C Programming style guide of widely used conventions

Copyright (c) 2004, 2005, Matthew Mondor,
All Rights Reserved.

17th November 2005

Contents

Abstract

This paper describes one of the most widely used C programming styles which programmers should follow to help make their code readable by others (and themselves). Getting aquainted with this programming style also permits the programmer to read other popular works which comply to the same formatting regulations. Some aspects of the style also pertain to solving portability issues. It is most important to follow strict formatting rules when programming projects with other people. This becomes especially true in Open Source software, but is also very useful for corporate work where multiple individuals are writing code, and auditors are later on left to analyze it.

Although it may at first seem tedious to follow every single rule suggested herein, the consistency and readability of one's code generally clearly distinguishes the professional programmer from the amateur. If someone is eventually bound to read some of your code, they will immediately notice the professionalism involved in proper formatting and it becomes much easier to give some credit to the author, as well as confidence in his ability to write other good software in the future. By writing in the proper way, you are in a sense leading the way to respect of your work. Reading sloppy code generally causes one to remember not to hire that programmer for anything serious, or to even avoid to read other work from that same author.

C, being a low level language, requires a throughly understanding of its internal processes to achieve proper results. As its compiling process does not enforce any rules on programming style and consistency, which is a definite requirement for proper programming in C, a document like this helps to set forth this style, which has already been established and widely used within the industry, in projects like popular complete Unix systems.

The author does not intend to limit creativity or to impose his own personal programming style with this document. The techniques and conventions described herein were the results of observations when working with open source and closed source projects with other programmers. Most of these pedantic rules were already in use in wide code bases when the author was still struggling learning C. Because C learning books put emphasis on the language itself, and that style is secondary at that point, this document is beleived to be useful to set good habits for programmers, which often take a considerable amount of time to learn these techniques the hard way. Having a hard time to complete or maintain a project, which often needs to be rewritten from scratch, is one obvious lack of programming organization and style consistency. However, this is also targetted at C experts who need to work as part of an organization, where style standarization is ideal. Any relatively wide project where participation of a group of programmers is necessary should consider adhering to a particular style for consistency within the project. This guide aims to be an ideal tool to help in this area.

This document is intended for people with at least an intermediate C experience. We recommend the "K&R 2" or "Programming in ANSI C revised edition" books as a good starting point before this document can be useful. Some have asked the author why this document was written

this late, when Java and C++ are getting increasingly popular. The main reason is that C is still in wide use today, and that this document was needed to develop actual projects in C in today's world on which the author participated. For some projects, C still remains the ideal choice and as a result it won't be obsolete anytime soon.

Moreover, there exist a variety of programming and scripting languages which syntax are somewhat C derived (Java, JavaScript, PHP being examples) for wich most of these conventions can still be used.

1 General C Standards Compliance

These sections deal with areas where debates exist between C programmers pertaining to some standards which have been included or rejected by some compiler implementations. It gives advice on how to avoid potential problems, while still emphasizing on good style and consistency.

1.1 Wording conversions in this document

We will sparsely use two definitions in this document pertaining to code indenting.

soft tab A tabulation of 4 columns.
These are used when lines exceed the maximum line length (usually between 76-79 columns), to indent the following lines below the first one. Thus, we can say that all lines following the first line of code which exceeds the maximum columns width are indented with a *soft tab*.

hard tab A tabulation of 8 columns.
These are used to indent code blocks to respect code hierarchy. All code within a function is indented by one *hard tab*. Another code block such as an if will cause its code block to further be indented by another *hard tab*.

1.2 Function definitions and prototypes

We generally attempt to make the code as closely compliant to ANSI C (C89) widely spread standard. It is recommended to get a book which properly describes this standard, such as "K&R 2" or "Programming in ANSI C revised edition".

Notably, in this standard the function definitions are as follows:

```
int
func(int a, int b, int c)
{
```

rather than:

```
int func(a, b, c)
    int a, b, c;
{
```

which corresponds to the older K&R format.

Moreover, the arguments list should wrap near 80 columns as needed, and those continueing lines should be indented by one *soft tab* level (4 spaces). The function opening brace follows on the first column by itself after the arguments list. The function body will then be indented using one *hard tab* level (8 spaces).

For each function a prototype should exist, either at the top of the current file for local static functions, or within a headerfile (.h) which can then be included by the necessary modules (.c files) as needed. That prototype should follow the same semantics as the function definition, but should not hold the variable names of the arguments, except in special circumstances where it is considered impossible to give any sense to the prototype for humans otherwise. It is common practice to align prototype function names together using *hard tabs*. In the case where the prototype line exceeds the maximum number of columns for a line, it should continue with a *soft tab* indent for its following lines. For instance, valid prototypes would consist of:

```
static int      func(int, int, int);
static void     func2(int, int, int, int, int, int, int, int,
    int, int int);
```

Note that there always is a space after each argument separator ',' comma, but not before. Also note that there are no spaces before the opening '(' parenthesis.

Another point to remember is that when a C string is required as a function parameter and that the function is known not to modify the string in any way, *const char ** should be used, as opposed to *char ** to pass strings which are to be modified in any way by the function.

Also, when declaring arguments to functions, the pointer form using * is prefered to the one using *[]*, unless the function needs to explicitely use *sizeof()* on the supplied array, where it is allowed to use *name[n]*. This however is a very rare occurance in proper design, and the size of the array will generally be passed as a second argument to the function where needed, along with a * style pointer to the first element. A proper example is thus:

```
int func(int *, int);

/* ... */

int
func(int *array, int size)
{
        /* ... */
}
```

Functions which return nothing should be declared as returning *void* for both the prototype and their corresponding function declaration. Functions which require no arguments should also explicitely specify *void* in the parenthesis in both:

```
void func(void);

/* ... */

void
```

```
func(void)
{
        /* ... */
}
```

When a function has no need for any new variables, a blank line should be left instead of their definition block. Otherwise, all needed variables for the function should be specified before any other code in it, that is, at the top of the body. Of course, we will allow other variables to be defined later on in brace enclosed code blocks as well, but always at the top of the blocks. Examples:

```
int
function(int arg)
{
        int i;

        /* ... */
}

int
function2(int arg)
{

        /* ... */

        if (arg != 0) {
                int i;

                /* ... */
        }

        /* ... */
}
```

These rules should also be observed when working with C99 standard C compilers, even if the C99 standard allows new variables to be arbitrarily defined anywhere. Also, we do not normally use parenthesis for the return value of *return* unless required for readibility when a complex expression is used. Let's now resume with this example (definitely not a good example of functionality, but it serves to demonstrate the style):

```
int             main(int, char **);
static int      function(int);
static int      function2(int, int, int, int, int, int, int,
    int, int, int, int, int, int, int, int);

int
```

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

        return function(4);
}

static int
function(int val)
{
        int t;

        t = function2(val, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
            12, 13, 14);

        return t;
}

static int
function2(int a, int b, int c, int d, int e, int f, int g,
    int h, int i, int j, int k, int l, int m, int n, int o)
{

        return a + b + c + d + e + f + g + h + i + j + k + l +
            m + n + o;
}
```

1.3 Style of comments to use

Comments are very important in C code, but should be appropriate as much as possible. It is often meaningless and a waste to use comments where it is unnecessary, in cases such as:

```
/* Increment the i variable */
i++;
```

C89 (ANSI C and K&R) style comments should be used as opposed to C++ (or alternate C99 allowed) ones. For instance:

```
// This is a comment
```

should instead be represented like:

```
/* This is a comment */
```

Also, multiple lines comments should be represented as follows:

```
/*
 * This consists of a multiline comment, which continues on
 * the following lines.
 */
```

The comments should be indented at the same depth as the current code scope level.

It is a good idea to also place a comment before each defined function breifly describing its purpose, parameters, return values and special calling conditions. Of course, this should be no substitute to *BSD mdoc nroff* pages to provide references for exported library APIs.

Even tools such as doxygen, although useful to generate fast references, generally introduce other problems where the generated documentation lack important details, and where the code becomes bloated with too much documentation. Full pages of comments before every function in the source also can be a factor of discouragement when maintaining or auditing the code. It is often best to keep the descriptions before each function short but useful, and to maintain a general reference document separately for users of the library.

1.4 Avoiding implicit int type variables

It is a convention that variables declared without a type be automatically considered to be of type *int* in C. However, you should make sure to never declare variables without specifying a type. An *int* variable should therefore explicitely be declared to be of type *int*.

1.5 Avoiding returning or assigning structures or unions

For portability reasons, as well as code obviousness and consistency, it is recommended to not define functions returning structures or unions. Instead, they can be provided with a pointer to a structure to fill in with the results. It is also possible to provide the function with a pointer to a pointer, which it can then use to automatically allocate the structure if needed (supplied pointer is *NULL*), then assigning the supplied pointer to the object location. Where necessary, the same function, or a corresponding function, may also automatically detect that the old structure was not freed via the supplied pointer, free the resource and initialize back the pointer to *NULL*.

The following examples, although not at all realistic or useful in real life, show the two methods:

```
struct rectangle {
        int x, y;
};



void
mouse_position(struct rectangle *r)
{

        r->x = mousex;
        r->y = mousey;
```

```
}


int
mouse_get(struct rectangle **r)
{
        if (r != NULL) {
                if (*r == NULL)
                        *r = malloc(sizeof(struct rectangle));
                if (*r != NULL) {
                        *r->x = mousex;
                        *r->y = mousey;
                        return 0;
                }
        }

        return -1;
}

void
mouse_free(struct rectangle **r)
{
        if (r != NULL && *r != NULL) {
                free(*r);
                *r = NULL;
        }
}
```

In the case where a structure or union needs to be copied over another object of the same type, it is recommended to use *memcpy(3)* or *memmov(3)* to do it, or to copy its elements in a serialized way (often faster when dealing with moderate sized structures of *int* or *long* sized elements), rather than using a single assignment '=' operator.

Examples:

```
struct rectangle r1, r2;


r2.x = r1.x;
r2.y = r2.y;


(void) memcpy(&r2, &r1, sizeof(struct rectangle));
```

Another frequent case is where functions allocate a structure and return a pointer to the new structure in memory on success, or *NULL* if an error occured. An example is the stdio *fopen(3)* function which returns a FILE pointer

(*FILE* is a *typedef* to a structure). This however can be slower if requently used, especially if it only uses the standard *malloc(3)* function to allocate the structure, rather than a specially optimized pool allocator for fixed sized objects. Where performance is critical, it is often best to let the caller specify the memory address to initialize. This however also leaves the responsibility to allocate and free that memory as needed, and therefore is not adequate in all circumstances.

1.6 Do not confuse NULL with 0

It is advised to not assume *NULL* to be *0L* or *(void *)0*. To do this, *NULL* pointer checking should be done with explicit comparision with *NULL*, and *NULL* assignments or initializations should explicitely be set to *NULL* rather than using *memset(3)* to zero them. See 2.7.5 on page 27 for more details.

1.7 Do not confuse '\0' with 0

The end of C string *NIL* character *'\0'* should not be confused with *0*. Whenever one has to deal with a C string and set the terminator, an assignment of the char should be set to *'\0'*, not *0*. This is also true when running through a string and comparing the characters to detect the end of string, they should be compared against *'\0'*, not against *0*. See 2.7.6 on page 27for more details.

2 Consistency within projects

Consistency is very important. It makes the whole difference between a system which appears to be made from many unrelated parts which just have been glued together in a quick hack, and dependable systems with a solid interface and code.

A team of very consistent programmers following the same style and rules is able to help a project grow on strong bases. Such a team is also able to easily grow in the proper direction because new programmers joining in can be breifed about its standard rules so that no discrepencies are possible (or the less possible).

Although this document is fairly generic, it was written with exactly this goal, to sold together a strong team of programmers in a corporation. Although it is a nice tool to help achieve this, and that it can also be used for intermediate C level teaching, it cannot be enough to achieve full consistency for any C project.

Organizations will therefore need to follow this example and write themselves their standards rules which need to be followed by the development teams to achieve the wanted results. At least, this document attempts to fulfill this as far as C writing is concerned, helping in this direction.

2.1 The importance of spaces and tabs

2.1.1 Spaces

It is good practice to use spaces a lot in the file to make the code more readable. For instance, assignment and arithmetic operators should be separated with spaces from their values. Spaces should also follow the commas within function declarations and arguments. Here are some examples:

```
if (port < PORT_MINIMUM || port > PORT_MAXIMUM) {

for (i = PORT_MINIMUM; i <= PORT_MAXIMUM; i++) {

(void) printf("port = %d (max = %d)\n", port, PORT_MAXIMUM);

v = (seed % 127773 * 16807) - (seed / 127773 * 2836);

flags |= (FLAG_TEST | FLAG_SET);
```

Also, when declaring pointer or array variables, the following is discouraged:

```
char* a;
char b[];
char c [10];
char d[][];
char e[10][];
```

The correct equivalents consist of:

```
char    *a;
char    *b;
char    c[10];
char    **d;
char    *e[10];
```

Similarily, when declaring arguments to functions, the pointer form using * is prefered to the one using *[]*, unless the function needs to explicitely use *sizeof()* on the supplied array, where it is allowed to use *name[n]*. This however is a very rare occurance in proper design, and the size of the array will generally be passed as a second argument to the function where needed, along with a * style pointer to the first element.

2.1.2 Tabs

It is important for the *hard tab* width of the editor and readers to be set to 8. This ensures that the code remains readable when passed through text terminal readers and printers. It however is to the discretion of the project leader to establish if tabs should be used or not within the code of a project. Some will prefer to not have any tabs in the code but to instead only use spaces. Most editors provide the automatic capability of tab to spaces conversion which can be used then.

It is generally best to use 4 columns sized level indenting for *soft tabs* and 8 columns for *hard tabs*. This means that lines which are within a statement or definition block, would normally use 8 columns of indenting and ones which are wrapped from a long argument list or command would normally use 4 columns of indenting relatively to the indentation level of their parent. Even if real tabs (*hard tabs*) are used in the code, most editors support a "*soft tab stop*" parameter which can set the indenting level to 4, while still using tabs. This means that a combination of tabs and spaces would be used transparently as necessary by the editor.

A popular example for *VIm(1)* editor configuration. setting the *hard tab* to 8 colmns and the *soft tab* to 4 columns:

```
set ai cin ts=8 sw=4
```

This style is commonly seen in kernel code of the BSDs for instance. When using such wide indentation it is however very easy to quickly reach an indentation level where the code becomes unreadable as it needs to be wrapped too often. It then becomes necessary to use techniques to avoid this, such as opting for an elimination process of bad conditions to sanitize with possibly a *goto* to the end of the function which can perform the necessary cleanup a necessary, instead of using multiple level indentation.

It may also be an indication that the function should be split into more functions (which can lead to a loss of performance in some cases, however, but

it is definitely a good idea to use several short functions instead of very large ones for code readibility). See 5 on page 61 for more details. On the other hand, using one or two columns for indentation with many nesting generally proves to produce code which is harder to read.

See 6.2 on page 62 for more information about popular free code editors which can be configured to properly indent C code automatically.

2.1.3 Where to indent and when not to

It is important to indent by one level of *hard tab* the code which resides into a block statement, or within definitions blocks. Blocks are enclosed within '{' and '}' braces. For instance:

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

        if ((buf = malloc(1024)) != NULL) {
                /* ... */
        }
}
```

and:

```
struct test {
        /* ... */
};

enum test {
        /* ... */
};
```

In the case where conditionals do not lead to a statement block, but that only a single statement need to be executed, indenting by one *hard tab* level should also be done:

```
if ((buf = malloc(1024)) == NULL)
        return EXIT_FAILURE;
```

The continuation of lines which need to wrap to not exceed 78 columns use one level of *soft tab* indenting, until the line ends and ';' character reached. For instance:

```
if (port >= PORT_MINIMUM && port <= PORT_MAXIMUM &&
    ports[port].label != NULL) {
        /* ... */
}
```

Note that in the case where conditional, arithmetic or bitwise operators occur in a line which must be split, the operator should be left dangling at the end of the line *before* the split, as shown in the above example.

