C Utilities Documentation

Eric Vought
QLUE Consulting, Inc.

August 22, 1998

Contents

1 **Introduction** **4**

- 1.1 Library and Source Code Organization . . . 5
- 1.2 Test Framework . . . 6
- 1.3 Conventions . . . 7
- 1.4 Licensing . . . 7
  - 1.4.1 Preamble . . . 7
  - 1.4.2 Definitions . . . 8
  - 1.4.3 License . . . 8

2 **System Call and Standard Library Replacement Functions** **11**

- 2.1 Description . . . 11
- 2.2 Interface Summary . . . 12
- 2.3 Member Descriptions . . . 13
  - 2.3.1 bzero() . . . 13
  - 2.3.2 memcmp() . . . 13
  - 2.3.3 Δ usleep() . . . 13
  - 2.3.4 snprintf() . . . 14
  - 2.3.5 vsnprintf() . . . 15
  - 2.3.6 asprintf() . . . 15
  - 2.3.7 vasprintf() . . . 15
  - 2.3.8 strcasecmp() . . . 16
  - 2.3.9 strncasecmp() . . . 16
  - 2.3.10 strdup() . . . 16
  - 2.3.11 strspn() . . . 16
  - 2.3.12 strcspn() . . . 16
  - 2.3.13 strstr() . . . 17
- 2.4 Notes . . . 17
  - 2.4.1 Future Additions . . . 17

3 Hash Table 18
3.1 Description 18
3.2 Interface Summary 18
3.3 Member Descriptions 19
3.3.1 new() 19
3.3.2 clear() 19
3.3.3 delete() 20
3.3.4 add() 20
3.3.5 set() 20
3.3.6 remove() 20
3.3.7 exists() 21
3.3.8 fetch() 21
3.3.9 resize() 21
3.3.10 info() 21
3.3.11 check_invariants() 22
3.3.12 Hash_Table 22
3.3.13 DEFAULT_INITIAL_SIZE 23
3.4 Notes 23
3.4.1 General 23
3.4.2 Hash Algorithm 23
3.4.3 Lookup Caching 23

4 Net Utils 24
4.1 Description 24
4.2 Interface Summary 24
4.3 Member Descriptions 25
4.3.1 xsocket() 25
4.3.2 xbind() 25
4.3.3 xlisten() 25
4.3.4 xaccept() 25

5 String Utils 26
5.1 Description 26

5.2 Interface Summary 26
5.3 Member Descriptions 27
    5.3.1 xsnprintf() 27
    5.3.2 xstrncpy() 28
    5.3.3 xstrdup() 28
    5.3.4 xstrcasecmp() 28
    5.3.5 xstrncasecmp() 28
    5.3.6 split() 29
    5.3.7 rspn() 29
    5.3.8 rcspn() 29
    5.3.9 triml() 29
    5.3.10 trimr() 30
    5.3.11 trim() 30
    5.3.12 glue() 30
    5.3.13 toupper 30

6 Utilities **31**
6.1 Interface Summary 32
6.2 Member Descriptions 33
    6.2.1 bool 33
    6.2.2 fatal_error() 33
    6.2.3 daemonize() 33
    6.2.4 xfork() 34
    6.2.5 xmalloc() 34
    6.2.6 xcalloc() 34
    6.2.7 xrealloc() 34
    6.2.8 xmemdup() 34
    6.2.9 scalloc() 35
    6.2.10 mda_malloc() 35
    6.2.11 xgetlogin() 35
    6.2.12 is_daemon_proc 35
    6.2.13 fatal_error_log_level 35

1. Introduction

This document describes how to use and extend the QCI C Language Utilities Library.

This library is an in-house utilities package used by QCI for internal and external software development. The library contains standard data structures, wrappers over standard *libc* functions and POSIX system calls, and collections of commonly used but difficult to implement algorithms.

The C Utilities library was constructed to provide a solid, portable, consistent, and well tested foundation for rapid deployment of quality software products. The library is intended to be used in three overlapping roles:

1. The library provides a portability layer, providing or replacing useful system/library calls that are not present or do not function correctly on most platforms. On platforms where the system/library routines are present and work correctly, the native routine is used directly with no additional overhead.
2. The library provides a set of high quality, pre-tested components that can shorten development/testing time and raise the overall quality of the deployed code.
3. The library provides a set of generic, general purpose, structured components that can be used during rapid prototyping or in the early developmental stages of a project in order to test and finalize the internal interfaces before substituting optimized, special-purpose code.

Some of the system call replacements were borrowed from public domain sources. In particular, this product includes software developed by the Apache Group for use in the Apache HTTP server project (http://www.apache.org/). Only software that would not limit the redistributability of this library was considered for inclusion, and all included software has been examined and modified as necessary to meet QCI standards.

Each of the modules in this package is written to stringent coding standards and exhaustively tested, both using standard black box testing frameworks and white box code analysis tools. The white box analysis tool (currently X/Open's CI-Report) is used to locate potentially dangerous or ill-advised practices which can and do work when tested under controlled conditions but may behave unpredictably under unusual circumstances or as the surrounding code is modified. Additionally, the white box testing tool is used to verify that all of the code in the utilties library is standards conformant and readily portable.

Studies with SmallTalk have shown that a faster development process using *well-designed*, pre-packaged, and pre-tested tools in an interpreted language allows more time for careful profiling and often results in faster performance than code written in tool-poor compiled environments (e.g.: C/C++).[1] This library is an attempt to apply a similar approach to development in C.

