks as the separator, but not both. Commas are the preferred separator to avoid quoting issues, except in cases where commas would be expected to be part of the option-argument string. Whether a utility accepts either commas or spaces must be made clear in the documentation for each utility. It is up to the utility to parse a comma-separated list itself because `getopt()` just returns a single string. This situation was retained so that certain historical utilities would not violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated list, it should avoid generating non-integer values through one of the means that is influenced by setting the `LC_NUMERIC` variable (such as `awk`, `bc`, `printf` or `printf()`). The function `getsubopt(3c)` is provided to parse many option-argument formats.
9	SUS	All options should precede operands on the command line.	There exists a class of utilities where the application of this guideline is impractical. Specifically, these are utilities where differing option settings may be desired for each operand. The compilation linking tools (ld, *et al.*) are

#	Source	Guideline	Notes
			an example of this. In such cases, this guideline should be waived (but only for this reason).
10	SUS	The argument `--` should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the '`-`' character. The `--` argument should not be used as an option or as an operand.	Applications calling any utility with a first operand starting with `-` should usually specify `--` , as indicated by Guideline 10, to mark the end of the options. Utilities that do not support Guideline 10 indicate that fact in the OPTIONS section of the utility description.
11	SUS	The order of different options relative to one another should not matter, unless the options are documented as mutually exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.	The order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The make utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the -f option, for example: `make -f common_header -f specific_rules target`
12	SUS	The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.	
13	SUS	For utilities that use operands to represent files to be opened for either reading or writing, the '`-`' operand should be used only to mean standard input (or standard output when it is clear from context that an output file is being specified).	Guideline 13 does not imply that all of the standard utilities automatically accept the operand '`-`' to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple '`-`' operands. It simply says that, by default, '`-`' operands are not used for other purposes in the file reading or writing (but not when using stat(), unlink(), touch, and so forth) utilities. All information concerning actual treatment of the '`-`' operand should be included in the individual utility descriptions.
14	PSARC[2]	The form "command subcommand [options] [operands]" is appropriate for grouping similar operations. Subcommand names should follow the same conventions as command names as specified in guidelines 1 and 2.	
15	GNU[3]	Long-options should be preceded by `--`	This is reworded from the GNU form to

2 Platform Architecture Committee, case PSARC/1991/031, *Getopts Usage*
3 GNU getopt_long

#	Source	Guideline	Notes
		and should include only alphanumeric characters and hyphens from the portable character set. Option names are typically one to three words long, with hyphens to separate words.	follow the conventions of this document. The original form is "Long-options consist of `--` followed by a name made of alphanumeric characters and dashes. Option names are typically one to three words long, with hyphens to separate words." Dashes and hyphens refer to the same characters.
16	GNU	"`--name=argument`" should be used to specify an option-argument for a long-option. The form "`--name argument`" should also be accepted	This is interpreted as requiring that both forms must be accepted. An implication is that a long-option argument is still subject to guideline 7, "option-arguments shall not be optional". The form of an option followed by an equals sign and then whitespace ("`--name=`") is interpreted to specify the null string as the argument.
17	GNU	All utilities should support two standard long-options: "`--version`" (with the short-option synonym "`-V`") and "`--help`" (with the short-option synonym "`-?`").	A utility may claim clip conformance if it is unable to use "`-V`" as the short-option synonym for "`--version`" due to its prior assignment. A short-option synonym for "`--version`" must still be provided. See sections 6.1 and 6.2 below for descriptions of the output of these options. Both of these options stop further argument processing when encountered and after displaying the appropriate output, the utility successfully exits.
18	CLIP[4]	Every short-option should have exactly one corresponding long-option and every long-option should have exactly one corresponding short-option.	Synonymous options may be allowed in the interest of compatibility with historical practice or community versions of equivalent utilities. There are a few cases where synonymous short-options have been created by POSIX in the interest of merging together BSD and SysV utilities. Also, Sun has several utility that already use -h to get help; if they are converted to CLIP, "`-h`", "`-?`" and "`--help`" should clearly be allowed as synonyms. Such synonyms should always be justified to reviewers of the utility and (if appropriate) in the documentation.
19	CLIP	The short-option name should get its name from the long-option name	This algorithm is derived from the *Java Look and Feel Design Guidelines* section