2.1.4 According to pointer variables

It is generally bad practice to concatenate '*' pointer declarators directly as the variable type suffix. For instance, you should generally not use the following form:

```
char* strbuf[10];
```

Instead, the following recommended form should be used:

```
char    *strbuf[10];
int     *i1, *i2, i;
```

The main reason for avoiding this style is that *char** is not a separate type by itself according to C syntax parsing (it's a *char **). This can be proven by the fact that the following:

```
char* a, b, c;
```

Will produce the expected result for *a* only (a *char **), but will produce a *char* for both *b* and *c*. Of course, this is different in the case of a *typedef*, which defines new types, including pointer based ones. Thus:

```
typedef char *  caddr_t;

caddr_t a, b, c;
```

always produces *char ** for all of *a*, *b* and *c*.

2.2 Wrapping lines at column 78

There are several reasons why lines should be wrapped when reaching 78 columns. Here are the most common reasons:

- Versionning systems such as CVS (see 6.4 on page 64), RCS and many others need to be able to generate readable diffs. The diff files are obviously unreadable if the lines are endless. Moreover, in the most popular universal diff format (-u flag to diff(1) command) needs to be able to append a '+' or '-' character before each line. This means that lines should not exceed 78 characters in length.
- In some circumstances it is necessary to be able to print the code on paper. If the code exceeds a 80 character line the resulting prints are unreadable. Many tools such as a2ps, as well as many printers assume a maximum of 80 characters per line, with a fixed width font.
- It should be possible to use a simple text reader to read the code on a terminal with 80 columns, with readable results. It also is possible to have diffs and code sections included in email text message body, and it also should be readable then no matter the mail reader in use.

- Recently, the advent of common embedded devices such as cell phones, PDAs, pocket PCs and the like tend to ressucitate the 80 columns terminal. Being able to read code and or edit it on those systems can prove to be useful.

Thus, lines should be wrapped where a space normally would have occurred. This means that you should use common sense to wrap the code to the next line at a word boundary as necessary. The exceeding lines should be indented by one level of *soft tab*. Arguments to function calls and such should be split using common-sense if required, but generally an argument will not be split, the splitting will occur in between them as necessary, although allowing the most arguments per line as possible to fit.

Operators found near a split should be left at the end of the previous line, not at the beginning of the following line.

2.3 Where to use a space before the '(' opening parenthesis

It is advised to insert a space before the opening parenthesis of the following keywords:

```
for (
while (
if (
switch (
```

All other function calls, keywords and macro calls, function or macro definitions *should have no space before the opening parenthesis.* If a *return* statement needs parenthesis because it evaluates a condition or such (return statement normally is not used with parenthesis), it also should use a space before that opening parenthesis.

2.4 Where to use a '(' parenthesis and when not to use one

The *return* statement does not usually require parenthesis, which should generally be omited. However, in the case where it must perform some arithmetic in the same statement which needs to result into the return code, it is generally best to use parenthesis. For example:

```
return NULL;
return buf;
return (a - b);
```

It is recommended to use parenthesis with the preprocessor *sizeof()* keyword:

```
struct test     *st;

if ((st = malloc(sizeof(struct test)) != NULL) {
```

It is advised to use parenthesis in arithmetic expressions as required to make obvious what the programmer intended when writing the mathematical formula. We know that there is a specific order of expressions, but when debugging it can be hard to guess what the original writer really wanted to accomplish unless proper use of parenthesis to demonstrate it is present. Here is an example to demonstrate this common problem:

```
if ((v = (seed % 127773) * 16807 - seed / 127773 * 2836)) < 1)
        v += 0x7fffffff;
```

Now we know that multiplications and divisions would normally be evaluated first and that as such the substraction at the middle should thus result in a defined behavior. However, this does not show at all what the actual intention of the author was. If there was some reason why the result was wrong, how would we know if perhaps he actually needed a parenthesis somewhere to achieve the wanted result?

```
if ((v = (seed % 127773 * 16807) - (seed / 127773 * 2836)) < 1)
        v += 0x7fffffff;
```

This is rather obvious to show the author's intention. In many circumstances, it is better to use some parenthesis which are considered useless for compilation but which will prove very useful to those who will need to manage the code later on. Even for the original author. The compiler will not generate slower code because of them.

2.5 Where to place the definition and statement block '{' and '}' braces

In the case of functions, there is always an opening brace as the first column of the next line, and the closing brace is aligned with the opening brace as usual. For structures (*struct*), unions (*union*) or enumerations (*enum*), the opening brace is usually immediately followed the name after a space, and the closing brace is aligned with the first column, followed by a semicolon. Examples:

```
int
myfunc(void)
{
        /* ... */
}

struct mystruct {
        /* ... */
};
```

```
enum myenum {
        /* ... */
};
```

For *if ()*, *while ()*, *for ()* and *switch ()*, in the case of block statements rather than a single statement which requires no braces, the opening brace is found on the same line, after a space. In the case of *switch () / case* keywords, they are at the same level of indentation as their parent *switch ()*.

```
if (condition) {

while (condition) {

for (initialization; condition; statement) {

switch (var) {
case val:
       /* ... */
}
```

Some groups have been known to encourage the use of aligned braces found indented under the initial line, as follows:

```
if (condition)
  {
    /* ... */
  }
/* ... */
```

This style is however discouraged as it both tends to waste vertical space (number of visible lines of useful code in a window) and to not comply with most of the already written C code in the real world. This format is also considered harder to read in complex code. The GNU Project is known to encourage this style, although within the GNU projects many authors still use the standard K&R and BSD-style conventions. This includes the Linux kernel (although not technically a GNU project), the PTh library and many other GPL released packages out there, which are closer to observing the standard BSD style described in this document rather than the GNU proposed style.

Another mildly popular indenting and formatting style consists of the Allman one, which looks similar to the GNU one except for the fact that braces are aligned with the previous statement rather than indented of 2 spaces. The Allman style also generally uses 2 spaces indenting, which is considered too small for many programmers (and code auditors). It also has the problem where unnecessary vertical space is wasted, like for the GNU style. This style also does not differenciate important language statements by separating the openning parenthesis with a space. En excerpt of Allman style follows:

```
if(condition)
{
  /* ... */
}
```

A common convention used by all of these styles is that multiple statements on a same line separated by semicolons are discouraged. One statement per line is usually used in all common C styles, including the one presented in this book. Exceptions to this rule can sometime occur when using a macro to unroll blocks of code where a single call might need to be called a number of times without (or with very few) arguments.

2.6 Program flow

2.6.1 Infinite loops

There is a popular convention where *for (;;)* is used instead of *while (1)* or *while (TRUE)* for infinite loops. The reason is simple. The *while ()* statement assumes a conditional, and needs to evaluate it (although this doesn't mean that the compiler can't detect that it will always evaluate to *TRUE* and optimize it). The *for ()* statement, however, has each of the initialization, condition and iteration statements elements optional. This means that *for (;;)* does not imply this condition checking. Using it over *while (1)* will not only allow certain compiler to perform a better optimization of the loop, but also clearly states the author's intention to wait forever. No conditional variable or constant in the way. It is more elegant, and more consistent with the intention. It is also consistent with the *for ()* design where each of its three sections are optional.

In fact, the use of the */* CONSTCOND */* comment would be necessary if the *while* solution was used to achieve this. This kind of conditional constant use is discouraged, and should be marked as such if it needs to be used for a reason or another for *lint(1)*-like tools and human code auditors alike. See 2.17 for more information.

2.6.2 goto

Many programming students have explicitely been toold multiple times by their teachers never to use *goto*. The author agrees that whenever possible, its use should be discouraged. However, there are specific circumstances where it is ideal, sometimes for efficiency, but even for code readability (in which case dismissing the assumption that using *goto* generates unreadable code).

Of course, a program which mainly uses *goto* everywhere it needs to jump is less readable than another program generally using more structured flow control statements such as *while ()* and *for ()*. Using these later constructs is recommended whenever possible.

It should be noted that the C language has no *pop* instruction, and does not even permit one to specify a label to branch to at *break* or *continue* statements. Java implements such label support for break and continue, and totally removed

goto from the language. However, understanding the nature of the synchronization, Garbage Collection and Exceptions methods on which the Java language is based permits to understand that *goto* becomes totally useless in Java. This is not the case with C.

setjmp(3) and *longjmp(3)* as well as their popular unix friends with special signal handling in fact also consist of a form of wide *goto*, and also finds it's use at occasions, particularly in the handling of error events.

Often when writing C code an application needs to ensure to perform explicit freeing of allocated resources which are no longer in use, to prevent memory leaks which would result in the process growing ever larger until it eventually has problems, either exceeding the allowed heap size for the process or running out of memory. This is also true for other kinds of resources, such as locks and files. And C can also be used to write kernels, operating systems, drivers and other low level programs, not only simple user space programs which simply exit assuming their resources to be freed. For instance, locks may require to be released back by all means by the same function, which has many exit points. This often is where the *goto* statement finds its wise usage.

See 5 on page 61 later on for more details with examples.

2.6.3 Favorizing *for ()*

There are many cases where the programmer has the choice over using *while ()* or *for ()* for a particular task at hand. It is recommended to always use *for ()* instead of *while ()* for loops which either require particular initialization before it, or that need to inherently perform some repetitive iterative task like increasing a pointer, variable or setting a pointer to the next element into a list. Here is a common example where to run through a linked list both can be used:

```
node = DLIST_TOP(list);
while (node != NULL) {
        /* ... */
        node = DLIST_NEXT(node);
}

node = DLIST_TOP(list);
while (node != NULL) {
        tmp = DLIST_NEXT(node);
        /* ... */
        node = tmp;
}

node = list->top;
while (node != NULL) {
        /* ... */
        node = node->next;
}
```

```
node = list->top;
while (node != NULL) {
        tmp = node->next;
        /* ... */
        node = tmp;
}
```

Although it is rather simple to use *while ()* for all such loops, the *for ()* loop is especially optimized for this form of operation, and yields more readable results (it is even possible that it also be more optimized under some compilers and architectures combinations). Moreover, it is much less likely that the iteration or initialization parts be omited by mistake, as they inherently are part of the construct. Furthermore, the equivalent using *for ()* tends to waste less vertical lines:

```
for (node = DLIST_TOP(list); node != NULL;
    node = DLIST_NEXT(node)) {
        /* ... */
}

for (node = DLIST_TOP(list); node != NULL; node = tmp) {
        tmp = DLIST_NEXT(node);
        /* ... */
}

for (node = list->top; node != NULL; node = node->next) {
        /* ... */
}

for (node = list->top; node != NULL; node = tmp) {
        tmp = node->next;
        /* ... */
}
```

Similarily, for the frequent cases where an iterating operation is performed but that no initial initialization is needed, *for ()* also proves to be best:

```
char    *ptr;

for (ptr = str; *ptr != '\0' && *ptr != '.' ; ptr++) ;
if (*ptr == '.') {
        /* ... */
        for (; *ptr != '\0' && *ptr != ':' ; ptr++) ;
        if (*ptr == ':') {
                /* ... */
        }
}
/* ... */
```

2.6.4 switch and case

It often is useful to use *switch ()* and *case* in some circumstances where series of *if ()* with *else if ()* is considered unappropriate. Of course, *switch ()* should not be used where the use of *continue* or *break* within the conditionals should lend control to a larger scope. Its other serious limitation is that it can only deal with constant expressions (but which can sometimes be optimized in the form of a jump table by the compiler, which of course the programmer could do using an index array as needed for large comparision sets). In any case, where this is required, here is the expected format:

```
switch (expression) {
case <constant>:
        /* ... */
        break;
case <constant>:
        /* ... */
        break;
/* ... */
}
```

As shown, the *case* statements remain on the same indentation level as the parent *switch ()*. Obviously, fallthrough cases (without *break*) as well as *default* can often be used in conjunction with this style without problems as well. Note that there is a space between *switch* and its opening parenthesis.

2.7 Conditions

2.7.1 Where to place else and its optional braces

When the *else* keyword is used, its optional braces which depend upon if multiple statements are needed (a block statement) should appear on the same line with the *else*, but separated by a space.