As compilers and optimization techniques improve it has been found that removal of function overhead, elimination of subexpressions and other fine tuning techniques are best left to a good compiler given appropriate hints (use of const, defining utility functions inline, etc.). In most cases in an application, the performance overhead of using library defined functions over hand-coded routines is negligible. The overhead of a function call scarcely matters when the next call blocks on an I/O request or if the utility function adds a small constant overhead to an already time consuming internal process.

Even within critical sections of code, like internal processing loops and lengthy non-interactive calculations, the utilities library routines can still be appropriate during the development phase. After the code is complete, tested, and functional, a profiler can be used to point out the most critical sections of code which can then be re-implemented— leaving most of the library calls in place.

1.1 Library and Source Code Organization

The library is organized in a highly modular/structured manner— as nearly object oriented as can be cleanly accomplished in C. Modules that represent ADTs are organized as a single central structure definition with a collection of functions that act on that definition. All functions which are not involved in actually allocating the structure take a handle type as their first argument. All exported symbols begin with either the full name of the module or a module specific abbreviation [2]. As much of the underlying structure as is possible is masked from the caller by the module's functions. All operations which modify the structure and most operations that read the structure are hidden behind macros or functions. No global variables (other than constants) are used.

The library makes no attempt to be fully object-oriented. We believe that a clean structured design is more important than an attempt to twist the C language in a direction that it was not intended to go. In particular, there is no attempt to simulate operator overloading, inheritance, or virtual methods. Modules (e.g.: UTILITY) which do not operate on a central structure are presented as a collection of related functions.

For optimization purposes, for code clarity, and in order to maximize the amount of source code checking that the compiler can perform, compatibility with Common C (K&R C) has not been retained. This code is strictly ANSI C compliant

---

[1] I need to find citations for some of these studies.

[2] Except for functions that wrap system or standard library routines

and uses several ANSI C specific features:

The ANSI *const* keyword is used to flag function arguments that are not modified in the function body. This may be used by the compiler to eliminate subexpressions and avoid parameter passing overhead. The *const* keyword is also used for global constants in preference to #defines. Unlike macros, constant identifiers can be type checked by the compiler at no additional performance penalty. For compatibility with K&R C, const may be #defined to an empty token.

ANSI prototypes are used throughout the system. All functions, including internal routines, are declared at the top of the file in which they appear. This makes browsing, understanding, and maintaining the source code easier.

ANSI-style function pointer typedefs are used in several modules. ANSI allows function pointer typedefs to contain type information that can be checked by the compiler and argument names that inform the reader as to the intended purpose of the arguments.

All header files are protected from multiple inclusion by conditional compilation statements. Every module that directly uses declarations from another header file will include that header even if it is already included indirectly from a common header. This is intended to make it easier for maintainers to pick out symbol dependencies.

1.2 Test Framework

A unit test framework based on the OpenGroup[3]'s Test Environment Toolkit (TET) is used. TET provides a simple, standard, and portable harness for executing tests and recording the results. The tests are provided in such a way that they can be run either under the TET or as standalone executables when the library is distributed in binary form. Compiling the tests requires the TET libraries and header files which are freely available from The OpenGroup.

All testing information is stored under the testing directory of the distribution. Each module has a separate directory underneath `testing` that contain tests for that unit. Makefiles inside the individual test directories will build one or more executable programs. Configuration files under `testing` tell TET how to run individual tests or pre-defined sets of tests. See the TET documentation for more information.

Simply executing the generated programs manually will run the tests in standalone mode and dump the results in a file called tet_xres. When interpreting the results, care must be taken to interpret the results of the exception tests included with each module. Many of these tests are designed to verify that the unit will detect critical error conditions and abort smoothly. TET will register such (correct) behavior as a failure. When this is a case, the test case will place

---

[3] Formerly X/Open

a comment in the result file just before it aborts describing the error expected. These tests are always in a separate file called exception_tests.

Functions which are simple wrappers over standard functions, such as xstrdup, xmalloc, etc., are not tested at this time.

1.3 Conventions

The following comments apply to PostScript or PDF documentation generated directly from the original LaTeX source. Appearance in HTML or other generated formats may be different.

Module names are printed in small caps, LIKE THIS. Library symbol names (e.g.: functions, type names, and variables) are printed in sans serif, like_this. Function names will always have a trailing pair of parentheses, like_this(). Standard C Library or POSIX symbol names are printed in a sans serif slanted font, *like_this*.

Interface synopses are shown in a fixed width font.

Example code is shown in a floating, captioned box in a fixed-width font with line numbers added at increments for reference purposes.

Notes about probable future changes in the API are marked with a bold-face, capital delta (**Δ**).

When a module symbol is referenced inside the documentation for that module, the module specific prefix is dropped for brevity. The abbreviations appear at the top of each module description. All symbols referenced from other modules or symbols which do not use the standard prefix will appear with the full name as defined in C.

Because C and UNIX are both case-sensitive, the exact case is always used in identifiers, even where proper grammar demands that an uppercase character be used, such as at the beginning of a sentence.

1.4 Licensing

This license agreement is based on the Public Domain "Artistic License".

1.4.1 Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

1.4.2 Definitions

Package refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

Standard Version refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

Copyright Holder is whoever is named in the copyright or copyrights for the package.

You is you, if you're thinking about copying or distributing this Package.

Reasonable Copying Fee is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

Freely Available means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

Template means a generic file which is intended to be tailored for a specific use through mechanical textual substitution.