http://www.gnu.org/manual/glibc-2.2.5/html_node/Getopt-Long-Options.html

4 Defined in this document.

#	Source	Guideline	Notes
		according to these rules: 1. Use the first letter of the long-option name for the short-option name. 2. If the first letter conflicts with other short-option names, choose a prominent consonant. 3. If the first letter and the prominent consonant conflict with other short-option names, choose a prominent vowel. 4. If none of the letters of the long-option name are usable, select an arbitrary character.	about mnemonics.
20	CLIP	If a long-option name consists of a single character, it must use the same character as the short-option name.	Single character long-options should be avoided. They are only allowed for the exceptionally rare case that a single character is the most descriptive name.
21	CLIP	The subcommand in the form described in guideline 14 is generally required. In the case where it is omitted, the command shall take no operands and only options which are defined to stop further argument processing when encountered are allowed. Invoking a command of this form without a subcommand and no arguments is an error.	This guideline is provided to allow the common forms "`command --help`", "`command -?`", "`command --version`", and "`command -V`" to be accepted in the command-subcommand construct. The acceptance of other options without a subcommand is strongly discouraged. Under no circumstances will a command of the form described in guideline 14 be allowed to accept an operand when no subcommand is present, as that form is ambiguous.

6. Defined Option Output

If a utility has subcommands (see guideline 14), the options described in this section may be used instead of the subcommand name, as well as after the subcommand name. Differing information may be returned in either case.

6.1 Version Output (--version / -V)

The following text is freely adapted and reduced from the *GNU Standards for Command Line Utilities*. Direction from that reference that is only applicable to independently delivered components that don't comprise a product has been deleted.

This option should direct the program (utility) to print information about its name, version, origin and legal status, all on standard output, and then exit successfully. Other options and arguments should be ignored once this option is seen, and the program should not perform its normal function. The first line is meant to be easy for a program to parse; the version string proper starts after the last space. No other assumptions should be made about the generic ability to parse the version output. In addition, it contains the canonical name for this program, in this format:

```
cc 5.2
```

The program's name should be the “basename” of the utility as documented. The idea is to state the standard or canonical name for the program, not its file name. There are other ways to find out the precise file name where a command is found in PATH. If the program is a subsidiary part of a larger package or product, mention the name in parentheses, like this:

```
cc (Sun WorkShop 6 update 1) 5.2
```

If the package has a version number that is different from this program's version number, you can mention the package version number just before the close-parenthesis. The following line, after the version number line, should be a copyright notice. If more than one copyright notice is called for, put each on a separate line. Here's an example of output that follows these rules:

```
cc (Sun WorkShop 6 update 1) 5.2
Copyright 2002 Sun Microsystems, Inc.
All rights reserved.
Use is subject to license terms.
```

The exact form of the copyright text is not defined by this document, but rather is defined (and redefined) by the Sun Legal Department. The above form is the form specified by the Sun Legal Department as of the date of this document.

In contrast with the GNU guidelines, under no circumstances should the output contain a list of any Sun authors of the program. Should a imported utility contain a list of external author attributions, that list should be retained in an unmodified form.

6.2 Help Output (--help / -?)

This option should direct the program to print information about the usage of the utility and both the subcommands and options that it supports, all on standard output, and then exit successfully. Other options and arguments should be ignored once this option is seen, and the program should not perform its normal function.

The traditional usage message (SYNOPSIS from the manual page or equivalent) should be displayed first. Each distinct usage of a particular utility should be listed on a separate line. The first should be prefixed by "`Usage: `" while subsequent ones should be prefixed with "`  or: `". An example of this is:

```
$ foo -?
Usage: foo -x -y -z operand
  or: foo -w [-q arg]
```

```
   or: foo -v
   or: foo -?
```

Note that the `-v` and `-?` options should only be on separate SYNOPSIS lines when the utility would have more than one SYNOPSIS line for other reasons. In general, the `-v` and `-?` options are not exclusive of other options, they just have the special definition of stopping further processing of the command line.