Here are shown the four obvious situations as examples:

```
if (condition) {
        /* ... statements ... */
} else {
        /* ... statements ... */
}

if (condition)
        /* statement */
else
        /* statement */

if (condition) {
```

```
        /* ... statements ... */
} else
        /* statement */

if (condition)
        /* statement */
else {
        /* ... statements ... */
}
```

The same format applies for *else if*:

```
if (condition)
        /* statement */
else if (condition) {
        /* ... statements ... */
/* ... */
```

2.7.2 *? :* conditions

The *? :* C if-else style conditionals should normally be used enclosed within parenthesis. For example:

```
(void) printf("%s", (v ? "TRUE" : "FALSE"));
```

Where required because of the complexity of the conditional expression, it is also recommended to enclose the expression in parentheses. Moreover, if the two values returned by the condition are not of the same type, proper casting should be used so that it becomes the case. Other examples:

```
(void) printf("%s", ((v != 0) ? : "TRUE" : "FALSE"));

char    *ptr = ((optr != NULL) ? optr : (char *)iptr);
```

2.7.3 Boolean logic

The *if (x)* and *if (!x)* style conditional expression should normally only be used with values which are explicitly 0 or 1, in accordance with boolean mathematics semantics. Generally, the standard definitions *TRUE* and *FALSE* are defined as follows:

```
#define TRUE    /* CONSTCOND */(1)
#define FALSE   /* CONSTCOND */(0)
```

This means that the following should always be equivalent:

```
if (v) {

if (v == TRUE) {
```

as well as:

```
if (!v) {

if (v == FALSE) {
```

Using this convention, the boolean logic conditionals cannot be used over pointers nor on results of bitwise operations. It also prevents confusion and bugs where a particular value which is below 0 or over 1 would not be considered as it should. A function which does not explicitely return a boolean value (*TRUE* or *FALSE*) should never have its returned value compared using boolean logic. A common example is many functions which are ANSI or POSIX standard which return an *int* set to 0 on success or -1 on error. These will never be mismatched with boolean logic, and need to explicitely be compared against the integers 0 and -1.

Note that it can be useful to have a user defined type using *typedef int bool_t* for instance so that variables of type *bool_t* may clearly be identified as boolean variables. It is advised to however verify that the libraries and the system on which the applications are meant to compile on do not already have such a conflicting type of the same name. The curses library, for instance, is known to declare its own, using *typedef char bool*. This means that your library should avoid conflicts by ensuring to not define *bool* if using that library. Another choice like *bool_t* or *boolean_t* might be a better idea then. It is recommended to use *int* instead of *char* to define a custom boolean type.

2.7.4 Bitwise operators logic

It is not uncommon to see incorrect conditions related to bit state checking, using boolean logic instead of actual comparision with 0. Although both forms produce working code, the semantics are different between boolean logic and the result of bitwise operations, since they do not explicitely return 0 or 1, but rather 0 or a non-0 value. The wrong examples:

```
if (flags & FLAG_TEST) {

if (!(flags & FLAG_TEST)) {

if (flags & (1 << 2)) {
```

Should rather be expressed as:

```
if ((flags & FLAG_TEST) != 0) {

if ((flags & FLAG_TEST) == 0) {

if ((flags & (1 << 2)) != 0) {
```

This ensures code clarity and consistency. Because the result of a bitwise operator will either be zero or not, rather than *TRUE* or *FALSE*, explicit comparision against zero should be performed. Again, it also shows exactly what the author wanted.

2.7.5 NULL comparision

Many people use boolean logic to deal with *NULL* pointers. However, this is not good practice for various reasons. A system may not be using *(void *)0* or *0L* to define *NULL* is an example (although standard C compilers will always make sure that those comparision work). Moreover, pointers are not boolean variables by definition. Thus, the common code

```
if ((ptr = malloc(BUFSIZ))) {

if (!(ptr = malloc(BUFSIZ))) {

if (ptr) {

if (!ptr) {
```

should in fact be written properly as:

```
if ((ptr = malloc(BUFSIZ)) != NULL) {

if ((ptr = malloc(BUFSIZ)) == NULL) {

if (ptr != NULL) {

if (ptr == NULL) {
```

The second method not only avoids possible incompatibilities with some systems and compilers, but also makes obvious that we are dealing with a pointer, since only pointers can be assigned *NULL* values. And a *NULL* pointer is considered as pointing to invalid data.

It is also a bad idea to use the *memset(3)* or *bzero(3)* functions to initialize arrays or structures which contain pointers to *NULL*. It is generally best to explicitely assign those to *NULL*. In many cases, this will also be faster, since explicitely setting a pointer to *NULL* will generally consist in a word operation, where it is possible for *memset(3)* to need to perform multiple byte-sized operations, depending on the implementation and on the buffer size and alignment to clear. Also, do not initialize pointers to *0* instead of *NULL*, or compare them against *0* instead of *NULL*.

2.7.6 NIL comparision

The end of C string character *NIL* is *'\0'*. This means that explicit comparision against *'\0'* should be performed, or explicit assignment against *'\0'*, rather

than using *0* or relying on *memset(3)/bzero(3)* to set a valid end of C string delimiter. Most importantly, *NIL* is **not** *NULL*.

Discouraged examples (some of them also actually bad):

```
for (c = str; *c; c++) ;

for (c = str; *c != 0; c++) ;

for (c = str; (*c); c++) ;

for (c = str; c != NULL; c++) ;

for (c = str; c > 0; c++) ;
```

Proper example:

```
for (c = str; c != '\0'; c++) ;
```

2.7.7 -1 error code comparision

Many ANSI and POSIX standard functions in popular APIs normally return a 0 code on success or -1 on failure, with *errno* global variable set to the error condition (which can be printed out using the *strerror(3)* function). Because -1 is not 1, boolean conditional evaluation should not be performed for such functions. It is also more appropriate to compare the result against -1 than to verify if it is smaller than zero.

Bad and discouraged examples:

```
if (close(fd)) {
        /* ... error ... */

if (!close(fd)) {
        /* ... success ... */

if (close(fd) < 0) {
        /* ... error ... */
```

More obvious, encouraged consistent counterparts:

```
if (close(fd) == -1) {
        /* ... error ... */

if (close(fd) != -1) {
        /* ... success ... */

if (close(fd) == 0) {
        /* ... success ... */
```

```
if (close(fd) != 0) {
        /* ... error ... */
```

The main reason for this is that the API documentation specifies the returned values as specifically being *0* or *-1* for the *close(2)* system call. The encouraged example above is thus compliant with the function semantics. No ambiguity.

2.8 Statement flow

It is generally nice to only use one statement per line even if multiple ones could coexist separated by ';'. In general, in conditional statements or control flow statements which do not need a block of statements, the single statement will generally be stored on the next line, indented by one level of *hard tab*. For instance:

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

if (condition)
        statement;

statement1;
statement2;
```

2.9 size_t, ssize_t and off_t common data types

It is important to know that in general only byte quantities should be used into *size_t*, *ssize_t* and *off_t*. This means that by convention, you should not use variables of this type to hold number of elements, unless elements always are byte-sized.

size_t consists of the unsigned type (often *typedef unsigned long*), while *ssize_t* consists of the signed type (usually *typedef long*). The later is generally used so that a function may return -1 on failure rather than a number of bytes. This is true for instance for common Unix I/O functions corresponding to syscalls, such as *read(2)* and *write(2)*. The *off_t* type (usually *typedef quad_t, typedef unsigned long long*, varies with OS) is generally used to hold bytes offsets in files (file position pointers).

When a number of elements needs to be stored and that the size of an element does not consist of a byte, *int*, *unsigned int*, *long* and *unsigned long* are more suited.

It is considered to be a semantic mistake in the implementation of stdio *fread()* and *fwrite()* functions to use a *size_t* instead of an *int* for the number of members argument. The common assumption made by C++ programmers that *size_t* is the common type internally used to index arrays is also false in C. *size_t* should therefore only be used to express byte quantities.

2.10 Avoiding functions returning pointers to static storage

It is important to discourage functions which return pointers to results in static storage. These are functions which are not-reentrant, and cannot be used safely in threaded systems (inherantly sharing their memory) without special care with mutual exclusive locks. It is a better idea to allow the functions to be provided the buffer in which to place the results if they cannot return them in a native C type value, rather than to use an internal buffer and return a pointer to it, unless this buffer especially was allocated and needs to be freed by the caller afterwards (which however generally results in less efficient code).

There are special exceptions which can be made to this rule, however. Consider for instance a function like *getpass(3)* which may internally use a special page of memory allocated in a mode which will not be swapped out to disk to store passwords (like using a *MAP_PRIVATE* page locked/wired in physical RAM with *mlock(2)* on BSD systems). Then again, another API could provide *password_get()* and *password_free()* while achieving the same results and still being reentrant and thread-safe. Moreover, *password_free()* would then also take care to zero out the buffer before unlocking the page and freeing it, still providing another reason to choose such a system over *getpass(3)*.

The examples at *1.5 on page 11* show proper ways to deal with situations where the results of a function are too large to be returned into a native C variable.

2.11 Functions and definitions naming conventions

2.11.1 Functions

It is generally ideal to not use uppercase characters in functions, and to use underscore ('_') separators between the words of a function name. If the function is named after an already existing function which is standard, but that the new function does not exactly behave like the standard function of the same name, it should be prefixed with an underscore ('_').

For instance, one's version of *int strcmp(const char *str1, const char *str2)* which returns *TRUE* or *FALSE* if the strings match or not rather than 0 if they match (which should actually be called *streq()* even), should be named instead *int _strcmp(const char *str1, const char *str2)*. Note that this is also true for any slight modification, including the prototype definition (i.e. standard uses *const char ** and custom function *char **, etc).

It is a good idea to consider a multiword function name such as *port_open()* to have a hierarchy, where the first word is most significant to determine the library name or realm within which the open function applies. This hierarchy is also true for macro names. Thus, *port_open()*, *port_close()*, *port_send()* etc all operate within the same port realm. *port_send()* probably also has it's first argument specifying the port pointer or id on which to operate (as obtained from *port_open()*). This is somewhat contrary to some OOP languages where

the hierarchy is reversed and that upper case characters are used to delimit the words of a class rather than underscores. For instance, Java uses *SomeException* where convention in our style woul be to use something like *exception_some*.

This means that when writing function libraries, those functions usually all start with the same realm name, which may correspond to the library name in many cases. The library name is often set to the name of the type of object this library is set to operate on (i.e. *port.o* in the case of the example above).

It is important for each function to have a prototype. Moreover, if a function returns nothing, *void* should be its return type, both in the prototype and function definition. If it has no arguments, *void* should also be used both in its prototype and function definition as the only argument.

Depending on the project, the return type of the function should be placed alone on a line before the remaining of the function definition, or it may be immediately followed by the function. Try to stick with the form used by the particular project you are working on. We recommend placing the return type of the function alone on a line before the rest of the function declaration (the first example). For example:

```
int
strcmp(const char *str1, const char *str2)
{


int strcmp(const char *str1, const char *str2)
{
```

In all cases the corresponding function prototype should have the function name and arguments defined on the same line as the return type, however. Of course, common sense should be used to wrap and indent arguments of the prototype or function which exceed 78 columns, using a *soft tab* of indentation for exceeding lines:

```
int     function(const char *, const char *, const char *,
            const char *);


int
function(const char *a, const char *b, const char *c,
    const char *d)
{
```

2.11.2 Macros and constants

Macros and constants are defined using the *#define* preprocessor directive. These should all be in uppercase, and should use underscores ('_') as word separators as needed. These should also represent a hierarchy in their name, the same way that function do (see 2.11.1 on the page before).

A great use for macros is to improve performance when executing common statements which are best to be isolated into an abstract interface without

actually causing a function call. This is very useful for instance to implement linked lists primitives. Another usage is to perform conditionals or arithmetic which are rather complex and which make the code ugly when pasted in as-is. A macro can then be created and the code can then repetitively use that macro for cleaner code (and less debugging trouble, as the critical section is isolated at a unique location rather than duplicated everywhere). Sometimes, however, debugging can be harder with macros since one may occasionally need to look at the post-preprocessor results to know what caused a syntax error.

It is important for macros to use parentheses to enclose their arguments. This is because the preprocessor only performs text expansion and then compiles the result. If the macro for instance requires a pointer to a particular object, and that the caller supplies it using dereferencing etc, the lack of parenthesis would often cause compiling errors. Instead of requireing the caller to add useless parenthesis enclosing each argument passed to a macro, the macro should do so itself so that it may behave semantically as a function the most possible where it is intended to be called as such.

A macro which needs more than one line to be defined should carefully be wrapped with the '\' character used at the end of each line, generally aligned near the end of the line at a *hard tab* location. A macro which has more than a single statement to execute (which is not made to return a value but rather to execute statements) should be enclosed in a special *while ()*. Examples:

```
#define DLIST_INIT(lst) do {                            \
        (lst)->top = (lst)->bottom = NULL;              \
        (lst)->nodes = 0;                               \
} while (/* CONSTCOND */0)

#define DLIST_NODES(lst)        (((list_t *)(lst))->nodes)
```

2.11.3 Variables

Variables should generally be in lowercase, and may use '_' separators if they comport more than one word. They optionally may be followed by a number where appropriate. Wide scope variables, such as global ones, generally are multiword ones with the first word used to define their hierarchy, as for function names.

However, very small scope variables in tight loops for instance, are generally given smaller names, especially iterator variables such as i for index variable, n or *node* to represent a list or table node.

Global variables should generally be used the less possible, and functions should be passing the required arguments among them as necessary. It oten is also a good idea to use a structure describing a context and to pass that context pointer among the functions which need to access the context-wide variables. If a significant number of global variables are needed for a particular project, they should ideally be grouped together by category or purpose into structures. It is a good idea to use comments to describe their purpose.

Although C99 allows variable names to be defined at any location among the statements, it is advised to only declare them at the beginning of code blocks for compatibility with C89 and consistency with the vast majority of already existing code. Many programmers also find the result cleaner and less error-prone. Moreover, some unstandard compilers such as VC++ are known to not properly follow C99 semantics by default when variables are used in *for ()* loops initialization section.

The *register* keyword should be used with care, it only should be used for variables and pointers which are known to be used the most in the context of a particular tight loop. Defining all variables with *register* dismisses totally the use of the keyword since most processors have a very restricted set of registers (especially the x86). In addition, many compilers ignore the register keyword and will use their own heuristics to place the ones they can into registers. Although they normally should allow the programmer to specify which variables should have the most priority to be palced in registers, using the *register* keyword, some compilers don't care (it seems to be the case with GCC v2 on i386, at least). Defining variables which only need to be used in the small scope of a loop in the smallest scope as possible will often have the effect of allowing them to be placed into registers more easily, as is the case with GCC.

The *volatile* keyword should be used for variables which should only be accessed or set with atomic assignment operations. This is especially useful when implementing resource access locks or when programming drivers which deal with hardware registers. The operation of incrementing a *volatile* variable for instance will cause its value to be loaded, incremented and then be saved back over the old one (the atomic assignment operation), instead of attempting to increment the value on place. On a normal variable an atomic increment operation would generally be used, which is unsuitable for some types of hardware registers, where *volatile* must be used. Other optimized operations will often operate read and/or write operations on only some bits of the memory, while *volatile* read/write operations always read or write the entire memory word.

As for functions, global variables which are not to be exported to other modules should be declared *static*, just like for functions. The ones which will require external usage should be declared as *extern* in the headerfile (not in the C source). The *static* keyword may also be used for variables within the scope of a recursive function which needs to refer to the same variable among all recursions. This however has to be used with care since it causes this function to not be reentrant, which could be a problem if threads were used.

The variables which need to only be accessed for read operations into the code, such as an array of the days of the week to be used later on in a date formatting function, should be declared as *const*.

Example:

```
static const char *wdays[] = {
        "Sunday",
        "Monday",
        "Tuesday",
```

```
        "Wensday",
        "Thursday",
        "Fryday",
        "Saturday"
};

int
wday_print(int day)
{
        if (day < 0 || day > 6) {
                errno = EINVAL;
                return -1;
        }

        (void) printf("%s", wdays[day]);
        return 0;
}
```

It is generally unnecessary to include *int* with *char, short* or with *long*, or to specify *signed*. However, the C feature of implicitely considering undeclared variable types as *int* types should not be used, and *unsigned* should be used where appropriate.

Unsigned types should be used when bit operations are to be performed on the variable. This is also true for arguments and return types of functions which need to operate on the bits of such variables. For instance, byte swapping functions described at 4.1.1 on page 52 use unsigned types, since they operate on fixed bit-sized words and need to perform bitwise operations with them.

2.11.4 Enumerations

Enumerations are very useful when one has to define a lot of constant values which should be equivalent and compared against *int* integers. It is often a mess to have to *#define* 50 or more number-mapped constants which need to be contiguous. It is very easy to either miss a number, or to have one twice in the list by mistake when typing them in. Moreover, you often need to later on perform range sanity checking on user supplied arguments to make sure that it is a valid element. Here is an example on how an enumeration can be useful in this case:

```
enum elements {
    ELEMENT_SOMENAME1 = 0,
    ELEMENT_SOMENAME2,
    ELEMENT_SOMENAME3,
    ELEMENT_SOMENAME4,
    /* ... */
    ELEMENT_MAX
};
```

This makes it very easy to perform range checking against one of the valid elements. Let's say that the caller supplied us with *var*, an *int* containing one of *ELEMENT_ ** . We simply need to do the following:

```
if (var > -1 && var < ELEMENT_MAX) {
        /* ... within range ... */
```

After such range sanity checking it is then possible to also have each of those elements indexed to some custom wanted data using an array (for instance to convert an element code to element name string):

```
const char *elemements_labels[ELEMENT_MAX + 1] = {
        "ELEMENT_SOMENAME1",
        "ELEMENT_SOMENAME2",
        "ELEMENT_SOMENAME3",
        "ELEMENT_SOMENAME4",
        /* ... */
        NULL
};
```

First of all, because we use *[ELEMENT_MAX + 1]* we know that if the number of strings does not match a compilation error will occur immediately. Then, we know that after performing range checking we can safely use *elements_labels[var]* to map the number to string efficiently.

If we wanted to map the element string to a number, we could perform a sequencial search because we made sure to include a *NULL* string pointer which can be used to stop iterating searching for a matching string. As such the above method is a very powerful and safe construct, yet very simple and clean to read.

The hierarchy naming which is used for function names and macros also applies to definitions and enumeration names. The more consistent an application is about this, the easier the code is to manage, even when eventually getting a very large project.

2.11.5 Custom types

When writing libraries of code which will be used often, it is generally useful to use the *typedef* preprocessor directive to define a structure or variable so that it is easy to distinguish and short to type. For instance, the *<stdio.h>* popular *FILE* type consists of a *typedef* to a structure. Two easy ways to perform this are as follows:

```
typedef struct test {
        /* ... */
} test_t;
```

or:

```
struct test {
        /* ... */
};

typedef struct test test_t;
```

As a general convention to avoid conflict with exiting C keywords and types, it is a good idea to append a *_t* suffix to those custom types. As usual, it is a good idea to use a hierarchical name for the new type which corresponds to the library functions that operate on it.

Another popular convention is to use uppercase letters for the type name instead of appending the *_t* suffix. However, because the standard libraries are using this method already, there is less namespace available to prevent possible or eventual conflicts if one also uses this method. The decision to choose ones with *_t* suffix or uppercase names instead depends on the context and project. In any case, conflicts with existing definitions should be avoided as possible, and consistency in the project must be maintained.

2.11.6 Dealing with the lack of classes and objects

C, unlike C++ or Java, does not natively have *class* or object support. This, of course, does not limit the power of C, which can also use objects with associated functions (similarily to classes with associated methods).

There are two popular widely used methods to deal with this issue. For a good example of the first one, look at the *stdio* functions. The *FILE* typedef is defined in *<stdio.h>*, which corresponds to an internal structure, defined by the same headerfile. This is used as a handler, created by *fopen(3)*, destroyed with *fclose(3)*, and which can be used with *fread(3)*, *fwrite(3)*, *fgets(3)*, *fprintf(3)*. Let's show a similar variant, distinguished by the fact that these are macros, and that the "object" is initialized at the specified address for efficiency, rather than allocated. For simplicity we simply provide prototypes:

```
void DLIST_INIT(list_t *);
void DLIST_APPEND(list_t *, node_t *);
void DLIST_INSERT(list_t *, node_t *);
void DLIST_UNLINK(list_t *, node_t *);
/* ... */
```

In the above example, *DLIST_* can easily be used to distinguish the hierarchy/class/object type on which these all operate. The *list_t* pointer, which they all need, is supplied as the first argument for each of them, for consistency.

The second method involves in holding the function pointers of functions which need to operate on a particular object within the object. This can be especially useful when designing a library which has to arbitrarily deal with internal implementations, transparently. An example of a library which uses this method consists of the Berkeley db3/4 low level database library (more information on this library is provided later on). With this implementation

method, a function generally optionally allocates and initializes a structure with the required function pointers. We can then invoke the "methods" of the virtual "class" similarily to when using C++. The *mmlib mmfd* library also uses a similar method to abstract a number of functions which can be provided by the caller, such as thread-safe alternatives or for other reasons (also discussed later on). Here is provided a small example of code which permits to abstract *read(2)* and *write(2)*:

```
struct abstract_fd {
        int fd;
        ssize_t (*read)(int, void *, size_t);
        ssize_t (*write)(int, const void *, size_t);
};
typedef struct abstract_fd afd_t;

int
afd_init(afd_t *afd, int fd,
    ssize_t (*rf)(int, void *, size_t),
    ssize_t (*wf)(int, const void *, size_t)) {
        if (afd != NULL && rf != NULL && wf != NULL) {
                afd->fd = fd;
                afd->read = rf;
                afd->write = wf;
                return 0;
        }
        return -1;
}

ssize_t
afd_read(afd_t *afd, void *buf, size_t size) {
        return afd->read(afd->fd, buf, size);
}

ssize_t afd_write(afd_t *afd, const void *buf, size_t size) {
        return afd->write(afd->fd, buf, size);
}
```

As we can see above, *afd_read()* and *afd_write()* provide a handler-like interface, but are internally using a class-like convention to call the abstracted *read()* and *write()* functions. Who knows if those are dealing with actual filedescriptors, or some other system using the same interface. The caller doesn't need to know after *afd_t* initialization. Of course, this example is rather useless, but in a real world situation such a system would be providing additional functionality, internally using the supplied functions and filedescriptor when necessary.

2.12 Printing error codes

Libraries which are part of a standard C library usually ensure to set the *errno* global variable to a standard error code value, which should then be mapped to a string using the *strerror(3)* or *perror(3)* functions. The applications should not roll their own code to string conversion functions when these are already provided.

Some libraries which emphasize alot on threads will often directly return an error code from a function rather than setting a global variable. These libraries should also provide the caller with a function or macro to convert the return codes it provides to corresponding strings. The strings will generally be stored in a *const char *** array in the source file, the corresponding error codes *enum* into the headerfile, with the macro used to convert the errors to strings. The headerfile declares the array of strings *extern* so that it may be accessed by applications using the library.

This means that when writing a library, make sure to provide a proper means for the caller to map error codes to string. It also means that when using a library, its own provided means to print error codes should be used instead of rolling our own.

It must be noted that some *strerror(3)* implementations are not reentrant, and as such cannot be used safely with threads, for instance. This is more common nowadays with the advent of internationalization of error messages offered by operating systems. In such circumstances, *strerror_r(3)* is sometimes provided by the C library. Otherwise, it might be necessary to use a custom thread-friendly version of it, protected by a mutex.

2.13 Parsing command line arguments

For systems where *getopt(3)* is provided, this function should be used whenever possible to parse command line arguments provided to a program. When bad options are provided, or that the *-?* argument is provided, the application should print a clear list of the synopsis template to use the program.

Some people will prefer to use *getopt_long(3)* instead, like the GNU Project authors. However, it is recommended to then always provide short -<*c*> options equivalent for the long –<*option-name*> options counterparts. This is particularly true for terminal based programs where users do not want to have to type extremely long commands to specify the options they want to a utility. Also note that although *getopt(3)* is now POSIX standard, *getopt_long(3)* is not.

2.14 File End Of Line (EOL) format

Every line of source code should terminate with the newline character (*'\n'*), which is the convention for text file formats on unix variants, as well as on many other operating systems. The use of *"\r\n"* line termination (carriage return, followed by newline) is only used on MS-DOS derived systems and Windows,

and should be avoided. Good programmers editors will work with standard unix text file formats on such operating systems still, without problems.

As for terminals, which is independent from file format, many cause the cursor to only be reset at column 0 when the '\r' character is issued, and require an additionnal '\n' character to be issued to cause the cursor to move down (optionally scrolling if the bottom of the page was reached). Programs can still use files which do not contain the '\r' characters, and output accordingly "\r\n" sequences for each line output. Similarily, when user input includes "\r\n" trailing line terminators, these can be stripped, and "\n" output to files on disk.

As for many telnet-inspired internet protocols via TCP (including SMTP, POP3, NNTP, FTP, HTTP), it is important to also expect "\r\n" line terminators when receiving, and to output lines also terminated with "\r\n" when sending. The RFCs of the protocol always have the last word according to this. This still however does not mean that files on disk should use the "$|r|n$" line terminators.

2.15 Format of C modules (.c)

It is important to properly modularize projects in order to make them maintainable and expandable. See 6.6 on page 66 for more details about this. Here we are dealing with the format of a headerfile (.h file), as well as that of a C module or program (.c) file. Here are described one by one each element. Each element can be optional, depending on the context at hand. However, their order is critical.

2.15.1 Header

The header can consist of various comments. For instance, it is very common to have the first comment hold the CVS ID as follows:

```
/* $Id$ */
```

This often is automatically expanded to a comment such as

```
/* $Id: memory.c,v 1.7 2004/01/29 21:52:11 mmondor Exp $ */
```

Following often follows a larger comment with the license under which the file is to be distributed. Often this consists of a BSD-style or GPL/LGPL license header. Of course, it may also simply state the copyright and the author's name, or list of contributors following an initial author name.

It is then recommended to follow with a separate, third comment, describing breifly the role of the file/module within the project.

2.15.2 Headerfiles

All the headerfiles (.h files) which this module needs should be placed here, using the *#include* C preprocessor directive. Generally, the headerfiles which are

located under the *sys* directory appear first, and are ideally placed in alphabetical order. Then follow standard headerfiles, also ideally in alphabetical order. Sometimes, certain APIs require headerfiles to be included in a particular order, however.

It is a good idea to use strict compiler warnings so that a module makes sure to not omit a required headerfile. The headerfiles should include the ones holding the prototypes for all external functions which this module uses. Pendantic options of most compilers will allow to have it generate warnings or errors when unprototyped functions are invoked, which is a sign that a headerfile was omited.

2.15.3 Local definitions

When structures and definitions which are local to this module only are needed, here is where they should be defined. This includes *#include*, optionally followed by *typedef*, *struct*, *union*, and *enum*. Common sense must be used to maintain code obviousness when ordering these.

2.15.4 Local prototypes

The local functions which this module defines and is the only one to call should be declared *static*, and their prototypes should be defined here. The only exception is the prototype of the *main()* function, which if defined in this module, should be found here, and cannot be declared *static*. It is common practice to leave a blank line between the *main()* function prototype and the *static* local ones.

2.15.5 Local globals

The variables which are to only be used by this module, but which should be global to all functions of it, should be declared here, and should be *static*.

2.15.6 External globals

The variables which are to be exported to other modules, for which we provide a headerfile with the *extern* keyword should be declared here, and should not be *static*. Other modules which include our headerfile will be able to access these variables. Obviously, our headerfile will need to declare these external variables using *extern*.

2.15.7 Static functions

Then can be declared the actual local only *static* functions which correspond to the above prototypes. The only exception is *main()*, which cannot be declared static, but usually can be found here when this module consists of the main executable one of a command, before any static functions. A few blank lines are usually left between the main function and the static functions block. Similar blank lines delimiters should be used with comments whenever necessary to

separate function blocks which are to be considered independent working sets. At least one blank line should separate each function definition from eachother.

2.15.8 External functions

Other functions which should be available to other modules which we declare should be declared here. Obviously, we also should have a headerfile which holds the prototypes of these functions. Other modules which include that headerfile will be able to call these functions if the program has been linked with this module. These kind of external functions are generally only found in libraries. However, it is always a good idea to implement libraries to reduce the amount of duplicated code to a minimum. Weither to provide dynamically/shared or statically linked libraries depends on the project context.

There usually are a few blank lines separating the static functions block from the external ones, sometimes with a comment.

XXX See order, possibly that external functions should be before static ones. Check NetBSD code.

2.16 Format of C headerfiles (.h)

Most headerfiles are made to allow access to a common library by multiple C modules, although there can be rare exceptions where the local definitions of a module are declared separately into a headerfile because there are many or because the headerfile needs to be automatically generated by the build scripts.

2.16.1 Header

Like in the case of C modules described above, this section usually holds comments to optionally hold the CVS ID of the file, as well as licensing terms concerning this file. Then follows a third comment with a short description of the file's goal.

2.16.2 Preprocessor recursion protection

Both for performance and to prevent headerfiles from needing to be included in a particular order, it is a great idea to provide protection against self-recursion. If all the headerfiles have this kind of protection, it then becomes safe to include all other files which we need in the following section, without causing one which includes this one to cause recursion.

The common way to prevent from self-recursion is to use a preprocessor *#ifndef* conditional directive as well as a project-unique *#define*. All other sections of this headerfile from now on should be enclosed within the conditional so that they are ignored by the preprocessor if this file already was included. Here is an example:

```
#ifndef STDIO_H
#define STDIO_H
```

```
/* ... all other sections ... */

#endif /* !STDIO_H */
```

2.16.3 Headerfiles

The other headerfiles which this file needs should be included here. As with C modules, the headerfiles within *sys* should be included first, and it is a good idea to make the files appear in aphabetical order.

2.16.4 External definitions

These include macros which are part of the library, as well as constants definitions. These are generally defined with uppercase names. Custom types with *_t* prefix or uppercase names may also be defined here with *typedef*. Under some circumstances, it may be necessary to use preprocessor conditionals to verify if a name already has been defined prior to redefinition. This however generally is the result of some conflict which can usually be resolved by choosing a unique, non-conflicting name for the object to define.

2.16.5 External prototypes

Here are defined the prototypes of the functions of the library, generally prefixed with the *extern* keyword.

2.16.6 External globals

The static memory buffers which the library supplies to the caller (if any) can be defined as *extern* here with the corresponding variable type and names.

2.17 Lint tools and tips

For a long time project maintainers use tools such as *lint(1)* to discover a variety of common bugs which can usually easily be fixed before a project can be compiled and used. Although this is not always the case, since modern compilers become better at displaying warnings on common bugs lately, some lint-style comments and techniques are still being used by many programmers to make the work more readable for humans too. Here are described a few techniques which we recommend because they are very useful at code maintenance stages to better understand what the author wanted to do when programming. We recommend to read the *lint(1)* manual page for more details. We herein only describe the most useful of the *lint(1)* comments since they are of great importance to program manageability. Even when *lint(1)* is not used, it becomes of use to humans anyways.

2.17.1 /* CONSTCOND */

In general, preprocessor only conditional directives evaluate expressions using constants rather than variables, such as *#ifdef*. Those constants are usually defined using *#define*. However, there are instances where actual code may need to perform a conditional expression with constants. However, this can often generate compiler and/or *lint(1)* warnings. Also, a conditional using constants is sure to always return the same value, and it is often questionable to use those in the code, as preprocessor conditionals are more suitable in these situations. The author can clearly define that the intention to use conditionals with constants only in actual code is justified, using the */* CONSTCOND */* comment:

```
#define TRUE    /* CONSTCOND */(1)
```

Thus,

```
if (TRUE) {
```

expands to:

```
if (/* CONSTCOND */(1)) {
```

we know that 1 will never change, but the author clearly demonstrated the purpose of this constant conditional. Another example, which is more often found in real world situations:

```
#define DLIST_INIT(lst) do {                    \
        (lst)->top = (lst)->bottom = NULL;      \
        (lst)->nodes = 0;                       \
} while (/* CONSTCOND */0)
```

2.17.2 /* NOTREACHED */

It is not uncommon to have programs execute an endless loop, or to have alternate custom *exit(3)* replacement functions which terminate the program thus never returning. Because it is not always obvious when maintaining the code to detect these conditions, this comment is very useful. Here are a few examples. First is one which simply demonstrates the author's intention to never execute code after */* NOTREACHED */*:

```
for (;;) {
        /* ... */
}
/* NOTREACHED */
```

and the following describes a situation where a custom exit function is used:

```
void
freeall(int ret)
{
        /* ... free resources ... */
        exit(ret);
}

if (condition) {
        freeall(EXIT_FAILURE);
        /* NOTREACHED */
}
```

2.17.3 /* ARGSUSED */

In many cases an API can provide a way to pass an optional arguments pointer or such to a callback function. However, this function may or may not need this argument. In cases where a function is known to not use at least one of its passed arguments, it should clearly be defined as such, so that it does not be considered a bug.

Here is an example where the POSIX standard *sigaction(2)* defines the handler function to be handling the signal as follows:

```
void (*sa_handler)(int)
```

The *int* parameter of that function will tell the callback function (signal handler function) which signal occured causing it to be called. However, if this function performs the same action for any signal that it gets, or that it does not need to know which signal caused it to be called, the handler function would be declared as follows:

```
/* ARGSUSED */
void
sighandler(int sig)
{
        /* ... we don't use sig, voluntarily ... */
}
```

Thus, this clearly shows the author's intention to not use the argument.

2.17.4 /* FALLTHROUGH */

When a *switch () case* statement voluntarily avoids using a break statement to fall back to the other case condition, it is very useful to use a */* FALLTHROUGH */* comment. This helps in debugging where it is now certain that the missing break into a *switch ()* is not the result of a bug, but was voluntary. Example:

```
switch (c) {
case 'a':
```

```
        /* ... */
        break;
case 'b':
        /* FALLTHROUGH */
case 'c':
        /* ... */
        break;
}
```

This clearly specifies that cases for 'b' and 'c' should be executing the same code found in the 'c' case. The *break* statement omission into the 'b' case is not a bug.

2.17.5 /* NOOP */

This comment is useful to allow humans to know that a function intentionally contains no statements, that is, consists of a stub for now. Some object-oriented code may need to be specified a "method" function to be called for object-specific internal processing of some operation. It is not rare to supply empty functions to such APIs when *NULL* is not checked for before calling function pointers by the library. Example:

```
int
somefunc(void)
{
        /* NOOP */
}
```

2.17.6 (void) casting of function calls

When a function is known to not have a *void* return code and that the programmer specifically does not want to test its return value (he normally should, but under some circumstances this is useless in the context), the function should be casted with *(void)*. For example:

```
(void) fflush(fh);
(void) fclose(fh);
```

This demonstrates that it was the author's intentions to voluntarily drop the possible return results of the function. This not only is a good habit for the programmer as he needs to remember which functions can return an error code, but it also assists in debugging and security code audits. A space should be found between the *(void)* casting and the function, as opposed to normal variable casting where no spaces are used, which additionally allows to easily track where these occurences can be found within the code using *grep(1)* or other text searches.

3 Common security issues

This section describes some of the most common bugs which generally become security concerns in software. All of the bugs mentionned here are related to buffer overflows or bad pointers. It is true that most high-level languages are safer to program in considering these factors. However, this does not mean that C code cannot be secure. In fact, an experienced and aware C programmer will generally generate code which is much more efficient with C, while taking care not to introduce security holes. Moreover, being dependent on a high level language often means that when bugs creep within it, the programmer has no control and still becomes subject to security holes.

High level languages also generally are quite less powerful than C programs can be, and often execute much slower (such is the case with Java, which both requires an aweful amount of RAM because of its garbage collection to reach a decent performance, which still remains about two two ten times slower than C++ generated code in all cases. And C programs tend to even be faster than C++ ones. This means that when performance is critical high level languages usually cannot do the job properly. Java simply cannot be used in such circumstances, even with a Just In Time Compiler and 100 times the RAM needed for an equivalent C program.

All of the following security issues can be solved with careful programming and the fact that nothing must be assumed; Everything must have defined behavior and thus always operate on known variables or constants.

3.1 Buffer overflows, stack smashing and illegal memory access issues

3.1.1 Description

Because C leaves total control to the programmer, it is quite easy to write passed the end, or before the starting point, of allocated buffers. These buffers may either be allocated on the heap or on the stack (I.E. using auto variables).

When one writes outside of the actual allocated area, several desastrous results can happen. When the buffer is allocated on the stack, this even means that it is possible for the function to even jump to unexpected code upon returning rather than resuming to the caller's stack state. This is generally called stack smashing.

When we write past the bounds of a buffer which is allocated on the heap, we can accidentally overwrite other variables. We possibly even can overwrite the actual program text, although real Operating Systems will cause the program to exit before this happens (which is usually still a problem in itself, though).

In the case where the OS causes the program to exit because it reads or writes to illegal memory locations, this protects the program from being exploited from a user (which potentially can be remote depending on the context), but this still results in a Denial Of Service, since the program will no longer run to perform its critical tasks.

What often can happen when a program writes past the bounds of stack or heap allocated buffers is allow users to execute arbitrary code, leading to security problems such as privilege escalation. This can become a real threat not only to the application at hand but to the whole system in some circumstances. For instance, OpenSSH has a long history of remote superuser escalation through buffer overflows. As it has to run as the Unix superuser to be able to perform its normal tasks, small buffer overflow bugs often prooved fatal to simple attacks on machines running it.

This does not consist of a manual on how to crack in one's system (the author really means cracking here, since hacking does not correspond to what most people beleive. The author considers himself a hacker (computer programmer), but is far from being a cracker). We therefore will not expand on the subject on particular stack smashing techniques. Basically they can allow such control as to even permit a remote user from sending custom shell code and executing arbitrary commands, either as an unprivileged user or sometimes even as the superuser on the target machine depending on the circumstances of the bug and software. What we want to point here is that the C programmer makes the difference weither programs are secure or not, and we want to provide some tips to avoid the most common security related problems found in C programs.

Nontheless, these simple common cases and examples can never guarantee the security of a system; programmer errors exist, and there are complex algorithms in which it can be easy to cause problems even when buffer overflows are not possible. An example is when using function pointers. Whenever a function pointer is wrong, the next caller using that pointer will obviously not effect the intended behavior. Some bugs which don't cause crashes and remain unsuspected can also sometimes cause gradual corruption of internal data structures which could eventually lead to possible overflows and/or crashes as well, only after the application has been running for a lengthy period of time or under particular cumulated circumstance sets.

3.1.2 Prevention

As a general rule, it is important to define the values (either as variables or constants) consisting of the length of the buffer to use for a particular purpose. All functions dealing with that buffer should ensure to not exceed the boundaries defined by the corresponding variable or constant. That is, when running pointers through such a buffer, the maximum boundary should be calculated relatively to the start of the buffer and its size, and the running pointer loop must make sure not to exceed the boundary address. When dealing with indexing loops, respective comparision of the current index with the buffer size may be used. Of course, a buffer size may be represented in the maximum number of units it may hold, respectively to its unit size. This means that the buffer sizes will not always be of *size_t* or *ssize_t* type, of course.

Several standard string functions are well suited to deal with restricted buffer sizes, which may be used over their unbound equivalents. For instance, *strncmp(3)*, *strncpy(3)*, *strncat(3)*, *strncasecmp(3)*, *sprintf(3)*, *fgets(3)*, *strl-*

cpy(3), *strlcat(3)*, *memcmp(3)*, *memcpy(3)*, *memmov(3)* and *memset(3)* can all be specified a bound limit which it should respect. Such bound-sensitive functions are especially important to use when the source of the input, data or parameters originate from a user or are arbitrary.

It also is a good idea to use assertions in the code (at least in the form of macros which can be compiled in when debugging symbols are enabled for testing, but which can be compiled out otherwise for performance). In many cases, however explicit assertions may be necessary and may cause refusal of access to a user, for instance when we are dealing with authentication and security related code. See 6.5 on page 65 for more information on assertion techniques.

The use of enumerations can also be useful to avoid very common bugs when long integer to data mapping lists need to be defined. 2.11.4 on page 34 has example details about a simple yet efficient way to perform range checking and indexing.

When dealing with C strings, it is very important to always ensure that the NIL ('\0') character terminator is provided within the bounds of the buffer. It is equally important to always perform both '\0' boundary and buffer size boundary checking when writing string related functions, at all times.

Under particular circumstances it may be difficult to evaluate the necessary buffer size and bounds to use. This can happen in the case of some codecs, compression or encoding methods, especially when decoding/decompressing. It also may happen in the case where it isn't ideal to allocate a very large buffer which may not be fully used often, but that it has to be large enough in some rare cases. In such circumstances a system involving carefully crafted dynamic allocation techniques with threshold/bound sanity checking may be ideal. See 6.7.3 on page 66 for more details.

Before accessing pointers it is often useful for functions to use assertions, verifying that they are not *NULL*, and if necessary, that the address is within allowed range before using the pointer. A *NULL* pointer points to invalid data. Of course, when for performance considerations such assertions are avoided because the caller is known to always provide a proper pointer, it is advisable to use an assertion macro which can be compiled in and out easily via a preprocessor definition. See 6.5 on page 65 for more details on how to implement this. This way, the program can first be compiled in a slower version for testing, and then compiled again into a more optimized version for distribution and/or production use when it throughly has been tested.

When calling functions which use multiple arguments with *stdarg*, such as *snprintf(3)*, it is very important to never provide a user supplied string as the *fmt* argument. The obvious reason for this is that the user could supply a string embedding '%' characters, which would be causing a lot of trouble. It is important to always explicitely supply the *fmt* string outselves, which can then be optionally followed by user supplied data.

3.1.3 Example

The following example shows a simple yet secure way to read in lines provided by an arbitrary source via standard input, into an allocated buffer. The *BUFFER_SIZE* definition as well as the *malloc(3), fgets(3)* functions consist of the important factors to look at. Of course, in a real world situation it also may be necessary to monitor for an timeout event, and/or to write a function which can dynamically resize the buffer, filling it with the total of all input lines. There exist as many possible implementations as there are situations.