Instantiation means the process of mechanical textual substitution which tailors a generic Template for a specific use.

1.4.3 License

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

   (a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   (b) use the modified Package only within your corporation or organization.
   (c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
   (d) make other distribution arrangements with the Copyright Holder.

4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
   (a) distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
   (b) accompany the distribution with the machine-readable source of the Package with your modifications.
   (c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
   (d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's object code within an executable of yours (by linking); this shall be construed as a mere form of aggregation.

6. Source code modules that are created by Instantiating the Template Files included in this package are considered to be part of this package and are subject to the copyright of this package.

7. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.

8. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
9. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

2. System Call and Standard Library Replacement Functions

2.1 Description

This section describes the system calls or standard library functions which are transparently replaced by this library if the configuration scripts determine that they vare not present or do not function on a particular system. Replacement functions come in three categories:

1. Portions of the 1998 Single UNIX Specification (UNIX 1998) that are not yet implemented by many systems.
2. functions which are part of BSD, SVID4 or other standards that have not been added to the POSIX standards suite but which are useful to many application developers.
3. A handful of functions that have been or will be added that represent non-standard extensions provided by one or more vendors which we feel should become standardized (e.g.: asprintf, strfry).

Whenever a source file uses a function which could potentially be replaced by this library, the appropriate QCI utilities header should be included in addition to the standard system header. For example, if using snprintf, both string.h and qstring.h should be included. I find that this is often automatic because e.g. most files that use string functions also use convenience functions from qstring. At some point, Δ, files like qstring.h will probably publicly include the appropriate system headers so that only the QCI utilities headers need be included for commonly replaced functions.

The replacement routines are documented here for the benefit of developers on systems that do not have native versions of and documentation for them. On systems where these functions are documented, the vendor provided documentation takes precedence over this documentation. Most of the descriptions provided below are excerpts from the Linux man pages for these routines.

If you are not sure whether or not a particular routine has been replaced by this library, examine the output of configure as it runs. It will print lines like "checking for snprintf...". Alternatively, examine the generated qci_util_config.h.

2.2 Interface Summary

bzero() :

```
void bzero(void *s, int n);
```

memcmp() :

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Δusleep() :

```
void usleep(unsigned long usec);
```

snprintf() :

```
int snprintf(char *str, size_t n, const char *format, ... );
```

vsnprintf() :

```
int vsnprintf(char *buf, size_t len, const char *format, va_list ap)
```

asprintf() :

```
int asprintf(char** buffer, const char* format, ...);
```

vasprintf() :

```
int vasprintf(char** buffer, const char* format, va_list ap)
```

strcasecmp() :

```
int strcasecmp(const char *s1, const char *s2);
```

strncasecmp() :

```
int strncasecmp(const char *s1, const char *s2, size_t n);
```

strdup() :

```
char* strdup(const char* s);
```

strspn() :

```
size_t strspn(const char* s, const char* accept);
```

strcspn() :

```
size_t strcspn(const char* s, const char* reject);
```

strstr() :

```
char* strstr(const char *haystack, const char *needle);
```

2.3 Member Descriptions

2.3.1 bzero()

```
void bzero(void *s, int n);
```

The *bzero()* function sets the first `n` bytes of the byte string `s` to zero.

bzero() conforms to 4.3BSD but is non-POSIX and its use is deprecated.

Using xbzero tends to be less error prone than using memset directly, since it is relatively easy (and common[1]) to swap the second and third arguments to *memset()*. Since the arguments are of a compatible type, compilers will not catch the error, leading to obscure bugs.

2.3.2 memcmp()

```
int memcmp(const void *s1, const void *s2, size_t n);
```

memcmp() is a BSD 4.3, SVID3, and POSIX call that compares the first n bytes of the memory areas s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respec- tively, to be less than, to match, or be greater than s2.

This is a replacement function for the (few) systems that do not have *memcmp()* or for which *memcmp()* is buggy and not 8-bit clean. autoconf will run some basic tests on the local *memcmp()* implementation when the library is built.

2.3.3 Δ usleep()

```
void usleep(unsigned long usec);
```

usleep() behaves like *sleep()* except that the sleep time is measured in microseconds, not seconds. Note that the program will sleep for *at least* `usec` microseconds, but may sleep longer depending on system clock granularity, system load, etc. *usleep()* is a BSD 4.3 function that is also in the UNIX98 standard.

nanosleep() is a POSIX.1b (POSIX Real-Time Standard) function that is also available on some systems (notably Solaris 2.6, Linux, and QNX). It theoretically has much finer granularity, though it is severely limited by the system scheduler granularity on all but dedicated real-time systems. It also has the ability to easily continue a sleep that has been interupted by a signal. `usleep()`, however, has a simpler interface, is (relatively) portable, and should be sufficient for *most* uses.

---

[1] Stevens - *UNIX Network Programming, 2nd Ed.*

There is one major difficulty with using *usleep()*, however— no one can agree on the return type. Some vendors use int, some unsigned int, some void[2]. Additionally, the information conveyed by the returned value differs from implementation to implementation[3].

There were two alternatives to resolving this problem:

1. provide a function returning int, in compliance with the UNIX98 standard, and use the replacement on all platforms that have incorrect types. Manipulating system headers and macro definitions so that the new prototype, which conflicts with the system prototype, is accepted is very difficult to do correctly and consistently.
2. Use the system prototype where available and never use or assign the return value. If the native function does return something, the value will be harmlessly ignored. If the native function is void, everything will still work correctly.