Utilities without subcommands, should then display a short description of the utility and a list of all its options and what they are for.

If the help option appears in the absence of a subcommand, the utility will print the traditional usage message, a short description of the utility, and a short description of what each subcommand does. In this case, the options to the subcommands are not described.

If the option appears after the subcommand name, the utility will print the traditional usage messages of the subcommand, a short description of what the subcommand does and a list of all its options and what they are for.

Examples:

```
$ sscontrol --help
Usage: sscontrol SUBCOMMAND ARGUMENTS

Control a ServiceSpace server.

The accepted values for SUBCOMMAND are:

shutdown    Stops a ServiceSpace instance.
startup     Starts a ServiceSpace instance.

For more information, see sscontrol(1M).


$ sscontrol startup --help
Usage: sscontrol startup [OPTIONS...]

Starts a ServiceSpace server.

-b bundle, --bundle=bundle
                    Installs and starts bundles (none by
                    default)

-k, --check-bundles Check validity of bundles

-r path, --bundle-prefix=path
                    Prefix used to locate bundles (defaults to
                    $jesInstall/bundles/)

-u, --no-update     Does not update bundles on launch

-?, --help          Display this help list

-V, --version       Display program version information

For more information, see sscontrol(1M).
```

The purpose of the help output is to indicate anything special about the arguments that the utility expects, and so the output should not enumerate the possible option-arguments and operands.

Note that this output is intended to be a subset of the output generated by the GNU ARGP argument parser, as this provides an informal expectation for this information.

7. Conventional Names

CLIP seeks to promote consistency across utilities. An important aspect of this goal is for utilities to use consistent names for their option names and subcommands. The benefit of this is that users will come to associate certain results from particular options and subcommands, which will allow them to learn how to use new utilities more easily.

Utilities are encouraged to use names from this set, but not at the expense of ignoring terms commonly applied to the specific application area or “previous art”.

The following two subsections provide a set of option names and subcommands, and describes their meanings. These lists should be expanded as more experience is gained so there can be more consistency across utilities.

A utility may claim CLIP conformance if it uses names other than those suggested.

7.1 Conventional Option Names

If a utility supports an option that corresponds to one of the ones listed below, then that utility should use the specified long-option name. Similarly, a utility should not use one of these long-option names for a different meaning. If a utility uses one of the long-option names listed in the table below, CLIP recommends that the corresponding short-option name listed in this table be used.

Long-option Name	***Short-option Name***	***Meaning***
`--dry-run`	`-n`	Report what would have been done without actually doing it.
`--recursive`	`-R`	Recursively perform the operation on all members of a directory tree or other hierarchy.
`--hostname=`name	`-h` name	Specify a hostname.
`--username=`name	`-u` name	Specify a user name.
`--groupname=`name	`-g` name	Specify a group name.
`--quiet`	`-q`	Show no status information while performing the operation
`--verbose`	`-v`	Show additional status information while performing the operation.

7.2 Conventional Subcommand Names

If a utility supports one of the operations listed below, then that utility should use the subcommand name listed. A utility should not use one of these subcommand names to specify a different operation.

Subcommand	***Meaning***
`add`	Adds a reference to an existing item, specified by the operands, to an object.
`create`	Creates a new item(s) specified by the operand(s).
`delete`	Deletes an item(s) specified by the operand(s). This is the inverse of the "`create`" subcommand.
`list`	Prints information about the object(s) specified by the operands, or all objects if no operands are specified in the command.
`modify`	Modifies one or more aspects of one or more objects specified by the operand(s).
`remove`	Removes a reference to an item(s), specified with the operand(s), from an object. This is the inverse of the "`add`" subcommand.

The following names should never be used as subcommand names. They are overly redundant as synonyms for long-options of the same name, but using them for another meaning would be confusing.

Subcommand	***Meaning***
`help`	Redundant to "`--help`" or confusing with another interpretation.
`version`	Redundant to "`--version`" or confusing with another interpretation.

8. Manual Pages

Manual pages need two alterations to properly document the additional features defined by CLIP.