```
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE     1024

int     main(void);

int
main(void)
{
        char    *buffer;

        /* First allocate buffer */
        if ((buffer = malloc(BUFFER_SIZE)) == NULL) {
            (void) fprintf(stderr, "malloc() - %s",
                strerror(errno));
            exit(EXIT_FAILURE);
        }

        /* Read each line in buffer safely */
        while ((fgets(buffer, BUFFER_SIZE - 1, stdin)) != NULL)
                (void) fprintf(stdout, "%s", buffer);

    exit(EXIT_SUCCESS);
}
```

3.1.4 Other prevention methods

- There exist implementations of string libraries which do not use native C strings but rather a BCPL BSTR variant, where all string functions know the actual string boundaries. This can be considered useful for some projects. Such a string library is also trivial to implement. An example of such a library is the one used within the qmail MTA package. The Pascal language uses such strings by default as well.

- There exist compiler modifications which can cause the applications to exit as soon as a stack smashing condition is detected. Although this slows down executables, adding stack related checks at function entry and exit points, and that an application should generally rely on proper auditing and programming techniques rather than on such a solution, this can be useful to compile very security critical applications. An example of this is XXX

- Many new unix applications are now programmed in a way to perform the less operations possible as the superuser. Privilege separation methods can be employed when an application only needs to execute some instructions as the superuser, where a specific process of the application can remain running as the superuser and execute requests from the other processes of the application, for instance. This minimizes the possibility that exploits be able to escalate privileges in a way to compromise the entire system. The use of the *chroot(2)* system call is also encouraged a lot by recent software. Moreover, administration frontends should ideally not have to execute as the superuser, nor the services. To achieve this, some new applications use virtual users rather than unix users. This not only allows other application-specific permissions management methods than the standard unix ones, including ACLs to be used, but often restricts exploits to compromise one or a few unprivileged unix users in the case of serious exploits rather than the whole system.

- Some systems can further use *systrace(1)* such as NetBSD, to add syscall based restrictions on a process basis. It also has a *veriexec(4)* which can restrict applications which are allowed to execute to the strict set the administrator wants, although this isn't particularily related to C programming.

- FreeBSD systems have *jail(1)*.

- Some systems even use virtualization techniques so that multiple copies of the OS can run on a single system. The master host running the other operating systems virtually, can be configured to automatically restore ones which have been exploited by previous backups, etc. That master host is certain to not be compromized when one of its host operating systems are. An example of an innovation in this area is Xen (XXX) with NetBSD/Xen (XXX) which provide unprecedented performance in this area. Another example is User Mode Linux (UML) as a lower performance variant. Some people will even go as far as emulating a processor, such as the Java Virtual Machine, or running NetBSD/Vax under a Vax emulator ran on a master NetBSD system. This technique obviously reduces performance greatly.

- Use of strict debuggers and tools is encouraged during the testing of security critical software. When software is released for production purposes it

often is too late to use some of the methods; Especially ones which modify the executables and runtime environment also affecting performance.

- A decent resource of source auditors and beta-testers is highly recommended; Such teams help to discover the most bugs possible in the software testing and release engineering phases, before production releases are issued.

4 Common portability issues

Portability refers to the ability of the software to easily compile and run on other platforms (Operating Systems) or architectures (hardware, processors). It also pertains to the compatibility between those different versions according to file formats and network communications, thus, binary compatibility. Here are presented various tips to solve very common issues related to these topics.

An awesome portability feat consists of the NetBSD operating system (open source and distributed at *http://www.netbsd.org*). It consists of the operating system which compiles and runs on the largest number of architectures and is derived from 4.4BSD, licensed under a BSD-style license (less restrictive than the GNU GPL under which the Linux kernel is released). Its source can be built for any of the supported targets using cross compiling from alot of popular platforms, using a GNU GCC toolchain. It is a great environment to work on, providing POSIX APIs, and an incredible source of academic information. It is still actively maintained and thus remains a state-of-the-art modern Operating System evolving with the latest technologies, despite the long history of its heritage. Its revolutionary architecture-independent device driver framework is an especially great component of NetBSD, allowing device drivers to be compiled as-is on any architecture which can support the hardware via its bus. For instance, PCI drivers compile as-is for all architectures which have a PCI bus. Moreover, its libc, kernel APIs, devices and their *ioctl(2)* are well documented via manual pages. (See *bus_space(9)* and *bus_dma(9)*). NetBSD also is a full Operating System, including the kernel and userland, unlike Linux which only consists of a kernel, where distribution maintainers must do the work to assemble the userspace and kernelspace parts (such as Debian, RedHat, Slackware, etc, which can be considered as Linux-based Operating Systems, for which multiple architecture support is generally limited).

Such a great operating system is well worth studying, especially considering the clean code it features through its tree. Another great feature is that it supports excellent binary compatibility for several operating systems on the same architecture it runs on, such as support to run Linux native binaries. Binary compatibility implementation, to a certain point, can also be considered portability-oriented code.

Let's first define what *endian byte order*, *native word size* and *stack growing direction* consist of.

4.1 Definitions and examples

XXX

4.1.1 Endian byte order

This relates to how a particular processor represents a value in binary form internally. Most languages abstract this totally, but because C allows bitwise operators on the native processor words (for instance, int generally corresponds

to a native processor word), it is important for the programmer to know about these differences. First note that the byte order is only significant when dealing with multiple byte sized words. This means that there is no special precautions to take when dealing with individual bytes.

In this document, we often relate to *network byte order* and *host byte order*, which are universal terms. The *network byte order* representation can safely be used in binary files and accross networks, and actually corresponds to *big endian byte order*. The *host byte order* consists of the internal byte order in use by the current processor the code is running on, and may correspond to *big endian byte order* or *little endian byte order*, depending on the architecture.

In the *big endian byte order* (this includes the *network byte order*), the most significant bytes of a multibyte word is placed at the left of the least significant ones, such as the way we order the arab base-10 numbers on paper. They however are base-256 since each byte may hold 256 possibilities. Consider *u_int16_t* for instance, which is known to hold two 8-bit bytes internally (two *u_int8_t*). Let's assign the value 258 to our *u_int16_t*, which exceeds one byte for our example. The internal representation consists of 0x0102 ((0x01 * 256) + 0x02) = 258. The *host byte order* of the popular processors PPC, m68k, sparc, ultrasparc is *big endian byte order*.

In the case of *little endian byte order*, the most significant byte of a multibyte word is placed at the right of the least significant ones, contrary to with the *big endian byte order*. In our previous example where our *u_int16_t* variable is assigned 258, the internal representation becomes 0x0201 ((0x01 * 256) + 0x02) = 258. The *host byte order* of the popular i386 and derived processors is *little endian byte order*.

Although our examples were restricted to 16-bit words, this expands to other word sizes. For instance, the *big endian byte order* for 0x12345678 becomes 0x78563412 in *little endian byte order*.

Most BSD derived systems, or the ones importing the BSD sockets API (most networking capable Operating Systems) provide the *htons(3)*, *htonl(3)*, *ntohs(3)* and *ntohl(3)* functions which are portable and can be used to convert 16 and 32 bit words between *host byte order* and *network byte order* and vice-versa. Of course, on a *big endian byte order* architecture, these will perform no conversions, while on a *little endian byte order* one they will perform unconditional reversal of all the bytes within the supplied word.

Operating systems which support the sunrpc (Sun Remote Procedure Call) also provide endian conversion functions within the XDR(3) API (this includes Solaris, the BSDs and Linux). These functions are less restrictive in that they can be told to convert from 8 to 64 bit sized words, performing conversions from network to host or vice-versa as necessary. Moreover, this library also includes I/O based wrappers which are ideal to use when reading or writing from or to files via an stdio FILE or over the network via filedescriptors.

There unfortunately is no other standard for architectures to define their endian type so that programs may provide their own endian conversion functions if wanted. This however can be supplied easily at compilation time using a definition, or via a configuration file if run-time endian changes support is

needed. These functions are easy to write using conditionals and bitwise operators. If those conversions are expected to occur very frequently in the code, it is recommended to program the 16-bit and 32-bit byte swapping functions in assembly. The 64-bit one can then perform two *bswap32()* calls for instance. Most libc and XDR(3) implementations internally do this.

There also exists the *pdp byte order*, but that hardware is generally rare today it only was used by the early DEC PDP processors. Relative to the *big endian byte order* (or *network byte order*), a 32-bit word which represents 0x01020304 in network byte order becomes 0x02010403 in *pdp byte order*.

An example C implementation of byte swapping functions which can deal with *big endian* and *little endian* byte orders follows:

Headerfile (byteorder.h):

```
#if defined(_LITTLE_ENDIAN)
#define BYTEORDER_NETWORK16     byteorder_bswap16
#define BYTEORDER_HOST16        byteorder_bswap16
#define BYTEORDER_NETWORK32     byteorder_bswap32
#define BYTEORDER_HOST32        byteorder_bswap32
#define BYTEORDER_NETWORK64     byteorder_bswap64
#define BYTEORDER_HOST64        byteorder_bswap64
#elif defined(_BIG_ENDIAN) /* !_LITTLE_ENDIAN */
#define BYTEORDER_NETWORK16(w)  (w)
#define BYTEORDER_HOST16(w)     (w)
#define BYTEORDER_NETWORK32(w)  (w)
#define BYTEORDER_HOST32(w)     (w)
#define BYTEORDER_NETWORK64(w)  (w)
#define BYTEORDER_HOST64(w)     (w)
#else /* !_BIG_ENDIAN && !_LITTLE_ENDIAN */
error "Undefined byte order";
#endif /* _BIG_ENDIAN/!_LITTLE_ENDIAN */

#ifdef _LITTLE_ENDIAN
extern u_int16_t byteorder_bswap16(u_int16_t);
extern u_int32_t byteorder_bswap32(u_int32_t);
extern u_int64_t byteorder_bswap64(u_int64_t);
#endif /* _LITTLE_ENDIAN */
```

Library C module (byteorder.c):

```
#ifdef _LITTLE_ENDIAN
u_int16_t
byteorder_bswap16(u_int16_t w)
{
        u_int8_t *p = (u_int8_t *)&w;

        return (u_int16_t)(p[0] << 8 | p[1]);
}
```

```
u_int32_t
byteorder_bswap32(u_int32_t w)
{
        u_int8_t *p = (u_int8_t *)&w;

        return (u_int32_t)(p[0] << 24 | p[1] << 16 |
            p[2] << 8 | p[3]);
}

u_int64_t
byteorder_bswap64(u_int64_t w)
{
        u_int32_t *p = (u_int32_t *)&w, t;

        t = byteorder_bswap32(p[0]);
        p[0] = byteorder_bswap32(p[1]);
        p[1] = t;
        return w;
}
#endif /* _LITTLE_ENDIAN */
```

The *htons(3)* and *htonl(3)* functions could then be reproduced using *BYTEORDER_NETWORK16()* and *BYTEORDER_NETWORK32()* macros, and *ntohs(3)* and *ntohl(3)* by the *BYTEORDER_HOST16()* and *BYTEORDER_HOST32()* macros, respectively. Of course, *_LITTE_ENDIAN* or *_BIG_ENDIAN* will need to be defined (which can be done at building autoconfiguration, to pass *-D_BIG_ENDIAN* to the compier options for instance, or supplied through an architecture-dependent headerfile). Before writing to network socket or to file, *BYTEORDER_NETWORK*()* shall be used on the words, and after reading from network socket or from file, *BYTEORDER_HOST*()* shall be used to convert the words back. Of course, this is only an example and there may already exist byte order conversion functions to work with on your environment. A thing which is nice using these macros is that no function will even be called on *big endian* architectures as the *host byte order* already corresponds to the *network byte order*. On the other hand, *little endian* architectures will be allowed to perform the necessary conversions.

4.1.2 Native word size

This comes in consideration depending on the processor being used, and can also be affected from a C program's perspective by the compiler. Here are examples: On Borland Turbo C/C++ compilers for the 8086 and up (including i386), the size of an *int* is 16-bit, which corresponds to the native word size of the 8086 (or i386 in default 16-bit mode). Using most modern C compilers for i386, the size of *int* is 32-bit, corresponding to the native word size of the i386 in 32-bit mode. On some C compilers targetted at 64-bit architectures, the size of *int* is

64-bit, corresponding to the native word size of the processor. However, using the GNU GCC compiler on a 64-bit processor, the size of *int* remains 32-bit, while *long* is expanded to 64-bit (which remains 32-bit for i386). Depending on the portability scope of the software, it may be a burden to account for all those differences when using the *int* and *long* types.

A common solution to this is to only use *int* and *long* when the size of the internal word size is irrelevant and that it is known that their size will be large enough to hold the values at hand. This often can be used for indexing in loops for instance. Then typedefs are used, defined using *typedef* by a file included by the program (or included by *</sys/types.h>*) which define *u_int8_t*, *u_int16_t*, *u_int32_t*, *u_int64_t*, *int8_t*, *int16_t*, *int32_t*, *int64_t*, *ssize_t*, *size_t*, *quad_t*, *u_quad_t*, *off_t*, etc.

The latter types can then be used through the code, and can be changed to internal definitions corresponding to the architecture and compiler in use to compile the software. The *u_int8_t* and *int8_t* types can also be used where 8-bit values are specifically needed, since some C implementations (and C library implementations) do not use a *wchar_t* and corresponding library for wide character support but instead changed the size of the *char* type and the C library string functions to deal with it. The native 8-bit type of the compiler can then be mapped to the *int8_t* ones by the headerfile. Using an ANSI C89 standard compiler has no such issues pertaining to the *char* type, and it can be mapped directly to the 8-bit typedefs. It also sometimes is a good idea to define *pointer_t* to represent the size of a pointer or address for the architecture.

Example typedefs, which are commonly used on i386 and m68k systems with GCC:

```
typedef unsigned char           u_int8_t;
typedef char                    int8_t;
typedef unsigned short          u_int16_t;
typedef short                   int16_t;
typedef unsigned long           u_int32_t;
typedef long                    int32_t;
typedef unsigned long long      u_int64_t;
typedef long long               int64_t;
typedef u_int32_t               pointer_t;
```

If our target architecture consisted of a 64-bit ultrasparc processor, always with GCC, the *long* native type would now correspond to 64-bit. Therefore, the following would be used:

```
typedef unsigned char           u_int8_t;
typedef char                    int8_t;
typedef unsigned short          u_int16_t;
typedef short                   int16_t;
typedef unsigned int            u_int32_t;
typedef int                     int32_t;
typedef unsigned long           u_int64_t;
```