We opted for the second approach. Hopefully, as the UNIX98 standard gains acceptance, vendors will start moving to the new prototype and this will become less necessary. In the meantime, this replacement will work. The replacement function is based on *select()*, which is portable, reasonably reliable, and efficient.

2.3.4 snprintf()

```
int snprintf(char *str, size_t n, const char *format, ... );
```

snprintf() is a safe version of *sprintf()* included in a recent draft ANSI C standard and in UNIX98 in order to prevent a class of serious memory errors and security holes that occur in many programs (e.g.: sendmail). *snprintf()* takes an extra argument which is the size of the target buffer. Any characters which do not fit inside the target buffer will be discarded and the number of characters written will be returned.

Even though *snprintf()* is a draft standard, several vendors already include it as part of their standard C libraries at the time of this writing[4].

The replacement snprintf function was adapted from the function used by the Apache Group in the Apache httpd server. It is a partial reimplementation of some of the sprintf internals and has several limitations with respect to formatting floating point numbers. If full compatibility with sprintf and buffer overflow safety is needed, a C library with a native implementation of snprintf should be used instead (e.g: glibc).

---

[2] UNIX98 specifies int

[3] Some implementations use the return value as an error code, others as the number of microseconds actually slept, still others the number of microseconds remaining in case the sleep was interupted by a signal.

[4] Notably the GNU libc and the Solaris 2.6 libc

2.3.5 vsnprintf()

```
int vsnprintf(char *buf, size_t len, const char *format, va_list ap)
```

vsnprintf() is a variation of snprintf that operates on a varargs list rather than a variable number of arguments. This function is useful when creating printf-like functions that wrap snprintf.

See the cautions on using *snprintf*, above.

2.3.6 asprintf()

```
int asprintf(char** buffer, const char* format, ...);
```

asprintf() is a *printf()* variant that dynamically allocates a buffer of the correct size. This is considerably safer than using *sprintf()* and more convenient than using *snprintf()* at a modest performance penalty for small strings.

The replacement function is implemented on top of *snprintf()*, therefore the cautions on using the *snprintf()* replacement function also apply. This functions is reasonably efficient for strings less than 64 bytes in length. Performance decreases linearly as the size of the string increases because the buffer must be reallocated dynamically. No space is wasted for strings less than 64 bytes.

Note that the string buffer is allocated on the free store and must be passed to *free()* at some point.

It is generally recommended that asprintf be used in non-performance critical sections of the code. It was added to this library primarily for its usefullness in formatting error messages, particularly on calls to qci_fatal_error(), where, obviously, performance is not a concern and reclaiming the memory later doesn't matter.

Δ *Note:* In the current implementation, memory allocation is *checked* using xmalloc(), and this function will never return *NULL*.

2.3.7 vasprintf()

```
int vasprintf(char** buffer, const char* format, va_list ap)
```

vsnprintf() is a variation of *asprintf()* that operates on a varargs list rather than a variable number of arguments. This function is useful when creating printf-like functions that wrap *asprintf()*.

See the cautions on using *asprintf*, above.

2.3.8 strcasecmp()

```
int strcasecmp(const char *s1, const char *s2);
```

The *strcasecmp()* function compares the two strings `s1` and `s2`, ignoring the case of the characters. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.

strcasecmp() is a BSD 4.3 function.

2.3.9 strncasecmp()

```
int strncasecmp(const char *s1, const char *s2, size_t n);
```

The *strncasecmp()* function is similar to *strcasecmp()*, except it only compares the first `n` characters of `s1`.

2.3.10 strdup()

```
char* strdup(const char* s);
```

The *strdup()* function copies `s` into a newly allocated buffer and returns a pointer to that buffer. The memory is allocated using *malloc()* and must be freed using *free()*.

2.3.11 strspn()

```
size_t strspn(const char* s, const char* accept);
```

The *strspn()* function calculates the length of the initial segment of `s` which consists entirely of characters in `accept`.

2.3.12 strcspn()

```
size_t strcspn(const char* s, const char* reject);
```

The *strcspn()* function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`.

2.3.13 strstr()

```
char* strstr(const char *haystack, const char *needle);
```

This function returns a pointer to the first occurence of the substring needle that is found in the string haystack.

Δ The current implementation is a correct but brute force approach that is inefficient for large needles and/or haystacks. This function will be heavily optimized in the next release.

2.4 Notes

2.4.1 Future Additions

The following functions are targetted for possible future versions:

1. alloca (a workable, but not as efficient replacement)
2. strfry
3. memfrob
4. Any functions that we have problems with while porting our applications.

In addition, optional malloc/free replacements that register allocations for memory leak tracking will probably be added at some point for testing on systems not supported by Purify^{TM}.

3. Hash Table

Module: HASH TABLE

Header: hashtable.h

Abbreviation: HASH

Summary: A hashtable ADT using strings as both keys and values.

Dependencies: UTILITIES

Version Documented: 3.0

3.1 Description

This module provides a hashtable abstract data type. Both keys and values are represented by C strings (*char **). Major features of this hashtable type include automatic resizing and a bulk insert mode.

The hashtable uses separate chaining (linked lists as buckets) for collision detection and a node cache for fast lookups.