8.1 Long-Option Names

When documenting the options used by a utility, all synonymous forms of the option should be listed on individual lines, as shown in the example below, with the short-option synonym listed first. When a synonym starts with a different letter than the initial one, it should be repeated by itself later on in the options list. Options should be sorted alphabetically by the first option in each group. When options with long names accept an option-argument, they should always be shown with the "=". Also note that when an option accepts an option-argument, that argument is listed with all of the option names. Finally note that the `-V` and `-?` options are not given a synopsis line of their own; their definition of stopping further processing of the command line is sufficient. The following example demonstrates all of these concepts:

```
SYNOPSIS

     util [-a][-n][-q][-S suf][-V][-?] operands
```

...

```
OPTIONS
```

```
     This utility is CLIP compliant. The following options are
     supported:

     -a
     --text
          Ascii text mode: convert end-of-lines using local
          conventions. This option is supported only on some non-
          UNIX systems. For MSDOS, CR LF is converted to LF when
          compressing, and LF is converted to CR LF when
          decompressing.

     -n
     --name
          When compressing, do not save the original file name and
          time stamp by default. (The original name is always saved
          if the name had to be truncated.) When decompressing, do
          not restore the original file name if present.

     -q
     --quiet
          Suppress all warnings.

     -S suf
     --suffix=suf
          Use suffix suf instead of qy. Any suffix can be given,
          but suffixes other than z and qy should be avoided to
          avoid confusion when files are transferred to other
          systems.

     --text
          Ascii text mode. See -a.

     -V
     --version
          Display version information. (Stops interpretation of
          subsequent arguments.)

     -?
     --help
          Display help information. (Stops interpretation of
          subsequent arguments.)
```

The SYNOPSIS line should not be modified to show the long-option form. Including the short-option equivalents in the OPTIONS section is sufficient.

8.2 Subcommands

Utilities with subcommands should list each subcommand usage in the SYNOPSIS section on a separate line, and document the subcommands and their options in the DESCRIPTION section. Note that the `-V` and `-?` options are given separate lines in this case, otherwise they would need to be replicated on each line. The following example demonstrates these ideas:

```
SYNOPSIS

     superutil add [ -asu ] [ -n name ] file
     superutil remove [ -fr ] name ...
     superutil -V
     superutil -?
...
DESCRIPTION
     Manages named references to files.

     The add subcommand adds a reference to the file with the
     specified name.

     The remove subcommand removes the named reference(s).

...
OPTIONS
     The following options are supported:

     -a
     --absolute
          Store an absolute path name.

     -f
     --force
          Do not warn if the names are not present.

     -n name
     --name=name
          Specify the name for the reference. Defaults to "ref" and
          a unique number (e.g. ref2345) if not specified.

...
```

9. Disallowed Constructs

As mentioned above, one of the driving forces of the CLIP Specification is increased compatibility with the freeware communities. However, several behaviors have been implemented by the freeware community that have unacceptable long-term consequences and are explicitly not allowed in Sun software.

Most of these exclusions do not apply to Sun supplied software classified as External. However, Sun software that is originally classified as External but that expects to migrate to an Evolving or other public classification should strongly consider disallowing these options at initial delivery.

9.1 Non-POSIX Option/Operand Interleaving

The default behavior of GNU `getopt()` and the only behavior of GNU `getopt_long()` is to not enforce guideline 9, allowing the interleaving of options and operands. There are cases where this can result in the ambiguous interpretation of command lines and is explicitly not allowed under these guidelines.

It should be understood that there exists a class of utilities where the application of guideline 9 and/or 11 is impractical. Specifically, these are utilities where differing option settings may be desired for each operand. The compilation linking tools (ld) are an example of such utilities. In such cases, this guideline should be waived (but only for this reason). Such a utility can't claim either getopt or CLIP conformance, but it can describe itself as "conforming to the getopt (or CLIP) guidelines with the exception of guidelines 9 and 11".

9.2 Long-Option Completion

GNU ARGP provides that "Long option names can be abbreviated as long as those abbreviations are unique among long options". This can lead to the situation that the addition of a new option to a utility can cause the failure of existing scripts. Consider a utility which provides the "`--do-this`" option. A script could abbreviate this as "`--do`". Adding the "`--do-that`" option would cause "`--do`" to become ambiguous and hence fail.