```
typedef long                    int64_t;
typedef u_int64_t               pointer_t;
```

4.1.3 Stack growing direction

The stack is the Last In Last Out (LIFO) buffer which is generally used to store the C functions arguments and return address before calling a function. Although some might argue that C is not a stack-oriented language and that it is possible for the compiler to generate code which passes arguments into registers for optimizations, the recursivity and variable arguments features of the language imply that a stack is needed to store the arguments internally.

The stack grows in a particular direction, upwards or downwards, depending on the architecture (processor). Some architectures also support stacks which grow in both directions. As values are pushed onto the stack, it grows, and as they are popped back out it shrinks back. A stack which grows upwards has it's stack top address at the end of the actual LIFO buffer, while a stack growing downwards has it's top address at the top of the LIFO buffer in memory.

When solely dealing with C, this generally is not a concern, unless dealing with some assembly sections which have to interface with the C code, or when dealing with context switching functions in the implementation of threading systems and schedulers or of *setjmp(3)/longjmp(3)* and such functions.

For instance, the m68k processor has an *upwards stack growing direction*, while the i386 has a *downwards stack growing direction*.

4.2 Compiling on various platforms

Obviously, it is ideal to resort to the standard widely used APIs as much as possible, such as ANSI, POSIX, SUSV and X/Open Portability Guide defined functions. This is generally easy to do when programming on BSD, Linux and other UNIX or Unix-like systems. This isn't always obvious to do on such unstandard platforms as Win32. When possible, POSIX and such compatibility libraries may be installed on Win32, such as cygwin or mingw. Also, using OpenGL and DSL libraries is also favored over using OS-specific graphics libraries for portability. GTK and QT widget toolkits have also been ported to various platforms and architectures.

When Win32 supplies a form of a highly popular API, however, such as winsock.dll implementing BSD sockets API, because of inconsistencies with the BSD standards the networking code will often need to still be isolated from the rest of the code in order for preprocessor directives and conditionals to compile the proper code depending on the architecture.

It is to be noted that apart from curses API, which provides text-based GUI (Graphical User Environment) and GTK providing a graphical GUI, which can be ported to Win32, providing a common API for GUIs, most GUI dependent programs will require special tricks to be portable among platforms, because the native GUIs and their related APIs vary on them. It often is easier to program HTTP frontends which simply require users to use any standards compliant

HTTP (Web) browser than to code platform independent programs which need to use GUI libraries. Of course, this is not always desirable.

What can be done is abstract the GUI and other platform-independent functions which the program needs to access into separate modules, such that only those modules need modifications in order to port the software among platforms. Fortunately, there also already exist such abstraction libraries, such as SDL, OpenGL and related Glut toolkit, OGRE, curses, GTK and the standard ANSI library.

It is important to consider that even when using two Unix-like platforms differences may exist on various levels. For instance, BSDs are known to use 64-bit values for *off_t* since some while, when Linux (even 2.4 kernels) still appear to use a 32-bit type to represent *off_t* by default. This means that it might be more appropriate to convert *off_t* to *int64_t* when it is necessary to perform word size dependent operations on the offsets, and to then convert the *int64_t* value back to *off_t* as necessary to comply with the APIs.

A great aid into platform-independent compiling consists of the GNU tools, such as *autoconf*, *automake* and *gmake*, in combination with the *GCC* compiler. It is beyond the scope of this document to provide more information on those tools, but their documentation is included by the Free Software Foundation, as well as full source code for each of these utilities, on the *http://www.gnu.org* site. *autoconf* provides means to detect platform and architecture dependent differences so that initial setup of the auto building scripts may adapt compilation automatically. Although it generally takes some time for C programmers to properly learn how to use it, it generally proves worthwhile in the long run. One especially has to get acquanted with the *m4* macro preprocessor mixed with POSIX */bin/sh* shell scripts.

4.3 Compiling for various architectures

Two factors of interest to watch for when the software is to be compiled on various architectures or processors are *endian byte order, native word size* and *stack direction.*

It thus is a good idea to use common typedefs such as [*u_*]*int*<bits>*_t* types which are provided by most platforms and can be used including <*sys/types.h*>. For instance, when an application specifically needs to use an unsigned 64-bit word, it can then use *u_int64_t*. This avoids problems where an *int* or *long* has a different internal *native word size* among compilers and/or architectures. Where necessary for compatibility with APIs, the values will need to be converted to the proper size-dependent size variable for internal processing and then back to the native type which the API requires.

It is not uncommon to optimize operations on known word sizes by using left and right bit shifting and rotation instead of generally more complex and slower arithmetic to perform the same operation. This can however lead to unexpected results when the same code is compiled on another architecture, which has another *endian byte order*. For this reason, it generally consists of a good idea to first ensure that the value to operate on be in network/big endian byte order

before performing the bit shifting operations, and to then convert back the value to host byte order after the operation if necessary. Because converting a word to network order will perform nothing on big endian architectures but will on little endian ones, and that the same applies to conversion of a word from network order to local/host order, this allows the code to be portable among different processors.

The *stack growing direction* is usually only of interest when programming assembly sections. However, because this may be necessary in the realization of some projects, such as kernels, and that the assembly then interfaces with the C code, the *stack growing direction* will need to be taken in consideration in such cases.

4.4 Binary compatibility between ports

For programs to be able to provide binary compatibility between eachother when running on different platforms and architectures, it is important to take in consideration the following factors: The *native word size* of APIs and architecture in use and the *endian byte order* of the binary files and data trasmitted over the network.

The reason why the native word size of both the architecture and libraries (APIs) are to be taken in consideration is that for instance, the *off_t* popular typedef might not have the same size depending on the platform. Similarily, the *size_t* and *ssize_t* types might not have the same word size. It is then ideal to make sure to convert the necessary values to *u_int32_t*, *int32_t*, *int64_t* or *u_int64_t* which size is well known before writing to file or sending over the network, for example.

As far as *endian byte order* is concerned, the values then should be converted to *network byte order* before writing into a binary file or sent over the network. When read back from a file or received from the network, these values should then be converted back to *host byte order*. Internally, the *network byte order* and *host byte order* conversion macros and/or functions will do nothing or perform the byte swapping as required, depending on the *endian byte order* of the current processor.

Using these methods, binary compatibility can be maintained between multiple architectures communicating over the network or sharing files. The sunrpc (Sun Remote Procedure Call) network based protocol uses the XDR(3) library for instance, to accomplish network compatibility with other hosts. However, another alternative is available which may suit better some projects. This involves converting the in-memory format to a standard text representation (which is not subject to endian byte order issues), but also involves the CPU cycles of converting that text representation back into memory values before they can be used by the other network end or file reader.

Popular formats for this type of work are ASN.1 and XML, although there also exist a variety of other text-only standard and custom representations which can be used to achieve the same results. ASN.1 and XML popular formats are especially useful where the software needs to share the format with other

packages or third party software. XML is well suited to import and export data among various applications of different vendors for instance. The parsing cost of XML input in CPU cycles is considerable comparatively to some simpler text formats or to binary endian conversions, however.

5 Proper resources allocation and release

6 Building blocks requirements for most projects

6.1 Compiler and linker

-cross compiling notes too

6.2 Indenting editor and utility

It is recommended to use an editor which is powerful enough to handle multiple buffers/files and screen splitting, syntax highlighting and autoindenting with special C support. Such good, free and popular editors are *VIm(1)* (a much improved *vi(1)* clone) and *emacs(1)*. Both require some using to but are well worth learning. The author especially likes working with *VIm(1)*. For both, there exist console/text and GUI implementations, which run on all unix systems, as well as on win32. To properly use such featureful editors one is recommended to read their tutorial and reference manuals (which are also provided with them). A good example *~/.vimrc* configuration file for *VIm(1)* to program in C follows:

XXX TO UPDATE

```
set nocompatible
set ts=8
set noai nocin
set ruler
set showcmd
set equalprg=indent
set noerrorbells
set vb t_vb=
syntax on
augroup filetypedetect
 au! BufRead,BufNewFile *.h setfiletype c
augroup END
autocmd Filetype c set ai cin sw=4
syntax off
autocmd Filetype c syntax on
```

For emacs, the following may be added to the *~/.emacs* file:

```
(fset 'c-set-style 'bsd)
```

Sometimes, we need to deal with badly formatted code which we did not write. In such cases the very useful *gindent(1)* (GNU indent) utility is recommended. To format code in the general style described in this document, one may use for instance the command:

```
$ gindent -kr -ncs -ts8 -i4 <file.c>
```

Of course, some people also have been known to use any editor to write their code and to then format it using *gindent(1)*, too.

VIm(1) can be obtained at: *http://www.vim.org*
emacs(1) can be obtained at: XXX
gindent(1) can be found at: XXX

When using X11 with a text terminal, it often is most useful to also use the *screen(1)* utility, which allows one to have multiple terminals into a single one. The author usually uses *aterm(1)* X11 (*X(7)*) terminal with *screen(1)*, and *VIm(1)* in console mode. However, to have more decent colors when using syntax highlighting, it was useful to also set custom colors for *aterm(1)* through the use of X resources. This is generally done through the *~/.Xdefaults* file. This file can then be loaded using the *xrdb -load $HOME/.Xdefaults* command in the *~/.Xinitrc* file.

Some people prefer to use full environments including a custom editor with menus to build and test C programs. However, these are usually commercial programs, and generally tend to not be as productive as terminals in the long run. Most also require one to use a commercial operating system, which often can be unstable and unsuitable for serious programming. Another problem which often arise is that these environments often encourage the use of unstandard function libraries, making the organization eventually very dependent on a particular commercial product instead of comforming to open standards. This problem is most frequent with BASIC implementations. Some environments are open source, such as *Eclipse* enabled with with *CDT* (*http://www.eclipse.org*) and KDevelop (XXX). However, it must be considered if dependency on such heavy environments is wanted, and if learning to use them appropriately takes as much time as simply using terminals. Emacs can also be customized enough to replace those environments while providing the same features without a graphical user environment being required.

6.3 Operating System or platform to use

When launching a new project, it may be needed to establish decisions about what operating system to use to achieve the particular goal at hand. The game industry and the high audience targets will generally favor the most widely used operating system in use by the user community like win32 systems.

However, embedded design will often use cross-compiling from an operating system to another smaller one for use in the embedded product. This may also require the need for simulators which can be ran on the faster and larger developping platform to save time, not requireing one to always upload the new code versions to the embedded platform for testing.

For large scale networking servers, systems very stable systems with strong TCP/IP stacks may be required. Unix-like systems such as BSD and Linux are widely used in this area because although free they provide impressive stability and performance. They also are great development platforms because a wide range of free utilities exist which are easy to install on them for a variety of languages and applications. Moreover, being very stable, they allow to centralize the sources (and often development). It is not rare to have all programmers of an organization logged in on a shell, each with their own user, on the same unix

server and all work at the same time without interfering. Moreover, tools like *ssh(1)*, *wall(1)*, *write(1)* and *talk(1)* and *cvs(1)* are inherently part of the system to break remote communication barriers. Where terminals are not considered an ideal medium, remote X11 sessions can even be used because of the networking nature of the X11R6 protocol.

Using free operating systems however often implies that the organization has a technical team (or at least a sysadmin) which can manage the system(s). Where this is at fault, it often becomes simpler to benefit from commercial operating systems for whcih technical support from third party corporations can easily be obtained, such as Wasabi version of NetBSD, Apple's Mac OS X, Sun Microsystems Solaris, RedHat Linux Enterprise Edition, Lindows or even Microsoft Windows, in the case where production servers are not required, and if the projects are home-user centric such as some games if their portability isn't a concern.

In cases where high portability accross platforms is needed, the team will need to have access to a variety of operating systems to develop the software on. This often has to be planned in advance. It is often necessary to also rely on already existing third party libraries which exist on many systems (for instance the BSD IPC/Socket API, SDL, OpenGL, GTK, etc).

If the goal is high portability among hardware architectures, NetBSD is definitely a solution to look into, especially as there also exist corporations distributing and supporting commercial implementations of NetBSD, such as Wasabi Systems (*http://www.wasabisystems.com*).

Even the free NetBSD at *http://www.netbsd.org* is the most portable operating system known among architectures and has great documentation (some people dispute this saying that Linux wears the hat nowadays, but no known Linux distribution is known to support as much platforms as NetBSD, considering that Linux is only a bare kernel on its own. Moreover, there is a distinction between portable code (which only requires a minimal effort to port such as NetBSD) and brute-force ported code (which is eventually ported because of the huge efforts of many people).

NetBSD is a very solid environment to work on, derived from 4.4BSD, and includes a GCC toolchain with awesome cross-compiling scripts to ease the work. Developers that need access to several architectures which they cannot obtain can use cross-compiling, and there even exists a section on the site for ones which need to obtain a shell provided by another NetBSD user running the wanted architecture, so that one may compile and test the software without actually owning a particular architecture. Its source is available and is among the cleanest ones around. And, the GPL tainted parts are isolated into a separate *gnu* directory, so that the rest of the Operating System is unobstructed by the GPL limitations.

6.4 CVS

CVS is provided as part of several operating systems, such as BSDs. It also is available as easy to install packages on most other unix-like systems including

Linux distributions. There also exists a win32 port. Its source can be obtained at: XXX

CVS stands for (Concurrent Versions System). Evolved from RCS, which was made to manage revisions of files, it adds support for directories and trees along with many new features. It allows a team of programmers to work at the same time on a project, and for the main source repository to be located on a single server. It is possible with CVS to allow read-only access to the public (if public CVS access is wanted), and read-write access to wanted users provided with shell access to the server. Even for server local users, unix filesystem permissions can be set to only allow access to wanted users, since CVS only uses files and directories for storage.

With CVS, it is possible to branch a project into several concurrent branches, to merge changes between branches, to go back in time and obtain the sources from a particular epoch and request diff files between arbitrary dates. And because it is a widely used system, many people wrote CVS-related and compatible utilities. Many organizations rely on CVS for project development, maintenance and release engineering. Some unix sysadmins even use it to manage system configuration files.

In the case of open source software, it is extremely easy for users which have read-only access to the repository to efficiently upgrade their sources to the latest revisions and recompile, or to request diffs from their own version relatively to the new one at the repository for them to see which changes were made and if an upgrade should be performed. This is also true for closed source software where particular parties have source access.

CVS can in fact be used for any kinds of text files collection, like documentation and sources. It however also supports binary files, although not all its features can be used with them.

Although not strictly useful for Open Source projects, a great book is recommended to any CVS user, "Open Source Development with CVS, 2nd Edition", written by Karl Fogel and Moshe Bar, distributed by Coriolis Technology Press. A free edition is available in Portable Document Format (PDF) at: XXX

The command documentation can also be found as a texinfo manual, using the *info cvs* command once *cvs(1)* is installed.

For organizations who need public read-only repository access there exists software which can help secure CVS servers alot. An implementation consists of *mmspawnd(8)* and *mmanoncvs(8)* from Matthew Mondor, which can be obtained and installed via CVS, using the command:

XXX

6.5 Assertion and debugging macros

- assert(3), example macros as _DIAGASSERT(3), etc.

XXX

6.6 Modularity

6.6.1 Project tree layout and build scripts

6.6.2 const, static and external

6.6.3 Libraries

6.6.4 Configuration files

6.7 Linked lists, arrays and dynamic allocation

6.7.1 Linked lists

6.7.2 Arrays

6.7.3 Dynamic memory allocation and release

6.7.4 minheap implementations

6.8 Hash tables and Trees

6.8.1 Hash tables

6.8.2 Trees in general

6.9 Finite state machines

6.10 Configuration files parsing

6.11 Useful libraries and APIs

There are many more available ready-made libraries which one can use when programming in C. The need to use one or another will depend on a variety of factors such as project size, the dependencies it may accept or not to require, as well as licensing issues (I.E. avoiding to taint a commercial or BSD licensed product with GNU General Public Licenced (GPL) components, etc).

In this section we only point out some of the popular, very widely used libraries in the open source world which the author could remember at the time of this writing. Good sources for additional libraries consist of the available packages list for your particular UNIX(-like) system distribution, a list of the */lib*, */usr/lib*, */usr/local/lib*, */usr/pkg/lib* directories on your filesystem, as well as the popular *http://www.sourceforge.net* and *http://www.freshmeat.net* sites.

6.11.1 The ANSI C library

6.11.2 mmlib (Matthew Mondor's general purpose library)

6.11.3 POSIX and X/Open

There are IEEE defined standards for POSIX (Portable Operating System Interface for uniX XXX). These are highly implemented and deployed over a wide variety of operating systems today. Most of the functions within this API were

derived from commonly found ones derived from BSD or commercial UNIX systems. There even exists some POSIX compatibility libraries alternatives for Win32, such as the ones included with *cygwin* and *wingw*.

On most systems implementing these, the functions are available from the native C library (libc). The reference documentation for those are then also generally implemented in the form of BSD mdoc or UNIX mman nroff manual pages. However, the definite standard can be located at: XXX

X/Open interfaces, similarily to POSIX have been accepted by a working group to form a standard to use on a wide range of operating systems. The definite X/Open Portability Guide documentation can be found at: XXX

6.11.4 PThreads (POSIX Threads API) vs fork(2)

Threads consist of an attempt to optimize programs which require inherent shared memory across multiple parallel concurrent executing tasks. As opposed to full weight unix processes, threads can, depending in the implementation, be composed of either full fledged processes, light weight processes, or even of a user space library implementing multiple contexts of execution with their stacks within a single process. In fact, a known problem is indeed related to the fact that many programmers who used the PThreads API on a system assumed too much about that platform-specific implementation, causing the same code to not work the same on other platforms also supposedly implementing the same POSIX API. Special care has to be taken when using PThreads to avoid this, and reading the POSIX official documentation is recommended to achieve portability using it, or implementing it for a particular system.

Traditionally, the *fork(2)* system call is used to create a parallel process when needed, creating another process which can execute concurrently. However, this also implies that special shared memory must first be setup if both processes need to access shared resources, and that synchronization locks often need to be used as well to protect these shared resources. The *fork(2)* system call uses the concept of inherently independent address spaces, with a minimum of shared resources among the processes, where the original process has it's allocated memory and resources duplicated in the new child process, rather than inherited in shared mode.

The thread paradigm, on the contrary, is based on the concept that all resources should be shared by all parallel tasks, including file descriptors and allocated memory. For this reason, under some circumstances it becomes cleaner to use parallel threads rather than processes. However, this does not exclude that synchronization locks (commonly rwlocks and mutexes) need to be used to access the shared resources if more than one process may compete to access them, causing race conditions. Technically speaking in POSIX terms, all threads started from a process are still considered part of that same process.

Tricky parts which have to especially be noted about the PThread API is that signaling a threaded process implies signalling the whole process (thus all threads), unless a thread specifically blocked the unwanted signal. Another thing is that system calls may or may not block the whole process (thus all

threads) until their completion, and popular standard C library functions may or may not be thread-safe (non-reentrant functions using static memory storage are especially problematic), causing race conditions and requireing explicit thread-friendly locking like for shared resources. Also, a programmer should not assume that each thread corresponds to an actuall process (which is the case in so called 1:1 implementations, such as the one used with glibc and Linux). Other implementations will use virtual processes (such as 1:n ones), Scheduler Activations and other specialized efficiency based techniques rather than actual processes (such as the threads in NetBSD-current, using SA with the concept of lightweight processes and virtual processors).

Another tricky aspect has to do with thread preemption, which is provided by some POSIX libraries and not by others. In a preemptive system, a thread spinning in an endless loop taking alot of CPU time will automatically be scheduled out to permit other concurrent threads from running as well. On non-preemptive implementations (such as in the case of GNU Pth's PThread wrapper library, unproven-threads and unreal-threads), such a thread will not allow any other thread to execute unless it explicitely yields control to another thread, either via a special implementation-specific function, or by calling a blocking system call or PThread function, allowing other threads to be scheduled some CPU time.

Because the data of the process is shared between all threads, they also should theoretically launch and exit, as well as perform context switches quite faster than when using multiple processes. The cost of traditionnal *fork(2)* was rather expensive, especially before implementations using Copy On Write (COW) pages, and context switches between processes require more kernel work, like switching the whole VM space tables. This efficiency increase factor using threads however greatly depends on the internals of one particular implementation.

On the other hand, it is not uncommon to use a prepared pool of *fork(2)* generated processes to save the cost of process creation, and to manage that pool as to need to stop and restart the processes the less often possible, while dynamically growing or shrinking the pool depending on the current load. An obvious advantage of real processes over threads is that each process consists of a totally independent kernel managed entity, within its own VM space. This can provide greater security in the case of exploits targetted at servers, as well as prevent the whole server from shutting down in the case of a process crashing. Moreover, the lack of reentrency of libc functions, or of preemtiveness, do not become problems anymore. A system with SMP (Synchroneous Multiprocessor Support) also always benefits from multiple processes, while thread implementations may or may not be able to take advantage of multiple processors. There actually are less possible variants when using traditionnal *fork(2)* than when using PThreads as far as wide platform portability is concerned. Privilege separations techniques are also implemented using multiple processes rather than threads, because of the fact that it requires multiple processes running under different user permissions.

Some other types of servers can also handle an awesome number of simultaneous clients in a single, unthreaded process. A great example of this consists

of the most widely used IRC servers. With a cleanly designed state machine, and using I/O features such as *select(2)*, *poll(2)*, signals or BSD *kqueue*, this becomes possible. For some applications this results in better performance, since no synchronization locks are necessary to use around shared resources, and there is no context launch/exit/switch cost. In fact, GNU Pth threads are scheduled by a *select(2)* based loop in a single process. This approach however cannot take advantage of SMP unless multiple such copies are running.

That said, the choice over using on-the-fly process creation with *fork(2)*, a pool of managed processes, threads, or a single non-threaded loop with a state machine, depends on a variety of factors: efficiency, security, portability, expected load and implementation decisions. Under many circumstances, clean design will even have the last word on which to choose for a particular problem at hand.

The offical POSIX documentation for the PThreads API can be found at: XXX

The Pth user-space non-preemptive PThreads library can be found at: XXX

6.11.5 The BSD sockets API (BSD Inter-Process Communication)

This API is provided by most networking capable operating systems (including Win32 through winsock.dll). This API originated from 4.4BSD which implemented the first widely available TCP/IP stack, from which derived alot of other networking commercial (including Win32, QNX, Mac OS X, Mach) and open sourced operating systems. Those systems which did not base themselves on any 4.4BSD code base usually still implement this API for compatibility purposes because of it's high popularity (this includes Linux and AmigaOS).

Although this does not consist of a BSD sockets API reference or tutorial, it is interesting to know that on most systems these are included as part of their native C library (libc). The most popular functions of this API consist of *socket(2)*, *bind(2)*, *listen(2)* and *connect(2)*. This API allows abstraction of a variety of network protocols, including IPv4 and IPv6.

A good tutorial on the BSD sockets API can be found at: XXX Advanced BSD Inter Process Communication

6.11.6 4.4BSD shared memory and 4.2BSD advisory locks

This system first was implemented on 4.4BSD and is still available in all modern BSD derived systems. It however also is available Linux which developped it's own implementation following the same API for compatibility.

XXX *mmap(2)* with *MAP_ANON* and *flock(2)*

It is important to know that the files which are used for advisory locking using *flock(2)* must be reopened by all children processes which need to access the shared resources. Otherwise, the processes will not compete for the access, but will be acquireing their own independent locks with success at the same time.

For systems where the 4.4BSD *MAP_ANON* feature is not available but where 4.2BSD *mmap(2)* is, it is always possible to use a temporary file on the filesystem with *truncate(2)* to use as shared memory. However, this may also require the use of *msync(2)*. Other systems will allow the *"/dev/zero"* device to be mapped into memory with *mmap(2)* to use instead of a temporary file, which ressembles *MAP_ANON*. Moreover, an alternative to 4.2BSD *flock(2)* to synchronize shared resources access among processes consists of *fcntl(3)* which is now POSIX standard. With *fcntl(2)* it is not necessary to reopen the lock files in each process.

6.11.7 System 5 shared memory, IPC and semaphores (SysV SHM, SEM and IPC)

This system is probably the most portable of IPC APIs, as all unix-like and commercial UNIX systems seem to support it. It also has special features which are useful in the case where programs that frequently must be restarted need to reattach to the same shared resources it previously was using the last time it ran. Those same resource persistency features are often considered as a design bug by many programmers as well :).

XXX

6.11.8 libmm (Apache's shared memory abstraction library)

XXX

6.11.9 Berkeley db3/db4 low level database library

Berkely DB3/4 provides access to hash or tree based databases, on a key-data basis. It allows to handle in-memory only databases as well as safe disk storage ones with recovery support using internal logs. It also optionally supports high concurrency transactions with commit/rollback capability.

It is low-level in the sense that it requires a little more work on the part of the C programmer to implement complete applications around this library than it is to use a simple SQL server frontend library such as *libmysqlclient*. However, it also leaves alot more control to the programmer on implementation details. There exists APIs to access this library from multiple languages, although the native interface is made for C. These APIs are well documented by HTML pages.

This library is open source and BSD licensed, but can also be purchased with a commercial license and support for the applications programmers. It is known to have been used in wide deployment critical systems such as routers, and to also be used by some LDAP servers to handle internal storage. Moreover, several SQL servers internally use Berkeley db3, including MySQL if compiled with transactions. It can be obtained at *http://www.sleepycat.com*.

6.11.10 OpenSSL for secure encrypted SSL/TLS channels

6.11.11 libmysqlclient SQL level database library

6.11.12 GSL (GNU Scientific Library)

6.11.13 glib (usually as part of gtk)

6.11.14 hlib (XXX check with skapare from freenode #C)

6.11.15 curses, ncurses

6.11.16 xlib, GTK, SDL, OpenGL/GLut

xlib consists of the library for GUI clients to access X11R6 servers. X11R6 supports networking and as such, it is possible to export a display on a wanted X11R6 graphics terminal. Most open source unix-like operating systems ship with XFree86, an implementation of the X11R6 protocol. This includes the BSDs and Linux. xlib can be used to access the X11R6 functionality directly, as opposed to requireing a fancy asbtraction toolkit which can be much heavier. Those open source implementations use an open source, BSD-style license called the XFree license. However, there also exist commercial implementations, following the same API. XXX xfree86 location, docs

GTK (Gimp Tool Kit) version 1.2 consists of a light yet powerful and themeable toolkit, usually implemented on top of xlib but which also has a port to win32. It originally was implemented as part of the Gimp graphics editor. GTK2, a late reimplementation, is much heavier but also supports much fancier theme and font support, as well as printing support. It is released under the GNU Lesser General Public License (LGPL) as open source. This library provides a nice interface to C programmers to implement Graphical User Interfaces (GUIs). Many applications use it. It also internally is used by GNOME. XXX location, docs

SDL is an abstraction library which can be used on a variety of platforms and architectures to provide graphics capabilities to C programs. This does not consist of a toolkit library, but rather of a general purpose 2D graphics library, which is also widely used in the implementation of games. This library is released under the LGPL as open source and can be found at: XXX location, docs

OpenGL consists of a standard C library API for 3D graphics. It originally was implemented on IRIX unix systems by Silicon Graphics (XXX confirm), but other registered and non-registered OpenGL commercial and open source libraries exist. Moreover, some graphics hardware vendors started to release OpenGL drivers to directly access the hardware capabilities of the cards (such as NVIDIA). For instance, on Linux people can use the proprietary binary driver provided by NVIDIA for enhanced OpenGL speed, or can be using MesaGL, an open source unregistered implementation library released under the terms of the LGPL emulating all OpenGL functions by software alone, for slower but very compatible results. GLut consists of a toolkit library built over OpenGL and

generally provided with OpenGL implementations. The official OpenGL documentation ca be located at: XXX. The MesaGL library can be found at: XXX. This OpenGL interface is internally used by higher level abstraction interfaces such as the C++ OGRE 3D engine where available. Many recent games, 3D modeling, realtime rendering and simulation applications are also written for OpenGL nowadays. Unlike DirectX, OpenGL is available everywhere.

6.11.17 XDR (provided with sunrpc implementations)

XXX

6.11.18 Expat library for XML support

XXX

6.11.19 SpiderMonkey (Netscape/Mozilla JavaScript engine)

XXX

6.11.20 Python

Python is increasingly used in open source applications to embed scripting functionality into C programs.

XXX