3.2 Interface Summary

new() :

```
Hash_Table HASH_new(int size);
```

clear() :

```
void HASH_clear(Hash_Table ht);
```

delete() :

```
void HASH_delete(Hash_Table ht);
```

add() :

```
void HASH_add(Hash_Table ht, const char* key, const char* value);
```

set() :

```
void HASH_set(Hash_Table ht, const char* key, const char* value);
```

remove() :

```
void HASH_remove(Hash_Table ht, const char* key);
```

exists() :

```
int HASH_exists(Hash_Table ht, const char* key);
```

fetch() :

```
const char* HASH_fetch(Hash_Table ht, const char* key);
```

resize() :

```
void HASH_resize(Hash_Table ht);
```

info() :

```
void HASH_info(Hash_Table ht);
```

check_invariants() :

```
void HASH_check_invariants(Hash_Table ht);
```

Hash_Table :

```
typedef struct
{
    int size;
    int count;
    int resize_threshold;

    /* ... */
} *Hash_Table;
```

HASH_DEFAULT_INITIAL_SIZE :

```
const int HASH_DEFAULT_INITIAL_SIZE;
```

3.3 Member Descriptions

3.3.1 new()

```
Hash_Table HASH_new(int size);
```

new() allocates and returns an empty hashtable of the specified size. If size is zero, DEFAULT_INITIAL_SIZE will be used.

3.3.2 clear()

```
void HASH_clear(Hash_Table ht);
```

clear() deallocates all nodes in the specified hashtable and returns the table to an empty state.

3.3.3 delete()

```
void HASH_delete(Hash_Table ht);
```

delete() deallocates all nodes in the hashtable and deallocates the hashtable structure itself.

3.3.4 add()

```
void HASH_add(Hash_Table ht, const char* key, const char* value);
```

Add a new node to the table. This call is optimized for bulk insertions. add() will not force a resize if the resize_threshold is overshot. add() will simply chain the new node, not checking for duplicate entries.

add() is significantly more efficient for bulk inserts than set() because no lookups are performed and because resizing is postponed until the final size is known. If, for instance, a hashtable is allocated with 64 buckets and 300 keys are inserted, set would perform 300 lookups and resize the table twice, ending up with 256 buckets. Using add() followed by an immediate resize would resize once from 64 to 300 buckets and would only check for duplicate entries when there is a collison during the resize operation.

Because add() is an optimization, it can cause strange behaviors when used improperly. A series of calls to add() should be followed immediately by a call to resize(). If it is not, duplicate values will not be properly dealt with. add() is guaranteed to preserve the last value for any given key entered, but a call to remove() without a resize may inadvertantly ressurect a previously entered value.

3.3.5 set()

```
void HASH_set(Hash_Table ht, const char* key, const char* value);
```

Associate a new value with key. If the key does not exist, it will be created. If the key does exist, it will be overwritten and the previous value will be freed.

If the count of nodes is over resize_threshold, a resize will occur *before* the new value is inserted.

3.3.6 remove()

```
void HASH_remove(Hash_Table ht, const char* key);
```

Remove a key from the table and free its value. See add(), above, for additional considerations.

remove will silently fail if given a non-existant key.

3.3.7 exists()

```
int HASH_exists(Hash_Table ht, const char* key);
```

Returns one if the specified key is stored in the table, zero otherwise. If the count of nodes is over resize_threshold, a resize will occur.

3.3.8 fetch()

```
const char* HASH_fetch(Hash_Table ht, const char* key);
```

Returns the value associated with key in the hashtable. If key is not stored in the table, fetch() returns *NULL*. Note that *NULL*will also be returned if a null string was associated with the key. Use exists() to determine whether or not the key was explicitly set and see the notes on node caching, below.

If the count of nodes is over resize_threshold, a resize will occur.

3.3.9 resize()

```
void HASH_resize(Hash_Table ht);
```

Force an immediate resize of the table. resize will reallocate the hash internal structure to a size which is optimal for the number of entries currently stored. This may cause the table to shrink if the hash is underfilled. In general, the number of buckets will be set to the number of entries and the threshold will be set to twice the number of entries.

If resize() is called on an empty hash, the size will be set to DEFAULT_INITIAL_SIZE.

3.3.10 info()

```
void HASH_info(Hash_Table ht);
```

info() prints out a breakdown of the current number of buckets and distribution of nodes.

3.3.11 check_invariants()

```
void HASH_check_invariants(Hash_Table ht);
```

check_invariants() will perform a series of integrity checks on the hashtable. If any of these checks fail, check_invariants() will abort with a failed assertion.

Invariants checked include:

- The hashtable is not *NULL*.
- The bucket array is not *NULL*.
- The bucket cache is a number within the range of the bucket array.
- The resize_threshold is larger than the current size.
- The node cache is either *NULL* or points to an existing and accessible node.
- The number of nodes equals the value of the count field.

These integrity checks are expensive in terms of performance, particularly with large tables, and are intended to be used only inside a test suite.

3.3.12 Hash_Table

```
typedef struct
{
    int size;
    int count;
    int resize_threshold;

    /* ... */
} *Hash_Table;
```

The fields of *Hash_Table documented here are reasonably safe for *read-only* access. These fields should never be written to, nor should any other field of *Hash_Table be accessed in any manner.

size The current number of buckets allocated.

count The number of nodes (keys) stored in the table.

resize_threshold The number of nodes which will trigger the next automatic resize.

3.3.13 DEFAULT_INITIAL_SIZE

```
const int HASH_DEFAULT_INITIAL_SIZE;
```

The number of buckets that will be allocated if `new()` is called with a non-positive integer or a forced resize is requested on an empty hashtable. This is intended as a way of responding gracefully to an error condition and should not be treated as a feature.