Long-option abbreviation/truncation is explicitly not allowed.

9.3 Single Dash Long-Options

GNU ARGP can be used in a mode so that long-options may be delimited by a single '`-`': "Long options names may be delimited by a single '`-`'". This can lead to ambiguity between a long-option and the aggregate of a number of short-options as provided by guideline 5.

Single dash long-options are explicitly not allowed.

9.4 Optional Option-Arguments

GNU getopt_long provides for optional option arguments. Although this is unambiguous in the case where the option is directly followed by an equals sign it becomes ambiguous in the case (also supported by getopt_long) where the option and its (potential) argument are separated by a space. Consider the option "`--optional-arg`". While the constructs "`--optional-arg= foo`" and "`--optional-arg=foo bar`" are unambiguous, the construct "`--optional-arg foo`" is ambiguous as to whether foo is option-argument or the first operand. As stated in notes accompanying guideline 16, the form "`--optional-arg= `" shall be interpreted as specifying the null string as an option-argument, not as no option-argument.

Optional option-arguments are explicitly not allowed.

9.5 POPT Configuration

POPT (used by GNONE, RedHat rpm and other utility sets) provides an option aliasing facility. It lets the user specify options that `popt()` expands into other options when they are specified. Aliases are specified in two places - `/etc/popt` and `$HOME/.popt`. This is more "rope" than is acceptable to Sun because it can be easily manipulated to circumvent system security. This feature of POPT has raised sufficient concern that a separate case has been spawned to clarify the disposition of POPT in Solaris. (PSARC/2003/XXX).

Until the disposition of POPT is clarified, its use should be limited and any use should disable the aliasing facility.

9.6 Authoring Credits

GNU allows (and perhaps encourages) authoring attributions in version output (and perhaps elsewhere). Unless legal obligations require it, Sun utilities shall not contain or display authoring attributions (even in source code). Note however, that it is generally a legal requirement to retain all existing attributions in imported software. In summary, Sun (generally) should neither add or delete such credits.

10.Argument Parsing

The existing `getopt(3c)` and `getopts(1)` facilities will be enhanced to be able to parse CLIP comformant argument arrays (`argv[]`). The will be done by expanding the syntactic specification of an option in the optstring parameter. Currently, the syntax of an option specification in optstring is:

c [`:`]

where the character *c* is the short-option character and the optional literal colon specifies if the option requires an option-argument. This syntax will be expanded to:

c [`:`] [`(` *name* `)` ...]

where the character c and the literal colon are defined as before. These may now be followed by a series of zero or more *name* character strings enclosed in literal parentheses. The name character strings are the long-option names to be aliased to the short-option name. A couple of notes:

1. The use of more than one long-option name as an alias for a short-option name violates the CLIP specification. However, it is expected that enough exceptions will be granted to this rule that it is appropriate to require the function to accept multiple long-options.
2. Although never allowed by the specification, this change will eliminate the ability of `getopt(3c)` to accept the left parentheses as a short-option name. The colon character is already not accepted.
3. The use of the question mark character will now become common in the `optstring` parameter (so that the long-option name alias --`help` can be associated with it). It will be documented as a Solaris extension to the `getopt()` function, but its use for anything other than a synonym for the help long-option will not be accepted. Note that `getopt(3c)`will also return a question mark character in response to a number of argument error conditions. This ambiguity can be resolved by checking the `optopt` variable. Generally, when the questionmark is due to an error condition, a

usage statement should be displayed rather than the full help message.

The possibility of introducing a new command line argument parsing function modeled after the GNU `getopt_long()` function was considered:

```
int getopt_clip (int argc, char *const *argv, const
char *shortopts, const struct option *longopts, int
*indexptr)
```

Although this approach had some minor desirable features, the awkwardness in implementing a similar interface for shell scripts (as `getopts(1)`) made it far less desirable than simply enhancing the existing `getopt(3c)` interface.

The proposed manual page for `getopt(3c)` is available.

11.Future Directions

As more experience is gained, the lists of conventional names provided in section 7 should be expanded.