3.4 Notes

3.4.1 General

If a null hashtable is passed to any Hash Table function, the call will abort on a failed assertion.

If a null key is a passed to any function, the call will abort on a failed assertion.

3.4.2 Hash Algorithm

The hashtable currently uses the hashpjw algorithm[1], a decent algorithm for hashing strings. The hashpjw algorithm is reasonably quick and will provide good distribution for well-mixed strings. The number of collisions will probably become unacceptable if many similar strings (such as pathnames with a common directory prefix) are hashed.

3.4.3 Lookup Caching

Because this table allows null values to be associated with keys, it is not possible to determine whether a key was associated with a null value or was not present in the table from the result of `fetch()`. Additionally, `set()` will always create a new node if one doesn't already exist and overwrite the existing node if already present with no additional feedback. It is common practice, therefore, to call `exists()` to determine if a key is present just before calling one of `set()`, `fetch()`, etc. If the table is crowded, a search for an existing node may have to walk the length of a long bucket list before it fails.

Any call resulting in a table lookup (e.g.: exists) will cache the bucket and node (if any) found. If the next call is searching for the same key, the cached values will be used without performing another search.

---

[1] Dragon Book, p 436

4. Net Utils

Module: NET UTILS

Header: net_util.h

Abbreviation: NUTIL

Summary: A collection of wrappers around standard network communication routines.

Dependencies: UTILITIES

4.1 Description

This module is a place to store common routines and declarations used in network communications (socket, XTI, etc.). At the moment, it mainly contains wrapper functions that perform additional error checking over the standard library calls.

Some of the wrapper functions below (e.g.: xsocket() will exit when an error occurs). In many cases, especially during the early stages of a project, this is exactly what is required and is considerably safer than ignoring the result of the call. It is expected that these wrappers would be slowly replaced with specialized error handling code over the lifetime of a project.

If a particular system does not define the POSIX *socklen_t* type, this module will supply an appropriate typedef.

4.2 Interface Summary

xsocket() :

```
int xsocket(int family, int type, int protocol);
```

xbind() :

```
void xbind(int fd, const struct sockaddr *addr, socklen_t addr_len);
```

xlisten() :

```
void xlisten(int fd, int queue_length);
```

xaccept() :

```
int xaccept(int fd, struct sockaddr *addr, socklen_t *addr_len);
```

4.3 Member Descriptions

4.3.1 xsocket()

```
int xsocket(int family, int type, int protocol);
```

xsocket() calls *socket()*, checks the result, and calls QCI_fatal_error() with a message if the socket cannot be created.

4.3.2 xbind()

```
void xbind(int fd, const struct sockaddr *addr, socklen_t addr_len);
```

xbind() calls *bind()*, checks the result, and calls QCI_fatal_error() with a message if the port cannot be bound.

4.3.3 xlisten()

```
void xlisten(int fd, int queue_length);
```

xlisten() calls *listen()*, checks the result, and calls QCI_fatal_error() with a message if the call fails.

4.3.4 xaccept()

```
int xaccept(int fd, struct sockaddr *addr, socklen_t *addr_len);
```

xaccept() calls *accept()*, and checks the result. If the error is transient and recoverable, then the call is retried, otherwise QCI_fatal_error() is called with an error message.

5. String Utils

Module: STRING UTILS

Header: qstring.h

Abbreviation: QSTR

Summary: Simple string manipulation routines and wrappers over standard library string functions.

Dependencies: UTILITIES

5.1 Description

This module is a collection of relatively simple string manipulation routines and safe wrappers over string manipulation routines included in the standard C library. Also included are routines which appear in the standard libraries on some platforms but are not standard.

All of the routines below depend on the QCI versions of the standard memory allocation routines and will exit with a fatal error when memory allocation fails.

Two routines, rspn() and rcspn() complement the standard functions, *spn()* and *cspn()*. triml(), trimr(), and trim() provide a set of routines for removing whitespace or other padding characters from the ends of strings.

5.2 Interface Summary

xstrncpy() :

```
char* xstrncpy(char* dest, const char* src, size_t n);
```

xstrdup() :

```
char* xstrdup(const char* s);
```

xstrcasecmp() :

```
int xstrcasecmp(const char* s1, const char* s2);
```

xstrncasecmp() :

```
int xstrncasecmp(const char* s1, const char* s2, size_t n);
```

split() :

```
int QSTR_split(const char* string, int delim, char*** list);
```

rspn() :
`size_t QSTR_rspn(const char* string, const char* accept);`

rcspn() :
`size_t QSTR_rcspn(const char* string, const char* reject);`

triml() :
`char* QSTR_triml(const char* string, const char* trimchars);`

trimr() :
`char* QSTR_trimr(const char* string, const char* trimchars);`

trim() :
`char* QSTR_trim(const char* string, const char* trimchars);`

glue() :
`char* QSTR_glue(int num, ...);`

toupper() :
`char* QSTR_toupper(const char* string);`

5.3 Member Descriptions

5.3.1 xsnprintf()

```
int xsnprintf(char* str, size_t n, const char* format, ...);
```

snprintf() is a safe version of *sprintf()* included in a recent draft ANSI C standard in order to prevent a class of serious memory errors and security holes that occur in many programs (e.g.: sendmail). *snprintf()* takes an extra argument which is the size of the target buffer. Any characters which do not fit inside the target buffer will be discarded and the number of characters written will be returned.

Even though *snprintf()* is a draft standard, several vendors already include it as part of their standard C libraries at the time of this writing[1]. xsnprintf() allows code to use *snprintf()* where it is available.

If the symbol HAVE_SNPRINTF is defined, the a macro will be defined by this header mapping xsnprintf() to *snprintf()*. Otherwise, xsnprintf() will be defined as a function that ignores the size argument and passes its remaining arguments to *sprintf()*.

Obviously, xsnprintf() adds no additional safety when used on systems which do not have *snprintf()*. It is not possible to implement *snprintf()* in a portable manner without reimplementing the internals of *sprintf()*. It is recommended that glibc be installed and used on systems that do not yet have *snprintf* until the remaining vendors catch up.

---

[1] Notably the GNU libc and the Solaris 2.6 libc

5.3.2 xstrncpy()

```
char* xstrncpy(char* dest, const char* src, size_t n);
```

This function is the same as *strncpy()* except that the target buffer is always null terminated.

5.3.3 xstrdup()

```
char* xstrdup(const char* s);
```

xstrdup() wraps *strdup()*, handling any memory allocation errors encountered by calling QCI_fatal_error(). If xstrdup() is called with *NULL*, *NULL* will be returned.

5.3.4 xstrcasecmp()

```
int xstrcasecmp(const char* s1, const char* s2);
```

strcasecmp() is a BSD 4.3 function which compares the two strings s1 and s2, ignoring the case of the characters. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

Since *strcasecmp()* is not required by either ANSI or POSIX, this library provides xstrcasecmp(). If the symbol HAVE_STRCASECMP is defined, xstrncasecmp() will be defined as a macro which calles the BSD function. Otherwise, it will be a function which provides the necessary functionality.

xstrcasecmp() first converts both strings to uppercase, then compares them normally using *strcmp()*. This is done for implementation simplicity and reliability but is not likely to be as fast as the system library implementations found on some systems.

See also xstrncasecmp(), below.

5.3.5 xstrncasecmp()

]

```
int xstrncasecmp(const char* s1, const char* s2, size_t n);
```

xstrncasecmp() is identical to xstrcasecmp() except that it only compares the first n digits of s1 and s2.

5.3.6 split()

```
int QSTR_split(const char* string, int delim, char*** list);
```

This routine searches the target string for occurences of the specified delimiter, splits the string based on the delimiter, and returns the resultant chunks as elements of an array of strings in list. The integer return value is the number of elements in list

5.3.7 rspn()

```
size_t QSTR_rspn(const char* string, const char* accept);
```

rspn() is equivalent to *strspn()* except that it starts at the end of the string and works backwards instead of at the head of the string.

rspn() returns the number of characters that form an unbroken span of characters in accept occuring at the end of the string.

See also rcspn().

5.3.8 rcspn()

```
size_t QSTR_rcspn(const char* string, const char* reject);
```

rcspn() is the equivalent of strcspn except that it starts at the end of the string and works backwards.

rcspn() returns the number of characters that form an unbroken span of characters at the end of the string that are not found in reject.

See also rspn().

5.3.9 triml()

```
char* QSTR_triml(const char* string, const char* trimchars);
```

This function returns a copy of the passed in string with characters found in trimchars trimmed from the left side (beginning). This is a non-destructive left trim.

See also trim(), and trimr().

5.3.10 trimr()

```
char* QSTR_trimr(const char* string, const char* trimchars);
```

This function returns a copy of the passed in string with characters found in trimchars trimmed from the right side (end). This is a non-destructive right trim.

See also trim(), and triml().

5.3.11 trim()

```
char* QSTR_trim(const char* string, const char* trimchars);
```

This function returns a copy of the passed in string with characters found in trimchars trimmed from both ends. This is a non-destructive trim function.

See also trim(), and triml().

5.3.12 glue()

```
char* QSTR_glue(int num, ...);
```

glue() takes the strings passed into it and concatenates them together into a newly allocated buffer which it returns. The number of char pointers to be passed in must be given as the first argument.

All arguments after the first must be non-null char pointers. Any errors in the number or type of the arguments will lead to unpredictable behavior and hard to diagnose errors.

5.3.13 toupper

```
char* QSTR_toupper(const char* string);
```

QSTR_toupper() is similar to *toupper()* except that it operates on whole strings rather than single characters. This function will copy the source string into a dynamically allocated buffer, converting each character to uppercase. As usual, memory allocations are checked and this function will never return *NULL*.

string should never be *NULL*, although it can be zero length.

6. Utilities

Module: UTILITIES

Header: qci_util.h

Abbreviation: QCI

Summary: A collection of routines used by all modules of the C Utilities Library.

Dependencies: None

Version Documented: 3.5

This module is a place to store common routines and declarations used by all modules of the C Utilities Library. At the moment, it mainly contains wrapper functions that perform additional error checking on calls to a handful of standard C or POSIX functions.

The wrapper functions below will exit with an error message when an out of memory condition occurs. Since there is generally no way to recover from an out of memory error in C (nearly any proposed handling mechanism will require memory allocation) and many programmers forget to check the results of these calls, it is felt that handling the condition immediately and cleanly is far better than returning *NULL* to the caller and aborting on a null-pointer exception somewhere distant from the original source of the problem.

These wrappers are used throughout this library and most code written by QCI unless there is a specific requirement to try to trap and recover from these (usually hopeless) conditions.

When this behavior is not desired, such as when allocating a very large buffer that may reasonably fail, the underlying system calls should be used directly, the return checked, and perhaps the call retried with a smaller value. In general, this is a bad practice. Facilities like *memmap()* can be used to do this sort of thing in a cleaner, more efficient, and less error-prone manner.

Critical programs that cannot be allowed to fail can be handled proactively by making sure that enough memory is available in the first place and setting strict resource quotas on non-critical processes. In our experience, the critical program is often rendered useless by constant page thrashing long before virtual memory is actually exhausted.

6.1 Interface Summary

bool :

```
#define bool int

#define FALSE 0
#define TRUE 1
```

fatal_error() :

```
void QCI_fatal_error(const char* msg, ...);
```

daemonize() :

```
void QCI_daemonize( const char* prog_name,
                    int syslog_options,
    int facility );
```

xfork() :

```
pid_t xfork();
```

xmalloc() :

```
void* xmalloc(size_t size);
```

xcalloc() :

```
void* xcalloc(size_t nmemb, size_t size);
```

xrealloc() :

```
void* xrealloc(void* ptr, size_t newsize);
```

xmemdup() :

```
void* xmemdup(const void* src, size_t n);
```

scalloc() :

```
void* QCI_scalloc(size_t n);
```

mda_malloc() :

```
void* QCI_mda_malloc( int esiz, int dims, ... );
```

xgetlogin() `char* xgetlogin();`

is_daemon_proc :

```
extern bool QCI_is_daemon_proc;
```

fatal_error_log_level :

```
extern int QCI_fatal_error_log_level
```

6.2 Member Descriptions

6.2.1 bool

```
#define bool int

#define FALSE 0
#define TRUE 1
```

bool is a basic boolean type represented as a set of macros. These are used as placeholders until compilers implement the new ANSI C standard. bool includes the constants TRUE and FALSE which follow the standard C definition (zero is FALSE, one is TRUE).

6.2.2 fatal_error()

```
void QCI_fatal_error(const char* msg, ...);
```

fatal_error() will print the specified message on *stderr* and call *exit()* to end the program. If is_daemon_process is *TRUE*, then the message is sent to syslog instead, using the log level fatal_error_log_level.

6.2.3 daemonize()

```
void QCI_daemonize( const char* prog_name,
                    int syslog_options,
    int facility );
```

Initializes a daemon process:

1. forks
2. creates a new session (*setsid()*)
3. forks again
4. sets is_daemon_proc to *TRUE*
5. changes the working directory to "/"
6. clears the umask
7. closes open file descriptors
8. initializes syslog

In short, nearly everything that a daemon has to do at startup is taken care of. stdin, stdout, and stderr are left as invalid descriptors- the calling code is expected to deal with them appropriately. ΔIn the future, there will probably be a separate utility to open "/dev/null" and dup it, as this is often what is desired, but some daemons like to assign stderr and stdout to a log file, so this is not done automatically.

6.2.4 xfork()

```
pid_t xfork();
```

This wrapper checks the return from *fork()*. Resource exhaustion is detected and fatal_error() is called. If this call returns, the fork succeeded.

6.2.5 xmalloc()

```
void* xmalloc(size_t size);
```

xmalloc() wraps *malloc()*. If an out of memory condition occurs, xmalloc() calls fatal_error() with an appropriate message.

6.2.6 xcalloc()

```
void* xcalloc(size_t nmemb, size_t size);
```

xcalloc() wraps *calloc()*. If an out of memory condition occurs, xcalloc() calls fatal_error() with an appropriate message.

6.2.7 xrealloc()

```
void* xrealloc(void* ptr, size_t newsize);
```

xrealloc() wraps *realloc()*. If an out of memory condition occurs, xrealloc() calls fatal_error() with an appropriate message.

6.2.8 xmemdup()

```
void* xmemdup(const void* src, size_t n);
```

xmemdup() is a simple utility function that duplicates the contents of the specified range of memory by calling xmalloc()() followed by *memcpy()()*. A pointer to the new copy is returned.

6.2.9 scalloc()

```
void* QCI_scalloc(size_t n);
```

This function is identical to xcalloc() except that it is more convenient for allocating structures since it only takes one argument.

6.2.10 mda_malloc()

```
void* QCI_mda_malloc( int esiz, int dims, ... );
```

This function allocates a homogenous multi-dimensional array of elements with size esize and the the number of dimensions dims. The remaining arguments should be integers that give the sizes of the individual dimensions.

mda_malloc() allocates its memory in one chunk so that the entire array can be deallocated with a single call to *free()*.

This function is based heavily on public domain code written by Paul Schlyter.

6.2.11 xgetlogin()

```
char* xgetlogin();
```

A wrapper for *getlogin()*. This function copies the string returned by the system into a dynamically allocated buffer, which it returns. The string returned by *getlogin()* is transient and is not safe to store long-term.

Memory allocation is checked, and this function will never return *NULL* due to an out of memory condition, although it can return *NULL* if the information is not available.

6.2.12 is_daemon_proc

```
extern bool QCI_is_daemon_proc;
```

This flag indicates whether or not the current process is a daemon. If it is *TRUE*, then fatal_error() will write to syslog instead of to stderr(). This flag defaults to *FALSE*.

6.2.13 fatal_error_log_level

```
extern int QCI_fatal_error_log_level
```

This variable sets the syslog priority level used to log messages from fatal_error(). This variable defaults to *LOG_ERR*.